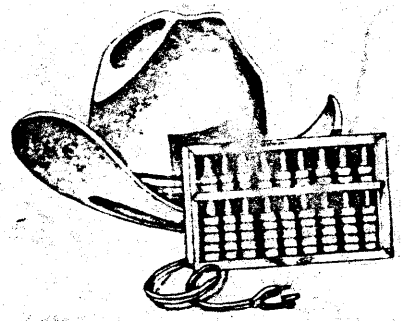
BD Software

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



Western Abacus Co., Ltd. 312/498-6660

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 xK bytes for the symbol table. If an "Out of symbol table space" error occurs, this option may be used to increase the amount of space allocated for the symbol table. Alternatively, if you draw an "Out of memory" error then **-r** may be used to decrease the symbol table size and provide more room for source text. A better recourse after running out of memory 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 *execing* and *execed* 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 specified 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.
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

he files making up the CASM package are as follows:

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

lso needed:

SM.COM (or MAC.COM)
DT.COM (or SID.COM)

escription:

The only means previously provided to BDS C users for creating relocatable object odules (CRL files) from assembly language programs was a painfully complex macro ackage (CMAC.LIB) that only operated in conjunction with Digital Research's macro ssembler (MAC.COM). This was especially bad because MAC, if not already owned, cost lmost as much as BDS C to purchase. This document describes the program "CASM", upplied to eliminate the need for "MAC". CASM is a preprocessor that takes, as input, n assembly language source file of type ".CSM" (mnemonic for C aSseMbly language) in format much closer to "vanilla" assembly language than the bizarre craziness of MAC.LIB, and writes out an ".ASM" file which may then be assembled by the standard, biquitous CP/M assembler (ASM.COM). CASM automatically recognizes which assembly anguage instructions require relocation parameters and inserts the appropriate seudo-operations and extra opcodes into the resulting ".ASM" file so that the file roperly assembles directly into CRL format. In addition, some rudimentary logic hecks are performed: doubly-defined and/or undefined labels are detected and eported, and similarly-named labels in different functions are ALLOWED and converted nto unique names so ASM won't complain.

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

UNCTION <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,argl, 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	
	rand()	D2	33	
igned	rcfsiz(fd)	D2A		
	read(fd, buffer, nsecs) char *buffer;	D2	47	
	rename(oldname, newname) char *oldname, *newname;	D2	46	
	rread(fd, buffer, nsecs) char *buffer;	D2A		15
	rseek(fd, offset, origin)	D2A		15
	rsrec(fd)	D2A		15
	rsvstk(n)	D2	38	
	rtell(fd)	D2A		15
	rwrite(fd, buffer, nsecs) char *buffer;	D2A		15
ır *	sbrk(nbytes)	D2	37	
	scanf(format, &argl, &arg2, ...) char *format;	C2	42	4,10
	seek(fd, offset, origin)	D2		
	setfcb(fcbaddr, filename) char *filename;	D2	53	
t	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,argl,arg2,...) char *str, *format;	C2	42	4,9
	srand(n)	D2	32	
	srandl(str) char *str;	D2	32	
t	sscanf(str,format,&argl,&arg2,...) char *str, *format;	C2	42	10
	strcat(s1, s2) char *s1, *s2;	C1	43	
t	strcmp(s1, s2) char *s1, *s2;	C1	43	
	strcpy(s1, s2) char *s1, *s2;	C1	43	
:t	strlen(str) char *str;	C1	44	
	swapin(filename,addr) char *filename; unsigned addr;	C2	36	
ıt	tell(fd)	D2	48	
ıar	tolower(c) char c;	C1	42.	
ıar *	topofmem()	D2	36	19
ıar	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	
ıt	write(fd, buffer, nsects) char *buffer;	D2	47	

)TES:

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

- 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.

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.

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.)

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

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.

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.

- 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.

- 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:

- 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();
```

- 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.

- 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

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

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

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.

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.

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
```

here `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.

•

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 */
```

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

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

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

C1 has not been recognizing illegal octal character constants as such; digits such s `8' and `9' within an octal constant will now draw an error in cases where they ould have been ignored before. Also, certain other forms of illegal constants (aside rom character constants) are now better diagnosed than before.

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

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

lso, CLINK will now search ALL default library files (DEFF.CRL, DEFF2.CRL and EFF3.CRL [if it exists]) when a carriage-return is typed in interactive mode. reviously, only the file DEFF.CRL was searched in response to carriage-return.

t has been brought to my attention that the ^Q-CR sequence required by CLINK in nteractive mode (to abort the linkage in progress) cannot be typed in under MP/M ystems, since ^Q is used to detach a process. If you are running MP/M, then just type ontrol-C instead of ^Q-CR; this will also work for CP/M systems...the only difference s that when ^Q-CR is used, then any currently active "submit file" processing is utomatically aborted by CLINK before returning to command level, as a convenience (I ssume 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.

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.

0. 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.

1. 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.

2. 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.

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.

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.)

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.

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.

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.

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.

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.)

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.

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].

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.

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.

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 cumentation have said that my new city and zip code were Allston, 02134, which is where THOUGHT I was. Actually, I'm in Brighton, 02135, and any mail sent me addressed to lston may have been returned to the sender stamped with something like "No such address own." Sorry about that.

re are the bug fixes/extensions for version 1.44:

(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.

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".

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

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.

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.

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.

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.

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()
{
        ...
}
```

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);
```

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!

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.)

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

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

BDS C User's Guide Addenda
vl.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 orrible bugs that have plagued previous versions of the compiler), I'd like to take a oment to answer one of the more common questions that have been asked of me by users and otential users of BDS C. Hopefully, this will save some of you the expense of a phone all (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 ibraries, in either source or object form (or both), may be freely distributed with ommercial (or non-commercial) application programs. The reason for this policy is to romote the use of C for anything and everything, without wrapping up potential pplications in miles of red tape and ineffective security measures. Software authors: LEASE include the source listings to your software with your packages! I understand that here are some markets where such generosity is considered suicidal, and I sympathize in any cases, but I also want to see BDS C selling more copies, and providing the source to pplications programs will encourage users to obtain the compiler. Hopefully, some of them ay even BUY it.

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

. 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.

.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.

. 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...)

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

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.

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...)

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.

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++;
```

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

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...)

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.

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
```

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);
```

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
```

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.

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.

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.)

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...

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                        *
*                                                            *
**************************************************************
```

order to allow C programs to be longer than physical memory, without resorting to "exec" or cl" (which may indeed get the job done, but resemble "chain" operations more than true entation tools), a new set of capabilities has been built into the CLINK program. ally, 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)
-------------------------------------------------------------------
```

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

mplement overlays, the first thing necessary is to decide just where the swapped-in code is reside. Earlier versions of BDS C had local data frames growing up from low memory, ting from where the externals ended, making it difficult to determine the lowest memory tion safe to swap into. The scheme suggested then for handling overlays was to leave icient room between the end of the root segment code (the root segment contains the "main" tion and run-time package; it loads at the start of the TPA, always remains in memory, and rols the top level of overlay swapping) and start of the external data area to accommodate largest possible swapped-in segment combination. This is still a viable scheme for version 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)
-------------------------------------------------------------------
```

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

version 1.4, it is just as feasible to put the overlay area AFTER the externals. Ty
:y 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)
-------------------------------------------------------------------
```

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

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

h the generalities out of the way, let me say something about just how to create "root" s and "swappable" segments with BDS C. First of all, we would like all functions defined the root segment to be accessible by the swapped segment(s)...this is accomplished by CLINK to write out a symbol table file (containing all function addresses) to disk when t segment is linked. The -w option to CLINK will do the trick; this symbol table will be ter when linking the swappable segments.

inking the root segment, use the -e option to set the external data area location; keep that there must be enough room below the externals to hold the largest swapped-in at run time (I'm using the term "below" in the sense that low memory is "below" high graphically, in the preceding memory maps, "below" means toward the top of the page.) -e option is omitted, CLINK will assume the external data starts immediately after the the root segment code; this is OK only if you're using the SECOND scheme.

the code of the root segment, then, a swappable segment is loaded into memory from disk ng:

```
swapin(name,addr);    /* read in a segment..don't run it */
```

"addr" is the location following the last byte of root segment code (for the first .) You can find this value by linking the root once without giving the -e option and the -s statistics written to the console after the linkage. To actually execute the , you have to call it indirectly using a pointer-to-function variable.

an example. We'll declare a pointer-to-function variable called "ptrfn", swap in a 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 */
...
```

vapin" routine returns -1 when a load error occurs. Note that the swapped-in code might turn any value, but the pointer-to-function must be declared with SOME kind of type. Use if nothing else comes to mind. When a segment is invoked, as above, control passes to the 's "main" function. There is no reason at all to require args to be of the "argc" and form; there is nothing special about a "main" function other than the property it has of called first. The "main" function within the swapped-in segment is the ONLY allowed point for the segment.

le "swapin" function is given in STDLIB2.C. It can be made shorter by skipping all the testing, or can be expanded to detect an attempted load over the external data area by ing the last address loaded with the contents of location ram+115h...if you've never done w-level hackery, you get the value of the 16-bit address at location ram+115h by using ction on a pointer-to-integer (or -unsigned.) Note that location RAM+115h ALWAYS contains dress of the base of the external data area.

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

swappable segment is basically just a normal C program, having a "main" function just
e root segment, except that the C.CCC run-time utility package is NOT tacked on to the
f a swappable segment (the C.CCC in the root segment will be shared by everyone.) T
ifference between a swappable segment and the root segment is the load address; whi
t segment always loads at ram+100h (where "ram" is 0 for standard CP/M, or 4200h for the
ed" CP/M), a swappable segment may be made to load anywhere. Once you've compiled the
le segment, you give a special form of the CLINK command to link it:

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

segmentname" is the name of the CRL file containing the segment, "-v" indicates to CLINK
. swappable segment is to be created (so that C.CCC is not attached), and "-l xxxx"
ell followed by a hex address) indicates the load address for the segment.

ou'll probably want to yank in the symbol file created by the linkage of the root
, use the -y option to do so. If you don't, then CLINK will yank in fresh copies of
ns like "PRINTF" and "FOPEN", etc., even if they have already been linked into the root
. It would be a waste to have multiple copies of those memory hogs in there at the same
y reading in the symbol table from the root segment, it is insured that any routines
linked in the root will be made available to the swapped-in segment. The root segment,
cannot know about functions belonging to the swapped-in segment through the use of a
table. That would require some kind of mutually referential linking system beyond the
f this package.

. When linking the segment, you may specify -s to generate a stat map on the console,
to write out an augmented symbol table containing not only the symbols read in from the
gment's symbol file, but also the swappable segment's own symbols. This new symbol file
n be used on another level of swapping, should that be desired.

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

ay you've got a program ROOT.C, which will swap in and execute SEG1.C and then overlay
with SEG2.C. ROOT.COM loads at 100h and ends, say, before 3000h. We'll load in the
s at 3000h, and set the base of the external data area to 5000h (this assumes neither
is longer than 2000h.)

kage of ROOT would be:

```
A>clink root -e 5000 -w -s <cr>
```

lls CLINK that ROOT.COM is to be a root segment (no "-v" option used), the externals
t 5000h, a symbol file called ROOT.SYM is to be written, and a statistics summary is to
ted to the console.

kage of each segment would appear as:

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

mand line tells CLINK that SEG1.COM is to be a swappable segment (the "-v" option) to
t location 3000h, the symbol file named ROOT.SYM should be scanned for pre-defined
n addresses, a statistics summary should be printed after the linkage, and the object
to be written out as SEG1 (as opposed to SEG1.COM, to avoid accidentally invoking it as
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)
```

ld result in a value having the type "char" and could never possibly equal EOF as ted for in the program. Should "getc" ever return EOF in such a case, `c' would end being equal to 255 (the "char" interpretation of the low order 8 bits of the value ). Thus, `c' is declared as an "int" so that the EOF comparison can make sense. s is awkward because `c' is used here for holding characters, and it would be nice have it declared as a character variable. There's actually a way to do it, at the ce of complete generality: if the EOF in the comparison were changed to 255, then would have to be be declared as a "char", and the program would work...EXCEPT for n an actual hex FF (decimal 255) byte is encountered in the input file! Now, while is a pretty safe bet to assume there aren't any hex FF bytes in your average text e, there may be exceptions. Also, there's no law that says filters can only be tten for text files. Consider a program to take a binary file and "unload" it, ating an Intel-format HEX file. Would we want it to halt when the first hex FF is ountered? No, the original method is clearly the most general.

e having determined that the end-of-file has not been encountered, the body of the ile" statement is executed. Here we use "toupper" to convert the character obtained m "getc" to upper case, and then we use "putc" to write the resulting byte out to output file. To be neat, errors are checked for: the program terminates if "putc" urns ERROR.

soon as an end-of-file condition is detected, we write out a final CPMEOF ntrol-Z) character to terminate the output file. The way this particular program is up, the CPMEOF will be appended to the output file whether or not the input file led with a CPMEOF. Next, "fflush" is called to flush the output buffer. This must ays be done before closing a buffered output file, to make sure that all characters it to "putc" since that last time the buffer filled up get written to disk. Finally, lose" is used to close the input and output files.

more examples of the usage of buffered I/O, see CONVERT.C, CCOT.C, TABIFY.C and NET.C. Also, take some time to inspect the files BDSCIO.H, STDLIB1.C and STDLIB2.C, ch 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 anisms available to BDS C users. When the major documentation for BDS C (i.e. the User's e) was being prepared, I had mistakenly assumed that users would automatically realize how "bdos" and "bios" library functions could be used to perform all CP/M and BIOS functions, cially direct console I/O (by which the system console device may be operated without the trating unsolicited interception of certain special characters by the operating system.) In , the use of the "bios" function for such purposes might only be obvious to experienced users, and then only to those having assembly language programming experience with the y-gritty characteristics of the CP/M console interface. Let's take a look at what really ens during console I/O...

The lowest (simplest) level of console-controlling software is in the BIOS (Basic t/Output System) section of CP/M. There are three subroutines in the BIOS that deal with ing and writing raw characters to the console; they are named `CONST' (check console us), `CONIN' (wait for and read a character FROM the console), and `CONOUT' (send a acter TO the console). The way to get at these subroutines when you're writing on the mbly language level is rather convoluted, but the BDS C library provides the `bios' tion to make it easy to access the BIOS subroutines from C programs. To check the console us directly, you use the subexpression `bios(2)', which returns a non-zero value when a ole character is available, or zero otherwise. To actually get the character after s(2)' indicates one is ready, or to wait until a character is ready and then get it, use s(3)'. To directly write a character `c' to the console, you'd say `bios(4,c)', but note the BIOS doesn't know anything about C's convention of using a single '\n' (newline) acter to represent a logical carriage-return/linefeed combination. The call `bios(4,'\n')' 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 outines is the ONLY way to both keep CP/M from intercepting some of your typing and insure portability of programs between different CP/M systems. (1)

The BDOS (Basic Disk Operating System) operations are the next higher level (above the) on which console I/O may be performed. Whenever the standard C library functions :har' and `putchar' are called, they perform their tasks in terms of BDOS calls...which in perform THEIR operations through BIOS calls, and this is where most of the confusion es. Just as there are the three basic BIOS subroutines for interfacing with the console, e are three similar but "higher level" BDOS operations for performing essentially the same s. These BDOS functions, each of which has its own code number distinct from its BIOS :erpart, are: "Console Input" to get a single character from the console (BDOS function 'Console Output" to write a single character to the console (BDOS function 2), and "Get

---

Even so there's no way to know what kind of terminal is being used--so "truly portable" ware either makes some assumptions about the kind of display terminal being used (whether ot it is cursor addressable, HOW to address the cursor, etc.) or includes provisions for -modification to fit whatever type of terminal the end-user happens to have connected to system.

le Status" to determine if there is a character available from the console input (BDOS
ion 11). The BDOS operations do all kinds of things for you that you may not even be fu
of. For instance, if the BDOS detects a control-S character present on the console input
g a console output call, then it will sit there and wait for another character to be typed
e console, and gobble it up, before returning from the original console output call. This
e fine if you want to be able to stop and start a long printout without having to code
feature into your C program, but it causes big trouble if you need to see EVERY character
on the console, including control-S. A little bit of thought as to how the BDOS does what
oes reveals some interesting facts: since it must be able to detect control-S on the
le input, the BDOS must read the console whenever it sees that a character has been typed.
e character ends up not being a control-S (or some other special character that might
re instant processing), then that character must be saved somewhere internally to the BDOS
at the next call to `Console Input' returns it as if nothing happened. Also, the BDOS must
sure that any subsequent calls made by the user to `Get Console Status' (before any are
to `Console Input') indicate that a character is available. This leads to a condition in
a BDOS call might say that a character is available, but the corresponding BIOS call
NOT, since, physically, the character has already been gobbled up by the BDOS during a
interaction with the BIOS.

If this all sounds confusing, bear in mind that it took me several long months of playing
CP/M and early versions of the compiler before even I understood what the hell was going
there. My versions of `getchar' and `putchar' are designed for use in an environment
the user does NOT need total direct control over the console; given that the BDOS would
ome nice things for us like control-S processing, I figured that I might as well throw in
more useful features such as automatic conversion of the '\n' character to a CR-LF
nation on output, automatic abortion of the program whenever control-C is detected on
or output (so that programs having long or infinite unwanted printouts may be stopp
ıt resetting the machine, even when no console input operations are performed), automat
rsion of the carraige-return character to a '\n' on input, etc. One early user remarked
ıe would like `putchar' to be immune from control-C; for him I added the `putch' library
ion, which works just like `putchar' except that control-C's would no longer stop the
ım. Much later it became evident that neither `putchar' nor `putch' suffice when CP/M must
revented from ever even sampling the physical console input. At this point I added the
' function, so that users could do their I/O directly through the BIOS and totally bypass
rustrating character-eating BDOS.

I promised some examples earlier, so let's get to it. First of all, here is a very
ntary set of functions to perform the three basic console operations in terms of the
' function, with no special conversions or interceptions AT ALL (i.e., nothing like the
--> 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 utines. If you include these in your C source program, then the linker will use them ad of the standard library versions of the similarly named functions--provided that some t reference to them is made before the default library file (DEFF2.CRL) is scanned. ly, in programs where such functions are necessary, there will be many explicit calls to har' and `putchar' to insure that the library versions aren't accidentally linked. A good le of a case where trouble might occur is when the entire program consists of, say, a e `printf' call followed by a custom version of `putchar'. Since the linker won't know `putchar' is needed until after `printf' is loaded from the library, the custom version of har' will be ignored and the old (wrong) version will be picked up from the DEFF2.CRL ry file. The way to avoid such a problem is to insert, somewhere in the source file, cit calls to any functions that are a) NOT explicitly called otherwise, and b) named the as some library function. This isn't an expecially neat solution, but it gets the job

OK, with that out of the way, let's consider some more sophisticated games that can be d with customized versions of the console I/O functions. For starters, how about a set performs conversions just like the library versions, detects control-C, and throws away haracters typed during output (except control-C, which causes a reboot)? No problem. s needed is automatic conversion of '\n' to CR-LF on output; conversion CR to '\n' and ^Z l on input with automatic echoing; and re-booting on control-C during both input and t.

```
/*
        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 */
```

ow, if you wanted to have control-S processing and a push-back feature (the two are ly quite related, since you must be able to push back anything except control-S that be detected during output), you could add some external "state" to the latest set of ons and keep track of what you see at the console input. Once this is done, though, probably better off going back to the original library versions of `getchar' and ıar', which let the BDOS handle all that grungy stuff.

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

o far, everything I've talked about has been in terms of the BIOS, and applies equally to P/M systems. Unfortunately, there is one console operation often needed when writing time interactive operations that is not supported by the BIOS, and thus there is no ole way to implement it under CP/M. What's missing is a way to ask the BIOS if the console nal is ready to ACCEPT a character for output. An example of the trouble this omission s is evident in the sample program RALLY.C; the case there is that the program must be to read input from the keyboard at any instant, and cannot afford to become tied ng for the terminal when the amount of data being sent to it has caused the X-ON/X-O col to lock up the program until a character can be sent. Given that the only "kosher" wa

ɜend a character to the console is through the CONOUT BIOS call, and that such a call might ɪny time tie up the program for longer than is tolerable, the only recourse is to bypass)UT completely and construct a customized output routine in C that can be more ɪisticated. This is done in RALLY.C, at the expense of non-portability for the object code; ɪ user must individually configure his BDSCIO.H header file to define the unique port ɜers, bit positions and polarities of the I/O hardware controlling his console. It would ɜ been SO much easier if the BIOS contained just one more itty bitty subroutine to test ole output status...but NoooooOOOOOOOoooooo, they had to leave that one OUT so we have to GE it...

Sorry. I get carried away sometimes. Oh well...I hope this has helped to demystify some of obscure behavior sometimes evident during console I/O operations. For the low-down on how library versions of `getchar', `putchar', etc. really work, see their source listings in 2.ASM. And if there's something you want to do with the console and can't figure out how ite this document, I'm always available for consultation (at least whenever I'm near the e.)

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 arm-boot when finished with their tasks. This is because the stack is usually placed in igh memory just below the BDOS, wiping out part of the CCP (console command processor) uring execution and requiring a warm-boot to bring back the CCP from the system tracks on isk. The following patches to the C.CCC run-time package file provide a way to generate OM files that do NOT perform a warm boot after execution, but instead return directly to a on-clobbered CCP. The price of avoiding a warm-boot is that there is less memory space vailable during execution (3000 bytes less by default); the advantage is that there is no aiting for the disk to seek and load the CCP every time the program is finished, improving verall performance and preserving the nerves of impatient hackers.

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

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

- 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.

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

a. 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.

). (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.

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.

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

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

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

NOBOOT	mnemonic	comments	NORMAL (OLD)
21	lxi h,0	;get system SP into HL	31
	.		
	.		
	.		
39	dad sp		00
22	shld spsave	;save until exit	00
79			00
05 or 47		;(47h for modified CP/M)	00
CD	call sppatch	;compute new SP value	00
34			00
01 or 43		;(43h for modified CP/M)	00
F9	sphl	;place into SP reg	00
	.		
	.		
	.		
	retpatch:		
2A	lhld spsave	;this is a patch from	C3
79		;the "vexit" routine,	FB
05 or 47		;to restore system SP...	0C
F9	sphl		CD
C9	ret	;...and return to CCP	96
	sppatch:		
2A	lhld ram+6	;get bdos pointer	0D
06			FE
00 or 42		;(42h for modified CP/M)	38
11	lxi d,-3000	;offset to bypass	CA
48		;the CCP	7B
F4			0C
19	dad d	;leave new SP value in HL	E6
C9	ret	;in HL and return	08
00			C4
00			82
00			11
	.		
	.		
	.		
C3	jmp retpatch	;ready to exit...now reset	C3
2F		; SP and return to CCP	00
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.

==========
BDS C v1.46 (c) 1982 by Leor Zolman
==========

This file describes all files supplied on the BDS C v1.46 distribution disk, and also lists all documentation that should be included in the package as purchased from Lifeboat.

FILES:

C1.COM, CC2.COM:		BDS C Compiler (parts I and II)
CLINK.COM		BDS C Linker
CLIB.COM		BDS C Librarian
DEFF.CRL, DEFF2.CRL		BDS C Standard Library object files
C.CCC		BDS C Run-time package ojbect code
BDSCIO.H		Standard C header file
STDLIB1.C, STDLIB2.C		Sources to the C-coded parts of the standard library (object in DEFF.CRL)
DEFF2.CSM, DEFF2A.CSM	(new)	Sources to the assembly-coded parts of the standard library (object in DEFF2.CRL)
BDS.LIB		Header file used for assembly-language function generation
CCC.ASM		Source to the run-time package
DIO.C, DIO.H		Directed I/O library, allowing for directed input, directed output and pipes (a la Unix*) (*Unix is a trademark of Bell Laboratories)
WILDEXP.C	(new)	Command line wild-card expansion utility
CASM.C, CASM.SUB	(new)	CSM-to-CRL assembly language preprocessor and companion submit-file.
FLOAT.DOC, FLOAT.C, FLOATSUM.C		Bob Mathias's floating point utility package
CONVERT.C, CCOT.C		Utilities for using BDS C on upper-case only terminals (such as the TRS-80 Mod I)
TELNET.C		A telecommunications program
GTHELLO.C		A game program
SIEVE.C		A benchmark, taken from the BYTE magazine high-level language benchmark article. Directions are included on how to make it compile and run a lot faster than it did in the article...
LPR.C		A line-printer driver utility.
SUBOOT.C	(new)	A utility to make C-generated COM files return quickly to the CCP after execution, instead of performing a warm-boot.
ALPH.C	(new)	A line-oriented file alphabetizing utility.

DOCUMENTATION:

The BDS C User's Guide	75 pages
The Kernighan & Ritchie book	228 pages

he following documentation items may either be bound in with the User's Guide
r included separately:

v1.46 User's Guide addenda	4 pages	(new)
v1.45 " " "	7 pages	
v1.44 " " "	4 pages	
v1.43 " " "	4 pages	
CASM Document	3 pages	(new)
Standard Library Summary	5 pages	(new)
Dynamic Overlay Guide	4 pages	
File I/O Tutorial	8 pages	
Console I/O Tutorial	5 pages	
Telnet Guide	4 pages	
BDS C User's Group application forms	2 pages	

```
/*
        The BDS C Standard I/O header file --  v1.46    3/4/82

        This file contains global definitions, for use in all C programs
        in PLACE of (yechhh) CONSTANTS. Characteristics of your system such
        as video screen size, interface port numbers and masks, buffered I/O
        allocations, etc., should all be configured just once within this
        file. Any program which needs them should contain the preprocessor
        directive:

                #include "bdscio.h"

        near the beginning.
        Go through and set all this stuff as soon as you get the package,
        and most terminal-dependent sample programs should run much better.
*/


/*
        Some console (video) terminal characteristics:
        (configured for ECS 4500)
*/

#define TWIDTH  80      /* # of columns */
#define TLENGTH 24      /* # of lines   */
#define CLEARS  "\014"  /* String to clear screen on console    */
#define INTOREV "\033I" /* String to switch console into reverse video  */
#define OUTAREV "\033N" /* String to switch console OUT of reverse video  */
#define CURSOROFF "\033Z"       /* String to turn cursor off    */
#define CURSORON "\033N"        /* String to turn cursor on     */
#define ESC     '\033'  /* Standard ASCII 'escape' character    */

/*
        Console serial port characteristics:
*/

#define CSTAT   0x09    /* status port  */
#define CDATA   0x08    /* data port    */
#define CIMASK  0x02    /* input data ready mask   */
#define COMASK  0x01    /* output data ready mask  */
#define CAHI    1       /* True if status active high   */
#define CRESET  0       /* True if status port needs to be reset after input */
#define CRESETVAL 0     /* If CRESET is true, this is the value to send */

/*
        Modem characteristics:
*/

#define MSTAT   0x09    /* status port  */
#define MDATA   0x08    /* data port    */
#define MIMASK  0x02    /* input data ready mask        */
#define MOMASK  0x01    /* ready to send a character mask       */
#define MAHI    1       /* True if status logic active high  */
#define MRESET  0       /* True if status port needs to be reset */
#define MRESETVAL 0     /* If MRESET true, this is the byte to send */

/*
        General purpose Symbolic constants:
*/
```

```
#define BASE 0          /* Base of CP/M system RAM (0 or 0x4200)  */
#define NULL 0
#define EOF -1          /* Physical EOF returned by low level I/O functions */
#define ERROR -1        /* General "on error" return value */
#define OK 0            /* General purpose "no error" return value */
#define JBUFSIZE 6      /* Length of setjump/longjump buffer    */
#define CPMEOF 0x1a     /* CP/M End-of-text-file marker (sometimes!)  */
#define SECSIZ 128      /* Sector size for CP/M read/write calls */
#define MAXLINE 135     /* Longest line of input expected from the console */
#define TRUE 1          /* general purpose true truth value     */
#define FALSE 0         /* general purpose false truth value    */

/*
   The NSECTS symbol controls the compilation of the buffered
   I/O routines within STDLIB2.C, allowing each user to set the
   buffer size most convenient for his system, while keeping
   the numbers totally invisible to the C source programs using
   buffered I/O (via the BUFSIZ defined symbol.) For larger
   NSECTS, the disk I/O is faster...but more ram is taken up.
   To change the buffer size allocation, follow these steps:

     1) Alter NSECTS to the desired value here in bdscio.h
     2) Re-compile STDLIB1.C and STDLIB2.C
     3) Use CLIB to combine STDLIB1.CRL and STDLIB2.CRL to make
        a new DEFF.CRL.

   Make sure you use declare all your I/O buffers with the a
   statement such as:

        char buf_name[BUFSIZ];
*/

#define NSECTS 8        /* Number of sectors to buffer up in ram */

#define BUFSIZ (NSECTS * SECSIZ + 6 )   /* Don't touch this */

struct _buf {                           /* Or this...       */
        int _fd;
        int _nleft;
        char *_nextp;
        char _buff[NSECTS * SECSIZ];
};

#define FILE struct _buf        /* Poor man's "typedef" */

/*
        If you plan to use the high-level storage allocation functions
        from the library ("alloc" and "free") then:

          1) Uncomment (enable) the "ALLOC_ON" definition, and comment out the
             "ALLOC_OFF" definition from this file.

          2) Re-compile STDLIB1.C, and use CLIB to transfer "alloc"
             and "free" into the DEFF.CRL library file.

          3) THIS IS IMPORTANT!!! Include the statement:

                _allocp = NULL;    /* initialize allocation pointer */

             somewhere in your "main" function PRIOR to the first use
```

```
            of the "alloc" function. DON'T FORGET THIS INITIALIZATION!!

        Remember to include bdscio.h in ALL files of your C program.
*/

#define ALLOC_OFF 1     /* disables storage allocation if uncommented */

                        /* only ONE of these two lines should be uncommented */

/*
#define ALLOC_ON 1      /* enables storage allocation if uncommented */
*/

#ifdef ALLOC_ON                 /* if storage allocation enabled, */

struct _header  {
        struct _header *_ptr;
        unsigned _size;
 };

struct _header _base;           /* declare this external data to  */
struct _header *_allocp;        /* be used by alloc() and free()  */

#endif
```

```
#include "a:bdscio.h"
#define float char
#define short char
#define string char
#define byte char       /* purge, more trouble than worth */
#define boolean char
#define YES 1
#define NO 0
#define NONE 0
#define EMPTY 0
#define DEL 0x7F
#define RUB_OUT 0x7F
#define AND &&
#define OR ||
#define NOT !
#define INPUT 0                 /* code for 'open()' */
#define OUTPUT 1
#define RANDOM 2
#define BASE 0                  /* code for 'seek()' */
#define FROM_BASE 0             /* r/w = offset */
#define FROM_HERE 1             /* r/w = current value + offset */
#define IS_BEFORE <0            /* code for 'strcmp()' */
#define IS_AFTER >0
#define IS_SAME ==0
#define IS_DIFFERENT !=0
#define BEFORE -1
#define AFTER 1
#define SAME 0

#define FLAG char
#define FILE struct _file
struct _file {
        int _rfd;               /* file descripter */
        int _secs;              /* # of sectors in buffer */
        unsigned _frstsec;      /* first sector in buffer */
        unsigned _cursec;       /* cp/m current random sec */
        byte _curbyt;           /* current random byte */
        byte *_nxtbyt;          /* next byte to be processed */
        byte *_bufbase;         /* location of base */
        byte *_pastbuf;         /* first byte beyond end of buffer */
        char _mode;             /* read, write, append, or direct */
        int _update;            /* buffer modified flag */
        unsigned _curblk;       /* currently addressed block */
        int _blksiz;            /* size of a logical block */
};

struct _file_ptr {
        unsigned _sector;
        byte _byt;
};
```

```
/*
        External data used by DIO.C for directed I/O simulation:
        (BDSCIO.H must also be #included in the main file.)
*/

/*
#define BUF_CONS 1                      /* uncomment if console buffering is
                                                desired (see DIO.C)       */
*/

char _diflag, _doflag;                  /* flag if directed I/O being used */
char _pipef, *_pipedest;                /* true if a pipe is being filled  */
char **_savei, **_nullpos;              /* used to remember position in
                                           command line when piping       */
char _dibuf[BUFSIZ], _dobuf[BUFSIZ];    /* I/O buffers used for direction  */

#ifdef BUF_CONS                         /* console buffering data          */

char _conbuf[MAXLINE + 2];              /* console input buffer used for
                                           non-directed standard input    */
char *_conbufp;                         /* pointer to next character to
                                                read from console buffer  */
#endif
```

```
/* PORTIO.H - I/O buffer data type and related definitions for use
   with the I/O routines in PORTIO.C.  */

struct iobuf {
        int fd;
        int isect;                      /* currently buffered sector */
        int nextc;                      /* index of next char in buffer */
        char written;                   /* anything written in current sector? */
        char buff [128];
        };
#define ABSOLUTE        0               /* seek codes */
#define RELATIVE        1
```

/*

Directed I/O package for BDS C v1.45 LZ -- 12/81

The following functions make up the directed I/O library:

1. dioinit(&argc,argv)	Make this the first thing you do in your "main" function, to process redirection commands on the CP/M command line.
2. getchar()	Gets a character from the keyboard, or from a directed input file if one was specified on the command line.
3. putchar(c)	Puts a character out to the console, or to a directed output file if one was specified on the command line.
4. dioflush()	Flushes directed output file, if open, and closes all directed I/O files (if any.) This must be called before your program exits or returns to CP/M.

To activate redirection: Four special arguments may be given on the command line to the generated COM file...

>foo	causes "putchar" to place characters into the file named "foo" instead of to the console.
+foo	like >foo except that the characters are ALSO sent to the console.
<foo	causes "getchar" to return characters from the file named "foo" instead of from the keyboard.
command !prog	causes the standard output of the command specified in "command" to be fed into the standard input of another program, "prog". (BOTH "command" and "prog" must be compiled with DIO)

(Note that there must never be any spaces between >,+,< or ! and the corresponding filename.)

When no "<" or "!" operator is used, standard input comes from the console and all standard line editing characters are recognized (a new feature of v1.45). To indicate end-of-file, you must type
^Z <CR>
(control-Z followed by a carriage-return.)

When no ">" or "!" operator is used, standard output goes to the console.

A program allowing redirection must have the following form:

```
#include "bdscio.h"            /* standard header file */
#include "dio.h"               /* directed I/O header  */

...                            /* other externals, if any */

main(argc,argv)
```

```
char **argv;
{
        ...                     /* declarations         */
        dioinit(&argc,argv);    /* initialize redirection */
        .
        .                       /* body of program      */
        .
        dioflush();             /* clean up redirection */
}
```

NOTES:

0. The console input may be raw (unbuffered, one char. at a time) or buffered (entire line must be typed before chars are returned, allowing standard editing features, and characters come back one at a time AFTER the entire line is typed). The default is raw; to have buffered console input, uncomment the "#define BUF_CONS" line in DIO.H and recompile this file and all files in your program.

1. Redirection and pipes work only for TEXT. This mechanism should not be used for binary data.

2. Use "-f dio" to link the program; this ensures that the proper versions of "getchar" and "putchar" are used. Do not define your own "getchar" or "putchar", or things will get confused.

3. Multiple pipes may be chained on one command line. For example, the following command feeds the output of program "foo" into the input of program "bar", the output of "bar" into the input of program "zot", and the output of "zot" into a file called "output":

        A>foo arg1 !bar !zot arg2 arg3 >output <cr>

   "arg1" is an actual argument to "foo", and "arg2" and "arg3" are actual arguments to "zot". This illustrates how actual arguments may be interspersed with redirection commands. The programs see the actual arguments, but command line preprocessing handled by the "dioinit" function cause the programs to never need to know about the redirection commands. Note that all three programs ("foo", "bar" and "zot") must have been compiled and linked to use the "DIO" package.

```
*/

#include "bdscio.h"
#include "dio.h"

#define CON_INPUT       1               /* BDOS call to read console       */
#define CON_OUTPUT      2               /* BDOS call to write to console   */
#define CON_STATUS      11              /* BDOS call to interrogate status */

#define CONTROL_C       3               /* Quit character                  */
#define STDERR          4               /* Standard Error descriptor (sorry,
                                           Unix fans, 2 was already used.) */
#define INPIPE          2               /* bit setting to indicate directed
                                           input from a temp. pipe file    */
#define VERBOSE         2               /* bit setting to indicate output is to
                                           go to console AND directed output */

/*
        The "dioinit" function must be called at the beginning of the
```

```
        "main" function:
*/

#define argc *argcp

dioinit(argcp,argv)
int *argcp;
char **argv;
{
        int i,j, argcount;

        _diflag = _doflag = _pipef = FALSE;  /* No directed I/O by default   */
        _nullpos = &argv[argc];

#ifdef BUF_CONS
        _conbuf[0] = 0;                 /* no characters in buffer yet  */
        _conbufp = _conbuf;             /* point to null buffer         */
#endif

        argcount = 1;

        for (i = 1; i < argc; i++)      /* Scan the command line for > and < */
        {
                if (_pipef) break;
                switch(*argv[i]) {

                   case '<':            /* Check for directed input:    */
                        if (!argv[i][1]) goto barf;
                        if (fopen(&argv[i][1], _dibuf) == ERROR)
                        {
                                fprintf(STDERR,"Can't open %s\n",&argv[i][1]);
                                exit();
                        }
                        _diflag = TRUE;
                        if (strcmp(argv[i],"<TEMPIN.$$$") == 0)
                                _diflag |= INPIPE;
                        goto movargv;

                   case '|':    /* Check for pipe: */
                        _pipef++;
                        _pipedest = &argv[i][1]; /* save prog name for execl */
                        if (argv[i][1])
                        {
                                argv[i] = ".TEMPOUT.$$$";  /* temp. output */
                                _savei = &argv[i];
                        }
                        goto foo;

                   case '+':
                        _doflag |= VERBOSE;

                foo:    case '>':   /* Check for directed output    */

                        if (!argv[i][1])
                        {
                         barf:   fprintf(STDERR,"Bad redirection/pipe specifier");
                                 exit();
                        }
                        unlink(&argv[i][1]);
                        if (fcreat(&argv[i][1], _dobuf) == ERROR)
```

```
                {
                        fprintf(STDERR,"Can't create %s\n",&argv[i][1]);
                        exit();
                }
                _doflag++;

        movargv:  if (!_pipef) {
                        for (j = i; j < argc; j++) argv[j] = argv[j+1];
                        (argc)--;
                        i--;
                        _nullpos--;
                 } else {
                        argc = argcount;
                        argv[argc] = 0;
                 }
                break;

            default:    /* handle normal arguments: */
                argcount++;
        }
    }
}


#undef argc

/*
    The "dioflush" function must be called before exiting the program:
*/

dioflush()
{
    if (_diflag)
    {
        fclose(_dibuf);
        if (_diflag & INPIPE) unlink("tempin.$$$");
    }

    if (_doflag)
    {
        putc(CPMEOF,_dobuf);
        fflush(_dobuf);
        fclose(_dobuf);
        unlink("tempin.$$$");   /* in case previous pipe was aborted */
        rename("tempout.$$$","tempin.$$$");
        if (_pipef)
        {
            *_savei = "<TEMPIN.$$$";
            *_nullpos = NULL;
            if (execv(_pipedest,_savei) == ERROR)
            {
                fprintf(STDERR,"\7Broken pipe\n");
                exit();
            }
        }
    }
}


/*
```

```
        This version of "getchar" replaces the regular version when using
        directed I/O. Note that the "BUF_CONS" defined symbol (in DIO.H)
        controls whether the console input is to be raw or buffered (see
        item 0. in NOTES above)
*/

getchar()
{
        int c;

        if (_diflag) {
                if ((c = getc(_dibuf)) == '\r') c = getc(_dibuf);
        }
        else

#ifdef BUF_CONS         /* For buffered console input, get a line of text   */
        {               /* from the BDOS (using "gets"), & insert newline:  */
                if (!*_conbufp) {
                        gets(_conbufp = _conbuf);
                        _conbuf[strlen(_conbuf) + 1] = '\0';
                        _conbuf[strlen(_conbuf)] = '\n';
                }
                c = *_conbufp++;
        }
#else                   /* for raw console input, simulate normal "getchar": */
                if ((c = bdos(CON_INPUT)) == CONTROL_C) exit();
#endif

        if (c == CPMEOF) return EOF;        /* Control-Z is EOF key    */

        if (c == '\r')
        {
                c = '\n';
#ifndef BUF_CONS
                if (!_diflag) bdos(2,'\n');  /* echo LF after CR to console */
#endif
        }
        return c;
}


/*
        This version of "putchar" replaces the regular version when using
        directed I/O:
*/

putchar(c)
char c;
{
        char *static;
        static = "";    /* remembers last character sent; start out null */

        if (_doflag)
        {
                if (c == '\n' && *static != '\r') putc('\r',_dobuf);
                *static = c;
                if(putc(c,_dobuf) == ERROR)
                {
                        fprintf(STDERR,"File output error: disk full?\n");
                        exit();
```

```
        }
        if (!(_doflag & VERBOSE)) return;
    }

    if (bdos(CON_STATUS) && bdos(CON_INPUT) == CONTROL_C) exit();
    if (c == '\n' && *static != '\r') bdos(CON_OUTPUT,'\r');
    bdos(CON_OUTPUT,c);
    *static = c;
}
```

```
/* Portable I/O Package functions */
/* Written by EBM on 13 DEC 1981  */

/* i/o buffer data type */
#include "portio.h"

#define TRUE            (-1)
#define FALSE   0

int copen (buf, name)
        struct iobuf *buf;
        char *name;

{       buf->isect = -1;        /* set values to force initial read */
        buf->nextc = 128;
        buf->written = FALSE;
        return (buf->fd = open (name, 2));
        }

int ccreat (buf, name)
        struct iobuf *buf;
        char *name;

{       buf->isect = 0; /* don't force initial write! */
        buf->nextc = 0;
        buf->written = FALSE;
        if ((buf->fd = creat (name)) < 0 || close (buf->fd) < 0) return (-1);
        return (buf->fd = open (name, 2));
        }

int cclose (buf)
        struct iobuf *buf;

{       if (cforce (buf) < 0) return (-1);
        return (close (buf->fd));
        }

int cread (buf, loc, len)
        struct iobuf *buf;
        char *loc;
        unsigned len;

{       char *oldloc;
        unsigned amt;

        oldloc = loc;
        while (len) {
                if ((amt = min (len, 128 - buf->nextc)) <= 0) {
                        if (cforce (buf) < 0  ||
                            seek (buf->fd, ++buf->isect, ABSOLUTE) < 0  ||
                            read (buf->fd, buf->buff, 1) != 1) break;
                        buf->nextc = 0;
                        continue;
                        }
                movmem (&buf->buff[buf->nextc], loc, amt);
                buf->nextc += amt;
                loc += amt;
                len -= amt;
                }
        return (loc - oldloc);
```

```
        }

int cwrite (buf, loc, len)
        struct iobuf *buf;
        char *loc;
        int len;

{       char *oldloc;
        unsigned amt;

        oldloc = loc;
        while (len) {
                if ((amt = min (len, 128 - buf->nextc)) <= 0) {
                        if (cforce (buf) < 0) break;
                        ++buf->isect;
                        buf->nextc = 0;
                        continue;
                        }
                movmem (loc, &buf->buff[buf->nextc], amt);
                buf->nextc += amt;
                loc += amt;
                len -= amt;
                buf->written = TRUE;
                }
        return (loc - oldloc);
        }

int cforce (buf)
        struct iobuf *buf;

{       if (buf->nextc > 0 && buf->written &&
            (seek (buf->fd, buf->isect, ABSOLUTE) < 0  ||
             write (buf->fd, buf->buff, 1) <= 0)) return (-1);
        buf->written = FALSE;
        return (1);
        }

int cflush (buf)
        struct iobuf *buf;

{       if (buf->nextc & 0x7f) {
                setmem (&buf->buff[buf->nextc], 128 - buf->nextc, ('Z' - '@'));
                buf->written = TRUE;
                }
        return (cforce (buf));
        }

int cseek (buf, amt, mode)
        struct iobuf *buf;
        int amt, mode;

{       int newsect, newpos;

        if (mode == RELATIVE)
                {if (amt < 0) {         /* backwards */
                        amt = -amt;
                        newsect = buf->isect - (amt >> 7);
                        newpos = buf->nextc - (amt & 0x7f);
                        while (newpos < 0) {
                                newpos += 128;
```

```
                --newsect;
                }
            if (newsect < 0) return (-1);
            }
        else    {
            newsect = buf->isect + (amt >> 7);
            newPos = buf->nextc + (amt & 0x7f);
            while (newPos >= 128) {
                newPos -= 128;
                ++newsect;
                }
            }
        }
else if (mode == ABSOLUTE) {
        if (amt < 0) return (-1);
        newsect = (amt >> 7);
        newPos = (amt & 0x7f);
        }
else return (-1);
if (newsect != buf->isect  &&
    (cforce (buf) < 0  ||
        seek (buf->fd, newsect, ABSOLUTE) < 0  ||
        read (buf->fd, buf->buff, 1) != 1)) return (-1);
buf->isect = newsect;
buf->nextc = newPos;
buf->written = FALSE;
return (1);
}
```

```
/*
        Directed I/O package for use with BDS C v1.4x.

        The following functions make up the directed I/O library:

        1. dioinit(&argc,argv)          Make this the first thing you do in
                                        your "main" function, to process
                                        redirection commands on the CP/M
                                        command line.

        2. getchar()                    Gets a character from the keyboard,
                                        or from a directed input file if one
                                        was specified on the command line.

        3. putchar(c)                   Puts a character out to the console,
                                        or to a directed output file if one
                                        was specified on the command line.

        4. dioflush()                   Flushes directed output file, if open,
                                        and closes all directed I/O files (if
                                        any.) This must be called before your
                                        program exits or returns to CP/M.

        To activate redirection: Four special arguments may be given
        on the command line to the generated COM file...

                >foo    causes "putchar" to place characters into the file
                        named "foo" instead of to the console.

                +foo    like >foo except that the characters are ALSO sent
                        to the console.

                <foo    causes "getchar" to return characters from the file
                        named "foo" instead of from the keyboard.

         command !prog  causes the standard output of the command specified in
                        "command" to be fed into the standard input of another
                        program, "prog". (BOTH "command" and "prog" must be
                        compiled with DIO)

         (Note that there must never be any spaces between >,+,< or ! and the
          corresponding filename.)

        Thus, a C program using redirection has the following form:

                #include "bdscio.h"          /* standard header file */
                #include "dio.h"             /* directed I/O header  */

                ...                          /* other externals, if any */

                main(argc,argv)
                char **argv;
                {
                        ...                  /* declarations        */
                        dioinit(&argc,argv)  /* initialize redirection */
                        ...                  /* body of program     */
                        dioflush();
                }

        NOTES:
```

```
        0. Redirection and pipes work only for TEXT. This mechanism should
           not be used for binary data.

        1. The "getchar" and "putchar" functions should each be used EXPLICITLY
           at least once in your main source file, so that the correct versions
           are picked off from DIO.CRL instead of the incorrect ones from
           DEFF2.CRL (because of the way the linker works.)

        2. The "putc" library function should be modified so that an iobuf
           value of 4 sends a character to the CP/M console via a "bdos"
           call (as opposed to using "putchar"), and that a '\n' character
           thus sent should be expanded into a CR-LF combination. This
           is easily accomplished by adding the following clause to the "putc"
           function, recompiling STDLIB1.C, and updating DEFF.CRL by
           transferring in the new "putc" with CLIB.COM:

                if (_iobuf == 4) {
                        if (c == '\n') bdos(2,'\r');
                        bdos(2,c);
                }
           (This may already have been done in the version you have.)

        3. The "execv" function, used by this package, is available in the
           file EXECV.ASM; it should be assembled, renamed EXECV.CRL, and
           then transferred into DEFF2.CRL using CLIB.COM.
           (This may already have been done in the version you have.)

*/

#include "bdscio.h"
#include "dio.h"

#define CON_INPUT 1                     /* BDOS call to read console       */
#define CON_OUTPUT 2                    /* BDOS call to write to console   */
#define CON_STATUS 11                   /* BDOS call to interrogate status */

#define CONTROL_C 3                     /* Quit character                  */
#define STDERR 4                        /* Standard Error descriptor (sorry,
                                           Unix fans, 2 was already used.) */
#define INPIPE 2                        /* bit setting to indicate directed
                                           input from a temp. pipe fil     */
#define VERBOSE 2                       /* bit setting to indicate output is to
                                           go to console AND directed output */

/*
        The "dioinit" function must be called at the beginning of the
        "main" function:
*/

#define argc *argcp

dioinit(argcp,argv)
int *argcp;
char **argv;
{
        int i,j, argcount;

        _diflag = _doflag = _pipef = FALSE;  /* No directed I/O by default   */
        _nullpos = &argv[argc];
```

```
argscount = 1;

for (i = 1; i < argc; i++)      /* Scan the command line for > and < */
{
        if (_pipef) break;
        switch(*argv[i]) {

           case '<':            /* Check for directed input: */
                if (!argv[i][1]) goto barf;
                if (fopen(&argv[i][1], _dibuf) == ERROR)
                {
                        fprintf(STDERR,"Can't open %s\n",&argv[i][1]);
                        exit();
                }
                _diflag = TRUE;
                if (strcmp(argv[i],"<TEMPIN.$$$") == 0)
                        _diflag |= INPIPE;
                goto movargv;

           case '!':    /* Check for pipe: */
                _pipef++;
                _pipedest = &argv[i][1]; /* save prog name for execl */
                if (argv[i][1])
                {
                        argv[i] = ".TEMPOUT.$$$";  /* temp. output */
                        _savei = &argv[i];
                }
                goto foo;

           case '+':
                _doflag |= VERBOSE;

     foo:   case '>':   /* Check for directed output    */

                if (!argv[i][1])
                {
            barf:   fprintf(STDERR,"Bad redirection/pipe specifier");
                    exit();
                }
                unlink(&argv[i][1]);
                if (fcreat(&argv[i][1], _dobuf) == ERROR)
                {
                      fprintf(STDERR,"Can't create %s\n",&argv[i][1]);
                      exit();
                }
                _doflag++;

     movargv:   if (!_pipef) {
                        for (j = i; j < argc; j++) argv[j] = argv[j+1];
                        (argc)--;
                        i--;
                        _nullpos--;
                 } else {
                        argc = argscount;
                        argv[argc] = 0;
                 }
                break;

            default:    /* handle normal arguments: */
                argscount++;
```

```
                }
        }
}


#undef argc

/*
        The "dioflush" function must be called before exiting the program:
*/

dioflush()
{
        if (_diflag)
        {
                fclose(_dibuf);
        }

        /* Cleanup unconditionally so rename below can't screw up */
        unlink("tempin.$$$");

}



/*
        This version of "getchar" replaces the regular version when using
        directed I/O:
*/

getchar()
{
        char c;

        if (_diflag) {
                if ((c = getc(_dibuf)) == '\r') c = getc(_dibuf);
        } else
                if ((c = bdos(CON_INPUT)) == CONTROL_C) exit();

        if (c == CPMEOF) return EOF;            /* Control-Z is EOF key    */
        if (c == '\r')
        {
                c = '\n';
                if (!_diflag) bdos(2,'\n');  /* echo LF after CR to console */
        }
        return c;
}



/*
        This version of "putchar" replaces the regular version when using
        directed I/O:
*/

putchar(c)
char c;
{
        if (_doflag)
        {
                if (c == '\n') putc('\r',_dobuf);
                if(putc(c,_dobuf) == ERROR)
```

```
            {
                    fprintf(STDERR,"File output error: disk full?\n");
                    exit();
            }
            if (!(_doflag & VERBOSE)) return;
    }

    if (bdos(CON_STATUS) && bdos(CON_INPUT) == CONTROL_C) exit();
    if (c == '\n') bdos(CON_OUTPUT,'\r');
    bdos(CON_OUTPUT,c);
}
```

```
;
; BD Software C Standard Library Machine Language Functions
; Written by Leor Zolman
; v1.46,       3/22/82
;
; This file is in "CSM" format; to convert to CRL format,
; use CASM.SUB in conjunction with CASM.COM, ASM.COM and DDT.COM.
;
; Functions appearing in this file:
;
;       getchar kbhit   ungetch putchar puts    gets    rand    srand
;       srand1  nrand   csw     setmem  movmem  call    calla   inp
;       outp    peek    poke    sleep   pause   setfcb  read    write
;       open    close   creat   unlink  seek    tell    rename  fabort
;       fcbaddr exit    bdos    bios    codend  externs endext  topofmem
;       exec    execl   execv   sbrk    rsvstk
;

        maclib bds

        FUNCTION        getchar
        lda     ungetl  ;any character pushed back?
        ora     a
        mov     l,a
        jz      gch2
        xra     a       ;yes, return it and clear the pushback
        sta     ungetl  ;byte in C.CCC.
        mvi     h,0
        ret

gch2:   push    b
        mvi     c,conin
        call    bdos
        pop     b
        cpi     cntrlc  ;control-C ?
        jz      base    ;if so, reboot.
        cpi     1ah     ;control-Z ?
        lxi     h,-1    ;if so, return -1.
        rz
        mov     l,a
        cpi     cr      ;carriage return?
        jnz     gch3
        push    b
        mvi     c,conout        ;if so, also echo linefeed
        mvi     e,lf
        call    bdos
        pop     b
        mvi     l,newlin        ;and return newline (linefeed)..

gch3:   mvi     h,0
        ret
        ENDFUNC

        FUNCTION        kbhit
        lda     ungetl  ;any character ungotten?
        mvi     h,0
        mov     l,a
        ora     a
        rnz             ;if so, return true
```

```
        push    b
        mvi     c,cstat ;else interrogate console status
        call    bdos
        pop     b

        ora     a       ;0 returned by BDOS if no character ready
        lxi     h,0
        rz              ;return 0 in HL if no character ready
        inr     l       ;otherwise return 1 in HL
        ret
        ENDFUNC kbhit

        FUNCTION        unsetch
        lda     unsetl
        mov     l,a
        push    h
        call    ma2toh
        sta     unsetl
        pop     h
        mvi     h,0
        ret
        ENDFUNC unsetch

        FUNCTION        putchar
        call    maltoh  ;get character in A
        push    b
        mvi     c,conout
        cpi     newlin  ;newline?
JRNZ    jnz     put1    ;if not, just go put out the character
        mvi     e,cr    ;else...put out CR-LF
        call    bdos
        mvi     c,conout
        mvi     a,lf

put1:   mov     e,a
        call    bdos

put2:   mvi     c,cstat ;now, is input present at the console?
        call    bdos
        ora     a
JRNZ    jnz     put3
        pop     b       ;no...all done.
        ret

put3:   mvi     c,conin ;yes, sample it (this will always echo the
        call    bdos    ;       character to the screen, alas)
        cpi     cntrlc  ;is it control-C?
        jz      base    ;if so, abort and reboot
        pop     b       ;else ignore it.
        ret
        ENDFUNC

        FUNCTION        putch
        call    maltoh
        push    b
        mvi     c,conout
        mov     e,a
        cpi     newlin
        jnz     putch1  ;if not newline, just put it out
        mvi     e,cr    ;else put out CR-LF
```

```
        call    bdos
        mvi     c,conout
        mvi     e,lf
putch1: call    bdos
        pop     b
        ret
        ENDFUNC

        FUNCTION        gets
        call    maltoh  ;get destination address
        push    b       ;save BC
        push    h
        push    h
        lxi     h,-150  ;use space below stack for reading line
        dad     sp
        push    h       ;save buffer address
        mvi     m,88h   ;Allow a max of about 135 characters
        mvi     c,getlin
        xchg            ;put buffer addr in DE
        call    bdos    ;get the input line
        mvi     c,conout
        mvi     e,lf    ;put out a LF
        call    bdos
        pop     h       ;get back buffer address
        inx     h       ;point to returned char count
        mov     b,m     ;set B equal to char count
        inx     h       ;HL points to first char of line
        pop     d       ;DE points to start destination area
copyl:  mov     a,b     ;copy line to start of buffer
        ora     a
        jz      gets2
        mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jmp     copyl

gets2:  xra     a       ;store terminating null
        stax    d
        pop     h       ;return buffer address in HL
        pop     b
        ret
        ENDFUNC


        FUNCTION        rand
        lhld    rseed
        xchg
        mvi     a,48h
        ana     e
        jz      rand1
        jpe     rand1
        stc
rand1:  lhld    rseed+2
        mov     a,h
        ral
        mov     h,a
        mov     a,l
        ral
```

XRA A
MOV C,A
LDIR

JR Z
JR
JR Z

```
        mov     l,a
        shld    rseed+2
        mov     a,d
        ralR    D
        mov     h,a     mov H,D
        mov     a,e
        ralR    E
        mov     l,a     mov L,E
SDED    shld    rseed
        mov     a,h
        ani     7fh
        mov     h,a
        ret
        ENDFUNC

        FUNCTION        srand
        call    maitoh
        mov     a,h
        ora     l
JRZ     jz      srand2
        shld    rseed
        shld    rseed+2
        ret

srand2: lxi     d,st$1
        push    b
        mvi     c,9
        call    bdos
        lxi     h,0bdbdh
srand3: push    h
        mvi     c,11
        call    bdos
        pop     h
        inx     h
        inx     h
        inx     h
        ani     1
JRZ     jz      srand3
        shld    rseed
        shld    rseed+2
        mvi     c,conout
        mvi     e,cr
        call    bdos
        mvi     c,conout
        mvi     e,lf
        call    bdos
        mvi     c,conin ;clear the character
        call    bdos
        pop     b
        ret
st$1:   db 'Wait a few seconds, and type a CR: $'
        ENDFUNC


        FUNCTION        srand1
        EXTERNAL        puts
        call    maitoh
        push    h
        call    puts    ;print prompt string
        pop     h
```

```
        push    b
        lxi     h,5678h
srla:   push    h
        mvi     c,cstat
        call    bdos
        pop     h
        inx     h
        inx     h
        inx     h
        ora     a
  JRZ   jz      srla
        shld    rseed
        shld    rseed+2
        pop     b
        ret
        ENDFUNC

        FUNCTION        nrand
        EXTERNAL        puts
        call    arghak
        lhld    arg1    ;get n (1st arg)
        mov     a,h
        ana     l
        cpi     255     ;was it -1 (set seed) ?
  JRNZ  jnz     nrand1
        lhld    arg2    ;copy seed
        shld    seed
        lhld    arg3
        shld    seed+2
        lhld    arg4
        shld    seed+4
        ret             ;all done

nrand1: push    b
        mov     a,h     ;look at first arg again
        ora     l
  JRNZ  jnz     nrand3  ;is it 0 (randomize)?
        lhld    arg2
        push    h       ;yes. print out string
        call    puts    ;call  puts
        pop     d
        lxi     h,5a97h ;yes. start w/something odd
nrand2: push    h
        mvi     c,cstat ;interrogate console status
        call    bdos
        pop     h
        inx     h       ;and keep it odd
        inx     h       ;and growing
        ora     a
  JRZ   jz      nrand2  ;until user types something.
        shld    seed    ;then plaster the value all over the
        shld    seed+2  ;seed.
        shld    seed+4
        pop     b
        ret

nrand3: lda     seed    ;now compute next random number. from this
        ori     1       ; point on, the code is that of Prof. Paul Gans
        sta     seed    ;lsb of SEED must be 1
```

```
        mvi     b,6     ;clear 6 PROD bytes to 0
        lxi     h,prod
randm1: mvi     m,0
        inx     h
        dcr     b
        jnz     randm1

        lxi     b,6     ;set byte counter
randm2: lxi     h,plier-1
        dad     b       ;make addr of lsb of PLIER
        mov     a,m     ;PLIER byte
        push    b       ;save byte counter
        mvi     b,8     ;set bit counter

randm3: mov     d,a     ;save PLIER byte
        lxi     h,prod  ;shift whole PROD left one bit
        mvi     c,6
        xra     a
randm4: mov     a,m     ;set byte
        ral             ;shift left
        mov     m,a     ;put byte
        inx     h
        dcr     c
        jnz     randm4

        mov     a,d     ;recover PLIER byte
        ral             ;look at current high bit
        jnc     randm6  ;0 means no add cycle

        push    psw     ;add SEED to PROD
        xra     a
        mvi     c,6
        lxi     h,prod
        lxi     d,seed
randm5: ldax    d
        adc     m
        mov     m,a
        inx     h
        inx     d
        dcr     c
        jnz     randm5
        pop     psw

randm6: dcr     b       ;test bit counter
        jnz     randm3  ;go cycle more bits
        pop     b       ;recover byte counter
        dcr     c       ;test it
        jnz     randm2  ;go process more bytes

        mvi     b,6     ;complement PROD, add 1 to it,
        lxi     h,seed  ;and transfer it to SEED.
        lxi     d,prod
        xra     a
        cmc
randm7: ldax    d
        cma
        aci     0
        mov     m,a
        inx     h
        inx     d
```

```
        dcr     b
        jnz     randm7

        dcx     h       ;put the two high order bytes
        mov     a,m     ;into HL for return to C, not
        ani     7fh     ;neglecting to zero the high
        mov     h,a     ;order bit so a positive int
        lda     seed+4  ;is returned
        mov     l,a
        pop     b
        ret

plier:  db      0c5h,87h,1
        db      0eh,9ah,0e0h

seed:   db      1,0,0,0,0,0

prod:   db      0,0,0,0,0,0
        ENDFUNC

        FUNCTION        csw
        in      255
        mov     l,a
        mvi     h,0
        ret
        ENDFUNC

        FUNCTION        setmem
        call    arghak
        push    b
        lhld    arg2
        xchg
        lhld    arg1
        lda     arg3
        mov     c,a
        inx     d
setm2:  dcx     d
        mov     a,d
        ora     e
        jnz     setm3
        pop     b
        ret

setm3:  mov     m,c
        inx     h
        jmp     setm2
        ENDFUNC

        FUNCTION        movmem
        call    arghak
        lhld    arg3    ;get block length
        mov     a,h
        ora     l
        rz              ;do nothing if zero length
        push    b
        mov     b,h
        mov     c,l     ;set BC to length
        lhld    arg2    ;get dest addr
        xchg            ;put in DE
        lhld    arg1    ;get source addr in HL
```

```
        call    cmphd   ;if source < dest, do tail-first
        jrc     tailf   ;else do head-first

headf:  mvi     a,2     ;test for Z-80
        inr     a
        jpe     m8080h  ;Z80?
        db      0edh,0b0h       ;yes. do block move.
        pop     b
        ret             ;and done.

m8080h: mov     a,m
        stax    d
        inx     h
        inx     d
        dcx     b
        mov     a,b
        ora     c
        jnz     m8080h
        pop     b
        ret

tailf:  dcx     b       ;tail first. Compute new source
        dad     b       ;and destination addresses
        xchg
        dad     b
        xchg
        inx     b
        mvi     a,2     ;test for Z80
        inr     a
        jpe     m8080t  ;Z80?
        db      0edh,0b8h       ;yes. do block move.
        pop     b
        ret

m8080t: mov     a,m
        stax    d
        dcx     h
        dcx     d
        dcx     b
        mov     a,b
        ora     c
        jnz     m8080t
        pop     b
        ret

cmphd:  mov     a,h
        cmp     d
        rnz
        mov     a,l
        cmp     e
        ret
        ENDFUNC

        FUNCTION        call
        call    argshak
        push    b
        lhld    arg5
        xchg
        lhld    arg4
        mov     b,h
```

```
        mov     c,l
        lda     arg2
        lxi     h,call2
        push    h
        lhld    arg1
        push    h
        lhld    arg3
        ret

call2:  pop     b
        ret
        ENDFUNC

        FUNCTION        calla
        call    arghak
        push    b
        lhld    arg5    ;get de value
        xchg
        lhld    arg4    ;get bc value
        mov     b,h
        mov     c,l
        lda     arg2    ;get a value
        lxi     h,calla2  ;get return address
        push    h       ;push   it
        lhld    arg1    ;get address of routine
        push    h
        lhld    arg3    ;get hl value
        ret             ;call   routine

calla2: mov     l,a     ;put A value in HL
        mvi     h,0     ;clear high byte
        pop     b
        ret
        ENDFUNC

        FUNCTION        inp
        call    maltoh
        sta     iohack+1        ;store as arg to ram area input subroutine
        call    iohack          ;call the subroutine to get value
        mov     l,a             ;and put into HL
        mvi     h,0
        ret
        ENDFUNC

        FUNCTION        outp
        call    maltoh          ;get port number
        sta     iohack+4        ;store as arg to ram area output subroutine
        call    ma2toh          ;get data byte
        call    iohack+3        ;output it
        ret
        ENDFUNC

        FUNCTION        peek
peek:   call    maltoh
        mov     l,m
        mvi     h,0
        ret
        ENDFUNC peek
```

```
        FUNCTION        poke
        call    argshak
        lhld    arg1
        lda     arg2
        mov     m,a
        ret
        ENDFUNC

        FUNCTION        sleep
        call    maltoh
        push    b
        inx     h
sl1:    dcx     h
        mov     a,h
        ora     l
 JRNZ   jnz     sl1a
        pop     b
        ret

sl1a:   lxi     d,10000
sl2:    dcx     d
        mov     a,d
        ora     e
 JRNZ   jnz     sl2
        push    h
        mvi     c,cstat
        call    bdos
        ora     a
        pop     h
   JRZ  jz      sl1
        push    h
        mvi     c,conin
        call    bdos
        cpi     cntrlc
        jz      base
        pop     h
   JR   jmp     sl1
        ENDFUNC

        FUNCTION        pause
        push    b
pausl:  mvi     c,cstat
        call    bdos
        ora     a
  JRZ   jz      pausl
        pop     b
        ret
        ENDFUNC


        FUNCTION        setfcb
        call    argshak
        push    b
        lhld    arg2    ;set pointer to name text
issp:   mov     a,m
        inx     h
        cpi     ' '
   JRZ  jz      issp
        cpi     tab
   JRZ  jz      issp
```

```
        dcx     h
        xchg            ;set DE pointing to 1st non-space char
        lhld    arg1    ;get --> fcb area
        call    setfcb  ; do it
        lxi     h,0     ;all OK.
        pop     b
        ret
        ENDFUNC

        FUNCTION        read
        call    arghak
        lda     arg1
        call    fgfd
        jc      error   ;error if illegal fd
        mov     a,m
        ani     2       ;open for read?
        jz      error   ;error if not
        push    b
        lda     arg1
        call    fgfcb
        shld    tmp2    ;tmp2 will hold dma addr
        lxi     h,0     ;count of # of successful sectors read
        shld    tmp2a   ; will be kept at tmp2a
read2:  lhld    arg3    ;done?
        mov     a,h
        ora     l
  JRZ   jz      read4

read2a: lhld    arg2    ;else read another sector
        xchg            ;DE is dma addr
        mvi     c,sdma
        call    bdos    ;set DMA
        lhld    tmp2
        xchg            ;DE is fcb addr
        mvi     c,reads
        push    d       ;save de so we can fudge nr field if
        call    bdos    ;we stop reading on extent boundary...
        pop     d               ; CP/M sucks!
        cpi     2
        pop     b
        jz      error   ;if error, abort
        push    b
        cpi     1
  JRNZ  jnz     read6   ;EOF?

read3:  lxi     h,32    ;yes. are we on extent boundary?
        dad     d       ;if so, adjust for CP/M's stupidity here
        mov     a,m     ;by turning an 80h sector count into 00h.
        cpi     80h
  JRNZ  jnz     read4
        mvi     m,0     ;yes. reset nr to 0...CP/M leaves it at 80h!
read4:  lhld    tmp2a
read5:  pop     b
        ret

read6:  lhld    arg3
        dcx     h
        shld    arg3
        lhld    arg2
        lxi     d,128
```

```
        dad     d
        shld    ars2
        lhld    tmp2a
        inx     h
        shld    tmp2a
        jmp     read2
        ENDFUNC

        FUNCTION        write
        call    arshak
        lda     ars1
        call    fsfd
        jc      error
        mov     a,m
        ani     4
        jz      error
        push    b
        lda     ars1
        call    fsfcb
        shld    tmp2
        lxi     h,0
        shld    tmp2a
        lxi     d,tbuff         ;80 for normal CP/M, else 4280
        mvi     c,sdma
        call    bdos

writ1:  lhld    ars3    ;done yet?
        mov     a,h
        ora     l
        lhld    tmp2a   ;if so, return count
        jz      writ3
        lhld    ars2    ;else copy next 128 bytes down to tbuff
        lxi     d,tbuff ;80 for normal CP/M, else 4280
        mvi     b,128
writ2:  mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jnz     writ2
        shld    ars2    ;save -> to next 128 bytes
        lhld    tmp2    ;set addr of fcb
        xchg
        mvi     c,writs ;go write
        call    bdos
        ora     a       ;error?
        lhld    tmp2a   ;if so, return # of successfully written
        jnz     writ3   ;  sectors.

        inx     h       ; else bump successful sector count,
        shld    tmp2a
        lhld    ars3    ; debump countdown,
        dcx     h
        shld    ars3
        jmp     writ1   ; and go try next sector
writ3:  pop     b
        ret
        ENDFUNC

        FUNCTION        open
```

```
        call    arghak
        xra     a
        call    fgfcb   ;any fcb's free?
        jc      error   ;if not, error
        sta     tmp
        xchg
        lhld    arg1
        xchg
        push    b
        call    setfcb
        mvi     c,openc
        call    bdos
        cpi     errorv  ;successful open?
        pop     b
        jz      error   ;if not, error
        lda     tmp
        call    fgfd    ;get HL pointing to fd table entry
        lda     arg2
        ora     a       ;open for read?
        mvi     d,3
JRZ     jz      open1
        dcr     a
        mvi     d,5
JRZ     jz      open1   ;write?
        dcr     a
        jnz     error   ;else must be both or bad mode.
        mvi     d,7
open1:  mov     m,d
        lda     tmp
        mov     l,a
        mvi     h,0
        ret
        ENDFUNC

        FUNCTION        close
        jmp     close   ;jump to the close routine in C.CCC
        ENDFUNC



        FUNCTION        creat
        EXTERNAL        unlink,open
        call    arghak
        lhld    arg1
        push    b
        push    h
        call    unlink  ;erase any old versions of file
        pop     d
        mvi     c,creatc
        lxi     d,fcb
        call    bdos
        cpi     errorv
        pop     b
        jz      error
        lxi     h,2
        push    h
        lhld    arg1
        push    h
        call    open
        pop     d
```

```
        pop     d
        ret
        ENDFUNC creat


        FUNCTION        unlink
        call    maltoh
        push    b
        xchg
        lxi     h,fcb
        call    setfcb
        mvi     c,delc
        call    bdos
        lxi     h,0
        pop     b
        ret
        ENDFUNC


        FUNCTION        seek
        EXTERNAL        tell
        call    arghak  ;copy arguments to args area
        lda     arg1
        call    fgfcb
        jc      error   ;error if file not open
        push    b
        push    h       ;save fcb address

        lhld    arg1
        push    h
        call    tell    ;get r/w pointer position for the file
        pop     d

        xchg            ;put present pos in DE
        lda     arg3
        lhld    arg2    ;get offset in HL
        ora     a       ;absolute offset?
JR2     jz      seek2   ;if so, offset is new position
        dad     d       ;else add offset to current position
seek2:  mov     a,l     ;convert to extent and sector values
        rlc
        mov     a,h
        ral
        ani     7fh
        sta     tmp
        xthl
        lxi     d,12
        push    h
        dad     d
        cmp     m       ;jumping over extent boundary?
JR2     jz      seek5
        xthl            ;yes.
        xchg
        mvi     c,closec        ;close old extent
        push    d
        call    bdos
        pop     d
        pop     h
        cpi     errorv
JRNZ    jnz     seek4
```

```
seek3:  pop     d
        pop     b
        jmp     error

seek4:  lda     tmp
        mov     m,a
        push    d
        mvi     c,openc ;and open new one.
        call    bdos
seek5:  pop     d
        cpi     errorv
        jrz     seek3
        lxi     h,32    ;and set nr field
        dad     d
        pop     d
        mov     a,e
        ani     7fh
        mov     m,a
        xchg            ;return new sector # in HL
        pop     b
        ret
        ENDFUNC

        FUNCTION        tell
        call    maitoh  ;set fd value in A
        call    fsfcb
        jc      error
        push    b
        lxi     d,12
        dad     d
        mov     b,m     ;put extent # in B
        lxi     d,20
        dad     d
        mov     c,m     ;put sector # in C
        xra     a       ;rotate extent right one bit, old b0 --> Carry
        mov     a,b
        rar
        mov     h,a     ;rotated value becomes high byte of tell position
        mvi     a,0     ;rotate b0 of extent into A
        rar
        mov     b,a     ;save rotated extent number in B
        add     c       ;add rotated extent number to sector number
        mov     l,a     ;and result becomes low byte of tell position
        mov     a,c     ;if both rotated extent # and sector # has bit 7 hi,
        ana     b       ;then the sum had an overflow, so...
        jp      tell2
        inr     h       ;bump position number by 256
tell2:  pop     b       ;and all done.
        ret
        ENDFUNC

        FUNCTION        rename
        call    arshak
        push    b
renam:  lhld    ars1
        xchg
        lxi     h,wfcb
        call    setfcb
        lhld    ars2
        xchg
```

```
        lxi     h,wfcb+16
        call    setfcb
        lxi     d,wfcb
        mvi     c,renc
        call    bdos
        pop     b
        cpi     errorv
        jz      error
        lxi     h,0
        ret
wfcb:   ds 53
        ENDFUNC

        FUNCTION        fabort
        call    maltoh
        call    fsfd
        jc      error
        mvi     m,0     ;clear entry in fd table
        lxi     h,0
        ret
        ENDFUNC

        FUNCTION        fcbaddr
        call    maltoh
        call    fsfd    ;is it an open file?
        jc      error
        call    maltoh
        call    fsfcb   ;set fcb addr in HL
        ret
        ENDFUNC

        FUNCTION        exit
        jmp     exit
        ENDFUNC

        FUNCTION        bdos
        call    arghak
        push    b
        lda     arg1    ;set C value
        mov     c,a
        lhld    arg2    ;set DE value
        xchg            ;put in DE
        call    bdos    ;make the bdos call
        pop     b
        ret             ;and return to caller
        ENDFUNC

        FUNCTION        bios
        call    arghak
        push    b
        lhld    base+1  ;set addr of jump table + 3
        dcx     h       ;set to addr of first jump
        dcx     h
        dcx     h
        lda     arg1    ;set function number (1-85)
        mov     b,a     ;multiply by 3
        add     a
        add     b
        mov     e,a     ;put in DE
        mvi     d,0
```

```
        dad     d       ;add to base of jump table
        push    h       ;and save for later
        lhld    arg2    ;get value to be put in BC
        mov     b,h     ;and put it there
        mov     c,l
        lxi     h,retadd        ;where call to bios will return to
        xthl            ;get address of vector in HL
        pchl            ;and go to it...
retadd: mov     l,a     ;all done. now put return value in HL
        mvi     h,0
        pop     b
        ret             ;and return to caller
        ENDFUNC

        FUNCTION        codend
        lhld    codend
        ret
        ENDFUNC

        FUNCTION        externs
        lhld    extrns
        ret
        ENDFUNC

        FUNCTION        endext
        lhld    freram
        ret
        ENDFUNC

        FUNCTION        topofmem
        lhld    base+6
        lda     tpa     ;check for "NOBOOT" hackery
        cpi     21h     ; "lxi h" at start of C.CCC (as inserted by NOBOOT)?
        dcx     h       ;if CCC doesn't begin with "lxi h," then top of
        rnz             ;memory is just below the base of the bdos
        lxi     d,-2100 ;else subtract CCP size (plus little more for good
        dad     d       ;measure) and return that as top of memory.
        ret
        ENDFUNC

        FUNCTION        exec
        EXTERNAL        execl
        call maltoh     ;get filename
        lxi d,0         ;load null parameter in DE
        push d          ;push null parameter
        push h          ;push filename
        call execl      ;do an execl
        pop d           ;clean up stack
        pop d
        ret
        ENDFUNC

        FUNCTION        execl
        call    arghak
        push    b
        lhld    arg1
        xchg
        lxi     h,-60   ;compute &nfcb for use here
        dad     sp
        push    h       ; save for much later (will pop into BC)
```

```
        push    h       ;make a few copies for local use below
        push    h
        call    setfcb  ;set up COM file for execl-ing
        pop     h       ;set new fcb addr
        lxi     b,9     ;set extension to COM
        dad     b
        mvi     m,'C'
        inx     h
        mvi     m,'O'
        inx     h
        mvi     m,'M'
        pop     d       ;set new fcb addr again
        mvi     c,openc ;open the file for reading
        call    bdos
        cpi     errorv
 JRNZ   jnz     noerrr
err:    pop     h
        pop     b
        jmp     error

noerrr: lhld    arg2    ;any first parameter?
        mov     a,h
        ora     l
  JRNZ  jnz     excl0
        lxi     d,arg2  ;no...null out first default fcb slot
        push    d
        lxi     h,fcb
        call    setfcb
        pop     h
   JR   jmp     excl0a  ;and so null out 2nd fcb slot

excl0:  xchg            ;yes.. place into first default fcb slot
        lxi     h,fcb
        call    setfcb
        lhld    arg3    ;any second parameter given?
        mov     a,h
        ora     l
  JRNZ  jnz     excl0a
        lxi     h,arg3

excl0a: xchg            ;yes: stick it into second default fcb slot
        lxi     h,fcb+16
        call    setfcb
        lxi     d,tbuff+1   ;now construct command line:
        xra     a       ;  zero tbuff+1 just in case there
        stax    d       ;  are no arg strings
        lxi     h,8     ;set pointer to 1st arg string in HL
        dad     sp      ;   by offsetting 4 objects from the current SP
        mvi     b,0     ;char count for com. line buf.
excl1:  push    h       ;and construct command line
        mov     a,m     ;set addr of next arg string pointer
        inx     h
        mov     h,m
        mov     l,a     ;0000 indicates end of list.
        ora     h       ;end of list?
   JRZ  jz      excl3

        mvi     a,' '   ;no. install next string
        dcx     h
excl2:  call    mpuc    ;convert to upper case for command line buffer
```

```
        stax    d
        inx     d
        inr     b
        inx     h
        mov     a,m
        ora     a       ;end of string?
        jnz     excl2
        pop     h       ;yes.
        inx     h       ;bump param pointer
        inx     h
        jmp     excl1   ;and so do next string

excl3:  pop     h       ;clean up stack
        mov     a,b     ;check for command buffer overflow
        cpi     53h
        jc      excl30  ;if no overflow, so load file
        lxi     d,errmsg
        mvi     c,9     ;else complain and abort...
        call    bdos
        jmp     err

errmsg: db      7,'EXECL: Too much text',cr,lf,'$'

excl30: lxi     h,tbuff ;set length of command line
        mov     m,b     ;at location tbuff

excl3a: lxi     d,code0 ;copy loader down to end of tbuff
        lxi     h,tpa-42
        mvi     b,42    ;length of loader
excl4:  ldax    d
        mov     m,a
        inx     d
        inx     h
        dcr     b
        jnz     excl4

        pop     b       ;set fcb pointer in BC
                        ;reset the SP:
        lhld    base+6  ;set BDOS pointer in HL
        lda     tpa     ;look at first op byte of run-time pkg
        cpi     31h     ;begin with "lxi sp,"?
        jnz     go0     ;if so, use the same value now...
        lhld    tpa+1   ;else get special SP value
        jmp     go1

go0:    cpi     21h     ;begin with "lxi h" (the NOBOOT sequence?)
        jnz     go1     ;if not, just use the BDOS addr as top of memory
        lxi     d,-2050 ;for NOBOOT, subtract 2100 from BDOS addr
        dad     d       ;and make that the new SP
go1:    sphl

        lxi     h,base
        push    h       ;set base of ram as return addr
        jmp     tpa-42  ;(go to `code0:')

mpuc:   cpi     61h     ;convert character in A to upper case
        rc
        cpi     7bh
        rnc
        sui     32
```

```
        ret

;
; This loader code is now: 42 bytes long.
;

code0:  lxi     d,tpa   ;destination address of new program
code1:  push    d       ;push   dma addr
        push    b       ;push   fcb pointer
        mvi     c,sdma  ;set DMA address for new sector
        call    bdos
        pop     d       ;get pointer to working fcb in DE
        push    d       ;and re-push    it
        mvi     c,reads ;read a sector
        call    bdos
        pop     b       ;restore fcb pointer into BC
        pop     d       ;and dma address into DE
        ora     a       ;end of file?
        jz      tpa-8   ;if not, get next sector (goto `code2:')
        mvi     c,sdma  ;reset DMA pointer
        lxi     d,tbuff
        call    bdos
        jmp     tpa     ;and go invoke the program

code2:  lxi     h,80h   ;bump dma address
        dad d
        xchg
        jmp     tpa-39  ;and go loop (at code1)
        ENDFUNC

        FUNCTION        execv
        EXTERNAL        execl
        call    argshak
        lhld    arg2    ;get -> arg list
        mvi     b,0     ;clear arg count
execv1: inr     b       ;bump arg count
        mov     e,m
        inx     h
        mov     d,m
        inx     h
        mov     a,d
        ora     e       ;last arg?
        jnz     execv1  ;if not, keep looking for last one

        mov     a,b     ;save arg count in case of error
        sta     savcnt

        dcx     h       ;HL -> next to last arg
execv2: mov     d,m     ;now push args on stack
        dcx     h
        mov     e,m
        dcx     h
        dcr     b
        push    d
        jnz     execv2

execv3: lhld    arg1    ;get program name
        push    h       ;save as first arg to execl
        call    execl   ;go do it; shouldn't come back.
        lda     savcnt  ;woops, we're back. Must've been an error...
```

```
        add     a
        mov     l,a     ;put size of passed parameter list
        mvi     h,0     ;into HL, and adjust stack
        dad     sp
        sphl
        lxi     h,-1    ;return error value
        ret

savcnt: ds      1       ;save arg count here
        ENDFUNC



        FUNCTION        sbrk
        call    maltoh  ;get # of bytes needed in HL
        xchg            ;put into DE
        lhld    allocp  ;get current allocation pointer
        push    h       ;save it
        dad     d       ;get tentative last address of new segment
        jc      brkerr  ;better not allow it to go over the top!
        dcx     h
        xchg            ; now last addr is in DE
        lhld    alocmx  ;get safety factor
        mov     a,h     ;negate
        cma
        mov     h,a
        mov     a,l
        cma
        mov     l,a
        inx     h
        dad     sp      ;get HL = (SP - alocmx)

        call    cmpdh   ;is DE less than HL?
        jnc     brkerr  ;if not, can't provide the needed memory.
        xchg            ;else OK.
        inx     h
        shld    allocp  ;save start of next area to be allocated
        pop     h       ;get pointer to this area
        ret             ;and return with it.

brkerr: pop     h       ;clean up stack
        jmp     error   ;and return with -1 to indicate can't allocate.

cmpdh:  mov     a,d
        cmp     h
        rc
        rnz
        mov     a,e
        cmp     l
        ret
        ENDFUNC

        FUNCTION        rsvstk
        call    maltoh  ;get the value to reserve
        shld    alocmx  ;and set new safety factor
        ret
        ENDFUNC
```

```
;
; BD Software C Standard Library Machine Language Functions
; Written by Leor Zolman
; v1.46,        3/22/82
;
; This file is in "CSM" format: to convert to CRL format,
; use CASM.SUB in conjunction with CASM.COM, ASM.COM and DDT.COM.
;
; Functions appearing in this file:
;
;       rread   rwrite  rtell   rseek   rsrec   rcfsiz
;       setjmp  longjmp
;       setplot clrplot line    plot    txtplot
;       index   setline
;
;
;
; The random-record file I/O function contained here are NOT documented
; in the User's Guide, because they are non-portable to pre-2.0 CP/M
; Systems.
;

        maclib  bds

;
; Here are the new random-access file I/O routines
; for use with CP/M version 2.x ONLY...these functions
; will NOT work under pre-2.x CP/M's.
;
; The new functions are: rread, rwrite, rtell, rseek,
;                        rsrec, rcfsiz
;



;
; Rread:
;
; Read a number of sectors randomly.
; Usage:
;
;       i = rread(fd, buf, n);
;
; The return value is either the number of sectors successfully
; read, 0 for EOF, or 1000 + (BDOS ERROR CODE)
;
; The Random Record Field is incremented following each successful
; sector is read, just as if the normal (sequential) read function
; were being used. Rseek must be used to go back to a previous
; sector.
;

        FUNCTION rread

        call    argchak
        lda     arg1
        call    fgfd
        jc      error
        mov     a,m
        ani     2
        jz      error
```

```
        push    b
        lda     arg1
        call    fgfcb
        shld    tmp2
        lxi     h,0
        shld    tmp2a
r2:     lhld    arg3
        mov     a,h
        ora     l
        lhld    tmp2a
        jnz     r2a
        pop     b
        ret

r2a:    lhld    arg2
        xchg
        mvi     c,sdma
        call    bdos
        lhld    tmp2
        xchg
        mvi     c,readr ;code for BDOS random read
        push    d       ;save de so we can fudge nr field if
        call    bdos    ;we stop reading on extent boundary...
        pop     d       ; CP/M sucks!
        ora     a
        jz      r4      ;go to r4 if no problem
        cpi     1
        jz      r2b     ;EOF?
        mov     c,a     ;put return error code in BC
        mvi     b,0
        lxi     h,1000  ;add to 1000
        dad     b
        pop     b
        ret

r2b:    lxi     h,32    ;yes. are we on extent boundary?
        dad     d
        mov     a,m
        cpi     80h
        jnz     r3
        mvi     m,0     ;yes. reset nr to 0...CP/M leaves it at 80!
r3:     lhld    tmp2a   ;(note: the above "bug" in CP/M was supposedly fixed
        pop     b       ; for 2.x, but one can never be sure...)
        ret

r4:     lhld    arg3
        dcx     h
        shld    arg3
        lhld    arg2
        lxi     d,128
        dad     d
        shld    arg2
        lhld    tmp2a
        inx     h
        shld    tmp2a
        lhld    tmp2    ;get address of fcb
        lxi     b,33    ;get addr of random record field
        dad     b
        mov     c,m     ;bump
        inx     h       ;    value
```

```
        mov     b,m     ;         of
        inx     b       ;           random
        mov     m,b     ;                 field
        dcx     h       ;                       by one
        mov     m,c
        jmp     r2
        ENDFUNC

;
; Rwrite:
;
; The random "write" routine, which always copies the sector
; to be written down to tbuff before writing. Returns
; the # of sectors successfully written, or -1 on hard error.
; (the "1000 + error code" business is not used for rwrite)
;

        FUNCTION rwrite

        call    arghak
        lda     arg1
        call    fgfd
        jc      error
        mov     a,m
        ani     4
        jz      error
        push    b
        lda     arg1
        call    fgfcb
        shld    tmp2
        lxi     h,0
        shld    tmp2a
        lxi     d,tbuff ;80 for normal CP/M, else 4280
        mvi     c,sdma
        call    bdos

nwr2:   lhld    arg3    ;done yet?
        mov     a,h
        ora     l
        lhld    tmp2a   ;if so, return count
        jnz     nwr2a
        pop     b
        ret

nwr2a:  lhld    arg2    ;else copy next 128 bytes down to tbuff
        lxi     d,tbuff ;80 for normal CP/M, else 4280
        mvi     b,128
nwr3:   mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jnz     nwr3
        shld    arg2    ;save -> to next 128 bytes
        lhld    tmp2    ;get addr of fcb
        xchg
        mvi     c,writr ;go write randomly
        call    bdos
        ora     a       ;error?
        lhld    tmp2a   ;if so, return # of successfully written
```

```
        pop     b       ; sectors.
        rnz
        push    b

        inx     h       ; else bump successful sector count,
        shld    tmp2a
        lhld    arg3    ; debump countdown,
        dcx     h
        shld    arg3
        lhld    tmp2    ; get address of fcb
        lxi     b,33    ; get address of random field
        dad     b
        mov     c,m     ; bump 16-bit value at random
        inx     h       ; record
        mov     b,m     ;       field
        inx     b       ;           of
        mov     m,b     ;             fcb
        dcx     h       ;                by one
        mov     m,c
        jmp     nwr2    ; and go try next sector
        ENDFUNC

;
; rseek:
;
; rseek(fd, offset, origin)
;         seeks to offset records if origin == 0,
;     to present position + offset if origin == 1,
;       or to end of file + offset if origin == 2.
; (note that in the last case, the offset must be non-positive)
;

        FUNCTION rseek

        call    argshak
        lda     arg1
        call    fgfcb
        jc      error
        push    h
        call    rtell2
        lhld    arg2
        lda     arg3    ;is origin == 0?
        ora     a
        jz      rseek2  ;if so, HL holds new position
        dcr     a       ;no. is origin == 1?
        jnz     rseek1
        dad     d       ;yes. add offset to current position
        jmp     rseek2  ;and result is in HL

rseek1: pop     d       ;else origin must be 2...
        push    d
        push    b
        mvi     c,cfsize ;compute end of file position
        call    bdos
        pop     b
        pop     h       ;set back fcb
        push    h
        call    rtell2  ;set DE = position
        lhld    arg2    ;add offset
        dad d           ;and HL holds new position
```

```
rseek2: xthl            ;get fcb, push  new position
        lxi     d,33
        dad     d       ;HL points to random field of fcb
        pop     d       ;get new position in DE
        mov     m,e     ;and put into fcb
        inx     h
        mov     m,d
        xchg            ;and return the position value
        ret

rtell2: lxi     d,33
        dad     d
        mov     e,m
        inx     h
        mov     d,m
        ret
        ENDFUNC

;
; Rtell:
;
; Return random record position of file:
;

        FUNCTION rtell
        call    arghak
        lda     arg1
        call    fgfcb
        jc      error
        lxi     d,33    ;go to random record field
        dad     d
        mov     e,m     ;get value into DE
        inx     h
        mov     d,m
        xchg            ;put into HL
        ret
        ENDFUNC

;
; Rsrec:
;
; Set random field from serial access mode:
;

        FUNCTION rsrec
        call    arghak
        lda     arg1
        call    fgfcb
        jc      error
        push    h
        xchg
        push    b
        mvi     c,srrecc
        call    bdos
        pop     b
        pop     h
        lxi     d,33
        dad     d
        mov     a,m
        inx     h
```

```
        mov     h,m
        mov     l,a
        ret
        ENDFUNC

;
; Rcfsiz:
;
; set random record field to end-of-file:
;

        FUNCTION        rcfsiz
        call    argchk
        lda     arg1
        call    getfcb
        jc      error
        push    h
        xchg
        push    b
        mvi     c,cfsizc
        call    bdos
        pop     b
        pop     h
        lxi     d,33
        dad     d
        mov     a,m
        inx     h
        mov     h,m
        mov     l,a
        ret
        ENDFUNC

        FUNCTION        setjmp
        call    maltoh
        mov     m,c     ;save BC
        inx     h
        mov     m,b
        inx     h
        xchg
        lxi     h,0
        dad     sp
        xchg
        mov     m,e     ;save SP
        inx     h
        mov     m,d
        inx     h
        pop     d       ;save return address
        push    d
        mov     m,e
        inx     h
        mov     m,d
        lxi     h,0     ;and return 0
        ret
        ENDFUNC

        FUNCTION        longjmp
        call    maltoh  ;get buffer address
        mov     c,m     ;restore BC
        inx     h
        mov     b,m
```

```
        inx     h
        mov     e,m     ;restore SP...first put it in DE
        inx     h
        mov     d,m
        inx     h
        shld    temp    ;save pointer to return address
        call    ma2toh  ;get return value
        xchg            ;put return val in DE, old SP in HL
        sphl            ;restore SP with old value
        pop     h       ;pop retur address off stack
        lhld    temp    ;get back ptr to return address
        mov     a,m
        inx     h
        mov     h,m
        mov     l,a     ;HL holds return address
        xchg            ;put ret addr in DE, get return value in HL
        push    d       ;push return address on stack
        ret             ;and return...
temp:   ds 2
        ENDFUNC


        FUNCTION        setplot
        call    arghak
        push    b
        lhld    arg1    ;get base address
        shld    pbase   ;       initialize
        lhld    arg3    ;get y size
        shld    ysize   ;       initialize
        xchg            ;leave it in DE
        lhld    arg2    ;get x size
        shld    xsize   ;       initialize
        call    usmul   ;figure out screen size
        shld    psize   ;       initialize
        pop     b
        ret
        ENDFUNC

        FUNCTION        clrplot
        lhld    psize   ;put screen size
        xchg            ;       in DE
        lhld    pbase   ;get screen base in HL
clr2:   mvi     m,' '   ;and
        inx     h       ;   clear
        dcx     d       ;       each
        mov     a,d     ;         location
        ora     e       ;               (all DE of 'em)
        jnz     clr2
        ret
        ENDFUNC

        FUNCTION        line
        call    arghak  ;get args
        push    b
        lda     arg2    ;put one set of endpoint data in DE in
        mov     c,a     ;format:  D = x = arg2, E = y = arg3
        lda     arg3
        mov     b,a
        mov     d,b
        mov     e,c
```

```
        call    put     ; put up one endpoint at BC
        lda     arg4    ;put other endpoint data in HL
        mov     c,a
        lda     arg5
        mov     b,a
        call    put     ;(but first put up the point from BC)
        mov     h,b
        mov     l,c
        call    liner   ;now connect them...
        pop     b
        ret             ;all done.

liner:  mov     a,d
        sub     h
        call    abs
        cpi     2
        jnc     line2   ;are points far enough apart
                        ;in both dimensions to warrant
        mov     a,e     ;drawing a line?
        sub     l
        call    abs
        cpi     2
        jnc     line2
        ret             ;if not, return.

line2:  call    midp    ;find midpoint
        call    put     ;put it up
        push    d       ;set up recursive calls
        mov     d,b
        mov     e,c
        call    liner
        xthl
        call    liner
        xchg
        pop     h
        ret             ;and we are done!

midp:   push    h
        push    d

        mov     a,h
        sub     d
        ani     1
        jz      mid3

        mov     a,h
        cmp     d
        jc      mid2a
        inr     d
        jmp     mid3

mid2a:  dcr     h

mid3:   mov     a,l
        sub     e
        ani     1
        jz      mid4

        mov     a,l
        cmp     e
```

```
        jc      mid3a
        inr     e
        jmp     mid4

mid3a:  dcr     l

mid4:   mov     a,h
        add     d
        ora     a
        rrc
        mov     b,a
        mov     a,l
        add     e
        ora     a
        rrc
        mov     c,a
        pop     d
        pop     h
        ret

put:    push    h
        push    d
        mov     a,b
        lhld    ysize
        xchg
        lhld    pbase
        inr     a
put1:   dcr     a
        jz       put2
        dad     d
        jmp     put1

put2:   mov     e,c
        mvi     d,0
        dad     d
        lda     arg1
        mov     m,a
        pop     d
        pop     h
        ret

abs:    ora     a
        rp
        cma
        inr     a
        ret
        ENDFUNC

        FUNCTION        plot
        call    arghak
        lda     arg1
        lhld    ysize
        xchg
        lhld    pbase
        inr     a
plot1:  dcr     a
        jz      plotc
        dad     d
        jmp     plot1
```

```
plotc:  lda     arg2
        mov     e,a
        mvi     d,0
        dad     d
        lda     arg3
        mov     m,a
        ret
        ENDFUNC

        FUNCTION        txtplot
        call    argshak
        push    b
        lhld    arg2
        xchg
        lhld    ysize
        call    usmul
        xchg
        lhld    arg3
        dad     d
        xchg
        lhld    pbase
        dad     d
        xchg
        lhld    arg1
        mvi     b,0
        lda     arg4
        ora     a
        jz      txt2
        mvi     b,80h
txt2:   mov     a,m
        ora     a
        jnz     txt3
        pop     b
        ret

txt3:   ora     b
        stax    d
        inx     h
        inx     d
        jmp     txt2
        ENDFUNC

;
; Index(str,substr)
; char *str, *substr;
;
; Returns index of substr in str, or -1 if not found.
;

        FUNCTION        index
        call    argshak
        lhld    arg1
        xchg            ;main str ptr in DE
        lhld    arg2    ;substr ptr in HL
        dcx     d
index1: inx     d
        ldax    d       ;end of str?
        ora     a
        jnz     index2
        lxi     h,-1    ;yes, not found.
```

```
        ret
index2: cmp     m       ;quick check for dissimilarity
        jnz     index1  ;loop if not same right here
        push    d       ;else do long compare
        push    h
index3: inx     h
        inx     d
        mov     a,m     ;end of substr?
        ora     a
        jnz     index4  ;if not, go on testing
        pop     d       ;else matches
        pop     d       ;get starting address of substr in DE
        lhld    arg1    ;subtract beginning of str
        call    cmh
        dad     d       ;and return the result
        ret

index4: ldax    d       ;current char match?
        cmp     m
        jz      index3  ;if so, keep testing
        pop     h       ;else go on to next char in str
        pop     d
        jmp     index1
        ENDFUNC

;
; Getline(str,lim)
; char *str;
;
; Gets a line of text from the console, up to 'lim' characters.
;

        FUNCTION        getline
        push    b
        call    ma3toh  ;get max no. of chars
        mov     c,a     ;save in C
        call    ma2toh  ;get destination address
        push    h
        lxi     h,-150  ;use space below stack for reading line
        dad     sp
        push    h       ;save buffer address
        mov     m,c     ;Set max # of characters
        mvi     c,getlin
        xchg            ;put buffer addr in DE
        call    bdos    ;get the input line
        mvi     c,conout
        mvi     e,lf    ;put out a LF
        call    bdos
        pop     h       ;get back buffer address
        inx     h       ;point to returned char count
        mov     b,m     ;set B equal to char count
        inx     h       ;HL points to first char of line
        pop     d       ;DE points to start destination area
        mov     c,b     ;save char count in C
copy1:  mov     a,b     ;copy line to start of buffer
        ora     a
        jz      gets2
        mov     a,m
        stax    d
        inx     h
```

```
        inx     d
        dcr     b
        jmp     copy1

gets2:  xra     a       ;store terminating null
        stax    d
        mov     l,c     ;return char count in HL
        mvi     h,0
        pop     b
        ret
        ENDFUNC
```

```
/*  LCHECK by Richard Conn

        LCHECK displays to the user the nesting level number of each
BEGIN/END ({/}) group, thereby helping him to identify problem areas
in his C programs.  It recognizes quoted material and comments and
ignores { and } within these.

*/

#define vers    12      /* Version Number */

#include        "a:bdscio.h"

#define SSCROLL TRUE    /* Set TRUE for Smooth Scrolling on TVI 950 */

#define squote  0x27    /* Single Quote */
#define dquote  0x22    /* Double Quote */
#define BS      0x08    /* Back Space Char */
#define TAB     0x09    /* Tab Char */
#define LF      0x0a    /* Line Feed Char */
#define CR      0x0d    /* Carriage Return Char */
#define YES     'Y'
#define NO      'N'
#define ovfl    YES     /* Line Overflow */
#define noovfl  NO      /* No Line Overflow */

char    iobuf[BUFSIZ];
int     level, chval, pos, nroutines;

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

        if (argc == 1) {
                printf("LCHECK, Version %d.%d\n",vers/10,vers%10);
                printf("Format of Command Line is --\n");
                printf("  LCHECK filename.typ");
                exit(FALSE);
                }
        if (fopen(argv[1],iobuf) == ERROR) {
                printf("Cannot Find File %s\n",argv[1]);
                exit(FALSE);
                }
        if (SSCROLL) printf("%c%c",ESC,'8');  /* Smooth Scroll */
        printf("LCHECK, Version %d.%d -- File:  %s\n",vers/10,
          vers%10,argv[1]);
        level = 0; nroutines = 0;  /* Init nesting level, routine count */
        prlevel();  /* Print level number */
        do {
                getit();  /* Get next char */
                if (chval == squote) do {  /* If squote, flush to end squote */
                                getit();
                        } while (chval != squote);
                if (chval == dquote) do {  /* If dquote, flush to dquote */
                                getit();
                        } while (chval != dquote);
                if (chval == '/') {  /* Possible comment */
                        getit();
```

```
                        if (chval == '*') {  /* Yes, it is a comment */
                                getit();
                                done = FALSE;
                                do {
                                        if (chval == '*') {  /* End comment? */
                                                getit();
                                                if (chval == '/')  /* Yes */
                                                        done = TRUE;
                                                }
                                        else getit();
                                } while (!done);
                                }
                        }
                if (chval == '{') level++;  /* BEGIN */
                if (chval == '}') {  /* END */
                        level--;
                        if (level == 0) {
                                nroutines++;
                                printf("\n** Routine %d **", nroutines);
                                }
                        }
        } while ((chval != CPMEOF) && (chval != ERROR));

        printf("\nProgram Level Check is ");
        if (level == 0) printf("OK");
                else printf("NOT OK");
        printf("\nNumber of Routines Encountered: %d",--nroutines);
        if (SSCROLL) printf("%c%c",ESC,'9');  /* Hard Scroll */
}

getit()  /* Get and Echo Character */
{
        chval = getc(iobuf);
        if ((pos >= TWIDTH) & (chval != CR)) prlevel(ovfl);
        if (chval != CPMEOF) echo(chval);
}

echo(chval)  /* Echo Char with tabulation */
char chval;
{
        switch (chval) {
                case TAB : putchar(' '); pos++;
                           while (pos%9 != 0) {
                                putchar(' ');
                                pos++;
                                }
                           break;
                case BS  : putchar(BS);
                           pos--;
                           break;
                case LF  : prlevel(noovfl);
                           break;
                case CR  : putchar(CR);
                           pos = 0;
                           break;
                default  : if (chval >= ' ') {
                                putchar(chval);
                                pos++;
                                }
                           break;
```

```
            }
}

prlevel(ovfl_flag)  /* Print Level Number and Set Col Count */
char ovfl_flag;
{
        putchar(LF);
        if (level < 10) printf(" %d",level);
                else printf("%d",level);
        if (ovfl_flag == YES) putchar('-');
                else putchar(':');
        putchar(' ');
        pos = 5;
}
```

```
;The segment is now moved to high memory, but not
;properly relocated.  The bit table which specifies
;which addresses need to be adjusted is located
;just after the last byte of the source segment,
;so (HL) is now pointing at it.
        POP     D       ;beginning of newly moved code.
        LXI     B,SEGLEN;length of segment
        PUSH    H       ;save pointer to reloc info
        MOV     H,D     ;offset page address
;
FIXLOOP:
;Scan through the newly moved code, and adjust any
;page addresses by adding (H) to them.  The word on
;top of the stack points to the next byte of the
;relocation bit table.  Each bit in the table
;corresponds to one byte in the destination code.
;A value of 1 indicates the byte is to be adjusted.
;A value of 0 indicates the byte is to be unchanged.
;
;Thus one byte of relocation information serves to
;mark 8 bytes of object code.  The bits which have
;not been used yet are saved in L until all 8
;are used.
;
        MOV     A,B
        ORA     C               ;test if finished
        JZ      FIXDONE
        DCX     B               ;count down
        MOV     A,E
        ANI     07H             ;on 8-byte boundry?
        JNZ     NEXTBIT
;
NEXTBYT:
;Get another byte of relocation bits
        XTHL
        MOV     A,M
        INX     H
        XTHL
        MOV     L,A             ;save in register L
;
NEXTBIT MOV     A,L             ;remaining bits from L
        RAL                     ;next bit to CARRY
        MOV     L,A             ;save the rest
        JNC     NEXTADR
;
;CARRY was = 1.  Fix this byte.
        LDAX    D
        ADD     H               ;(H) is the page offset
        STAX    D
;
NEXTADR INX     D
        JMP     FIXLOOP
;
FIXDONE:
;Finished.  Jump to the first address in the new
;segment in high memory.
;
;First adjust the stack.  One garbage word was
;left by fixloop.
        INX     SP
```

```
        INX     SP
;
;(HL) still has the page address
        MOV     L,A     ;move zero to l
        PCHL            ;Stack is valid
SETUP:
;Any one-shot initialization code goes here.
;
        LXI     H,NOLOAD
        SHLD    CCPIN+1         ;Prevent reentry
;
        CPM     VER             ;Test version of CP/M in use
        CPI     20H             ;2.0 or better?
        JC      BADVER          ;No, bitch and quit.
;
        CALL    REPARS          ;Re-parse command line
;
        LXI     D,MEMBER+9      ;Check member filetype
        LDAX    D
        CPI     ' '             ;If blank,
        BLKMOV  ,COMLIT,3,Z     ; default to COM.
;
        LXI     D,LBRFIL+9      ;Check library filetype
        LDAX    D
        CPI     ' '             ;If blank,
        BLKMOV  ,LBRLIT,3,Z     ; default to LBR
;
        LXI     D,LBRFIL+1      ;Check name
        LDAX    D
        CPI     ' '             ;If blank,
        BLKMOV  ,DFLTNAM,8,Z    ; use default name.
;
;
DIROPN: CPM     OPN,LBRFIL      ;Open for directory read.
        INR     A               ;Was it found?
        JNZ     DIROK           ;Yes, ok
        LXI     H,LBRFIL        ;No, test drive spec
        MOV     A,M             ; to see if it's
        ORA     A               ; explicit
        JNZ     NODIR           ;It is explicit.  Out of luck
        INR     M               ;It was defaulted. Look on A:
        JMP     DIROPN          ; before giving up.
;
DIROK:
        CPM     DMA,TBUFF
FINDMBR:
        CPM     FRD,LBRFIL      ;Read the directory
        ORA     A
        JNZ     FISHY           ;Empty file, Give up.
        LXI     H,TBUFF
        MOV     A,M
        ORA     A
        JNZ     FISHY           ;Directory not active??
        MVI     B,8+3           ;Check for blanks
        MVI     A,' '
VALIDLOOP:
        INX     H
        CMP     M
        JNZ     FISHY
        DCR     B
```

```
        JNZ     VALIDLOOP
;
        LHLD    TBUFF+1+8+3     ;Index must be 0000
        MOV     A,H
        ORA     L
        JNZ     FISHY
;
        LHLD    TBUFF+1+8+3+2   ;Get directory size
        DCX     H               ;We already read one.
        PUSH    H               ;Save on stack
        JMP     FINDMBRN        ;Jump into loop
FINDMBRL:
        POP     H               ;Read sector count from TOS
        MOV     A,H
        ORA     L               ;0 ?
        JZ      NOMEMB          ;Member not found in library
        DCX     H               ;Count down
        PUSH    H               ;and put it back.
        CPM     FRD,LBRFIL      ;Get next directory sector
        ORA     A
        JNZ     FISHY


FINDMBRN:
        LXI     H,TBUFF         ;Point to buffer.
        MVI     C,128/32        ;Number of directory entries
;
FINDMBR1:
        CALL    COMPARE         ;Check if found yet.
        JZ      GETLOC          ;Found member in .DIR
        DCR     C
        JZ      FINDMBRL
;
        LXI     D,32            ;No match, point to next one.
        DAD     D
        JMP     FINDMBR1
;
GETLOC:         ;The name was found now get index and length
        POP     B       ;Clear stack garbage
        XCHG            ;Pointer to sector address.
        MOV     E,M     ;Get First
        INX     H
        MOV     D,M
        XCHG
        SHLD    INDEX   ;Save it
        XCHG
        INX     H       ;Get Size to DE
        MOV     E,M
        INX     H
        MOV     D,M
        XCHG            ; Size to HL
        SHLD    LENX
        CALL    PACKUP  ;Repack command line arguments
        CPM     CON,CR  ;do <cr> only (look like CCP)
        RET
;               End of setup.
;
;       Utility subroutines
NEGDE:                  ;DE = -DE
        MOV     A,D
```

```
        CMA
        MOV     D,A
;
        MOV     A,E
        CMA
        MOV     E,A
        INX     D
        RET
;
;       REPARSE re-parses the fcbs from the command line,
;       to allow the "-" character to prefix the library name
;
REPARS: LXI     D,MEMBER        ;first reinitialize both fcbs
        CALL    NITF
        LXI     D,LBRFIL
        CALL    NITF
        LXI     H,TBUFF         ;store a null at the end of
        MOV     E,M             ; the command line (this is
        MVI     D,0             ; done by CP/M usually, except
        XCHG                    ; in the case of a full com-
        DAD     D               ; mand line
        INX     H
        MVI     M,0
        XCHG                    ;tbuff pointer back in hl
SCANBK: INX     H               ;bump to next char position
        MOV     A,M             ;fetch next char
        ORA     A               ;reached a null? (no arguments)
        JZ      HELP            ;interpret as a call for help
        CPI     ' '             ;not null, skip blanks
        JZ      SCANBK
        CPI     '-'             ;library name specifier?
        JNZ     NOTLBR          ;skip if not
        INX     H               ;it is, skip over flag character
        LXI     D,LBRFIL        ;parse library name into FCB
        CALL    GETFN
NOTLBR: LXI     D,MEMBER        ;now parse the command name
        CALL    GETFN
        LXI     D,HOLD+1        ;ptr to temp storage for rest of cmd line
        MVI     B,-1            ;init a counter
CLSAVE: INR     B               ;bump up counter
        MOV     A,M             ;fetch a char
        STAX    D               ;move it to hold area
        INX     H               ;bump pointers
        INX     D
        ORA     A               ;test whether char was a terminator
        JNZ     CLSAVE          ;continue moving line if not
        MOV     A,B             ;it was, get count
        STA     HOLD            ;save it in hold area
        RET
;
;       PACKUP retrieves the command line stored at
;       HOLD and moves it back to tbuff, then reparses
;       the default file control blocks so the command
;       will never know it was run from a library
;
PACKUP: LXI     H,HOLD          ;point to length byte of HOLD
        MOV     C,M             ;get length in BC
        MVI     B,0
        INX     B               ;bump up to because length byte doesn't
        INX     B               ;  include itself or null terminator
```

```
        BLKMOV  TBUFF           ;moving everybody to Tbuff
        LXI     H,TBUFF+1       ;point to the command tail
        LXI     D,TFCB1         ;first parse out tfcb1
        CALL    GETFN
        LXI     D,TFCB2         ;then tfcb2
        CALL    GETFN
        RET
;
;       Here when HELP is requested (indicated
;       by LRUN with no arguments)
;
HELP:   CPM     MSG,HLPMSG      ;print the HELP message
EXIT:   LHLD    SPSAVE          ;find CCP re-entry adrs
        SPHL                    ;fix & return
        RET
;
;       the HELP message
;
HLPMSG: DB      CR,LF,'Correct syntax is:'
        DB      CR,LF
        DB      LF,TAB,'LRUN [-<lbrname>] <command line>'
        DB      CR,LF
        DB      LF,'Where <lbrname> is the optional library name'
        DB      CR,LF,'(Note the preceding "-".  ) If omitted,'
        DB      CR,LF,'the default command library is used.'
        DB      LF
        DB      CR,LF,'<command line> is the name and parameters'
        DB      CR,LF,'of the command being run from the library,'
        DB      CR,LF,'just as if a separate .COM file were being run.'
        DB      CR,LF,'$'
;
;
COMPARE:                ;Test status, name and type of
        PUSH    H               ;a directory entry.
        MVI     B,1+8+3
        XCHG                    ;with the one we're
        LXI     H,MEMBER        ;looking for.
COMPAR1:
        LDAX    D
        CMP     M
        JNZ     COMPEXIT
        INX     D
        INX     H
        DCR     B
        JNZ     COMPAR1
COMPEXIT:                       ;Return with DE pointing to
        POP     H               ;last match + 1, and HL still
        RET                     ;pointing to beginning.
;
;
;       File name parsing subroutines
;
; setfn sets a file name from text pointed to by reg hl into
; an fcb pointed to by reg de.  leading delimeters are
; ignored.
; entry hl      first character to be scanned
;       de      first byte of fcb
; exit  hl      character following file name
;
;
```

```
;
GETFN:  CALL    NITF    ;init 1st half of fcb
        CALL    GSTART  ;scan to first character of name
        RZ              ;end of line was found - leave fcb blank
        CALL    GETDRV  ;get drive spec. if present
        CALL    GETPS   ;set primary and secondary name
        RET


;
; nitf fills the fcb with dflt info - 0 in drive field
; all-blank in name field, and 0 in ex,s1,s2 and rc flds
;
NITF:   PUSH    D       ;save fcb loc
        XCHG            ;move it to hl
        MVI     M,0     ;zap dr field
        INX     H       ;bump to name field
        MVI     B,11    ;zap all of name fld
NITLP1: MVI     M,' '
        INX     H
        DCR     B
        JNZ     NITLP1
        MVI     B,4     ;zero others
NITLP2: MVI     M,0
        INX     H
        DCR     B
        JNZ     NITLP2
        XCHG            ;restore hl
        POP     D       ;restore fcb pointer
        RET
;
; gstart advances the text pointer (reg hl) to the first
; non delimiter character (i.e. ignores blanks).  returns a
; flag if end of line (00h or ';') is found while scaning.
; exit  hl      pointing to first non delimiter
;       a       clobbered
;       zero    set if end of line was found
;
GSTART: CALL    GETCH   ;see if pointing to delim?
        RNZ             ;nope - return
        CPI     ';'     ;end of line?
        RZ              ;yup - return w/flag
        ORA     A
        RZ              ;yup - return w/flag
        INX     H       ;nope - move over it
        JMP     GSTART  ;and try next char
;
; getdrv checks for the presence of a drive spec at the text
; pointer, and if present formats it into the fcb and
; advances the text pointer over it.
; entry hl      text pointer
;       de      pointer to first byte of fcb
; exit  hl      possibly updated text pointer
;       de      pointer to second (primary name) byte of fcb
;
GETDRV: INX     D       ;point to name if spec not found
        INX     H       ;look ahead to see if ':' present
        MOV     A,M
        DCX     H       ;put back in case not present
        CPI     ':'     ;is a drive spec present?
```

```
        RNZ             ;nope - return
        MOV     A,M     ;yup - get the ascii drive name
        SUI     'A'-1   ;convert to fcb drive spec
        DCX     D       ;point back to drive spec byte
        STAX    D       ;store spec into fcb
        INX     D       ;point back to name
        INX     H       ;skip over drive name
        INX     H       ;and over ':'
        RET
;
; getps gets the primary and secondary names into the fcb.
; entry hl      text pointer
; exit  hl      character following secondary name (if present)
;
GETPS:  MVI     C,8     ;max length of primary name
        CALL    GETNAM  ;pack primary name into fcb
        MOV     A,M     ;see if terminated by a period
        CPI     '.'
        RNZ             ;nope - secondary name not given
                        ;return default (blanks)
        INX     H       ;yup - move text pointer over period
FTPOINT:MOV     A,C     ;yup - update fcb pointer to secondary
        ORA     A
        JZ      GETFT
        INX     D
        DCR     C
        JMP     FTPOINT
GETFT:  MVI     C,3     ;max length of secondary name
        CALL    GETNAM  ;pack secondary name into fcb
        RET
;
; getnam copies a name from the text pointer into the fcb for
; a given maximum length or until a delimiter is found, which
; ever occurs first.  if more than the maximum number of
; characters is present, characters are ignored until a
; a delimiter is found.
; entry hl      first character of name to be scaned
;       de      pointer into fcb name field
;       c       maximum length
; exit  hl      pointing to terminating delimiter
;       de      next empty byte in fcb name field
;       c       max length - number of characters transfered
;
GETNAM: CALL    GETCH   ;are we pointing to a delimiter yet?
        RZ              ;if so, name is transfered
        INX     H       ;if not, move over character
        CPI     '*'     ;ambigious file reference?
        JZ      AMBIG   ;if so, fill the rest of field with '?'
        STAX    D       ;if not, just copy into name field
        INX     D       ;increment name field pointer
        DCR     C       ;if name field full?
        JNZ     GETNAM  ;nope - keep filling
        JMP     GETDEL  ;yup - ignore until delimiter
AMBIG:  MVI     A,'?'   ;fill character for wild card match
GFILL:  STAX    D       ;fill until field is full
        INX     D
        DCR     C
        JNZ     GFILL   ;fall thru to ingore rest of name
GETDEL: CALL    GETCH   ;pointing to a delimiter?
        RZ              ;yup - all done
```

```
        INX     H       ;nope - ignore antoher one
        JMP     GETDEL
;
; getch gets the character pointed to by the text pointer
; and sets the zero flag if it is a delimiter.
; entry hl      text pointer
; exit  hl      preserved
;       a       character at text pointer
;       z       set if a delimiter
;
GETCH:
        MOV     A,M     ;set the character
        CPI     '.'
        RZ
        CPI     ','
        RZ
        CPI     ';'
        RZ
        CPI     ' '
        RZ
        CPI     ':'
        RZ
        CPI     '='
        RZ
        CPI     '<'
        RZ
        CPI     '>'
        RZ
        ORA     A       ;Set zero flag on end of text
        RET
;
;
; Error routines:
;
BADVER:
        CALL    ABEND
        DB      'Can''t run under CP/M 1.4'
NODIR:
        CALL    ABEND
        DB      'Library not found'
        DB      '$'
FISHY:
        CALL    ABEND
        DB      'Name after "-" isn''t a library'
        DB      '$'
NOMEMB:
        CALL    ABEND
        DB      'Command not in directory'
        DB      '$'
NOLOAD:
        CALL    ABEND
        DB      'No program in memory'
        DB      '$'
NOFIT:
        CALL    ABEND
        DB      'Program too large to load'
        DB      '$'
;
COMLIT: DB      'COM'
;
```

```
DFLTNAM:DB      'COMMAND ' ; <---change this if you like---
LBRLIT: DB      'LBR'
;
ABEND:
        CPM     MSG,NEWLIN
        POP     D
        CPM     MSG
        CPM     DEL,SUBFILE
        CPM     MSG,ABTMSG
        JMP     EXIT
ABTMSG: DB      '...ABORTED.$'
NEWLIN: DB      CR,LF,'$'
SPSAVE: DS      2               ;stack pointer save
;
        PAGE
;Adjust location counter to next 256-byte boundry
@BASE   ORG     ($ + 0FFH) AND 0FF00H
@RLBL   SET     0
;
; The segment to be relocated goes here.
; Any position dependent (3-byte) instructions
; are handled by the "R" macro.
;************************************************
 R      <LHLD   LENX>   ;Get length of .COM member to load.
        MVI     A,TPA/128
        ADD     L       ;Calculate highest address
        MOV     L,A     ;To see if it will fit in
        ADC     H       ;available memory
        SUB     L
        MOV     H,A
        REPT    7
        DAD     H
        ENDM
        XCHG
        CALL    NEGDE   ;IT'S STILL IN LOW MEMORY
 R      <LXI    H,PROTECT>
        DAD     D
        JNC     NOFIT   ;Haven't overwritten it yet.
LBROPN:
; The library file is still open.  The open FCB has been
; moved up here into high memory with the loader code.
;
 R      <LHLD   INDEX>          ;Set up for random reads
 R      <SHLD   RANDOM>
        XRA     A
 R      <STA    RANDOM+2>
;
        LXI     H,TPA
 R      <SHLD   LOADDR>

; This high memory address and above, including CCP, must be
; protected from being overlaid by loaded program
PROTECT:
;
LOADLOOP:                       ;Load that sucker.
 R      <LHLD   LENX>           ;See if done yet.
        MOV     A,L
        ORA     H
 R      <JZ     LOADED>
        DCX     H
```

```
 R      <SHLD   LENX>
;
 R      <LHLD   LOADDR>         ;Increment for next time
        MOV     D,H
        MOV     E,L
        LXI     B,80H
        DAD     B
 R      <SHLD   LOADDR>
        CPM     DMA             ;but use old value (DE)
;
 R      <LXI    D,LBRFIL>
        CPM     RRD             ;Read the sector
        ORA     A               ;Ok?
 R      <JNZ    ERR>            ;No, bail out.
;
 R      <LHLD   RANDOM>         ;Increment random record field
        INX     H
 R      <SHLD   RANDOM>
;
 R      <JMP    LOADLOOP>       ;Until done.
;
ERR:
        MVI     A,( JMP )       ;Prevent execution of bad code
        STA     TPA
        LXI     H,BOOT
        SHLD    TPA+1
;
 R      <LXI    D,LDMSG>
        CPM     MSG
 R      <LXI    D,SUBFILE>      ;Abort SUBMIT if in progress
        CPM     DEL
LOADED:
        CPM     DMA,TBUFF       ;Restore DMA adrs for user pgm
        CPM     CON,LF          ;Turn up a new line on console
        JMP     TPA
;
LDMSG:
        DB      CR,LF,'BAD LOAD$'
INDEX   DW      0
LENX    DW      0
SUBFILE:
        DB      1,'$$$     SUB',0,0,0,0
        ;If used, this FCB will clobber the following one.
        ;but it's only used on a fatal error, anyway.
LBRFIL:
        DS      32              ;Name placed here at setup
        DB      0               ;Normal FCB plus...
OVERLAY SET     $               ;(Nothing past here but DS's)
RANDOM  DS      3               ;...Random access bytes
MAXMEM  DS      2
LOADDR  DS      2
;************************************************
;End of segment to be relocated.
        IF      OVERLAY EQ 0
OVERLAY SET     $
        ENDIF
;
PAGES   EQU     ($-@BASE+0FFH)/256+8
;
SEGLEN  EQU     OVERLAY-@BASE
```

```
        ORG     @BASE+SEGLEN
        PAGE
;       Build the relocation information into a
; bit table immediately following.
;
@X      SET     0
@BITCNT SET     0
@RLD    SET     ??R1
@NXTRLD SET     2
        RGRND   %@RLBL+1        ;define one more label
;
        REPT    SEGLEN+3
        IF      @BITCNT>@RLD
        NXTRLD  %@NXTRLD        ;next value
        ENDIF
        IF      @BITCNT=@RLD
@X      SET     @X OR 1         ;mark a bit
        ENDIF
@BITCNT SET     @BITCNT + 1
        IF      @BITCNT MOD 8 = 0
        DB      @X
@X      SET     0       ;clear hold variable for more
        ELSE
@X      SET     @X SHL 1        ;not 8 yet. move over.
        ENDIF
        ENDM
;
        DB      0
HOLD:   DB      0,0             ;0 length, null terminator
        DS      128-2           ;rest of HOLD area
MEMBER:
        DS      16
;
        END     CCPIN
```

```
/*
        Line printer formatter

        Written by Leor Zolman
                   May 28, 1980

        First prints all files named on the command line, and then
        asks for names of more files to print until a null line is typed.
        Control-Q aborts current printing and goes to next file.

        Paper should be positioned ready to print on the first page; each
        file is always printed in an even number of pages so that new files
        always start on the same phase of fan-fold paper.

        Tabs are expanded into spaces.
*/

#include "bdscio.h"

#define FF 0x0c         /* formfeed character, or zero if not supported */
#define PGLEN 66        /* lines per lineprinter page */

int colno, linesleft;

main(argc,argv)
char **argv;
{
        int i, pgno, fd;
        char date[30], linebuf[135];    /* date and line buffers */
        char fnbuf[30], *fname;         /* filename buffer & ptr */
        char ibuf[BUFSIZ];              /* buffered input buffer */
        char *gets();

        pgno = colno = 0;
        linesleft = PGLEN;
        printf("What is today's date? ");
          gets(date);

        while (1)
        {
                if (argc-1)
                 {
                        fname = *++argv;
                        argc--;
                 }
                else
                 {
                        printf("\nEnter file to print, or CR if done: ");
                        if (!*(fname = gets(fnbuf))) break;
                 }

                if ((fd = fopen(fname,ibuf)) == ERROR)
                 {
                        printf("Can't open %s\n",fname);
                        continue;
                 }
                else printf("\nPrinting %-13s",fname);

                for (pgno = 1; ; pgno++)
                 {
```

```
                putchar('*');
                sprintf(linebuf,"\n\n%28s%-13s%5s%-3d%20s\n\n\n",
                        "file: ",fname,"page ",pgno,date);
                linepr(linebuf);

        loop:   if (!fgets(linebuf,ibuf)) break;
                if (kbhit() && getchar() == 0x11) break;
                if (linepr(linebuf)) continue;
                if (linesleft > 2) goto loop;
                formfeed();
            }
            formfeed();
            if (pgno % 2) formfeed();
            fabort(fd);
        }
}

/*
        Print a line of text out on the list device, and
        return true if a formfeed was encountered in the
        text.
*/

linepr(string)
char *string;
{
        char c, ffflag;
        ffflag = 0;
        while (c = *string++)
          switch (c) {
            case FF:
                ffflag = 1;
                break;
            case '\n':
                putlpr('\r');
                putlpr('\n');
                colno = 0;
                linesleft--;
                break;

            case '\t':
                do {
                  putlpr(' ');
                  colno++;
                } while (colno % 8);
                break;

            default:
                putlpr(c);
                colno++;
        }
        if (ffflag) formfeed();
        return ffflag;
}

putlpr(c)
char c;
{
        bios(5,c);
}
```

```
formfeed()
{
        if (FF) putlpr(FF);
        else while (linesleft--) putlpr('\n');
        linesleft = PGLEN;
}
```

```
;
; BDS.LIB              for BDS C v1.45           October 14, 1980
;
; Addresses within C.CCC and the ram area to be used by machine
; language CRL functions.
;
; If you alter C.CCC by reassembling CCC.ASM, be sure to go through
; this file and make sure all the addresses are equated to the
; appropriate values resulting from the reassembly. Then the library
; functions will be ready to reassemble.
;

        page 76

CPM:    EQU 1           ;true if running under CP/M; else 0

;
; System addresses:
;

        if not cpm
CCCORG: EQU WHATEVER    ;IF NOT RUNNING UNDER CP/M, SET THIS TO LOAD ADDR.
RAM:    EQU WHATEVER2   ;SET THIS TO RAM AREA,
BASE:   EQU WHATEVER3   ;AND THIS TO THE BASE OF SYSTEM MEMORY (`BASE' IS
                        ;THE RE-BOOT LOCATION UNDER CP/M; FOR NON-CP/M OPER-
                        ;ATION, IT SHOULD BE SET TO A SAFE PLACE TO JUMP TO ON
                        ;ERROR OR USER-ABORT.
        endif


        if cpm
base:   equ 0000h       ;either 0 or 4200h for CP/M systems
fcb:    equ base+5ch    ;default file control block
tbuff:  equ base+80h    ;sector buffer
bdos:   equ base+5      ;bdos entry point
tpa:    equ base+100h   ;transient program area
nfcbs:  equ 8           ;max number of open files allowed at one time
errorv: equ 255         ;error value returned by BDOS calls
cccorg: equ tpa         ;where run-time package resides *TESTING*

                        ;*****************************************************
ram:    equ cccorg+471h ;THIS WILL PROBABLY CHANGE IF YOU CUSTOMIZE CCC.ASM
                        ;*****************************************************
        endif


cr:     equ 0dh         ;ASCII codes:   carriage return
lf:     equ 0ah         ;               linefeed
newlin: equ lf          ;               newline
tab:    equ 9           ;               tab
bs:     equ 08h         ;               backspace
cntrlc: equ 3           ;               control-C

;
; Subroutines in C.CCC (the addresses should be that of the
; appropriate jump vector entry points):
;

error:  equ cccorg+1dh  ;return -1 in HL:
```

```
exit:   equ error+3     ;close all open files and reboot

        if cpm
close:  equ error+6
setfcb: equ error+9     ;set up fcb at HL from text at DE
fsfd:   equ error+12    ;set C according to whether file fd is open
fsfcb:  equ error+15    ;figure address of internal fcb for file fd
        endif


eqwel:  equ cccorg+0e5h

smod:   equ cccorg+10fh
usmod:  equ cccorg+129h
smul:   equ cccorg+13fh
usmul:  equ cccorg+16bh
usdiv:  equ cccorg+189h
sdiv:   equ cccorg+1cbh

cmphd:  equ cccorg+1ddh
cmh:    equ cccorg+1fah
cmd:    equ cccorg+202h

ma1toh: equ cccorg+20ah ;set 1st stack element into HL and A
ma2toh: equ cccorg+213h ;    2nd
ma3toh: equ ma2toh+6    ;    3rd
ma4toh: equ ma2toh+12   ;    4th
ma5toh: equ ma2toh+18   ;    5th
ma6toh: equ ma2toh+24   ;    6th
ma7toh: equ ma2toh+30   ;    7th

arghak: equ ma2toh+36   ;copy first 6 or so stack elements to argc area
setdma: equ cccorg+460h ;set CP/M internal DMA pointer to BASE+80h (tbuff)

;
; The following addresses will depend on the value of RAM if you
; customize CCC.ASM....be sure they correspond to the assembly
; results of CCC.ASM in such cases. If you remove some of the data
; areas from CCC.ASM (in case they aren't needed), be sure to remove
; from here also.
;

        org ram

room:   ds 30           ;misc. scratch area (for use by BDS...you can have
                        ;the last ten bytes or so, though, if you really
                        ;need them)

pbase:  ds 2            ;DMA video plotting base
ysize:  ds 2            ;screen width (# of columns)
xsize:  ds 2            ;screen length (# of lines)
psize:  ds 2            ;screen size (ysize * xsize)

rseed:  ds 8            ;random number seed scratch area

args:   ds 14           ;where arghak puts arg values off the stack

iohack: ds 6            ;room for input and output ops for "inp" and "outp"

allocp: ds 2            ;storage allocation pointer
```

```
alocmx: ds 2            ;highest addr useable by storage allocator

;
; This is the end of the user-customizable area. The remaining
; equated values are not to be altered.
;

;
; Special locations in C.CCC containing interesting pointers:
;

extrns: equ cccorg+15h  ;base of external data area (set by CLINK)
cccsiz: equ cccorg+17h  ;size of C.CCC for use by CLINK only
codend: equ cccorg+19h  ;address of byte following last byte of program code
                        ; (set by CLINK)
freram: equ cccorg+1bh  ;first free address after external area
                        ; (set by CLINK)


arg1:   equ args        ;these are just convenient names for
arg2:   equ args+2      ;the words in the "args" area
arg3:   equ args+4
arg4:   equ args+6
arg5:   equ args+8
arg6:   equ args+10
arg7:   equ args+12

tmp:    equ room        ;some scratch data areas used by library
tmp1:   equ room+1      ;functions.
tmp2:   equ room+2
tmp2a:  equ room+4
unget1: equ room+6
lastc:  equ room+7


;
; BDOS call codes:
;

        if cpm
conin:  equ 1           ;get a character from console
conout: equ 2           ;write a character to console
lstout: equ 5           ;write a character to list device
dconio: equ 6           ;direct console I/O (only for CP/M 2.0)
pstrng: equ 9           ;print string (terminated by '$')
getlin: equ 10          ;get buffered line from console
cstat:  equ 11          ;get console status
select: equ 14          ;select disk
openc:  equ 15          ;open a file
closec: equ 16          ;close a file
delc:   equ 19          ;delete a file
reads:  equ 20          ;read a sector (sequential)
writs:  equ 21          ;write a sector (sequential)
creatc: equ 22          ;make a file
renc:   equ 23          ;rename file
sdma:   equ 26          ;set dma
readr:  equ 33          ;read random sector
writr:  equ 34          ;write random sector
cfsizc: equ 35          ;compute file size
srecc:  equ 36          ;set random record
```

```
endif
```

```
/*
        NOBOOT.C        written by Leor Zolman
                                3/82

        Given the name of a C-generated COM file (linked with the standard
        distribution version of the C.CCC run-time package), this program
        changes that COM file so that it does not perform a warm-boot after
        its execution is complete, but instead preserves the CCP (Console
        Command Processor) that is in memory when execution begins and
        returns to the CCP directly following execution.

        NOTE: If a command is the object of a PIPE operation using DIO,
        then a warm-boot will always occur after its execution, whether
        or not NOBOOT has been applied to it.
*/

#include "bdscio.h"

main(argc,argv)
char **argv;
{
        int fd;
        int i;
        char c;
        char nambuf[30];
        char workbuf[0x500];

        if (argc != 2) {
                puts("Usage: noboot <C-generated COM file name>\n");
                exit();
        }

        for (i=0; (c = argv[1][i]) && c != '.'; i++)
                nambuf[i] = c;
        nambuf[i] = '\0';
        strcat(nambuf,".COM");

        if ((fd = open(nambuf,2)) == ERROR) {
                puts("Can't open: ");
                puts(nambuf);
                exit();
        }

        i = read(fd,workbuf+0x100,8);
        if (i != 8) puts("Couldn't read in at least 8 sectors...\n");

        workbuf[0x100] = 0x21;
        workbuf[0x101] = 0x00;
        workbuf[0x102] = 0x00;
        workbuf[0x103] = 0x39;
        workbuf[0x104] = 0x22;
        workbuf[0x105] = 0x79;
        workbuf[0x106] = 0x05;
        workbuf[0x107] = 0xcd;
        workbuf[0x108] = 0x34;
        workbuf[0x109] = 0x01;
        workbuf[0x10a] = 0xf9;

        workbuf[0x12f] = 0x2a;
        workbuf[0x130] = 0x79;
```

```
        workbuf[0x131] = 0x05;
        workbuf[0x132] = 0xf9;
        workbuf[0x133] = 0xc9;

        workbuf[0x134] = 0x2a;
        workbuf[0x135] = 0x06;
        workbuf[0x136] = 0x00;
        workbuf[0x137] = 0x11;
        workbuf[0x138] = 0xcc;
        workbuf[0x139] = 0xf7;
        workbuf[0x13a] = 0x19;
        workbuf[0x13b] = 0xc9;
        workbuf[0x13c] = 0x00;
        workbuf[0x13d] = 0x00;
        workbuf[0x13e] = 0x00;

        workbuf[0x443] = 0xc3;
        workbuf[0x444] = 0x2f;
        workbuf[0x445] = 0x01;

        seek(fd,0,0);
        if (write(fd,workbuf+0x100,8) != 8) {
                puts("Write error.\n");
                exit();
        }

        if (close(fd) == ERROR) {
                puts("Close error\n");
        }
}
```

```
/*
        NOBOOT.C        written by Leor Zolman
                        3/82

        Given a list of C-generated COM files (linked with the standard
        distribution version of the C.CCC run-time package), this program
        changes those COM files so that they do not perform a warm-boot after
        their execution is complete, but instead preserve the CCP (Console
        Command Processor) that is in memory when execution begins and
        return to the CCP directly following execution.

        NOTE: If a command is the object of a PIPE operation using DIO,
        then a warm-boot will always occur after its execution, whether
        or not NOBOOT has been applied to it.

        link by:
                A>clink noboot wildexp
        (or)    A>l2 noboot wildexp
*/

/*
        Cleaned up screen output by use of "CLEARS" string from
        BDSCIO.H - if you have not configured BDSCIO.H you can
        comment out the line "puts(CLEARS);" OR configure BDSCIO.H.

                        - Larry Clive
                        - 6/11/82
*/

#include "bdscio.h"

main(argc,argv)
char **argv;
{
   int fd;
   int i;
   char c;
   char nambuf[30];
   char workbuf[0x500];
   int loop;

   if (argc == 1) {
        puts("Usage: noboot <list of C-generated COM file names>\n");
        exit();
}


   for (loop = 1; loop < argc; loop++)

   {
        puts(CLEARS);       /* see second comment above - LC */
        puts("\n\nNOBOOT version 3.0\n\n-=> NOBOOT-ing ");
        puts(argv[loop]);
        putchar('\n');

        for (i=0; (c = argv[loop][i]) && c != '.'; i++)
                nambuf[i] = c;
        nambuf[i] = '\0';
        strcat(nambuf,".COM");
```

```
        if ((fd = open(nambuf,2)) == ERROR) {
                puts("Can't open: ");
                puts(nambuf);
                exit();
    }

        i = read(fd,workbuf+0x100,8);
        if (i != 8) puts("Couldn't read in at least 8 sectors...\n");

        workbuf[0x100] = 0x21;
        workbuf[0x101] = 0x00;
        workbuf[0x102] = 0x00;
        workbuf[0x103] = 0x39;
        workbuf[0x104] = 0x22;
        workbuf[0x105] = 0x79;
        workbuf[0x106] = 0x05;
        workbuf[0x107] = 0xcd;
        workbuf[0x108] = 0x34;
        workbuf[0x109] = 0x01;
        workbuf[0x10a] = 0xf9;

        workbuf[0x12f] = 0x2a;
        workbuf[0x130] = 0x79;
        workbuf[0x131] = 0x05;
        workbuf[0x132] = 0xf9;
        workbuf[0x133] = 0xc9;

        workbuf[0x134] = 0x2a;
        workbuf[0x135] = 0x06;
        workbuf[0x136] = 0x00;
        workbuf[0x137] = 0x11;
        workbuf[0x138] = 0xcc;
        workbuf[0x139] = 0xf7;
        workbuf[0x13a] = 0x19;
        workbuf[0x13b] = 0xc9;
        workbuf[0x13c] = 0x00;
        workbuf[0x13d] = 0x00;
        workbuf[0x13e] = 0x00;

        workbuf[0x443] = 0xc3;
        workbuf[0x444] = 0x2f;
        workbuf[0x445] = 0x01;

        seek(fd,0,0);
        if (write(fd,workbuf+0x100,8) != 8) {
                puts("Write error.\n");
                exit();
        }

        if (close(fd) == ERROR) {
                puts("Close error\n");
        }
    }
}
```

Floating Point Package for BDS C
****
Written by: Bob Mathias
this doc by: Leor Zolman

Components of the floating point package:

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

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

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

NOTE THAT THE RESULT MAY BE PLACED INTO EITHER OF THE ARGUEMENTS WITH NO ILL EFFECTS. I.e., the operation:

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

will successfully square 'foo' and place the result in 'foo'.

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

ftoa: converts a floating point number to an ASCII string (which you can then print out with "puts")
NOTE: explicit use of this function has been made obsolete by the new "sprintf." See FLOAT.C.

atof: converts an ASCII string (null terminated) to a floating point number

itof: converts integer to floating point.

Here are Bob's descriptions of the functions:

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

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

Returns:
an integer 1 if op1 > op2
an integer -1 if op1 < op2
a zero if op1 = op2

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

char *fpadd(result,op1,op2)
char result[5], op1[5], op2[5];
                Stores the result of op1 + op2 in result. op1
                and op2 must be floating point numbers.
                Returns a pointer to the beginning of result.


char *fpsub(result,op1,op2)
char result[5],op1[5],op2[5];
                Stores the result of op1 - op2 in result.  op1
                and op2 must be floating point numbers.
                Returns a pointer to the beginning of result.

char *fpmult(result,op1,op2)
char result[5],op1[5],op2[5];
                Stores the result of op1 * op2 in result.  op1
                and op2 must be floating point numbers. Returns
                a pointer to the beginning of result.

char *fpdiv(result,op1,op2)
char result[5],op1[5],op2[5];
                Stores the result of op1 / op2 in result.  op1
                and op2 must be floating point numbers.
                A divide by zero will return zero as result.
                Returns a pointer to the beginning of result.

char *atof(op1,s1)
char op1[5],*s;
                Converts the ASCII string s1 into a floating
                point number and stores the result in op1.
                The function will ignore leading white space
                but NO white space is allowed to be embedded
                within the number. The following are legal
                examples:
                "2", "22022222222383.333", "2.71828e-9",
                "334.3333E32".
                "3443.33 E10" would be ILLEGAL because
                it contains an embedded space.
                The value of the exponent must be within the
                range: -38 <= exponent <= 38.
                A pointer to the result is returned.

char *ftoa(s1,op1)
char *s1,op1[5];
                Converts the floating point number op1 to an
                ASCII string.  It will be formatted in
                scientific notation with seven (7) digits of
                precision. The string will be terminated by
                a null.
                Returns a pointer to the beginning of s1.

char *itof(op1, n)
char op1[5];
int n;
                Sets the floating pt. number op1 to the value
                of integer n. n is assumed to be a SIGNED
```

integer.

General observations:

Because floating point operations must be thought of in terms of FUNCTION CALLS rather than simple in-line expressions, special care must be taken not to confuse the abilities of the compiler with the abilities of the floating point package. To give a floating point number an initail value, for instance, you cannot say:

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

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

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

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

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

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

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

Some more examples:

The following will add 100.2 & -7.99 and store the result at the five character array location 'a':

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

(note that "b" and "c" must also be five character arrays)

The following would NOT add 1 to 'a' as both op1 and op2 must be floating point numbers (actually pointers to characters...):

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

Thus, it can get a bit hairy when all floating point numbers are really character arrays; but still, it's better than nothing.

All of the above functions are written in C, but most of them call a single workhorse function called "fp" to do all the really hairy work. This function has been placed into the DEFF2.CRL; it is the only machine-coded part of the package.

```
/*
        Floating point package support routines

        Note the "fp" library function, available in DEFF2.CRL,
        is used extensively by all the floating point number
        crunching functions.

        (see FLOAT.DOC for details...)

        ------------------------------------------------------------

        Usage: After compiling your program, link with this library
        by typing:

        A>clink <your program files> -f float <cr>
        ------------------------------------------------------------

        NEW FEATURE: a special "printf" function has been included
                     in this source file for use with floating point
                     operands, in addition to the normal types. The
                     printf presented here will take precedence over
                     the DEFF.CRL version when "float" is specified
                     on the CLINK command line at linkage time.
                     Note that the "fp" function, needed by most of
                     the functions in this file, resides in DEFF2.CRL
                     and will be automatically collected by CLINK.

        All functions here written by Bob Mathias, except printf and
        _spr (written by Leor Zolman.)
*/

#include "bdscio.h"

#define NORM_CODE       0
#define ADD_CODE        1
#define SUB_CODE        2
#define MULT_CODE       3
#define DIV_CODE        4
#define FTOA_CODE       5

fpcomp(op1,op2)
        char *op1,*op2;
{
        char work[5];
        fpsub(work,op1,op2);
        if (work[3] > 127) return (-1);
        if (work[0]+work[1]+work[2]+work[3]) return (1);
        return (0);
}

fpnorm(op1) char *op1;
{       fp(NORM_CODE,op1,op1);return(op1);}

fpadd(result,op1,op2)
        char *result,*op1,*op2;
{       fp(ADD_CODE,result,op1,op2);return(result);}

fpsub(result,op2,op1)
        char *result,*op1,*op2;
```

```
        {fp(SUB_CODE,result,op1,op2);return(result);}

fpmult(result,op1,op2)
        char *result,*op1,*op2;
{       fp(MULT_CODE,result,op1,op2);return(result);}

fpdiv(result,op1,op2)
        char *result,*op1,*op2;
{       fp(DIV_CODE,result,op1,op2);return(result);}

atof(fpno,s)
        char fpno[5],*s;
{
        char *fpnorm(),work[5],ZERO[5],FP_10[5];
        int sign_boolean,power;

        initb(FP_10,"0,0,0,80,4");
        setmem(fpno,5,0);
        sign_boolean=power=0;

        while (*s==' ' || *s=='\t') ++s;
        if (*s=='-'){sign_boolean=1;++s;}
        for (;isdigit(*s);++s){
                fpmult(fpno,fpno,FP_10);
                work[0]=*s-'0';
                work[1]=work[2]=work[3]=0;work[4]=31;
                fpadd(fpno,fpno,fpnorm(work));
        }
        if (*s=='.'){
                ++s;
                for (;isdigit(*s);--power,++s){
                        fpmult(fpno,fpno,FP_10);
                        work[0]=*s-'0';
                        work[1]=work[2]=work[3]=0;work[4]=31;
                        fpadd(fpno,fpno,fpnorm(work));
                }
        }
        if (toupper(*s) == 'E') {++s; power += atoi(s); }
        if (power>0)
                for (;power!=0;--power) fpmult(fpno,fpno,FP_10);
        else
        if (power<0)
                for (;power!=0;++power) fpdiv(fpno,fpno,FP_10);
        if (sign_boolean){
                setmem(ZERO,5,0);
                fpsub(fpno,ZERO,fpno);
        }
        return(fpno);
}
ftoa(result,op1)
        char *result,*op1;
{       fp(FTOA_CODE,result,op1);return(result);}

itof(op1,n)
char *op1;
int n;
{
        char temp[20];
        return atof(op1, itoa(temp,n));
}
```

```
itoa(str,n)
char *str;
{
        char *sptr;
        sptr = str;
        if (n<0) { *sptr++ = '-'; n = -n; }
        _uspr(&sptr, n, 10);
        *sptr = '\0';
        return str;
}

/*
        This is the special formatting function, which supports the
        "e" and "f" conversions as well as the normal "d", "s", etc.
        When using "e" or "f" format, the corresponding argument in
        the argument list should be a pointer to one of the five-byte
        strings used as floating point numbers by the floating point
        functions. Note that you don't need to ever use the "ftoa"
        function when using this special printf/sprintf combination;
        to achieve the same result as ftoa, a simple "%e" format
        conversion will do the trick. "%f" is used to eliminate the
        scientific notation and set the precision. The only [known]
        difference between the "e" and "f" conversions as used here
        and the ones described in the Kernighan & Ritchie book is that
        ROUNDING does not take place in this version...e.g., printing
        a floating point number which happens to equal exactly 3.999
        using a "%5.2f" format conversion will produce " 3.99" instead
        of " 4.00".
*/


_spr(line,fmt)
char *line, **fmt;
{
        char _uspr(), c, base, *sptr, *format;
        char wbuf[MAXLINE], *wptr, pf, ljflag, zfflag;
        int width, precision, exp, *args;

        format = *fmt++;        /* fmt first points to the format string */
        args = fmt;             /* now fmt points to the first arg value */
        while (c = *format++)
          if (c == '%') {
            wptr = wbuf;
            precision = 6;
            ljflag = pf = zfflag = 0;

            if (*format == '-') {
                    format++;
                    ljflag++;
             }

            if (*format == '0') zfflag++;       /* test for zero fill */

            width = isdigit(*format) ? _av2(&format) : 0;

            if ((c = *format++) == '.') {
                    precision = _av2(&format);
                    pf++;
                    c = *format++;
```

```
}

switch(toupper(c)) {
    case 'E':  if (precision>7) precision = 7;
               ftoa(wbuf,*args++);
               strcpy(wbuf+precision+3, wbuf+10);
               width -= strlen(wbuf);
               goto pad2;

    case 'F':  ftoa(&wbuf[60],*args++);
               sptr = &wbuf[60];
               while ( *sptr++ != 'E')
                    ;
               exp = atoi(sptr);
               sptr = &wbuf[60];
               if (*sptr == ' ') sptr++;
               if (*sptr == '-') {
                    *wptr++ = '-';
                    sptr++;
                    width--;
                }
               sptr += 2;

               if (exp < 1) {
                    *wptr++ = '0';
                    width--;
                }

               pf = 7;
               while (exp > 0 && pf) {
                    *wptr++ = *sptr++;
                    pf--;
                    exp--;
                    width--;
                }

               while (exp > 0) {
                    *wptr++ = '0';
                    exp--;
                    width--;
                }

               *wptr++ = '.';
               width--;

               while (exp < 0 && precision) {
                    *wptr++ = '0';
                    exp++;
                    precision--;
                    width--;
                }

               while (precision && pf) {
                    *wptr++ = *sptr++;
                    pf--;
                    precision--;
                    width--;
                }

               while (precision>0) {
```

```
                    *wptr++ = '0';
                    precision--;
                    width--;
                }

                goto pad;


        case 'D': if (*args < 0) {
                    *wptr++ = '-';
                    *args = -*args;
                    width--;
                }
        case 'U': base = 10; goto val;

        case 'X': base = 16; goto val;

        case 'O': base = 8;

            val: width -= _uspr(&wptr,*args++,base);
                 goto pad;

        case 'C': *wptr++ = *args++;
                  width--;
                  goto pad;

        case 'S': if (!pf) precision = 200;
                  sptr = *args++;
                  while (*sptr && precision) {
                      *wptr++ = *sptr++;
                      precision--;
                      width--;
                  }

            pad: *wptr = '\0';
           pad2: wptr = wbuf;
                 if (!ljflag)
                     while (width-- > 0)
                             *line++ = zfflag ? '0' : ' ';

                 while (*line = *wptr++)
                     line++;

                 if (ljflag)
                     while (width-- > 0)
                             *line++ = ' ';
                 break;

         default: *line++ = c;

      }
    }
    else *line++ = c;

  *line = '\0';
}
```

```
/*
        New functions for BDS C v1.4x: "lprintf" and "lputs"
        Written 1/18/81 by Leor Zolman
*/

#include <bdscio.h>
#define LISTDEV 2

/*
        Formatted output to the list device. Usage:

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

        Works just like "printf", except the output line is written
        to the lineprinter instead of to the console.
*/

lprintf(format)
char *format;
{
        char txtlin[MAXLINE];
        _spr(txtlin,&format);
        lputs(txtlin);
}

/*
        Put a line out to the list device. Usage:

        lputs(str)
        char *str;

        Works just like "puts", except the output line goes to the
        printer instead of to the console:
*/

lputs(str)
char *str;
{
        char c;
        while (c = *str++) {
                if (c == '\n') putc('\r',LISTDEV);
                putc(c,LISTDEV);
        }
}
```

```
/*
        CASM.C  -- written by Leor Zolman, 2/82

        CP/M ASM preprocessor: renders MAC.COM and CMAC.LIB unnecessary.

        See the CASM document (included with BDS C v1.46) for more info.

        Compile by:
                cc casm.c -o -e4000
*/

#include "bdscio.h"

#define CAREFUL 0               /* Setting this to 1 makes CASM check for
                                   and reject old "CMAC.LIB" pseudo-ops  */
#define TPALOC  (BASE+0x100)    /* base of TPA in your system   */
#define EQUMAX  500             /* maximum number of EQU ops    */
#define FUNCMAX 100             /* maximum number of functions  */
#define NFMAX   100             /* maximum number of external
                                   functions in one function    */
#define LABMAX  150             /* max number of local labels in one func */
#define TXTBUFSIZE 2000         /* max # of chars for labels and needed
                                   function names for a single function */
#define DEFDISK "C:"            /* default disk for include files       */
#define CASMEXT ".CSM"          /* extension on input files     */
#define ASMEXT  ".ASM"          /* extension on output files    */
#define DIRSIZE 512             /* max # of bytes in CRL directory      */

                /* Global data used throughout processing
                   of the input file:                   */

char    fbuf[BUFSIZ];           /* I/O buffer for main input file       */
char    incbuf[BUFSIZ];         /* I/O buffer for included file         */
char    obuf[BUFSIZ];           /* I/O buffer for output file           */

char    *cbufp;                 /* pointer to currently active input buf */
char    *cfilnam;               /* pointer to name of current input file */
char    nambuf[30],             /* filenames for current input */
        nambuf2[30],            /* and output files.            */
        onambuf[30];

char    *equtab[EQUMAX];        /* table of absolute symbols    */
int     equcount;               /* # of entries in equtab       */

char    *fnames[FUNCMAX];       /* list of functions in the source file */
int     fcount;                 /* # of entries in fnames               */

int     lino,savlino;           /* line number values used for error    */
                                /* reporting.                           */

char    doingfunc;              /* true if currently processing a function */

char    errf;                   /* true if an error has been detected   */

                /* Global data used during the processing of a
                   single function in the source file:          */

char    *nflist[NFMAX];         /* list of needed functions for a function */
int     nfcount;                /* number of entries in nflist  */
```

```
struct {
        char *labnam;           /* name of function label */
        char defined;           /* whether it has been defined yet */
} lablist[LABMAX];

int     labcount;               /* number of local labels in a function */

char    txtbuf[TXTBUFSIZE],     /* where text of needed function names  */
        *txtbufp;               /* and function labels go               */

char    linbuf[150],            /* text line buffers    */
        linsav[150],
        workbuf[150],
        pbuf[150], *pbufp;

char    *cfunam;                /* pointer to name of current function */
int     relblc;                 /* relocation object count for a function */

char    pastnfs;                /* true if we've passed all needed function */
                                /* declarations ("external" pseudo ops)     */

int     argcnt;                 /* values set by the "parse_line" function */
char    *label,
        *op,
        *argsp,
        *args[40];

char    *gpcptr;                /* general-purpose text pointer */

/*
 * Open main input file, open output file, initialize needed globals
 * and process the file:
 */

main(argchc,argchv)
char **argchv;
{
        int i,j,k;
        char c;

        puts("BD Software CRL-format ASM Preprocessor v1.46\n");

        initequ();              /* initialize EQU table with reserved words */
        fcount = 0;             /* haven't seen any functions yet */
        doingfunc = 0;          /* not currently processing a function */
        errf = 0;               /* no errors yet */

        if (argchc != 2)
                exit(puts("Usage:\ncasm <filename>\n"));

                                /* set up filenames with proper extensions: */

        for (i = 0; (c = argchv[1][i]) && c != '.'; i++)
                nambuf[i] = c;
        nambuf[i] = '\0';
        strcpy(onambuf,nambuf);
        strcat(nambuf,CASMEXT);         /* input filename */
        cbufp = fbuf;                   /* buffer pointer */
        cfilnam = nambuf;               /* current filename pointer */
        if (fopen(cfilnam,cbufp) == ERROR)
```

```
            exit(printf("Can't open %s\n",cfilnam));

        strcat(onambuf,ASMEXT);         /* output filename */
        if (fcreat(onambuf,obuf) == ERROR)
                exit(printf("Can't open %s\n",onambuf));

                                        /* begin writing output file */
        fprintf(obuf,"\nTPALOC\t\tEQU\t%04xH\n",TPALOC);

        lino = 1;                       /* initialize line count */

        while (get_line()) {            /* main loop */
                process_line();         /* process lines till EOF */
                lino++;
        }

        if (doinsfunc)                  /* if ends inside a function, error */
                abort("File ends, but last function is unterminated\n");

        fputs("\nEND$CRL\t\tEQU\t$-TPALOC\n",obuf);     /* end of functions */
        fputs("SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR \"SAVE\" !.\n",obuf);
        putdir();                       /* now spit out CRL directory */
        fputs("\t\tEND\n",obuf);         /* end of file */
        putc(CPMEOF,obuf);              /* CP/M EOF character */
        fclose(cbufp);                  /* close input file */
        fflush(obuf);                   /* flush and close output file */
        fclose(obuf);
        if (errf)
                printf("Fix those errors and try again...\n");
        else
                printf("\n%s is ready to be assembled.\n",onambuf);
}

/*
 * Get a line of text from input stream, and process
 * "include" ops on the fly:
 */

int get_line()
{
        int i;

        if (!fgets(linbuf,cbufp)) {             /* on EOF: */
                if (cbufp == incbuf) {          /* in an "include" file? */
                        fabort(cbufp->_fd);             /* close the file */
                        cbufp = fbuf;           /* go back to mainline file */
                        cfilnam = nambuf;
                        lino = savlino + 1;
                        return get_line();
                }
                else return NULL;
        }

        parse_line();                           /* not EOF, Parse line */
        if (streq(op,"INCLUDE")  ||             /* check for file inclusion */
            streq(op,"MACLIB")) {
                if (cbufp == incbuf)            /* if already in an include, */
                 abort("Only one level of inclusion is supported"); /* error */
                if (!argsp)
                 abort("No filename specified");
```

```
                cbufp = incbuf;                 /* set up for inclusion */
                savlino = lino;
                lino = 1;

                for (i = 0; !isspace(argsp[i]); i++)    /* put null after */
                        ;                               /* filename       */
                argsp[i] = '\0';

                *nambuf2 = '\0';

                if (*argsp == '<') {            /* look for magic delimiters */
                        if (argsp[2] != ':')    /* if no explicit disk given */
                                strcat(nambuf2,DEFDISK); /* then use default */
                        strcat(nambuf2,argsp+1);
                        if (nambuf2[i = strlen(nambuf2) - 1] == '>')
                                nambuf2[i] = '\0';
                } else if (*argsp == '"') {
                        strcpy(nambuf2,argsp+1);
                        if (nambuf2[i = strlen(nambuf2) - 1] == '"')
                                nambuf2[i] = '\0';
                } else
                        strcpy(nambuf2,argsp);

                if (fopen(nambuf2,cbufp) == ERROR) {
                        if (nambuf2[strlen(nambuf2) - 1] != ',') {
                                strcat(nambuf2,".LIB");
                                if (fopen(nambuf2,cbufp) != ERROR)
                                        goto ok;
                        }
                        printf("Can't open %s\n",nambuf2);
                        abort("Missing include file");
                }

        ok:     cfilnam = nambuf2;
                return get_line();
        }
        return 1;
}

parse_line()
{
        int i;
        char c;

        label = op = argsp = NULL;
        argcnt = 0;

        strcpy2(pbuf,linbuf);
        strcpy2(linsav,linbuf);
        pbufp = pbuf;

        if (!isspace(c = *pbufp)) {
                if (c == ';')
                        return;         /* totally ignore comment lines */
                label = pbufp;          /* set pointer to label */
                while (isidchr(*pbufp)) /* pass over the label identifier */
                        pbufp++;
                *pbufp++ = '\0';        /* place null after the identifier */
        }
```

```
        skip_wsp(&pbufp);
        if (!*pbufp || *pbufp == ';')
                return;
        op = pbufp;                     /* set pointer to operation mnemonic */
        while (isalpha(*pbufp))
                pbufp++;                /* skip over the op             */
        if (*pbufp) *pbufp++ = '\0';    /* place null after the op      */


                                        /* now process arguments        */
        skip_wsp(&pbufp);
        if (!*pbufp || *pbufp == ';')
                return;
        argsp = linsav + (pbufp - pbuf);        /* set pointer to arg list */

                                        /* create vector of ptrs to all args
                                           that are possibly relocatable */
        for (argcnt = 0; argcnt < 40;) {
                while (!isidstrt(c = *pbufp))
                        if (!c || c == ';')
                                return;
                        else
                                pbufp++;

                if (isidchr(*(pbufp - 1))) {
                        pbufp++;
                        continue;
                }

                args[argcnt++] = pbufp;
                while (isidchr(*pbufp)) pbufp++;
                if (*pbufp) *pbufp++ = '\0';
        }
        error("Too many operands in this instruction for me to handle\n");
}

process_line()
{
        char *cptr, c;
        int i,j;

        if (op) {
                        /* check for definitions of global data that will be
                           exempt from relocation when encountered in the
                           argument field of assembly instructions:        */

          if (streq(op,"EQU") || streq(op,"SET") ||
                (!doingfunc &&
                        (streq(op,"DS") || streq(op,"DB") || streq(op,"DW"))))
          {
                fputs(linbuf,obuf);
                cptr = sbrk2(strlen(label) + 1);
                strcpy(cptr,label);
                equtab[equcount++] = cptr;
                if (equcount >= EQUMAX)
                        abort(
                  "Too many EQU lines...increase 'EQUMAX' and recompile CASM");
                return;
          }
```

```
if (streq(op,"EXTERNAL")) {
     if (!doingfunc) abort(
      "'External's for a function must appear inside the function");
     if (pastnfs) error(
      "Externals must all be together at start of function\n");
     for (i = 0; i < argcnt; i++) {
             nflist[nfcount++] = txtbufp;
             strcpy(txtbufp,args[i]);
             bumptxtp(args[i]);
     }
     if (nfcount >= NFMAX) {
       printf("Too many external functions in function \"%s\"\n",
                              cfunam);
       abort("Change the NFMAX constant and recompile CASM");
     }
     return;
}

if (streq(op,"FUNCTION")) {
     if (!fcount) {
             fputs("\n; dummy external data information:\n",obuf);
             fputs("\t\tORG\tTPALOC+200H\n",obuf);
             fputs("\t\tDB\t0,0,0,0,0,0\n",obuf);
     }

     if (doingfunc) {
             printf("'Function' op encountered in a function.\n");
             abort("Did you forget an 'endfunc' op?");
     }
     if (!argcnt)
             abort("A name is required for the 'function' op");

     cfunam = sbrk2(strlen(args[0]) + 1);
     fnames[fcount++] = cfunam;
     strcpy(cfunam,args[0]);

     printf("Processing the %s function...          \r",cfunam);

     doingfunc = 1;
     txtbufp = txtbuf;
     labcount = 0;
     nfcount = 0;
     pastnfs = 0;
     fprintf(obuf,"\n\n; The \"%s\" function:\n",cfunam);
     fprintf(obuf,"%s$BEG\tEQU\t$-TPALOC\n",cfunam);
     return;
}

if (streq(op,"ENDFUNC") || streq(op,"ENDFUNCTION")) {
     if (!doingfunc)
       abort("'Endfunc' op encountered while not in a function");

     if (!pastnfs) flushnfs();    /* flush needed function list */
     fprintf(obuf,"%s$END\tEQU\t$\n",cfunam);
     doreloc();                   /* flush relocation parameters */

     for (i = 0; i < labcount; i++)  /* detect undefined labels */
       if (!lablist[i].defined) {
             printf("The label %s in function %s is undefined\n",
                             lablist[i].labnam,cfunam);
```

```
                    errf = 1;
                }
            doingfunc = 0;
            return;
        }
    }

#if CAREFUL
    if (streq(op,"RELOC") || streq(op,"DWREL") || streq(op,"DIRECT") ||
        streq(op,"ENDDIR") || streq(op,"EXREL") || streq(op,"EXDWREL") ||
        streq(op,"PRELUDE") || streq(op,"POSTLUDE") || streq(op,"DEFINE"))
            error("Old macro leftover from \"CMAC.LIB\" days...\n");
#endif
                        /* No special pseudo ops, so now process
                           the line as a line of assemby code:  */

    if (streq(op,"END")) return;            /* don't allow "end" yet     */

    if (!doingfunc || (!label && !op))      /* if nothing interesting on */
            return fputs(linbuf,obuf);      /* line, ignore it      */

    if (!pastnfs)                           /* if haven't flushed needed */
            flushnfs();                     /* function list yet, do it  */

                                            /* check for possible label  */
    if (label) {
            fprintf(obuf,"%s$L$%s\t\tEQU\t$-%s$STRT\n",
                    cfunam, label, cfunam);
            for (i=0; linbuf[i]; i++)
                    if (isspace(linbuf[i]) || linbuf[i] == ':')
                            break;
                    else
                            linbuf[i] = ' ';
            if (linbuf[i] == ':') linbuf[i] = ' ';
            for (i = 0; i < labcount; i++)    /* check if in label table */
              if (streq(label,lablist[i].labnam)) {         /* if found, */
                    if (lablist[i].defined) {  /* check for redefinition */
                            error("Re-defined label:");
                            printf("%s, in function %s\n",
                                            lablist[i].labnam,cfunam);
                    }
                     else
                            lablist[i].defined = 1;
                    goto out;
              }
            lablist[i].labnam = txtbufp;    /* add new entry to */
            lablist[i].defined = 1;         /* label list       */
            strcpy(txtbufp,label);
            bumptxtp(label);
            labcount++;
    }
out:
    if (!op) return fputs(linbuf,obuf);     /* if label only, all done   */

                                            /* if a non-relocatable op,  */
    if (norelop(op)) return fputs(linbuf,obuf);     /* then we're done   */

    if (argcnt && doingfunc)
      for (i = 0; i < argcnt; i++) {
            if (norel(args[i])) continue;
```

```
            if (spcptr = isef(args[i]))
               sprintf(workbuf,"%s$EF$%s-%s$STRT",
                        cfunam,spcptr,cfunam);
            else {
                    sprintf(workbuf,"%s$L$%s",cfunam,args[i]);
                    for (j = 0; j < labcount; j++)
                            if (streq(args[i],lablist[j].labnam))
                                    goto out2;
                    lablist[j].labnam = txtbufp;    /* add new entry to */
                    lablist[j].defined = 0;         /* label list       */
                    strcpy(txtbufp,args[i]);
                    bumptxtp(txtbufp);
                    labcount++;
            }

    out2:
            replstr(linbuf, workbuf, args[i] - pbuf, strlen(args[i]));

            if (streq(op,"DW")) {
                    fprintf(obuf,"%s$R%03d\tEQU\t$-%s$STRT\n",
                            cfunam, relblc++, cfunam);
                    if (argcnt > 1)
                      error("Only one relocatable value allowed per DW\n");
            }
            else
                    fprintf(obuf,"%s$R%03d\tEQU\t$+1-%s$STRT\n",
                            cfunam, relblc++, cfunam);
            break;
      }
    fputs(linbuf,obuf);
}


/*
        Test for ops in which there is guaranteed to be no need
        for generation of relocation parameters. Note that the list
        of non-relocatable ops doesn't necessarily have to be complete,
        because for any op that doesn't match, an argument must still
        pass other tests before it is deemed relocatable. This only
        speeds things up by telling the program not to bother checking
        the arguments.
*/

norelop(op)
char *op;
{
        if (streq(op,"MOV")) return 1;
        if (streq(op,"INR")) return 1;
        if (streq(op,"DCR")) return 1;
        if (streq(op,"INX")) return 1;
        if (streq(op,"DCX")) return 1;
        if (streq(op,"DAD")) return 1;
        if (streq(op,"MVI")) return 1;
        if (streq(op,"DB")) return 1;
        if (streq(op,"DS")) return 1;
        if (op[2] == 'I') {
                if (streq(op,"CPI")) return 1;
                if (streq(op,"ORI")) return 1;
                if (streq(op,"ANI")) return 1;
                if (streq(op,"ADI")) return 1;
```

```
                if (streq(op,"SUI")) return 1;
                if (streq(op,"SBI")) return 1;
                if (streq(op,"XRI")) return 1;
                if (streq(op,"ACI")) return 1;
        }
        if (streq(op,"ORG")) return 1;
        if (streq(op,"TITLE")) return 1;
        if (streq(op,"PAGE")) return 1;
        if (streq(op,"IF")) return 1;
        if (streq(op,"EJECT")) return 1;
        if (streq(op,"MACRO")) return 1;
        return 0;
}


flushnfs()
{
        int i,j, length;

        pastnfs = 1;
        relblc = 0;

        fputs("\n\n; List of needed functions:\n",obuf);
        for (i=0; i < nfcount; i++) {
                strcpy(workbuf,"\t\tDB\t'");
                length = strlen(nflist[i]);
                length = length < 8 ? length : 8;
                for (j = 0; j < length - 1; j++)
                        workbuf[6+j] = nflist[i][j];
                workbuf[6+j] = '\0';
                fprintf(obuf,"%s',%c'+80H\n",workbuf,nflist[i][j]);
        }

        fputs("\t\tDB\t0\n",obuf);
        fputs("\n; Length of body:\n",obuf);
        fprintf(obuf,"\t\tDW\t%s$END-$-2\n",cfunam);
        fputs("\n; Body:\n",obuf);
        fprintf(obuf,"%s$STRT\tEQU\t$\n",cfunam);
        if (nfcount) {
                fprintf(obuf,"%s$R%03d\tEQU\t$+1-%s$STRT\n",
                        cfunam,relblc++,cfunam);
                fprintf(obuf,"\t\tJMP\t%s$STRTC-%s$STRT\n",cfunam,cfunam);
        }
        fprintf(obuf,"%s$EF$%s\tEQU\t%s$STRT\n",cfunam,cfunam,cfunam);
        for (i=0; i < nfcount; i++)
                fprintf(obuf,"%s$EF$%s\tJMP\t0\n",cfunam,nflist[i]);
        fprintf(obuf,"\n%s$STRTC\tEQU\t$\n",cfunam);
}


doreloc()
{
        int i;
        fputs("\n; Relocation parameters:\n",obuf);
        fprintf(obuf,"\t\tDW\t%d\n",relblc);
        for(i = 0; i < relblc; i++)
                fprintf(obuf,"\t\tDW\t%s$R%03d\n",cfunam,i);
        fputs("\n",obuf);
}
```

```
putdir()
{
        int i,j, length;
        int bytecount;

        bytecount = 0;

        fputs("\n\t\tORG\tTPALOC\n\n; Directory:\n",obuf);
        for (i = 0; i < fcount; i++) {
                strcpy(workbuf,"\t\tDB\t'");
                length = strlen(fnames[i]);
                length = length < 8 ? length : 8;
                for (j = 0; j < length - 1; j++)
                        workbuf[6+j] = fnames[i][j];
                workbuf[6+j] = '\0';
                fprintf(obuf,"%s',%c'+80H\n",workbuf,fnames[i][j]);
                fprintf(obuf,"\t\tDW\t%s$BEG\n",fnames[i]);
                bytecount += (length + 2);
        }
        fputs("\t\tDB\t80H\n\t\tDW\tEND$CRL\n",obuf);

        bytecount += 3;
        if (bytecount > DIRSIZE) {
                printf("CRL Directory size will exceed 512 bytes!\n");
                printf("Break the file up into smaller chunks, please!\n");
                exit(-1);
        }
}


initequ()
{
        equtab[0] = "A";
        equtab[1] = "B";
        equtab[2] = "C";
        equtab[3] = "D";
        equtab[4] = "E";
        equtab[5] = "H";
        equtab[6] = "L";
        equtab[7] = "M";
        equtab[8] = "SP";
        equtab[9] = "PSW";
        equtab[10]= "AND";
        equtab[11]= "OR";
        equtab[12]= "MOD";
        equtab[13]= "NOT";
        equtab[14]= "XOR";
        equtab[15]= "SHL";
        equtab[16]= "SHR";
        equcount = 14;
}


int isidchr(c)  /* return true if c is legal character in identifier */
char c;
{
        return isalpha(c) || c == '$' || isdigit(c) || c == '.';
}
```

```
int isidstrt(c) /* return true if c is legal as first char of idenfitier */
char c;
{
        return isalpha(c);
}


int streq(s1, s2)       /* return true if the two strings are equal */
char *s1, *s2;
{
        if (*s1 != *s2) return 0;       /* special case for speed */
        while (*s1) if (*s1++ != *s2++) return 0;
        return (*s2) ? 0 : 1;
}


skip_wsp(strptr)        /* skip white space at *strptr and modify the ptr */
char **strptr;
{
        while (isspace(**strptr)) (*strptr)++;
}


strcpy2(s1,s2)  /* copy s2 to s1, converting to upper case as we go */
char *s1, *s2;
{
        while (*s2)
             *s1++ = toupper(*s2++);
        *s1 = '\0';
}


/*
        General-purpose string-replacement function:
                'string'        is pointer to entire string,
                'insstr'        is pointer to string to be inserted,
                'pos'           is the position in 'string' where 'insstr'
                                is to be inserted
                'lenold'        is the length of the substring in 'string'
                                that is being replaced.
*/

replstr(string, insstr, pos, lenold)
char *string, *insstr;
{
        int length, i, j, k, x;

        length = strlen(string);
        x = strlen(insstr);
        k = x - lenold;
        i = string + pos + lenold;
        if (k) movmem(i, i+k, length - (pos + lenold) + 1);
        for (i = 0, j = pos; i < x; i++, j++)
                string[j] = insstr[i];
}


error(msg)
char *msg;
```

```
{
        printf("\n\7%s: %d: %s ",cfilnam,lino,msg);
        errf = 1;
}


abort(msg)
char *msg;
{
        error(msg);
        putchar('\n');
        if (cbufp == incbuf) fclose(incbuf);
        fclose(fbuf);
        exit(-1);
}


sbrk2(n)        /* allocate storage and check for out of space condition */
{
        int i;
        if ((i = sbrk(n)) == ERROR)
                abort("Out of storage allocation space\n");
        return i;
}

bumptxtp(str)   /* bump txtbufp by size of given string + 1 */
char *str;
{
        txtbufp += strlen(str) + 1;
        if (txtbufp >= txtbuf + (TXTBUFSIZE - 8))
         abort("Out of text space. Increase TXTBUFSIZE and recompile CASM");
}


int norel(id)   /* return true if identifier is exempt from relocatetion */
char *id;
{
        if (isequ(id)) return 1;
        return 0;
}


int isequ(str)  /* return true if given string is in the EQU table */
char *str;
{
        int i;
        for (i = 0; i < equcount; i++)
                if (streq(str,equtab[i]))
                        return 1;
        return 0;
}


char *isef(str) /* return nflist entry if given string is an external */
char *str;      /* function name */
{
        int i;
        for (i = 0; i < nfcount; i++)
                if (streq(str,nflist[i]))
                        return nflist[i];
```

```
        return 0;
}

}
```

```
TITLE   'LRUN  Library Run--a utility for .LBR files'
VERSION EQU     1$0     ;82-08-06 Initial source release
        PAGE    60
;
; Requires MAC for assembly.  Due to the complexity of
; the relocation macros, this program may take a while
; to assemble.  Be prepared for periods of no disk activity
; on both passes before pressing panic button.  G.P.N.
;

;-------------------------NOTICE-----------------------------
;
;   (c) Copyright 1982  Gary P. Novosielski
;       All rights reserved.
;
;   The following features courtesy of Ron Fowler:
;       1) command line reparsing and repacking (this allows
;       the former load-only program to become a load & run
;       utility).
;       2) code necessary to actually execute the loaded file
;       3) the HELP facility (LRUN with no arguments)
;       4) modified error routines to avoid warm-boot delay
;          (return to CCP directly instead)
;
;       Permission to distribute this program in source or
;       object form without prior written approval is granted
;       only under the following conditions.
;
;               1. No charge is imposed for the program.
;               2. Charges for incidental costs including
;                  but not limited to media, postage, tele-
;                  communications, and data storage do not
;                  exceed those costs actually incurred.
;               3. This Notice and any copright notices in
;                  the object code remain intact
;
;                       (signed)  Gary P. Novosielski
;
;------------------------------------------------------------
;
; LRUN is intended to be used in conjunction with libraries
; created with LU.COM, a library utility based upon the
; groundwork laid by Michael Rubenstien, with some additional
; inspiration from Leor Zolman's CLIB librarian for .CRL files.
;
; The user can place the less frequently used command (.COM)
; files in a library to save space, and  still be able to run
; them when required, by typing:
;       LRUN <normal command line>.
; The name of the library can be specified, but the greatest
; utility will be achieved by placing all commands in one
; library called COMMAND.LBR, or some locally defined name,
; and always letting LRUN use that name as the default.
;

;Syntax:
;       LRUN [-<lbrname>] <command> [<parameters>]
;
;where:
;<lbrname>      is the optional library name.  In the
```

```
;               distrubution version, this defaults to
;               COMMAND.LBR.  If the user wishes to use a
;               different name for the default, the 8-byte
;               literal at DFLTNAM below may be changed to
;               suit local requirements. The current drive
;               is searched for the .LBR file, and if not
;               found there, the A: drive is searched.
;               **Note that the leading minus sign (not a part
;               of the name) is required to indicate an
;               override library name is being entered.
;
;<command>      is the name of the .COM file in the library
;
;<line>         is the (possibly empty) set of parameters
;               which are to be passed to <command>, as in
;               normal CP/M syntax.  Notice that if the
;               library name is defaulted, the syntax is
;               simply:
;     LRUN <command line>
;               which is just the normal command line with
;               LRUN prefixed to it.
;
@SYS    SET     0
@KEY    SET     1
@CON    SET     2
@RDR    SET     3
@PUN    SET     4
@LST    SET     5
@DIO    SET     6
@RIO    SET     7
@SIO    SET     8
@MSG    SET     9
@INP    SET     10
@RDY    SET     11
@VER    SET     12
@LOG    SET     13
@DSK    SET     14
@OPN    SET     15
@CLS    SET     16
@DIR    SET     17
@NXT    SET     18
@DEL    SET     19
@FRD    SET     20
@FWR    SET     21
@MAK    SET     22
@REN    SET     23
@CUR    SET     25
@DMA    SET     26
@CHG    SET     30
@USR    SET     32
@RRD    SET     33
@RWR    SET     34
@SIZ    SET     35
@REC    SET     36
@LOGV   SET     37      ;2.2 only
@RWR0   SET     40      ;2.2 only
;
CPMBASE EQU     0
BOOT    SET     CPMBASE
BDOS    SET     BOOT+5
```

```
TFCB    EQU     BOOT+5CH
TFCB1   EQU     TFCB
TFCB2   EQU     TFCB+16
TBUFF   EQU     BOOT+80H
TPA     EQU     BOOT+100H
CTRL    EQU     ' '-1           ;Ctrl char mask
CR      SET     CTRL AND 'M'
LF      SET     CTRL AND 'J'
TAB     SET     CTRL AND 'I'
FF      SET     CTRL AND 'L'
BS      SET     CTRL AND 'H'
FALSE   SET     0
TRUE    SET     NOT FALSE
;
CPM     MACRO   FUNC,OPERAND,CONDTN
        LOCAL   PAST
        IF      NOT NUL CONDTN
        DB      ( J&CONDTN ) XOR 8
        DW      PAST
        ENDIF           ;;of not nul condtn
        IF      NOT NUL OPERAND
        LXI     D,OPERAND
        ENDIF           ;;of not nul operand
        IF      NOT NUL FUNC
        MVI     C,&&FUNC
        ENDIF
        CALL    BDOS
PAST:
        ENDM
;
BLKMOV  MACRO   DEST,SRCE,LEN,COND
        LOCAL   PAST
        JMP     PAST
@BMVSBR:
        MOV     A,B
        ORA     C
        RZ
        DCX     B
        MOV     A,M
        INX     H
        STAX    D
        INX     D
        JMP     @BMVSBR
BLKMOV  MACRO   DST,SRC,LN,CC
        LOCAL   PST
        IF      NOT NUL CC
        DB      ( J&CC ) XOR 8
        DW      PST
        ENDIF
        IF      NOT NUL DST
        LXI     D,DST
        ENDIF
        IF      NOT NUL SRC
        LXI     H,SRC
        ENDIF
        IF      NOT NUL LN
        LXI     B,LN
        ENDIF
        CALL    @BMVSBR
        IF      NOT NUL CC
```

```
PST:
        ENDIF
        ENDM
PAST:   BLKMOV  DEST,SRCE,LEN,COND
        ENDM

;
OVERLAY SET     0
; Macro Definitions
;
RTAG    MACRO   LBL
??R&LBL EQU     $+2-@BASE
        ENDM
;
RGRND   MACRO   LBL
??R&LBL EQU     0FFFFH
        ENDM
;
R       MACRO   INST
@RLBL   SET     @RLBL+1
        RTAG    %@RLBL
        INST-@BASE
        ENDM
;
NXTRLD  MACRO   NN
@RLD    SET     ??R&NN
@NXTRLD SET     @NXTRLD + 1
        ENDM
;
;
; Enter here from Console Command Processor (CCP)
;
CCPIN   ORG     TPA
        JMP     INTRO           ;Jump around signon
;
SIGNON:
        DB      'LRUN Ver '     ;Signon message
        DB      VERSION/10+'0'
        DB      '.'
        DB      VERSION MOD 10+'0'
        DB      CR,LF
        DB      ' Copyright (c) 1982  Gary P. Novosielski '
        DB      '$',CTRL AND 'Z'
;
INTRO:
        LXI     H,0             ;set the CCP entry stackpointer
        DAD     SP              ;(used only if HELP request
        SHLD    SPSAVE          ; is encountered)
        CPM     MSG,SIGNON;     ;Display signon
        CALL    SETUP           ;initialize.
        LHLD    BDOS+1          ;find top of memory
        MOV     A,H             ;page address
                                ;Form destination...
        SUI     PAGES           ;...address in
        MOV     D,A             ;DE pair.
        MVI     E,0
        PUSH    D               ;save on stack
;
        BLKMOV  ,@BASE,SEGLEN   ;Move the active segment.
;
```

```
/*
        This program is a simple example of how to use
        Bob Mathias's floating point package.
        After compiling this and the FLOAT.C library, link by saying:

        A>clink floatsum -f float <cr>

        Note: the "printf" function resulting from this linkage
        will support the "e" and "f" floating point conversion
        characters, but the regular "printf" would not. The reason:
        the special version of "_spr" in the FLOAT.C source file
        is loaded before the library version of "_spr", and
        thus supports the extra features.
*/

main()
{
        char s1[5], s2[5];
        char string[30];
        char sb[30];
        int i;
        atof(s1,"0");
        while (1) {
                printf("sum = %10.6f\n",s1);
                printf("\nEnter a floating number: ");
                fpadd(s1,s1,atof(s2,gets(string)));
        }
}
```

```
#include "bdscio.h"

/*
        STDLIB1.C -- for BDS C v1.46 -- Leor Zolman, 3/5/82

        The files STDLIB1.C and STDLIB2.C contain the source for
        all functions in the DEFF.CRL library file.

        Functions appearing in this source file:

        fopen           getc            ungetc          getw
        fcreat          putc            putw
        fflush          fclose
        atoi
        strcat          strcmp          strcpy          strlen
        isalpha         isupper         islower         isdigit
        isspace         toupper         tolower
        qsort **        _swp
        initw           initb           getval
        alloc *         free *
        abs             max             min

        * -- Compilation of alloc and free must be explicitly enabled by
             swapping the commenting of the ALLOC_ON and ALLOC_OFF definitions
             in BDSCIO.H.

        ** - Qsort has been rendered more efficient by having the "_swp"
             function use the "movmem" library function to swap objects,
             allocating temporary space on the stack. The defined symbol
             "MAX_QSORT_WIDTH" specifies the largest allowable size for a
             single instance of the objects being sorted. If you ever plan
             to sort object of greater width, change this define!

*/


#define         MAX_QSORT_WIDTH 513             /* Largest object "qsort"
                                                   can sort             */

/*
        Buffered I/O for C:
*/

#define STD_IN  0
#define STD_OUT 1
#define STD_ERR 4

#define DEV_LST 2
#define DEV_RDR 3
#define DEV_PUN 3


int fopen(filename,iobuf)
FILE *iobuf;
char *filename;
{
        if ((iobuf -> _fd = open(filename,0))<0) return ERROR;
        iobuf -> _nleft = 0;
        return iobuf -> _fd;
```

```
}


int getc(iobuf)
FILE *iobuf;
{
        int nsecs;
        if (iobuf == STD_IN) return getchar();
        if (iobuf == DEV_RDR) return bdos(3);
        if (!iobuf->_nleft--)           /* if buffer empty, fill it up first */
         {
                if ((nsecs = read(iobuf -> _fd, iobuf -> _buff, NSECTS)) <= 0)
                        return iobuf -> _nleft++;
                iobuf -> _nleft = nsecs * SECSIZ - 1;
                iobuf -> _nextp = iobuf -> _buff;
         }
        return *iobuf->_nextp++;
}

/*
        Buffered "unget" a character routine. Only ONE
        byte may be "ungotten" between consecutive "getc" calls.
*/

int ungetc(c, iobuf)
FILE *iobuf;
char c;
{
        if (iobuf == STD_IN) return ungetch(c);
        if ((iobuf < 7) || iobuf -> _nleft == (NSECTS * SECSIZ)) return ERROR;
        *--iobuf -> _nextp = c;
        iobuf -> _nleft++;
        return OK;
}


int getw(iobuf)
FILE *iobuf;
{
        int a,b;
        if (((a=getc(iobuf)) >= 0) && ((b= getc(iobuf)) >=0))
                        return 256*b+a;
        return ERROR;
}


int fcreat(name,iobuf)
char *name;
FILE *iobuf;
{
        if ((iobuf -> _fd = creat(name)) < 0 ) return ERROR;
        iobuf -> _nextp = iobuf -> _buff;
        iobuf -> _nleft = (NSECTS * SECSIZ);
        return iobuf -> _fd;
}


int putc(c,iobuf)
char c;
FILE *iobuf;
```

```
{
        if (iobuf <= 4)                 /* handle special device codes: */
         {
                switch (iobuf)
              -  {
                        case STD_OUT: return putchar(c);  /* std output */
                        case DEV_LST: return (bdos(5,c)); /* list dev.  */
                        case DEV_PUN: return (bdos(4,c)); /* to punch   */
                        case STD_ERR: if (c == '\n')      /* to std err */
                                              bdos(2,'\r');
                                      return bdos(2,c);
                 }
         }
        if (!iobuf -> _nleft--)         /*   if buffer full, flush it   */
         {
                if ((write(iobuf -> _fd, iobuf -> _buff, NSECTS)) != NSECTS)
                        return ERROR;
                iobuf -> _nleft = (NSECTS * SECSIZ - 1);
                iobuf -> _nextp = iobuf -> _buff;
         }
        return *iobuf -> _nextp++ = c;
}


int putw(w,iobuf)
unsigned w;
FILE *iobuf;
{
        if ((putc(w%256,iobuf) >=0 ) && (putc(w / 256,iobuf) >= 0))
                                return w;
        return ERROR;
}


int fflush(iobuf)
FILE *iobuf;
{
        int i;
        if (iobuf < 4) return OK;
        if (iobuf -> _nleft == (NSECTS * SECSIZ)) return OK;

        i = NSECTS - iobuf->_nleft / SECSIZ;
        if (write(iobuf -> _fd, iobuf -> _buff, i) != i)
                        return ERROR;
        i = (i-1) * SECSIZ;

        if (iobuf -> _nleft) {
                movmem(iobuf->_buff + i, iobuf->_buff, SECSIZ);
                iobuf -> _nleft += i;
                iobuf -> _nextp -= i;
                return seek(iobuf->_fd, -1, 1);
         }

        iobuf -> _nleft = (NSECTS * SECSIZ);
        iobuf -> _nextp = iobuf -> _buff;
        return OK;
}

int fclose(iobuf)
FILE *iobuf;
```

```
{
        if (iobuf < 4) return OK;
        return close(iobuf -> _fd);
}


/*
        Some string functions
*/

int atoi(n)
char *n;
{
        int val;
        char c;
        int sign;
        val=0;
        sign=1;
        while ((c = *n) == '\t' || c== ' ') ++n;
        if (c== '-') {sign = -1; n++;}
        while (  isdigit(c = *n++)) val = val * 10 + c - '0';
        return sign*val;
}


char *strcat(s1,s2)
char *s1, *s2;
{
        char *temp; temp=s1;
        while(*s1) s1++;
        do *s1++ = *s2; while (*s2++);
        return temp;
}


int strcmp(s,t)
char s[], t[];
{
        int i;
        i = 0;
        while (s[i] == t[i])
                if (s[i++] == '\0')
                        return 0;
        return s[i] - t[i];
}


char *strcpy(s1,s2)
char *s1, *s2;
{
        char *temp; temp=s1;
        while (*s1++ = *s2++);
        return temp;
}


int strlen(s)
char *s;
{
        int len;
```

```
        len=0;
        while (*s++) len++;
        return len;
}


/*
        Some character diddling functions
*/

int isalpha(c)
char c;
{
        return isupper(c) || islower(c);
}


int isupper(c)
char c;
{
        return c>='A' && c<='Z';
}


int islower(c)
char c;
{
        return c>='a' && c<='z';
}


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


int isspace(c)
char c;
{
        return c==' ' || c=='\t' || c=='\n';
}


char toupper(c)
char c;
{
        return islower(c) ? c-32 : c;
}


char tolower(c)
char c;
{
        return isupper(c) ? c+32 : c;
}
```

```
qsort(base, nel, width, compar)
char *base; int (*compar)();
unsigned width,nel;
{       int i, j;
        unsigned gap, ngap, t1;
        int jd, t2;

        t1 = nel * width;
        for (ngap = nel / 2; ngap > 0; ngap /= 2) {
           gap = ngap * width;
           t2 = gap + width;
           jd = base + gap;
           for (i = t2; i <= t1; i += width)
              for (j =  i - t2; j >= 0; j -= gap) {
                if ((*compar)(base+j, jd+j) <=0) break;
                        _swp(width, base+j, jd+j);
              }
        }
}

_swp(w,a,b)
char *a,*b;
unsigned w;
{
        char swapbuf[MAX_QSORT_WIDTH];
        movmem(a,swapbuf,w);
        movmem(b,a,w);
        movmem(swapbuf,b,w);
}


/*
        Initialization functions
*/


initw(var,string)
int *var;
char *string;
{
        int n;
        while ((n = getval(&string)) != -32760) *var++ = n;
}

initb(var,string)
char *var, *string;
{
        int n;
        while ((n = getval(&string)) != -32760) *var++ = n;
}

int getval(strptr)
char **strptr;
{
        int n;
        if (!**strptr) return -32760;
        n = atoi(*strptr);
        while (**strptr && *(*strptr)++ != ',');
        return n;
}
```

```
/*
        Storage allocation functions:
*/

#ifdef ALLOC_ON         /* Compilation of alloc and free is enabled only
                           when the ALLOC_ON symbol is #defined in BDSCIO.H */

char *alloc(nbytes)
unsigned nbytes;
{
        struct _header *p, *q, *cp;
        int nunits;
        nunits = 1 + (nbytes + (sizeof (_base) - 1)) / sizeof (_base);
        if ((q = _allocp) == NULL) {
                _base._ptr = _allocp = q = &_base;
                _base._size = 0;
         }
        for (p = q -> _ptr; ; q = p, p = p -> _ptr) {
                if (p -> _size >= nunits) {
                        if (p -> _size == nunits)
                                q -> _ptr = p -> _ptr;
                        else {
                                p -> _size -= nunits;
                                p += p -> _size;
                                p -> _size = nunits;
                         }
                        _allocp = q;
                        return p + 1;
                 }
                if (p == _allocp) {
                        if ((cp = sbrk(nunits * sizeof (_base))) == ERROR)
                                return NULL;
                        cp -> _size = nunits;
                        free(cp+1);     /* remember: pointer arithmetic! */
                        p = _allocp;
                }
         }
}


free(ap)
struct _header *ap;
{
        struct _header *p, *q;

        p = ap - 1;     /* No need for the cast when "ap" is a struct ptr */

        for (q = _allocp; !(p > q && p < q -> _ptr); q = q -> _ptr)
                if (q >= q -> _ptr && (p > q || p < q -> _ptr))
                        break;
        if (p + p -> _size == q -> _ptr) {
                p -> _size += q -> _ptr -> _size;
                p -> _ptr = q -> _ptr -> _ptr;
         }
        else p -> _ptr = q -> _ptr;

        if (q + q -> _size == p) {
                q -> _size += p -> _size;
```

```
                q -> _ptr = p -> _ptr;
        }
        else q -> _ptr = p;

        _allocp = q;
}

#endif



/*
        Now some really hairy functions to wrap things up:
*/

int abs(n)
{
        return (n<0) ? -n : n;
}

int max(a,b)
{
        return (a > b) ? a : b;
}

int min(a,b)
{
        return (a <= b) ? a : b;
}
```

```
/*
        STDLIB2.C -- for BDS C v1.46 -- Leor Zolman, 3/5/82

        This file contains the source for the following
        library functions:

        printf          fprintf         sprintf         _spr
        scanf           fscanf          sscanf          _scn
        fgets
        puts            fputs
        swapin
*/

#include "bdscio.h"

char toupper(), isdigit();

printf(format)
char *format;
{
        char line[MAXLINE];
        _spr(line,&format);     /* use "_spr" to form the output */
        puts(line);             /* and print out the line        */
}


int scanf(format)
char *format;
{
        char line[MAXLINE];
        gets(line);                     /* get a line of input from user */
        return _scn(line,&format);      /* and scan it with "_scn"       */
}


int fprintf(iobuf,format)
char *format;
struct _buf *iobuf;
{
        char text[MAXLINE];
        _spr(text,&format);
        return fputs(text,iobuf);
}


int fscanf(iobuf,format)
char *format;
struct _buf *iobuf;
{
        char text[MAXLINE];
        if (!fgets(text,iobuf)) return 0;
        return _scn(text,&format);
}

sprintf(buffer,format)
char *buffer, *format;
{
        _spr(buffer,&format);   /* call _spr to do all the work */
}
```

```
int sscanf(line,format)
char *line, *format;
{
    return _scn(line,&format);      /* let _scn do all the work */
}


_spr(line,fmt)
char *line, **fmt;
{
        char _uspr(), c, base, *sptr, *format;
        char wbuf[MAXLINE], *wptr, pf, ljflag, zfflag;
        int width, precision,  *args;

        format = *fmt++;    /* fmt first points to the format strins    */
        args = fmt;         /* now fmt points to the first arg value    */

        while (c = *format++)
          if (c == '%') {
            wptr = wbuf;
            precision = 6;
            ljflag = pf = zfflag = 0;

            if (*format == '-') {
                    format++;
                    ljflag++;
             }


            if (*format == '0') zfflag++;       /* test for zero-fill */

            width = (isdigit(*format)) ? _av2(&format) : 0;

            if ((c = *format++) == '.') {
                    precision = _av2(&format);
                    pf++;
                    c = *format++;
             }

            switch(toupper(c)) {

                case 'D':  if (*args < 0) {
                                *wptr++ = '-';
                                *args = -*args;
                                width--;
                           }

                case 'U':  base = 10; goto val;

                case 'X':  base = 16; goto val;

                case 'O':  base = 8;  /* note that arbitrary bases can be
                                         added easily before this line */

                     val:  width -= _uspr(&wptr,*args++,base);
                           goto pad;

                case 'C':  *wptr++ = *args++;
                           width--;
```

```
                        goto pad;

                case 'S':  if (!pf) precision = 200;
                        sptr = *args++;
                        while (*sptr && precision) {
                             *wptr++ = *sptr++;
                             precision--;
                             width--;
                         }

                    pad:  *wptr = '\0';
                    pad2: wptr = wbuf;
                             if (!ljflag)
                                  while (width-- > 0)
                                          *line++ = zfflag ? '0' : ' ';

                             while (*line = *wptr++)
                                  line++;

                             if (ljflag)
                                  while (width-- > 0)
                                          *line++ = ' ';
                             break;

                 default:  *line++ = c;

               }
             }
             else *line++ = c;

        *line = '\0';
}

/*
        Internal routine used by "_spr" to perform ascii-
        to-decimal conversion and update an associated pointer:
*/

int _sv2(sptr)
char **sptr;
{
        int n;
        n = 0;
        while (isdigit(**sptr)) n = 10 * n + *(*sptr)++ - '0';
        return n;
}

char _uspr(string, n, base)
char **string;
unsigned n;
{
        char length;
        if (n<base) {
                *(*string)++ = (n < 10) ? n + '0' : n + 55;
                return 1;
        }
        length = _uspr(string, n/base, base);
        _uspr(string, n%base, base);
        return length + 1;
}
```

```
/*
        General formatted input conversion routine. "line" points
        to a string containing ascii text to be converted, and "fmt"
        points to an argument list consisting of first a format
        string and then a list of pointers to the destination objects.
*/

int _scn(line,fmt)
char *line, **fmt;
{
        char sf, c, base, n, *sptr, *format;
        int sign, val, **args;

        format = *fmt++;        /* fmt first points to the format string */
        args = fmt;             /* now it points to the arg_list */

        n = 0;
        while (c = *format++)
        {
           if (isspace(c)) continue;    /* skip white space in format string */
           if (c != '%')                /* if not %, must match text */
            {
               if (c != _igs(&line)) return n;
               else line++;
            }
           else         /* process conversion */
            {
               sign = 1;
               base = 10;
               sf = 0;
               if ((c = *format++) == '*')
                {
                      sf++;             /* if "*" given, supress assignment */
                      c = *format++;
                }
               switch (toupper(c))
                {
                  case 'X': base = 16;
                            goto doval;

                  case 'O': base = 8;
                            goto doval;

                  case 'D': if (_igs(&line) == '-') {
                               sign = -1;
                               line++;
                             }

           doval:  case 'U': val = 0;
                            if (_bc(_igs(&line),base) == ERROR)
                              return n;
                            while ((c = _bc(*line++,base)) != 255)
                              val = val * base + c;
                            line--;
                            break;

                  case 'S': _igs(&line);
                            sptr = *args;
```

```
                        while (c = *line++)   {
                           if (c == *format) {
                                   format++;
                                   break;
                            }
                           if (!sf) *sptr++ = c;
                         }
                        if (!sf) {
                           n++;
                           *sptr = '\0';
                           args++;
                         }
                        continue;

                  case 'C': if (!sf) {
                           poke(*args++, *line);
                           n++;
                        }
                        line++;
                        continue;

                  default:  return n;
                 }
                if (!sf)
                 {
                       **args++ = val * sign;
                       n++;
                 }
             }
           if (!*line) return n;        /* if end of input string, return */
        }
        return n;
}

char _iss(sptr)
char **sptr;
{
        char c;
        while (isspace(c = **sptr)) ++*sptr;
        return (c);
}

int _bc(c,b)
char c,b;
{
        if (isalpha(c = toupper(c))) c -= 55;
         else  if (isdigit(c))  c -= 0x30;
         else return ERROR;
        if (c > b-1) return ERROR;
               else return c;
}

puts(s)
char *s;
{
        while (*s) putchar(*s++);
}


char *fgets(s,iobuf)
```

```
char *s;
struct _buf *iobuf;
{
        int count, c;
        char *cptr;
        count = MAXLINE;
        cptr = s;
        if ( (c = getc(iobuf)) == CPMEOF || c == EOF) return NULL;

        do {
                if ((*cptr++ = c) == '\n') {
                  if (cptr>s+1 && *(cptr-2) == '\r')
                        *(--cptr - 1) = '\n';
                  break;
                }
         } while (count-- && (c=getc(iobuf)) != EOF && c != CPMEOF);

        if (c == CPMEOF) ungetc(c,iobuf);       /* push back control-Z */
        *cptr = '\0';
        return s;
}


fputs(s,iobuf)
char *s;
struct _buf *iobuf;
{
        char c;
        while (c = *s++) {
                if (c == '\n') putc('\r',iobuf);
                if (putc(c,iobuf) == ERROR) return ERROR;
        }
        return OK;
}

swapin(name,addr)
char *name;
{
        int fd;
        if (( fd = open(name,0)) == ERROR) {
                printf("Swapin: cannot open %s\n",name);
                return ERROR;
        }
        if ((read(fd,addr,512)) < 0) {
                printf("Swapin: read error on %s\n",name);
                close(fd);
                return ERROR;
        }
        close(fd);
        return OK;
}
```

```
/*
        WILDEXP.C       v1.1    3/21/82
        BDS C Command-line Wild-card expansion utility
        Written by Leor Zolman

        Lets ambiguous file names appear on the command line to C programs,
        automatically expanding the parameter list to contain all files that
        fit the afn's.

        An afn preceded by a "!" causes all names matching the given afn to
        be EXCLUDED from the resulting expansion list. Thus, to yield a
        command line containing all files except "COM" files, you'd say:

                A>progname !*.com <cr>

        Another example: to get all files on B: except .C files, say:

                A>prognam b:*.* !b:*.c <cr>

        When giving a "!" afn, "*" chars in the string matches to the end of
        either the filename or extension, just like CP/M, but "?" chars match
        ONE and ONLY ONE character in either the filename or extension.


        To use WILDEXP, begin your "main" function as follows:

        ------------------------------------------
        main(argc,argv)
        char **argv;
        {
                ...                     /* local declarations  */
                wildexp(&argc,&argv);   /* first statement in program  */
                dioinit(&argc,argv);    /* if using DIO, put this here */
                .
                .
                .
        ------------------------------------------

        and link WILDEXP.CRL in with your program. That's all there is to
        it; note that "wildexp" uses the "sbrk" function to obtain storage,
        so don't go playing around with memory that is outside of the
        external or stack areas unless you obtain the memory through "sbrk"
        or "alloc" calls.

*/

#include        "bdscio.h"
#define         MAXITEMS        200     /* max no. of items after expansion */
#define         SEARCH_FIRST    17      /* BDOS calls */
#define         SEARCH_NEXT     18

wildexp(oargcp, oargvp)
int     *oargcp;                /* pointer to old argc */
char    ***oargvp;              /* pointer to old argv */
{
        int     nargc;          /* new argc */
        char    **nargv;        /* new argv */
        char    **oargv;        /* old argv */
        int     oargc;          /* old argc */
        char    fcb[36];        /* fcb used for search for first/next calls */
```

```
char    dmapos;         /* value returned by search calls */
char    first_time;     /* used in search routine */
char    tmpfn[20],      /* temp filename buffer */
        *tmpfnp;
char    *notfns[20];    /* list of !<afn> entries */
int     notcount;       /* count of entries in notfns */
char    cur_drive;      /* currently logged drive */
int     i,j,k;

cur_drive = bdos(25);

oargv = *oargvp;
oargc = *oargcp;
nargc = 1;
notcount = 0;

if ((nargv = sbrk(MAXITEMS * 2 + 2)) == ERROR)
        return ERROR;

for (i = 1; i < oargc; i++)
        if (oargv[i][0] == '!') {
                if (i == 1) {
                        oargv[oargc] = "*.*";
                        oargc++;
                }
                notfns[notcount++] = &oargv[i][1];
        }
        else if (!haswild(oargv[i]))
                nargv[nargc++] = oargv[i];
        else {
           setfcb(fcb,oargv[i]);

           tmpfnp = tmpfn;
           if ((tmpfn[1] = oargv[i][1]) == ':') {
                tmpfn[0] = oargv[i][0];
                tmpfnp = tmpfn + 2;
                bdos(14,tmpfn[0] - 'A');
           }

           first_time = TRUE;
           while (1) {                  /* find all matching files */
                dmapos = bdos(first_time ? SEARCH_FIRST : SEARCH_NEXT,
                                                        fcb);
                if (dmapos == 255) break;
                first_time = FALSE;
                hackname(tmpfnp,(BASE + 0x80 + dmapos * 32));
                if ((nargv[nargc] = sbrk(strlen(tmpfn) + 1)) == ERROR)
                        return ERROR;
                strcpy(nargv[nargc++], tmpfn);
           }
           bdos(14,cur_drive);          /* restore to current drive */
        }

for (i = 0; i < notcount; i++)
        for (j = 1; j < nargc; j++)
                while (match(notfns[i],nargv[j],cur_drive))
                {
                        if(j == --nargc)
                                break;
                        for (k = j; k < nargc; k++)
```

```
                                        nargv[k] = nargv[k+1];
                        }
        *oargcp = nargc;
        *oargvp = nargv;
        return 0;
}

hackname(dest,source)
char *dest, *source;
{
        int i,j;

        j = 0;

        for (i = 1; i < 9; i++)
        {
                if (source[i] == ' ') break;
                dest[j++] = source[i];
        }
        if (source[9] != ' ')
                dest[j++] = '.';

        for (i = 9; i < 12; i++)
        {
                if (source[i] == ' ') break;
                dest[j++] = source[i];
        }
        dest[j] = '\0';
        return dest;
}

int haswild(fname)
char *fname;
{
        char c;

        while (c = *fname++)
                if (c == '*' || c == '?')
                        return TRUE;
        return FALSE;
}

int match(wildnam, filnam, cur_drive)
char *wildnam, *filnam, cur_drive;
{
   char c;

   if (wildnam[1] != ':')
   {
        if (filnam[1] == ':')
                if (filnam[0] - 'A' == cur_drive)
                        filnam += 2;
                else
                        return FALSE;
   }
   else
   {
        if (filnam[1] != ':')
                if (wildnam[0] - 'A' == cur_drive)
                        wildnam += 2;
```

```
            else
                    return FALSE;
    }

    while (c = *wildnam++)
        if (c == '?')
                if ((c = *filnam++) && c != '.')
                        continue;
                else
                        return FALSE;
        else if (c == '*')
        {
                while (c = *wildnam)
                {       wildnam++;
                        if (c == '.') break;
                }
                while (c = *filnam)
                {       filnam++;
                        if (c == '.') break;
                }
        }
        else if (c == *filnam++)
                continue;
        else return FALSE;

    if (!*filnam)
        return TRUE;
    else
        return FALSE;
}
```

```
/*-------------------------------------------------------------------*/
/*                                                                   */
/*  This is a library of private routines for use with BDS C prog-   */
/*  rams.  The comment lines preceding each entry are intended       */
/*  to give a sufficient explanation of the routine that follows.    */
/*  To link any of these routines to a BDS C program, merely name    */
/*  PRVLIB as a argument following the name of the main program in   */
/*  the CLINK command line.                                          */
/*                                                                   */
/*-------------------------------------------------------------------*/


/*
    Move k bytes from blk1 to blk2.
    The two blocks may overlap.
    Since k must be positive, this routine is limited to
    moving blocks less than 32k in length.
    Added by M. Goldberg, 25-DEC-79.
*/
movblk(blk1, blk2, k)
    char *blk1, *blk2;
    int k;
    {
    int m,n,t,u;
    if ((k <= 0) || (!(t = blk1 - blk2))) return;
    if (t > 0) {m = 0; n = k;}
    else {m = 1 - k; n = 1;}
    for (t = m; t < n; ++t)
        {
        u = (t < 0 ? -t : t);
        *(blk2 + u) = *(blk1 + u);
        }
    }


/*
    ASCII counter -- increments a field of ASCII digits by one.
    Arguments are a pointer to the field (high-order digit)
    and the length of the field.
    The routine stops if it encounters a non-digit character
    in the field.
    Added by M. Goldberg, 25-DEC-79.
*/
asc_cntr(addr, len)
    char *addr;
    int len;
    {
    addr += len;
    do
        {
        if (!isdigit(*(--addr))) break;
        if (++(*addr) <= '9') break;
        *addr = '0';
        }
        while (--len);
    }


/*
    Sends a CR-LF pair to the CP/M LIST device.
```

```
    Added by M. Goldberg, 25-DEC-79.
*/
#define    LF          0x0A
#define    CR          0x0D
newline()
    {
    bdos(5, CR); bdos(5,LF);
    }


/*
    Sends a line of dashes to the CP/M LIST device.
    The argument is the number of dashes in the line.
    Added by M. Goldberg, 16-FEB-80.
*/
dashes(n)
    char n;
    {
    char i;
    for (i = 0; i < n; ++i) bdos(5, '-');
    newline();
    }


/*
    Causes a block of bytes to be displayed at the CP/M
    console device as a vector of two-digit hex numbers.
    Spaces are used to separate one hex number from another.
    It was written as a debug aid, that is, to be used to take
    a snapshot of a memory during program execution.

    The arguments are:
        blkp = a pointer to the beginning of the memory block
    and
        n = the number of bytes in the block.

    Added by M. Goldberg, 6-MAR-80.
*/
puthx(blkp, n)
    char *blkp;
    int n;
    {
    char c;
    while (n-- > 0)
        {
        prhd(((c = *blkp++) & 0xF0) >> 4);
        prhd(c & 0x0F);
        putchar(' ');
        }
    }


/*
    Outputs a message to the CP/M console device and
    stops the program.  The argument is a pointer to the
    message string.
    Added by M. Goldberg, 15-MAR-80.
*/
stop(mss)
    char mss[];
```

```
{
puts(msg);
exit();
}
```

```
/*
        USERCODE.C: A Nice Idea Killed By A Stupid CP/M MisFeature.....

        Idea: Extend the filename syntax for user with ALL file I/O to
              allow a user area prefix of the form "n/" on all filenames.

        Written by Leor Zolman, 12/81

        ********************************************************************
        **      FOR CP/M 2.x SYSTEMS ONLY!!!                              **
        ********************************************************************


        Generalized replacements for "open", "creat" and "unlink"
        library functions, allowing a user area prefix to be attached
        to all filenames (except those used as arguments to the "rename"
        function). The new filename syntax becomes:

                [whitespace][nn/][d:][filename.ext]

        E.g, to reference file "foo.bar" on the currently logged disk in
        user area 7, you'd use:

                7/foo.bar

        To reference foo.bar in user area 9 on disk b:, you'd say:

                9/b:foo.bar

        and so on. The user area prefix must always come first if both it and
        a disk designator need to be specified.

        NOTE: THIS WHOLE THING DOESN'T REALLY WORK FOR WRITING FILES INTO
        USER AREAS DIFFERENT FROM THE CURRENTLY ACTIVE USER AREA, BECAUSE
        GODDAMN CP/M DOESN'T LET YOU CLOSE A FILE THAT WAS OPENED IN A USER
        AREA DIFFERENT FROM THE CURRENTLY ACTIVE ONE. DAMN!!!!!!!!!!!!!!!!!!!

        To install this library, follow these steps:
                1) compile this file (USERCODE.C)
                2) invoke CLIB and give it the following commands:
                        *o 0 usercode
                        *o 1 deff2
                        *e 1 open
                        *a 0 open_old
                        *e 1 creat
                        *a 0 creat_old
                        *e 1 unlink
                        *a 0 unlink_old
                        *c 0
                        *q
                3) Link the programs you wish to have recognize the user code
                   on filenames by including "-f usercode" on the CLINK
                   command line.
*/

int open_old();
int creat_old();
int unlink_old();

open(filename, mode)
```

```
{
        return usercode(&open_old,filename,mode);
}

creat(filename)
{
        return usercode(&creat_old,filename);
}

unlink(filename)
{
        return usercode(&unlink_old,filename);
}

int usercode(funcptr, filename, extra_args)
int (*funcptr)();
char *filename;
int extra_args;
{
        int i, cur_user, new_user;
        char *savnam;

        while (isspace(*filename)) filename++;  /* skip over whitespace */
        savnam = filename;              /* save in case of false start  */

        if (!isdigit(*filename)) return (*funcptr)(filename,extra_args);

        cur_user = bdos(32, 0xff);      /* save current user number     */
        new_user = atoi(filename);      /* get new user number          */
        while (isdigit(*++filename))    /* skip over user number text   */
                ;
        if (*filename != ':' || new_user > 31)
                return (*funcptr)(savnam,extra_args);
        bdos(32,new_user);
        i = (*funcptr)(filename + 1,extra_args);
        bdos(32,cur_user);
        return i;
}
```

```
                ;
                ; CCC.ASM (C.CCC)  V1.45                              11/22/81
                ;
                ; NOTE: IF YOU ARE RUNNING UNDER MP/M II, BE SURE TO SET THE MPM2
                ; EQUATE TO 1.
                ;
                ; THIS IS THE BDS C RUN-TIME PACKAGE. NORMALLY, IT RESIDES AT
                ;    THE START OF THE TPA (AT ADDRESS BASE+100H, WHERE BASE IS EITHER
                ;    0000H OR 4200H DEPENDING ON CP/M IMPLEMENTATION.) THE CODE
                ;    GENERATED BY THE COMPILER ALWAYS SITS IMMEDIATELY AFTER THE END OF
                ;    THIS RUN-TIME PACKAGE CODE.
                ;
                ; EQUATE STATEMENTS IN CAPITAL LETTERS MAY BE CUSTOMIZED BY THE
                ;    USER IN ORDER TO CHANGE A) THE ORIGIN OF THE RUN-TIME PACKAGE,
                ;    AND B) THE ORIGIN OF THE RUN-TIME RAM AREA. IF YOU WILL BE
                ;    GENERATING CODE TO RUN IN A NON-CP/M ENVIRONMENT, SET THE CPM
                ;    EQUATE TO ZERO AND MAKE SURE TO SET THE ORIGIN, RAM AND
                ;    EXITAD EQUATES TO FIT YOUR CUSTOM RUN-TIME CONFIGURATION.
                ;
                ; THE "LXI SP,0" INSTRUCTION AT THE START IS REPLACED BY THE SEQUENCE:
                ;
                ;               LHLD    BASE+6
                ;               SPHL
                ;
                ;    BY CLINK AT LINK TIME, UNLESS THE -T OPTION IS USED WITH CLINK,
                ;    IN WHICH CASE THE "LXI     SP" REMAINS THERE AND THE VALUE USED TO
                ;    INITIALIZE THE SP IS THE ARGUMENT GIVEN TO THE "-T" OPTION.
                ;

                        TITLE 'BDS C Run-Time Module (c.ccc)  v1.45  11/22/81'


0001 =          CPM:    EQU     1       ;TRUE IF TO BE RUN UNDER CP/M OR MP/M

0000 =          MPM2:   EQU     0       ;TRUE ONLY IF RUNNING UNDER MP/M II

0000 =          DMAVIO: EQU     0       ;TRUE IF USING DMA VIDEO LIBRARY ROUTINES AND
                                        ;NEED PARAMETERS INITIALIZED
                        IF CPM
0000 =          BASE:   EQU     0       ;START OF RAM IN SYSTEM (EITHER 0 OR 4200H FOR CP/M)
0005 =          BDOS:   EQU     BASE+5  ;REST OF THESE USED BY CP/M-BASED CONFIGURATIONS.
0100 =          TPA:    EQU     BASE+100H
0008 =          NFCBS:  EQU     8       ;MAXIMUM # OF FILES OPEN AT ONE TIME
0080 =          TBUFF:  EQU     BASE+80H
0100 =          ORIGIN: EQU     TPA
0000 =          EXITAD: EQU     BASE    ;WARM BOOT LOCATION
                        ENDIF

                        IF NOT CPM              ;FILL IN THE APPROPRIATE VALUES...
                ORIGIN: EQU     NEWBASE         ;ADDRESS AT WHICH PROGRAMS ARE TO RUN
                RAM:    EQU     WHATEVER        ;R/W MEMORY AREA FOR NON-CP/M CONFIGURATIONS
                                                ;  (DEFAULT: JUST AFTER C.CCC UNDER CP/M)
                EXITAD: EQU     WHENDONE        ;WHERE TO GO WHEN DONE EXECUTING
                        ENDIF
```

```
                ;
                ; THE LOCATION OF THE JUMP VECTORS AND UTILITY ROUTINES MUST REMAIN
                ; CONSTANT RELATIVE TO THE BEGINNING OF THIS RUN-TIME MODULE.
                ;
                ; DO NOT CHANGE ANYTHING BETWEEN HERE AND THE START OF THE
                ; "INIT" ROUTINE!!!!!!!!
                ;

0100                    ORG     ORIGIN
0100 310000             LXI     SP,0    ;THIS IS CHANGED BY CLINK TO LHLD       BASE+5H
0103 00                 NOP             ;THIS FIRST IS USUALLY TURNED INTO SPHL BY CLINK

0104 0000               NOP! NOP        ;SIMPLE INITIALIZATION OR PATCHES MAY BE
0106 000000             NOP! NOP! NOP   ;INSERTED HERE, BUT BETTER TO DO ALL THAT
0109 000000             NOP! NOP! NOP   ;IN THE "INIT" ROUTINE

010C CD4B03             CALL    INIT    ;DO ARGC & ARGV PROCESSING, PLUS MISC. INITIALIZATIONS
010F CD8607             CALL    MAIN    ;GO CRUNCH!!!!
0112 C31304             JMP     VEXIT   ;CLOSE OPEN FILES AND REBOUT

0115            EXTRNS: DS      2               ;SET BY CLINK TO EXTERNAL DATA BASE ADDRESS
0117 8606       CCCSIZ: DW      MAIN-ORIGIN     ;SIZE OF THIS CODE (FOR USE BY CLINK)
0119            CODEND: DS      2               ;SET BY CLINK TO (LAST ADDR OF CODE + 1)
011B            FRERAM: DS      2               ;SET BY CLINK TO (LAST ADDR OF EXTERNALS + 1)

                ;
                ; JUMP VECTORS TO SOME FILE I/O UTILITY ROUTINES:
                ;

011D C30F04     ERROR:  JMP     VERROR  ;LOADS -1 INTO HL AND RETURNS
0120 C31304     EXIT:   JMP     VEXIT   ;CLOSE ALL OPEN FILES AND REBOOT

                        IF      CPM
0123 C32E04     CLOSE:  JMP     VCLOSE  ;CLOSE A FILE
0126 C37604     SETFCB: JMP     VSETFCB ;SET UP FCB AT HL GIVEN FILENAME AT DE
0129 C35A04     FGFD:   JMP     VFGFD   ;RETURN C SET IF FILE FD IN A NOT OPEN
012C C30205     FGFCB:  JMP     VFGFCB  ;COMPUTE ADDRESS OF INTERNAL FCB FOR FD IN A
                        ENDIF

                        IF      NOT CPM ;IF NOT UNDER CP/M, FILE I/O ROUTINES
                CLOSE:  JMP     VERROR  ;ARE NOT USED.
                SETFCB: JMP     VERROR
                FGFD:   JMP     VERROR
                FGFCB:  JMP     VERROR
                        ENDIF

012F                    DS      16      ;RESERVED

                        IF      CPM
                SETFCB3:
013F 77                 MOV     M,A     ;THIS IS A PATCH FROM THE "VSETFCB" ROUTINE,
0140 23                 INX     H       ;WHICH CAUSES THE RANDOM RECORD BYTES OF THE
0141 77                 MOV     M,A     ;FCB BEING INITIALIZED TO BE ZEROED. (FORMER
0142 23                 INX     H       ;VERSIONS HAD A "DS 30" ABOVE, SO THIS KEEPS
0143 77                 MOV     M,A     ;ALL THE ADDRESSES CONSISTENT BETWEEN THIS
0144 D1                 POP     D       ;AND EARLIER 1.4'S)
```

```
0145 C1             POP     B
0146 C9             RET

                PATCHNM:
0147 CDAE04         CALL    SETNM   ;ANOTHER PATCH FROM "VSETFCB"
014A C3C604         JMP     SETNM3
                    ENDIF

                    IF      NOT CPM
                    DS      14      ;KEEP ADDRESSES THE SAME FOR NON-CP/M IMPLEMENTATIONS
                    ENDIF

                ;
                ; THE FOLLOWING ROUTINES FETCH A VARIABLE VALUE FROM EITHER
                ; THE LOCAL STACK FRAME OR THE EXTERNAL AREA, GIVEN THE RELATIVE
                ; OFFSET OF THE DATUM REQUIRED IMMEDIATELY FOLLOWING THE CALL:
                ; FOR THE "LONG DISPLACEMENT" ROUTINES, THE OFFSET MUST BE 16 BITS,
                ; FOR THE "SHORT DISPLACEMENT" ROUTINES, THE OFFSET MUST BE 8 BITS.
                ;

                ;
                ; LONG-DISPLACEMENT, DOUBLE-BYTE EXTERNAL INDIRECTION:
                ;
                ;       FORMAT: CALL LDEI               ; GET 16-BIT VALUE IN HL
                ;               DW OFFSET_FROM_EXTRNS   ; >= 256
                ;

014D E1         LDEI:   POP     H       ;GET ADDRESS OF OFFSET
014E 5E                 MOV     E,M     ;PUT OFFSET IN DE
014F 23                 INX     H
0150 56                 MOV     D,M
0151 23                 INX     H
0152 E5                 PUSH    H       ;SAVE RETURN ADDRESS
0153 2A1501             LHLD    EXTRNS  ;ADD OFFSET TO EXTERNAL AREA BASE
0156 19                 DAD     D
0157 7E                 MOV     A,M     ;AND GET THE VALUE INTO HL
0158 23                 INX     H
0159 66                 MOV     H,M
015A 6F                 MOV     L,A
015B C9                 RET

                ;
                ; SHORT-DISPLACEMENT, DOUBLE-BYTE EXTERNAL INDIRECTION:
                ;
                ;       FORMAT:         CALL SDEI               ; GET 16-BIT VALUE IN L
                ;                       DB OFFSET_FROM_EXTRNS   ; < 256
                ;

015C E1         SDEI:   POP     H
015D 5E                 MOV     E,M
015E 23                 INX     H
015F E5                 PUSH    H
0160 1600               MVI     D,0
0162 2A1501             LHLD    EXTRNS
0165 19                 DAD     D
0166 7E                 MOV     A,M
```

```
0167 23             INX     H
0168 66             MOV     H,M
0169 6F             MOV     L,A
016A C9             RET

                    ;
                    ; LONG-DISPLACEMENT, SINGLE-BYTE EXTERNAL INDIRECTION:
                    ;
                    ;       FORMAT:         CALL    LSEI            ; GET 8-BIT VALUE IN L
                    ;                       DW OFFSET_FROM_EXTRNS   ; >= 256
                    ;

016B E1         LSEI:   POP     H
016C 5E             MOV     E,M
016D 23             INX     H
016E 56             MOV     D,M
016F 23             INX     H
0170 E5             PUSH    H
0171 2A1501         LHLD    EXTRNS
0174 19             DAD     D
0175 6E             MOV     L,M
0176 C9             RET

                    ;
                    ; SHORT-DISPLACEMENT, SINGLE-BYTE EXTERNAL INDIRECTION:
                    ;
                    ;       FORMAT:         CALL    SSEI            ; GET 8-BIT VALUE IN L
                    ;                       DB OFFSET_FROM_EXTERNS  ; < 256
                    ;

0177 E1         SSEI:   POP     H
0178 5E             MOV     E,M
0179 23             INX     H
017A E5             PUSH    H
017B 1600           MVI     D,0
017D 2A1501         LHLD    EXTRNS
0180 19             DAD     D
0181 6E             MOV     L,M
0182 C9             RET

                    ;
                    ; LONG-DISPLACEMENT, DOUBLE-BYTE LOCAL INDIRECTION:
                    ;
                    ;       FORMAT:         CALL    LDLI            ; GET 16-BIT VALUE IN HL
                    ;                       DW OFFSET_FROM_BC       ; >= 256
                    ;

0183 E1         LDLI:   POP     H
0184 5E             MOV     E,M
0185 23             INX     H
0186 56             MOV     D,M
0187 23             INX     H
0188 E5             PUSH    H
0189 EB             XCHG
018A 09             DAD     B
018B 7E             MOV     A,M
```

```
018C 23             INX     H
018D 66             MOV     H,M
018E 6F             MOV     L,A
018F C9             RET

                    ;
                    ; SHORT-DISPLACEMENT, DOUBLE-BYTE LOCAL INDIRECTION:
                    ;
                    ;       FORMAT:         CALL    SDLI            ; GET 16-BIT VALUE IN HL
                    ;                       DB OFFSET_FROM_BC       ; < 256
                    ;

0190 E1         SDLI:   POP     H
0191 5E                 MOV     E,M
0192 23                 INX     H
0193 E5                 PUSH    H
0194 EB                 XCHG
0195 2600               MVI     H,0
0197 09                 DAD     B
0198 7E                 MOV     A,M
0199 23                 INX     H
019A 66                 MOV     H,M
019B 6F                 MOV     L,A
019C C9                 RET

                    ;
                    ; FLAG CONVERSION ROUTINES:
                    ;

019D 210100     PZINH:  LXI     H,1     ;RETURN HL = TRUE IF Z SET
01A0 C8                 RZ
01A1 2B                 DCX     H
01A2 C9                 RET

01A3 210000     PNZINH: LXI     H,0     ;RETURN HL = FALSE IF Z SET
01A6 C8                 RZ
01A7 23                 INX     H
01A8 C9                 RET

01A9 210100     PCINH:  LXI     H,1     ;RETURN HL = TRUE IF C SET
01AC D8                 RC
01AD 2B                 DCX     H
01AE C9                 RET

01AF 210000     PNCINH: LXI     H,0     ;RETURN HL = FALSE IF C SET
01B2 D8                 RC
01B3 23                 INX     H
01B4 C9                 RET

01B5 210100     PPINH:  LXI     H,1     ;RETURN HL = TRUE IF P (PLUS) FLAG SET
01B8 F0                 RP
01B9 2B                 DCX     H
01BA C9                 RET

01BB 210100     PMINH:  LXI     H,1     ;RETURN HL = TRUE IF M (MINUS) FLAG SET
01BE F8                 RM
```

```
01BF 2B             DCX     H
01C0 C9             RET

01C1 110100   PZIND:  LXI     D,1     ;RETURN DE = TRUE IF Z SET
01C4 C8             RZ
01C5 1B             DCX     D
01C6 C9             RET

01C7 110000   PNZIND: LXI     D,0     ;RETURN DE = FALSE IF Z SET
01CA C8             RZ
01CB 13             INX     D
01CC C9             RET

01CD 110100   PCIND:  LXI     D,1     ;RETURN DE = TRUE IF C SET
01D0 D8             RC
01D1 1B             DCX     D
01D2 C9             RET

01D3 110000   PNCIND: LXI     D,0     ;RETURN DE = FALSE IF C SET
01D6 D8             RC
01D7 13             INX     D
01D8 C9             RET

01D9 110100   PPIND:  LXI     D,1     ;RETURN DE = TRUE IF P (PLUS) FLAG SET
01DC F0             RP
01DD 1B             DCX     D
01DE C9             RET

01DF 110100   PMIND:  LXI     D,1     ;RETURN DE = TRUE IF M (MINUS) FLAG SET
01E2 F8             RM
01E3 1B             DCX     D
01E4 C9             RET


              ;
              ; RELATIONAL OPERATOR ROUTINES: TAKE ARGS IN DE AND HL,
              ; AND RETURN A FLAG BIT EITHER SET OR RESET.
              ;
              ; ==, >, < :
              ;

01E5 7D       EQWEL:  MOV     A,L     ;RETURN Z IF HL == DE, ELSE NZ
01E6 BB             CMP     E
01E7 C0             RNZ             ;IF L <> E, THEN HL <> DE
01E8 7C             MOV     A,H     ;ELSE HL == DE ONLY IF H == D
01E9 BA             CMP     D
01EA C9             RET

01EB EB       BLAU:   XCHG            ;RETURN C IF HL < DE, UNSIGNED
01EC 7A       ALBU:   MOV     A,D     ;RETURN C IF DE < HL, UNSIGNED
01ED BC             CMP     H
01EE C0             RNZ             ;IF D <> H, C IS SET CORRECTLY
01EF 7B             MOV     A,E     ;ELSE COMPARE E WITH L
01F0 BD             CMP     L
01F1 C9             RET
```

```
01F2 EB       BGAU:  XCHG           ;RETURN C IF HL > DE, UNSIGNED
01F3 7C       AGBU:  MOV    A,H     ;RETURN C IF DE > HL, UNSIGNED
01F4 BA              CMP    D
01F5 C0              RNZ            ;IF H <> D, C IS SET CORRECTLY
01F6 7D              MOV    A,L     ;ELSE COMPARE L WITH E
01F7 BB              CMP    E
01F8 C9              RET

01F9 EB       BLAS:  XCHG           ;RETURN C IF HL < DE, SIGNED
01FA 7C       ALBS:  MOV    A,H     ;RETURN C IF DE < HL, SIGNED
01FB AA              XRA    D
01FC F2EC01          JP ALBU        ;IF SAME SIGN, DO UNSIGNED COMPARE
01FF 7A              MOV    A,D
0200 B7              ORA    A
0201 F0              RP             ;ELSE RETURN NC IF DE IS POSITIVE AND HL IS NEGATIVE
0202 37              STC            ;ELSE SET CARRY, SINCE DE IS NEGATIVE AND HL IS POS.
0203 C9              RET

0204 EB       BGAS:  XCHG           ;RETURN C IF HL > DE, SIGNED
0205 7C       AGBS:  MOV    A,H     ;RETURN C IF DE > HL, SIGNED
0206 AA              XRA    D
0207 F2F301          JP     AGBU    ;IF SAME SIGN, GO DO UNSIGNED COMPARE
020A 7C              MOV    A,H
020B B7              ORA    A
020C F0              RP             ;ELSE RETURN NC IS HL IS POSITIVE AND DE IS NEGATIVE
020D 37              STC
020E C9              RET            ;ELSE RETURN C, SINCE HL IS NEG AND DE IS POS


                     ;
                     ; MULTIPLICATIVE OPERATORS: *, /, AND %:
                     ;

020F 7A       SMOD:  MOV    A,D     ;SIGNED MOD ROUTINE: RETURN (DE % HL) IN HL
0210 F5              PUSH   PSW     ;SAVE HIGH BIT OF DE AS SIGN OF RESULT
0211 CD5A02          CALL   TSTN    ;GET ABSOLUTE VALUE OF ARGS
0214 EB              XCHG
0215 CD5A02          CALL   TSTN
0218 EB              XCHG
0219 CD2902          CALL   USMOD   ;DO UNSIGNED MOD
021C F1              POP    PSW     ;WAS DE NEGATIVE?
021D B7              ORA    A       ;IF NOT,
021E F0              RP             ;       ALL DONE
021F 7C              MOV    A,H     ;ELSE MAKE RESULT NEGATIVE
0220 2F              CMA
0221 67              MOV    H,A
0222 7D              MOV    A,L
0223 2F              CMA
0224 6F              MOV    L,A
0225 23              INX    H
0226 C9              RET

0227 00              NOP            ;MAINTAIN ADDRESS COMPATIBILITY WITH SOME
0228 00              NOP            ; PRE-RELEASE V1.4'S.

0229 7C       USMOD: MOV    A,H     ;UNSIGNED MOD: RETURN (DE % HL) IN HL
```

```
022A B5             ORA   L
022B C8             RZ
022C D5             PUSH  D
022D E5             PUSH  H
022E CD8902         CALL  USDIV
0231 D1             POP   D
0232 CD6B02         CALL  USMUL
0235 7C             MOV   A,H
0236 2F             CMA
0237 67             MOV   H,A
0238 7D             MOV   A,L
0239 2F             CMA
023A 6F             MOV   L,A
023B 23             INX   H
023C D1             POP   D
023D 19             DAD   D
023E C9             RET

023F AF       SMUL: XRA   A       ;SIGNED MULTIPLY: RETURN (DE * HL) IN HL
0240 325905         STA   TMP
0243 CD5A02         CALL  TSTN
0246 EB             XCHG
0247 CD5A02         CALL  TSTN
024A CD6B02         CALL  USMUL
024D 3A5905   SMUL2: LDA   TMP
0250 1F             RAR
0251 D0             RNC
0252 7C             MOV   A,H
0253 2F             CMA
0254 67             MOV   H,A
0255 7D             MOV   A,L
0256 2F             CMA
0257 6F             MOV   L,A
0258 23             INX   H
0259 C9             RET

025A 7C       TSTN: MOV   A,H
025B B7             ORA   A
025C F0             RP
025D 2F             CMA
025E 67             MOV   H,A
025F 7D             MOV   A,L
0260 2F             CMA
0261 6F             MOV   L,A
0262 23             INX   H
0263 3A5905         LDA   TMP
0266 3C             INR   A
0267 325905         STA   TMP
026A C9             RET

026B C5       USMUL: PUSH  B       ;UNSIGNED MULTIPLY: RETURN (DE * HL) IN HL
026C CD7102         CALL  USM2
026F C1             POP   B
0270 C9             RET

0271 44       USM2: MOV   B,H
```

```
0272 4D          MOV   C,L
0273 210000      LXI   H,0
0276 78    USM3: MOV   A,B
0277 B1          ORA   C
0278 C8          RZ
0279 78          MOV   A,B
027A 1F          RAR
027B 47          MOV   B,A
027C 79          MOV   A,C
027D 1F          RAR
027E 4F          MOV   C,A
027F D28302      JNC   USM4
0282 19          DAD   D
0283 EB    USM4: XCHG
0284 29          DAD   H
0285 EB          XCHG
0286 C37602      JMP   USM3

0289 7C    USDIV: MOV   A,H     ;UNSIGNED DIVIDE: RETURN (DE / HL) IN HL
028A B5           ORA   L       ;RETURN 0 IF HL IS 0
028B C8           RZ
028C C5           PUSH  B
028D CD9402       CALL  USD1
0290 60           MOV   H,B
0291 69           MOV   L,C
0292 C1           POP   B
0293 C9           RET


0294 0601  USD1:  MVI   B,1
0296 7C    USD2:  MOV   A,H
0297 B7           ORA   A
0298 FAA002       JM    USD3
029B 29           DAD   H
029C 04           INR   B
029D C39602       JMP   USD2

02A0 EB    USD3:  XCHG

02A1 78    USD4:  MOV   A,B
02A2 010000       LXI   B,0
02A5 F5    USD5:  PUSH  PSW
02A6 CDDD02 USD6: CALL  CMPHD
02A9 DAB702       JC    USD7
02AC 03           INX   B
02AD D5           PUSH  D
02AE 7A           MOV   A,D
02AF 2F           CMA
02B0 57           MOV   D,A
02B1 7B           MOV   A,E
02B2 2F           CMA
02B3 5F           MOV   E,A
02B4 13           INX   D
02B5 19           DAD   D
02B6 D1           POP   D
02B7 AF    USD7:  XRA   A
```

```
02B8 7A             MOV     A,D
02B9 1F             RAR
02BA 57             MOV     D,A
02BB 7B             MOV     A,E
02BC 1F             RAR
02BD 5F             MOV     E,A
02BE F1             POP     PSW
02BF 3D             DCR     A
02C0 C8             RZ
02C1 F5             PUSH    PSW
02C2 79             MOV     A,C
02C3 17             RAL
02C4 4F             MOV     C,A
02C5 78             MOV     A,B
02C6 17             RAL
02C7 47             MOV     B,A
02C8 C3A602         JMP     USD6

02CB AF     SDIV:   XRA     A       ;SIGNED DIVIDE: RETURN (DE / HL) IN HL
02CC 325905         STA     TMP
02CF CD5A02         CALL    TSTN
02D2 EB             XCHG
02D3 CD5A02         CALL    TSTN
02D6 EB             XCHG
02D7 CD8902         CALL    USDIV
02DA C34D02         JMP     SMUL2

02DD 7C     CMPHD:  MOV     A,H     ;THIS RETURNS C IF HL < DE
02DE BA             CMP     D       ; (UNSIGNED COMPARE ONLY USED
02DF D8             RC              ;  WITHIN C.CCC, NOT FROM C)
02E0 C0             RNZ
02E1 7D             MOV     A,L
02E2 BB             CMP     E
02E3 C9             RET

                    ;
                    ; SHIFT OPERATORS  << AND >>:
                    ;

02E4 EB     SDERBL: XCHG            ;SHIFT DE RIGHT BY L BITS
02E5 1C     SHLRBE: INR     E       ;SHIFT HL RIGHT BY E BITS
02E6 1D     SHRBE2: DCR     E
02E7 C8             RZ
02E8 AF             XRA     A
02E9 7C             MOV     A,H
02EA 1F             RAR
02EB 67             MOV     H,A
02EC 7D             MOV     A,L
02ED 1F             RAR
02EE 6F             MOV     L,A
02EF C3E602         JMP     SHRBE2

02F2 EB     SDELBL: XCHG            ;SHIFT DE LEFT BY L BITS
02F3 1C     SHLLBE: INR     E       ;SHIFT HL LEFT BY E BITS
02F4 1D     SHLBE2: DCR     E
02F5 C8             RZ
```

```
02F6 29             DAD     H
02F7 C3F402         JMP     SHLBE2


                    ;
                    ; ROUTINES TO 2'S COMPLEMENT HL AND DE:

02FA 7C         CMH:    MOV     A,H
02FB 2F                 CMA
02FC 67                 MOV     H,A
02FD 7D                 MOV     A,L
02FE 2F                 CMA
02FF 6F                 MOV     L,A
0300 23                 INX     H
0301 C9                 RET

0302 7A         CMD:    MOV     A,D
0303 2F                 CMA
0304 57                 MOV     D,A
0305 7B                 MOV     A,E
0306 2F                 CMA
0307 5F                 MOV     E,A
0308 13                 INX     D
0309 C9                 RET


                    ;
                    ; THE FOLLOWING ROUTINES YANK A FORMAL PARAMETER VALUE OFF THE STACK
                    ; AND PLACE IT IN BOTH HL AND A (LOW BYTE), ASSUMING THE CALLER
                    ; HASN'T DONE ANYTHING TO ITS STACK POINTER SINCE IT WAS CALLED.
                    ;
                    ; THE MNEMONICS ARE "MOVE ARG #N TO HL",
                    ; WHERE ARG #1 IS THE THIRD THING ON THE STACK (WHERE THE FIRST
                    ; AND SECOND THINGS ARE, RESPECTIVELY, THE RETURN ADDRESS OF THE
                    ; ROUTINE MAKING THE CALL       TO HERE, AND THE PREVIOUS RETURN
                    ; ADDRESS TO THE ROUTINE WHICH ACTUALLY PUSHED THE ARGS ON THE
                    ; STACK.) THUS, A CALL  TO "MA1TOH" WOULD RETURN WITH THE FIRST
                    ; PASSED PARAMETER IN HL AND A; "MA2TOH" WOULD RETURN THE SECOND,
                    ; ETC. NOTE THAT IF THE CALLER HAS PUSHED [N] ITEMS ON THE STACK
                    ; BEFORE CALLING "MA [X] TOH", THEN THE [X-N]TH FORMAL PARAMETER
                    ; VALUE WILL BE RETURNED, NOT THE [X]TH.
                    ;

030A 210400     MA1TOH: LXI     H,4     ;GET FIRST ARG
030D 39         MA0TOH: DAD     SP
030E 7E                 MOV     A,M
030F 23                 INX     H
0310 66                 MOV     H,M
0311 6F                 MOV     L,A
0312 C9                 RET

0313 210600     MA2TOH: LXI     H,6     ;GET 2ND ARG
0316 C30D03             JMP     MA0TOH

0319 210800     MA3TOH: LXI     H,8     ;GET 3RD ARG
031C C30D03             JMP     MA0TOH
```

```
031F 210A00    MA4TOH: LXI     H,10    ;GET 4TH ARG
0322 C30D03            JMP     MAOTOH

0325 210C00    MA5TOH: LXI     H,12    ;GET 5TH ARG
0328 C30D03            JMP     MAOTOH

032B 210E00    MA6TOH: LXI     H,14    ;GET 6TH ARG
032E C30D03            JMP     MAOTOH

0331 211000    MA7TOH: LXI     H,16    ;GET 7TH ARG
0334 C30D03            JMP     MAOTOH

               ;
               ; THIS ROUTINE TAKES THE FIRST 7 ARGS ON THE STACK
               ; AND PLACES THEM CONTIGUOUSLY AT THE "ARGS" RAM AREA.
               ; THIS ALLOWS A LIBRARY ROUTINE TO MAKE ONE CALL        TO ARGHAK
               ; AND HENCEFORTH HAVE ALL IT'S ARGS AVAILABLE DIRECTLY
               ; THROUGH LHLD'S INSTEAD OF HAVING TO HACK THE STACK AS IT
               ; GROWS AND SHRINKS. NOTE THAT ARGHAK SHOULD BE CALLED AS THE
               ; VERY FIRST THING A FUNCTION DOES, BEFORE EVEN PUSHING BC.
               ;

0337 118705    ARGHAK: LXI     D,ARGS  ;DESTINATION FOR BLOCK MOVE IN DE
033A 210400            LXI     H,4     ;PASS OVER TWO RETURN ADDRESS
033D 39                DAD     SP      ;SOURCE FOR BLOCK MOVE IN HL
033E C5                PUSH    B       ;SAVE BC
033F 060E              MVI     B,14    ;COUNTDOWN IN B
0341 7E        ARGHK2: MOV     A,M     ;COPY LOOP
0342 12                STAX    D
0343 23                INX     H
0344 13                INX     D
0345 05                DCR     B
0346 C24103            JNZ     ARGHK2
0349 C1                POP     B       ;RESTORE BC
034A C9                RET

               ;
               ; UP TO THIS POINT, ABSOLUTELY NO CHANGES SHOULD EVER BE MADE
               ; TO THIS SOURCE FILE (EXCEPT FOR CUSTOMIZING THE EQU STATEMENTS
               ; AT THE BEGINNING OF THE FILE).
               ;


               ;
               ; THIS ROUTINE IS CALLED FIRST TO DO ARGC & ARGV PROCESSING (IF
               ; RUNNING UNDER CP/M) AND SOME ODDS AND ENDS INITIALIZATIONS:
               ;

034B E1        INIT:   POP     H       ;STORE RETURN ADDRESS
034C 225B05            SHLD    TMP2    ; SOMEWHERE SAFE FOR THE TIME BEING

                       IF      CPM
034F 214807            LXI     H,ARGLST-2      ;SET THE "ARGV" THAT THE C MAIN PROGRAM
                       ENDIF
```

```
                        IF      NOT CPM
                        LXI     H,0
                        ENDIF

0352 E5                 PUSH    H               ; WILL GET.

                                        ;INITIALIZE STORAGE ALLOCATION POINTERS:
0353 2A1B01             LHLD    FRERAM  ;GET ADDRESS AFTER END OF EXTERNALS
0356 229B05             SHLD    ALLOCP  ;STORE AT ALLOCATION POINTER (FOR "SBRK.")
0359 21E803             LXI     H,1000  ;DEFAULT SAFETY SPACE BETWEEN STACK AND
035C 229D05             SHLD    ALOCMX  ; HIGHEST ALLOCATABLE ADDRESS IN MEMORY
                                        ; (FOR USE BY "SBRK".).

                                        ;INITIALIZE RANDOM SEED:
035F 21DC59             LXI     H,59DCH ;LET'S STICK SOMETHING WIERD INTO THE
0362 227F05             SHLD    RSEED   ;FIRST 16 BITS OF THE RANDOM-NUMBER SEED

                                        ;INITIALIZE I/O HACK LOCATIONS:
0365 3EDB               MVI     A,0DBH          ;"IN" OP, FOR "IN XX; RET" SUBROUTINE
0367 329505             STA     IOHACK
036A 3ED3               MVI     A,0D3H          ;"OUT" OP FOR "OUT XX; RET" SUBROUTINE
036C 329805             STA     IOHACK+3
036F 3EC9               MVI     A,0C9H          ;"RET" FOR ABOVE SOBROUTINES
0371 329705             STA     IOHACK+2        ;THE PORT NUMBER IS FILLED IN BY THE
0374 329A05             STA     IOHACK+5        ;"INP" AND "OUTP" LIBRARY ROUTINES.

                                                ;INITIALIZE DMA VIDEO PARAMETERS:
                        IF      DMAVIO          ;IF WE'RE USING DMA VIDEO ROUTINES,
                        LXI     H,0CC00H        ;SET UP DEFAULT VALUES (MAY BE CHANGED
                        SHLD    PBASE           ;TO WHATEVER SUITS). VIDEO BOARD ADDRESS,
                        LXI     H,16
                        SHLD    XSIZE           ;# OF LINES,
                        LXI     H,64
                        SHLD    YSIZE           ;# OF COLUMNS,
                        LXI     H,1024
                        SHLD    PSIZE           ;AND TOTAL # OF CHARACTERS ON SCREEN
                        ENDIF

                        IF      CPM     ;UNDER CP/M: CLEAR CONSOLE, PROCESS ARGC & ARGV:
0377 0E0B               MVI     C,11    ;INTERROGATE CONSOLE STATUS TO SEE IF THERE
0379 CD0500             CALL    BDOS    ;  HAPPENS TO BE A STRAY CHARACTER THERE...

037C B7                 ORA     A       ;(USED TO BE `ANI 1'...THEY TELL ME THIS WORKS
037D 00                 NOP             ; BETTER FOR CERTAIN BIZARRE CP/M-"LIKE" SYSTEMS)

037E CA8603             JZ      INITZZ
0381 0E01               MVI     C,1     ;IF INPUT PRESENT, CLEAR IT
0383 CD0500             CALL    BDOS

0386 218000     INITZZ: LXI     H,TBUFF         ;IF ARGUMENTS GIVEN, PROCESS THEM.
0389 11C706             LXI     D,COMLIN        ;GET READY TO COPY COMMAND LINE
038C 46                 MOV     B,M             ;FIRST GET LENGTH OF IT FROM LOC. BASE+80H
038D 23                 INX     H
038E 78                 MOV     A,B
038F B7                 ORA     A       ;IF NO ARGUMENTS, DON'T PARSE FOR ARGV
0390 C29903             JNZ     INITL
```

```
0393 110100          LXI   D,1     ;SET ARGC TO 1 IN SUCH A CASE.
0396 C3F703          JMP   I5

0399 7E      INITL:  MOV   A,M     ;OK, THERE ARE ARGUMENTS. PARSE...
039A 12              STAX  D       ;FIRST COPY COMMAND LINE TO COMLIN
039B 23              INX   H
039C 13              INX   D
039D 05              DCR   B
039E C29903          JNZ   INITL
03A1 AF              XRA   A       ;PLACE ZERO FOLLOWING LINE
03A2 12              STAX  D

03A3 21C706          LXI   H,COMLIN        ;NOW COMPUTE POINTERS TO EACH ARG
03A6 110100          LXI   D,1             ;ARG COUNT
03A9 014A07          LXI   B,ARGLST        ;WHERE POINTERS WILL ALL GO
03AC AF              XRA   A               ;CLEAR "IN A STRING" FLAG
03AD 325A05          STA   TMP1
03B0 7E      I2:     MOV   A,M     ;BETWEEN ARGS...
03B1 23              INX   H
03B2 FE20            CPI   ' '
03B4 CAB003          JZ    I2
03B7 B7              ORA   A
03B8 CAF703          JZ    I5      ;IF NULL BYTE, DONE WITH LIST
03BB FE22            CPI   '"'
03BD C2C603          JNZ   I2A     ;QUOTE?
03C0 325A05          STA   TMP1    ;YES. SET "IN A STRING" FLAG
03C3 C3C703          JMP   I2B

03C6 2B      I2A:    DCX   H
03C7 7D      I2B:    MOV   A,L     ;OK, HL IS A POINTER TO THE START
03C8 02              STAX  B       ;OF AN ARG STRING. STORE IT.
03C9 03              INX   B
03CA 7C              MOV   A,H
03CB 02              STAX  B
03CC 03              INX   B
03CD 13              INX   D       ;BUMP ARG COUNT
03CE 7E      I3:     MOV   A,M
03CF 23              INX   H       ;PASS OVER TEXT OF THIS ARG
03D0 B7              ORA   A       ;IF AT END, ALL DONE
03D1 CAF703          JZ    I5
03D4 C5              PUSH  B       ;IF TMP1 SET, IN A STRING
03D5 47              MOV   B,A     ; (SO WE HAVE TO IGNORE SPACES)
03D6 3A5A05          LDA   TMP1
03D9 B7              ORA   A
03DA 78              MOV   A,B
03DB C1              POP   B
03DC CAEB03          JZ    I3A
03DF FE22            CPI   '"'     ;WE ARE IN A STRING.
03E1 C2CE03          JNZ   I3      ;CHECK FOR TERMINATING QUOTE
03E4 AF              XRA   A       ;IF FOUND, RESET "IN STRING" FLAG
03E5 325A05          STA   TMP1
03E8 2B              DCX   H
03E9 77              MOV   M,A     ;AND STICK A ZERO BYTE AFTER THE STRING
03EA 23              INX   H       ;AND GO ON TO NEXT ARG
03EB FE20    I3A:    CPI   ' '     ;NOW FIND THE SPACE BETWEEN ARGS
03ED C2CE03          JNZ   I3
```

```
03F0 2B             DCX   H       ;FOUND IT. STICK IN A ZERO BYTE
03F1 3600           MVI   M,0
03F3 23             INX   H
03F4 C3B003         JMP   I2      ;AND GO ON TO NEXT ARG

03F7 D5       I5:   PUSH  D       ;ALL DONE FINDING ARGS. SET ARGC.

03F8 0608           MVI   B,NFCBS ;NOW INITIALIZE ALL THE FILE INFO
03FA 21BF06         LXI   H,FDT   ;(JUST ZERO THE FD TABLE)
03FD 3600     I6:   MVI   M,0
03FF 23             INX   H
0400 05             DCR   B
0401 C2FD03         JNZ   I6
                    ENDIF

                    IF    NOT CPM ;IF NOT UNDER CP/M, FORCE ARGC VALUE
                    LXI   H,1     ; OF ONE.
                    PUSH  H
                    ENDIF

0404 AF             XRA   A
0405 325F05         STA   UNGETL  ;CLEAR THE PUSH-BACK BYTE
0408 326005         STA   LASTC   ;AND LAST CHARACTER BYTE

040B 2A5B05         LHLD  TMP2
040E E9             PCHL          ;ALL DONE INITIALIZING.

                ;
                ; GENERAL PURPOSE ERROR VALUE RETURN ROUTINE:
                ;

040F 21FFFF     VERROR: LXI   H,-1    ;GENERAL ERROR HANDLER...JUST
0412 C9               RET           ;RETURNS -1 IN HL

                ;
                ; HERE ARE FILE I/O HANDLING ROUTINES, ONLY NEEDED UNDER CP/M:
                ;

                ;
                ; CLOSE ANY OPEN FILES AND REBOOT:
                ;

                VEXIT:
                      IF    CPM             ;IF UNDER CP/M, CLOSE ALL OPEN FILES
0413 3E0F             MVI   A,7+NFCBS       ;START WITH LARGEST POSSIBLE FD
0415 F5         EXIT1: PUSH PSW             ;AND SCAN ALL FD'S FOR OPEN FILES
0416 CD5A04           CALL  VFGFD           ;IS FILE WHOSE FD IS IN A OPEN?
0419 DA2404           JC    EXIT2           ;IF NOT, GO ON TO NEXT FD
041C 6F               MOV   L,A             ;ELSE CLOSE THE ASSOCIATED FILE
041D 2600             MVI   H,0
041F E5               PUSH  H
0420 CD2E04           CALL  VCLOSE
0423 E1               POP   H
0424 F1         EXIT2: POP  PSW
0425 3D               DCR   A               ;AND GO ON TO NEXT ONE
0426 FE07             CPI   7
```

```
0428 C21504           JNZ     EXIT1
                      ENDIF

042B C30000           JMP     EXITAD          ;DONE CLOSING: NOW REBOOT CP/M OR WHATEVER.

                ;
                ; CLOSE THE FILE WHOSE FD IS 1ST ARG:
                ;

                      IF      CPM     ;HERE COMES A LOT OF CP/M STUFF...
042E CD4805     VCLOSE: CALL    SETDMA  ;LIBRARY FUNCTION JUST JUMPS HERE.
0431 CD0A03             CALL    MA1TOH  ;GET FD IN A
0434 CD5A04             CALL    VFGFD   ;SEE IF IT IS OPEN
0437 DA0F04             JC      VERROR  ;IF NOT, COMPLAIN
043A 7E                 MOV     A,M
043B E604               ANI     4

                      IF      NOT MPM2        ;IF NOT MP/M, AND
043D CA5004             JZ      CLOSE2  ;THE FILE ISN'T OPEN FOR WRITE, DON'T BOTHER TO CLOSE
                      ENDIF

                      IF MPM2         ;ALWAYS CLOSE ALL FILES UNDER MP/M
                      NOP
                      NOP
                      NOP
                      ENDIF

0440 E5                 PUSH    H       ;SAVE FD TABLE ENTRY ADDR
0441 CD1303             CALL    MA2TOH  ;MOVE ARG1 TO A
0444 C5                 PUSH    B
0445 CD0205             CALL    VFGFCB  ;GET THE APPROPRIATE FCB ADDRESS
0448 EB                 XCHG            ;PUT IT IN DE
0449 0E10               MVI     C,16    ;GET BDOS FUNCTION # FOR CLOSE
044B CD0500             CALL    BDOS    ;AND DO IT!
044E C1                 POP     B
044F E1                 POP     H
0450 3600       CLOSE2: MVI     M,0     ;CLOSE LOGICALLY
0452 FEFF               CPI     255     ;IF 255 COMES BACK, WE GOT PROBLEMS
0454 210000             LXI     H,0
0457 C0                 RNZ             ;RETURN 0 IF OK
0458 2B                 DCX     H       ;RETURN -1 ON ERROR
0459 C9                 RET

                ;
                ; DETERMINE STATUS OF FILE WHOSE FD IS IN A...IF THE FILE
                ; IS NOT OPEN, RETURN C FLAG SET, ELSE CLEAR C FLAG:

045A CD4805     VFGFD:  CALL    SETDMA
045D 57                 MOV     D,A
045E D608               SUI     8
0460 D8                 RC              ;IF FD < 8, ERROR
0461 FE08               CPI     NFCBS
0463 3F                 CMC             ;DON'T ALLOW TOO BIG AN FD EITHER
0464 D8                 RC
0465 D5                 PUSH    D
0466 5F                 MOV     E,A     ;OK, WE HAVE A VALUE IN RANGE. NOW
```

```
0467 1600           MVI     D,0     ;  SEE IF THE FILE IS OPEN OR NOT
0469 21BF06         LXI     H,FDT
046C 19             DAD     D
046D 7E             MOV     A,M
046E E601           ANI     1       ;BIT 0 IS HIGH IF FILE IS OPEN
0470 37             STC
0471 D1             POP     D
0472 7A             MOV     A,D
0473 C8             RZ              ;RETURN C SET IF NOT OPEN
0474 3F             CMC
0475 C9             RET             ;ELSE RESET C AND RETURN

                    ;
                    ; SET UP A CP/M FILE CONTROL BLOCK AT HL WITH THE FILE WHOSE
                    ; SIMPLE NULL-TERMINATED NAME IS POINTED TO BY DE:
                    ; FORMAT FOR FILENAME MUST BE: "[WHITE SPACE][D:]FILENAME.EXT"
                    ;

                    VSETFCB:
0476 CD4805             CALL    SETDMA  ;SET UP AN FCB AT HL FOR FILENAME AT DE
0479 C5                 PUSH    B
047A CDF404             CALL    IGWSP   ;IGNORE BLANKS AND TABS
047D 0608               MVI     B,8
047F E5                 PUSH    H
0480 13                 INX     D
0481 1A                 LDAX    D
0482 1B                 DCX     D
0483 FE3A               CPI     ':'     ;DEFAULT DISK BYTE VALUE IS 0
0485 3E00               MVI     A,0     ; (FOR CURRENTLY LOGGED DISK)
0487 C29204             JNZ     SETF1
048A 1A                 LDAX    D       ;OH OH...WE HAVE A DISK DESIGNATOR
048B CDEB04             CALL    MAPUC   ;MAKE IT UPPER CASE
048E D640               SUI     '@'     ;AND FUDGE IT A BIT
0490 13                 INX     D
0491 13                 INX     D
0492 77         SETF1:  MOV     M,A
0493 23                 INX     H
0494 CD4701             CALL    PATCHNM ;NOW SET FILENAME AND PAD WITH BLANKS
0497 1A                 LDAX    D
0498 FE2E               CPI     '.'     ;AND IF AN EXTENSION IS GIVEN,
049A C29E04             JNZ     SETFCB2
049D 13                 INX     D
049E 0603       SETFCB2 MVI     B,3     ;SET THE EXTENSION AND PAD WITH BLANKS
04A0 CDAE04             CALL    SETNM
04A3 AF                 XRA     A       ;AND ZERO THE APPROPRIATE FIELDS OF THE FCB
04A4 77                 MOV     M,A
04A5 111400             LXI     D,20
04A8 19                 DAD     D
04A9 77                 MOV     M,A
04AA 23                 INX     H
04AB C33F01             JMP     SETFCB3 ;FINISH UP ELSEWHERE TO KEEP ADDRESSES CONSISTENT
                                        ;WITH PRIOR RELEASES

                    ;
                    ; THIS ROUTINE COPIES UP TO B CHARACTERS FROM MEMORY AT DE TO
                    ; MEMORY AT HL AND PADS WITH BLANKS ON THE RIGHT:
```

```
                ;
04AE C5         SETNM:  PUSH    B
04AF 1A         SETNM1: LDAX    D
04B0 FE2A               CPI     '*'     ;WILD CARD?
04B2 3E3F               MVI     A,'?'   ;IF SO, PAD WITH ? CHARACTERS
04B4 CAD104             JZ      PAD2

04B7 1A         SETNM2: LDAX    D
04B8 CDD904             CALL    LEGFC   ;NEXT CHAR LEGAL FILENAME CHAR?
04BB DACF04             JC      PAD     ;IF NOT, GO PAD FOR TOTAL OF B CHARACTERS
04BE 77                 MOV     M,A     ;ELSE STORE
04BF 23                 INX     H
04C0 13                 INX     D
04C1 05                 DCR     B
04C2 C2AF04             JNZ     SETNM1  ;AND GO FOR MORE IF B NOT YET ZERO
04C5 C1                 POP     B
04C6 1A         SETNM3: LDAX    D       ;SKIP REST OF FILENAME IF B CHARS ALREADY FOUND
04C7 CDD904             CALL    LEGFC
04CA D8                 RC
04CB 13                 INX     D
04CC C3C604             JMP     SETNM3

04CF 3E20       PAD:    MVI     A,' '   ;PAD WITH B BLANKS
04D1 77         PAD2:   MOV     M,A     ;PAD WITH B INSTANCES OF CHAR IN A
04D2 23                 INX     H
04D3 05                 DCR     B
04D4 C2D104             JNZ     PAD2
04D7 C1                 POP     B
04D8 C9                 RET

                ;
                ; TEST IF CHAR IN A IS LEGAL CHARACTER TO BE IN A FILENAME:
                ;

04D9 CDEB04     LEGFC:  CALL    MAPUC
04DC FE2E               CPI     '.'     ; '.' IS ILLEGAL IN A FILENAME OR EXTENSION
04DE 37                 STC
04DF C8                 RZ
04E0 FE3A               CPI     ':'     ;SO IS ':'
04E2 37                 STC
04E3 C8                 RZ
04E4 FE7F               CPI     7FH     ;DELETE IS NO GOOD
04E6 37                 STC
04E7 C8                 RZ
04E8 FE21               CPI     '!'     ;IF LESS THAN EXCLAMATION PT, NOT LEGAL CHAR
04EA C9                 RET             ;ELSE GOOD ENOUGH

                ;
                ; MAP CHARACTER IN A TO UPPER CASE IF IT IS LOWER CASE:
                ;

04EB FE61       MAPUC:  CPI     'a'
04ED D8                 RC
04EE FE7B               CPI     'z'+1
04F0 D0                 RNC
```

```
04F1 D620           SUI   32      ;IF LOWER CASE, MAP TO UPPER
04F3 C9             RET

                    ;
                    ; IGNORE BLANKS AND TABS AT TEXT POINTED TO BY DE:
                    ;

04F4 1B       IGWSP:  DCX   D
04F5 13       IGWSP1: INX   D
04F6 1A               LDAX  D
04F7 FE20             CPI   ' '
04F9 CAF504           JZ    IGWSP1
04FC FE09             CPI   9
04FE CAF504           JZ    IGWSP1
0501 C9               RET

                    ;
                    ; THIS ROUTINE DOES ONE OF TWO THINGS, DEPENDING
                    ; ON THE VALUE PASSED IN A.
                    ;
                    ; IF A IS ZERO, THEN IT FINDS A FREE FILE SLOT
                    ;  (IF POSSIBLE), ELSE RETURNS C SET.
                    ;
                    ; IF A IS NON-ZERO, THEN IT RETURNS THE ADDRESS
                    ; OF THE FCB CORRESPONDING TO AN OPEN FILE WHOSE
                    ; FD HAPPENS TO BE THE VALUE IN A, OR C SET IF THERE
                    ; IS NO FILE ASSOCIATED WITH FD.
                    ;

0502 C5       VFGFCB: PUSH  B
0503 CD4805           CALL  SETDMA
0506 B7               ORA   A       ;LOOK FOR FREE SLOT?
0507 4F               MOV   C,A
0508 C22D05           JNZ   FGFC2   ;IF NOT, GO AWAY
050B 0608             MVI   B,NFCBS ;YES. DO IT...
050D 11BF06           LXI   D,FDT
0510 219F05           LXI   H,FCBT
0513 0E08             MVI   C,8
0515 1A       FGFC1:  LDAX  D
0516 E601             ANI   1
0518 79               MOV   A,C
0519 C21E05           JNZ   FGFC1A  ;FOUND FREE SLOT?
051C C1               POP   B       ;YES. ALL DONE.
051D C9               RET

051E D5       FGFC1A: PUSH  D
051F 112400           LXI   D,36    ;FCB LENGTH TO ACCOMMODATE RANDOM I/O
0522 19               DAD   D
0523 D1               POP   D
0524 13               INX   D
0525 0C               INR   C
0526 05               DCR   B
0527 C21505           JNZ   FGFC1
052A 37       FGFC1B: STC
052B C1               POP   B
052C C9               RET           ;RETURN C IF NO MORE FREE SLOTS
```

```
052D CD5A04    FGFC2:  CALL    VFGFD   ;COMPUTE FCB ADDRESS FOR FD IN A:
0530 DA2A05            JC      FGFC1B  ;RETURN C IF FILE ISN'T OPEN

0533 D608              SUI     8
0535 6F                MOV     L,A     ;PUT (FD-8) IN HL
0536 2600              MVI     H,0
0538 29                DAD     H       ;DOUBLE IT
0539 29                DAD     H       ;4*A
053A 54                MOV     D,H     ;SAVE 4*A IN DE
053B 5D                MOV     E,L
053C 29                DAD     H       ;8*A
053D 29                DAD     H       ;16*A
053E 29                DAD     H       ;32*A
053F 19                DAD     D       ;36*A
0540 EB                XCHG            ;PUT 36*A IN DE
0541 219F05            LXI     H,FCBT  ;ADD TO BASE OF TABLE
0544 19                DAD     D       ;RESULT IN HL
0545 79                MOV     A,C     ;AND RETURN ORIGINAL FD IN A
0546 C1                POP     B
0547 C9                RET

0548 D5        SETDMA: PUSH    D       ;JUST A PREVENTATIVE MEASURE,
0549 C5                PUSH    B       ;SINCE THE DEFAULT I/O BUFFER
054A F5                PUSH    PSW     ;TENDS TO MAGICALLY CHANGE
054B E5                PUSH    H       ;AROUND BY ITSELF WHEN LEFT
054C 0E1A              MVI     C,26    ;IN CP/M'S HANDS !!
054E 118000            LXI     D,TBUFF
0551 CD0500            CALL    BDOS
0554 E1                POP     H
0555 F1                POP     PSW
0556 C1                POP     B
0557 D1                POP     D
0558 C9                RET

                       ENDIF           ;END OF CP/M-RELATED FILE I/O ROUTINES


                       IF      NOT CPM
               MAIN:   EQU     $       ;WHERE MAIN PROGRAM RESIDES WHEN NOT UNDER CP/M
                                       ;(UNDER CP/M, THE DATA AREA COMES FIRST)
                       ENDIF


               ;
               ; RAM AREA:
               ;

                       IF      NOT CPM ;IF NOT UNDER CP/M, USE CUSTOM RAM AREA ADDRESS
                       ORG     RAM
                       ENDIF

0559           ROOM:   DS      30      ;ROOM FOR RANDOM STUFF

0577           PBASE:  DS      2       ;SCREEN-DMA ADDRESS
0579           YSIZE:  DS      2       ;SCREEN WIDTH
```

```
057B            XSIZE:  DS      2       ;SCREEN HEIGHT
057D            PSIZE:  DS      2       ;SCREEN LENGTH

057F            RSEED:  DS      8       ;THE RANDOM GENERATOR SEED

0587            ARGS:   DS      14      ;"ARGHAK" PUTS ARGS PASSED ON STACK HERE.

0595            IOHACK: DS      6       ;ROOM FOR I/O SUBROUTINES FOR USE BY "INP"
                                        ;AND "OUTP" LIBRARY ROUTINES

059B            ALLOCP: DS      2       ;POINTER TO FREE STORAGE FOR USE BY "SBRK" FUNC
059D            ALOCMX: DS      2       ;HIGHEST LOCATION TO BE MADE AVAILABLE TO THE
                                        ;STORAGE ALLOCATOR

0559 =          TMP:    EQU     ROOM    ;THIS IS MISC. GARBAGE SPACE
055A =          TMP1:   EQU     ROOM+1
055B =          TMP2:   EQU     ROOM+2
055D =          TMP2A:  EQU     ROOM+4
055F =          UNGETL: EQU     ROOM+6  ;WHERE CHARACTERS ARE "UNGOTTEN"
0560 =          LASTC:  EQU     ROOM+7  ;LAST CHAR TYPED

                ;
                ;---------------------------------------------------------------------
                ; THE FOLLOWING DATA AREAS ARE NEEDED ONLY IF RUNNING UNDER CP/M:
                ;
                        IF      CPM
                ;
                ; THE FCB TABLE (FCBT): 36 BYTES PER FILE CONTROL BLOCK
                ;

059F            FCBT:   DS      36*NFCBS        ;RESERVE ROOM FOR FCB'S (EXTRA BYTE FOR IMDOS)

                ;
                ; THE FD TABLE: ONE BYTE PER FILE SPECIFYING R/W/OPEN AS FOLLOWS:
                ;       BIT 0 IS HIGH IF OPEN, LOW IF CLOSED
                ;       BIT 1 IS HIGH IF OPEN FOR READ
                ;       BIT 2 IS HIGH IF OPEN FOR WRITE
                ; (BOTH B1 AND B2 MAY BE HIGH)
                ;

06BF            FDT:    DS      NFCBS   ;ONE BYTE PER FCB TELLS IF IT IS ACTIVE, R/W, ETC.

                ;
                ; THE COMMAND LINE IS COPIED HERE BY INIT:
                ;

06C7            COMLIN: DS      131     ;COPY OF THE COMMAND LINE POINTED TO BY ENTRIES
                                        ;IN ARGLST

                ;
                ; THIS IS WHERE "INIT" PLACES THE ARRAY OF ARGUMENT POINTERS:
                ;

074A            ARGLST: DS      60      ;THE "ARGV" PARAMATER POINTS HERE (WELL,
                                        ;ACTUALLY TO 2 BYTES BEFORE ARGLST). THUS,
                                        ;UP TO 30 PARAMETERS MAY BE PASSED TO "MAIN"
```

```
                        ENDIF           ;(ENOUGH FOR YOU, ANDY?)

                    ;
                    ; END OF CP/M-ONLY DATA AREA
                    ;-------------------------------------------------------------------------

                        IF      CPM
0786 =              MAIN:   EQU     $       ;WHERE "MAIN" PROGRAM WILL BE LOADED UNDER CP/M
                        ENDIF

0786                    END
```