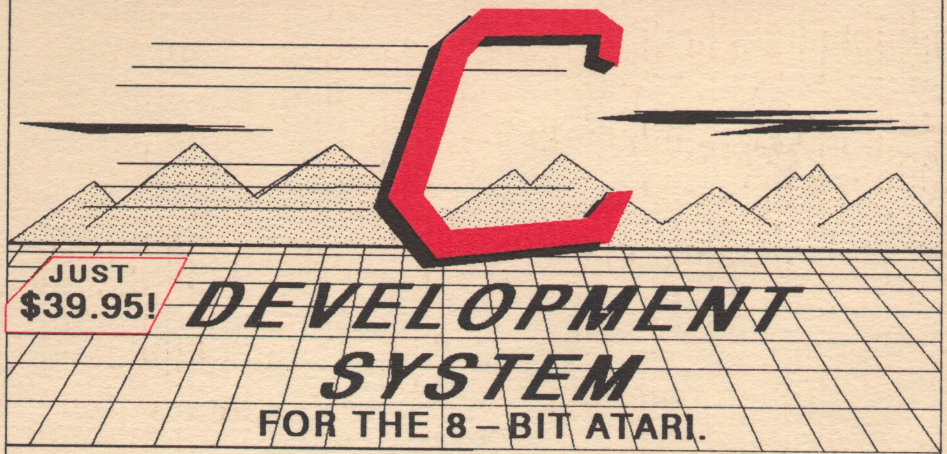
Faster Than LIGHTSPEED C



JUST $39.95!

DEVELOPMENT SYSTEM

FOR THE 8-BIT ATARI.

Over 100 standard C and specialized functions to take advantage of your Atari. With commands to access Player Missile Graphics, Sound, Joystick & Paddle, Dos, System Clock, 16 floating point functions and much, much more.

Supports the extra memory in the 130XE / compatible upgrades, and the Mosaic Board for 800's.

From within the INCLUDED editor, enter one command and you can compile, optimize, link and run your programs.

Your program can run under nearly any DOS, in single or double density.

Sieve of Eratosthenes

Language	*Time to run:*
Basic................	*31 minutes*
Deep Blue C......	*9 minutes*
Lightspeed C....	*4:49 min.*

For Atari 400/800/130 XL/XE with min. 48k and 1 drive.

Quality Software From:

Ralph E. Walden

and ...

Clearstar Softechnologies

Where the Solution is Clear.

Send all correspondance to:

Clearstar Softechnologies
1501 Wood Ave. Suite #36
Sumner, Wa. 98390

ATTN: Consumer Applications Division

Table Of Contents

INTRODUCTION

The Lightspeed C compiler is a subset of standard C as defined in the book "The C Programming Language", by Brian Kernighan and Dennis M. Ritchie. It contains all of the statements and most of the operators found in standard C, except for those dealing with structures, unions, and bit fields. Lightspeed C is a development compiler, designed to compile and link programs much faster than C compilers on other computers, and to produce very small runtime files.

You can run C programs and the Lightspeed C Development System itself with virtually any DOS system. However, Lightspeed will work best if you use the provided Lightspeed DOS, Sparta DOS 3.2 from ICD, or any DOS from Optimized Systems. These DOS systems have a command line that allows programs to work interactively with each other. For example, a single command from the editor can save your source code, compile, optimize, link and run it. If an error occurs, the editor is re-entered, your source code loaded, and the line with the error listed. The Lightspeed DOS itself contains over 100 functions that may be accessed by programs written in C, ACTION!, and assembly language (ACTION! and MAC/65 assembly language are available from Optimized Systems). You may access most of these functions using other DOS systems by using the provided runtime modules.

On disk #3 of the Lightspeed C Development System are three separate runtime files which will allow you to run C programs with other DOS systems. The three different versions of the runtime have different starting addresses, and therefore affect which DOS you can run your program with. All three versions are lacking the ACTION! and MAC/65 routines, and the C functions getsec() and putsec(). LRUNTIME.OBJ contains all the other functions described in this manual. It has a starting address of around 0x2100 (hex). MRUNTIME.OBJ is lacking sin(), cos(), and atn() functions, and has a starting address of around 0x2280 (hex). SRUNTIME.OBJ lacks the functions sin(), cos(), atn(), circle(), sound(), and pmload(). It has a starting address of around 0x2400 (hex). You should choose which of these three files is the shortest, but contains all the functions you need, and then append it onto the end of your C program. The program DCOPY.COM, described in this manual will allow you to append a runtime onto the end of your file. WARNING! Once you have appended a runtime file onto your program, you can NOT run it under Lightspeed DOS. If you do, your computer will crash.

If you have a ram disk and Sparta DOS 3.2, you will probably want to use Lightspeed with Sparta DOS. You should set up a ram disk as D8:, and copy whichever ?RUNTIME.OBJ file has all the functions you will need. You need to rename it as RUNTIME.OBJ. On side #3, you will find several .COM programs which already have RUNLOAD8.OBJ appended to them. You should copy these to a Sparta disk and rename them (remove the preceeding SP). You should copy RUNLOAD8.OBJ to your ramdisk and rename it RUNLOAD.OBJ. When the Sparta version of LINK.COM creates a .LNK file it will include RUNLOAD.OBJ. Now every time you run one of the Lightspeed .COM files, it will first load in the needed runtime from your ram disk (takes about half a second). NOTE: Lightspeed DOS provides a command line 120 characters long, whereas Sparta and OSS DOS systems only allow 64 characters.

You could use Lightspeed with an earlier version of Sparta DOS, but because it u(es unbuffered I/O, the compiler speed will be halved, and the editor will load/save files 6 times slower. You could also run Lightspeed on another DOS without a ram disk, but you would have to(wait for the runtime to be loaded in on every file (about 7 seconds longer per program). If you have Sparta 3.2 and a ram disk, use it, otherwise you should stick to Lightspeed DOS.

Please note that you must remove any cartridges before running the compiler, linker or optimizer. Hold down the OPTION key when booting an XL or XE computer. If you accidentally boot an XL or XE computer without holding down the OPTION key, then you may turn off Basic by exiting to DOS and typing OFF. This runs the program OFF.COM which will turn off the Basic ROM. It can also be used to turn off an Optimized Systems Supercartridge.

Once you have used the Lightspeed Development System for awhile, you should run CUSTOM.COM and set option #6 to turn the screen off for compiling and linking. This typically speeds up compiling and linking by about 20%.

There are 4 sides to the Lightspeed Development System. Sides 1-3 are in single density, and side 4 is in double density. The following is a list of the files included on the disks in this package. PLEASE, back up the disks BEFORE you use them!

Side #1

DOS.SYS - Lightspeed DOS
AUTORUN.SYS - gives directory of disk
BATCH.COM - creates batch file
OFF.COM - turns basic off
CB.COM - C beautifier
CC.COM C compiler
CEDIT.COM - C editor
COMPACT.COM - removes extra block headers
CONFIG.COM - change disk density
COPY.COM - file copy
CUSTOM.COM - customize Lightspeed DOS
DCOPY.COM - multi-purpose utility
DIR.COM - 2 column directory
DO.COM - multiple commands on a line
FASTER.COM - C optimizer
FILECMP.COM - compares two files
FLOAT.ACT - floating point for ACTION!
FLOAT.M65 - floating point for MAC/65
GETNAME.C - routine for getting a filename
GRAPHICS.CCC - extended graphics
HEXDMP.COM - dump a file in hex
LINK.COM - C linker
MACRO.M65 - MAC/65 macros
MGR.C - equates for mgr graphics
MGR.OBJ - High speed graphics
MGRDEMO.COM - demo of mgr graphics
MOSDRIVE.COM - Mosaic ramdisk
PMDEMO.COM - demo of player/missile graphics
RAMLOAD.COM - batch file to load ramdisk
RUNTIME.ACT - runtime for ACTION!
STDIO.ACT - I/O routines for ACTION!
STDIO.C - equates for runtime functions
STDIO.CCC - runtime functions
STDIO.H - for those who want a stdio.h file...
STDIO.M65 - I/O for MAC/65
TYPE.COM - type text a screen at a time

XEDRIVE.COM - ramdisk for 130XE

Side #2

DOS.SYS - Lightspeed DOS
FCALC.ACT - floating point calculator in ACTION!
FCALC.ASM - floating point calculator in assembly
BATCH.C - source code
CB.C - source code
COMPACT.C - source code
CONFIG.C - source code
COPY.C - source code
CUBE.C - source code 3d cubes
CUSTOM.C - source code
DIR.C - source code
FCALC.C - source code for C floating point calculator
FILECMP.C - source code
FSIZE.C - source code to give file sioze
GRAPHICS.C - source code
HEXDMP.C - source code
HVAL.C - source code for hex/int conversion
ISORT.C - source code for insertion sort
LIFE.C - source code for game of Life
MENU.C - source code
MGRDEMO.C - source code
PMDEMO.C - source code
QSORT.C - source code for quick sort
SIEVE.C - source code
SNAKE.C - source code
SORTDIR.C - source code for Sparta directory sort
TYPE.C - source code
WATFALL.C - source code
DIR.COM - 2 column directory
LIFE.COM - the game of life
LIFE.LNK
MGRDEMO.LNK
PMDEMO.LNK
LIFE.OBJ

Side #3

DO NOT run these .COM files with Lightspeed DOS! They are for use with Sparta DOS or other DOS systems ONLY!

LRUNTIME.OBJ - long runtime
MRUNTIME.OBJ - medium runtime
SRUNTIME.OBJ - short runtime
RUNLOAD1.OBJ - load runtime from D1:
RUNLOAD8.OBJ - load runtime from D8:
RUNLOAD.M65 - source code in MAC/65 tokenized format
SPCB.COM - C beautifier for non-Lightspeed DOS
SPCC.COM - C compiler for non-Lightspeed DOS
SPCEDIT.COM - C editor for non-Lightspeed DOS
SPCOMPAC.COM - file compacter for non-Lightspeed DOS
SPDCOPY.COM - multi-purpose utility for use with Sparta DOS ONLY!
SPFASTER.COM - C optimizer for non-Lightspeed DOS
SPLINK.COM - C linker for non-Lightspeed DOS

Side #4 Double Density
This contains duplicates of most of the files found on side #1 and #2, but in double density.

LIGHTSPEED C AND STANDARD C

The Lightspeed C compiler is a subset of standard C as defined in the book "The C Programming Language" by Kernighan and Ritchie. Most programs written using Lightspeed C can be run on any computer supporting standard C without alteration. For a complete comparison of Lightspeed C and standard C, compare Appendix A at the end of this documentation with Appendix A in the Kernighan and Ritchie book. Lightspeed C supports the following:

1) The declarations char, int, pointer and extern.

2) Single dimension arrays. No pointer arrays - they won't generate an error, but won't work like standard C. Note: floating point functions are provided, but you must define a floating point number as: char name[6];. It will act as a char array except in floating point functions.

3) Unary operators: * , & , - , ! , ++ , -- , ~ (tilde).

4) Binary operators:
+ , - , * , / , % , | , & , ^ , << , >> , == , != , < , > , <= , >= , && , || , ? : , (comma), (op)= (example: var += 3 (var=var+3).)

5) Statements: asm, break, case, continue, default, do, else, for, goto, if, jsr, return, switch, while.

6) Compiler directives: #define, #include, #if, #else, #endif, #ifref

7) Constants: decimal, hexadecimal, octal, backslash.

Unsupported features of C (these are NOT in Lightspeed C):

1) Structures, unions.

2) Multidimensional arrays, pointer arrays.

3) Local static declarations.

4) Full floating point implementation.

5) Function types (all functions return a 2 byte word which may be used as an integer, pointer, or character).

6) Unary operator: sizeof.

7) Binary operator: type casting.

8) The types unsigned, long, short, float, double, and register.

9) Macro expansion in #defines

10) Assignments within declarations (int val=5; is not allowed)

11) **argc** and **argv**. Use the function **getdos()** instead.

Differences from standard C:

1) Characters are unsigned, and range from 0-255.

2) Strings and comments may NOT continue onto the next line. (line refers to a logical line which may be up to 120 characters.)

3) C source code lines can be a maximum of 199 characters long after any expansion from #define's.

4) Functions may have a maximum of 126 arguments. They always return a word which may be used as a char, int, or pointer.

5) When Lightspeed C encounters a /*, it considers that the end of the line. Therefore, comments may not be imbedded within a line, and they may not cross over to a new line.

6) Lightspeed C will do constant evaluation at compile time provided there are no parenthesis between the numbers and operators: num=5*6; will be num=30; at runtime. num=5*(6); will remain the same. There is no order of precedence; evaluation is strictly right to left. num=5*6+1; is num=35. Use parenthesis to override constant evaluation.

7) A tab character is not "white space"; CC will try to compile it.

8) Lightspeed C has an assembly language interface using **asm** and **jsr**. **asm** will accept arguments from the function it defines, **jsr** does a direct subroutine call.

9) Braces in standard C are replaced with #(and #).
The tilde is replaced with ^.

10) Variables may not be declared within a statement block.

11) labels for the goto must be preceded by a colon.

12) Globals are not cleared at runtime.

BACKSLASH: Use of the backslash within a single or double quote will generate the following codes:

\b	backspace delete
\d	cursor down
\e	escape
\f	clear screen
\g	bell
\l	cursor left
\m	control M
\n	return
\r	cursor right
\t	tab
\u	cursor up

Additionaly, a backslash followed by a number will convert the number to its actual value. "\65\66" would generate the string "AB". Note: you cannot put a control comma (graphics heart) within a string since this is the end of the string character; you must use \0.

CREATING A C PROGRAM

The programs used in developing C programs all normalize the filename you enter with the default drive and an extension (dependent upon the program). The default drive used is the drive you were logged onto when the program was run (the DOS prompt gives the default drive, normally D1:). You may override the default drive by including the drive specification at the beginning of the filename. If you need to override the extension, you may either include your own extension, or follow the primary name with a period if you want no extension. The compiler, the linker, and the optimizer all require 48K, so you must remove any cartridges before using them. After you read about how to use the compiler, linker, and optimizer, you may want to skip to section XII for a practice session writing C programs.

The Compiler

The compiler takes source programs with the extension ".C" and compiles them into an intermediary file with the extension ".CCC". To run the compiler, type CC. After the CC you may type a space and up to three lines of filenames to compile all separated by spaces. You do not need to include the extension ".C", and if the file(s) is on the default drive, then you do not need to specify the drive. You may include the filename FASTER which will run the optimizer on the last file compiled. You may also include the filename LINK which will take the name of the last file compiled, remove any trailing number, add the extension ".LNK" and link the program. If you have included the name LINK, then you may also include the name RUN which will cause the program to be run after linking it.

Example:

```
    CC TEST1 TEST2 FASTER LINK RUN
```

Effect: this will compile TEST1 and TEST2, run the optimizer on TEST2, link the program using TEST.LNK, and then run the program TEST.COM.

After all filenames passed to CC from DOS are compiled (and you didn't include either FASTER or LINK), you will be asked for a filename. You may either enter another filename to compile, or press RETURN to exit to DOS.

When CC compiles a file, it creates a new file on the same drive as the source file with the extension ".CCC" for use by the linker. If any error is encountered, all compilation will stop and the line with the error and an error message will be displayed. CC will ask you which drive CEDIT.COM is on. When you press the drive number CEDIT will be run, and the source file with the error will be loaded.

CEDIT will then display both the line with the error and the preceding line, followed by the error message. If you do not wish CEDIT to be run when CC finds an error, then you must press SYSTEM RESET. It is often the case that an error occurs earlier in the program than the line listed. The compiler only lists the line it couldn't compile. If, for example, you had too many opening braces, then the error might not show up for many lines after the original error.

While the program is compiling, the compiler will list each function it is working on. If you use CUSTOM.COM (see pg. 41) to turn off the screen during compilation (this speeds up CC by about 20%), then you won't see the function names until compilation is complete, or an error occurs.

Note: Even though CC can compile almost any size source file, the largest
".CCC" file the linker can handle is 9K (about 73 single density sectors). It's safer to create several small files and link them together then to have one large file.

There are six compiler directives which you may include in the source file. They are as follows:

#define string string - whenever CC encounters the first string (all characters until the next space) it will replace it with the second string (everything to the end of the line or /*).

Example:

```
    #define EOL 154+1 /* this comment will be ignored */
    putchar(EOL); /* this will be changed to putchar(155) (note
constant evaluation) */
```

The first string is only significant to 8 places. Constant evaluation will be done on the second string.

#include filename - this will include the filename specified. The filename will be normalized with the default drive and the extension ".H" The included file is usually used as a header file containing extern declarations and #defines. You may optionally include quotes around the filename to maintain compatibility with standard C. If the filename is in lower case, it will automatically be converted to upper case.

There are four conditional directives. They are #if, #else, #endif, and #ifref. Unlike standard C, conditionals can not be nested and constant evaluation is not done on the object of the conditional. For example, #if 5*0 would evaluate as #if 5 instead of the expected zero. All conditionals must be the first statement on a line.

#if - Upon encountering an #if the compiler checks to see if the next non-space character is a zero. If so, compilation is suspended until the next conditional. If it is not a zero, then compilation will continue normally.

#else - if the previous #if or #ifref evaluated true, then all compilation will stop until the next conditional. If the previous #if or #ifref evaluated false, then compilation will continue after the #else. Remember that conditionals may not be nested.

#endif - this may be used to end any conditional block started by #if, #else, or #ifref.

Example:
```
      #if DEBUG
        printf("%d\n",ival);
      #else
        fast();
      #endif
```
Effect: if DEBUG was #defined as 1, then the value of ival would be sent to the screen. If DEBUG was #defined as 0, then the screen would be turned off.

#ifref name - This checks to see if the name has been previously referenced or defined as a global or as a function. If it hasn't, compilation will cease until the next conditional. Only the current source file will be checked for the name. By using #ifref you could compile only functions which have been called in the main program. A library of functions could be created, all preceded by #ifref, and ending with #endif. You could then #include the filename at the end of your program. This will work fine as long as the resulting ".CCC" file is less than 9K (about 73 single density sectors).

Note: The Lightspeed C compiler will not accept line numbers in the source code. If you have source code with line numbers, you must first load it with CEDIT and then save it. CEDIT will remove the offending line numbers.

The Linker

LINK.COM joins together all the files that are to be part of the same program. You must first create a file with the extension ".LNK" which contains a list of the filenames to link together. The only file types allowed are ".CCC" and ".OBJ" (object). All filenames are normalized with the default drive and the extension ".CCC", so the extension is only needed for ".OBJ" files (assembly language files should have the extension ".OBJ"). To run the linker, type LINK. After the LINK you may type a space and up to three lines of filenames to link all separated by spaces. You may optionally include the name RUN which will cause the last file linked to be run. All filenames will be normalized with the default drive and the extension ".LNK". If no ".LNK" file is found matching the filename then the linker will create one with the filename you gave it and the file STDIO.CCC. If the last character of the filename is a number, then the number will be deleted. In this manner, you could have several files with the same primary name and a different ending number, and link the file using only the primary name.

Example:
```
    LINK TEST1
```
Effect: this would link the file TEST.LNK. If TEST.LNK did not exist, then it would be created with two filenames:

```
TEST.CCC
STDIO.CCC
```

To override the number deletion, include the extension ".LNK" with the filename.

The linker will create a file with the extension ".COM" which is ready to run. When all files have been linked, the linker will ask you for another filename. You may enter another filename, or press RETURN to exit to DOS. After exiting to DOS, you may run a file created by the linker in the same manner as any ".COM" file. Type in the name (without the extension ".COM") and press RETURN. Remember that the resulting ".COM" files can only be run if you have first booted the Lightspeed DOS.SYS file. See the Introduction page for getting the programs to run on a non-Lightspeed DOS system.

If errors are encountered while linking a program, they will be listed on the screen. If the errors are fatal, then compilation will stop and the error will be listed. You must then press a key to exit to DOS. (There is no way for the linker to continue after a fatal error.) There are a few errors the linker may not catch. It will not catch duplicate function names in different files. It will also not catch a function and a global with the same name if they are defined in different files.

If you have used CUSTOM.COM to turn the screen off while linking a file (it speeds up LINK by about 20%), the screen will be turned on when all files are linked, or an error occurred.

Note: a ".COM" file will be created even if an error occurs. Do not try to run this file unless you want to crash your computer.

The ".LNK" file may have some additional lines in it besides filenames. A line beginning with a non-alphabetic character is considered to be a comment. It will be displayed on the screen, but otherwise ignored. '-' and '+' have special meanings.

A '-' followed by a hex number will start the program at that address, leaving free space (for assembly language routines) between 0x4004 and the new address. For example, -4800 would leave free space from 0x4004 to 0x4800. The C program would begin at 0x4800. You may locate the C program anywhere above 0x4004 (your program will crash if you locate it below 0x4004).

A '+' followed by a hex number will locate the stack at that address. For example, +C200 would locate the stack at 0xC200 (useful if you are using an XL or an XE with the Atari Translator, and you are short on memory for your program).

Note: If you use the Sparta DOS version of LINK, then the default .LNK file will also contain the line:
RUNLOAD.OBJ

The Optimizer

FASTER.COM may be used to speed up a C program. FASTER works on a previously compiled program (extension ".CCC") and optimizes it producing smaller code that runs about 30 percent faster. To run FASTER, type FASTER followed by up to 3 lines of filenames to optimize. Each filename will be normalized with the default drive and the extension ".CCC". After each file is optimized, it's name will be listed and the number of new instructions created (the new instructions usually replace two or more old instructions). When there are no more filenames in the command buffer, you will be asked for a filename. You may enter a filename to optimize, or press RETURN to exit to DOS. Note: FASTER is fairly slow. To speed it up, the screen will be turned off while it is working. A lengthy program could take several minutes to optimize.

You may also include the filename LINK in the command string which will cause the linker to link the last filename you entered. This allows FASTER to be part of the compile batch process.

Example:

```
    CC TEST FASTER LINK RUN.
```

Effect: this would compile TEST.C, optimize TEST.CCC, link TEST.LNK, and run TEST.COM.

Example:

```
    CC TEST1 TEST2 FASTER TEST1 LINK RUN
```

Effect: this would compile TEST1.C and TEST2.C, optimize TEST2.C and TEST1.C, link TEST.LNK, and run TEST.COM.

Tips

Once you are familiar with the process of sending filenames through the compilation process, there is way to save yourself some typing. When sending a command line to the compiler, a '+' can be used to invoke FASTER, and a '-' can be used to invoke the Linker. The linker also accepts '-' which will run the program and '+' which will pass the program on to COMPACT.

Example:

```
    CC TEST + - +
```

Effect: compiles, optimizes, links, and compacts TEST.C producing TEST.COM.
j,150,810,$

The optimizer is fairly slow, as it takes as many passes through your code as necessary until it can't optimize it further. While you are developing your program, do NOT use the optimizer - it'll just take longer to get your program debugged. But once your program is working the way you want it to, then use both FASTER and COMPACT for the most efficient file. Even if your program doesn't need to be faster, FASTER will usually reduce it's size so it will take up less disk space.

Lightspeed DOS

Lightspeed DOS was written specifically for developing and running programs in C, ACTION! and assembly language. You you can write programs in BASIC, but you will have 8K less space to do so. Since the DOS.SYS file contains both the runtime library and the C interpreter, the C programs you create using the Lightspeed C compiler will ONLY run if you have first booted a disk with the Lightspeed DOS.SYS file on it. If you want your program to run under a different DOS, then you will need to append one of the ?RUNTIME.OBJ files to it (see Introduction page for details). If you append a ?RUNTIME.OBJ file to your program, it will NOT run under Lightspeed DOS, so do this only if you want to run the program under a different DOS.SYS.

There are several important differences between Lightspeed DOS and other DOS systems. They are as follows:

1) Lightspeed DOS allows a maximum of two disk drives, either single or double density, and one single density RAM disk. Lightspeed DOS comes configured as two single density drives. If you want to change the density of a drive, you MUST use the program CONFIG.COM. You can make that change permanent by writing out the DOS.SYS file using DCOPY.COM.
2) MEMLO is fixed at 0x4000 (hex). Do not try to load or run any program that has a load address between 0x700 and 0x4000. Use the program COMPACT.COM if you are unsure of a program's load address.
3) You may point to ANY place on a disk. Because of this ability, Lightspeed DOS will not catch a file number mismatch error (#164).
4) There is no DUP.SYS. The program DCOPY.COM contains most of the functions found in DUP.SYS.

Lightspeed DOS contains five parts: the file management system, the command processor, the function library, the C interpreter, and the Operating System patches. Each section is discussed below.

FILE MANAGEMENT:

This is essentially the same as the Atari file management system. The major difference is that Lightspeed DOS does not check for a file number mismatch. This allows you to point to any sector on the disk and read or write to that sector.

COMMAND PROCESSOR:

This handles commands when you enter DOS. DOS will display Dn:, where n is the default drive number. You have 3 options while in DOS:

1) You may change the default drive number by typing Dn: and pressing RETURN (n is the new drive number).

2) You may enter a cartridge by typing a control C and pressing RETURN. A cartridge will ALWAYS do a cold start when entered from DOS ($08 is set to zero).

3) You may run a program by typing it's name and pressing RETURN. You do not need to include the extension for ".COM" programs. You may override the default drive by entering the drive you wish in front of the filename. You may pass up to 3 lines of parameters to the program being run by entering them after the filename, separated by one or more spaces. In the following examples, the default drive is assumed to be D1:.

Example:
```
CEDIT
```
Effect: runs the program D1:CEDIT.COM

Example:
```
D2:CEDIT
```
Effect: runs the program D2:CEDIT.COM

Example:
```
CC TEST
```
Effect: compiles D1:TEST.C

RUNTIME LIBRARY:

This contains over 100 assembly language functions that you may call using C, ACTION!, or assembly language. Their description is in the documentation for STDIO.C.

INTERPRETER:

This is what actually runs the programs you create using the Lightspeed C compiler and linker. It's use is virtually invisible. It does mean that you must boot a disk with the Lightspeed DOS.SYS file in order to run any C programs unless you have appended a ?RUNTIME.OBJ file to them. Note: most of the programs that come with the Lightspeed package were written in C and require the Lightspeed DOS.SYS file.

OPERATING SYSTEM PATCHES:

This redefines three of the keys on the keyboard, and also allows you to change some of the Operating System defaults. The redefined keys are:

1) CAPS LOWER - this works like a toggle key. The first time you press it you will get lower case keys, and the next time you press it you will get upper case keys. This affects only pre-XL computers. It has already been changed in the XL and XE computers.

2) INVERSE - This now works identically to CONTROL 1. Pressing it will halt screen display until you press it again. To get an inverse character, you must press the CONTROL key while pressing the INVERSE key. CONTROL 1 is still active so you may use either key.

3) CONTROL/SHIFT ESCAPE - by holding down both the CONTROL key and the SHIFT key and pressing ESCAPE, you will immediately exit to DOS. Any program in memory will be lost.

The program CUSTOM.COM allows you to change some of the Operating System defaults. The possible changes are: left margin, screen background color, character brightness, sound on/off during disk operations, key repeat speed, and screen on/off during compiling and linking. Run the program CUSTOM.COM to make any changes. You may use the W option in DCOPY to write the DOS.SYS file and make the changes effective whenever you boot the disk.

Additional notes: When Lightspeed DOS is first entered after booting, it will attempt to run the file D1:AUTORUN.SYS. It will display Error #170 if the file is not found. You may ignore the error as it has no effect other than notifying you that the file doesn't exist. You can create an AUTORUN.SYS file with the program BATCH.COM, or rename any .COM file to AUTORUN.SYS. The AUTORUN.SYS file that comes with Lightspeed is simply DIR.COM renamed to AUTORUN.SYS.

If you have an Atari with 52K of RAM, DOS will set MEMHI (0x6A) to 0xC0 rather than the expected 0xD0. SYSTEM RESET will temporarily open the screen at 0xCC40, effectively erasing any data between 0xCC20-0xCFFF. Also, clearing the screen will erase 0xC000-0xC03F, or as high as 0xC200 with a high graphics mode. With a 52K Atari, 0xC200-0xCC1F is free for permanent data storage. For an XL or XE used with the translator disk, 0xC200-0xCEFF is free.

ASSEMBLY LANGUAGE INTERFACE

On the average, a function written in assembly language will run about 10 times faster than the same function written in C. Lightspeed C provides two instructions for combining assembly language functions with C programs. The most commonly used is the **asm** instruction. To define a function using **asm**, type:

```
func() asm address-of-function;
```

The address of the assembly language function must be a constant, normally given in hex. You do not need to declare any arguments in func(). Any arguments between the parenthesis of the calling function will be placed on a stack in the order that they occur. When your routine gets control, the pointer to that stack will be at $C6. To get the low byte of an argument use:

```
   LDY #[argumentnumber-1]*2
   LDA ($C6),Y (To get the high byte, use LDY
#[argumentnumber-1]*2+1.)
```

If you want to return an integer, load A with the low byte, and X with the high byte. To return a character, load A with the character, and X with a zero. End your code with an RTS. Note: all stack entries are two bytes. If a char is placed on the stack, the first byte will be it's value, and the second byte a zero.

The other instruction for interfacing with an assembly language routine is:

```
jsr address-of-function;
```

The **jsr** statement will not pass any arguments to the assembly language instruction, though you may return a value.

Example:
```
    val=jsr 0x3033;
```
Effect: this would be the same as val=getkey();.

The page zero locations that are free are $80 through $BB. Lightspeed C uses the Operating System's floating point registers, so though you may use them, don't leave a value in them expecting it to be there the next time your routine is called. You may use memory from $400 to $580, and from $600 to $700. You can locate your code within the C program itself by using the -address option in the ".LNK" file, and locate your routine starting at $4004. For example, -4800 in the ".LNK" file would give you free memory from $4004 to $4800. Assemble your program with the extension ".OBJ". Be sure to include the filename in the ".LNK" file with the extension ".OBJ".

FLOATING POINT

Lightspeed DOS contains 16 floating point functions that can be called from C, ACTION! or assembly language. Their use is virtually identical in all three languages. The following description is for the C language. See the section on ACTION! and MAC/65 for using floating point with these languages. For an example of floating point in all three languages, look at the files FCALC.C, FCALC.ACT, and FCALC.ASM. The program FCALC.COM is the compiled and linked version of FCALC.C.

All floating point operations are done using functions. For example instead of using c=a+b, you would use fadd(a,b,c). In all functions where a calculation takes place (as opposed to a type conversion), the last argument will be where the result is stored.

To define a number as floating point use:

```
char name[6];
```

Example:
```
    fdiv(a,b,c);
```
Effect: c=a/b

Example:
```
    sin(a,b);
```
Effect: b=sin(a)

Note: before calling **atn()**, **cos()**, or **sin()** for the first time in a program, you must first call either **deg()** or **rad()**.

The following is a list of the floating point functions available.

atn(fp,result)
arctangent - takes the arctangent of **fp** and stores it in **result**.

atof(fp,string)
ascii to floating point - converts the ascii number in **string** to a floating point number and stores it in **fp**

clog(fp,result)
base 10 logarithm - takes the base 10 logarithm of **fp** and stores it in **result**.

cos(fp,result)
cosine - takes the cosine of **fp** and stores it in **result**.

deg()
degree mode for sine, cosine, and arctangent

exp(fp1,fp2,result)
exponentiation - **fp1** is taken to the **fp2** power and stored in **result**. (result=fp1^fp2)

fadd(fp1,fp2,result)
addition - adds **fp2** to **fp1** and stores it in **result**. (result=fp1+fp2)

fdiv(fp1,fp2,result)
division - divides **fp1** by **fp2** and stores it in **result**. (result=fp1/fp2)

fmul(fp1,fp2,result)
multiplication - multiplies **fp1** by **fp2** and stores it in **result**. (result=fp1*fp2)

fsub(fp1,fp2,result)
subtraction - subtracts **fp2** from **fp1** and stores it in **result**. (result=fp1-fp2)

ftoi(fp)
floating point to integer - RETURNS integer value of **fp**. Rounding is performed.

itof(ival,fp)
integer to floating point - converts the integer **ival** to floating point and stores
it in **fp**.

log(fp1,result)
natural logarithm - takes the natural logarithm of fp and stores it in **result**. (result=log(fp))

rad().
radians for sine, cosine, and arctangent

sin(fp1,result)
sine - takes the sine of **fp** and stores it in **result**.
(result=sin(fp))

sqr(fp,result)
square root - takes the square root of **fp** and stores it in **result**. (result=sqr(fp))

There is no '=' function for floating point. If you need to assign one floating point number to another (fa=fb) then use the function:

```
move(fb,fa,6);
```

STDIO.C

The following is an alphabetical list of the functions provided in STDIO.C. The compiled version of this file is STDIO.CCC which may be included in your ".LNK" files. In the absence of a ".LNK" file, the linker will automatically include this file with your main program. Globals and function names may not be the same, so you can not use any of these function names as global names. Functions working with floating point numbers will expect the name (address) of a previously defined 6 member character array. Functions which return a specific value will be indicated by a "RETURN" in all caps. If not specifically indicated, a function returns a useless value. Most functions involving input/output will return the negative of the error number if an error has occurred. (func()>=0) will be true if the function did not encounter an error. True is defined as any non-zero number, false is always a zero. The actual code for these routines is within the DOS.SYS itself. Therefore these functions are also available to languages such as ACTION! and assembly language as long as the Lightspeed DOS.SYS file was booted. C programs running outside of Lightspeed DOS may also access these functions since they are included int the ?RUNTIME.OBJ that must be appended to the .COM file.

STDIO.C contains some function names that are duplicates of existing functions - this is so you can compile both programs written with Deep Blue C that used non-standard function names, and standard C functions.

Note: unlike standard C, you cannot define a function type. However, the value returned from a function may be used as a char, an int, or a pointer. FILE type under Lightspeed C is int, rather then the structure used in most standard C compilers.

abs(i)
  int i;
  RETURNS the positive integer value of **i**.

atn(fp1,fp2)
  char fp1[6],fp2[6];
  Takes the arctangent of the floating point number in **fp1** and stores the result in **fp2**.

atof(fp,str)
  char fp[6],str[];
  Converts the ascii number in the string **str** to a floating point number, and stores the result in fp.

atoi(str)
char str[];
RETURNS the integer value of the ascii number in **str**. Rounding is performed if there is a decimal portion.

Example:
atoi("1.7");
Effect: this would return a 2.

bgets(addr,len,iocb) /* use fread() for standard compatibility */
char *addr;
int len,iocb;
Gets a maximum of **len** number of bytes from the iocb number and stores them starting at **addr**. RETURNS the actual number of bytes received. Does NOT RETURN an error number! (use **status()**)

bputs(addr,len,iocb) - use fwrite() for standard compatibility

char *addr;
int len,iocb;
Put **len** number of bytes to the iocb starting at **addr**.
RETURNS true if no error, else the negative of the error number.

brkey()
RETURNS true if the break key was pressed.

calloc(bytes)
int bytes;
Clears and allocates the amount of memory in **bytes**, and RETURNS a pointer to the beginning of the allocated memory, or zero if none found. The allocated memory will be cleared (set to zero). Memory is allocated from the top of computer memory downward. Since the stack grows upward, care must be taken that malloc() and calloc() memory is not overwritten by a later function which has a large stack call (large local arrays). malloc() and calloc() may be called as many times as needed as long as there is enough memory.

cgetc(iocb) /* same as fgetc() */
int iocb;
RETURNS the character (byte) from the iocb specified, or a negative error number. cgetc() will first check to see if a character was put back by ungetc() for that iocb, and if so it will get that character.

cgets(str,iocb)
char str[];
int iocb;
Gets a record from the iocb and puts it in **str**. RETURNS the length or the negative error number. The incoming record may be of any length. It is the users responsibility to dimension the string for the largest possible record. The ending return

character is converted to a zero to indicate the end of the string. cputs(str,iocb) will put the return character back when writing the string.

chain(filespec)

char filespec[];

Runs the file named in **filespec**. The entire string is passed, so you may pass commands to the file being run.

Example:

```
     chain("CC TEST");
```

Effect: this will run the compiler and compile the file TEST.

ciov(iocb,command,addr,len,ax1,ax2)

int iocb,command,len,ax1,ax2;

char *addr;

A direct call to the Atari Operating System STDIO function. Use a -1 to ignore any parameter.

Example:

```
     ciov(4,3,"P:",8,-1);
```

Effect: this would open the printer using iocb #4.

RETURNS true, or the negative error number.

circle(xc,yc,radius)

int xc,yc,radius;

Draws an approximation to a circle using **xc,yc** as the center point and the radius specified. Degree of roundness will be determined by the graphics mode used.

clear(addr,len)

char *addr;

int len;

Stores zeros into **len** number of bytes starting at **addr**.

clog(fp1,fp2)

char fp1[6],fp2[6];

Takes the base 10 log of the floating point number in **fp1**, and stores the result in **fp2**.

close(iocb)

int iocb;

Closes the iocb specified.

closeall()

Closes iocbs 1 through 7.

color(c)

char c;

Sets the color to be used in graphics functions (plot(), drawto(), circle(), etc.)

clrtime()
Sets the lower 2 bytes of the System clock to zero. (See gtime()).

console()
RETURNS 1 if START is pressed, 2 if SELECT is pressed, 3 if OPTION is pressed, else it returns zero.

copen(filespec,mode)
char filespec[],mode;
Opens the device or file in **filespec** in the mode specified. RETURNS the iocb number or the negative error number. The mode specified should be a character constant in either upper or lower case. The following constants are available:

'r' -read
'w' -write
'u' -update (read/write)
'a' -append
'd' -directory

cos(fp1,fp2)
char fp1[6],fp2[6];
Takes the cosine of the floating point number in **fp1** and stores the result in **fp2**. RETURNS -146 if an error occurred.

cputc(c,iocb)
char c;
int iocb;
Send **c** (byte) to the iocb number specified. RETURNS true or the negative error number.

cputs(str,iocb)
char str[];
int iocb;
Send the string in **str** to the iocb number specified. RETURNS true or the negative error number. A string always ends with a zero. Cputs replaces that zero with a return character when it sends it out.

deg()
Degree mode for sin, cos and atn functions.

dfast()

Turns off the screen and turns on the critic mode only if the user has selected screen off using CUSTOM.COM. Look at the description for **fast()** for another way of doing this.

dpeek(addr)
char *addr;
RETURNS the 2-byte integer at **addr**.

dpoke(addr,i)
char *addr;
int i;
Stores the 2-byte integer i into **addr**. RETURNS the previous value. The integer is stored in low byte, high byte order.

drawto(x,y)
int x,y;
Draws a line from the current cursor position (from previous plot) to x,y. RETURNS true or the negative error number.

exit()
Exits to DOS. Exit will turn on the screen if a fast() has turned it off.

exp(fp1,fp2,fp3)
char fp1[6],fp2[6],fp3[6];
fp1 is raised to the **fp2** power and stored in **fp3**. Example: exp(fp1,fp2,fp3) where fp1=4, fp2=0.5 would store a 2 in **fp3** (square root). (fp3=4^0.5) **fp1**, fp2, and **fp3** are floating point numbers

fadd(fp1,fp2,fp3)
char fp1[6],fp2[6],fp3[6];
Floating point addition. Adds **fp1** and **fp2** and stores result in **fp3**.

fast()
Turns off the screen and turns on critic mode. Opposite of slow(). Speeds up programs by about 20%.

fclose(iocb)
int iocb;
Closes the iocb specified.

fgetc(iocb)
int iocb;
RETURNS the character (byte) from the iocb specified, or a negative error number. fgetc() will first check to see if a character was put back by ungetc() for that iocb, and if so it will get that character.

fgets(str,iocb)
char str[];
int iocb;
Gets a record from the iocb and puts it in **str**. RETURNS the length or the negative error number. The incoming record may be of any length. It is the users responsibility to dimension the string for the largest possible record. The ending return character is converted to a zero to indicate the end of the string. fputs(str,iocb) will put the return character back when writing the string.

fopen(filespec,mode)
char filespec[],mode[];
Opens the device or file in **filespec** in the mode specified. RETURNS the iocb number or the negative error number. The mode specified should be a string containing the character constant in either upper or lower case. The following constants are available:
'r' -read
'w' -write
'u' -update (read/write)
'a' -append
'd' -directory

fputc(c,iocb)
char c;
int iocb;
Send **c** (byte) to the iocb number specified. RETURNS true or the negative error number.

fputs(str,iocb)
char str[];
int iocb;
Send the string in **str** to the iocb number specified. RETURNS true or the negative error number. A string always ends with a zero. Cputs replaces that zero with a return character when it sends it out.

fdiv(fp1,fp2,fp3)
char fp1[6],fp2[6],fp3[6];
Floating point division. Divides **fp1** by **fp2** and stores result in **fp3**.

ferase(filespec)
char filespec[];
Erases the file specified by **filespec**. RETURNS true or the negative error number. Example: ferase("D1:TEST"); would erase that file.

find(addr,len,c)
char *addr,c;
int len;
Search for the byte c starting at **addr** for **len** bytes. RETURNS offset from **addr** if found, else -1. Find ignores string boundaries. Use strchr() to limit the search to a string.

flock(filespec)
char filespec[];
Locks the named file. RETURNS true or the negative error number.

fmul(fp1,fp2,fp3)
char fp1[6],fp2[6],fp3[6];
Floating point multiplication. Multiplies **fp1** by **fp2** and stores result in **fp3**.

fread(addr,size,num,iocb)
char *addr;
int size,num,iocb;
Gets a maximum of **size*num** number of bytes from the iocb number and stores them starting at **addr**. RETURNS the actual number of bytes received. Does NOT RETURN an error number! (use **status()**)

free(bytes)
int bytes;
This frees **bytes** amount of memory that was previously allocated by malloc() or calloc(). free() works in reverse order to malloc() and calloc() calls, i.e., it frees the memory allocated by the last malloc() or calloc() call. **bytes** should be the same number as the previous malloc() or calloc() call.

frename(filespec)
char filespec[];
Renames the first filename in **filespec** to the second name in **filespec**. RETURNS true or the negative error number.

Example:
```
     frename("D1:MARIE EDITH");
```
Effect: D1:MARIE will be renamed as D1:EDITH.

fsub(fp1,fp2,fp3)
char fp1[6],fp2[6],fp3[6];
Floating point subtraction. Subtract **fp2** from **fp1** and store the result in **fp3**.

ftoi(fp)
char fp[6];
RETURNS the integer value of the floating point number in fp. Rounding is performed. For example a 1.5 would be converted to a 2, a 1.4 would be converted to a 1.

funlock(filespec)
char filespec[];
Unlock the file specified in **filespec**. RETURNS true or the negative error number.

fwrite(addr,size,num,iocb)
char *addr;
int size,num,iocb;
Put **size*num** number of bytes to the iocb starting at **addr**. RETURNS true if no error, else the negative of the error number.

getchar()
RETURNS the ascii value of the key pressed by the user and echoes it to the screen. Functionally equivalent to putchar(getkey());.

getdos(str)
  char str[];
  RETURNS zero if there are no more commands passed from DOS, else true. When a C program is called from DOS, parameters may be passed to it by separating them with spaces after the program name. For example, D1: TEST 3 DATA.DT would run the program TEST. The first time **getdos()** is called, D1:3 would be placed in **str**. The second time it is called, D1:DATA.DT would be placed in **str**. The third time, **str** would be unchanged, and a zero would be returned. Note: the default drive number will always be placed at the beginning of the string unless the user includes their own. For example, TEST would be converted to D1:TEST, whereas D2:TEST would remain the same.

getkey()
  RETURNS the ascii value of the key pressed by the user. Does not echo the key to the screen.

gets(str)
  char str[];
  Gets a record from the user (via the screen editor) and places it in **str**. RETURNS the length of the string or the negative error number. You will get a negative error number if the user presses the break key.

getsec(addr,sector,drive)
  char *addr,drive;
  int sector;
  This reads the specified sector into **addr** from the drive specified. It will read either 128 or 256 bytes depending on whether the drive is single or double density.

getw(iocb)
  int iocb;
  RETURNS a 2-byte integer from the iocb specified or the negative error number.

graphics(mode)
  int mode;
  Opens iocb #6 as the screen in the graphics mode specified (same as Basic). No check is made to see if the screen overwrites data or program area. graphics() will destroy any data allocated by malloc() and calloc(). If you need to allocate a block of memory after a graphics call, use highmem() to get the address.

gtime()
  RETURNS the integer value of the lower 2 bytes of the Operating System's clock. The value returned will be in 60ths of a second. Use gtime()/60 for seconds, gtime()/360 for minutes. You may clear the clock by calling the function clrtime().

highmem()
RETURNS the address of the highest useable memory location (located just below the display list). Note: highmem() is not affected by malloc() and calloc() calls. You should use one or the other in your program, but not both.

index(str1,str2)
char str1[],str2[];
Searches for the first occurrence of the string **str2** in **str1** and RETURNS the address if found, else it RETURNS zero.

inkey()
RETURNS true if the user has pressed a key which is waiting to be processed, else it returns zero.

isalnum(c)
char c;
RETURNS true if c is alphabetic or numeric.

isalpha(c)
char c;
RETURNS true if c is alphabetic.

isascii(c)
char c;
RETURNS true if c < 128. Note: a return character is 155, and would evaluate as false. All inverse characters will evaluate as false.

isdigit(c)
char c;
RETURNS true if c is a digit (number 0-9).

isspace(c)
char c;
RETURNS true if c is a space.

itof(i,fp)
int i;
char fp[6];
Converts the integer i to floating point and stores the result in fp.

locate(x,y)
int x,y;
RETURNS the value of the point at location x,y or the negative error number. (Same as Basic).

log(fp1,fp2)
char fp1[6],fp2[6];
Takes the natural log of the floating point number in **fp1**, and stores the result in **fp2**.

lomem(offset)
int offset;
RETURNS the address of the current stack location plus the offset. This is often used for setting a pointer to the beginning of an undimensioned array. Variable declarations within a function are placed on the stack, so **offset** should be larger than the total amount of memory needed by any function.
See the program CSORT.C for an example of this function's use.

malloc(bytes)
int bytes;
Allocates bytes amount of memory and RETURNS a pointer to that memory, or zero if none is found. Memory is allocated from the top of computer memory downward. **malloc()** is the same as **calloc()**, only the memory is not cleared.

move(from,to,len)
int from,to,len;
Moves **len** number of bytes at address **from** to address **to**.
move() will correctly handle overlapping data.

normalize(filename,ext)
char filename[],ext[];
Forces the filename to uppercase and adds the default drive to the front of the filename if not specified, and adds the extension in ext if no extension is included in filename. The default drive is the drive you were logged onto when the program was run.

```
Example:
     char s[20];
     strcpy(s,"TEST");
     normalize (s,"DAT");
Effect: this will change s to "D1:TEST.DAT".
```

note(iocb,sector,byte)
int iocb,*sector,*byte;
Stores the sector and byte numbers of the current place in the file specified by the iocb number. When calling this function you must pass the ADDRESS of sector and byte, rather than just their names. A normal call might look like:

```
note(iocb,&sector,&byte);
```

open(iocb,ax1,ax2,filespec)
int iocb,ax1,ax2;
char filespec[];
Opens the iocb number specified using **filespec** as the name and ax1 and ax2 as the mode. RETURNS true or the negative error number. Use -1 to ignore either ax1 or ax2.

peek(addr)
char *addr;
RETURNS the byte at **addr**.

plot(x,y)
int x,y;
Plot a point at x,y using the color previously set by color(c). RETURNS true or the negative error number.

point(iocb,sector,byte)
int iocb,sector,byte);
Sets the file pointer for the iocb specified to the sector and byte number specified. RETURNS true or the negative error number. Note: with Lightspeed DOS you may point anywhere on a disk. There is no check for mismatched file numbers and no check to see if you are still pointing within the file opened using **iocb**.

poke(addr,c)
char *addr,c;
Pokes c into the address specified and RETURNS the previous value.

position(x,y)
int x,y;
Positions the cursor at the **x,y** position on the screen. **x** is the horizontal position, and **y** is the vertical position.

printf()
The printf function enables you to print information to the screen or any device. It takes a variable number of arguments. If the first argument is less than 255, it is assumed to be an iocb #, and everything is sent to that iocb. Otherwise the output is sent to the screen (iocb #0). The next argument is the format string, followed by any additional arguments. Everything in the format string will be sent out until a % is encountered. The character(s) following the % determine how the argument is to be printed. The first % is for the first argument after the format string, the second for the second argument, and so on. Too few arguments will cause garbage to be printed. The following characters may be used after the %:

c - print a character
d - print an integer
f - print a floating point number
n - prints a return (no argument) (used by ACTION!)
s - print a string
u - print an unsigned decimal number
x - print a hexadecimal number
% - print a percent sign (no argument)

Note: A character, integer, or hexadecimal value may be obtained from any variable defined as either char or int. A floating point number is assumed to have been defined as a 6 element char array. A number between the % and the character defines the field width. If there aren't enough characters or numbers in the argument, it will be filled out with spaces. A '-' before the number will left justify the field. A decimal point and a number will determine the decimal part printed in a floating point number, padded with up to 2 zeros if there aren't enough. Here are some examples:

```
printf(")abcd(");  produces                   )abcd(
printf(")%s(","abcd"); produces               )abcd(
printf(")%10s(","abcd"); produces             )      abcd(
printf(")%-10s(","abcd"); produces            )abcd      (
printf("%c %d %x",65,65,65); produces         A 65 41
```

putchar(c)
char c;
Send c to the screen.

putsec(addr,sector,drive)
char *addr,drive;
int sector;
This writes 128 or 256 bytes from **addr** to the sector on the disk drive specified. It will write 128 bytes on a single density disk, and 256 on a double density disk. No checks are made to see what sector you are writing to. Use with great caution!

putw(i,iocb)
int i,iocb;
Sends the 2 byte integer i to the iocb specified in low byte, high byte format. RETURNS true or the negative error number.

rad()
Changes to radians for **sin()**, **cos()**, and **atn()** functions.

rindex(str1,str2)
char str1[],str2[];
Finds the last occurrence of **str2** in **str1** and RETURNS the address or zero if not found.

rand(max)
int max;
Returns a random integer between 0 and max. **max** must be less than 256. Sending a zero is the same as sending a 256, the returned value will be between 0 and 255 inclusive.

Example:
```
    rand(2);
```
Effect: this would randomly return a 0 or a 1.

scanf()
scanf is like the reverse of printf. It enables you to input numbers, strings or characters from the screen or any device. It takes a variable number of arguments. If the first argument is less than 255, it is assumed to be an iocb #, and input is from

that iocb. The next argument is the format string, followed by the ADDRESS of all arguments needed by the format string. scanf() will RETURN 1 if there are no errors, else it RETURNS a negative error number. Arguments in the format string should be preceded by a %. The character after the % determines the type of the argument to store the value in. There are four possible types:

c - character (ignores spaces, commas, and tabs)

d - integer (equivalent to the **atoi()** function)

f - floating point number (equivalent to the **atof()** function)

s - string (a string begins with the first character which is not a space, comma or tab, and ends with the first space, comma, tab or return character.)

The input may be from a device, or from the screen. Input must not exceed 120 characters, and always ends with a return. Arguments can be separated with spaces, commas, tabs, or returns. scanf() will keep reading from the device or screen until all arguments are filled.

```
Example:
     char str[20],c,fp[6];
     int id;
     scanf("%c %d %f %s",&c,&id,fp,str);
Input line: A -346.1    3.14 Hello, world!
Values:
     c == 'A'
     d == -346  (Note that rounding took place)
     fp == 3.14
     str == "Hello"
```

setblock(addr,len,byte)
char *addr,byte;
int len;
Fills **addr** to **addr+len** with **byte**.

setcolor(n,hue,intensity)
int n;
char hue,intensity;
Sets color **n** to the hue and intensity specified. Same as BASIC.

sfind(str1,str2)
char str1[],str2[];
Searches for **str2** in **str1** and RETURNS the offset from the beginning of **str1** if found, else it RETURNS -1.

```
Example:
     sfind("Hi there, Edith!","Edith");
Effect: returns a 10.
```

sin(fp1,fp2)
char fp1[6],fp2[6];
Takes the sine of the floating point number in **fp1**, and stores the

result in **fp2**. RETURNS -146 if an error occurred.

slow()
Used after a **fast()** or **dfast()** call to turn the screen back on.

smatch(str,str2)
char str1[],str2[];
Checks to see if **str2** is the same as **str1** even if **str1** is larger.
RETURNS true if a match, else false.

Example:
smatch("abcdefg","abc");
Effect: would return true, whereas strcmp("abcdefg","abc");
would return -100.

sound(voice,pitch,distortion,volume)
int voice,pitch,distortion,volume;
Turns on the sound register specified by voice. Same as Basic.

sprintf(str,...)
char str[];
Same as printf, only the output is to the string **str**.

sqr(fp1,fp2)
char fp1[6],fp2[6];
Takes the square root of the floating point number in **fp1** and stores the result in **fp2**. The square root is an approximation. For example, the **sqr()** of 144 would be 11.999998 instead of the expected 12.

status(iocb)
int iocb;
RETURNS the status of the iocb specified. A return value of 128 or greater indicates an error.

stick(n)
int n;
RETURNS the value of joystick number n.

strcat(str1,str2)
char str1[],str2[];
Places a copy of **str2** at the end of **str1**.

strchr(str,c)
char str[],c;
This searches for the character c in the the string **str**. It RETURNS the position in the **str**. It RETURNS the position in the string if found, or -1 if not found. It is the same as the **find()** function, only the search is limited to a string.

strcmp(str1,str2)
char str1[],str2[];
Subtracts **str2** from **str1**. Zero means the strings are equal, negative means **str2>str1**, positive means **str2<str1**.

Example:
```
strcmp("abc","cde");
```
Effect: returns -2.

strcpy(str1,str2)
char str1[],str2[];
Copies **str2** to **str1** and RETURNS the length of the string copied.

strig(n)
int n;
RETURNS zero if the button on joystick #n is pressed, else it RETURNS one.

strlen(str)
char str[];
RETURNS the length of the string **str**.

toascii(i,str)
int i;
char str[];
This converts the integer in **i** to an ascii string and stores it in **str**.

tolower(c)
char c;
RETURNS the lowercase value of **c**.

toupper(c)
char c;
RETURNS the uppercase value of **c**.

ungetc(c,iocb)
char c;
int iocb;
Puts back the character **c** to the iocb specified. You may only put back one character, which will erase any previous character stored. The character put back may ONLY be retrieved via the **cgetc()** function. You cannot put back a zero.

The following is a list of the function names and their arguments by category.

FLOAT

atn(fp1,fp2) - arctangent
atof(fp,ascii) - ascii to float
clog(fp1,fp2) - base 10 logarithm
cos(fp1,fp2) - cosine
deg() - degree mode
exp(fp1,fp2,fp3) - exponentiation
fadd(fp1,fp2,fp3) - addition
fdiv(fp1,fp2,fp3) - division
fmul(fp1,fp2,fp3) - multiplication

fsub(fp1,fp2,fp3) - subtraction
ftoi(fp) - floating point to integer
itof(i,fp) - integer to floating point
log(fp1,fp2) - natural logarithm
rad() - radians
sin(fp1,fp2) - sine
sqr(fp1,fp2) - square root

GRAPHICS

circle(xc,yc,radius) - draws a circle
color(i) - sets color for plotting
drawto(x,y) - draws a line
graphics(mode) - sets graphics mode
locate(x,y) - returns color at x,y
plot(x,y) - plots a point

INPUT/OUTPUT

bgets(addr,len,iocb) - block input
bputs(addr,len,iocb) - block output
chain(filespec) - runs **filespec**
cgetc(iocb) - inputs a character
cgets(iocb) - inputs a record
ciov(iocb,command,addr,len,ax1,ax2) - calls CIO (Operating System)
close(iocb) - closes iocb
copen(filespec,mode) - opens **filespec** in mode specified
cputc(ch,iocb) - puts a character to iocb
cputs(addr,iocb) - puts a record to iocb
ferase(filespec) - erases **filespec**
fclose(iocb) - closes iocb
fgetc(iocb) - inputs a character
fgets(iocb) - inputs a record
flock(filespec) - locks **filespec**
fopen(filespec,mode) - opens **filespec** in mode specified
fputc(ch,iocb) - puts a character to iocb
fputs(addr,iocb) - puts a record to iocb
frename(filespec) - renames **filespec**
funlock(filespec) - unlocks **filespec**
getchar() - gets character from user
getdos(str) - gets next command from DOS command buffer
getkey() - gets next key pressed
gets(addr) - gets a record from the user
getsec(addr,sector,drive) - read a sector
getw(iocb) - gets a word from iocb
inkey() - true if a key is waiting
note(iocb,&sector,&byte) - get current location in iocb
open(iocb,ax1,ax2,filespec) - opens iocb
point(iocb,sector,byte) - sets pointer to iocb
printf() - formatted output
putchar(c) - sends character to the screen
putsec(addr,sector,drive) - write a sector

putw(w,iocb) - sends word to iocb
status(iocb) - gets status of the iocb
ungetc(c,iocb) - returns byte to iocb

MEMORY

calloc(bytes) - clear and allocate memory
clear(addr,len) - set block of memory to zeros
free(bytes) - free allocated memory
highmem() - returns high memory address
lomem(offset) - pointer to low memory
malloc(bytes) - allocate memory
move(from,to,len) - move block
setblock(addr,len,ch) - set block of memory to character ch

STRING FUNCTIONS

index(st1,str2) - searches for **str2** in **str1**
isalnum(c) - true if alphabetic or numberic
isalpha - true if alphabetic
isascii(c) - true if ascii (not-inverse)
isdigit(c) - true if numeric
isspace(c) - true if a space
normalize(filespec,ext) - add default drive and extension to **filespec**
rindex(str1,str2) - last occurrence of **str2** in **str1**
sfind(str1,str2) - find **str2** in **str1**
sprintf() - printf() to a string
strcat(str1,str2) - concatenate **str2** to **str1**
strcmp(str1,str2) - compare **str1** and **str2**
strcpy(str1,str2) - copy **str2** to end of **str1**
strlen(str) - length of **str**
toascii(i,str) - convert integer to a string
tolower(c) - lowercase of c
toupper(c) - uppercase of c

NOTE: **malloc()** and **calloc()** are provided for compatibility with standard C. With only 31K available for program/data/stack space, they're not very useful on the Atari. It is suggested that you use **lomem()** and **highmem()** instead. Also, calling a graphics mode after a **malloc()** or **calloc()** call will destroy their data.

PLAYER/MISSILE GRAPHICS

The file GRAPHICS.C is the source code for GRAPHICS.CCC and contains several functions primarily for doing player/missile graphics and alternate character sets. To use these functions include the name GRAPHICS in your ".LNK" file. Look at the source file PMDEMO.C for an example of their use. The following is a list of the functions:

pminit()
This must be called before using any other player/missile functions or character set functions. It can ONLY be used with 48K of RAM (no cartridges!).

pmflush()
This should be called before exiting to DOS to turn off any player/missile graphics and to restore the character set.

hitclear()

This clears the collision register

hitpf(who,hitwho)
int who,hitwho;
This will RETURN a 1 if player **who** collided with playfield **hitwho**, or a zero if no collision occurred. If **hitwho** is a -1, then it will RETURN a 1 if player **who** collided with any playfield.

hitpl(who,hitwho)

int who,hitwho;
This will RETURN a 1 if player **who** collided with player **hitwho**, or a a zero if no collision occurred. If **hitwho** is a -1, then it will RETURN a 1 if player **who** collided with any player.
Note: neither **hitpf()** or **hitpl()** clear the collision register.

pladdr(n)

int n;
This returns the address of player n.

pmclear(n)
int n;
This clears player n

pmcolor(n,hue,intensity)
int n;
char hue,intensity;
This sets the color of player **n** to the hue and intensity specified. Same as setcolor(), only for the players.

pmgraphics(mode)
int mode;
This sets up the player/missile graphics. If **mode** is zero, it turns them off. If **mode** is a one, it sets up single line resolution graphics. If **mode** is a two, it sets up double line resolution.

pmload(n,x,y,shape)
int n,x,y;
char shape[];
This loads the shape into player **n** at the **x** and **y** position specified. **Shape** must be an array where the first byte contains the length of the array. Since the first byte defines the length, the array may contain zeros. By using zeros before and after the shape, you can erase the previous shape, allowing fast vertical motion. See PMDEMO.C source code for a demonstration of this function.

pmwidth(n,width)
int width;
This sets the width of player n.

```
width == 0 - normal width
width == 1 - double normal width
width == 3 - four times normal width
```

When **pminit()** is called, it sets up an alternate character set by moving the ROM character set to 0xB400. The following 3 functions are for use with this alternate character set. All moves involve 8 bytes regardless of whether there is a zero in the character array.

chaget(c,str)
char c,str[];
Get the ROM character definition for **c** and put it in **str**.

chget(c,str)
char c,str[];
Get the character definition in the alternate character set for **c** and put it in **str**.

chput(str,c)
char str[],c;
Move the 8 members of the array **str** into the alternate character set definition for **c**.

The following are miscellaneous functions.

fill(x,y,c)
int x,y;

char c;
Fill the graphics box pointed to by **x,y** with the character **c**. The arguments **x,y** should be the coordinates for the upper left corner.

hstick(n)

int n;
RETURNS 1 if joystick **n** is pushed to the right, -1 if to the left, or zero if not pushed left or right.

paddle(n)
int n;
RETURNS the value of paddle n.

ptrig(n)
int n;
RETURNS zero if paddle trigger **n** is pressed, 1 if not.

vstick(n)
int n;
RETURNS 1 if joystick **n** is pressed forwards, -1 if pulled backwards, or zero if not pressed forwards or backwards.

Note: **hstick()** and **vstick()** are easy ways to move players when no diagonal movement is required.

Example:

```
pmload(n,x+=hstick(1),y+=vstick(1),shape);
```

Effect: this would move the player in whatever direction the joystick was pressed and update the x and y values at the same time. If you defined **x** and **y** as type char, then you wouldn't have to worry about the player moving out of it's memory area.

MGR.OBJ

MGR contains a set of graphics routines that completely replace the normal Atari graphics. Functions such as plot and locate will only be slightly faster then normal, but line drawing and fill commands can be up to 13 times faster. If you run MGRDEMO you will get an idea of the speed difference between normal graphics and MGR graphics. Except for the last fills, the demo uses oarange for Atari graphics, and blue for MGR graphics so you can tell which is being used.

The most important part about using MGR.OBJ is correctly putting it in your .LNK file. The first two lines of your .LNK file should be:

```
-4800
MGR.OBJ
```

The -4800 is to allow room for the MGR.OBJ file which follows. The next think to do is to merge mgr.c with your source file so that you can access all the routines. The following functions are now available to you.

mcolor(c)
char c;
Sets the color for MGR graphics to c.

mplot(x,y)
int x,y;
Plot a point at x,y.

mlocate(x,y)
int x,y;
RETURNS the value of the point at location x,y.

mdrawto(x,y)
int x,y;
Draws a line from the current cursor position (last graphics point) to x,y.

mcircle(xc,yc,radius)
int xc,yc,radius;
Draws an approximation to a circle using xc, yc as the center point and the radius specified.

mbox(xr,yr,xl,yl)
int xr,yr,xl,xr;
Draws a box using xr, yr as one corner and xl, yl as the opposite diagonal corner.

mfbox(xr,yr,xl,yl)
int xr,yr,xl,yl;
Draws a solid box using xr, yr as one corner and xl, yl as the opposite diagonal corner.

mfill(x,y)

int x,y;

Fills a surronded area. Color used is from last mcolor() call, border area to stop fill must also be the same color. Unlike the Atari fill, the area to be filled MUST be complete surronded. mfill() works horizontally, so x,y should be set to the widest area of the object to fill.

mvfill(x,y)

int x,y;

Same as mfill(), only fill is done vertically, so the point should be set at the tallest area of the object to be filled.

Understanding the difference between Atari's fill(), mfill() and mvfill() can be difficult. The first thing to keep in mind is that mfill() and mvfill() require a COMPLETELY surronded area. It might help to think of mfill() drawing horizontal lines to fill the area, and mvfill() drawing vertical lines to fill the area. Under MGR, vertical lines are faster then horizontal lines, so mvfill() should be used whenever the shape to fill permits it. Close scrutiny of the MGRDEMO.C source code should help clarify the differences.

CEDIT

CEDIT is an editor that is specifically designed for creating and modifying C programs. It is designed to work interactively with the Lightspeed C compiler, linker, and optimizer. To run the editor from DOS, type CEDIT [filename]. The filename is optional; if included, CEDIT will load that file. The default drive and the extension ".C" will be added to the filename if you don't specify them. The editor is also run when the C compiler finds an error in your program. After displaying the error, the compiler will ask which drive CEDIT.COM is on. When you press the drive number, CEDIT will be run. CEDIT will then load the offending source file. CEDIT will display the line where the problem occurred, the preceding line, a pointer to the error, and a brief error message. The preceding line is displayed because many errors are caused by a problem with the previous line. The pointer points to where the compiler found an error, which may not be where the error actually occurred. For example, having too many opening braces may not show up as a problem until the end of a function definition.

CEDIT uses line numbers to keep track of the lines in your program. Some commands such as AUTO and MOVE, will automatically renumber the program. Whenever lines are renumbered, the numbering starts with line 1000, and increases by 10 for each line. Line numbers are removed when the program is saved or compiled, and added when the program is entered.

If you run CEDIT without including a filename to load, then you will see a menu of commands on the screen. At the lower left of the screen you will see the word "Command" in inverse. Whenever "Command" is displayed, you are in command mode, and you may invoke one of the menu options by pressing the corresponding letter. If you press the letter of a command that you don't want, then press the BREAK key to return to command mode. When commands require more than one line number, you may separate the numbers by either spaces or commas. You may use a semicolon in place of a number, which is equivalent to the last line number of your program.

There are three methods for entering text from the keyboard: line entry, auto entry, and text entry. Auto entry and text entry mode are entered by pressing 'A' or 'T' respectively. To enter line entry mode, press a number or a cursor key. You may now enter a new line, or edit an existing line that is displayed on the screen. In both cases, the line must begin with a number that is greater than 256, or an error message will be displayed and you will be returned to command mode. If a line contains only the line number, you will get a blank line rather than having the line deleted (you must use the DELETE command to delete a line). To edit a line displayed on the screen, move the cursor to that line, make the changes, and press

RETURN. If the line number already exists in your program, it will be replaced by the new line. When you are done making changes or adding lines, press the BREAK key to return to command mode.

The following is a list of the menu options and their descriptions. To invoke one of the commands, press the first letter of that command.

AUTO- Auto entry mode allows you to enter text without line numbers. First, you will be asked to give the line number that the new text will follow. You may enter a line number or press RETURN to indicate the end of the program. If you haven't entered any text yet, auto entry will begin after line 1000. You may enter as many new lines of text as you wish; all text will be inserted after the line number you specified. After you enter each line, the cursor will tab to the indentation determined by the previous line. Tab stops will be set every two spaces after this first tab position. The tabstop will be indented by two spaces if the previous line contains an opening brace, a case statement, or a default statement. If the line contains a closing brace, two spaces will be removed from the front of the line, and the tabstop will be set accordingly. In this manner, statement and function blocks will be automatically formatted. Once you have entered a line, you may not go back and change it while you are in auto entry mode. If you move the cursor over a line that you have already entered and press RETURN, that line will be entered as a new line without replacing the previous line (use line entry mode for editing an existing line). If you press RETURN while on a blank line, a blank line will be entered into your program. When you have finished entering text, press the BREAK key. Your entire program will be renumbered, the new text will be run through the syntax checker (see UNMATCH for a description), and the line number where your new text begins will be displayed.

COMPILE- This command first asks you for the filename to save your program to. If you have previously entered your program from a disk, then that filename will be displayed, and the cursor will be positioned at the beginning of the filename. You may use that filename, or change it to a different name. The default drive and the extension ".C" will be added if you don't specify them. You may also pass commands to the compiler by including them after the filename (separated by spaces). See the compiler documentation for a description of possible commands (pg. 7). If the filename displayed is correct, pressing the TAB key will position you one space after the filename for entering additional commands. When you press RETURN, your program will be saved, and the command line will be sent to DOS. DOS will then run the compiler (CC.COM) from the default drive. Note: CC.COM must be on the default drive for this command to work correctly. The other files/programs can be on any drive.

Example:

```
     TEST LINK RUN
```

Effect: this will save your program to the file TEST.C, compile the program, link it, and run it. If the compiler finds an error, then it will run CEDIT, and the file TEST.C will be loaded.

If an error occurs while trying to save the file from CEDIT, then the error number will be displayed, and you will be returned to command mode, rather than compiling the program.

DELETE- This is used to deleting one or more lines. You may enter one line number to delete one line, or two line numbers to delete a block of lines. You cannot recover a line which has been deleted.

ENTER- This command allows you to enter a file and either merge it with your current program, or replace it with the new program. You will be asked to give the filename of the program you wish to enter. The default drive and the extension ".C" will be added if you don't specify them. You may follow the filename with a space and a "-M" to merge the file at the end of your current program. If the file you enter contains line numbers, the line numbers will be removed. After the new program is entered, the entire program in memory will be renumbered starting at line 1000, in increments of 10. The first line number of the new section will be displayed. If the number is greater than 1000, then the new section was merged with the existing program in memory. Note: If you have written programs for the ACE-C compiler which have line numbers, you must first enter them with CEDIT to remove the line numbers before compiling them. The Lightspeed compiler does not accept line numbers.

FIND- This will ask you for the text to search for, and will then display any line in your program that contains that text. You may pause the command by pressing a key, and then restart it by pressing a key again. To stop the find command, press a key to pause the search, and then press the BREAK key. The text you enter must be at least two characters in length. If you need to search for something that includes trailing spaces, then you may end the line with a graphics heart. To get a graphics heart, first press the ESCAPE key and then press a CONTROL COMMA.

INVENTORY- This will give you an inventory (directory) of the filenames on a disk. When you give it the filespec to search for, all filenames matching that filespec will be listed in two columns on the screen.

Example:

```
D2:*.C
```

Effect: this would list all filenames with the extension ".C" found on drive number two.

LIST- This will ask you for the line numbers to list. You may optionally precede the line numbers with a filename or device (for example, "P:" to send the listing to the printer). Pressing RETURN without a line number will cause the entire program to be listed. If you enter one number, then only that line will be listed. If more than 20 lines are to be listed, then the screen will be cleared first. You may pause the listing by pressing a key, and restart it by pressing a key again. To stop the listing, press a key to pause the listing, and then press the BREAK key. Note: this command lists your programs with the line numbers, whereas the PRINT command lists the program without line numbers.

MOVE- This allows you to move one or more lines from one part of the program to another, and optionally delete the first block of lines. MOVE requires three numbers! If you only wish to move one line, then use that line number twice. The third line number is the line the text is to follow.

Example:

```
    1150 1160 1030
```

Effect: this would move a copy of lines 1150 through 1160 to the area following line 1030.

Example:

```
    1150 1150 1030
```

Effect: this would move only line 1150 to the area following line 1030.

Text is automatically renumbered after a move command, and the line number is displayed where the new text is. After the third number you may optionally add a space and a "D" (either upper or lower case) which will cause the source lines to be deleted after the move. It is not necessary for the third line to exist in your program. The move will be to the first space available following where the line number would be. Using a 900, for example, will cause the block of lines to be moved to the beginning of your program. Using a semicolon will cause the lines to be moved to the end of your program.

NEW- This will ask you if you wish to erase your entire program. If you press the "Y" key, your program will be erased. Press BREAK or any other key but "Y" if you don't want to erase your program. You cannot recover a program
which has been erased.

PRINT- This works exactly the same as the LIST command, only the line numbers are not listed. See the LIST command for details on using this command. Note: the Print command can be used if you wish to save only part of your program.

QUIT- This asks you if you want to exit to DOS. If you press a "Y", control will return to DOS. Any program in memory will be lost. If you don't want to exit, press the BREAK key, or any other key except "Y".

RENUMBER- This will immediately renumber your program starting at line 1000, in increments of 10. You cannot stop the renumber command.

SAVE- This will ask you for the filename to save your program to. The default drive and the extension ".C" will be added if you don't specify them. The screen will be turned off during the save operation. This will save your entire program. If you wish to save only part of your program, then use the PRINT command.

TEXT- This is identical to the AUTO command except that it does not do automatic indentation, and it does not check for syntax upon completion. It is normally used for creating ".LNK" files or other non-C program files. Note: when TEXT is entered, you will be in lowercase. You may enter filenames for the ".LNK" file in lowercase, as all filenames are converted to uppercase by the normalize function that LINK calls.

UNMATCH- This asks you for the line or lines to check for errors. If you press RETURN without a number, your entire program will be checked. UNMATCH will list lines with unmatched parenthesis, quotes, or apostrophes. It will also list a line that is missing a semicolon. If there are no errors, it will display the message "No errors". You may stop the check at anytime by pressing the BREAK key. UNMATCH is not perfect. It will very rarely miss an error, but it will sometimes flag a line as an error when it's not.

WHAT- This tells you the size of your program in bytes, how many bytes of memory are available, and what the last line number in your program is.

XCHANGE- This will ask you for the text to replace. After you enter that, it will ask you for the text to replace it with. After you enter that, it will ask you for the starting line number. If you want to search the entire program, then press RETURN without a line number. When a line is found containing the text to replace, the line will be listed. Underneath the line will be a pointer to the text to replace, followed by the text to replace it with displayed in inverse. You now have four options. If you press 'Y', the text will be replaced and the search will continue. If you press 'Q', you will be returned to command mode. If you press 'A', then all occurrences of the text throughout the rest of the program will be replaced. If you press any other key, the search will continue without replacing the text in the current line displayed. The text you search for must be at least two characters in length. If you need to include trailing spaces, you may end the search or replace text line with a graphics heart (ESCAPE, CONTROL COMMA).

?- This will display the menu of options available along with the filename of the
program you are editing (if the program was entered from a disk).

In addition to the regular commands, there is an additional set of commands which may be accessed by pressing the CONTROL key and the letter desired. These commands are shortcuts for typing in commonly used statements or punctuation marks. If you want to display the control letter rather than use it, press the ESCAPE key and then press the control letter. It is recommended that you try out each of the described letters to see how they work, and use the following summary as a reference rather than a full description.

[- Displays: $(
] - Displays: $) Equivalent to typing a closing brace and a

[- Displays: $(
] - Displays: $)   Equivalent to typing a closing brace and a RETURN.
/ - Displays: /*   The second time it is pressed, it displays */ .
; - Displays: ;   Equivalent to typing a semicolon and a RETURN and changing to lower case.
A - Displays: asm 0x   Switches to upper case.
B - Displays: break;
C - Displays: char
D - Displays: #define   Switches to upper case.
E - Displays: else
F - Displays: for(
H - Displays: 0x   Switches to upper case.
I - Displays: int
N - Displays: \n
P - Displays: printf("
R - Displays: return
S - Displays: switch(
W - Displays: while(

WARNING: If you use the control keys on an XL/XE with the key repeat rate sped up, you must be very careful to not press the key for very long or the line will get garbled. This is especially true of the keys that include a RETURN in their output. This is not a problem on an 800.

While in command mode, you may scroll the text by using the OPTION and SELECT keys. OPTION will move the text upward by displaying the next line after the last line listed on the screen. If there are no more lines at the end of your program, then OPTION will display the first line of your program. SELECT will move the text downward by displaying the previous line of the first line number displayed on the screen. If there is no previous line, then the last line number of your program will be displayed. Like the control keys, these two commands will be easier to understand if you actually use them rather than reading about them.

For those commands which request two or more line numbers (List, Print, Move, Delete, and Unmatch), you ma optionally enter a dash immediately followed by a function name. This will replace the first two line numbers. For example, pressing "L" for list and then typing "-MAIN" would list the entire function main() to the screen. Presssing "M" for move and then typing "-main 1200 d" would move the entire function main() to the area after line 1200. There are a few requirements for this feature to work properly. The function name may be entered in either upper or lower case; it will always be converted to lower case before any function name that has uppercase letters. The default for the List and Print commands is uppercase so that you can optionally enter a filename or device (P:) before the line number/function reference. Note that this is an easy way to save a single function to a file, or to print out a single function. Within

UTILITY FILES

The following utility files are for use with Lightspeed DOS. Only DCOPY.COM can be used with another DOS system. Many of the utility files were written in C, and the source code is included so that you can modify them, and use them as examples of C programs. DCOPY.COM is presented first, as it is probably the most commonly used. The additional files are listed in alphabetical order following the description of DCOPY.

DCOPY.COM - This contains many of the functions found in Atari's DUP.SYS file. To run, type DCOPY after the DOS prompt. DCOPY will display a menu of options, followed by the word "Command?". The menu may be redisplayed at any time by pressing the BREAK key. Commands which require a filename will display either the previous filespec used or just the default drive. The cursor will be positioned over the drive number. If the drive number is correct, than press the tab key to position the cursor after the colon.
You may invoke a command by pressing the letter to the left of the command. If you change your mind, press the BREAK key. The following is a description of the available commands.

A- Append a file. This will ask you for the filename to append, and then the filename to append the file to. Use this command when you want to run a C program under another DOS besides Lightspeed DOS. Once you have appended a runtime onto your C program it will run under other DOS systems, but will NOT run under Lightspeed DOS.

C- Copy a single file. This is the fastest way to copy a single file, and the only way to copy a file where the destination filename is different than the source filename. You will be asked for the name of the source file. If the name you enter contains any wildcards (? or *) then it will be assumed a multiple file copy is needed (see instructions for the M- command). If there are no wild cards in the source filename then you will be asked for the destination filename. If you specify different disk drive numbers, or one of the files is not a disk drive (cassette, printer, etc.) then copying will begin immediately. If the disk drive numbers are the same, then you will be prompted as to when to insert the source and destination disks. You may copy to or from single or double density disks. NOTE: do not try to copy DOS.SYS. Instead use the W- (write DOS.SYS) command.

D- Directory. This asks you for a filespec and then lists the directory of all filenames matching that filespec (same as the A option in DUP.SYS). The listing will be in two columns. You may pause the listing by pressing any key. Press a key again to continue with the listing, or press the BREAK key to abort the listing.

E- Erase a file. This asks for the filespec to erase, and then erases all files matching that filespec.

F- Format a disk. This asks for the drive number of the disk to format, and then formats the disk in that drive.

L- Lock a file. This asks for the filespec of the file(s) to lock, and then locks all files matching that filespec. You cannot write to, erase, or rename a file which has been locked.

M- Multiple file copy. This can be used for copying one or more files. It first asks for the filespec of the file(s) to copy, and then the device to copy the files to. For example, if you use D2: for the device, then all files would be copied to drive number 2. You may optionally include a 'Q' after the colon of the device; you will then be queried as to whether to copy each file. All files matching the first filespec except DOS.SYS will be copied (unless the Q option was invoked). If the drive numbers are different, then copying will proceed immediately. If the drive numbers are the same, then you will be prompted to insert source and destination disks as needed. The filenames being copied will be displayed on the screen. If you have used the Q option, then the filenames will be displayed followed by a question mark. Press the Y key to do the copy, or any other key to ignore copying the displayed files. Press the BREAK key to suspend file copying. Note: if you invoked the C command (single file copy) and the filespec you gave contained a wild card (? or *) then it will be treated just as if you had invoked the M- command.

P- Print directory. This will ask you to place your printer on-line. It will then ask for a filespec. All filenames matching the filespec you enter will be listed to the printer in a single column, and to the screen in a double column. Like the D-command, you may pause the listing by pressing any key, and then continue by pressing another key or stop by pressing the BREAK key.

Q- Quit. This will return you to DOS.

R- Rename a file. Enter the drive and filespec followed by a space and the filespec to rename it to. All filenames matching the first filespec will be renamed to the second filespec.

U- Unlock a file. This asks for a filespec and then unlocks all files matching that filespec. Unlock will allow any files previous locked to be written to, erased, or renamed.

W- Write DOS.SYS. This will display Dn:DOS.SYS, where n is the default drive. Change the drive number if needed and press RETURN. This will then create a DOS.SYS file on Dn. Any changes made previously with CUSTOM.COM will be saved with the DOS.SYS file.

Note: SPDCOPY is the Sparta version of DCOPY which can NOT be run with Lightspeed DOS.

BATCH - this program takes a command line that you create, and turns it into a .COM file which can be run repeatedly. By using the filename DO as the first command, you can cause several programs to be run. You can rename the .COM file created to AUTORUN.SYS and it will be run whenever DOS is booted.

Example:

```
     DO XEDRIVE;DIR D4:*.*
```

Effect: when you first boot the disk, this would run the program XEDRIVE, then run DIR with the command D4:*.*.

When BATCH is run, it will ask for a filename. If the file already exists, BATCH will load in the file, and display it so that you can correct it or make any changes. Note: you are limited to a single logical line of up to 120 characters.

OFF.COM - If you booted an XL or XE computer and you forgot to hold down the OPTION key to turn off BASIC, then you can run this program which will turn it off for you.

COMPACT - Many programs such as MAC/65 and the Lightspeed linker, create a program in small contiguous segments. COMPACT will combine all those segments. The resulting file will be slightly smaller, and load slightly faster. COMPACT will also remove any headers ($FFFF) except for the first one. It will read in the file, make any possible changes, and then write out the file. COMPACT may also be used to see where a program would load in memory. This is very important for programs not written using Lightspeed DOS. If a load address falls between $700 and $4000 then DO NOT run the program. At the least it could crash the computer, at worst it could destroy a disk.

CONFIG - This can be used with disk drives which are capable of switching between single and double density. When CONFIG is run it will display the density of D1: and D2: (the density of D2: will be displayed even if the drive doesn't exist). You now have two options. You may exit to DOS by pressing 'Q', or you may switch the density of either drive by pressing the drive number. CONFIG will then display the new density of the drive. CONFIG will not work correctly with Atari drives (since the 1050 isn't capable of double density). CONFIG will only change the density of drives 1 and 2, since Lightspeed DOS only recognizes two floppy disk drives, and a ramdisk must be single density. CONFIG should ONLY be run with Lightspeed DOS.

COPY - This is mainly designed as an example of file handling in C. In fact, the copy part is only functional if you have two disk drives (one of them can be a RAMDISK). You run COPY by typing COPY command filename, etc. Any single letter is interpreted as a command, and that command will work on all of the following filenames until another command is encountered. For example, COPY P TEST.C TEST.COM would protect D1:TEST.C and D1:TEST.COM. The following commands are available:

C - copy. Followed by filename, a space and the source and destination drive numbers (NOT separated by spaces).

Example:
```
    COPY C TEST.* 14
```
Effect: copies all files matching D1:TEST.* to drive number 4.

E - erases the following filenames.

P - protects (locks) the following filenames.

R - renames the following filename to the next filename.

U - unlocks the following filenames.

If COPY encounters a hyphen then the following filename(s) will be passed to DOS.

Example:
```
    COPY E D4:TEST.* C TEST.* 14 P D4:*.* - D1:DIR D4:*.*
```
Effect: Erases D4:TEST.*, copies D1:TEST.* to D4:, locks all files on D4:, runs DIR and lists directory of D4:

CUSTOM - This allows you to change some of the defaults set by DOS and the Operating System. The new defaults will be invoked immediately, and whenever SYSTEM RESET is pressed. The program is menu driven, and self-explanatory. The following changes are possible: left margin, background color, character color, TV sound during disk input/output, key repeat speed (even on an 800), and screen on/off during compiling and linking.

DIR - This is mostly a demo program for C, but it works the same as the DIR command in DOS XL and OS/A+. After typing DIR, give the drive number and filespec to search for. The screen will be cleared and any filenames matching the filespec will be listed to the screen in two columns.

DO - This program allows you to set up a series of commands which will then be entered as if you were typing them in yourself. The format is DO [command string], where command string is a logical line of up to 120 characters. Individual commands are separated by semicolons, which get converted to RETURNS.

Example:

```
DO COMPACT TEST;DIR TEST.*
```

Effect: this would compact the file TEST.COM, then list the directory of files matching TEST.*.

If DO is called without any commands following it, then the previous DO commands will be used. WARNING: DO uses memory starting at $680. If you use load a program in this area, it will wipe out any current or previous DO string. Note: if you are familiar with DOS XL, or OS/A+ from Optimized Systems, you will recognize the similarity between this version and Optimized System's version. The main difference is that you can enter up to 120 characters instead of 60, and entering DO without a command string will rerun the last DO command string.

FILECMP file1,file2 - This will compare two files, and show any differences or tell you if they are the same. If a difference is found, the conflicting bytes will be shown in decimal, ascii, and hexadecimal along with the preceding four bytes, and the following four bytes.

HEXDMP file - This will display the file in hexadecimal and ascii format. When HEXDMP is first run, it displays the file starting at position zero. You may then press RETURN to exit to DOS, or enter the position number in the file you want to look at. Entry should be in hexadecimal, or decimal if preceded by a period. You may use two numbers with a + or - sign which will do the appropriate arithmetic to get the actual position number. Example: 5C + .16 - this would display the file starting at $6C.

RAMLOAD.COM - If you have a RAMDISK, running this will copy the following files from D1: to D4: - CEDIT.COM, CC.COM, LINK.COM, STDIO.CCC, DCOPY.COM, FASTER.COM, COMPACT.COM and DIR.COM. This is a batch file, so you can modify it with BATCH.COM.

TYPE file - This lists a file to the screen 20 lines at a time.

RAMDISKS

XEDRIVE - This is for use with the Atari 130XE. It will use the extra memory as a single density RAMDISK with 499 free sectors. It may be accessed as D4:

MOSDRIVE - This is for an Atari 800 with two or three Mosaic memory boards. The program will ask how many banks you have. Enter the total number of banks. It will automatically leave bank #0 free. It will then set up a ramdisk accessed as D4:. The maximum number of sectors it will allow is 704.

ACTION!

The language ACTION! from Optimized Systems is missing several of the statements and much of the flexibility found in C. However, those limitations enable ACTION! to generate code that runs about 10 times faster than Lightspeed code. There are times when it is worth the greater effort to develop a program in ACTION! in order to gain more speed. But keep in mind that ACTION! code is not standard - it would take a lot of work to convert it to run under an ST or other computer, whereas many C programs can be compiled without modifications.

Lightspeed DOS contains a library of over 100 functions which may be called from ACTION!. The libraries STDIO.ACT, RUNTIME.ACT, and FLOAT.ACT include many of these routines, and can be used as guidelines for accessing all of the routines. Functions such as plot() and locate() will run almost twice as fast as the ACTION! library functions. Other functions such as printf (cprintf() in STDIO.ACT) provide much greater flexibility. There is, however, one very important difference between ACTION! and C. C defines a string as any sequence of characters ending with a zero. ACTION! defines a string where the first byte is it's length, and there is no ending zero. Since C string functions look for that ending zero, there is no direct compatability. Instead, if you are going to call any of the Lightspeed DOS functions which handle strings, you must first convert the string to a C string using stoc(), and when you return from the function you must reconvert the string to an ACTION! string using stoa() (both stoc() and stoa() are defined in the file STDIO.ACT). Most of the functions in STDIO.ACT do this for you, so the above procedure is needed only when you write your own functions to call the Lightspeed DOS functions. Note that this also applies to strings in quotes (stoc() and stoa() will work on strings in quotes).

Note: ACTION! also speeds up the auto key repeat. If you have sped it up using CUSTOM, you will need to slow it down again to be able to use the ACTION! editor. Use number 9 as the key speed in CUSTOM.

STDIO.ACT

STDIO.ACT provides access to about 40 of the functions available with Lightspeed DOS (note that you can only use the ".ACT" programs on this disk with Lightspeed DOS). The first part of STDIO.ACT contains three SET statements. The first two replace the LSH and RSH instructions. The new code handles 8 place shifts much faster than the original ACTION! code (example: val LSH 8). The third SET replaces the function call code, and is needed for the rest of the routines in this program. The resulting code is slightly faster than the original ACTION! code. The following is a list of the procedures and functions defined in STDIO.ACT. The description of the functions calling Lightspeed functions will be brief since they are described in full in section V. under STDIO.C. The first 6 routines are used for calling Lightspeed functions.

asetup1()
This is used with routines containing one argument. It must be the first function in your routine. If there is any code between asetup1() and the actual JSR to the Lightspeed function, than you must include the code block [$A0$1] (LDY #1) just prior to the Lightspeed function call.

asetup2()
This is used with routines containing two or more arguments. It must be called just before the JSR to the actual Lightspeed function.

stasetup()
This is used if the routine has one argument which is a string. It will first convert the string to a C string. It must come just before the JSR to the actual Lightspeed function.

stback()
This can be used if the function has one argument which is a string. It will store the value obtained from the Lightspeed function call, and convert the string back to an ACTION! string.

stoa(s)
This converts the C string s into an ACTION! string.

stoc(s)
This converts the action string in s to a C string.

--PROCEDURES--

circle(xc,yc,radius)
This draws an approximation of a circle using xc and yx as the center, and radius for the size.

clrtime()
This clears the system clock.

drawto(x,y)
This draws a line from the last plot position to x,y. Unlike it's C counterpart, it does not return a value.

fast()
This turns off the screen.

ferase(s)
This erases the file s.

flock(s)
This locks the file s.

frename(s)
This renames the file s.

funlock(s)
This unlocks the file s.

hitclear()
This clears the collision register.

normalize(name,ext)
This normalizes name with the default drive and the extension given in ext. Note that name and ext are ACTION! strings which are converted to C strings before calling the normalize routine, then converted back to ACTION! strings before returning.

plot(x,y)
This plots a point at x,y. This is about twice as fast as the ACTION! library routine. It uses the color stored in the ACTION! library color variable (color=n). Unlike it's C counterpart, it does not return a value.

putw(word,iocb)
Sends the two byte word to the iocb.

pmcolor(n,hue,intensity)
Set player color n to hue and intensity.

slow()
Turns the screen on again after a fast() call.

cprintf()
This is the same as the printf function in STDIO.C, only you can use a maximum of 8 arguments (ACTION! limitation). cprintf() will convert the format string to a C string before using it. However, if you are using any string arguments, you must first convert them to C stings using stoc(). If you intend to use the strings again, you must convert them back to ACTION! strings after the call using stoa(). Note that ACTION! does not have backslash characters, so you cannot use them in the format string. For a \n (RETURN) you can use a %n.

--FUNCTIONS--

abs(i)
RETURNS the positive value of i.

atoi(s)
RETURNS the integer value of the string s.

bgets(addr,len,iocb)
Block read of len bytes to addr from iocb. RETURNS actual length read. It does not indicate if an error occurred or not.

bputs(addr,len,iocb)
Sends len byte from addr to iocb. RETURNS true if successful, or the negative error number.

brkey()
RETURNS true if the break key was pressed, or zero if not. NOTE: the SET at the beginning of the file which redefines the function call code, also turns off the check for the break key which is why this function will work.

ciov(iocb,cmd,addr,len,ax1,ax2)
Direct call to STDIO (Operating System). Use a -1 to ignore any value. RETURNS true, or the negative error number.

console()
RETURNS 0 if no console key is pressed, 1 for START, 2 for SELECT, or 3 for OPTION.

copen(name,mode)
Opens name with mode specified. See description in STDIO.C (pg. 4) for possible modes.

dpoke(addr,word)
The same as pokec, only it RETURNS the value before the poke took place.

find(addr,len,c)
Search for c, starting at addr, for len bytes. RETURNS offset from addr if found, or -1 if not found.

fscanf(iocb,s,...)
Same as scanf, only from a specified iocb.

getchar()
RETURNS key pressed by user, and echoes it to the screen.

getdos(s)
RETURNS the next filename from the DOS command buffer.

getkey()
RETURNS key pressed by user, but does not echo it to the screen.

getw(iocb)
RETURNS a word (CARD) from iocb, or the negative error number.

gtime()
RETURNS the system clock in 60ths of a second.

hitpf(who,hitwho)
RETURNS true if player who hit playfield hitwho, else it RETURNS a zero. If hitwho is a -1, it will RETURN a 1 if player who hit any playfield.

hitpl(who,hitwho)
RETURNS true if player who hit player hitwho, else it RETURNS a zero. If hitwho is a -1, it will RETURN a 1 if player who hit any other player.

inkey()
RETURNS true if a key is waiting to be processed, else it RETURNS a zero.

isalnum(c)
RETURNS true if c is alphabetic or numeric, else it RETURNS a zero.

isalpha(c)
RETURNS true if c is alphabetic, else it RETURNS a zero.

isnumeric(c)
RETURNS true if c is numeric, else it RETURNS a zero.

isspace(c)
RETURNS true if c is a space, else it RETURNS a zero.

locate(x,y)
RETURNS point at x,y. Same as the ACTION! library function, only about twice as fast.

rand(max)
RETURNS a value from zero to max, or zero through 255 if max=0. Same as the ACTION! library routine, only about three times faster.

scanf(s, . . .)
Same as the scanf() function described in STDIO.C. Note that if strings are input, you will need to convert them to ACTION! strings with the stoa() function before you can use them. To input from a device, you must use fscanf. You can have a maximum of 8 arguments including the format string (ACTION! limitation).

toascii(i,s)
Converts the integer i to a string and places it in s.

tolower(c)
RETURNS the lowercase value of c.

toupper(c)
RETURNS the uppercase value of c.

FLOAT.ACT

FLOAT.ACT defines 16 floating point routines for use with ACTION!. By including FLOAT.ACT at the beginning of your program, you may define a floating point number as: FLOAT fp1(6),fp2(6), etc. All of the floating point functions work identically to their C counterpart. Please look up their full description in STDIO.C. The following is a brief description of the functions available.

atn(fp1,fp2) - arctangent
atof(fp,ascii) - ascii to float
clog(fp1,fp2) - base 10 logarithm
cos(fp1,fp2) - cosine
deg() - degree mode
exp(fp1,fp2,fp3) - exponentiation
fadd(fp1,fp2,fp3) - addition
fdiv(fp1,fp2,fp3) - division
fmul(fp1,fp2,fp3) - multiplication
fsub(fp1,fp2,fp3) - subtraction
ftoi(fp) - floating point to integer
itof(i,fp) - integer to floating point
log(fp1,fp2) - natural logarithm
rad() - radians
sin(fp1,fp2) - sine
sqr(fp1,fp2) - square root

Note: before you use atn(), cos(), or sin() for the first time in your program, you must first have called either deg() or rad().

RUNTIME.ACT

It is sometimes desirable to run an ACTION! program without the cartridge. If you limit yourself strictly to the routines provided in STDIO.ACT, FLOAT.ACT and RUNTIME.ACT, plus any of your own routines or other calls to Lightspeed functions, then your program can run without the cartridge. You must include the file STDIO.ACT, and then the file RUNTIME.ACT at the beginning of your program. RUNTIME.ACT replaces three more of the system functions via the SET command. The C multiplication and division routines used are slightly slower than ACTION!'s, but otherwise you shouldn't notice a difference. The following is a list of the routines which have been replaced in RUNTIME.ACT and may be used:

OPEN
PEEK
PEEKC
POKE
STICK
STRIG
CLOSE
GRAPHICS
MOVEBLOCK
POSITION
POKEC
SCOMPARE
SCOPY
SETBLOCK
SETCOLOR
SOUND
ZERO

You will note that there are none of the various Print and Input derivatives included. You can replace all of the ACTION! print and input routines with cprintf() and scanf() respectively. If you had to have the ACTION! versions, you could do something like the following for each print derivative:

```
PROC PrintIE(INT i)
  cprintf("%d%n",i)
RETURN
```

The above procedure works, but limits you to one argument and no formatting possibilities.

MAC/65

Lightspeed DOS is an excellent environment for developing programs in assembly language, provided the programs are intended to be run with Lightspeed DOS. By using macros to call the Lightspeed functions, you can create source code which is compact, understandable, and yet extremely fast. For example, FCALC.ASM contains only about 55 lines of actual code, yet with macro expansions it generates over 500 lines of code. There are three libraries provided in MAC/65 tokenized format (only MAC/65 can read these files!). FLOAT.M65 provides 16 macros for doing floating point calculations. STDIO.M65 provides macro calls to several of the functions found in STDIO.C. MACRO.M65 is a general purpose library, and also contains several equates for Lightspeed DOS.

Any of the functions in STDIO.C can be called from assembly language with a direct JSR call, or through a macro. If there are no arguments to call, just do a JSR address, where address is the address of the function (see source code for STDIO.C). If there are parameters to pass, then you have two possible approaches. The easiest approach is to include STDIO.M65 at the beginning of your file. Then use the macro ARGS for up to three absolute arguments (addresses), or IARGS for up to three immediate arguments. Now do a JSR to the function address. If the function returns a value, the lsb will be in the A register, and the msb in the X register. I suggest you look at the code for STDIO.M65 for examples of how to call functions in this manner. The second approach is to store the arguments in a stack, set the word at $C6 to point to the stack, load the Y register with a 1, and do a JSR to the function address. The stack used by STDIO.M65 is at $A0, which allows you to use some of the ACTION! functions.

Note: MAC/65 is available from Optimized Systems. It is unquestionably the best assembler available for the Atari, which is why it is the only one supported in this package. It was entrusted to write all of the code for the DOS and runtime functions containing more than 100K of source code.

FLOAT.M65

FLOAT.M65 contains 16 floating point macros which may be used with Lightspeed DOS and MAC/65. It is stored in MAC/65 tokenized format, so you can only load it with MAC/65. The parameters for the macros must be labels to floating point numbers. The easiest way to reserve space for a floating point number is to use:

```
LABEL .FLOAT 0
```

This will reserve the needed 6 bytes for floating point calculations. To print out a floating point number, you can use the macro PRINTFS described in the section STDIO.M65. Note that the macros are indentical in function to the floating point routines for use by C. The parameters and their order are the same. The following descriptions are given by way of reference. For a full description see their description under STDIO routines.

ATN fp1,fp2 - fp2=atn(fp1)
ATOF fp1,addr - converts the text stored in addr to floating point and stores it in fp1.
CLOG fp1,fp2 - fp2=clog(fp1)
COS fp1,fp2 - fp2=cos(fp1)
DEG - changes to degree mode for sine, cosine, and arctangent functions.
EXP fp1,fp2,fp3 - fp3=fp1^fp2
FADD fp1,fp2,fp3 - fp3=fp1+fp2
FDIV fp1,fp2,fp3 - fp3=fp1/fp2
FMUL fp1,fp2,fp3 - fp3=fp1*fp2
FSUB fp1,fp2,fp3 - fp3=fp1-fp2
FTOI fp1,addr - converts fp1 to an integer and stores it in addr.
ITOF ival,fp - converts ival to floating point and stores it in fp. ival must be a number, not an address.
LOG -fp1,fp2 - fp2=log(fp1)
RAD - changes to radians for sine, cosine, and arctangent functions.
SIN fp1,fp2 - fp2=sin(fp1)
FTOI fp - converts fp to an integer and returns A msb, X lsb.

STDIO.M65

STDIO.M65 contains several macros which call functions within Lightspeed DOS. It is stored in MAC/65 tokenized format, so you can only load it with MAC/65. By following the examples of the macros within this file, you can create a macro to call any of the functions described in STDIO.C. If the function returns a value, then A will have the lsb, and X will have the msb. In the following descriptions either [IMMEDIATE] or [ABSOLUTE] may follow the macro name. IMMEDIATE means that the arguments must be actual numbers, not addresses where the numbers are stored. ABSOLUTE means that the arguments must be the address where the value can be found. The exception is string arguments which may be either the address of the string, or the string itself in quotes. If you give the address of the string, be sure the string ends with a zero. The following descriptions are for a reference. For a full description, see the section under STDIO. I/O functions will usually return with Y=1 for a good return, or Y=error #.

BGETS addr,len,iocb [IMMEDIATE] - block input

BPUTS addr,len,iocb [IMMEDIATE] - block output

CLEAR addr,len [IMMEDIATE] - set block of memory to zero.

CLRTIME - clear System clock.

COLOR n - sets the color to be used by graphics functions.

CONSOLE - RETURNS A=0 if no console key is pressed, 1 if START, 2 if SELECT, or 3 if OPTION.

DRAWTO x,y [ABSOLUTE]

FAST - turns off screen.

FERASE filespec - erases filespec.

FLOCK filespec - locks filespec.

FRENAME filespec - renames filespec.

FUNLOCK filespec - unlocks filespec.

GETDOS string - gets next filename from DOS command string. The string argument passed should be and address with enough space to pass the largest command available from DOS (normally 16 bytes).

GETSEC addr,sector,drive [ABSOLUTE] - read sector. Note: ALL arguments must be the address where the value can be found.

GRAPHICS mode [IMMEDIATE]

GTIME - RETURNS the System clock in 60ths of a second, A=low, X=high.

LOCATE x,y [ABSOLUTE] - RETURNS value in A register.

MOVE from,to,len [IMMEDIATE] - moves block of memory. Will correctly handle overlapping data.

NOTE iocb,sector,byte [ABSOLUTE] - Gets position on disk.

NORMALIZE name,ext [ABSOLUTE] - name and extension can be in quotes, or addresses where the strings can be found.

PLOT x,y [ABSOLUTE]

POINT iocb,sector,byte [ABSOLUTE] - positions place on disk.

POSITION x,y [IMMEDIATE]

PRINTF [ABSOLUTE] - Unlike C's printf, this will not print to an iocb number. The first argument must be the format string, either in quotes, or the address where it can be found. Note that all the rest of the arguments are addresses where the value can be found. This will NOT print strings or floating point numbers (see PRINTFS)! Note that there are no backslash characters, but you can use a %n for a RETURN.

PRINTFS [ABSOLUTE] - This is used for printing strings or floating point numbers.

PUTSEC addr,sector,drive [ABSOLUTE] - writes addr to sector on drive.

RND max [IMMEDIATE] - RETURNS a random number from 0 to max (not inclusive of max, i.e. 5 will generate any number between 0 and 4). Use zero to generate 0-255 inclusive.

SCANF [ABSOLUTE] - Unlike C's scanf, you cannot use an iocb number. The maximum number of arguments is 5.

SLOW - turns on screen after a FAST call.

SOUND voice,distortion,pitch,volume [IMMEDIATE]

The following three macros are used for doing multiplication and division on integers. The third parameter is optional; if included, then the result will be stored in it. If there is no third parameter, the result is in A=low, X=high.

IMUL n1,n2 (,n3) [ABSOLUTE] - signed multiplication of the word n1, by n2. RETURNS result in A=low, X=high, or **n3** if specified.

IDIV n1,n2 (,n3)[ABSOLUTE] - signed division of the word n2, by the word n1. RETURNS result in A=low, X=high, or **n3** if specified.

IREM n1,n2 (,n3) [ABSOLUTE] - RETURNS remainder of division of n1 by n2 in A=low, X=high, or **n3** if specified.

MACRO.M65

MACRO.M65 contains several equates to Lightspeed DOS, and several general purpose macros. It can only be loaded and used by MAC/65 as it is stored in tokenized format. The following is a list of the equates:

STDIO - Operating System's Central Input Output routine address.

SIO - Operating Systems Serial Input/Output address.

EOL - Return character.

PUTEDIT - a JSR to this address will send the contents of the A register to the screen.

PUTSCREEN - a JSR to this address will send the contents of the A register to the graphics screen (same as plot).

GETKEY - a JSR to this address will return in the A register the ascii value of the key pressed by the user in the A register. To be safe, use the macro GETKEY which zeros $2A first.

GETSCREEN - returns in the A register the graphics screen point from the current position (same as locate).

MESSAGE - Will send the text pointed to by A,X to the screen. See the macro DISPLAY for use of this.

ERROR - a JSR to this address will send to the screen "Error #n" where n is the number in the Y register.

MNUMBER - this will send to the screen in ascii format the integer in $D4 (FR0).

CLOSEALL - a JSR to this address will close iocb's 1-7.

DOSENTRY - address for entering DOS ($0A).

GETFILENAME a JSR to this address will get the next filename in the DOS command buffer and store it in FULLFILENAME. If the first byte of FULLFILENAME is a zero, then no file was found in the command buffer. This routine is equivalent to the getdos() function described in the documentation for STDIO.C.

DEFAULTDRIVE - this is the address of the 3-byte default drive (Dn:).

POINTER - this is the pointer to the DOS command buffer. Changing this pointer would affect what filenames are input from the

GETFILENAME and getdos() routines.

FULLFILENAME - this contains the filename retrieved from the DOS command buffer and normalized with the default drive. If the first byte is zero, then no filename was found.

COMMANDBUFFER this is a 120 byte area storing all the commands passed when DOS calls a file.

The following is a list of the macros available with MACRO.M65. If an argument is given in brackets, then the argument is optional.

READ addr,[iocb] - read a record into addr, use iocb #0 if none included.

WRITE addr,[iocb] - write a record to iocb. If iocb is not included, then the screen is assumed (iocb #0).

GET iocb - get a byte from iocb into the A register.

PUT iocb - put the byte in the A register to the iocb.

SCREEN - send 1 to 5 bytes to the screen.

Example:
```
    SCREEN 'H,'I,EOL
```
Effect: this would send HI and a RETURN to the screen.

ADD - All arguments are addresses of word values. If one argument, the value is increased by one. If two arguments, the second value is added to the first value, and the result stored in the first value. If three arguments, the second value is added to the first value and the result stored in the third value.

IADD - same as ADD only you must have at least two arguments, and the second argument is an immediate value rather than an address.

SUBTRACT - All arguments are addresses of word values. If one argument, the value is decreased by one. If two arguments, the second value is subtracted from the first value, and the result stored in the first value. If three arguments, the second value is subtracted from the first value and the result stored in the third value.

CLOSEALL - closes iocb's 1-7.

TRANSFER from,to - transfers the contents of the word **from** to the word to.

ISUBTRACT - same as SUBTRACT only you must have at least two arguments, and the second argument is an immediate value rather than an address.

DISPLAY text - This sends the text and up to three additional bytes to the screen. The text argument can be in quotes, or the address of where the text can be found. If you use an address, the string must end with a zero.

Example:

```
     DISPLAY "Hello World!",EOL
```

Effect: This would send the text Hello World! to the screen followed by a return.

GETKEY - this will return in the A register the ascii value of the next key pressed.

DPOKE addr,val - this will store the immediate word val into the word at address.

DZERO - this will store a zero into 1-5 WORDS.

ZERO - this will store a zero into 1-5 BYTES.

ERROR - this will display "Error #n" where n is the number in the Y register.

DOEDIT text - this will set up the text in the same manner as described in DO.COM (pg. 42). The text may be in quotes, or the address of the text. If using an address, the last byte must be a zero.

SETUPEDIT text - this will run the DO editor on the text previous converted by DOEDIT. See the documentation for DO.COM (pg. 42) for a description.

MISCELANEOUS

QSORT - This is an example of how you can use pointer arrays (even though technically they aren't implemented). The program sorts records and tosses out duplicates.

FCALC - This is a simple floating point calculator. Source codes are given in C, ACTION! and ASSEMBLY language (FCALC.ASM is in list form, but can only be assembled with MAC/65). It is intended only to show how to use floating point in the various languages.

LIFE - The game of LIFE, invented by the mathematician John Horton Conway, during the 1960's. For each generation the following rules apply:

1) Each live cell that touches less than one neighbor, will die from loneliness.
2) Each live cell that touches more than 3 neighbors will die from overcrowding.
3) Each empty cell that touches exactly 3 neighbors, will give birth to a live cell.

Try the following patterns:

The R pentomino:

```
 XX
XX
 X
```

A glider:

```
 X
  X
XXX
```

Z - glider spawner:

```
XXXXX
   X
  X
 X
XXXXX
```

A Practice Session

This section should be read while at your computer so you can try out the examples. It is designed to give you a hands on experience at writing C programs using the Lightspeed C development System. In the descriptions, the computer's responses will be given in bold type.

The first thing to do is to make a work disk. If you haven't made a backup disk yet, make one now and then boot the back up disk. Type DCOPY and press RETURN. Insert a blank disk into drive #1. Press F for format, enter D1:, and press RETURN. Next press W and again enter D1: to write the DOS.SYS files. Now you need to copy some files from your backup disk. The method used will vary depending on whether you have two drives, a ramdisk, a single drive, etc. Use the C option in DCOPY to copy the files. DCOPY will prompt you to insert source and destination disks if you have a single drive. Copy the following files to your new work disk:

```
CC.COM
LINK.COM
FASTER.COM
CEDIT.COM
STDIO.CCC
GRAPHICS.CCC
```

You could certainly copy other files such as DIR.COM and DCOPY, but the above files are essential for writing C programs.

You are now ready to write your first C program. Insert the work disk and type CEDIT and press RETURN. You will be presented with a menu of options. Press A and the computer will respond with:

Line 1000+ press BREAK when done Now type the following three lines:

```
main() $(
printf("\fHello World!\n");
$)
```

Press the break key. Note that CEDIT automatically indented the second line for you since the first line had an opening brace. Now press L and when you are asked for line numbers, just press RETURN. Your three lines will be displayed on the screen preceeded by line numbers. Check to make sure you didn't make any typos (if you did, move the cursor to the error, correct it, press RETURN, and press the BREAK key). Press C to compile the program. Type the following line:

```
TEST LINK RUN
```

CEDIT will now save the file to D1:TEST.C and exit to DOS. Next the compiler is loaded which compiles your program, then the linker links your program, then your program is run which clears the screen and displays "Hello World!".

Now type CEDIT TEST and press RETURN. The editor will be run and it will load your program. Press L to list your program (just press RETURN when it asks for line numbers). Now move the cursor over the first quote in printf(" and delete it. Press RETURN and press the BREAK key. Now press U and when it asks for the line numbers, just press RETURN. The computer responds by listing the lines with the erros, and what the errors were. The editor will catch errors like missing quotes, parenthesis and apostrophes if you ask it to. Let's try to compile it anyway. Press C. Note that this time the computer responded with:

Enter>TEST

This is because this was the program you loaded. Press the tab key and type
LINK RUN and press RETURN. The compiler will now run and will display:

missing bracket

Press 1 to tell the compiler that CEDIT is on D1: (if CEDIT is on the default drive, you could just press RETURN). The editor will be loaded along with TEST.C and it will display the same error message as the compiler did.

Note that in none of the above did we have to worry about creating a .LNK file. As long as your program is on one source file, and the only additional functions you call are in STDIO.CCC, then you don't need to make a .LNK file since LINK will make one for you. You might enter TEST.LNK at this point just to see the .LNK file that was created. Also, if you listed the directory of your disk you would find the following files:

TEST.C - your source program.
TEST.CCC - created by CC for LINK.
TEST.LNK - the LINK file.
TEST.COM - your program, ready to run.

May I suggest you try compiling and running SIEVE.C with and without using FASTER.COM to get an idea of the difference in speed. SIEVE.COM will take about 30 seconds to run even with the screen off. You might also want to compile some of the other source programs just to get an idea of how the whole process works.

APPENDIX A

This appendix is for use with Appendix A in the book "The C Programming Language" by Kernighan and Ritchie. It points out the differences between the Bell Labs implementation of C and Lightspeed C.

1. No major hardware differences.

2. Tabs are not considered white space.

2.1 Comments cannot be imbedded within a line. A comment does not continue to the next line even if there is no closing */.

2.2 8 characters, 2 cases.

2.3 Additional keywords: asm, jsr.

2.4.1 No long constants.

2.4.2 No long constants.

2.4.3 \f - clears the screen (not a form feed). \r - cursor right (not a carriage return). No bit patterns. Additional constants:
   \d - cursor down
   \e - escape key
   \g - bell
   \l - cursor left
   \m - control M
   \u - cursor up
A backslash followed by a number from 0 to 256 (may be decimal, octal, or hex) will be changed to it's character value. (\65 == 'A).

2.4.4 No floating constants.

2.5 Strings may not continue over a logical line (up to 120 characters)

2.6 char 8 bits
   int 16 bits
   float 48 bits (6 member char array)
No other data types supported.

4. Only automatic and external storage classes supported. Data types: char, int, pointers, single dimension arrays. No structures, unions or bit fields.

5. No change.

6. All operands are converted to type int. A char is not sign

extended.

6.1 No short or long integers.

6.2 Floating point is implemented through floating point functions operating on 6 member char arrays.

6.3 ftoi() and itof() is used for conversion between floating point and integer. (See section describing Floating Point).

6.4 No change.

6.5 No unsigned (except char pointers).

6.6 All operands are converted to int.

7. No check for division by zero.

7.1 No change. (No structures or unions, and no operators applying to them).

7.2 No casts or sizeof.

7.3 - 7.15 No changes.

8.1 - extern is the only storage class specifier which may be used.

8.2 Type specifiers: char, int.

8.3 No change.

8.4 Functions are all type integer and may not be declared.

8.5 No structures or unions.

8.6 No initialization.

8.7 Not implemented.

8.8 Not implemented.

9 - 9.13 No change except labeled statements must be preceeded by a colon.

10 - 10.2 No change.

11.1 Identifiers declared within a block exist throughout the function they are defined in.

11.2 Functions exist throughout the program.

12.1 No macros.

12.2 #includes may not be nested.

12.3 #if - no constant expression. #ifdef, #ifndef - not implemented. #ifref - true if identifier has been defined (identifier may be a global or a function) within the current source file. Conditionals may not be nested.

12.4 Not implemented.

13. extern must be followed by type. A type definition outside of any function is global (static), within a function it is auto.

14.1 Not implemented.

14.2 Not implemented.

14.3 Only single dimension arrays.

14.4 Pointers are 16 bits and may be passed from int to char pointer to int pointer. (A function returning a pointer, actually returns an integer).

15. No change.

16. Function arguments are evaluated left to right. Multi-character constants not implemented.

17. Not supported.