****
* This specification reflects the latest information available at *
* this time. Please be advised that updates will occur as the *
* system evolves and when implementation is completed. *
****

C-128 SOFTWARE FUNCTIONAL SPECIFICATION

Version 3.0

4/18/85

Approval _____ Date: _____

Approval _____ Date: _____

Approval _____ Date: _____

Acknowledgement:

This specification represents the contributions of several people including: Fred Bowen, Dave Haynie, Von Ertwine, Terry Ryan, Dave Siracusa, Dave Stong, and Julian Strauss.

CONTENTS

CHAPTER 1

ABSTRACT

1.1 ABSTRACT

This document is the functional specification of the software features to be provided in the C-128 product. References are made to the hardware to the extent necessary to describe the support required from hardware to implement the functions made available through the software. For detailed information on the hardware refer to the C-128 Hardware Specification.

CHAPTER 2

INTRODUCTION

2.1 MACHINE CONCEPT

The C-128 is intended as an upgrade and successor to the commercially successful C64. The C-128 takes advantage of the already developed base of C-64 software by providing a totally C-64 compatible mode of operation. In addition it provides an upgrade path by implementing features that are normally found in much more expensive machines. Specifically these features are:

1. 128K bytes of user accessible RAM
2. 80 character column output
3. CP/M 3.0 operation

There are other features that are being added, but the ones listed above are most significant in terms of value added versus the C-64.

2.2 SOFTWARE/HARDWARE SPECIFICATION OVERVIEW

The software/hardware features provided by the C-128 machine are summarized below:

C-64 Compatible Mode

1. Standard C-64 Kernal
2. BASIC 2.0
3. 40 column output via VIC II chip - (modified 1 or 2 MHz clock, extra keyboard lines, etc.)
4. Sound via SID chip
5. Access to 64K bytes of RAM
6. Standard C-64 keyboard layout
7. Full compatibility with C-64 peripherals including standard C-64 cassette, joystick,user port, and serial bus devices
8. Full compatibility with C-64 applications software
9. C-64 composite video and RF output
10. 8500 CPU at 1 MHZ

C-128 Mode

1. New enhanced C-128 kernal
2. BASIC 7.0
3. 40 column output via VIC II
4. 80 column output via 8563 chip
5. Sound via SID chip
6. Access to 128K bytes of RAM
7. Enhanced keyboard (numeric pad,escape,tab,caps lock,help key) in addition to standard C-64 keyboard.
8. Access to fast serial floppy disk and regular serial peripherals
9. Access to RAM disk when 256K x 1 RAM are available(not in current design)

CP/M Mode

1. CP/M 3.0 via integral Z-80A
2. 40 column output via VIC II chip

3. 80 column output via 8563 chip
4. Sound via SID chip
5. Access to 128K bytes of RAM
6. Access to full keyboard
7. Access to fast serial disk and regular serial peripherals
8. Access to RAM disk when 256 x 1 RAM are available (only in CP/M design)

2.3 HARDWARE COMPONENT SUMMARY

(For greater detail see C-128 Hardware Specification)

Processors:	8500	(C64,C-128 Modes,I/O support for CP/M)
	Z80-A	(CP/M Mode only)
Memory:		
ROM:	64K standard	(C-64 Kernal + BASIC, C-128 Kernal + BASIC, character ROMs and CP/M BIOS)
	1-32K slot available for function key software	
RAM:	128K bytes in 2-64K byte banks	
	16 K bytes screen RAM for 8563 video chip	
Video Chips:	8567/856x	40 column video (version for NTSC and PAL TV standards)
	8563	80 column video
Sound:	6581	SID Chip
I/O:	6526	Joystick ports/keyboard scan/cassette
	6526	User and serial ports
Memory Management:	PLA	(C-64 + C-128 Mapping Modes)
	MMU	(Custom gate array)

2.4 COMPATIBILITY OBJECTIVE

The C-128 system is designed as an upgrade to the C-64. The prime objective is to maintain hardware and software compatibilty with the C-64 when operating in C-64 Mode. The C-64 mode of the C-128 will be capable of running all C-64 application software. It also will support all C-64 peripherals except the external CP/M card (the C-128 has internal CP/M capability that supercedes that provided by the external card).

The C-128 mode is designed as a compatible superset to the C-64. Specifically, all kernal functions provided by the C-64 will be provided in the C-128 kernal. They are also provided at the same locations in the jump table of the C-128 kernal to allow compatibility with BASIC programs, However, locations within kernal/BASIC that may have been called directly will not be guaranteed to be at the same locations as in the C-64 kernal. Where possible new entries will be added to the C-128 jump table to replace these direct calls to kernal and BASIC routines. An attempt will be made to maintain zero page and other system variables at the same addresses they occupy in C-64 mode. This will simplify interfacing for many programs.

The objective to provide full C-64 compatibility also leads to certain constraints. The main constraint is that the C-64 Mode cannot take advantage of all the new features of the machine (this is why the C-128 mode was added). For example, a new fast serial disk will be

part of the C-128 system. Due to compatibility and memory constraints it is not possible to modify the C-64 kernal to support the new fast serial disk drive. C-64 will see this disk as a standard serial disk. Similarly, the C-64 mode does not have an 80 column screen editor for the same reasons. Also, C-64 BASIC does not automatically use the second 64K bank of memory. This second bank is used for variable storage by C-128 BASIC.

CHAPTER 3

C-64 COMPATIBLE MODE

The C-64 kernal and BASIC are included unchanged in the C-128 system to provide complete compatibility with C-64. For a complete description of the following items refer to the C-64 Programmer's Reference Guide.

1. C-64 BASIC commands, statements, and functions
2. How to use the VIC chip
3. How to use the SID chip
4. C-64 Kernal function accessible through jump table

Note: in addition to the 46 registers that can be accessed in the VIC chip there are 2 additional registers accessible to the user for the C-128. They are:

Reg. 47 - Keyboard Control Register

The keyboard control register determines the status of the three keyboard interface lines. Bits 0-2 of register 47 are directly reflected in lines K0-K3. Bits 4-7 are not connected and return a high when read.

Reg. 48 - 2 MHZ BIT

The 2 MHZ bit, bit 0 of register 48, sets the speed of the 2 MHZ clock out. This bit also stops all VIC accesses, except DRAM refresh, from occurring. VIC cannot generate any address except during these refresh cycles. For normal VIC operations tthis bit must be clear.

CHAPTER 4

MODE SWITCHING SUMMARY

The following table shows how to switch between the various C128 modes. The columns of the table show the current state the user is in. The rows of the table show the state the user would like the system to be in. To change the state of the machine the user should locate the column containing the state the machine is currently in. Then the user should find the row containing the state they would like the system to be in. The intersection of this column and row contains a brief description of the action the user should take to change states of the machine.

C128 MODE SWITCHING

CURRENT STATE / DESIRED STATE	OFF	C64	C128 80 COL	C128 40 COL	CP/M 40 COL	CP/M 80 COL
OFF	--	TURN POWER OFF	TURN POWER OFF	TURN POWER OFF	TURN POWER OFF	TURN POWER OFF
C64	TURN ON WITH C64 CARTRIDGE	--	GO64 COMMAND	GO64 COMMAND	TURN OFF SEE OFF STATE	TURN OFF SEE OFF STATE
C128 80 COL	TURN ON W/80 COL SWITCH SET	RESET W/80 COL SWITCH SET	--	ESC-X OR SET 40 COL SWITCH AND RESET OR RUN/STOP+ RESTORE	CYCLE PWR W/80 COL SWITCH SET	CYCLE PWR W/80 COL SWITCH SET
C128 40 COL	TURN ON W/40 COL SWITCH SET	RESET W/40 COL SWITCH SET	ESC X OR SET 40 COL SWITCH AND RESET OR RUN/STOP+ RESTORE	--	CYCLE PWR W/40 COL SWITCH SET	CYCLE PWR W/40 COL SWITCH SET
CP/M 40 COL	AUTO BOOT DISK IN DRIVE+40 COL SW SET	RESET WITH AUTO BOOT IN DRIVE+ 40 COL SW	BOOT CMD OR RESET+ AUTO BOOT+ 40 COL SW	BOOT CMD OR RESET+ AUTO BOOT+ 40 COL SW	--	ASSIGN VIA DEVICE CMD TO 40COL
CP/M 80 COL	AUTO BOOT DISK IN DRIVE+80 COL SW SET	RESET WITH AUTO BOOT IN DRIVE+ 80 COL SW	BOOT CMD OR RESET+ AUTO BOOT+ 80 COL SW	BOOT CMD OR RESET+ AUTO BOOT+ 80 COL SW	ASSIGN VIA DEVICE CMD TO 80COL	--

NOTES:

1. Pressing the reset switch reinitializes the machine in the same way as cycling power.
2. The "RUN/STOP + RESTORE" sequence means that the "RUN/STOP" is held down while the "RESTORE" key is momentarily pressed. This performs a subset of the operations performed by cycling power or pressing the reset switch.
3. An "AUTO BOOT DISK" is a special diskette that when loaded into a disk drive contains a program which will automatically load and run when the computer is powered on.
4. A C64 cartridge takes precedence over an auto boot disk at system power up. If a C64 cartridge is plugged in, the system always powers up in C64 mode.
5. In C128 mode the escape sequence ESC X toggles between 40 and 80 column independent of the state of the 40/80 column switch.
6. See the GRAPHIC command in BASIC for other ways of toggling 40 and 80 column screens.
7. Under certain conditions both 40 and 80 column screens can be made to appear active at the same time. Details on how this is done will appear in a later version of this spec.
8. In C64 mode the system MMU is not accessible. However, the RAM bank the system executes from will be that selected in the MMU before the C64 kernel and BASIC ROMS were enabled.

CHAPTER 5

C64/C-128 MODE

5.1 BASIC FUNCTIONS

This section lists BASIC 7.0 commands, statements, and functions by group. It gives a complete list of the rules (syntax) of the C-128 BASIC 7.0, along with a concise description of each. Within the section, the commands, statements, and functions are listed in alphabetical order. Commands are used mainly in direct mode, while statements are most often used in programs. In most cases, commands can be used as statements in a program if prefixed with a line number. The user is able to use many statements as commands by issuing them in direct mode (i.e., without line numbers).

The different types of operations in BASIC are listed in sections based on the following criteria:

1. COMMANDS: the commands used to work with the programs, edit, store, and erase them.

2. STATEMENTS: the BASIC program statements used in numbered lines of program.

3. FUNCTIONS: the string, numeric, and print functions.

4. VARIABLES AND OPERATORS: the different types of variables, legal variable names, and arithmetic and logical operators.

5. BASIC AND DOS ERROR MESSAGES

5.1.1 COMMAND AND STATEMENT FORMAT

The commands and statements presented in this section are governed by consistant format conventions designed to make them as clear as possible. In most cases, there are several actual examples to illustrate what the actual command looks like. The following example shows some of the format conventions that are used in the BASIC commands

```
EXAMPLE :     DLOAD"program name"[D0, U8]

                                   additional arguments

              keyword  argument    (possibly optional)
```

The parts of the command or statement that the user must type in exactly as they appear are in capital letters. Words that don't have to be typed exactly, such as the name of the program, are not capitalized. When quote marks (" ") appear (usually around a program or file name), the user should include them in the appropriate place according to the format example.

KEYWORDS, also called RESERVED WORDS, appear in uppercase letters. THESE KEYWORDS MUST BE ENTERED EXACTLY AS THEY APPEAR. However, many keywords have abbreviations that can also be used.

Keywords are words that are part of the BASIC language that the computer understands. Keywords are the central part of a command or statement. They tell the computer what kind of action to take. These words cannot be used as variable names.

ARGUMENTS (also called parameters) appear in lower case. Arguments are the parts of a command or statement; they complement keywords by providing specific information about the command or statement. For example, a keyword tells the computer to load a program, while the argument tells the computer which specific program to load and a second argument specifies which drive the disk containing the program is in. Arguments include filenames, variables, line numbers, etc.

SQUARE BRACKETS [] show OPTIONAL arguments. The user selects any or none of the arguments listed, depending on the requirements.

ANGLE BRACKETS <> indicates that the user MUST choose one of the arguments listed.

VERTICAL BAR | separates items in a list of arguments when the choices are limited to those arguments listed, and no other arguments can be used. When the vertical bar appears in a list enclosed in SQUARE BRACKETS, the choices are limited to the items in the list, but still have the option not to use any arguments.

ELLIPSIS ..., a sequence of three dots, means that an option or argument can be repeated more than once.

QUOTATION MARKS " " enclose character strings, filenames, and other expressions. When arguments are enclosed in quotation marks in a format, the quotation marks must be included in a command file or statement. Quotation marks are not conventions used to describe formats; they are required parts of a command or statement.

PARENTHESES () When arguments are enclosed in parentheses in a format, they must be included in a command or statement. Parentheses are not conventions used to describe formats; they are required parts of a command or statement.

VARIABLE refers to any valid BASIC variable name such as X, A$, or T%.

EXPRESSION means any valid BASIC expression, such as A+B+2 or .5*(X+3).

5.1.1.1 ALPHABETICAL LIST OF COMMANDS -

1. APPEND
2. AUTO
3. BACKUP
4. BEGIN/BEND
5. BLOAD
6. BOOT
7. BSAVE
8. CATALOG
9. CONCAT
10. COLLECT
11. CMD
12. CONT
13. COPY
14. DELETE
15. DIRECTORY
16. DLOAD
17. DSAVE
18. DVERIFY
19. FAST
20. FETCH
21. HEADER
22. HELP
23. KEY
24. LIST
25. LOAD
26. MONITOR
27. NEW
28. PLAY
29. RENAME
30. RENUMBER
31. RUN
32. SAVE
33. SCRATCH
34. SLEEP
35. SLOW
36. SOUND
37. SPRCOLOR
38. SPRDEF
39. SPRSAV
40. STASH
41. SWAP
42. VERIFY
43. WIDTH
44. WINDOW

5.1.1.2 COMMAND DESCRIPTION -

1. APPEND - file append

```
APPEND # logical_file_number,file_name
         [,Ddrive number][<ON|,>Udevice]
```

Opens file file_name for writing, and positions the file pointer at the end of the file. Subsequent PRINT# logical_file_number statements will cause data to be appended to the end of this file.

2. AUTO - enable/disable automatic line numbering

```
AUTO [line#]
```

Turns on the automatic line numbering feature which eases the job of entering programs by typing the line numbers for the user. As each program line is entered by pressing RETURN the next line number is printed on the screen, with the cursor in position to begin typing that line. The [line#] argument refers to the increment between line numbers. AUTO with NO ARGUMENT turns off auto line numbering, as does RUN. This statement is executable only in direct mode.

EXAMPLES:

```
AUTO 10   automatically numbers line in increments of ten.
AUTO 50   automatically numbers line in increments of fifty.
AUTO      turns off automatic line numbering.
```

3. BACKUP - drive to drive disk backup

```
BACKUP Ddrive_number TO Ddrive_number
       [<ON|,>Udevice]
```

This command copies all the files on a diskette to another on a dual drive system. A copy onto a new diskette can be done without first using the HEADER command to format the new diskette because the BACKUP command copies all the information on the diskette, including the format. BACKUP should always be used in case the original diskette is lost or damaged. Because the BACKUP command also HEADERS diskettes, it destroys any information on the diskette onto which the information is being copied. So if backing up onto a previously used diskette, make sure it contains no programs that are meant to be kept. See also the COPY command. Default is unit 8.

NOTE: This command can only be used with a dual disk drive.

EXAMPLES:

```
BACKUP D0 to D1          Copies all files from the disk in
                         drive 0 to the disk in drive 1.
                         (in disk drive unit 8).

BACKUP D0 TO D1, ON U9   Copies all files from drive 0 to
                         drive 1 in disk drive unit 9.
```

4. BEGIN/BEND

BEGIN/BEND are used to define a block of code which is considered by the IF statement to be one statement.

The normal usage of IF/THEN/ELSE would be along the following lines:

IF boolean THEN statement(s) : ELSE statement(s)

The main restriction is that the entire body of the IF/THEN/ELSE construct can only occupy one line. BEGIN/BEND allows either the 'THEN' or the 'ELSE' clause to run on for more than one line.

```
IF boolean THEN BEGIN : statements....
 statements...
 statements...  BEND : ELSE BEGIN
 statements...
 statements...  BEND
```

Remember, however, that this is only a way to extend the body for more than one line: all other 'IF/THEN' rules apply. For example:

```
100 IF x=1 THEN BEGIN : a=5
110 :  b=6
120 :  c=7
130 BEND : print "ah-ha!"
```

In the above example, "ah-ha!" would be printed ONLY if the expression 'x=1' is TRUE. This is consistant with the IF/THEN statement's policy of executing ALL statements up to the end of the line if the expression evaluates to 'TRUE' (unless an else is encountered).

5. BLOAD - loads a binary file into any memory location

```
BLOAD filename [[<ON|,>Udevice],Bbanknumber][Pstart]

      where filename is the name of your file,
            banknumber lets yo selectone of the
            16 banks,
            Pstart is the location to start loading
```

6. BOOT - load and execute program

```
BOOT file_name [,Ddrive_number][<ON|,>Udevice]
```

Loads the executable binary file file_name from device_number (default:U8), and begins execution at its starting address. BOOT with no arguments goes to the boot disk and loads (in the case of CP/M)

7. BSAVE - Saves a binary file from any specified memory location

```
BSAVE "filename"[,Bbanknumber][,Pstart][TO Pend]
                [<ON|,>Udevice]

     where filename is the name of the file to save,
           banknumber lets you select on of the 16 banks,
           Pstart is the location in the bank beginning
           the information to be saved,
           Pend is the location in the bank ending the
           information to be saved (last addr + 1).
```

8. CATALOG - same as DIRECTORY command

```
CATALOG [Ddrive_number][<ON|,>Udevice],
                       [wildcard_string]
```

9. COLLECT - free inaccessable disk space

```
COLLECT [Ddrive_number][<ON|,>Udevice]
```

Use this command to free up space allocated to improperly closed files and delete references to these files from the directory. Defaults to unit 8.

EXAMPLE:

```
COLLECT D0
```

10. CONCAT - merge files

```
CONCAT [Ds]"sourcefile" TO [Dd],"destfile"
       [<ON|,>Udevice]
```

Concatenates (merges) two data files. Here "s" represents the drive number of the disk drive containing the "sourcefile, "d" is the drive number of the "destfile", and "Udevice" is the drive unit number if more than one drive on the same unit is present.

11. CMD - redirect screen output

```
CMD logical_file_number [,write list]
```

CMD sends the output which normally would go to the screen (i.e. PRINT statement, LISTS, but not POKEs into the screen) to another device instead. This could be a printer, or a data file on tape or disk. This device or file must be OPENed first. The CMD command must be followed by a number or numeric variable refering to the file. The write list when specified, is sent to the logical file.

EXAMPLE:

```
OPEN 1,4     OPENS device #4, which is the printer.
CMD 1        All normal output now goes to the printer.
LIST         The LISTing goes to the printer,
             not the screen - even the word READY.
PRINT#1      Set output back to the screen.
CLOSE 1      Close the file.
```

12. CONT - continue program execution after STOP key depressed

```
CONT
```

This command is used to restart the execution of a program that has been stopped by either using the STOP key, a STOP statement, or an END statement within the program. The program will resume execution where it left off. CONT will not work if lines have been changed or added to the program (or even just moved the cursor to a program line and hit RETURN without changing anything), if the program stopped due to an error, or if the user caused an error before trying to re-start the program. The error message in this case is CAN'T CONTINUE ERROR.

13. COPY - copy files between devices

```
COPY [Ds,]"sourcefile" TO [Dd,]["otherfile]
     [<ON|,>Udevice]
```

COPYs a file on the disk in one drive (the source file) to

the disk in the same or other on a dual disk drive only, or creates a copy of a file on the same drive (with a different file name). Note: copying between units cannot be done.

EXAMPLES:

```
COPY D0, "test" TO D1, "test prog"
   Copies "test" from drive 0 to drive 1,
   renaming it "test prog" on drive 1.

COPY D0, "STUFF TO D1, "STUFF"
   Copies "STUFF from drive 0 to drive 1.

COPY D0 to D1
   Copies all files from drive 0 to drive 1.

COPY "WORK.PROG" TO "BACKUP"
   Copies "WORK.PROG" as a program called
   "BACKUP" on the same drive.
```

14. DELETE - deletes lines of BASIC text.

DELETE [first line#] [-[last line#]]

Deletes lines of BASIC text. This command can be executed only in direct mode.

EXAMPLES:

DELETE 75	Deletes line 75.
DELETE 10 - 50	Deletes line 10 through 50 inclusive.
DELETE - 50	Deletes all lines from the beginning of the program up to and including line 50.
DELETE 75-	Deletes all lines from 75 on to the end of the program.

15. DIRECTORY - display contents of disk directory on screen

DIRECTORY [Ddrive_number][<ON|,>Udevice][,filespec]

Displays a disk directory on the C-128 screen. Use CONTROL-S or SCROLL KEY to pause the display (any other key restarts the display after a pause). Use the COMMODORE KEY to slow it down. The DIRECTORY command cannot be used to print a hard copy. The disk directory must be loaded (destroying the program currently in memory) to do that.

EXAMPLES:

DIRECTORY	list all files on the disk(s) in unit 8.
DIRECTORY D1, U9, "work"	Lists the file on disk drive 1 of unit 9.
DIRECTORY "AB*"	Lists all files starting with the letters "AB", like ABOVE, ABOARD, etc. on all drives of unit 8.
DIRECTORY D0, "file ?.BAK"	The ? is a wild card that matches any single character

in that position: file 1.BAK, file 2.BAK, file 3.BAK all match the string.

NOTE: To print the DIRECTORY of drive 0, unit 8, use the following:

```
LOAD"$0",8
OPEN4,4:CMD4:LIST
PRINT#4:CLOSE4
```

16. DLOAD - load program file from disk

DLOAD "filename" [,Ddrive_number][<ON|,>Udevice]

This command loads a program from disk into current memory. (Use LOAD to load programs from tape.) A program name must be supplied.

EXAMPLES:

DLOAD "BANKRECS"	Searches the disk for the program "BANKRECS" and LOADs it.
DLOAD (A$)	LOADs a program from disk whose name is in the variable A$. An error is posted if A$ is empty.

The DLOAD command can be used within a BASIC program to find and RUN another program on disk. This is called chaining.

17. DSAVE - save program file to disk

DSAVE "filename" [,Ddrive_number] [<ON|,>Udevice]

This command stores a program on disk. (Use SAVE to store programs on tape.) A program name must be supplied.

EXAMPLES:

DSAVE "BANKRECS"	SAVEs the program "BANKRECS" to disk.
DSAVE (A$)	SAVEs the disk program whose name is in the variable A$.
DSAVE "PROG 3",D0,U9	SAVEs the program "PROG 3" to the disk with unit number (Serial bus address) of 9.

18. DVERIFY - verify program in memory with one on disk

DVERIFY filename [,Ddrive_number][<ON|,>Udevice]

This command causes the C-128 to check the program on the specified drive (the default is drive 0 of unit 8) against the program in memory.

NOTE: If a graphic area is allocated after a save (or deallocated after a save), verify and Dverify) will report an error. Technically this is correct. Because BASIC text moved from its original (saved) location, the link bytes changed. Hence, verify, which performs byte to byte comparisons, will fail even though the program is valid. To verify binary data see verify "name",8,1 form.

19. FAST - Put machine in 2 MHz mode of operation

The Fast command causes VIc 40 column screen to be blanked. All operations (except I/O) are sped up considerably. Graphics may be used, but will not be visible until a SLOW command is issued.

20. FETCH - get data from expansion memory

```
FETCH #bytes, intsa, expsa, expb
```

where #bytes = number of bytes to get from expansion memory (0-65535)
intsa = starting address of host ram (in decimal) (0-65535)
expa = starting address of expansion ram (in decimal)
expb = expansion ram bank number (xxxx 4 bits)
where xxxx = 0000 (first 64K bank)
= 0001 (second)
= 0010 .
= 0011 .
= 0100 .
= 0101 .
= 0110 .
= 1111 (16th)
xxxx is expressed in decimal 0-15

NOTE: if the number of bytes = 0 this means a length of 65536.

EXAMPLE :

```
FETCH 1000,52000,2000,7
```

21. HEADER - format diskette

```
HEADER "diskname" [,Ii.d.#] [,Ddrive_number]
                  [<ON|,>Udevice]
```

Before a new diskette can be used for the first time it must be formatted with the HEADER command. To erase an entire diskette for reuse use the HEADER command. This command divides the disk into sections called blocks, and it creates a table of contents, called a directory or catalog, on the disk. The diskname can be any name up to 16 characters long. The i.d. number is any 2 characters. Give each disk a unique i.d. number. Be careful when using the HEADER command because it erases all stored data. Giving no i.d. number performs a quick header. The old i.d. number is used. The quick header can only be used if the disk was previously formatted, since the quick header only clears out the directory rather than formatting the disk. Defaults to drive 0, unit 8. There is no prompt displayed or a display of files that were scratched when in program mode.

EXAMPLES:

```
HEADER "MYDISK", I23
HEADER "RECS", I45, D1, U8
```

22. HELP - show line where error occurred

```
HELP
```

The HELP command is used after an error has been reported

in a program. When HELP is typed, the line where the error occurred in a BASIC program is listed, with the portion containing the error displayed in reverse field for 40 column and underlined in 80 column mode.

23. KEY - define or list function keys

KEY [key #, string]

There are eight (8) function keys available to the user on the C-128 computer: four unshifted and four shifted. The C-128 allows a definition for what each key does when pressed. KEY with no parameter specified gives a listing displaying all the current KEY assignments. The data assigned to a key is typed out when that function key is pressed. The maximum length for all the definitions together is 246 characters. Entire commands (or a series of commands) can be assigned to a key.

EXAMPLES:

```
KEY 7, "GRAPHIC0" + chr$(13) + "LIST" + chr$(13)
```

This causes the computer to select text mode and list the program whenever the 'F7' key is depressed (in direct mode). The CHR$(13) is the ASCII character for RETURN. Use CHR$(141) for 'shifted RETURN' and CHR$(27) for 'ESCape'. Use CHR$(34) to incorporate a double quote into a KEY string. The keys may be redefined in a program. For Example:

```
10 KEY 2,"TESTING" + chr$(34):KEY3, "NO"

10 FOR I=1 to 8:KEY I, chr$(I+132):NEXT
```

defines the function keys as they are defined on the Commodore 64 and VIC 20.

To restore all function keys to their default values, reset the C-128 by turning it off and on, or press the RESET button.

24. LIST - list BASIC program

LIST [first line] [-[last line]]

The LIST command lets the user look at lines of a BASIC program that have been typed or LOADed into the C-128's memory. When used alone (without numbers following it), the C-128 gives a complete LISTing of the program on the screen, which may be slowed down by holding down the C= key, paused by CONTROL-S or SCROLL KEY (unpaused by pressing any other key), or stopped by hitting the RUN/STOP key. If the word LIST is followed by a line number, the C-128 shows only that line number. If LIST is typed with two numbers separated by a dash, the C-128 shows all lines from the first to the second line number. If LIST is typed followed by a number and just a dash, it shows all lines from that number to the end of the program. And if LIST is typed, a dash, and then a number, all lines from the beginning of the program to that line number are LISTed. By using these variations, any portion of a program can be examined or easily brought to the screen for modification. LIST can be used in a program and the program can continue with the use of the CONT statement.

EXAMPLES:

```
LIST         Shows entire program.
LIST 100-    Shows from line 100 until the end
             of the program.
LIST 10      Shows only line 10.
LIST-100     Shows lines from the beginning
             until line 100.

LIST 10-200  Shows lines from 10 to 200, inclusive.
```

25. LOAD - load program from device

LOAD "filename" [,device_number [,relocate flag]]

This is the command to use when a program stored on cassette tape or on disk is to be used. If LOAD is typed with no arguments followed by a RETURN key, the user is prompted as necessary before the screen goes blank. Press play, and the C-128 starts looking for a program on tape. When it finds one, the C-128 prints FOUND "filename". The Commodore key can be hit to LOAD or the spacebar to keep searching on the tape. Once the program is LOADed, it can be RUN, LISTed, or modified. Most often typed is the word LOAD followed by a program name in quotes ("program name"). The name may be followed by a comma (outside the quotes) and a number (or numeric value), which acts as a device number to determine where the program is stored (disk or tape). If no number is supplied, the C-128 assumes device number 1, which is the cassette tape recorder. The other device commonly used with the LOAD command is usually the disk drive, which is device number 8. (though the DLOAD command is usually used in this instance).

EXAMPLES:

```
LOAD             Reads in the next program from tape.
LOAD "HELLO"     Searches tape for a program called
                 HELLO, and LOADs if found.
LOAD A$          Looks for a program whose name is
                 in the variable called A$.
LOAD "HELLO",8   Looks for the program called HELLO
                 on the disk drive, or the drive
                 last accessed. (This is equivalent
                 to DLOAD "HELLO"
```

The LOAD command can be used within a BASIC program to find and RUN the next program on a tape or disk. This is called chaining. The RELOCATE FLAG determines where in memory a program is loaded. A relocate flag of 0 tells the C-128 to load the program at the start of the BASIC program area, and a flag of 1 tells it to LOAD from the point where it was SAVEd. The default value of the relocate flag is 0. This parameter is generally used when loading machine language programs.

26. MONITOR - enter C-128 machine language monitor

SEE SECTION 5.2 ON THE C-128 MONITOR.

27. NEW - clear program and data memory

NEW

This command erases the entire program in memory and clears out any variables that may have been used. Unless the program

was stored somewhere, it is lost until typed in again.
Be careful with the use of this command.
The NEW command also can be used as a statement in a BASIC program. When the C-128 gets to this line, the program is erased and everything stops. This is not especially useful under normal circumstances.

28. PLAY - Play musical string

```
PLAY "[Vn,On,Tn,Un,Xn,elements]"

      where   On = Octave (n=0-6)
              Tn = Tune envelope # (n=0-9)
                   (defaults)
                   0= piano
                   1= accordion
                   2= calliope
                   3= drum
                   4= flute
                   5= guitar
                   6= harpsichord
                   7= organ
                   8= trumpet
                   9= xylophone
              Un = Volume (n=0-9)
              Vn = Voice (n=1-3)
              Xn = filter on (n=1), off (n=0)

              Elements =
                     A,B,C,D,E,F,G   ... Notes
                     # ..............  Sharp (precedes note)
                     $ ..............  Flat (precedes note)
                     . ..............  Dotted (precedes
                     W ..............  Whole note follows
                     H ..............  Half note follows
                     Q ..............  Quarter note follows
                     I ..............  Eighth note follows
                     S ..............  Sixteenth note follows
                     R ..............  Rest
                     M..............  Wait for all voices
                                       currently playing to
                                       end. (a measure)
```

29. RENAME - rename files

```
RENAME "old name" TO "new name" [,Ddrive_number]
                                [<ON|,>Udevice]
```

Used to rename a file on a diskette.

EXAMPLE:

```
RENAME "TEST" TO "FINALTEST",D0    Changes the name of
                                   the file "TEST" to
                                   "FINALTEST"
```

30. RENUMBER - renumber lines of BASIC program

```
RENUMBER [new starting line #
         [,[increment][,old starting line #]]]
```

The new starting line is the number of the first line in the program after renumbering. It defaults to 10. The increment is the spacing between line numbers, i.e. 10, 20, 30

etc. It also defaults to 10. The old starting line number is the line number in the program where renumbering is to begin. This allows renumbering of a portion of the program. It defaults to the first line of the program. This command can only be executed from direct mode.

EXAMPLES:

RENUMBER 20, 20, 1	Starting at line 1, renumbers the program. Line 1 becomes line 20, and other lines are numbered in increments of 20.
RENUMBER, , 65	Starting at line 65, renumbers in increments of 10. Line 65 becomes line 10.

31. RUN - execute BASIC program

```
RUN [line #]
RUN ["filename" [,Ddrive_number][<ON|,>Udevice]
```

Once a program has been typed into memory or LOADed, the RUN command executes it. RUN clears all variables in the program before starting program execution. If there is no number following the command RUN, the computer starts with the lowest numbered program line. If there is a number following the RUN command, execution starts at that line number. If a filename is entered with the drive number and device number, the program is loaded into memory and executed. RUN may be used within a program.

EXAMPLES:

```
RUN       Starts program execution from lowest number.
RUN 100   Starts program execution at line 100.
RUN  "TEST"  Loads the program TEST from the default drive and starts
                program execution at the lowest line number.
```

32. SAVE - save program to device

```
SAVE ["filename" [,device_number[,EOT flag]]]
```

This command stores a program currently in memory onto a cassette tape or disk. If the word SAVE is typed alone followed by RETURN, the C-128 attempts to store the program on the cassette tape. It has no way of checking if there is already a program on the tape in that location, so be careful with the tapes. If SAVE is typed followed by a name in quotes or a string variable name, the C-128 gives the program that name, so it may be more easily located and retrieved in the future. If a device number is to be specified for the SAVE, follow the name by a comma (after the quotes) and a number or numeric variable. Device number 1 is the tape drive, and number 8 is the disk. After the number on a tape command, there can be a comma and a second number, which is either 1,2, or 3. If the second number is 1 a machine language program is saved (alt load addr), if the number is a 2 write an END-OF-TAPE marker (EOT flag) after the program (tape output only), and 3 does both . If trying to LOAD a program and the C-128 finds one of these markers rather than the program trying to be LOADed, a FILE NOT FOUND ERROR is posted.

EXAMPLES:

```
SAVE                Stores program to tape without a name.
SAVE "HELLO"        Stores on tape with the name HELLO.
SAVE A$             Stores on tape with the name in
                    variable A$.
SAVE "HELLO", 8     Stores on disk with name HELLO.
                    (equivalent to DSAVE "HELLO")

SAVE "HELLO", 1, 1  Stores on tape with name HELLO and
                    places an END-OF-TAPE marker after
                    the program.
```

33. SCRATCH - delete file from disk directory

```
SCRATCH "filespec" [,Ddrive_number] [<ON|,>Udevice]
```

Deletes a file from the disk directory. As a precaution, the system asks "Are you sure"(in direct mode only) before the C-128 completes the operation. Type a Y to perform the SCRATCH or type N to cancel the operation. Use this command to erase unwanted files, to create more space on the disk. Note: file name may contain template or wildcards.

EXAMPLE:

```
SCRATCH "MY BACK", D1     Erases the file MY BACK from
                          the disk in drive 1, unit 8.
```

34. SLEEP

```
SLEEP N
          where N in seconds 0<= N <= 65535
```

35. SLOW - Put machine in 1Mhz mode and unblanks VIC

36. SOUND - produce sound effects

```
SOUND v, f, d [,[dir] [,[m] [,[s] [,[w] [,p] ]]]]

where : v = voice (1..3)
        f = frequency value (0..65535)
        d = duration (0..32767 jiffys)
      dir = step direction (0(up) ,1(down) or 2(oscillate))
default=0
        m = minimum frequency (if sweep is used) (0..65535)
default=0
        s = step value for effects (0..65535) default=0
        w = waveform (0=triangle,1=saw,2=square,3=noise) default=2
        p = pulse width (0..4095) default=2048 (50% duty cycle)
```

37. SPRCOLOR - Set Multi-color1 and/or Multi-color2 colors for all sprites

```
SPRCOLOR [smcr-1][,smcr-2]

   where  [smcr-1] sets Multi-color1 for all sprites,
          [smcr-2] sets Multi-color2 for all sprites.

   Either of these parameters may be any color from 1 - 16
```

38. SPRDEF - Define sprite

```
SPRDEF  user input
```

user input	description
1-8	selects destination sprite (prompted)
A	Automatic cursor movement toggle
C	Copies sprites (prompted)
CRSR keys	Moves cursor
RETURN key	Moves cursor to start of next line
RETURN key	Exits sprite designer mode (prompted)
HOME key	Moves cursor to top left of grid
CLR key	Erases entire grid
1-4	selcts color source
<CTRL> 1-8	selects sprite foreground color (1-8)
Commodore key 1-8	selects sprite foreground color (9-16)
STOP key	cancels changes and returns to prompt
SHIFT RETURN	saves sprites and returns to prompt
X	Expands sprite in X toggle
Y	Expands sprite in Y toggle
M	Multicolor sprite toggle

39. SPRSAV - save sprite

SPRSAV <origin>,<destination>

SPRSAV 1,A$: REM transfers the dot patttern of sprite 1
to the string named A$.
SPRSAV B$,2: REM transfers the string B$ into sprite 2.

40. STASH - Move contents of host memory to expansion ram

STASH #bytes,intsa,expsa,expb

refer to FETCH command for description of parameters.

41. SWAP - swap contents of host ram with contents of expansion ram.

SWAP #bytes,intsa,expsa,expb

refer to FETCH command for description of parameters.

42. VERIFY - verify program in memory with one on device

VERIFY "filename" [,device_number [,relocate flag]]

This command causes the C-128 to check the program on tape or disk against the one in memory. This is proof that the program just SAVEd is really SAVEd, in case the tape is bad or something isn't working. This command is also very useful for positioning a tape so that C-128 writes after the last program on the tape. It will do so, and inform the user that the programs don't match. Now the tape is positioned properly, and the next program can be stored without fear of erasing the old one. VERIFY with no arguments after the command causes the C-128 to check the next program on tape, regardless of its name, against the program now in memory. VERIFY followed by a program name in quotes or a string variable searches the tape for that program and then checks it when found against the program in memory. VERIFY followed by a name and a comma and a number checks the program on the device with that number (1 for tape, 8 for disk). The relocate flag is the same as in the

LOAD command.

EXAMPLES:

```
VERIFY              Checks the next program on the tape.
VERIFY "HELLO"      Searches for HELLO on tape, checks
                    against memory.
VERIFY "HELLO",8,1  Searches for HELLO on disk,
                    then checks against memory.
```

NOTE: If a graphic area is allocated after a save (or deallocated after a save),(VERIFY and DVERIFY) will report an error. Technically this is correct. Because BASIC text moved from its original (saved) location, the link bytes changed. Hence, verify, which performs byte to byte comparisons, will fail even though the program is valid.

43. WIDTH

WIDTH n

This is a command to set the width of lines drawn using BASIC's graphic commands to either single (1) or double (2) width. This is used primarily in high res mode to make skinny lines more visible (in certain color combinations).

44. WINDOW

WINDOW allows the BASIC user to define a logical window within the physical text screen.

```
WINDOW top_left_col,top_left_row,bot_right_col,bot_right_row
       [,clear]
```

The coordinates must be legal in context of the current console screen. The clear flag, if provided, causes a screen-clear to be performed (only within the confines of the newly described window!). 'CLEAR', if provided, can be 0 (no cls), or 1 (do a cls).

5.1.1.3 ALPHABETICAL LIST OF STATEMENTS -

1. BANK
2. BOX
3. CHAR
4. CIRCLE
5. CLOSE
6. CLR
7. COLLISION
8. COLOR
9. DATA
10. DCLEAR
11. DCLOSE
12. DEF FN
13. DIM
14. DO/LOOP/WHILE/UNTIL/EXIT
15. DOPEN
16. DRAW
17. END
18. ENVELOPE
19. FILTER
20. FOR/TO/STEP/NEXT
21. GET
22. GETKEY
23. GET#
24. GO64
25. GOSUB
26. GOTO/GO TO
27. GRAPHIC
28. IF/THEN/ELSE
29. INPUT
30. INPUT#
31. LET
32. LOCATE
33. MOVSPR
34. ON
35. OPEN
36. PAINT
37. POKE
38. PRINT
39. PRINT#
40. PRINT USING
41. PUDEF
42. READ
43. RECORD
44. REM
45. RESTORE
46. RESUME
47. RETURN
48. RREG
49. SCALE
50. SCNCLR
51. SPRITE
52. SSHAPE/GSHAPE
53. STOP
54. SYS
55. TEMPO
56. TRAP
57. TRON
58. TROFF
59. VOL
60. WAIT

5.1.1.4 STATEMENT DESCRIPTION -

1. BANK - alter 64K bank for PEEK, POKE, WAIT and SYS

```
BANK  banknumber

      where banknumber = 0 - 15
      (bank number configurations listed in MONITOR section)
```

Specify a 64K bank to be used for subsequent PEEK, POKE, WAIT and SYS statements.

2. BOX - draw box at specified position on screen

```
BOX  [color source #], a1, b1[,[ a2, b2]
      [,[angle] [,paint]]]

    color source .... Color source (0-3);
                      default is 1 (foreground)
    a1, b1 .......... Corner coordinate (scaled)
    a2, b2 .......... Corner opposite a1,
                      b1 (scaled); default
                      is the PC.
    angle ........... Rotation in clockwise
                      degrees; default is
                      0 degrees
    paint ........... Paint shape with color
                      (0:off, 1:on)
                      default is 0
```

NOTE: wrapping occurs if the degree is greater than 360.

This command allows the user to draw a rectangle of any size anywhere on the screen. To get the default value, include a comma without entering a value. Rotation is based on the center of the rectangle. The Pixel Cursor (PC) is left at a2, b2 after the BOX statement is executed.

EXAMPLES:

```
BOX 1, 10, 10, 60, 60     Draws the outline of a
                          rectangle
BOX , 10, 10, 60, 60, 45, 1   Draws a filled, rotated
                              box (a diamond)
BOX , 30, 90, , 45, 1     Draws a filled, rotated
                          polygon
```

3. CHAR - display characters at specified position on screen

```
CHAR [color source #],x,y [,[string][,rvs]]

      color source ..... Color source (0-3)
      x ................ Character column (0-79)
      y ................ Character row (0-24)
      string ........... String to print
      rvs  ............. reverse field flag
                         (0=off, 1=on)
```

Text (alphanumeric strings) can be displayed on any screen at a given location by the CHAR statement. Character data is read from the C-128 character ROM area. The user supplies the x and y coordinates of the starting position and the text string to be displayed, color source and reverse imaging are

optional.

The string is continued on the next line if it attempts to print past the right edge of the screen (in 40 column mode. When used in TEXT mode, the string printed by the CHAR command works just like a PRINT string, including reverse field, cursors, none, etc. These control functions inside the string do not work when the CHAR command is used to display text in GRAPHIC mode.

4. CIRCLE - draw circles, ellipses, arcs, etc. at specified position on screen

```
CIRCLE [color source] ,[a,b], xr [, [yr] [, [sa] [, [ea]
       [, [angle] [,inc] ]]]]

       color source ......... Color source (0-3)
       a,b ........ Center coordinate (scaled)
                    (defaults to the Pixel
                    Cursor [PC])
       xr ......... X radius (scaled)
       yr ......... Y radius (default is xr)
       sa ......... Starting arc angle (default 0)
       ea ......... Ending arc angle (default 360)
       angle ...... Rotation in clockwise degrees
                    (default is 0 degrees)
       inc ........ Degrees between segments
                    (default is 2 degrees)
```

With the CIRCLE command the user can draw a circle, ellipse, arc, triangle, octagon or other polygon. The final Pixel Cursor (PC) is on the circumference of the circle at the ending arc angle. Any rotation is about the center. Setting the Y radius equal to the X radius does not draw a circle, since the X and Y coordinates are scaled differently (see scale). Arcs are drawn from the starting angle clockwise to the ending angle. The segment increment controls the coarseness of the shape, with lower values for inc creating rounder shapes.

EXAMPLES:

```
CIRCLE , 160,100,65,10      Draws an ellipse.
CIRCLE  ,  160,100,65,50    Draws a circle.
CIRCLE , 60,40,20,18,,,,45  Draws an octagon.
CIRCLE , 260,40,20,,,,,90   Draws a diamond.
CIRCLE , 60,140,20,18,,,,120   Draws a triangle.
```

5. CLOSE - Close logical file

```
CLOSE file #
```

This command completes and closes any files used by the DOPEN or OPEN statements. The number following the word CLOSE is the file number to be closed.

EXAMPLE:

```
CLOSE 2       logical file 2 is closed.
```

6. CLR - Clear program variables

```
CLR
```

This statement erases any variables in memory, but leaves the program itself intact. This statement is automatically executed when a RUN or NEW command is given. It is not necessary to 'CLR' after editing because variables and text no longer share memory.

7. COLLISION - Define handling for sprite collision interrupt

COLLISION type [,statement]

type ..... Type of interrupt:
0 = Sprite to sprite collision
1 = Sprite to display data collision
2 = Light pen - 40 col VIC only
statement ... BASIC line number of a subroutine

When the specified situation occurs, BASIC will finish processing the currently executing instruction and perform a GOSUB to the line number given. When the subroutine terminates (it must end with a RETURN) BASIC will resume processing where it left off. Interrupt action continues until a COLLISION of the same type but without any address is specified. More than one type interrupt may be enabled at the same time, but only one interrupt can be handled at a time (i.e., no recursion and no nesting of interrupts). Note that what caused an interrupt may continue causing interrupts for some time unless the situation is altered or the interrupt disabled. When a sprite is totally "off screen" and not visible it cannot generate an interrupt. To determine which sprites have collided since the last check use the BUMP function: BUMP(1) records which sprites have collided with each other and BUMP(2) records which sprites have collided with display data. COLLISION need not be active to use BUMP. The bit positions (7-0) in the BUMP value correspond to a sprite's number. In the case of multiple compares consider a sprite's position (via RSPPOS) to determine which sprite hit what. BUMP(n) is reset to zero after each call.

8. COLOR - define colors for each color source

COLOR area#, color #

Assigns a color to one of the seven color sources:

Area	Source
0	VIC background
1	Graphic foreground
2	Graphic multicolor 1
3	Graphic multicolor 2
4	VIC border
5	Text color - whatever current text screen is (40 or 80)
6	8563 background color

Colors that are useable are in the range 1 - 16 (BLACK, WHITE ...). VIC colors differ slightly from 8563 colors.

9. DATA - define data to be used by program in the program

DATA list of constants separated by commas

This statement is followed by a list of items to be used by READ statements. The items may be numbers or strings, and are separated by commas. Words need not be inside quote marks, unless they contain any of the following characters: SPACE, colon, or comma. If two commas have nothing between them, the value will be READ as a zero for a number, or an empty string. Also see the RESTORE statement, which allows the C-128 to reread data.

EXAMPLE:

```
DATA 100, 200, FRED, "HELLO MOM", , 3, 14, ABC123
```

10. DCLEAR - clear all open channels on disk drive

```
DCLEAR  Ddrive_number [<ON|,>Udevice]
```

Clears and closes all open channels on drive drive_number of the optionally specified unit device_number. Default is U8, drive 0.

11. DCLOSE - close disk file

```
DCLOSE  [#lf][<ON|,>Udevice]
```

Closes a single file or all the files currently open on a disk unit. Here "lf" represents the logical file number of the file to be closed on the disk and device_number is the number of the disk drive.(default = 8).

EXAMPLES:

```
1. DCLOSE         (closes all files currently
                   open on unit # 8.)
2. DCLOSE #lf     (closes the file associated
                   with the logical file number "lf"
                    on unit # 8).

3. DCLOSE ON Uz   (closes all files currently
                   open on unit "z")
```

12. DEF FN - define function

```
DEF FN  (DEFine FunctioN)
DEF FN  name (variable) = expression
```

This statement allows definiton of a complex calculation as a function. In the case of a long formula that is used several times within a program, this can save a lot of space. The name given to the function begins with the letters FN, followed by any numeric (non-integer). First define the function by using the statement DEF followed by the name given to the function. Following the name is a set of parentheses () with a dummy numeric variable name (in this case, X) enclosed. Then there is an equal sign, followed by the formula to be defined. The formula can be "called", substituting any number for X, using the format shown in line 20 of the example below:

EXAMPLE:

```
10 DEF FNA(X)=12*(34.75-X/.3)+X
20 PRINT FNA(7)
```

The number 7 is inserted each place X is located in the formula given in the DEF statement.

13. DIM - declare array dimensions

DIM variable (subscripts) [,variable(subscripts)]...

Before arrays of variables can be used, the program must first execute a DIM statement to establish DIMensions of that array (unless there are 11 or fewer elements in the array). The statement DIM is followed by the name of the array, which may be any legal variable name. Then, enclosed in parentheses, put the number (or numeric variable) of elements in each dimension. An array with more than one dimension is called a matrix. Any number of dimensions may be used, but keep in mind that the whole list of variables being created takes up space in memory, and it is easy to run out of memory if too many are used. To figure the number of variables created with each DIM, multiply the total number of elements in each dimension of the array. (Each array starts with elememnt 0.)

NOTE: Integer arrays take up 2/5ths of the space of floating point arrays.

More than one array can be dimensioned in a DIM statement by separating the arrays by commas. If the program executes a DIM statement for any array more than once, the message "re'DIMed array error" is posted. It is good programming practice to place DIM statements near the beginning of the program.

EXAMPLE:

```
 10 DIM A$(40),B7(15),CC%(4,4,4)
           |      |         |
 41 elements   16 elements  125 elements
```

14. DO/LOOP/WHILE/UNTIL/EXIT - program loop definition and control

DO[UNTIL boolean argument / WHILE boolean argument]
statements [EXIT]

LOOP[UNTIL boolean argument / WHILE boolean argument]

Performs the statements between the DO statement and the LOOP statement. If no UNTIL or WHILE modifies either the DO or the LOOP statement, execution of the intervening statements continues indefinitely. If an EXIT statement is encountered in the body of a DO loop, execution is transferred to the first statement following the LOOP statement. Do loops may be nested, following the rules defined for FOR-NEXT loops. If the UNTIL parameter is used, the program continues looping until the boolean argument is satisfied (becomes true). The WHILE parameter is basically the opposite of the UNTIL parameter: the program continues looping as long as the boolean argument is TRUE. An example of a boolean argument is A=1, or G>65.

EXAMPLE:

```
DO UNTIL X=0 or X=1
   :
     LOOP

DO WHILE A$="C": GETKEY A$: LOOP
```

15. DOPEN - open disk file

```
DOPEN #lf, "filename[,<S|P>]" [,Ly][,Ddrivenumber]
                        [<ON|,>Udevice][,w]

                        where S = sequential file
                              P = program file
```

This command opens a data file for a read and/or write access. Here "lf" represents the logical file number of the file to be opened. "y" is the record length for a relative file. "drive_number" is the disk drive number in which the disk containing the file is located, and "device_number" is the drive unit number if more than one drive unit is present. A sequential file is opened for write access if "W" is present: it is opened for read access if "W" is not present.

16. DRAW - draw lines and shapes at specified position on screen

```
DRAW  color source [, a1, b1,][ TO a2, b2] ...
```

With this statement individual dots, lines, and shapes can be drawn. The user supplies the color source (0-3), starting (a1, b1) and ending points (a2, b2).

EXAMPLES:

```
a dot:   DRAW 1, 100, 50 -- no endpoint specified,
                            defaults to a1, b1 value
                            for a2, b2 to create a dot

lines:   DRAW , 10,10, TO 100,60
         DRAW  TO 25,30

a shape: DRAW , 10,10 TO 10,60 TO 100,60 TO 10,10
```

17. END - define end of program execution

```
END
```

When the program executes the END statement, the program stops RUNning immediately. The CONT command can be used to restart the program at the next statement following the END statement (if any).

18. ENVELOPE - define musical instruments envelopes

```
ENVELOPE n, [,[atk] [,[dec] [,[sus] [,[rel]
         [,[wf] [,pw] ]]]]]

   n.............. Envelope number (0-9)
   atk ........... Attack rate (0-15)
   dec ........... Decay rate (0-15)
   sus ........... Sustain rate (0-15)
   rel ........... Release rate (0-15)
   wf ............ Waveform: 0 = triangle
                            1 = sawtooth
                            2 = pulse (square)
                            3 = noise
                            4 = ring modulation
   pw ............ Pulse width (0 - 4095)
```

A parameter that is not specified will retain its current value. Pulse width applies to pulse waves (wf=2) only and is determined by the formula (pwout = pw/40.95 %), so that pw =

2048 produces a square wave and values of 0 and 4095 produce constant DC output. The C-128 initializes the ten (10) tune envelopes to:

	n	A	D	S	R	wf	pw	instrument
ENVELOPE	0,	0,	9,	0,	0,	2,	1536	piano
ENVELOPE	1,	12,	0,	12,	0,	1		accordion
ENVELOPE	2,	0,	0,	15,	0,	0		calliope
ENVELOPE	3,	0,	5,	5,	0,	3		drum
ENVELOPE	4,	9,	4,	4,	0,	0		flute
ENVELOPE	5,	0,	9,	2,	1,	1		guitar
ENVELOPE	6,	0,	9,	0,	0,	2,	512	harpsichord
ENVELOPE	7,	0,	9,	9,	0,	2,	2048	organ
ENVELOPE	8,	8,	9,	4,	1,	2,	512	trumpet
ENVELOPE	9,	0,	9,	0,	0,	0		xylophone

19. FILTER - define sound filter parameters

FILTER [freq] [,[lp] [,[bp] [,[hp] [,res]]]]

freq ..... Filter cut-off frequency (0-2047)
lp ..... Low pass filter on (1), off (0)
bp ..... Band pass filter on (1), off(0)
hp ..... High pass filter on (1), off(0)
res ..... Resonance (0-15)

Unspecified parameters result in no change to the current value. The filter frequency is determined by the following formula:

The filter output modes are additive. For example, both low pass and high pass filters can be selected to produce a notch (or band reject) filter response. For the filter to have an audible effect at least one filter output mode must be selected and at least one voice must be routed through the filter.

20. FOR/TO/STEP/NEXT - program loop definition and control

FOR variable = start value TO end value [STEP increment]

This statement works with the NEXT statement to set up a section of the program that repeats for a set number of times. The user may just want the C-128 to count up to a large number so the program pauses for a few seconds, in case something needs to be counted, or something must be done a certain number of times (such as printing). The loop variable is the variable that is added to or subtracted from during the FOR/NEXT loop. The start value and the end value are the beginning and ending counts for the loop variable. The logic of the FOR statement is as follows. First, the loop variable is set to the start value. When the program reaches a line with the statement NEXT, it adds the STEP increment (default = 1) to the value of the loop variable and checks to see if it is higher than the end of the loop value. If it is not higher, the command line executed is that immediately following the FOR statement. If the loop variable is larger than the end of the loop number, then the next statement executed is the one following the NEXT statement. The opposite is true if the step size is negative. See also the NEXT statement. The maximum number of nested loops is determined by the amount of available stack space, which is 199 bytes. If the stack is overrun, then the message "formula too complex" is posted.

EXAMPLE:

```
10 FOR L = 1 TO 10
20 PRINT L
30 NEXT L
40 PRINT "I'M DONE! L = "L
```

This program prints the numbers from one to ten on the screen, followed by the message I'M DONE! L = 11.

The end of the loop value may be followed by the word STEP and another number or variable. In this case, the value following the STEP is added each time instead of one. This allows counting backwards, by fractions, or any way necessary.

The user can set up loops inside one another. This is known as nested loops. Care must be taken when nesting loops so that the last loop to start is the first one to end.

EXAMPLE OF NESTED LOOPS:

```
10 FOR L = 1 TO 100
20 FOR A = 5 TO 11 STEP .5
30 NEXT A
40 NEXT L
```

A FOR ... NEXT loop (line 20) is "nested" inside the larger one (line 10).

21. GET - get input data from keyboard (no wait)

GET variable list

The GET statement is a way to get data from the keyboard one character at a time. When the GET is executed, the character that was typed is received. If no character was typed, then a null (empty) character is returned, and the program continues without waiting for a key. There is no need to hit the RETURN key, and in fact the RETURN key can be received with a GET. The word GET is followed by a variable name, usually a string variable. If a numeric were used and any key other than a number was hit, the program would stop with an error message. The GET statement may also be put into a loop, checking for an empty result, that waits for a key to be struck to continue. The GETKEY statement could also be used in this case. This statement can only be executed within a program.

EXAMPLE:

```
10 DO:GET A$: LOOP UNTILE A$ ="A"

This line waits for the A key to be pressed
to continue.
```

22. GETKEY - get input character from keyboard (wait for key)

GETKEY variable list

The GETKEY statement is very similar to the GET statement. Unlike the GET statement, GETKEY waits for the user to type a character on the keyboard. This lets it be used easily to wait for a single character to be typed. This statement can only be executed within a program.

EXAMPLE:

```
10 GETKEY A$
```

This line waits for a key to be struck.
Typing any key will continue the program.

23. GET# - get input data from file

GET# file number,variable list

Used with a previously OPENed device or file to input one character at a time. Otherwise, it works like the GET statement. This statement can only executed within a program.

EXAMPLE:

```
10 GET#1,A$
```

24. GO64 - Switch to C64 mode

GO64

This statement switches from C128 mode to C64 mode. The question "Are You Sure?" Y/N (in direct mode only) is posted for the user to respond to. If Y is typed then the currently loaded program is lost and control is given to C64 mode. This statement can be used in direct mode or within a program.

25. GOSUB - execute subroutine

GOSUB line #

This statement is like the GOTO statement, except that the C-128 remembers from where it came. When a line with a RETURN statement is encountered, the program jumps back to the statement immediately following the GOSUB. The target of a GOSUB statement is called a subroutine. A subroutine is useful if there is a routine in the program that can be used by several different portions of the program. Instead of duplicating the section of program over and over, it can be set up as a subroutine, and GOSUB to it from the different parts of the program. See also the RETURN statement.

EXAMPLE:

```
20 GOSUB 800     means go to the subroutine
                 beginning at line 800 and execute it
  :
  :
800 PRINT "HI THERE": RETURN
```

26. GOTO/GO TO - transfer program execution to specified line number

GOTO line number

After a GOTO statement is executed, the next line to be executed will be the one with the line number following the word GOTO. When used in direct mode, GOTO line number allows starting of execution of the program at the given line number without clearing the variables.

EXAMPLE:

```
10 PRINT"COMMODORE"
20 GOTO 10
```

The GOTO in line 20 makes line 10 repeat continuously until RUN/STOP is pressed.

27. GRAPHIC - select graphic mode

```
GRAPHIC mode[,clear[,s]]
GRAPHIC CLR
```

This statement puts the C-128 in one of 6 graphics modes:

mode	description
0	40 column normal text
1	high-resolution graphics
2	high-resolution graphics, split screen
3	multicolor graphics
4	multicolor graphics, split screen
5	80 column text

where ,s = number of first line of text in split modes (0..25) default is 19.

When executed, GRAPHIC 1 - 4 allocates a 9K bit mapped area, and the start of BASIC text area is moved above the hi res area. This area remains allocated even if the user returns to TEXT mode (GRAPHIC 0). If 1 is given in the GRAPHIC statement as the second argument, the screen is also cleared. Executing a GRAPHIC CLR command then deallocates to 9K bit mapped area, and makes it available once again for BASIC text and variables.

28. IF/THEN/ELSE - conditional program execution

```
IF expression THEN then-clause [:ELSE else-clause]
```

IF...THEN lets the computer analyze a BASIC expression preceded by IF and take one of two possible courses of action. If the expression is true, the statement following THEN is executed. This expression can be any BASIC statement. If the expression is false, the program goes directly to the next line, unless an ELSE clause is present. The ELSE clause, if present, must be in the same line as the IF-THEN part. When an ELSE clause is present, it is executed when the THEN clause isn't executed. In other words, the ELSE clause executes when the expression is FALSE.

The expression being evaluated may be a variable or formula, in which case it is considered true if nonzero, and false if zero. In most cases, there is an expression involving relational operators (=, <, >, <=, >=, <>).

EXAMPLE:

```
50 IF X>0 THEN PRINT "OK": ELSE END
```

Checks the value of X. If X is greater than 0, the THEN clause is executed, and the ELSE clause isn't. If X is less than or equal to 0, the ELSE clause is executed and the THEN clause isn't.

NOTE: The colon is required after THEN for <escape>

commands.

29. INPUT - prompt on screen for input from keyboard

INPUT [prompt string;] variable list

The INPUT statement allows the computer to ask for data from the user running the program and places it into a variable or variables. The program stops, prints a question mark (?) on the screen, and waits for the user to type the answer and hit the RETURN key. The word INPUT is followed by a variable name or list of variable names separated by commas. There may be a message inside of quotes before the list of variables to be input. If this message (called a prompt) is present, there must be a semicolon (;) after the closing quote of the prompt. When more than one variable is to be INPUT, they should be separated by commas when typed in. If not, the computer asks for the remaining values by printing two question marks (??). If the RETURN key is pressed without INPUTting a value, the INPUT variable retains the value previously input for that variable. This statement can only be executed within the program.

EXAMPLE:

```
10 INPUT "PLEASE TYPE A NUMBER";A
20 INPUT "AND YOUR NAME";A$
30 INPUT B$
40 PRINT "BET YOU DIDN'T KNOW WHAT I WANTED!"
```

30. INPUT# - input data from file

INPUT# file number, variable list

This works like INPUT, but takes the data from a previously OPENed file or device. No prompt string is allowed. This statement can only be used in a program.

EXAMPLE:

```
10 INPUT#2, A$, C, D$
```

31. LET - infrequently used keyword used with assignment statements

[LET] variable = expression

The word LET is hardly ever used in programs, since it is not necessary. Whenever a variable is defined or given a value, LET is always implied. The variable name that is to get the result of a calculation is on the left side of the equal sign, and the number,string, or formula is on the right side. Multiple assignments on LET statements are not allowed.

EXAMPLE:

```
10 LET A = 5
20 B = 6
30 C = A * B + 3
40 D$ = "HELLO"
```

32. LOCATE - position graphics pixel cursor on screen

LOCATE x-coordinate, y-coordinate

The LOCATE statement allows the Pixel Cursor (PC) to be put anywhere on the screen. The PC is the current default location of the starting point of the next drawing. Unlike the regular cursor, the PC cannot be seen, but it can be moved with the LOCATE statement.

EXAMPLE:

```
LOCATE 160,100
```

Positions the PC in the center of the high resolution screen. Nothing will be seen until something is drawn. The PC can be found at any time by using the RDOT(0) function to get the X-coordinate and RDOT(1) to get the Y-coordinate. The color source of the dot at the PC can be found by PRINTing RDOT(2).

33. MOVSPR - move sprite

```
MOVSPR <number> <,x1,y1>

                    +/- x1, +/- y1 = relative position
                    x1#y1  = angle and speed

          <number> is sprite's number (1 through 8)
        <,x1,y1> is coordinate of new sprite location
```

This statement is used to either initiate sprite motion at a specified rate, or to locate a sprite at a specific location on the screen.

34. ON - conditional branching

```
ON expression <GOTO/GOSUB> line #1 [, line #2...]
```

This statement can make the GOTO and GOSUB statements into special versions of the IF statement. The word ON is followed by a formula, then either GOTO or GOSUB, and a list of line numbers separated by commas. If the result of the calculation of the formula (expression) is 1, the first line in the list is executed. If the result is 2, the second line number is executed. If the result is 0, or larger than the number of line numbers in the list, the next line executed is the statement following the ON statements. If the number is negative, an ILLEGAL QUANTITY ERROR results.

EXAMPLE:

```
10 INPUT X:IF X<0 THEN 10
20 ON X GOTO 50, 70, 30        When X=1,ON sends control
                               to the first line number
                               in the list (50).
                               When X=2, ON sends control
                               to the second line (70),etc
25 Print "FELL THROUGH":GOTO 10
30 PRINT "TOO HIGH":GOTO 10
50 PRINT"TOO LOW":GOTO 10
70 PRINT "THAT'S IT"
```

35. OPEN - open files for input or output

```
OPEN file#, device_number [,secondary address]
      <[,"filename, type, mode"]/[,cmd string]>
```

The OPEN statement allows the C-128 to access devices such as the cassette recorder and disk for data, a printer, or even the screen of the C-128. The word OPEN is followed by a logical file number, which is the number to which all other BASIC statements will refer. This number is from 0 to 255. There is always a second number after the first called the device number. Device number 0 is the C-128 keyboard, 1 is the cassette recorder, 3 is the C-128 screen, 4-7 is the printer(s), 8-11 is usually the disk(s). It is often a good idea to use the same file number as the device number because it makes it easy to remember which is which. Following the second number may be a third number called the secondary address. In the case of the cassette, this can be 0 for read, 1 for write, and 2 for write with END-OF-TAPE marker at the end. In the case of the disk, the number refers to the channel number. In the printer, the secondary addresses are used to set the mode of the printer. There may also be a string following the third number, which could be a command to the disk or the name of the file on tape or disk. The type and mode refer to disk files only. (File types are prg, seq, rel, and usr; modes are read and write.)

EXAMPLES:

```
10 OPEN 3,3           OPENs the screen as a device
10 OPEN 1,0           OPENs the keyboard as a device.
20 OPEN 1,1,0,"DOT"   OPENs the cassette for reading,
                      file to be searched for is "DOT"
OPEN 4,4              OPENs a channel to use the printer
OPEN 15,8,15          OPENs the command channel on the
                      disk.

5 OPEN 8,8,12,"TESTFILE,SEQ,WRITE"   creates a seq.
                                     disk file for
                                     writing, called
                                     "testfile".
```

See also: CLOSE, CMD, GET#, INPUT#, and PRINT# statements, system variables ST, DS, and DS$.

36. PAINT - fill area with color

PAINT [color source] [,[a,b] [,mode]]

```
color source ..... (0-3); default is 1,
                    foreground color
a,b ............... starting coordinate,
                    scaled (default at PC)
mode .............. 0 = paint an area defined
                        by the color
                        source selected
                    1 = paint an area defined
                        by any non-background
                        source
```

The PAINT command fills an area with color. It fills in the area around the specified point until a boundary of the same color (or any non-background color, depending on which mode chosen) is encountered. The final position of the PC will be at the starting point (a,b).

NOTE: If the starting point is already the color of color source (or any non-background when mode 1 is used), there is no change.

EXAMPLE:

```
10 CIRCLE , 160,100,65,50  draws outline of circle
20 PAINT  ,  160,100       fills in the circle with
                           color
                           (using default color of
                            foreground)
```

37. POKE - change data in RAM

POKE address, value

The POKE statement allows changing of any value in the C-128 RAM, and allows modification of many of the C-128 Input/Output registers. POKE is always followed by two numbers, (or expressions). The first number is a location inside the C-128 memory. This could have a value from 0 to 65535. The second number is a value from 0 to 255, which is placed in the location, replacing any value that was there previously. The POKE occurs into the currently selected BANK. This command is bank sensitive, i.e. the user must "BANK 15" before I/O will be in map.

EXAMPLE:

```
10 POKE 28000,8       Sets location 28000 to 8
20 POKE 28*1000,27    Sets location 28000 to 27
```

NOTE: PEEK, a related function to POKE, is listed under FUNCTIONS (SECTION 5.1.1.3)

38. PRINT - output to text screen

The PRINT statement is the major output statement in BASIC. While the PRINT statement is the first BASIC statement most people learn to use, there are many variations to be mastered here as well. The word PRINT can be followed by any of the following:

```
Characters inside of quotes     ("text lines")
Variable names                  (A, B, A$, X$)
Functions                       (SIN(23), ABS(33))
Punctuation marks               (; ,)
```

The characters inside of quotes are often called literals because they are printed exactly as they appear. Variable names have the value they contain (either a number or a string) printed. Functions also have their number values printed. Punctuation marks are used to help format the data neatly on the screen. The comma divides the screen into 4 columns for data, while the semicolon doesn't add spaces. Either mark can be used as the last symbol in the statement. This results in the next PRINT statement acting as if it is continuing the last PRINT statement.

EXAMPLE:

```
                                    RESULT
                                    ------
10 PRINT "HELLO"                    HELLO
20 A$="THERE":PRINT "HELLO,A$       HELLO THERE
30 A=4:B=2:PRINT A+B                6
50 J=41:PRINT J;:PRINT J-1          41 40
```

```
60 C=A+B:D=A-B:PRINT A;B;C,D    4  2  6         2
```

See also: POS(), SPC(), TAB(), and CHAR FUNCTIONS

39. PRINT# - output to files

PRINT# file#, print list

There are a few differences between this statement and the PRINT. First of all, the word PRINT# is followed by a number, which refers to the device or data file previously OPENed. The number is followed by a comma, and a list of things to be PRINTed. The comma and semi- colon act in the same manner for spacing as they do in the PRINT statement. Some devices may not work with TAB and SPC.

EXAMPLE:

```
100 PRINT#1,"HELLO THERE!",A$,B$
```

40. PRINT USING - output using format

PRINT [#filenumber] USING format list; print list;

These statements define the format of string and numeric items for printing to the text screen, printer, or other device. The format is put in quotes. This is the list format. Then add a semicolon and a list of what is to be printed in the format for the print list. The list can be variables or the actual values to be printed.

EXAMPLE:

```
 5 X=32: Y=100.23: A$="CAT"
10 PRINT USING "$##.##";13.25,X,Y
20 PRINT USING "###>#";"CBM",A$
```

When this is RUN, line 10 prints out:

```
$13.25      $32.00   $*****    prints ***** instead
                               of Y value because
                               Y has 5 digits, which
                               does not conform
                               to format list
                               (as explained below)
```

Line 20 prints this:

```
CBM   CAT             leaves three spaces before printing
                     "CBM" as defined in format list
```

CHARACTER	NUMERIC	STRING
Pound sign (#)	X	X
Plus sign (+)	X	
Minus sign (-)	X	
Decimal Point (.)	X	
Comma (,)	X	
Dollar Sign ($)	X	
Four Carets (^^^^)	X	
Equal Sign (=)		X
Greater Than Sign (>)		X

The pound sign (#) reserves room for a single character in the output field. If the data item contains more characters than # in the format field, the entire field is filled with asterisks (*). No numbers are printed.

EXAMPLE:

```
10 PRINT USING "####";X
```

For these values of X, this format displays:

```
A = 12.34      12
A = 567.89     568
A = 123456     ****
```

For a STRING item, the string data is truncated at the bounds of the field. Only as many characters are printed as there are pound signs (#) in the format item. Truncation occurs on the right.

The plus (+) and minus (-) signs can be used in either the first or last position of a format field but not both. The plus sign is printed if the number is positive. The minus sign is printed if the number is negative.

If a minus sign is used and the number is positive, a blank is printed in the character position indicated by the minus sign.

If neither a plus sign or a minus sign is used in the format field for a numeric data item, a minus sign is printed before the first digit or dollar symbol if the number is negative and no sign is printed if the number is positive. This means that one more character is printed if the number is positive. If there are too many digits to fit into the field specified by the pound sign and +/- signs, then an overflow occurs and the field is filled with asterisks (*).

A decimal point (.) symbol designates the position of the decimal point in the number. There can be only one decimal point in any format field. If a decimal point is not specified in the format field, the value is rounded to the nearest integer and printed without any decimal places.

When a decimal point is specified, the number of digits preceding the decimal point (including the minus sign, if the value is negative) must not exceed the number of pound signs before the decimal point. If there are too many digits an overflow occurs and the field is filled with asterisks(*).

A comma (,) allows placing of commas in numeric fields. The position of the comma in the format list indicates where the commas appears in a printed number. Only commas within a number are printed. Unused commas to the left of the first digit appear as the filler character. At least one pound sign must precede the first comma in a field.

If commas are specified in a field and the number is negative, then a minus sign is printed as the first character even if the character position is specified as a comma.

EXAMPLES:

FIELD	EXPRESSION	RESULT	COMMENT
##.#	-.1	-0.1	Leading zero added
##.#	1	1.0	Trailing zero added
####	-100.5	-101	Rounded to no decimal places.
####	-1000	****	Overflow because 4 digits and minus sign

			cannot fit in field
###.	10	10.	Decimal point added
#$##	1	$1	Leading dollar sign

A dollar sign ($) symbol shows that a dollar sign will be printed in the number. If the dollar sign is to float (always be placed before the number), specify at least one pound sign before the dollar sign. If a dollar sign is specified without a leading pound sign, the dollar sign is printed in the position shown in the format field. If commas and/or a plus or minus sign is specified in a format field with a dollar sign, the program prints a comma or sign before the dollar sign. The four up arrows or carets symbol is used to specify that the the number is to be printed in E + format(scientific notation). A pound sign must be used in addition to the four up arrows to specify the field width. The arrows can appear either before or after the pound sign in the format field. Four carats must be specified when a number is to be printed in E format. If more than one but fewer than four carats are specified, a syntax error results. If more than four carats are specified only the first four are used. The fifth carat is interpreted as a no text symbol. An equal sign(=) is used to center a string in a field. The field width is specified by the number of characters (pound sign and =) in the format field. If the string contains fewer characters than the field width, the string is centered in the field. If the string contains more characters that can be fit into the field, then the rightmost characters are truncated and the string fills the entire field. A greater than sign (>) is used to right justify a string in a field.

41. PUDEF - redefine symbols in PRINT USING statements

PUDEF "nnnn"

PUDEF allows redefinition of up to 4 symbols in the PRINT USING statement. Blanks, commas, decimal points, and dollar signs can be changed into some other character by placing the new character in the correct position in the PUDEF control string.

Position 1 is the filler character. The default is a blank. Place a new character here for another character to appear in place of blanks.
Position 2 is the comma character. Default is a comma.
Position 3 is the decimal point.
Position 4 is the dollar sign.

EXAMPLES:

```
10 PUDEF "*"        PRINTs * in the place of blanks.
20 PUDEF " @"       PRINTs @ in place of commas.
```

42. READ - read data from DATA statements

READ variable list

This statement is used to get information from DATA statements into variables, where the data can be used. The READ statement variable list may contain both strings and numbers. Care must be taken to avoid reading strings where the READ statement expects a number, which produces an ERROR message.

EXAMPLE:

READ A, G$,

43. RECORD - adjust relative file pointers

RECORD# lf, rno [,bno]

Adjusts a relative file pointer to select any byte (character) of any record in the relative file. (SEE DOPEN , OPEN)

44. REM - add explanatory text to program listing

REM message

The REMark is just a note to whomever is reading a LIST of the program. It may explain a section of the program, give information about the author, etc. REM statements in no way effect the operation of the program, except to add length to it (and therefore slow it down). No other executable statement can follow a REM on the same line.

EXAMPLE:

10 NEXT X: REM THIS LINE BUMPS X.

45. RESTORE - reposition READ pointer at specified DATA statement

RESTORE [line #]

When executed in a program, the pointer to the item in a DATA statement that is to be read next is reset to the first item in the DATA statement. This provides the capability to reREAD the information. If a [line number] follows the RESTORE statement, the pointer is set to that line. Otherwise the pointer is reset to the beginning of the BASIC program.

46. RESUME - define program execution after trap

RESUME [line # / NEXT]

Used to return to execution after TRAPping an error. With no arguments, RESUME attempts to re-execute the line in which the error occurred. RESUME NEXT resumes execution at the next statement following the one containing the error; RESUME line number will GOTO the specific line and resume execution from that line number. RESUME can only be used in program mode.

47. RETURN - return from subroutine

RETURN

This statement is always used with the GOSUB statement. When the program encounters a RETURN statement, it goes to the statement immediately following the last GOSUB command executed. If no GOSUB was previously issued, then a RETURN WITHOUT GOSUB ERROR message is posted, and program execution is stopped.

Note: All subroutines should be exited via a return to reduce stack verhead.

48. RREG - get register values on return from SYS call

```
RREG [a_var][,[x_var][,[y_var][,s_var]]]
```

Assign to variables a_var, x_var, y_var and s_var the contents of the a, x, y, and s registers on return from the last SYS call.

49. SCALE - alter scaling in graphics mode

```
SCALE [1/0] [,xmax,ymax]
            ,xmax >= 320
            ,ymax >= 200
                default xmax,ymax =1023
```

The scaling of the bit maps in multicolor and high resolution modes can be changed with the SCALE statement. Note: SCALE 0 turns scaling off. All future graphics will be drawn at default scale (300x200).

Entering:

```
SCALE 1

  turns scaling on. Coordinates may then be
 scaled from 0 to 1023 in both X and Y
 rather than the normal scale values,
 which are:

multicolor mode ......  X = 0 to 159  Y = 0 to 199
high resolution mode .  X = 0 to 319  Y = 0 to 199
```

50. SCNCLR - clear screen

```
SCNCLR [arg]
```

With no arg clears the graphic screen, if any, else the current text screen. where arg = 0 for 40 column normal text 1 for high res graphics 2 for high res graphics, split screen 3 for multicolor graphics 4 for multicolor graphics, split screen 5 for 80 column text

51. SPRITE - Set up sprite attributes

```
SPRITE <number> [,[on/off][,[fgnd][,[priority][,[x-exp]
                [,[y-exp][,mode]]]]]]

      where <number>  is sprite number (1-8)
            on/off    is sprite enable (1) or disable(0)
            fgnd      is sprite foreground color (1-16)
            priority  is sprite to display data priority:
                       0/ sprite over display data
                       1/ display data over sprite
            x-exp,y-exp  - sprite expansion on (1) or off (0)
            mode         - Sprite mode: 0/ High resolution
                                        1/ Multicolor
```

Unspecified parameters cause those characteristics to stay. Don't confuse graphics modes (multicolor, high resolution, split screen) with sprite modes (multicolor, high resolution). High resolution sprites can appear in multicolor graphic mode and vice versa. To check the characteristics of a sprite use the RSPRITE function.

52. SSHAPE/GSHAPE - save/retrieve shapes using strings

SSHAPE and GSHAPE are used to save and restore rectangular areas of multicolor or high resolution screens using BASIC string variables. The command to save an area is :

```
SSHAPE string_variable, a1, b1 [,a2,b2]

    string_variable .. String name to save data in
    a1,b1 ............ Corner coordinate (scaled)
    a2,b2 ............ Corner coordinate opposite
                       (a1,b1) (default is the PC)
```

NOTE: This is a subset of MOVESHAPE command. It is retained only for C64 mode.

Because BASIC limits strings to 255 characters, the size of the area that can be saved is limited. The string size required can be calculated using one of the following (unscaled) formulas:

```
L(mcm) = INT ( (ABS(a1-a2) + 1) / 4 + .99) * (ABS(b1-b2)+1)+4
L(h-r) = INT ( (ABS(a1-a2) + 1) / 8 + .99) * (ABS(b1-b2)+1)+4
```

```
GSHAPE string [, [a,b] [,mode] ]

   string .... Contains shape to be drawn
   a,b ....... Top left coordinate telling where to
               draw the shape (scaled - the default
               is the PC)
   mode ...... Replacement mode:
                 0: place shape as is (default)
                 1: place field inverted shape
                 2: OR shape with area
                 3: AND shape with area
                 4: XOR shape with area
```

NOTE: beware using modes 1-4 with Multi-color shapes, as the color source is part ot the bit pattern in multi-color mode.

53. STOP - halt program execution

```
STOP
```

This statement halts the program. A message, BREAK IN LINE xxx, (only in program mode vs. direct mode) where xxx is the line number containing the STOP. The program can be restarted at the statement following STOP if the CONT command is used. The STOP statement is usually used while debugging a program.

54. SYS - execute machine language subroutine at specified address

```
SYS address [,[a][,[x][,[y][,s]]]
```

Perform a call to subroutine at given address in a memory configuration set up by the BANK statement. Optionally, arguments a,x,y and s are loaded into the accumulator, x, y and status registers before the subroutine is called.

Address range is 0 to 65535. The program begins executing the machine language program starting at that memory location. This is similar to the USR function, but does not pass a parameter. SYS is bank sensitive. (SEE BANK)

55. TEMPO - define note duration

TEMPO n

n ..... Relative duration (1-255)

The actual duration is determined by using the formula given below:

duration = 19.22/n seconds

The default value is 8, and note duration increases with n.

56. TRAP - handle error processing

TRAP [line #]

When turned on, TRAP intercepts all BASIC execution error conditions (including the STOP KEY) except "UNDEF'D STATEMENT ERROR". In the event of any execution error, the error flag is set, and execution is transferred to the line number in the TRAP statement. The line number in which the error occurred can be found by using the system variable EL. The specific error condition is contained in system variable ER. The string function ERR$(ER) gives the error message corresponding to any error condition ER.

NOTE: An error in a TRAP routine cannot be trapped. The RESUME statement can be used to resume execution. TRAP with no line number turns off error TRAPping.

57. TRON - BASIC program trace enable

TRON

TRON is used in program debugging. This statement begins trace mode.

58. TROFF - BASIC program trace disable

TROFF

This statement turns off trace mode.

59. VOL - define output level of sound

VOL volume level

This statement sets the volume for SOUND statements. VOLUME can be set from 0 to 15, where 15 is the maximum volume, and 0 is off. VOL affects all 3 voices.

60. WAIT - halt program execution until data condition satisfied

WAIT <location>, <mask-1> [,mask-2>]

The WAIT statement causes program execution to be suspended until a given memory address recognizes a specified bit pattern. In other words, WAIT can be used to halt the program until some external event has occurred. This is done by monitoring the status of bits in the Input Output registers. The data items used with the WAIT can be any numeric expressions, but they will be converted to integer values. For most programmers, this statement should never be used. It

causes the program to halt until a specific memory location's bits change in a specific way. This is used for certain I/O operations and almost nothing else. The WAIT statement takes the value in the memory location and performs a logical AND operation with the value in mask-1. If there is a mask-2 in the statement, the result of the first operation is exclusively ORed with mask-2. In other words mask-1 "filters out" any bits not to be tested. Where the bit is 0 in mask-1, the corresponding bit in the result will always be 0. The mask-2 value flips any bits, so that an off condition can be tested for as well as an on condition. Any bits being tested for a 0 should have a 1 in the corresponding position in mask-2. If corresponding bits of the <mask-1> and <mask-2> operands differ, the exclusive-OR operation gives a bit result of 1. If the corresponding bits get the same result the bit is 0. It is possible to enter a infinite pause with the WAIT statement, in which case the RUN/STOP and RESTORE keys can be used to recover. Hold down the RUN/STOP key and the press RESTORE. The first example below WAITs until a key is pressed on the tape unit to continue with the program. The second example will WAIT until a sprite collides with the screen background. NOTE: The bank must be set.

EXAMPLES:

```
WAIT 1, 32, 32
WAIT 53273, 6, 6
WAIT 36868, 144, 16
```

(144 & 16 are masks. 144 = 10010000
in binary and 16 = 10000 in binary.

The WAIT statement will halt the
program until the 128 bit is on or
until the 16 bit is off)

NOTE: Wait may require use of BANK statement before use to access memory not currently in context.

5.1.1.5 ALPHABETICAL LIST OF FUNCTIONS -

1. ABS
2. ASC
3. ATN
4. BUMP
5. CHR$
6. COS
7. DEC
8. ERR$
9. EXP
10. FNxx
11. FRE
12. HEX$
13. INSTR
14. INT
15. JOY
16. LEFT$
17. LEN
18. LOG
19. MID$
20. PEEK
21. PEN
22. PI
23. POINTER
24. POS
25. POT
26. RCLR
27. RDOT
28. RGR
29. RIGHT$
30. RND
31. RSPCOLOR
32. RSPPOS
33. RSPRITE
34. RWINDOW
35. SGN
36. SIN
37. SPC
38. SQR
39. STR$
40. TAB
41. TAN
42. USR
43. VAL
44. XOR

5.1.1.6 FUNCTION DESCRIPTION -

FUNCTION (argument)

Where the argument can be a numeric value, variable, or string.

1. ABS - absolute value

ABS(X)

The absolute value function returns the positive value of the argument X.

2. ASC - returns CBM ASCII code for character

ASC(X$)

This function returns the ASCII code of the first character of X$.

3. ATN - returns angle whose tangent is X radians

ATN(X)

This function returns the angle whose tangent is X, measured in radians.

4. BUMP - returns sprite collision information

BUMP(N)

To determine which sprites have collided since the last check use the BUMP function: BUMP(1) records which sprites have collided with each other and BUMP(2) records which sprites have collided with the display data. COLLISION need not be active to use BUMP. The bit positions (7-0) in the BUMP value correspond to a sprite's number. In the case of multiple compares consider a sprite's position (via RSPRITE) to determine which sprite hit what. BUMP(n) is reset to zero after each call.

5. CHR$ - returns ASCII character for specified CBM ASCII code

CHR$(X)

This is the opposite of ASC, and returns the string character whose CBM ASCII code is X.

NOTE: When using CHR$(14) and CHR$(142) in 40 column text mode the entire screen reflects all upper or all lower case, whereas in 80 column text mode, upper and lower case characters can be displayed at the same time on the screen, giving the user the capability to display 512 characters.

6. COS - returns cosine for angle of X radians

COS(X)

This function returns the value of the cosine of X, where X is an angle measured in radians.

7. DEC - returns decimal value of hexadecimal number string, "0-FFFF"

DEC (hexidecimal-string)

8. ERR$ - returns string describing error condition

ERR$(N)

This function returns a string describing an error condition.

9. EXP - return value of an approximation of e (2.7182813) raised to the X power

EXP(X)

This function returns a value of e (2.71828183) raised to the power of X.

10. FNxx - returns value from user defined function

FNxx(x)

This function returns the value from the user-defined function xx created in a DEF FNxx statement.

11. FRE - returns number of unused bytes in memory

FRE(X)

where x = bank number (0-15)
where x = 0 for text storage available
x = 1 for variable storage available

This function returns the number of unused bytes available in memory. X is a bank #.

12. HEX$ - returns hexadecimal number string from decimal number

HEX$(X)

This function returns a 4 character string containing the hexadecimal representation of value X (0<=X<65535).

13. INSTR - returns position of string 1 in string 2

INSTR (string 1, string 2 [,starting position])

This function returns the position of string 1 in string 2 at or after the starting position. The starting position defaults to the beginning of string 1. If no match is found, a value of 0 is returned.

14. INT - returns integer form (truncated) of floating point value

INT(X)

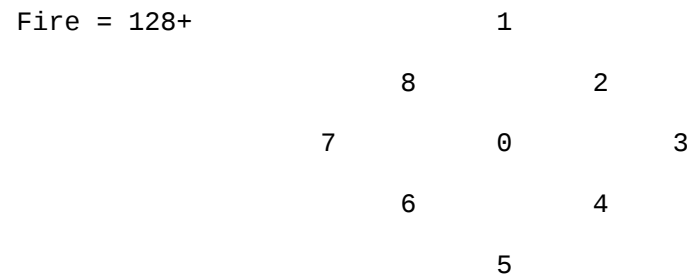
This function returns the integer value of the expression. If the expression is positive, the fractional part is left out. If the expression is negative, any fraction causes the next lower integer to be returned.

15. JOY - returns position of joystick and fire button state

JOY(N)

When N=1 returns position of joystick 1
When N=2 returns position of joystick 2

Any value of 128 or more means that the fire button is also depressed. The direction is indicated as follows:



EXAMPLE:

JOY(2) = 135 Joystick 2 fires to the left.

16. LEFT$ - returns N leftmost characters of string

LEFT$ (<string>,<integer>)

This function returns a string comprised of the leftmost <integer> characters of the string. The integer argument value must be in the range 0 to 255. If the integer is greater than the length of the string, the entire string will be returned. If an <integer> value of zero is used, then a null string (of zero length) will be returned.

17. LEN - returns length of string

LEN (<string>)

This function returns the number of characters in the string expression. Non-printed characters and blanks are counted.

18. LOG - returns natural log of X

LOG(X)

This function returns the natural log of X. The natural log is log to the base e (see EXP(X)). To convert to log base 10, divide by LOG(10).

19. MID$ - return substring from a larger string

MID$ (<string>, <numeric-1> [,<numeric-2>])

This function returns a sub-string which is taken from within a larger <string> argument. The starting position of the substring is defined by the <numeric-1> argument and the length of the sub-string by the <numeric-2> argument. Argument 1 is the pointer and can range from 1 to 255. Argument 2 is the length and can range from 0 to 255. If the <numeric-1> value is greater than the length of the <string>, or if the <numeric-2> value is zero, then MID$ gives a null string value. If the <numeric-2> argument is left out, then the computer will assume that a length of the rest of the string is to be used.

20. PEEK - returns content at specified memory location

PEEK(X)

This function gives the contents of memory location X, where X is located in the range 0 to 65535, returning a result from 0 to 255. This often used in conjunction with the POKE statement. The data will be from the 64K bank selected by the most recent BANK command.

21. PEN - returns X and Y coordinates of light pen

PEN (n)

when n=0 ..... X coordinate of light pen position on VIC
n=1 ..... Y coordinate of light pen position on VIC
n=2 ..... X row and column of character in 80 column mode
n=3 ..... Y row and column of character in 80 column mode
n=4 ..... returns 0 if no light pen triggered
returns 1 if light pen triggered
- resets so next read will be different

EXAMPLE:

```
100 DO UNTIL PEN(4):LOOP
105 PRINT PEN(2),PEN(3)
```

Note that, like sprite coordinates, the PEN value is not scaled and uses "real" coordinates, not graphic bit map coordinates. The X position is given as an even number ranging from approximately 60 to 320 while the Y position can be any number from about 50 to 250, staying within the surrounding border area. A value of zero for either position means the light pen is off screen and has not triggered an interrupt since the last read. Note that COLLISION need not be active to use PEN. A white background is usually required to stimulate the light pen. PEN values vary from CRT to CRT.

22. PI symbol -

PI symbol - returns value for PI (3.14159265)

23. POINTER - return address of the descriptor for variable name

POINTER (variable_name)

EXAMPLE:

```
A = POINTER(maxcnt)  - returns the address
                       of the descriptor maxcnt
                       in A.
```

NOTE: All references to the descriptor must have the bank set to bank 1 (where descriptors are located)

24. POS - returns character cursor position on line

POS (<dummy>)

This function returns the current cursor column within the current screen window.

25. POT - returns value of game paddle potentiometer

POT (n)
when n=1 ..... Position of paddle #1
n=2 ..... Position of paddle #2
n=3 ..... Position of paddle #3
n=4 ..... Position of paddle #4

The values for POT range from 0 to 255. Any value of 256 or more means that the fire button is also depressed.

26. RCLR - returns color of color source

RCLR(N)

This function returns the color assigned to source n (0<= n <= 6)

Where 0 = VIC background
1 = foreground
2 = multicolor 1
3 = multicolor 2
4 = VIC border
5 = text color
6 = 8563 background color

Color range is 1 to 16

27. RDOT - returns current position or color of pixel cursor.

RDOT(N)

This function returns information about the current position of the pixel cursor (PC) at XPOS/YPOS.

Where N = 0 for XPOS
= 1 for YPOS
= 2 color source

28. RGR - returns current graphic mode

RGR(X)

This function returns current graphic mode (X is a dummy argument).

29. RIGHT$ - returns substring from rightmost end of string

RIGHT$ (<string>, <numeric>)

This function returns a sub-string taken from the rightmost end of the <string> argument. The length of the sub-string is defined by the <numeric> argument which can be any integer in the range of 0 to 255. If the value of the numeric expression is zero, then a null string ("") is returned. If the value given in the <numeric> argument is greater than the length of the <string> then the entire string is returned.

30. RND - returns random number

RND(X)

This function returns a random number between 0 and 1. This is useful in games, to simulate dice rolls, and other elements of chance, and is also used in some statistical applications. The first random number should be generated by the formula RND(-TI), to start things off differently every negative value for X seeds the random number generator using X and gives a random number sequence. Using the same negative number for X as a seed results in the same sequence of random

numbers. A positive value gives random numbers based on the previous seed.

31. RSPCOLOR - Check what sprite Multi-Color values last set.

```
RSPCOLOR (<register>)

     where register =1 for return of multi-color
     color mode as a number 1-16.
     where register =2 for return of color code
     for SPRITE multi-color 2
```

32. RSPPOS - check speed and position of sprite

```
RSPPOS(<sprite>,<data>)

        where sprite identifies which sprite is being checked,
              data specifies what information is to be returned.

              data=0 - current X position is returned
                  =1 - current Y position is returned
                  =2 - speed is returned ( 0-15)
```

33. RSPRITE - returns sprite attributes

```
RSPRITE (N,X)

 Where N = sprite number (1-8)
 Where  x= 0      Enabled(1) / disabled(0)
           1      Sprite color  (1-16)
           2      Priority over background  yes =1, no=0
           3      Expand in X direction     yes =1, no=0
           4      Expand in Y direction     yes= 1, no=0
           5      multicolor                yes= 1, no=0

  RSPRITE is used to test various sprite attributes
  and properties.
```

34. RWINDOW

This is a function that returns information about the current console display environment.

```
RWINDOW (n)

where: n=0 : number of lines in the current window
       =1 : number of columns in the current window
       =2 : returns either 40 or 80, depending on the
            current console device
```

35. SGN - return sign of argument

```
SGN(X)
```

This function returns the sign of X. The result is + 1 if X > 0, 0 if X = 0, and -1 if X < 0.

36. SIN - return sine of argument

```
SIN(X)
```

This is the trigonometric sine function. The result is the sine of X. X is measured in radians.

37. SPC - adds space characters to output

SPC (<numeric>)

The SPC function is used to control the formatting of data, as either an output to the screen or into a logical file. The number of SPaCes given by the <numeric> argument is the # of character positions skipped over.. For screen or tape files the value of the argument is in the range 0 to 255 and for disk files up to 254. For printer files, an automatic carriage-return and line-feed will be performed by the printer if a SPaCe is printed in the last character position of a line. No SPaCes are printed on the following line.

38. SQR - returns square root of argument

SQR (<numeric>)

This function returns the value of the SQuare Root of the <numeric> argument. The value of the argument must not be negative, or the BASIC error message ?ILLEGAL QUANTITY is posted.

39. STR$ - returns string representation of number

STR$ (<numeric>)

This function returns the STRing representation of the numeric value of the argument. When the STR$ value is converted to each variable represented in the <numeric> argument, any number shown is followed by a space and, if it's positive, it is also preceded by a space. (negative #'s are preceeded by a "-" sign). Numbers in exponential form are printed as such also.

40. TAB - move cursor to tab position in present statement

TAB (<numeric>)

The TAB function moves the cursor forward if possible to a relative position on the text screen given by the <numeric> argument, starting with the left most position of the current line. The value of the argument can range from 0 to 255. The TAB function should only be used with the PRINT statement, since it has no effect if used with the PRINT# to a logical file. It also has no effect if the curent cursor position is beyond the given tab column.

41. TAN - returns tangent of X.

TAN(X)

This function returns the tangent of X, where X is an angle in radians.

42. USR - call user defined subprogram

USR(X)

When this function is used, the program jumps to a machine language program whose starting point is contained in memory locations 4633H and 4634H. The parameter X is passed to the machine language program in the floating point accumulator. Another number is passed back to the BASIC program through the

calling variable. In other words, this allows the user to exchange a variable between machine code and BASIC. The user must redirect the vector to his code before using, else "Illegal Quantity Error" is returned

Example:

```
A=USR(X)
```

43. VAL - returns numeric value of a number string

```
VAL(X$)
```

This function converts the string X$ into a number, and is essentially the inverse operation of STR$. The string is examined from the left most character to the right, for as many characters as are in recognizable number format. If the C-128 finds illegal characters, only the portion of the string up to that point is converted.

44. XOR

This function provides the exclusive - or of the argument values.

```
x = XOR (n1, n2)
```

where n1, n2 are 2 byte (0-65535) unsigned values.

5.1.1.7 VARIABLES -

The C-128 uses three types of variables in BASIC. These are: normal numeric, integer numeric, and string (alphanumeric) variables.

Normal NUMERIC VARIABLES, also called floating point variables, can have any value from up to nine digits of accuracy. When a number becomes larger than nine digits can show, as in +10 or -10, the computer displays it in scientific notation form, with the number normalized to 1 digit and eight decimal places, followed by the letter E and the power of ten by which the number is multiplied. For example, the number 12345678901 is displayed as 1.23456789E+10.

INTEGER VARIABLES can be used when the number is a signed whole number from +32767 to -32768. Integer data is a number like 5, 10, or -100. Integers take up less space than floating point variables, particularly when used in an array.

STRING VARIABLES are those used for character data, which may contain numbers, letters, and any other character that the C-128 can make. An example of string data is "Commodore 128".

VARIABLE NAMES may consist of a single letter, a letter followed by a number, or two letters. Variable names may be longer than 2 characters, but only the first two are significant. An integer is specified by using the percent (%) sign after the variable name. String variables have a dollar sign ($) after their names.

EXAMPLES:
Numeric Variable Names: A, A5, BZ
Integer Variable Names: A%, A5%, BZ%
String Variable Names: A$, A5$, BZ$

ARRAYS are lists of variables with the same name, using an extra number (or numbers) to specify an element of the array. Arrays are defined using the DIM statement, and may be floating point, integer, or string variable arrays. The array variable name is followed by a set of parentheses () enclosing the number of the variable in the list.

EXAMPLE:
A(7), BZ%(11), A$(87)

Arrays can have more than one dimension. A two dimensional array may be viewed as having rows and columns, with the first number identifying the row and the second number identifying the column (as if specifying a certain grid on the map).

EXAMPLE:
A(7,2), BZ%(2,3,4), Z$(3,2)

RESERVED VARIABLE NAMES are names that are reserved for use by the C-128, and may not be used for another purpose. These are the variables DS, DS$, ER, ERR$, EL, ST, TI, and TI$. KEYWORDS such as TO and IF or any other names that contain KEYWORDS, such as RUN, NEW, or LOAD cannot be used.

ST is a status variable for input and output (except normal screen/keyboard operations). The value of ST depends on the results of the last I/O operation. In general, if the value of ST is 0 then the operation was successful.

TI and TI$ are variables that relate to the real-time clock built into the C-128. The system clock is updated every 1/60th of a second.

It starts at 0 when the C-128 is turned on, and is reset only by changing the value of TI$. The variable TI gives the current value of the clock in 1/60ths of a second. TI$ is a string that reads the value of the real-time clock as a 24 hour clock. The first two characters of TI$ contain the hour, the 3rd and 4th characters are the minutes, and the 5th and 6th characters are the seconds. This variable can be set to any value (so long as all characters are numbers), and will be automatically updated as a 24 hour clock.

EXAMPLE:

TI$ = "101530" sets the clock to 10:15 and 30 seconds (AM)

The value of the clock is lost when the C-128 is turned off. It starts at zero when the C-128 is turned on, and is reset to zero when the value of the clock exceeds 235959 (23 hours, 59 minutes, and 59 seconds).

The variable DS reads the disk drive command channel, and returns the current status of the drive. To get this information in words, PRINT DS$. These status variables are used after a disk operation, like a DLOAD or DSAVE, to find out why the red error light on the disk drive is blinking.

ER, EL, and ERR$ are variables used in error trapping routines. They are usually only useful within a program. ER returns the last error encountered since the program was RUN. EL is the line where the error occurred. ERR$ is a function that allows the program to print one of the BASIC error messages. PRINT ERR$(ER) prints out the proper error message.

5.1.1.8 OPERATORS -

The BASIC OPERATORS include ARITHMETIC, RELATIONAL, and LOGICAL OPERATORS. The ARITHMETIC operators include the following signs:

+	addition
-	subtraction
*	multiplication
/	division
^	raising to a power (exponentiation)

On a line containing more than one operator, there is a set order in which operations always occur. If several operators are used together, the computer assigns priorities as follows: First, exponentiation, then multiplication and division, and last, addition and subtraction. If two operators have the same priority, then calculations are performed in order from left to right. If these operations are to occur in a different order, C-128 BASIC allows giving a calculation a higher priority by placing parentheses around it. Operations enclosed in parentheses will be calculated before any other operation. Make sure that the equations have the same number of left and right parentheses, or a SYNTAX ERROR message is posted when the program is run.

There are also operators for equalities and inequalities, called RELATIONAL operators. Arithmetic operators always take priority over relational operators.

=	is equal to
<	is less than
>	is greater than
<= or =<	is less than or equal to
>= or =>	is greater than or equal to
<> or ><	is not equal to

Finally, there are three LOGICAL operators, with lower priority than both arithmetic and relational operators:

AND
OR
NOT

These are most often used to join multiple formulas in IF ... THEN statements. When they are used with arithmetic operators, they are evaluated last (i.e., after + and -). If the relationship stated in the expression is the true the result is assigned an integer value of -1 and if false a value of 0 is assigned.

EXAMPLES:

```
IF A=B AND C=D THEN 100    requires both A=B & C=D to be true
IF A=B OR C=D THEN 100     allows either A=B or C=D to be true
A=5:B=4:PRINT A=B          displays a value of 0
A=5:B=4:PRINT A>3          displays a value of -1
PRINT 123 AND 15:PRINT 5 OR 7    displays 11 and 7
```

5.1.1.9 BASIC ERROR MESSAGES -

The following error messages are displayed by BASIC. Error messages can also be displayed with the use of the ERR$() function. The error number refers only to the number assigned to the error for use with this function.

ERROR #	ERROR NAME	DESCRIPTION
1	TOO MANY FILES	There is a limit of 10 files OPEN at one time.
2	FILE OPEN	An attempt was made to open a file using the number of an already open file.
3	FILE NOT OPEN	The file number specified in an I/O statement must be opened before use.
4	FILE NOT FOUND	Either no file with that name exists (disk) or an end-of-tape marker was read (tape).
5	DEVICE NOT PRESENT	The required I/O device not available or buffers deallocated (cassette).
6	NOT INPUT FILE	An attempt made to GET or INPUT data from a file that was specified as output only.
7	NOT OUTPUT FILE	An attempt was made to send data to a file that was specified as input only.
8	MISSING FILE NAME	filename missing in command.
9	ILLEGAL DEVICE NUMBER	An attempt made to use a device improperly (SAVE to the screen, etc).
10	NEXT WITHOUT FOR	Either loops are nested incorrectly, or there is a variable name in a NEXT statement that doesn't correspond with one in FOR.
11	SYNTAX ERROR	A statement is unrecognizable by BASIC. This could be because of missing or extra parenthesis, mispelled keyword, etc.
12	RETURN WITHOUT GOSUB	A RETURN statement encountered when no GOSUB statement was active.
13	OUT OF DATA	A READ statement encountered, without data left unREAD.
14	ILLEGAL QUANTITY	A number used as the argument of a function or statement is outside the allowable range.
15	OVERFLOW	The result of a computation is larger than the largest number allowed (1.701411833E+38).
16	OUT OF MEMORY	Either there is no more room for program and program variables, or there are too many DO, FOR, or GOSUB statements in effect.
17	UNDEF'D STATEMENT	A line number referenced doesn't exist in the program.
18	BAD SUBSCRIPT	The program tried to reference an element of an array out of the range specified by the DIM statement.
19	REDIM'D ARRAY	An array can only be DIMensioned once.
20	DIVISION BY ZERO	Division by zero is not allowed.
21	ILLEGAL DIRECT	INPUT or GET statements are only allowed within a program.

22	TYPE MISMATCH	This occurs when a number is used in place of a string or vice-versa.
23	STRING TOO LONG	A string can contain up to 255 characters.
24	FILE DATA	Bad data read from a tape file.
25	FORMULA TOO COMPLEX	Simplify the expression (break into two parts or use fewer parentheses).
26	CAN'T CONTINUE	The CONT command does not work if the program was not RUN, there was an error, or a line has been edited.
27	UNDEFINED FUNCTION	A user defined function referenced that was never defined.
28	VERIFY	The program on tape or disk does not match the program in memory.
29	LOAD	There was a problem loading. Try again.
30	BREAK	The stop key was hit to halt program execution.
31	CAN'T RESUME	A RESUME statement encountered without TRAP statement in effect.
32	LOOP NOT FOUND	The program has encountered a DO statement and cannot find the corresponding LOOP.
33	LOOP WITHOUT DO	LOOP encountered without a DO statement active.
34	DIRECT MODE ONLY	This command allowed only in direct mode, not from a program.
35	NO GRAPHICS AREA	A command (DRAW, BOX, etc.) to create graphics encountered before the GRAPHIC command was executed.
36	BAD DISK	An attempt failed to HEADER a diskette, because the quick header method (no ID) was attempted on an unformatted diskette, or the diskette is bad.
37	BEND NOT FOUND	A BEND statement not found for BEGIN
38	LINE NUMBER TOO LARGE	Line number cannot exceed 64000

5.1.1.10 DOS ERROR MESSAGES -

The following error messages are returned through the DS and DS$ variables. NOTE: Error message numbers less than 20 should be ignored with the exception of 01, which gives information about the number of files scratched with the SCRATCH command.

ERROR #	DESCRIPTION
20:	READ ERROR (block header not found) The disk controller is unable to locate the header of the requested data block. Caused by an illegal sector number, or the header has been destroyed.
21:	READ ERROR (no sync character) The disk controller in unable to detect a sync mark on the desired track. Caused by misalignment of the read/write head, no diskette is present, or unformatted or improperly seated diskette. Can also indicate a hardware failure.
22:	READ ERROR (data block not present) The disk controller has been requested to read or verify a data block that was not properly written. This error occurs in conjunction with the BLOCK commands and indicates an illegal track and/or sector request.
23:	READ ERROR (checksum error in data block) This error message indicates that there is an error in one or more of the data bytes. The data has been read into the DOS memory, but the checksum over the data is in error. This message may also indicate grounding problems.
24:	READ ERROR (byte decoding error) The data or header has been read into the DOS memory, but a hardware error has been created due to an invalid bit pattern in the data byte. This message may also indicate grounding problems.
25:	WRITE ERROR (write-verify error) This message is generated if the controller detects a mismatch between the written data and the data in the DOS memory.
26:	WRITE PROTECT ON This message is generated when the controller has been requested to write a data block while the write protect switch is depressed.
27:	READ ERROR This message is generated when a checksum error is in the header.
28:	WRITE ERROR This error message is generated when a data block is too long.
29:	DISK ID MISMATCH This message is generated when the controller has been requested to access a diskette which has not been initialized. The message can also occur if a diskette has a bad header.
30:	SYNTAX ERROR (general syntax) The DOS cannot interpret the command sent to the command channel. Typically, this is caused by an illegal number of file names, or patterns are illegally used. For example, two file names appear on the left side of the COPY command.
31:	SYNTAX ERROR (invalid command) The DOS does not recognize the command. The command must start in the first position.
32:	SYNTAX ERROR (invalid command) The command sent is longer than 58 characters.
33:	SYNTAX ERROR (invalid file name) Pattern matching is invalidly used in the OPEN or SAVE command.
34:	SYNTAX ERROR (no file given)

The file name was left out of the command or the DOS does not recognize it as such.

39: SYNTAX ERROR (invalid command)
This error may result if the command sent to the command channel (secondary address 15) is unrecognized by the DOS.

40: UNIMPLEMENTED COMMAND
Command is not implemented at this time.

41: FILE READ
The file cannot be read

50: RECORD NOT PRESENT
Result of disk reading past the last record through INPUT# or GET# commands. This message will also occur after positioning to a record beyond end_of_file in a relative file. If the intent is to expand the file by adding the new record (with a PRINT# command), the error message may be ignored. INPUT and GET should not be attempted after this error is detected without first repositioning.

51: OVERFLOW IN RECORD
PRINT# statement exceeds record boundary. Information is truncated. Since the carriage return which is sent as a record terminator is counted in the record size, this message will occur if the total characters in the record (including the final carriage return) exceeds the defined size.

52: FILE TOO LARGE
Record position within a relative file indicates that disk overflow will result.

60: WRITE FILE OPEN
This message is generated when a write file that has not been closed is being opened for reading.

61: FILE NOT OPEN
This message is generated when a file is being accessed that has not been opened in the DOS. Sometimes, in this case, a message is not generated; the request is simply ignored.

62: FILE NOT FOUND
The requested file does not exist on the indicated drive.

63: FILE EXISTS
The file name of the file being created already exists on the diskette.

64: FILE TYPE MISMATCH

65: NO BLOCK

66: ILLEGAL TRACK AND SECTOR
The DOS has attempted to access a track or block which does not exist in the format being used. This may indicate a problem reading the pointer of the next block.

67: ILLEGAL SYSTEM T OR S
This special error message indicates an illegal system track or sector.

70: NO CHANNEL (available)
The requested channel is not available, or all channels are in use. A maximum of five sequential files may be opened at one time to the DOS. Direct access channels may have six opened files.

71: DIRECTORY ERROR

72: DISK FULL
Either the blocks on the diskette are used or the directory is at its entry limit. DISK FULL is sent when two blocks are available on the 1571 to allow the current file to be closed.

73: DOS MISMATCH
DOS 1 and 2 are read compatible but not write compatible. Disks may be interchangeably read with either DOS, but a disk formatted on one version cannot be written upon with the other version because the format is different. This er-

ror is displayed whenever an attempt is made to write upon a disk which has been formatted in a non-compatible format. (A utility routine is available to assist in converting from one format or another.) This message may also appear after power up.

74: DRIVE NOT READY
An attempt has been made to access the Floppy Disk Drive without any diskette present.

5.2 MACHINE LANGUAGE MONITOR

5.2.1 INTRODUCTION

The C-128 MONITOR is a built in machine language program that lets the user easily write machine language programs. C-128 MONITOR includes a machine language monitor, a mini assembler, and a disassembler.

Machine language programs written using C-128 MONITOR can run by themselves, or be used as very fast 'subroutines' for BASIC programs since C-128 MONITOR has the ability to coexist peacefully with BASIC.

Care must be taken to position the assembly language programs in memory so that the BASIC program does not overwrite them.

5.2.2 C-128 MONITOR COMMANDS

1. A ASSEMBLE - Assemble a line of 6502 code
2. C COMPARE - Compare two sections of memory and report differences
3. D DISASSEMBLE - Disassemble a line of 6502 code
4. F FILL - Fill memory with the specified byte
5. G GO - Start execution at specified address
6. H HUNT - Hunt through memory within a specified range for all occurences of a set of bytes
7. L LOAD - Load a file from tape or disk
8. M MEMORY - Display the hexidecimal values of memory locations
9. R REGISTERS - Display the 6502 registers.
10. S SAVE - Save to tape or disk
11. T TRANSFER - Transfer code from one section of memory to another
12. V VERIFY - Compare memory with tape or disk
13. X EXIT - EXIT C-128 MONITOR
14. (period) - Assembles a line of 6502 code
15. > (greater than) - Modifies memory
16. ; (semi-colon) - Modifies 6502 register displays
17. @ (at sign) - Display disk status

1/ The MONITOR now accepts binary, octal, decimal and hexidecimal values for any numeric field. This was accomplished by totally re-coding PARSE and portions of ASSEM, and installing a new routine called EVAL. Numbers prefixed by one of the characters $ + & % are interpreted by EVAL as base 16, 10, 8, or 2 values respectively. In the absence of a prefix, the base defaults to hexidecimal always. ASSEM will use the zero-page form wherever possible unless the value is preceeded by extra zeros to force the absolute form (except binary notation).

2/ The MONITOR now performs limited number conversion. Additions were made to MAIN1 and CMDCHR and a new routine called CONVERT was installed to handle the conversions. Any of the characters $ + & % entered as a command and prefixing a numeric value are PARSEd (see #1 above) and the hexidecimal value printed. Full conversion between bases may be added in a later release.

The 5th most significant nybble of the address field specifies the memory configuration to implement at the time the given command is executed. There are 16 (0-$F) possible memory configurations. Refer to the 'Memory Configuration Table' below for the specific assignments. (These assignments apply to all BASIC and KERNAL routines).

Example of memory display in monitor mode:

MEMORY CONFIGURATION TABLE

where x =

0 - RAM 0 only
1 - RAM 1 only
2 - RAM 2 only
3 - RAM 3 only
4 - INT ROM, RAM 0, I/O
5 - INT ROM, RAM 1, I/O
6 - INT ROM, RAM 2, I/O
7 - INT ROM, RAM 3, I/O
8 - EXT ROM, RAM 0, I/O
9 - EXT ROM, RAM 1, I/O
A - EXT ROM, RAM 2, I/O
B - EXT ROM, RAM 3, I/O
C - KERNAL + INT (lo), RAM 0, I/O
D - KERNAL + EXT (lo), RAM 1, I/O
E - KERNAL + BASIC, RAM 0, CHARROM
F - KERNAL + BASIC, RAM 0, I/O

5.2.2.1 C-128 MONITOR COMMAND DESCRIPTIONS -

COMMAND: A
PURPOSE: Enter a line of assembly code.
SYNTAX: A <address> <opcode mnemonic> <operand>

<address> A hexadecimal number indicating the location in memory to place the opcode.
<opcode mnemonic> A standard MOS technology assembly language mnemonic, eg., LDA, STX, ROR
<operand> The operand, when required, can be of any of the legal addressing modes. (For zero-page modes a 2 digit hex number is whose value is less than $100. For non-zero page addresses 4 digit hex numbers are required.)

A RETURN is used to indicate the end of the assembly line. If there are any errors on the line, a question mark is displayed to indicate an error, and the cursor moves to the next line. The screen editor can be used to correct the error(s) on that line.

EXAMPLE:

```
.A 1200 LDX #$00
.A 1202
```

NOTE: A period (.) is equal to the ASSEMBLE command.

EXAMPLE:

```
. 2000 LDA #$23
```

COMMAND: C
PURPOSE: Compare two areas of memory.
SYNTAX: C <address 1> <address 2> <address 3>

<address 1> is a hexadecimal number indicating the start of the area of memory to compare against.
<address 2> is a hexadecimal number indicating the end of the area of memory to compare against.
<address 3> is a hexadecimal number indicating the start of the other area of memory to compare with.

Addresses that do not agree are printed on the screen.

COMMAND: D
PURPOSE: Disassemble machine code into assembly language mnemonics and operands.
SYNTAX: D [<address>] [<address 2>]

<address> A hexadecimal number setting the address to start the disassembly.
<address 2> An optional hexadecimal ending address of code to be disassembled.

The format of the disassembly is only slightly different than the input format of an assembly. The difference is that the first character of a disassembly is a comma rather than an A (for readability), and the hexadecimal of the code is listed as well.

A disassembly listing can be modified using the screen editor. Make any changes to the mnemonic or operand on the screen, then hit the carriage return. This enters the line and calls the assembler for

further instructions.

A disassembly can be paged. Typing a D <return>causes the next page of disassembly to be displayed.

EXAMPLE:

```
D 3000 3003
   .3000 A900   LDA #$00
   .3002 FF     ???
   .3003 D0 2B  BNE $3030
```

COMMAND: F
PURPOSE: Fill a range of locations with a specified byte.
SYNTAX: F <address 1> <address 2> <byte>

<address 1> The first location to fill with the <byte>.
<address 2> The last location to fill with the <byte>.
<byte value> A 1 or 2 digit hexadecimal number to be written

This command is useful for initializing data structures or any other RAM area.

EXAMPLE:

```
F 0400 0518 EA
```

Fills memory locations from $0400 to $0518 with $EA (a NOP instruction).

COMMAND: G
PURPOSE: Begin execution of a program at a specified address.
SYNTAX: G <address>

<address> The address where execution is to start. When address is left out, execution begins at the current PC. (The current PC can be viewed using the R command.)

The GO command restores all registers (displayable by the R command) and begins execution at the specified starting address. Caution is recommended in using the GO command. To return to C-128 MONITOR mode after executing a machine language program, use the BRK instruction at the end of the program.

EXAMPLE:

```
G 140C
```

Execution begins at location $140C.

COMMAND: H
PURPOSE: Hunt through memory within a specified range for all occurences of a set of bytes.
SYNTAX: H <address 1> <address 2> <data>

<address 1> beginning address of hunt procedure
<address 2> ending address of hunt procedure
<data> data set to search for data may be hexadecimal or an ASCII string.

EXAMPLE:

```
H A000 A101 A9 FF 4C   Search for data $A9,
```

$FF, $4C, from A000
to A101.

COMMAND: L
PURPOSE: Load a file from cassette or disk.
SYNTAX: L <"filename">[, <device>] [,alt_load_address]

<"filename"> Is any legal C-128 filename.
<device> Is a hexadecimal number indicating the device to load from.

[alt_load_address] option to load a file to a specified address.

1 is cassette
8 is disk (or 9, A, etc.)

The LOAD command causes a file to be loaded into memory. The starting address is contained in the first two bytes of the file (a program file). In other words, the LOAD command always loads a file into the same place it was saved from. This is very important in machine language work, since few programs are completely relocatable. The file is loaded into memory until the end of file (EOF) is found.

EXAMPLE:

L "PROGRAM 1",8 Load the file named PROGRAM from the disk.

COMMAND: M
PURPOSE: To display memory as a hexadecimal and ASCII dump within the specified address range.
SYNTAX: M [<address 1>] [<address 2>]

<address 1> First address of memory dump. Optional; if omitted, one page is displayed. The first byte is the bank number to be displayed, the next 4 bytes are the first address to be displayed.
<address 2> Last address of memory dump. Optional; if omitted, one page is displayed. The first byte is the bank number to be displayed, the next 4 bytes are the ending address to be displayed.

Memory is displayed in the following format:

>1A048 41 E7 00 AA AA 00 98 65 45 :A!.*..VE

Memory content may be edited using the screen editor. Move the cursor to the data to be modified and type the desired correction and hit return. If there is a bad RAM location or an attempt to modify ROM has occured, an error flag (?) is displayed. An ASCII dump of the data is displayed in REVERSE (to contrast with other data displayed on the screen) to the right of the hex data. When a character is not printable, it is displayed as a reverse period (.). As with the disassembly command, paging down is accomplished by typing M and RETURN.

EXAMPLE:

>21C00 41 E7 00 AA AA 00 98 56 45 :A!.*..VE
>21C08 42 43 02 AZ AD 11 94 57 44 :BC.*..WD
>21C10 45 E7 00 DF FE 07 06 46 47 :E!.*..EF

COMMAND: R
PURPOSE: Show important 6502 registers. The program status register, the program counter, the accumulator, the X and Y index registers and the stack pointer are displayed.
SYNTAX: R

EXAMPLE:

```
R
    PC   SR  AC  XR  YR  SP
;  1002  01  02  03  04  F6
```

NOTE: ; (semi-colon) can be used to modify register displays in the same fashion as > can be used to modify memory registers.

COMMAND: S
PURPOSE: Save the contents of memory onto tape or disk.
SYNTAX: S <"filename">,<device>,<address 1>,<address 2>

<"filename">	Any legal C-128 filename. To save the data, the filename must be enclosed in double quotes. Single quotes cannot be used.
<address 1>	Starting address of memory to be saved.
<address 2>	Ending address of memory to be saved + 1. All data up to but not including the byte of data at this address is saved.

The file created by this command is a program file. The first two bytes contain the starting address <address 1> of the data. The file may be recalled by the L command.

EXAMPLE:

```
S "GAME",8,0400,0BFF
```

Saves memory from $0400 to $0BFF onto disk.

COMMAND: T
PURPOSE: Transfer segments of memory from one memory area to another.
SYNTAX: T <address 1> <address 2> <address 3>

<address 1>	Starting address of data to be moved.
<address 2>	Ending address of data to be moved.
<address 3>	Starting address of new location where data will be moved.

Data can be moved from low memory to high memory and vice-versa. Additional memory segments of any length can be moved forward or backward. An automatic 'compare' is performed as each byte is transfered and any differences are listed by address.

EXAMPLE:

```
T 1400 1600 1401
```

Shifts data from $1400 up to and including $1600 one byte higher in memory.

COMMAND: V
PURPOSE: Verify a file on cassette or disk with the memory contents.
SYNTAX: V <"filename">[, <device>] [,alt_start_address]

```
                <"filename">   Is any legal C-128 filename.
                <device>       Is a hexadecimal number indicating which
                               drive the file is on;

             cassette is 1 or 01, disk is 8 or 08, 09, etc.

                [alt_start_address] option to start verification at
                                    this address.
```

The verify command compares a file to memory contents. The C-128 responds with VERIFYING. If an error is found, the word ERROR is added; if the file is successfully verified, the cursor reappears.

EXAMPLE:

```
V "WORKLOAD", 8
```

```
COMMAND:   X
PURPOSE:   Exit to BASIC
SYNTAX:    X


COMMAND:   > (greater than)
PURPOSE:   Can be used to set 1 to 8 memory locations at a time.
SYNTAX:    > address  data byte 1 <data byte 2...8>

             address:           First memory address to set
             data byte 1        Data (in HEX) to be put at address
             <data byte 2...8>: Data to be placed in the successive
                                memory locations following the first
                                address. (optional)

COMMAND:   @ (at sign)
PURPOSE:   Can be used to display the disk status
SYNTAX:    @ [ unit#], disk cmd string

             unit #             device unit number (in HEX)
             disk cmd string    String command to disk

           NOTE: @ alone gives the status of the disk drive.
```

5.3 C-128 EDITOR ESCAPE SEQUENCES

This section contains a definition of the escape sequences that are present in the C128.

The following is a definition of the ESCAPE sequences that are available on the C128 and a brief description of what each sequence does. ESCAPE sequences are entered by momentarily pressing the "ESC" key followed by the key listed below.

KEY	FUNCTION
A	Enable auto-insert mode
B	Set bottom of screen window at cursor position
C	Disable auto-insert mode
D	Delete current line
E	Set cursor to non-flashing mode
F	Set cursor to flashing mode
G	Enable bell (control-G)
H	Disable bell
I	Insert line
J	Move to start of current line
K	Move to end of current line
L	Enable scrolling
M	Disable scrolling
N	Return screen to normal (non-reverse video) state (80 column screen only)
O	Cancel insert, quote, and reverse modes
P	Erase to start of current line
Q	Erase to end of current line
R	Set screen to reverse video (80 column screen only)
S	Change to block cursor (80 column screen only)
T	Set top of screen window at cursor postion
U	Change to under (80 column screen only)
V	Scroll up
W	Scroll down
X	Swap 40/80 column display output device
Y	Set default tab stops (8 spaces)
Z	Clear all tab stops
@	Clear to end of screen

5.4 C128 EDITOR CONTROL CODES

The following control characters in the CBM ASCII table have been added or changed from those found in the C64. Codes not shown in this table have the same function as found in the C64.

CHR$ VALUE	KEYBOARD CONTROL	CHARACTER FUNCTION
2	B	Underline on (80 column screen only)
7	G	Produces bell tone
9	I	Tab character
10	J	Line feed character
11	K	Disable shift Commodore key (formerly code 9)
12	L	Enable shift Commodore key (formerly code 8)
15	O	Turn on flash on (80 column screen only)
24	X	Tab set/clear
27	[	Escape character
130		Underline off (80 column screen only)
143		Turn flash off (80 column screen only)

CHAPTER 6

SYSTEM MEMORY MANAGEMENT

6.1 INTRODUCTION

The MEMORY MANAGEMENT UNIT (MMU) is designed to allow complex control of the C-128 system memory resources. It handles all of the standard C64 modes of operation in a fashion as to be completely compatible with the C64. Additionally, it controls the management of particular C-128 modes including the Z80 mode. Below is a list of MMU features:

1. Generation of Translated Address Bus (TA8 - TA15).
2. Generation of control signals for different processor modes (C-128, C64, Z80).
3. Generation of CAS select lines for RAM banking.

6.2 C128 MEMORY ORGANIZATION

Essential to the understanding of the MMU is an understanding of the C-128 system memory organization, which is controlled through the registers of the MMU. These registers control the MMU's translation of addresses from the 8500 processor, totalling 64K bytes of address space, into the 1M bytes of RAM and up to 96K internal bytes and 32K bytes external ROM available to the C-128 system. Following is a diagram of the C-128 memory map.

insert map here from MMU spec pg 2-2

6.2.1 C-128 ROM MEMORY ORGANIZATION

Refer to figure 6.1 entitled C128 Memory Map. The memory map is an important consideration in maintaining C64 compatibility. The standard map is shown for the C64 mode and some of the alternate modes are shown in figure 6.2, C64 Alternate Memory Organization (To be supplied). All C64 modes are completely compatible with the C64 computer, as the C-128 basically becomes a C64 when in C64 mode. The details of MMU Register location/operation are discussed further in this chapter.

C-128 mode is achieved at system reset, and is controlled by a bit in the MMU configuration register. In C-128 mode the MMU asserts itself in the C-128 memory map at $FF00 and in the I/O space starting at $D500. Use of MMU registers located at $FF00 allows memory management without actually having the I/O block banked in at the time and with a minimum loss of contiguous RAM. The MMU is completely removed from the memory map in C64 mode (though it is still used by the hardware to manage memory).

The ROM's in C64 mode, both internally and externally, look just like the C64 ROM's. The internal BASIC and KERNAL provide the C64 mode with the normal C64 operating system in ROM. This ROM actually duplicates some of the ROM used in C-128 Mode. In C-128 mode, up to 48K of Operating System is present, with the exact amount being set by software control. This allows quicker access to underlying RAM by turning off uneeded sections of the Operating System.

The External ROM's represented on the memory map are those used in the C64 mode, and obey the C64 rules for mapping, i.e. cartridges assert themselves in hardware via the /EXROM and /GAME lines. External ROM's in C-128 mode obey rules similar to the TED scheme for ROM banking, i.e. they are polled at system initialization to see if they exist and what priority they are to run with. This allows much more flexibility than the hardwire ROM substitution method, since the kernal and Basic ROM,s can be swapped out for an application program, swapped out for external program control, or turned off all together. This banking manipulation is accomplished by writing to the Configuration Register at location $D500 or $FF00 in the MMU.

The hardware also features the ability to store preset values for the configuration and force a load of the configuration register by writing to one of the LCR (LOAD CONFIGURATION REGISTER) registers. This allows the programmer to imply that any time an access to a bank where data is stored occurs, for example, that ROM not appear in the bank by default.

insert C64 Alt mem map here

6.2.2 C-128 RAM MEMORY ORGANIZATION

Refer again to figure 6.1, the C128 Memory Map. The RAM present in the system is actually composed of 128K by 8 bytes of contiguous DRAM. The RAM is accessed by banking in pages of 64K (the maximum range of the 8500 and Z80). The area shown as RAM is representative of what the microprocessor would see if all ROM were disabled. Bank switching can be accomplished in one of two ways.

The bank in use is a function of the value stored in the configuration register. (There are actually a couple of memory modes that override parts of the bank as selected here. These modes are mentioned and covered in detail in a later section.) A store to this register will always take effect immediately. An indirect store to this register, using programmed bank configuration values, can be accomplished by writing to one of the indirect load registers known as LCR's (Load Configuration Register), located in the $FF00 region of memory. By writing to an LCR the contents of its corresponding PCR (PreConfiguration Register) will be latched into the configuration register. This allows the programmer to set up four different preprogrammed configurations that allow each bank to be personalized ahead of time. i.e., Bank 1 being a Data Bank might be strictly a RAM bank with no ROM or I/O enabled, where Bank 0 being the system ROM and I/O enabled by default. Additionally , reading any LCR will return the value of its corresponding PCR.

When dealing with 64K banks of memory at once, it may be desirable to bank in Bank 1 but still retain the system RAM (Stack, Zero page, Screen,etc.) The MMU has provisions for what is referred to as commom RAM. This is RAM that does not bank, and is programmable in size as to whether it appears at the top, bottom, or both in the memory map. The size is set by bits 0 and 1 in the RAM Configuration Register (RCR). If the value of the bits is zero, 1K will be common. Values of one, two and three produce common areas of 4K, 8K, and 16K respectively. If bit two of the RCR is set, bottom memory is held common, if bit 3 is set, then top of memory is common. In all cases, common RAM is physically located in Bank 0.

Zero page and page one can be located (or relocated) independently of the RCR. When the processor accesses an address that falls within zero page or page 1, the MMU adds to the high order processor address the contents of the P0 register pair or the P1 register pair, respectively, and puts this new address on the bus, including the extended addressing bits A16 and A17. RAM banking will occur as appropriate to access the new address. Write to the P0 and P1 registers will be stored in a prelatch until a write to the low byte of the zero page register occurs. This prevents any change to the system configuration from occuring until all changes have been completed, preventing an invalid interim state.

At the same time, the contents of the P0 and P1 registers are applied to a digital comparator, and a reverse substitution occurs if the address of the 8500 falls within the page pointed to by the register. This results in not just relocating the zero or one page but swapping the zero or one page with the memory that it replaced. Note that upon system reset, the pointers are set to true zero page and true page one.

For VIC chip access, two bits in the MMU status register substitute for address lines A16 and A17, making it possible to steer the VIC anywhere in the 256K range.

Note that AEC is the mechanism that the MMU uses to steer a VIC

space address, i.e., when AEC is low a VIC access is assumed. This results in the VIC bank being selected as well for an outside DMA, since this too will pull the AEC line low.

6.3 MMU AND I/O MEMORY ORGANIZATION

The block of memory represented by the I/O block on figure 6-3 is an expanded view of the memory block entitled I/O +CHAROM, shown on the C-128 memory map, figure 6-1. When the I/O block exists (And it may not depending on the configuration in place at the time), access to VIC, SID, and I/O as well as the addition of the MMU can be accomplished. All I/O functions remain as they were previously on the C64 with the exception that the MMU has been added. With the exception of four registers that are asserted in the zero page in the C-128 mode, all new MMU registers appear in an unused slot in the memory I/O block, though they will only appear in C-128 mode. The descriptions for the MMU registers can be found in the section entitled THE MEMORY MANAGEMENT UNIT.

insert figure 6-2 here I/O block

6.4 MMU REGISTER DESCRIPTION

The MMU is the mechanism by which the various memory modes shown in the C-128 Memory Map are chosen. Additionally, the MMU provides for Z80 mode, which is not shown on that diagram. Following is a description of the MMU register types. The figure entitled MMU Register Map is provided to illustrate some of the text that is to follow. Note that in C64 mode the MMU completely disappears from the system's memory map.

Note that the data out of the MMU is valid only on AEC high. This is necessary to avoid bus contention during a VIC cycle.

insert MMU map here - fig 6-4.

6.4.1 THE CONFIGURATION REGISTER

The CONFIGURATION REGISTER (CR) controls the ROM, RAM, and I/O configuration of the C-128 system. It is located at $D500 in I/O space and at $FF00 in system space.

In C-128 mode, bit 0 controls whether an I/O space ($D000 - $DFFF) or a ROM/RAM access occurs. A low will select I/O, a high will enable some kind of ROM/RAM acccess, the nature of which is controlled by other bits in this register. The value of this bit is stored in a prelatch until the fall of the clock in order to prevent its changing in an unstable situation. In C64 mode of the I/OSE line, the hardware line driven by this bit, is forced high. Note that when not I/O space, the ROM/RAM access is controlled by the defined ROM Hi configuration bits, that are described later. This bit resets to 0. When the I/O bit is high, MMU registers D500 to D508 will assert themselves; when the bit is low, these registers disappear from the memory map. MMU registers FF00 to FF04 are always available in C128 mode. The hardware line I/OSE always reflects the polarity of this bit when in C128 mode.

Bit number 1 controls processor access to ROM low space ($4000 - $7FFF), in C-128 mode. If the bit is high, the area appears as RAM, and a RAM access CAS enable is generated to the appropriate RAM bank, which is determined by the other bits in this register. If low, system ROM is located in the space. This bit affects the memory status lines MS0 and MS1 which are decoded by the PLA to generate ROM chip selects. Selecting ROM here drives both memory status lines low when the processor address falls within the specified low space range. This bit resets low to include the C-128 Basic Low ROM.

The next two bits, bits 2 and 3, determine for C-128 the type of memory that will be located in the mid space ($8000 $BFFF). If they are both low, system ROM will be located here. If bit 2 alone is high, internal function ROM is located here. External function ROM appears for bit 3 being alone high, and RAM appears, along with the proper CAS generation, for both bits set high. These bits also affect the hardware memory access lines. When in the aforementioned mid block address range, MS0 reflects the status of bit 3, and MS1 reflects the status of bit 2. These bits both reset low to start out with Basic high.

Bits 4 and 5 determine the contents of the high block ($C000 - $FFFF) for C-128 mode, and have no effect on C64 mode. As with the mid space, both bits zero set up system ROM, bit 4 high sets up internal function ROM., bit 5 high sets up external function ROM, and both bits high set up RAM. Note that the I/O configuration bit, when set for I/O space, leaves the area from $D000 to $DFFF as I/O space regardless of the values of these bits. If not set for I/O space, $D000 to $DFFF contain the character ROM; thus, there is always a hole in high ROM. As

with the other ROM selection bits, these bits are reflected by the memory status lines when this region of address is accessed. Bit 5 corresponds to MS0 and bit 4 to MS1. Both of these bits reset to low to permit Kernal and Character ROM to locate this address space.

Note that there is always a hole in high ROM during C128 mode for the MMU registers at FF00 to FF04. This hole is brought about by holding both MS lines high and both CAS enable lines high.

Finally, bits 6 and 7 control the RAM bank selection. Their actionis dependent, though, upon the version of the MMU. For the current, 128K system, only bit 6 is actually significant. Bit 6 low will select bank zero by dropping CAS0; bit 6 will select bank one by dropping CAS1. Bit 7 in this configuraation does not do anything. Note that for a RAM share status that is non-zero will override the normal CAS enable generation to provide CAS0 for all shared memory. Also, for any area of memory that does not have its ROM bank bits set for RAM access, both CAS enable lines will be high. For any access to the MMU registers from FF00 to FF04, both CAS enable lines and both MS lines will be high. Not ethat in C64 mode the bank used follows the same rules as in C128 mode, thoough banks cannot be changed once in C64 mode. The 256K bonding option replaces the two CAS lines with translated address lines TA16 and TA17. In this configuration, bit 6 becomes TA16 and bit 7 becomes TA17. Thus, when both bits are low, RAM bank 0 is selected. When bit 6 is high, RAM bank 1 is selected. Bit 7 high and 6 low selects RAM bank 2, and both high selects RAM bank 3. At the present no simple way has been defined to turn off RAM selection for holes like the FF00 to FF04 MMU registers with this method of selection as there was with CAS steering for the 128K MMU. Regardless of the MMU version, the reset configuration of both of these is zero, selecting ROM bank 0. The schemes of extended addressing used here are also used for Page Zero and Page One offset and for VIC bank selection.

6.4.2 THE PRECONFIGURATION MECHANISM

The preconfiguration mechanism is a feature of the MMU that allows the Configuration Register to be loaded with one of several memory configurations with a minimum of time and memory on part of the user. The scheme makes use of two sets of registers, the PreConfiguration Registers and the Load Configuration Registers.

The PreConfiguration registers (PCRA - PCRD) are used to store several different memory configurations that may be changed between with a single store instruction. The format of each PCR is the same as for the CR, but when a value is stored to a PCR, no immediate effect takes place. They occupy I/O space from $D501 to $D504. These registers always reset to all zeros.

Load Configuration Registers (LCRA - LCRD) directly correspond with the PCR on a one to one basis. A write to an LCR causes the contents of the corresponding PCR to be transferred to the CR. A read of any LCR returns the value of its corresponding PCR. LCR's are located in system space from $FF01 to $FF04. Neither the LCR's nor the PCR's have any effect in C64 mode. These registers reset to all zeros. Note that these and the configuration register at $FF00 will always be available, completely independent of the ROM, RAM, or bank configuration defined for HI ROM space; any address in this range causes the MMU to force both memory status lines and both CAS enable lines high.

6.4.3 THE MODE CONFIGURATION REGISTER

The control of the current system mode is governed by the Mode Configuration Register (MCR). It controls which processor (8500 or Z80) and which operating system mode (C64 or C-128) is currently in

operation, and handles other overhead of the different operating modes. This register is located in the I/O space at $D505.

The first bit, bit 0, controls which processor is in control. It is usually seen inverted as the output signal /Z80EN. When low, it indicates that the processor is the 8502. This is the reset configuration. When high, the Z80 processor takes over, and all accesses to memory from $0000 to $0FFF, is translated to $D000 to $DFFF, where the CP/M ROM BIOS physically exists in ROM. A change to this bit is held in prelatch until a clock transition in order to prevent processor changing in the midst of an instruction execution. For Z-80 BIOS operation the memory status lines MS0 and MS1 reflect system ROM (both low) for accesses from 001F to 0FFF, otherwise they will both be high. RAM can still be banked by the A16 bit, which controls CAS0 and CAS1. Note that in C64 mode the /Z80EN line is forced low regardless of the value of this bit.

Bits 1 and 2 are unused, but are reserved for future expansion as port lines. Currently, they return high if read, and cannot be written to.

Bit 3 is the FSDIR control bit. It is used as an output to control the fast serial disk data direction buffer hardware, and as an input to a sense fast disk enable signal. The MMU pin FSDIR reflects the status of this bit, which is reset to zero. This register bit is implemented as a bidirectional port, similar to a bit of an MOS 6529 port. The value written to the port effects the output polarity and the value that will be read unless the port is driven low externally, at which time it will read low. If the external driving source is removed, the bit will then resume its previous state.

Bits 4 and 5 are the /GAME and /EXROMIN sense bits, respectively. They directly reflect the hardware cartridge control lines /GAME and /EXROM as used in C64 mode. They are not used by C-128 cartridges, so if they are detected in C-128 mode, a C64 cartridge is present and C64 mode should be asserted. These bits are defined to be bidirectional bits which act very much like the bits in a 6529 port, i.e., a value written to the bit effects the output polarity of its corresponding hardware line and the value thatwill be read from the bit unless the port is driven low externally, at which time it will read low. If the external driving source is removed the bit will resume its previous state. The /GAMEIN bit is used as an output to control the banking of color RAM that permits split screen graphics in C128 mode. The /EXROMIN sense bit has no dedicated output function at the present time.

The operating system mode is set by the next bit, bit 6. This bit is cleared to zero upon reset and its presence enables all MMU registers and other C-128 features, as well as asserting the C-128 control line in hardware. Setting this bit removes the MMU from the memory map and sets the system up in C64 mode. Note that the C-128/64 (MS3) hardware line reflects a logical inversion of the level of this bit.

Bit 7 is a read-only bit used to detect the status of the screen mode switch, as presented in hardware to the 40/80 column pin. If this bit is high, the 40/80 column switch is open, if low, the switch is closed. The display mode is set according to a software interpretation of this bit. This bit is also available as an output bit, similar in characteristics to the /GAMEIN and /EXROMIN bits, but its output function is undedicated at this time.

6.4.4 THE RAM CONFIGURATION REGISTER

The RAM Configuration Register sets up the RAM segmenting parameters for both the processor and the block pointer for the VIC

chip. This register is located in the I/O space at $D506.

Bits 0 and 1 function together to determine the size of the RAM to be shared between banks (assuming that sharing is enabled). With common RAM, the RAM bank bits of the configuration register are basically overridden, as the selected bank of RAM is used for the non-common areas, while bank 0 is used for the specified common areas. ROM and I/O block configuration bits, however, are still important, If the value of the bits together is 0, then 1K of RAM is held common. If the value is 1, then 4K, 2, then 8K, 3, then 16K. These bits have no effect in C64 mode, and the reset value of both bits is defined to be zero.

Bits 2 and 3 function to determine how and if RAM is kept common. If both are low, no sharing takes place. If bit 2 is set, the bottom RAM is shared. If bit 3 is set, the top RAM is shared. Both may be set for sharing in top and bottom. The reset configuration sets both of these bits zero, such that no common memory is present. For the 128K MMU, the selection of the common memory is done by forcing CAS0 low and CAS1 high for all common memory accesses. In the 256K MMU, accesses to common memory must be translated to access only bank 0 RAM.

The next two bits, numbers 4 and 5, are not used presently, but are reserved as block pointer bits for addressing up to 1 megabyte of RAM, in 256K blocks. Taking 5 and 4 together as a number, if they are set to zero, block 1 is in effect; if 1 block 2 is in effect; and so on, permitting four blocks of 256K to be selected. The present MMU does nothing with these bits, they always return low when read. A future MMU may take advantage of the extended banking that these bits provide, but to do so, two pins must be freed. Currently, the A4/A5 and A6/A7 inputs can become one A4/A5/A6/A7 input, freeing one pin. The other pin to be freede will be MUX. The MMU bonding scheme will allow for these aformentioned pins to become A18 and A19.

Bits 6 and 7 function together as a RAM block pointer for VIC. For the 128K MMU, bit 7 is ignored, while bit 6 is used to drive CAS0 low when set low or CAS1 low when set high, thus selecting either RAM bank 0 or RAM bank 1. A 256K MMU selects the bank by bringing out bit 6 as TA16 and bit 7 as TA17. Either scheme allows the VIC to be selected independently from the processor bank. When in 2MHz mode the 80-column chip takes over causing the VIC to be diswabled. The disabling is affected by the VIC chip itself holding AEC constantly high, and thus is not directly affected by actions of the MMU. Note that since a VIC cycle is detected by AEC low, that any DMA will put the MMU into VIC configuration, as it too brings AEC low. In future systems with multiple 256K blocks, the VIC chip will be able to access RAM in Block 0 only.

6.4.5 THE PAGE POINTERS

The page pointers are four registers that allow independent relocation of pages zero and one when running under either processor. These are especially useful when running under the 8502 as they help to remove some of the zero page and stack limitations normally associated with the 6502 family processors.

For zero page relocation the MMU provides the Page Zero Pointer High (P0H) and Page Zero Low (P0L) registers. Bits 0 and 3 of the P0H register correspond to translated address TA16 and TA19, respectively, for any page zero access ($0000 - $00FF). In the 128K system, bit 0 controls the generation of CAS0 or CAS1 depending on whether it is low or high, nad bits 1 to 3 are ignored. In a 256K system bits 0 and 1 are directly translated to TA16 and TA17 and bits 2 and 3 are ignored. In a 1 Megabyte extended system, bits 2 and 3 become extended addressing bits TA18 and TA19. The remaining bits will always return zero. These bits

override the RAM bank bits. The ROM block, and the I/O block bits determine which physical page appears as zero page for all zero page accesses. A write to the POH register is stored in prelatch until a write to the P0L register occurs. Bits 0 to 7 of the P0L correspond to Translated Addresses TA8 to TA15 for any zero page access, thus relocating the zero page. Any access to the area that has become the relocated zero page is switched back to the original zero page. A write to this register sets up the zero page transfer, which can occur as soon as the next low clock cycle. Register P0L is located in the I/O space at $D507, while register P0H is located at $D508.

The registers for page one relocation, the Page One Pointer High (P1H) and the Page One Pointer Low (P1L) do for page one essentially what P0H and P0L do for the zero page. The functions and bit correspondences are exactly the same. P1L is located in the I/O space at $D509 and P1H at $D50A. Note that both register pairs are initialized to zero upon reset, forcing true page zero and true page one access.

6.4.6 SYSTEM VERSION REGISTER

The final register discussed here is the System Version Register, which is located at $D50B in the I/O block. This register is a read-only register that returns a code containing the version of the MMU and the size and capability of the system's memory. The lower nybble, bits 0 through 3, contain the version of the MMU in the system. The upper nybble, bits 4 through 7, contains a code relating the number of memory blocks available in the system. This allows software to compensate for any later systems with more memory available, and should make it quite simple for the current C-128 to remain compatible with any software written in the future for an expanded C-128. The initial C128 will read 2 2 here, indicating two 64K blocks are available. A zero in this nybble would indicate sixteen 64K blocks.

Note : For a physical description and MMU pin layout and definitions, absolute maximum ratings, operating conditions refer to the MMU specification.

CHAPTER 7

KERNAL JUMP TABLE

7.1 C/64 MODE AND C128 MODE KERNAL JUMP TABLE

Vectors for C128

HEX ADDRESS	VECTOR NAME	FUNCTION
FF53	boot_call	;boot load program from disk
FF56	phoenix	;call all function card's cold start routines
FF59	lkupla	;search tables for given la
FF5C	lkupsa	;search tables for given sa
FF5F	swapper	;swap to alternate display device (editor)
FF62	dlchr	;init 80-col character ram (editor)
FF65	pfkey	;program function key (editor)
FF68	setbnk	;set bank for load/save/verify
FF6B	getcfg	;convert bank to mmu configuration
FF6E	jsrfar	;JSR to any bank, RTS to calling bank
FF71	jmpfar	;JMP to any bank
FF74	indfet	;LDA (fetvec),Y from any bank
FF77	indsta	;STA (stavec),Y to any bank
FF7A	indcmp	;CMP (cmpvec),Y to any bank
FF7D	primm	;print immediate (always JSR to this routine!)

C64 vectors

HEX ADDRESS	VECTOR NAME	FUNCTION
FF80	.byte 0	;release number of kernal
FF81	cint	;init screen editor & display chips (editor)
FF84	ioinit	;init i/o devices (ports, timers, etc.)
FF87	ramtas	;initialize ram for system
FF8A	restor	;restore vectors to initial system
FF8D	vector	;change vectors for user
FF90	setmsg	;control o.s. messages
FF93	secnd	;send sa after listen
FF96	tksa	;send sa after talk
FF99	memtop	;set/read top of memory
FF9C	membot	;set/read bottom of memory
FF9F	key	;scan keyboard (editor)
FFA2	settmo	;set timeout in ieee ????????? unused ???????????
FFA5	acptr	;handshake serial byte in
FFA8	ciout	;handshake serial byte out
FFAB	untlk	;send untalk out serial
FFAE	unlsn	;send unlisten out serial
FFB1	listn	;send listen out serial
FFB4	talk	;send talk out serial
FFB7	readss	;return i/o status byte
FFBA	setlfs	;set la,fa, sa
FFBD	setnam	;set length and fn adr
FFC0	open	;open logical file
FFC3	close	;close logical file
FFC6	chkin	;open channel in
FFC9	ckout	;open channel out
FFCC	clrch	;close i/o channel
FFCF	basin	;input from channel
FFD2	bsout	;output to channel
FFD5	loadsp	;load from file
FFD8	savesp	;save to file
FFDB	settim	;set internal clock
FFDE	rdtim	;read internal clock

```
FFE1   stop         ;scan stop key
FFE4   getin        ;get char from queue
FFE7   clall        ;close all files
FFEA   clock        ;increment clock
FFED   jscrorg      ;return screen size
FFF0   jplot        ;read/set x,y coord                    (editor)
FFF3   jiobase      ;return i/o base

FFFA   nmi
FFFC   reset
FFFE   irq
```

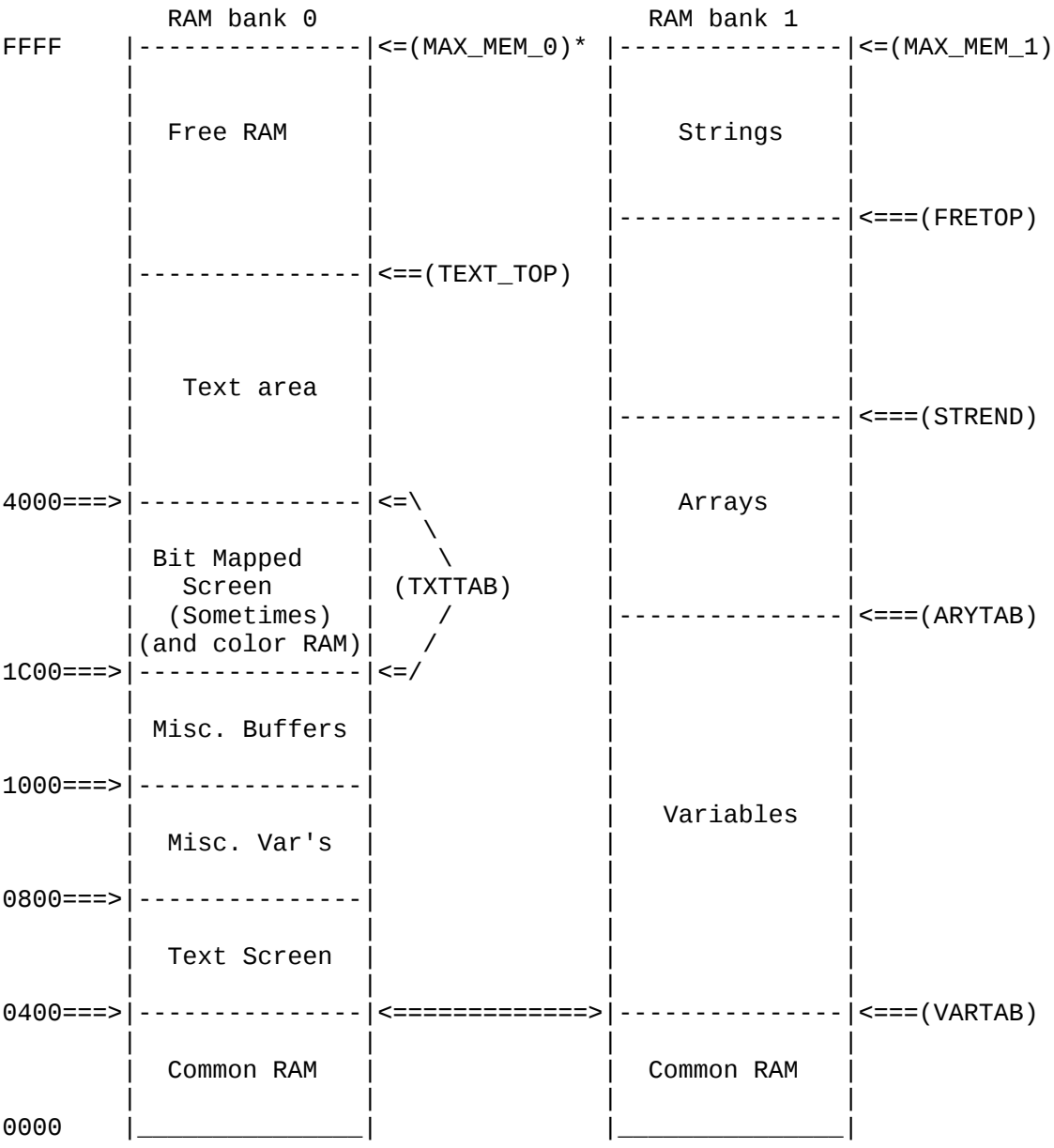
CHAPTER 8

OVERALL DETAILED SYSTEM MEMORY MAP

There are three memory maps shown on the following pages. The maps include: 1.) C128 BASIC MAP, 2.) C128 DISPLAY MAP and 3.) C128 RAM MAP.

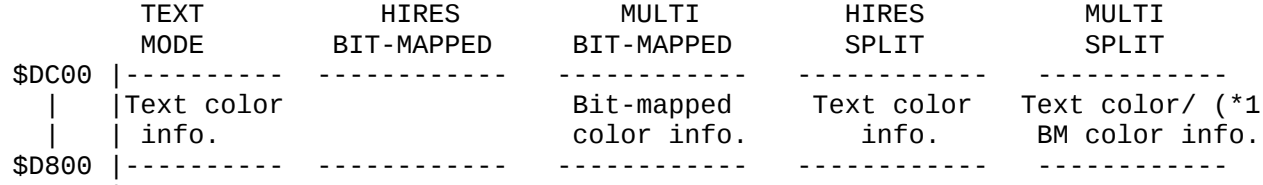
8.1 C128 BASIC MAP

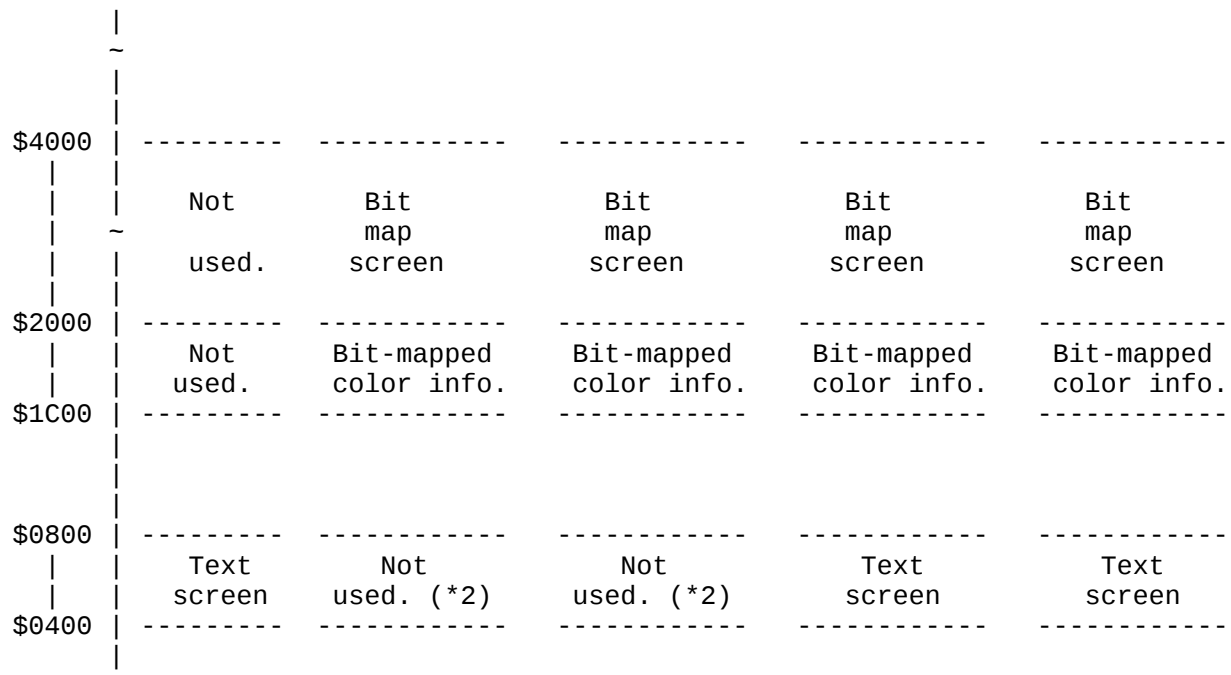
a brief explanation of the pointer structure in basic:



1. * indicates a new pointer.

8.2 C128 DISPLAY MAP





(*1) There are actually 2 banks of RAM that can be mapped into this slot in the map. By selecting one bank during the Bit Mapped portion of the screen (top), and the other during the TEXT portion of the screen (bottom), each mode will have unique RAM for it's own purposes.

(*2) Although the information on the TEXT screen is not actually being displayed at this time, it is still being accessed and updated during any operation normally routed to the screen (such as default print statements, error messages, etc.) "Not used" is NOT meant to imply that during this mode, all print operations are going into the bit-bucket.

3) Text mode requires 1000 character pointer for screen data, and 1000 low bit nybbles for foreground color.

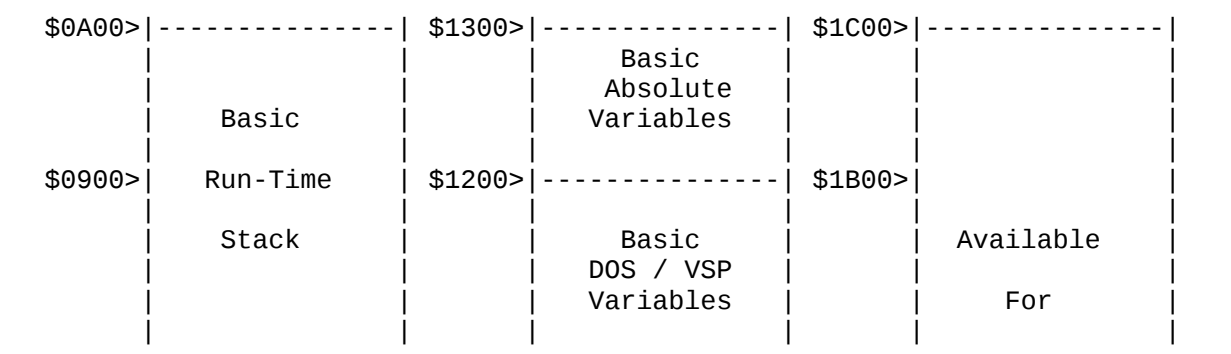
4) Hi-Res mode requires 8000 bytes for the bit map screen and 1000 bytes for foreground and background data.

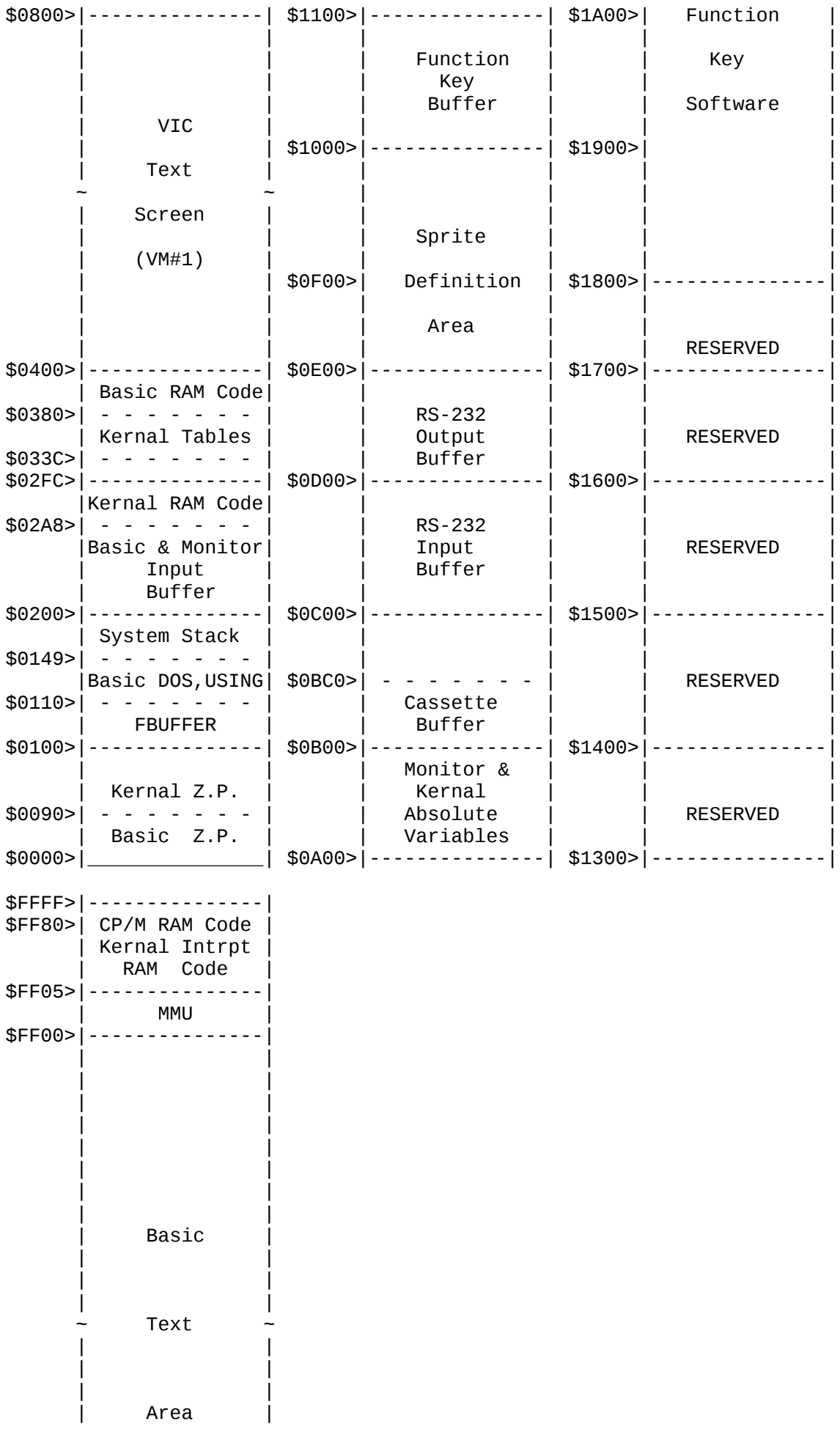
5) Multi-color bit map mode requires 8000 bytes for the bit map screen and 2000 bytes for multicolor data.

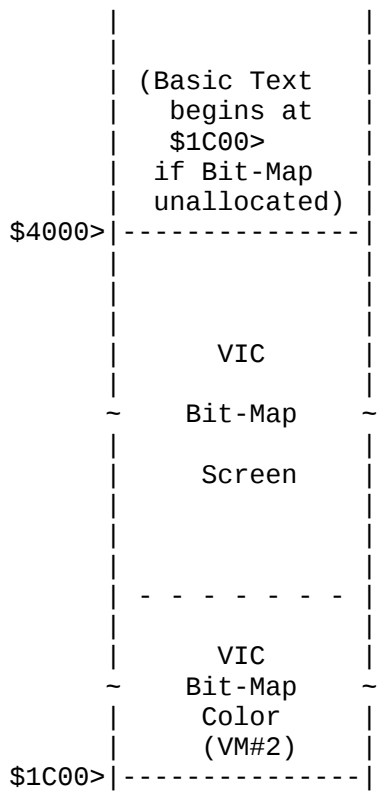
6) Hi-Res split requires the same memory as Hi-Res plus the memory normally used in text mode

7) Multi-split requires the same memory as Multi Color Bit Mapped Mode plus the same amount of memory as text mode. Per note 1 above separate memory must be used to separate text color nybbles from the multi-color mode nybbles.

8.3 C128 RAM MAP




$0800>
VIC
Text
Screen
(VM#1)
$0400>
Basic RAM Code
$0380>
Kernal Tables
$033C>
$02FC>
Kernal RAM Code
$02A8>
Basic & Monitor
Input
Buffer
$0200>
System Stack
$0149>
Basic DOS,USING
$0110>
FBUFFER
$0100>
Kernal Z.P.
$0090>
Basic Z.P.
$0000>
$1100>
Function
Key
Buffer
$1000>
Sprite
Definition
Area
$0F00>
$0E00>
RS-232
Output
Buffer
$0D00>
RS-232
Input
Buffer
$0C00>
$0BC0>
Cassette
Buffer
$0B00>
Monitor &
Kernal
Absolute
Variables
$0A00>
$1A00>
Function
Key
Software
$1900>
$1800>
RESERVED
$1700>
RESERVED
$1600>
RESERVED
$1500>
RESERVED
$1400>
RESERVED
$1300>
$FFFF>
$FF80>
CP/M RAM Code
Kernal Intrpt
RAM Code
$FF05>
MMU
$FF00>
Basic
Text
Area


(Basic Text
begins at
$1C00>
if Bit-Map
unallocated)
$4000>
VIC
Bit-Map
Screen
VIC
Bit-Map
Color
(VM#2)
$1C00>

CHAPTER 9

DETAILS OF SOFTWARE INTERFACE TO 8563 (80 COLUMN CHIP)

9.1 OVERVIEW

The 80-column chip integrates several features that are presently implemented with a 6545E and a large number of TTL packages. The 80-column text display chip is designed to work like a 6545E CRT-Controller. To fully understand the features of the chip, refer to the 8563 DESIGN SPEC (2/10/84) for the pin and register layouts.

The CPU interface (D0-D7, Phase-2,R/W, RS, CS) remains the same as the 6545E. The LPEN, RES, HSYNC and VSYNC pins also remain the same. The display RAM Address lines are extended to 16 bits, and are multiplexed onto 8 pins in an address-low/address-high configuration. Timing signals RAS and CAS are output to drive dynamic RAMs. A CCLOCK output is generated to allow this chip to supply a phase-2 signal for a processor in a shared memory addressing configuration. The DISPLAY ENABLE output pin performs the same function as before but now has programmable location and duration.

A single Display RAM interfaced to the 80-column chip serves as storage for character pointers and character data. An 80-column double line buffer is implemented on-chip for both character pointers and attributes. The chip has an output pixel shift register. An 8 bit Display Data Bus is used to transfer data to and from the Display RAM.

The Display Data Bus has several uses: to transfer CPU data that is passed transparently through the 80-column chip, to transfer character pointers and attributes from the Display RAM to the on-chip line buffers and to transfer character data from the Display RAM to the on-chip shift register.

Four video outputs reflect either the four-bit character foreground attribute, the background register bits or the foreground register bits (if an individual character attribute is disabled). A high speed Dot Clock (DCLK) is used as a clock input.

All the registers of the 6545E are implemented exactly the same on this chip with the exception of Register 8 (Mode Control): Bit 7 is not used as well as bit 6, bit 5 and bit 4. New registers are needed to implement new functions. (These registers can be found in detail in the 8563 Spec, as well as timing diagrams, sequencing of display memory bus accesses, DRAM timing, and AC parameters for the 8563). This spec is not intended to cover those particular areas in detail.

CHAPTER 10

CP/M MODE

10.1 GENERAL SYSTEM LAYOUT

The C-128 is a two processor system with the primary processor being the 8500 and the secondary processor being the Z80. The 8500 has the same instruction set as the 6502. The C-128 powers up running BASIC using the 8500. The Z80's primary function is to run CP/M Plus. This chapter relates to the requirements, methods and solutions for implementing CP/M Plus within the C-128. When CP/M is running, the normal functions of the C128 are not supported (CP/M and BASIC cannot run at the same time). CP/M does not directly support all of the display modes of the VIC chip (an application could be written to run under CP/M that could use the extra graphics capabilities, but the application must keep track of all the details such as memory maps).

There are a number of system parameters that are configurable by running a configuration program. Following is a list of options that are selectable:

1. RS232C XON/XOFF on receiver enabled disabled
2. RS232C XON/XOFF on transmitter enabled disabled
3. Number of disk drives
4. Type of printer(s) connected

10.2 SYSTEM MEMORY ORGANIZATION

The memory map is limited to 64K at any one point in time. However, the RAM bank can be selected and then different ROM areas can overlay the RAM (with bleed-thru on write lines). The actual memory map is controlled by the MMU. The MMU can be accessed in the I/O area or through the load configuration registers located at FF00 through FF04.

If the load configuration registers are read then the current value is read. A write to FF00 changes the configuration after completing the current instruction. A write to FF01 to FF04 updates the current configuration to the value stored in the preconfiguration registers (the data written is not used). The MMU page pointers have both a low (page) and a high (page) pointer, the high is written first and latched in the MMU, the high value is updated from the latch when the low byte is written. The MMU control registers are as follows:

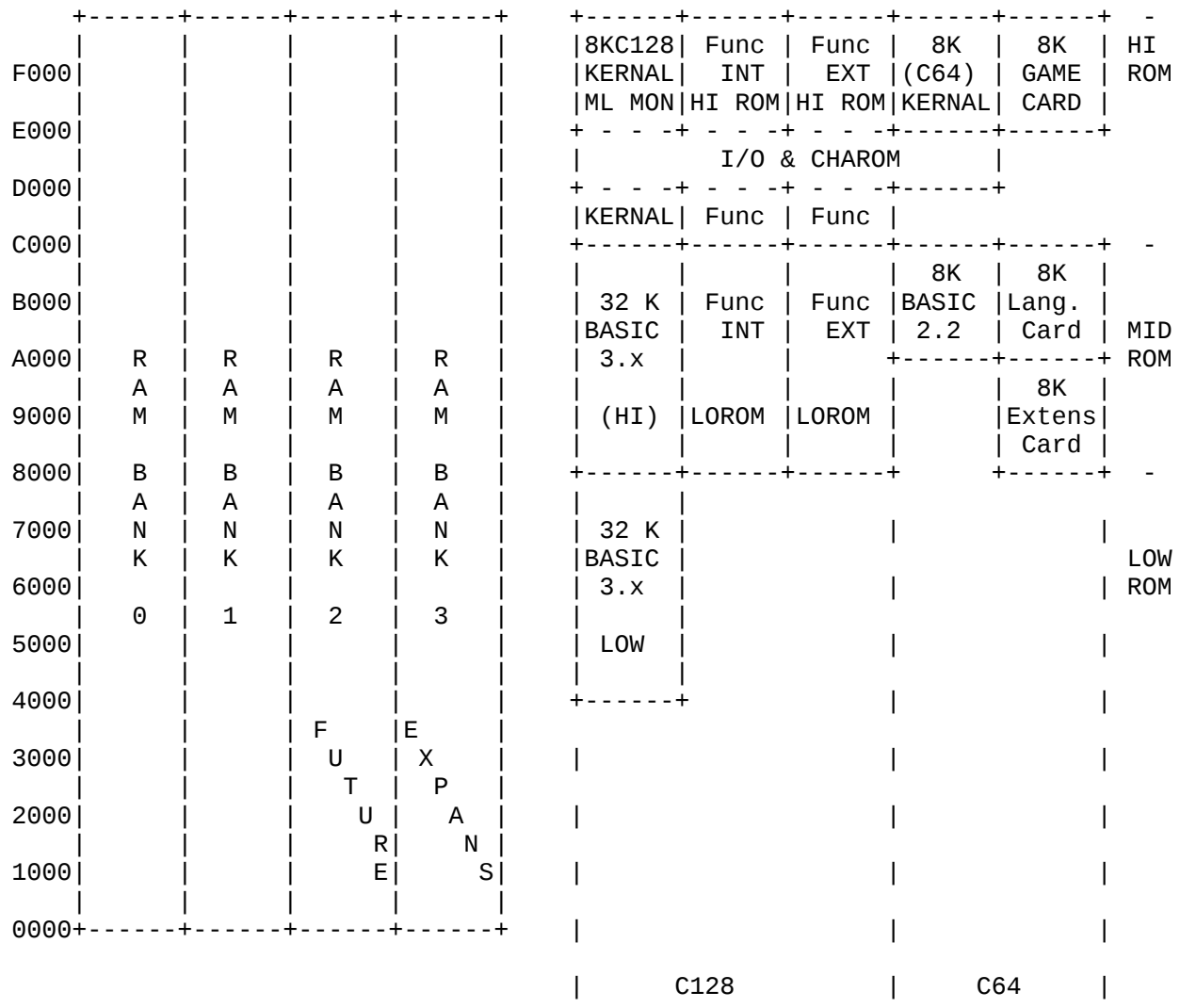
D500 (FF00)	Configuration Register	7	A17 RAM Bank -	reserved
D501 (FF01)	Preconfig Register A	6	A16 RAM Bank -	0 - 1
D502 (FF02)	Preconfig Register B	5	ROM HI Block -	00-System ROM
D503 (FF03	Preconfig Register C	4	ROM HI Block -	01-INT Func ROM
D504 (FF04)	Preconfig Register D	3	ROM MID Block -	10-EXT Func ROM
		2	ROM MID Block -	11-RAM
		1	ROM LOW Block -	0-System ROM, 1-RAM
		0	I/O Block	0-System I/O, 1-HI ROM
D505	Mode Configuration Register	7	40/80 Sense	
		6	OS Mode	0-C128, 1-C64
		5	/EXROM Sense	
		4	/GAME Sense	
		3	Fast Serial Port	0-Data IN, 1-Data OUT
		2	(reserved)	
		1	(reserved)	

		0	Proc Mode 0-8500, 1-Z80
D506	RAM Configuration Register	7	VA17 - VIC RAM BANK
		6	VA16 -
		5	RAM Block - 00 0-256K, 01 256K-512K *
		4	RAM Block - 10 512-768K, * 01 768K-1M *
		3	Shared RAM Top 0-No, 1-Yes
		2	Shared RAM Bot 0-No, 1-Yes
		1	Shared RAM Size 00-1K,01-4K
		0	Shared RAM Size 10-8K,11-16K
D507	Page 0 Pointer Low		High Pointer
D508	Page 0 Pointer High	xxx	xxx xxx xxx ??? ??? A17 A16
D509	Page 1 Pointer Low		Low Pointer
D50A	Page 1 Pointer High	A15	A14 A13 A12 A11 A10 A09 A08

* denotes not in C128 implementation - reserved for future use

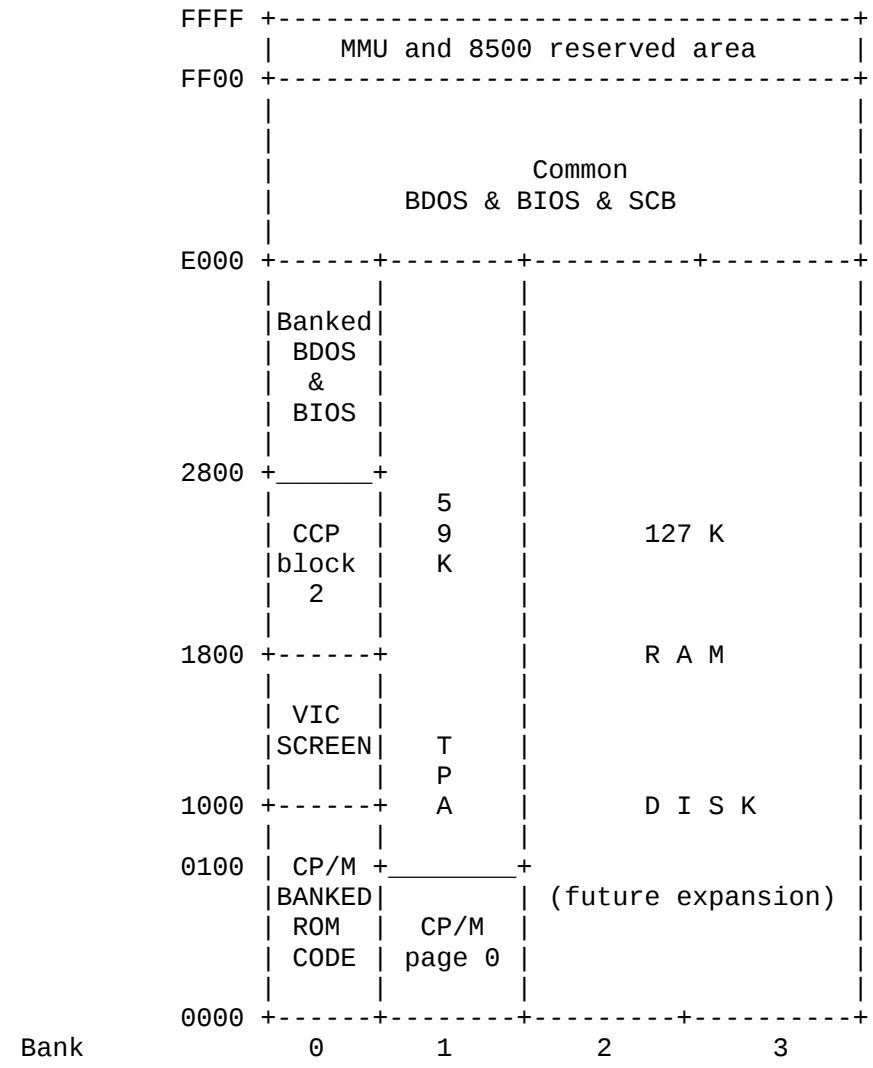
10.2.1 COMMON MEMORY MAP

COMMON MEMORY MAP



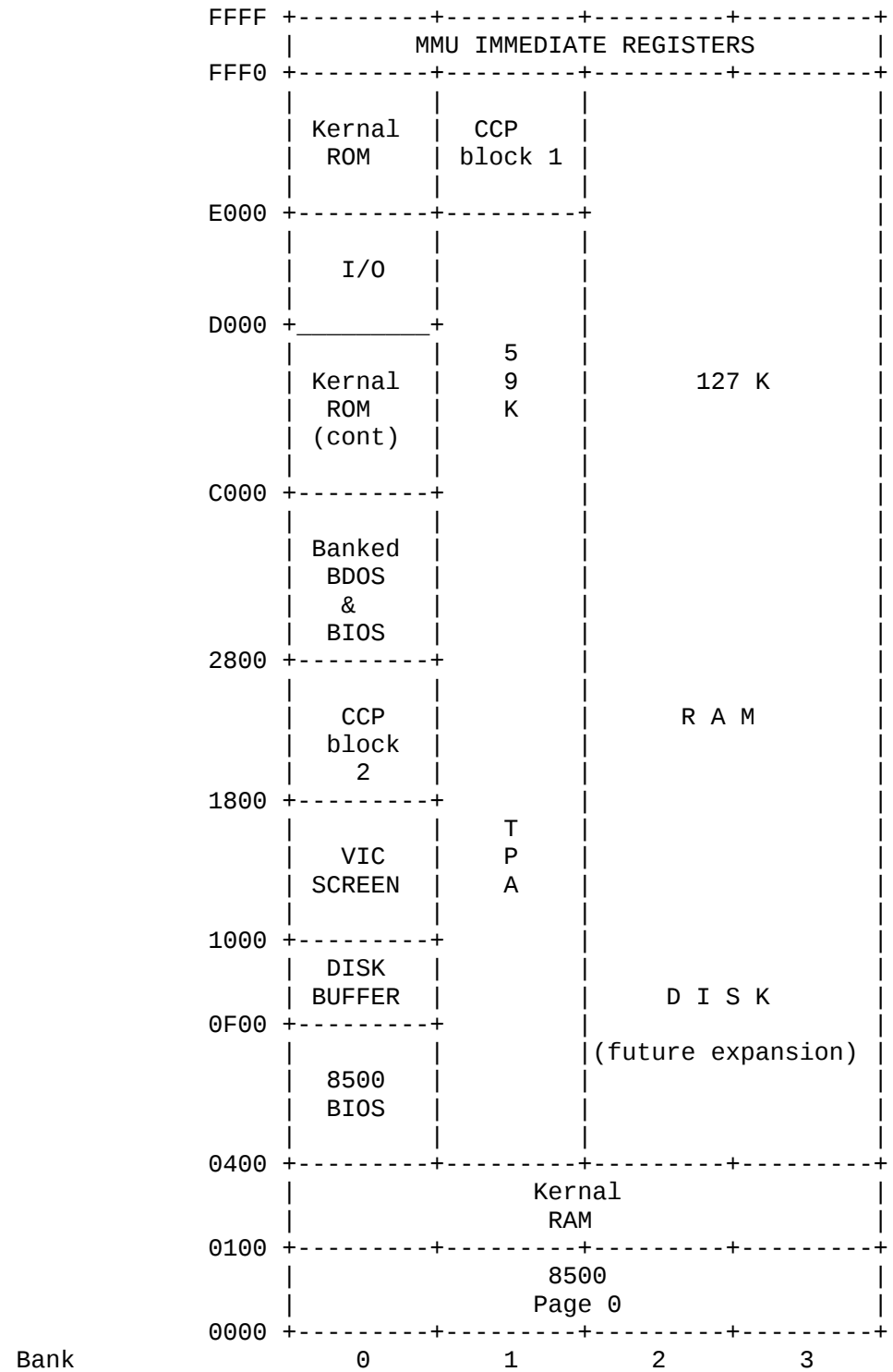
10.2.2 Z80 MEMORY MAP

Z80 Memory Map



10.2.3 8500 MEMORY MAP

8500 Memory Map



10.3 1571 DISK ORGANIZATION

CP/M Plus supports many different diskette formats. The first being compatible with CP/M 2.2 that runs on the C64. With this format the File Control Block (FCB) is set up as 32 tracks of 17 sectors each and a track offset of two. The BIOS routine adds a 1 to tracks greater than 18 (this is the C64 directory track). The second format is new and takes advantage of the full disk capacity. This is done by setting up the FCB with 638 tracks of one sector each and a track offset of one (1 sector). This has the effect of having CP/M set the track to the block number relative to the beginning of the disk with the sector always set to zero. The following algorithm is used to convert the requested TRACK to a real track and sector number.

The remaining formats require that the user have the new 1571 disk drive. This disk drive supports both single and double sided diskettes and both the Commodore GCR and industry standard MFM data coding formats. The third GCR format is double sided. The disk is treated as 1276 sectors of data with a track offset of one. Side one is used first then side two is used.

Note: This is not the normal way to handle a two sided disk but by allocating the disk in this manner the user with a 1541 may still be able to read data written at the start of a two sided disk.

```
Effective Track   (1-35) = INT(Function)
Effective Sector  (0-20) = MOD(Function)
```

Block Number	Function	Region
000 >= TRACK > 357	((TRACK-000-0)/21)+01	1
357 > TRACK > 490	((TRACK-357-1)/19)+18	2
490 > TRACK > 598	((TRACK-490-1)/18)+25	3
598 > TRACK > 683	((TRACK-598-1)/17)+31	4

The effective sector is then translated to provide a skew that speeds up operations. The skew is used only with the new larger format. A different skew table is used for each region of the disk.

10.3.1 C64 CP/M DISK FORMAT

C/64 CP/M Disk Format

This format is provided to maintain compatibility with the C64 CP/M 2.2 system currently on the market. Notice all of the unused space.

	Sector																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	x	x	x	x
1	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	x	x	x	x
2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
7	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
9	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
10	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
11	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
12	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
13	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
14	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
16	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x
17	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D		
18	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
19	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
20	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
21	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
22	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
23	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x		
24	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
26	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
27	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
28	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
29	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x			
30	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
32	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
33	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
34	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				

. Used by CP/M
B Boot Sector (System)
D Directory Sector
x Not used by CP/M

10.3.2 C-128 CP/M DISK FORMAT

C128 CP/M Plus Disk Format

	Sector																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	B	B	B	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
12	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
13	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
14	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
16	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
17	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
18	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
19	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
20	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
21	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
22	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
23	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
24	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
26	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
27	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
28	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
29	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			
30	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
32	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
33	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				
34	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.				

. Used by CP/M , reg. = region(1-4)
B Boot Sector (System)

SKEW TABLE

reg.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	00	17	13	09	05	01	18	14	10	06	02	19	15	11	07	03	20	16	12	08	04
2	00	04	08	12	16	01	05	09	13	17	02	06	10	14	18	03	07	11	15		
3	00	11	04	15	08	01	12	05	16	09	02	13	06	17	10	03	14	07			
4	00	07	14	04	11	01	08	15	05	12	02	09	16	06	13	03	10				

10.4 MFM DISK FORMATS

The following disk formats will be built in and selectable from the keyboard: OSBORNE, KAYPRO, EPSON, and IBM CP/M-86. Other formats will be available and selected by running a configuration program. The built in formats will be selectable from the keyboard by using a special key sequence.

The IBM CP/M-86 capability is provided so that data can be transfered between machines. We cannot run CP/M-86 programs on the C128 (CP/M Plus is running on a Z80 not an 8088).

```
OSBORNE (single sided only)
KAYPRO
       Single sided
       Double sided
EPSON
IBM CP/M-86
     four different formats
       Single sided 8 sectors/track (SS-8)
       Single sided 9 sectors/track (SS-9)
       Double sided 8 sectors/track (DS-8)
       Double sided 9 sectors/track (DS-9)
       SS-8 (IBM)
            40 tracks/side   1 side
             8 sectors 512 bytes sectors/track
             1 track offset
             1K allocation size
             64 directory entries (2 allocation units)
       SS-9 (IBM)
             40 tracks/side   1 side
             9 sectors 512 bytes sectors/track
             1 track offset
             1K allocation size
             64 directory entries (2 allocation units)
       DS-8 (IBM)
             40 tracks/side   2 sides
             8 sectors 512 bytes sectors/track
             1 track offset
             2K allocation size
             64 directory entries (2 allocation units)
       DS-9 (IBM)
             40 tracks/side   2 sides
             9 sectors 512 bytes sectors/track
             1 track offset
             2K allocation size
             64 directory entries (2 allocation units)
```

10.5 MEMORY DISK ORGANIZATION

The Memory Disk will only be available in systems that support 256K of DRAM. System utilities allow the user to generate a system with or without the RAM disk. The RAM disk file control block (FCB) is set up as two tracks each with 508 sectors of 128 bytes. The track then becomes the bank and the sector is an offset.

This Memory Disk should NEVER use any LRU blocks (CP/M uses LRU's to buffer disk sector, the method used in maintaining these buffers is called Least Recently Used (LRU). This wastes memory by duplicating copies of sectors in memory. This is also the reason for using 128 byte sectors. Larger sectors require blocking/deblocking LRU block.

NOTE: This is not part of C128, maybe future C256, also possible external DMA RAM disk.

10.6 CODE ORGANIZATION OVERVIEW

The 8500 and Z80 share many of the low level FUNCTIONS for CP/M Plus. This is done to optimize the speed and code requirements. The following functions are performed and are defined within the following sections.

1. Booting of CP/M Plus
2. Keyboard scanning
3. Update 40/80 column display
4. Printer interface on serial bus
5. Get a char from RS232C adapter (with XON/XOFF)
6. Send a char to RS232C adapter
7. Read a sector of data from disk (fast and slow)
8. Write a sector of data to disk (fast and slow)
9. Read a sector of data from the RAM disk
10. Write a sector of data to the RAM disk
11. Format disk
12. Copy CCP to Hidden RAM from TPA 100H
13. Copy CCP to TPA 100H from Hidden RAM
14. Set system time
15. Update system time
16. Memory to memory move

10.6.1 BLOCK TRANSFER OPERATIONS

There are three different types of block operations that must be performed. These are Disk support, RAM Disk support, and moving the CCP to and from a holding buffer.

The Z80 has a routine that can move data from any address and bank to any other address and bank (non-overlapping). The routine relies on the fact that the MMU page 0 and page 1 can be pointed to any page boundary within the system. This routine uses th LDIR instruction but must break down the transfer into moves of 256 bytes or less. This routine is a major routine in the development of the CP/M Plus system. All other routines use this routine to do block moves between banks.

There are two types of disk drives for the C128. The Commodore 1541, a slow disk drive and the Commodore 1571, a new high speed disk drive that normally is sold with the C128. The BIOS85 module determines which type of drive that is connected and calls the appropriate KERNAL routines. All data transfers done by the BIOS85 are done to a 256 byte buffer. The Z80 is then responsible for moving data to the proper memory location (bank and address).

10.6.1.1 BOOTING OF CP/M PLUS -

The autoboot method used with the C128 is to read the first sector on the disk, check that it is a boot sector (first three bytes are CBM). If it is a boot sector it is loaded into memory at $1000 and executed. The CP/M boot sector will meet the above requirements and in turn load the 8500 BIOS. Control is then given to the 8500 BIOS. The 8500 BIOS sets up the Z80 memory map and transfers control to the Z80 (at location 0000 bank 0). The CP/M ROM gets control and loads the CP/M system from disk (using 8500 BIOS to do disk functions). After CP/M is loaded, control is transferred to CP/M cold boot. The cold boot routine then reads CCP.COM to 100H in bank 1. A copy of the CCP.COM file is then transferred to banked memory so that warm boots do not have to load CCP.COM from disk, but instead transfers the CCP from banked memory. With the CCP in the TPA program control is transferred to it at location 100H in bank 1.

The structure of the boot sector is defined and is presented below. The boot sector has the following format.

1/ read disk block at track 1 sector 0 into ram at $0b00 ('tbuffr').
2/ check for auto-boot signature. RTS if not. the relative format is:

```
$00 $01 $02 $03  $04  $05  $06 ----------  A  ----------  B  -----> $FF
 |   |   |   |    |    |    |  (optional)  |  (optional)  |
 C   B   M  adrl adrh bank #blk diskname   0   filename   0  6502 code
```

3/ IF #blk > 0 THEN block_read sequential sectors into ram at given adrl, adrh, bank location.
4/ IF filename THEN load filename into ram bank0 (normal load).
5/ JSR to user code following filename (system map).

10.6.1.2 READ A SECTOR OF DATA FROM DISK (FAST AND SLOW) -

Prior to calling this routine CP/M sets the track, sector, DMA address, DMA bank and the number of sectors. CP/M sets the track and sector using the data in the DPB. This means that for the new disk format the sector is always zero and the track is the block number relative to the start of the disk. The Z80 BIOS translates this to the proper track and sector number for the disk drive (sector translation also). If the disk is the older type (C64 CP/M) then the track and sector numbers set by CP/M do not need to be converted. If the disk drive is the new high speed drive, then the BIOS80 determines the number of consecutive sectors on the same track. The BIOS85 routine is called to set up the multiple sector transfer. Control is then returned to BIOS80 and the Z80 then reads the disk data from the I/O buffer directly, and writes it to system memory (at the DMA address). If the disk drive is the older and slower 1541, then the BIOS85 reads one sector at a time to a 256 byte sector buffer and the BIOS80 moves the data to the correct location in memory (DMA address). The BIOS85 routines are responsible for doing retries and reporting hard errors and relogging of the disk to the Z80.

10.6.1.3 WRITE A SECTOR OF DATA TO DISK (FAST AND SLOW) -

Prior to calling this routine CP/M sets the track, sector, DMA address, DMA bank and sector count. CP/M sets the track and sector using the data in the DPB. This means that for the new disk format the sector is always zero and the track is the block number relative to the start of the disk. The Z80 BIOS translates this to the proper track and sector number for the disk drive (sector translation also).If the disk drive is the new high speed drive, then the BIOS80 counts the number of consecutive sectors on the same track and passes this info to the BIOS85. The BIOS85 routine then sets up the transfer of multiple sectors and returns to the BIOS80. The BIOS80 then reads the data from the current DMA address and writes this data directly to the I/O port. If the disk is the older type, then BIOS80 writes one sector at a time by moving the DMA buffer contents to the 256 byte disk read/write buffer. The BIOS85 is then called and the buffer is written to disk.The 8500 BIOS also checks whether a disk has been changed and returns the appropriate error code if it has.

10.6.1.4 READ A SECTOR OF DATA FROM RAM DISK -

Copies 128 bytes of data from the RAM disk addressed by the track and sector to the current DMA bank and address. This function is performed totally by the Z80.

10.6.1.5 WRITE A SECTOR OF DATA TO RAM DISK -

Copies 128 bytes of data from the current DMA address and bank to the RAM disk addressed by the track and sector. This function is performed totally by the Z80.

10.6.1.6 COPY CCP TO HIDDEN RAM FROM TPA 100H -

This function is performed only when the system is first booted. The CCP.COM file is loaded from disk into the TPA. Normally the CCP.COM file would be read from disk each time the system is warm started (going back to the A> prompt). The reloading of the CCP is done so often, which slows down the system (by doing a lot of disk accessing), that it makes it worth keeping a copy of the CCP in banked memory. This routine is responsible for saving a copy of the CCP.COM file the first time it is loaded. This copy of the CCP is then copied to 100H in bank 1 (the TPA area) when the system is warm started. This function is performed

totally by the Z80.

10.6.1.7 COPY CCP TO TPA 100H FROM HIDDEN RAM -

This function is called when the system performs a warm boot. See section 10.5.1.6 for more details on how and why the function is used. The CCP is copied from its RAM buffer to 100H in Bank 1. This function is performed totally by the Z80.

10.6.1.8 FORMAT DISK -

This function reformats the disk. The fill character is normally E5's for CP/M systems but this is only required for the directory sectors (first track). This function should only be used with a format program that can create a system disk. The format program is able to create two different disk formats. The first being the old C64 CP/M format. There is an option of copying the C64 system tracks from another disk or just leave the system tracks blank (the directory track (18) is also set up). The second format is the new C-128 CP/M disk format. This version creates the boot sectors on the new disk, formats the directory area and gives the user the option of copying the CPM.SYS and/or the CCP.COM files to the newly formatted disk.

10.6.2 CHARACTER TRANSFER OPERATIONS

The character transfer operations are handled mostly by the 8500. One major exception to this is the 40/80 column display system which is handled by the Z80.

10.6.2.1 KEYBOARD SCANNING -

The keyboard scan routine that is called to get a keyboard character returns the key code of the pressed key of FFh if no key is currently being pressed. The keyboard scan code is also responsible for handling programmable keys, programmable function keys, setting character and background colors, selecting MFM disk formats, and selecting current screen emulation type.

Any key on the keyboard can be defined to generate a code or function except the following keys:

Left Shift (and lock key)
Right Shift
Commodore Key
Control Key
Restore (8502 NMI)
80/40 Display (used to select key)
Caps lock Key

The keyboard recognizes the following special functions:

Cursor left key - define key
Cursor right key - define string (points to function keys)
ALT Key - toggle key filter

Define key:

This function allows the user to define the code that a key can produce. Each key has four definitions; Normal, alpha shift, shift, and control. The alpha shift is toggled on/off by pressing the Commodore Key. After entering this mode a small white box will appear on the bottom of the screen. The first key that is pressed is the key to be defined. The current HEX value defined to this key is displayed and the user can then type the new hex code for the key or abort by typing a non-hex key. The following is a definition of the codes that can be assigned to a key. (In ALT mode, codes are returned to the application. See ALT. mode)

```
00h         Null (same as not pressing a key)
01h-7Fh     normal ASCII codes
80h-9Fh     string assigned
A0h-AFh     80 column character color
B0h-BFh     40 column backgroung color
C0h-CFh     40 column character color
D0h-DFh     40 colomn background color
E0h-EFh     40 column border color
F0h-FFh     special functions

  F0h       toggle enable/disable disk status indicator
  F1h       XON/XOFF display (pause)
  F2h       (undefined)
  F3h       40 column screen window right
  F4h       40 column screen window left
  F5        undefined
   .           .
  FFh       undefined
```

Define string:

This function allows the user to assign more than one key code to a single key. Any key that is typed in this mode is placed in the string. The user can see the results of typing in a long box at the bottom of the screen. NOTE: some keys may not display what they are. To allow the user control of entering data, five special functions are available. They are all control right shift functions (this allows the user to enter any key into the buffer).

EDIT string functions: (CTRL RT. SHIFT)

```
RETURN          -  done defining a string
main '+'        -  insert a space into string
main '-'        -  delete cursor character
lf arrow        -  cursor left
rt arrow        -  cursor right
```

ALT mode:

This function is a toggle (on/off) and is provided to allow the user to send 8 bit codes to an application without the keyboard driver eating the code from 80h to FFh.

10.6.2.2 UPDATE 40/80 COLUMN DISPLAY -

There are two totally different display systems within the C-128. The first is the VIC chip that produces a 25 line by 40 column display and has many graphics modes of operation and can be used with a standard color TV or color monitor. The only mode of operation that is being used by CP/M is normal character mode with each character and screen background having up to 16 colors. The second display system will only work with a monitor (black and white or color), called the 8563, and is structured after the 6545 display controller. The display format of this controller is 24 lines by 80 columns with character color attributes. The VIC chip is a memory mapped display and the 6845E is I/O controlled. The two display subsystems are being treated as two totally separate displays. CP/M Plus can assign them both or only one to the console output device. Both displays are controlled by a common terminal emulation package (probably a ADM-3A driver). The display subsystems are controlled totally by the Z80 BIOS. The emulations software is written in such a way so that different terminal emulators can be substituted for the ADM-3A emulation by the user.

The terminal driver is divided into two different parts: terminal emulation and terminal functions. A number of different emulations may be provided. A few of the options are the ADM-3A, ADM-31, H19, VT52,

and VT100. The terminal emulation is part of the Z80 BIOS and the terminal function part is primarily in the Z80 ROM.

10.6.2.3 TERMINAL EMULATION PROTOCOLS -

10.6.2.3.1 LEAR SIEGLER ADM-3A -

LEAR SIEGLER ADM-3A

	Character Sequence	HEX Char Code
Position cursor	<ESC>=(row#+32)(col#+32)	1B 3D 20+ 20+
Cursor Left	^H	08
Cursor Right	^L	0C
Cursor Down	^J	0A
Cursor Up	^K	0B
Home & Clear Screen	^Z	1A
Carriage Return	^M	0D
Escape	^[	1B
Bell	^G	07

NOTE: display is 24 (1-24) by 80 (1-80), cursor origin is always 1/1

10.6.2.3.2 LEAR SIEGLER ADM-31 -

LEAR SIEGLER ADM-31

		Characater Sequence	HEX Char Codes
	Clear to End of Line	<ESC>T	1B 54
		<ESC>t	1B 74
	Clear to End of Screen	<ESC>Y	1B 59
		<ESC>y	1B 79
	Home & Clear Screen	<ESC>:	1B 3A
		<ESC>*	1B 2A
	Half Intensity On	<ESC>)	1B 29
	Half Intensity Off	<ESC>(	1B 28
	Reverse Video On	<ESC>G4	1B 47 34
	Blinking On	<ESC>G2	1B 47 32
*	Underline On	<ESC>G3	1B 47 33
*	Select Alt Char Set	<ESC>G1	1B 47 31
	Rev. Video & Blinking Off	<ESC>G0	1B 47 30
	Insert Line	<ESC>E	1B 45
	Insert Character	<ESC>Q	1B 51
	Delete Line	<ESC>R	1B 52
	Delete Character	<ESC>W	1B 57
*	Set screen colors	<ESC ESC ESC> color #	
		where color # = 20h to 2Fh - character color	
		30h to 3Fh - backgrd color	
		40h to 4Fh - border color (40 col only)	

* NOTE: This is not a normal ADM31 sequence.

Note: display is 24 (1-24) by 80 (1-80), cursor origin is always 1/1

10.6.2.3.3 VT52 -

VT52

Cursor Up	<ESC>A
Cursor Down	<ESC>B
Cursor Right	<ESC>C
Cursor Left	<ESC>D
Enter Graphics Mode	<ESC>F
Exit Graphics Mode	<ESC>G

Cursor Home	<ESC>H
Reverse Line Feed	<ESC>I
Erase to End of Screen	<ESC>J
Erase to End of Line	<ESC>K
Cursor Addressing	<ESC>Y (row#+1Fh)(col#+1Fh)
Identify	<ESC>Z
Enter Alt. Keypad Mode	<ESC>=
Exit Alt. Keypad Mode	<ESC>>
Enter VT100 Mode	<ESC><

NOTE: display is 24 (1-24) by 80 (1-80), cursor origin is always 1/1

10.6.2.3.4 VT100 -

VT100

A subset of the ANSI X3.64-1979 and X3.41-1974 standards

Video Attributes <ESC>[(Ps);...;(Ps)m

where (Ps) is:

- 0 Normal (all att. off)
- 1 Bold
- 4 Underscore
- 5 Blink
- 7 Reverse

Character Sets	G0 Set	G1 Set
UK	<ESC>(A	<ESC>)A
ASCII	<ESC>(B	<ESC>)B
Special Graphics	<ESC>(0	<ESC>)0
Alt. ROM	<ESC>(1	<ESC>)1
Alt. ROM Special Graphics	<ESC>(2	<ESC>)2

Character Size

Double High Top Half	<ESC>#3
Double High Bottom Half	<ESC>#4
Single Width Single Height	<ESC>#5
Double Width Single Height	<ESC>#6

Cursor Movements

Position Cursor	<ESC>[(row#dec);(col#dec)H	1B xx 30+ xx 30+ 48
	<ESC>[(row#dec);(col#dec)f	1B xx 30+ xx 30+ 66
Cursor Left	<ESC>[(#)D	1B xx 30+ 44
Cursor Right	<ESC>[(#)C	1B xx 30+ 43
Cursor Down	<ESC>[(#)B	1B xx 30+ 42
Cursor Up	<ESC>[(#)A	1B xx 30+ 41
Index	<ESC>D	1B 44
New Line	<ESC>E	1B 45
Reverse Index	<ESC>M	1B 4D
Save Cursor & Attrib.	<ESC>7	
Restore Cursor & Att.	<ESC>8	

Erase

Clear to End of Line	<ESC>[K	1B xx xx
Clear to End of Screen	<ESC>[J	1B xx xx
Clear from Start of Line	<ESC>[1K	1B xx 31 xx
Clear from Start of Scrn	<ESC>[1J	1B xx 31 xx
Clear cursor line	<ESC>[2K	1B xx 32 xx
Clear Entire Screen	<ESC>[2J	1B xx 32 xx

LED's <ESC>[(Ps);...;(Ps)q

- 0 Extinguish All
- 1 L1 on
- 2 L2 on
- 3 L3 on
- 4 L4 on

```
Media Copy                      <ESC>[(Ps)h
Modes
     Enable                     <ESC>[(Ps)h
     Disable                    <ESC>[(Ps)l
        where (Ps) is:
           20   New Line            Line Feed
           ?1   Application         Cursor      (cursor key mode)
           ?2   n/a                 VT52 mode
           ?3   132 Column          80 Column
           ?4   Smooth Scrolling    Jump Scrolling
           ?5   Screen Rev Video    Screen Normal Video
           ?6   Screen Origin
                      Relative      Screen Origin Absolute
           ?7   Line Wrap On        Line Wrap Off
           ?8   Auto Repeat On      Auto Repeat Off
           ?9   Interlace On        Interlace Off

      Keypad Applications Mode      <ESC>=
      Keypad Numeric Mode           <ESC>>

 Reports
      Cursor Position
            -invoked by             <ESC>[6n
            -response               <ESC>[(Pl);(Pc)R
      Status Report
            -invoked by             <ESC>[5
            -response               <ESC>[0n  (terminal OK)
                                    <ESC>[3n  (terminal not OK)
      Terminal Parameters
            -invoked by             <ESC>
            -response               <ESC>

      What are you
            -invoked by             <ESC>[c or ( <ESC>Z )
            -response               <ESC>[?];(Ps)c
               where (Ps) is
                   0  Base VT100, no options
                   1
                   2
                   3
                   4
                   5
                   6
                   7

 Reset to Initial State             <ESC>c
 Scrolling Region                   <ESC>[(Pt);(Pb)r
      where       (Pt) is top line #
                  (Pb) is bottom line #
 Tabs
      Set tab                        <ESC>H
      Clear tab                      <ESC>g
      Clear all tabs                 <ESC>3g
 Test
      Fill screen with E's           <ESC>#8
 Null                                                    00
 Bell                            ^G                      07
 HT                              ^I                      09
 LF                              ^J                      0A
 VT (same as LF)                 ^K                      0B
 FF (same as LF)                 ^L                      0C
 CR                              ^M                      0D
 SO (select G1 char set)         ^N                      0E
```

SI (select G0 char set)	^O	0F
CAN (terminate escape seq)	^X	
SUB (same as CAN)	^Z	

Note: Display is 24(1-24) by 80(1-80) or 24 by 132, cursor origin can be set from 1/1 to 24/1.

10.6.2.4 PRINTER INTERFACE ON SERIAL BUS -

The Commodore printers are very cost effective but connect to the serial bus and are not very fast. For this reason two different printer drivers are installed in the system. The RS232C driver allows the connection of any RS232C printer but at a much higher cost and the loss of the RS232 port for other operations. The driver for this interface is described in section 10.5.2.6 This section deals with the serial bus printers. All of these printers work with PET ASCII (two sets of printable characters, upper case with graphics, and an upper and lower case with upper and lower case swapped) and a few also work with normal ASCII. Thus the real ASCII characters from CP/M must be convertedto PET ASCII in most cases before it is sent to the printer. To further complicate things some of the printers only have upper case so a force to upper case may also be necessary. The Z80 BIOS handles all of the conversion and gives the character to the 8500 to send to the printer. The 8500 is set up to time out if the printer is missing or out of service so that the system does not hang.

The following printers are supported:

MPS-801	Dot Matrix Printer	(50 cps)
MPS-802	Dot Matrix Printer	(80 cps)
MPS-803	Dot Matrix Printer	(60 cps)
DPS-1101	Daisy Wheel Printer	(supports normal ASCII)

10.6.2.5 GET A CHARACTER FROM RS232C ADAPTER (WITH XON/XOFF) -

The RS232C port does not have a USART to do the dirty work for it. This means that the serialization of the data must be done by the microprocessor. The 8500 microprocessor KERNAL contains routines that receive the bits and builds the characters. For these routines to work the 8500 processor must be running. To get around this problem XON/XOFF logic is used. When the Z80 wants a character it asks the 8500 to get it. The 8500 then sends an XON character. After the first character is received an XOFF is sent. The 8500 waits for a full character time after the XOFF is sent or until the next received character is received before the Z80 is turned back on. Any characters that came in after the first are buffered and the next time the Z80 wants a character, if one is in the buffer, it is returned to the Z80 without turning on the channel again (with an XON).

10.6.2.6 SEND A CHARACTER TO RS232C ADAPTER -

The RS232C port output is simpler than the input (see 10.5.2.5). The Z80 gives the character to be sent to the 8500 and the 8500 retains control until the character is totally sent (including the stop bits). Then the Z80 is given back control. One special case that is handled is that before the first character is sent, an XOFF is sent so that the device that is being communicated with knows that it cannot talk. Also, the device being communicated with may send an XON or XOFF. Since we may not be in there when it comes in, we must stay in the Send Char routine long enough to receive a control character. Some devices may never send an XON/XOFF and then the added delay may not be necessary. Thus, this added delay is menu selectable. The effect of the added XON/XOFF delay is to divide the effective BAUD rate by approximately 2. (1200 BAUD would have an effective rate of approximately 600 BAUD).

10.6.2.7 SET RS232C PARAMETERS -

The BAUD rate can be set from 110 to 1200 BAUD. This function changes the BAUD rate to the value supplied and sends an XOFF char to the connected device (see 10.5.2.5)

10.6.3 SYSTEM OPERATIONS

10.6.3.1 SET SYSTEM TIME -

The time of day is set with this function. The time of day is stored in packed BCD format in the System Control Block (SCB) in three locations (hours,minutes,seconds). This routine reads the SCB time and writes that time to the time of day clock within the 6526. This time is updated on the chip and is used by CP/M. The Z80 is able to set and read the 6526 directly.

10.6.3.2 UPDATE SYSTEM TIME -

The SCB time is updated from the time of day clock on the 6526. This function is done by the Z80.

10.6.3.3 MEMORY TO MEMORY MOVE -

This routine is the key to using the extra RAM banks. This routine is passed the source address and bank number, the destination address and bank number and the number of bytes to transfer. The transfer is done using the two MMU page pointers and the Z80's block move. There are three basic cases that are handled. The first is 256-byte moves that are page aligned. The second is a page aligned move of less than 256 bytes. The third case is where the 8 LSB's are not equal, and the effective addresses must be computed twice for each 256 byte block moved. The third case should not occur normally, but it should be tested for and handled in the event that it does occur. The majority of the moves are of the first type and this routine is optimized for speed. The second and third cases occur less frequently and do not need to be as fast. This code MUST reside in the common memory area and bank 0 cannot be enabled when it's invoked. The reason for this is that the Z80 ROM will overlay page 0 and 1 when bank 0 is selected.

10.7 8500 BIOS ORGANIZATION

The 8500 is responsible for most of the low level I/O functions. The request for these functions is made through a set of mailboxes. Once the mailboxes are set up the Z80 shuts down and the 8500 starts up (BIOS85). The 8500 looks at the command in the mailbox and performs the required task, sets the command status and shuts down. The Z80 is reenabled and looks at the command status and takes the appropriate actions.

The structure of the mailboxes is as follows:

Command	1 byte
Status	1 byte
Drive	1 byte
Track	1 byte
Sector	1 byte

The Commands are :

Reset, return control to the 8500 (BASIC)
Disk read
Disk write
Format the disk
Get a key
Send a byte to RS232C

Read a byte from RS232C

SEE BELOW

The status format is as follows :

MSB LSB

X7 X6 X5 X4 X3 X2 X1 X0

No errors if all bits are off

If X7 is on then an error has occured and the error type can be found by looking at the rest of the bits. The meaning of the rest of the bits is dependent on the command.

C O M M A N D S

00 Reset, return control to 8500 (BASIC)

Input: None

Output: None

Function: Returns control to the 8500 BASIC. The CP/M environment is lost, and the system must be rebooted to re-enable CP/M.

01 Disk read (slow)

Input: Track, sector, and drive number

Output: Command status

Function: Read a sector of data from disk to the sector buffer.

02 Disk write (slow)

Input: Track, sector, and drive number

Output: Command status

Function: Write a sector of data from the sector buffer to disk.

03 Disk read setup (fast)

Input: Track, sector, and drive number

Output: Command status

Function: Used to setup a read of the specified number of sectors from disk.

04 Disk write setup (fast)

Input: Track, sector, and drive number

Output: Command status

Function: Used to setup a write of the specified number of sectors to disk.

05 Format a new disk

Input: Drive number

Output: Command status

Function:

06 Send character to the printer

Input: Character to send

Output: Command status

Function: Send a character to the printer on the serial bus.

10.8 CP/M BIOS ORGANIZATION (BIOS80)

0 BOOT

Bank: 0

Input: None

Output: None

Function: This code does all of the hardware initialization, sets up zero page, prints any sign-on message and loads the CCP and then transfers control to the CCP.

1 WBOOT

Bank: 0 or 1

Input: None

Output: None

Function: This code sets up page zero, reloads the CCP and then executes the CCP. (relocate the stack to common memory to make BDOS calls).

2 CONST

Bank: 0 or 1

Input: None

Output: A=0FFH if console character
A=00H if no console character

Function: Checks the console input status of the current console device. If any of the devices have a character available, FFH is returned, else 00H is returned.

3 CONIN

Bank: 0 or 1

Input: None

Output: A=ASCII console character

Function: Reads a character from any ONE of the assigned console input devices. A scan of each assigned device is done until an input character is found. The character is returned in the A register.

4 CONOUT

Bank: 0 or 1

Input: C=ASCII character to display

Output: None

Function: Send the character in C to ALL devices that are currently assigned to the console. Wait for all slower devices.

5 LIST

Bank: 0 or 1

Input: C=ASCII character to print

Output: None

Function: Send the character in C to ALL devices that are currently assigned to the LIST device. Wait for all slower devices.

6 AUXOUT

Bank: 0 or 1

Input: C=ASCII Character to send to AUX device

Output: None

Function: Send the character in C to ALL devices that are currently assigned to the AUXOUT device. Wait for ALL slower devices.

7 AUXIN

Bank: 0 or 1

Input: None

Output: A=ASCII character from AUX device

Function: Reads a character from any ONE of the assigned AUXIN devices. A scan of each assigned device is done until an input character is found. The character is returned in the A register.

8 HOME

Bank: 0

Input: None

Output: None

Function: Homes the head on the currently selected disk drive. This function sets the current track to 0 and does not move the head of the disk in this product.

9 SELDSK

Bank: 0

Input: C=Disk Drive (0-15) (A=0)
E=Initial Select Flag

Output: HL=Address of disk parameter Header (DPH) if drive exists.
HL=000H if drive does not exist.

Function: Selects the disk drive whose address is in C as the current drive for all further disk operations. If the LSB of the E register is a zero, then this is the first time logging of this disk. The disk type (C64 CP/M or C128 CP/M) should be checked and the DPB parameters adjusted for the diskette currently in the drive.

10 SETTRK

Bank: 0

Input: BC=Track number (0-34)

Output: None

Function: Register pair BC contains the track number to be used in the subsequent disk access. This value is saved.

11 SETSEC

Bank: 0

Input: BC=Sector number

Output: None

Function: Register pair BC contains the sector number to be used in the subsequent disk access. This value is saved. The value in BC is the value returned by the sector translation routine.

12 SETDMA

Bank: 0

Input: BC=Direct memory access address

Output: None

Function: The value in BC is saved as the current DMA address. This is the address where ALL disk read or writes occur to or from. The DMA address that is set is used until it is changed by a future call to this routine to change it.

13 READ

Bank: 0

Input: None

Output: A=000H if no errors

A=001H if nonrecoverable error

A=0FFH if media has changed

Function: Reads the sector addressed by the current disk, track and sector to the current DMA address. If the data is read with no errors then A=0 on return. If an error occurs, the operation is tried several more times, and if a good read does not occur then A is set to 001H. A test for media change should be done each time this routine is called and A is set to -1 if the media has been changed.

14 WRITE

Bank: 0

Input: C=Deblocking code (not used)

Output: A=000H if no errors

A=001H if nonrecoverable error

A=002H if disk is read only

A=0FFH if media has changed

Function: Writes the sector addressed by the current disk, track and sector from the current DMA address. If the data is written with no errors, then A is set to 0 on return. If an error occurs, the operation is tried several more times, and if a good write does not occur, then A is set to 001H. A test for media change should be done each time this routine is called and A is set to -1 if the media has been changed. Also, if an attempt is made to write to a read-only disk, then the A register is set to 002H.

15 LISTST

Bank: 0 or 1

Input: None

Output: A=00H if list device is not ready to accept a character.
A=0FFH if list device is ready to accept a character.

Function: This routine scans the currently assigned list devices and returns with A set to 0FFH if ALL assigned devices are ready to accept a character. If any assigned device is not ready then A is set to 00H.

16 SECTRN

Bank: 0

Input: BC=Logical sector number (0-n)
DE=Translation table address (from DPB)

Output: HL=Physical sector number

Function: This routine converts the physical sector number to a logical sector number. If no translation is needed then move the BC register to HL and return.

17 CONOST

Bank: 0 or 1

Input: None

Output: A=0FFH if Ready
A=000H if not Ready

Function: This routines scans the currently assigned console devices and returns with A set to 0FFH if ALL assigned

devices are ready to accept a character. If any assigned device is not ready then A is set to 000H.

18 AUXIST

Bank: 0 or 1

Input: None

Output: A=0FFH if console character present

A=000H if no console characeter

Function: Checks the status of the current AUXIN device. If any of the devices have a character available, 0FFH is returned, else 000H is returned.

19 AUXOST

Bank: 0 or 1

Input: None

Output: A=0FFH if ready

A=000H if not ready

Function: This routine scans the currently assigned AUXOUT devices and returns with A set to 0FFH if ALL devices are ready to accept a character. If any assigned device is not ready then A is set to 00H.

20 DEVTBL

Bank: 0 or 1

Input: None

Output: HL=address of character I/O table

Function: This routine returns the address of the Character I/O table. This table is used to name each of the driver modules and set/control the baud rate and XON/XOFF logic for each driver. Note: the device drive mechanism is used to replace the IOBYTE used with CP/M 2.2.

21 DEVINI

Bank: 0 or 1

Input: C=device number

Output: None

Function: Used to initialize the physical character device

specified in the C register to the BAUD rate in the DEVTBL.

22 DRVTBL

Bank: 0

Input: None

Output: HL=address of the drive table

Function: Returns the address of the drive table in HL (NOTE: first instruction MUST be LXI H,DRVTBL). The drive table is a list of 16 word pointers that point to the DPH for that drive. If a drive is not present in the system, then the pointer for that drive is set to zero.

23 MULTIO

Bank: 0

Input: C=multisector count

Output: None

Function: The Multisector count is set before the track, sector, and DMA address and the reads/writes of the sectors occur. A maximum of 16K can be transferred by each multisector count. The 1541 has 256 byte sectors, thus the maximum count is 64. When sector skewing is done, a list should be created that contains the physical track, sector, and DMA address of each read/write sector and after the last sector is entered into the list. The list should be sortedby track and sector. The read/write operation should then be performed. Multisector transfer is ONLY worth while with the new faster disk drive.

24 FLUSH

Bank: 0

Input: None

Output: A=000H if no errors

A=001H if nonrecoverable error

A=002H if disk is read-only

A=0FFH if media has changed

Function: This routine is used only if blocking/deblocking is done in the BIOS (it is not). This code ALWAYS returns

with A =00H.

25 MOVE

Bank: 0 or 1

Input: HL=destination address

DE=source address

BC=count

Output: HL=HL(in)+BC(in)

DE=DE(in)+BC(in)

Function: Move a block of data. Data to be moved is to/from the current memory bank (or common) unless the XMOVE routine is called first, then the move is an interbank data movement.

26 TIME

Bank: 0 or 1

Input: C=000H (Time Get) / 0FFH (Set Time)

Output: None

Function: This function is called with C=00H if the system time in the SCB needs to be updated by the clock. If C=0FFH, then the time in the SCB has just been updated and the clock should be set to the SCB time. NOTE: HL and DE MUST be preserved.

27 SELMEM

Bank: 0 or 1

Input: A=memory bank

Output: None

Function: Used to change the current memory bank. This code MUST be in common memory. NOTE: ONLY A can be changed.

28 SETBNK

Bank: 0

Input: A=DMA memory bank

Output: None

Function: Set the DMA bank for the next READ/WRITE operation.

29 XMOVE

Bank: 0

Input: B=destination bank

C=source bank

Output: None

Function: Provides the system with the ability to do memory to memory DMA through the entire system space. This function does not have to be supplied. If this function is missing, the first instruction MUST be a RET.

30 USERF

Bank: N/A

Input: N/A

Output: N/A

Function: N/A

31 RESERV1

Bank: N/A

Input: N/A

Output: N/A

Function: N/A

32 RESERV2

Bank: N/A

Input: N/A

Output: N/A

Function: N/A

10.8.1 DATA STRUCTURES

System Control Block - SCB

The System Control Block is a 100 byte data structure. The data structure is used as the basic communication between the various modules that make up the CP/M Plus system. The contents of the data structure are system parameters and variables.

DRVTBL Drive Table

DPH0 through DPH15

A list of 16 word pointers (reverse byte format). The first pointer (DPH0) is drive A and the last pointer (DPH15) is drive P. The pointers point to the XDPH for that disk drive. Any drive that is not in the system has its pointers set to zero.

XDPH Extended Disk Parameter Header (Normal DPH with a header)

WRT	READ	LOGIN	INIT	TYPE	UNIT	XLT	-0-	MF	DPB	CSV	ALV	DIRBCB	DTABCB	HASH	HBANK
16b	16b	16b	16b	8b	8b	16b	72b	8b	16b	16b	16b	16b	16b	16b	8b
-10	-8	-6	-4	-2	-1	0	+2	+11	+12	+14	+16	+18	+20	+22	+24

WRT Contains the address of the sector write routine for this drive.
READ Contains the address of the sector read routine for this drive.
LOGIN Contains the address of the login routine for this drive.

INIT Contains the address of the first time initialization routine for this drive.

TYPE This byte is used by the BIOS to keep track of density and media type.

UNIT Contains the drive number relative to the disk controller.

XLT Contains the address of the sector translation table or zero if none.

-0- BDOS scratch area. (9 bytes).

MF Media flag set to zero if disk logged in. BIOS sets to 0FFH if media has changed.

DPB Contains a pointer to the current DPB that describes the current media type.

CSV Contains a pointer to directory checksum area (one per disk drive).

ALV Contains a pointer to allocation vector area (one per disk drive).

DIRBCB Contains a pointer to a single directory Buffer Control Block (BCB).

DTSBCB Contains a pointer to a single data Buffer Control Block (BCB).

HASH Contains a pointer to an optional directory hashing table (FFFFH is not used).

HBANK Contains a bank number of the directory hashing table.

Disk Parameter Block - DPB

SPT	BSH	BLM	EXM	DSM	DRM	AL0	AL1	CKS	OFF	PSH	PHM
16b	8b	8b	8b	16b	16b	8b	8b	16b	16b	8b	8b

SPT Number of 128 records per track

BSH Data allocation block shift factor

BLM Block mask

EXM Extent mask

DSM Number of allocation block on disk minus one.

DRM Number of directory entries minus one.

AL0 First byte: directory block allocation vector. Filled from MSB to LSB

AL1 Second byte: (up to 16 allocation blocks can be used for the directory.

CKS Size of the directory check vector, (DRM+1)/4.

OFF Number or reserved tracks at the beginning of the disk.

PSH Physical record shift factor.

PHM Physical record mask.

Buffer Control Block (LRU Control Block) - BCB

DRV	REC#	WFLG	00	TRACK	SECTOR	BUFFAD	BANK	LINK
8b	24b	8b	8b	16b	16b	16b	8b	16b

DRV Drive associated with this record. Set to 0FFH when not used.

REC# Contains the absolute sector number of the buffer.

WFLG Set to 0FFH when buffer contains data that must be written to disk

00 Scratch byte used by BDOS

TRACK Physical track address of buffer

SECTOR Physical sector address of buffer.

BUFFAD Address of the buffer associated with this BCB.

BANK   Bank number of buffer associated with this BCB.

LINK   Contains the address of the next BCB in this linked list.
       Set to zero if last.

CHAPTER 11

FAST DISK INTERFACE

11.1 SERIAL BUS INTERFACE

The FSD implements a modified 1541 serial interface. This bus is compatible with the Commodore 64, Vic 20, and the Plus 4 Series Computers. In addition it is compatible with peripherals such as printers, plotters, and especially the 1541 disk drive.

Fast Serial communication is transparent to the slower peripherals. There are three types of operations over a serial bus - Control, Talk, and Listen. The host is the controller and initiates all protocol on the serial bus. The host requests the peripheral to listen or talk (if the peripheral is capable of talking). All devices connected to the serial bus receive data transmitted over the bus. To allow the host to route its data to an intended destination, each device has a bus

address. Device addresses are as follows:

DEVICE ADDRESS 0-30 POSSIBLE

0-3	INTERNAL DEVICES OS
4-7	NORMAL CBM PRINTERS
8-11	NORMAL CBM DISK UNITS
12-30	UNUSED CBM DEVICES

The bus consists of the following:

Pin 1 - SRQ (Service Request)

Unused by the current serial bus. Fast Serial will use this line as a bi-direction fast clock line.

Pin 2 - GND

Chassis ground.

Pin 3 - ATN (in)

The host brings this signal low which then generates an interrupt on the controller board. The attention sequence is followed by an address. If the device does not respond within a preset time the host will assume the device addressed is not on the bus.

Pin 4 - CLK (in/out)

This signal is used for timing the data sent on the serial bus (software clocked).

Pin 5 - DATA (in/out)

Data on the serial bus is transmitted one bit at a time (software toggled). In addition this line is wire 'ored' and used as a FAST DATA line to compliment the FAST CLOCK on the SRQ line.

Pin 6 - RESET

This line will reset the peripheral upon host reset.

11.2 FAST SERIAL PROTOCOL

The FSD drive powers up in the slow serial mode. The host has to init the drive to fast mode (The drive will remain in the fast mode until the command has terminated). There are no addition kernal calls required to interface to a FSD. Existing kernal routines are modified to allow slow and fast serial operation. Within the kernal a fast serial flag contains whether the current addressed peripheral is a fast device. To initiate the FSD as a fast peripheral the host must send a HRF (Host Request Fast) message. (see Fig. 1) This is accomplished by sending eight clock pulses down the SRQ (Service Request) line. The 6526 on the drive's controller board will sense the transitions and the 6526 will generate an interrupt. Within the drive a flag is toggled to the fast mode. While in the fast mode the drive will send fast bytes to the host, and when addressed as a listener the drive will send a DRF (Device Request Fast) message (see Fig. 1). This message lets the host know that the addressed peripheral can receive bytes fast (or slow). The fast serial flag within the host can be cleared in the following ways: unlisten, untalk, serial bus error, and <run-stop> <restore>. As mentioned before bytes are clocked on the SRQ line with the data line be toggled to the appropiate state. Existing routines used to accept bytes were modified to accept a fast byte as well as a slow byte.

11.3 SERIAL BUS COMMANDS - MODIFIED

EXPLANATION OF TERMS:

HRF - Host request fast

drf - device request fast (sourced by the drive)

LA - Listen address

TA - Talk address

SA - Secondary address

SA(O) - Secondary address open

SA(C) - Secondary address close

DB - Data Byte

FN - File name byte

eoi - End or identify handshake

TKATN - Talk-Atn handshake

	Command	Abbreviation	Binary value
*	host request fast	HRF	1111 1111
**	device request fast	DRF	0000 0000
	talk address	(TA)	010x xxxx
	listen address	(LA)	001x xxxx
	untalk	(UNTLK)	0101 1111
	unlisten	(UNLSN)	0011 1111
	secondary address open	(SA(O))	1111 yyyy
	secondary address close	(SA(C))	1110 yyyy
	secondary address normal	(SA)	011z zzzz

* Fast Byte messages (clocked over the SRQ line)

** Fast byte message sourced by the drive

Device Address (TA) (LA) = x xxxx values 0-30 possible

0-3	Internal devices
4-7	Normal CBM printers
8-11	Normal disk units
12-30	Unused

Channel Address (SA(O)) (SA(C)) = yyyy values 0-15 possible

Disk Units 154X

0 - PRG-TYPE Read Data Channel	(special)
1 - PRG-TYPE Write Data Channel	(special)
2-14 - Channel for all file types	(read/write)
15 - Unit command channel	(read/write)

Normal Secondary Address (SA) = z zzzz values 0-31 possible

11.4 STANDARD KERNAL CALLS

Load:

This routines loads data bytes from any input device directly into the hosts memory.

```
HRF LA SA(O) drf FN1 FN2...FNn-1 eoi FNn HRF UNLSN =>
HRF TA SA TKATN DB1 DB2...DBn-1 eoi DBn UNTLK =>
HRF TA SA(C) HRF UNLSN
```

Save:

This routines saves a section of memory.

```
HRF LA SA(O) FN1 FN2...FNn-1 eoi FNn HRF UNLSN =>
HRF LA SA drf DB1 DB2...DBn eoi DBn-1 HRF UNLSN =>
HRF LA SA(C) HRF UNLSN
```

Open: with SA

This routine is used to open a logical file for I/O operations.

```
HRF LA SA(O) drf FN1 FN2...FNn-1 eoi FNn HRF UNLSN
```

Close: with SA

This routine is used to close a logical file after all I/O operations have been completed on that file.

HRF LA SA(C) HRF UNLSN

Chkout: with SA

This routine must be called before any data is sent to any output device.

HRF LA SA

Chkin: with SA

This routine is called to define any previously opened

channel as a input channel.

HRF TA SA TLKATN

Chrout:

This uses a single character buffer, and will send previously buffered character, if any exists. This buffer is also sent along with eoi, prior to sending any SERIAL BUS COMMAND sequence (HRF, LA, TA, SA(O), SA(C), SA, UNTLK, UNLSN).

Chrin:

This routine is called to get a byte of data from a channel already setup as a input channel.

DBc or eoi DBc (if external device sends eoi)

Getin:

- see Chrin -

Clrchn:

This routine is used to clear and restore all open

channels to there default values.

If Chkin channel open: UNTLK.

If Chkout channel open: eoi DBc HRF UNLSN

Clall:

- see Clrchn -

Stop:

This routine is used to detect the stop key. If stop key down, Clrchn called.

11.5 BURST COMMANDS ADDED TO DOS 2.65

NOTE: ALL BURST COMMANDS ARE SENT VIA KERNAL I/O CALLS.

BURST CMD ONE - READ

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	T	E	B	S	0	0	0	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE:

MFM ALL VALUES ARE DETERMINED UPON THE PARTICULAR DISK FORMAT.

GCR SEE SOFTWARE SPECIFICATIONS.

SWITCHES:

T - TRANSFER DATA (1=NO TRANSFER)

E - IGNORE ERROR (1=IGNORE)

B - BUFFER TRANSFER ONLY (1=BUFFER TRANSFER ONLY)

S - SIDE SELECT (MFM ONLY)

N - DRIVE NUMBER

PROTOCOL:

BURST HANDSHAKE.

CONVENTIONS:

CMD ONE MUST BE PRECEEDED WITH CMD 3 OR CMD 6 ONCE TO LOG THE DISK IN, THEN READS OR WRITES CAN BE PERFORMED UNTIL THE DISK IS CHANGED.

OUTPUT:

ONE BURST STATUS BYTE PRECEEDING BURST DATA WILL BE SENT FOR EVERY SECTOR TRANSFERED. ON AN ERROR CONDITION DATA WILL NOT BE SENT UNLESS THE E BIT IS SET.

BURST CMD TWO - WRITE

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	T	E	B	S	0	0	1	N
03	DESTINATION TRACK							
04	DESTINATION SECTOR							
05	NUMBER OF SECTORS							
06	NEXT TRACK (OPTIONAL)							

RANGE:

MFM ALL VALUES ARE DETERMINED UPON THE PARTICULAR DISK FORMAT.

GCR SEE SOFTWARE SPECIFICATIONS.

SWITCHES:

T - TRANSFER DATA (1=NO TRANSFER)

E - IGNORE ERROR (1=IGNORE)

B - BUFFER TRANSFER ONLY (1=BUFFER TRANSFER ONLY)

S - SIDE SELECT (MFM ONLY)

N - DRIVE NUMBER

PROTOCOL:

BURST DATA TO THE DRIVE, THEN HOST MUST PERFORM THE FO-
LLOWING: FAST SERIAL INPUT, PULL THE CLOCK LOW AND WAIT
FOR THE BURST STATUS BYTE, PULL CLOCK HIGH, GO OUTPUT
FOR MULTI-SECTOR TRANSFERS AND CONTINUE.

CONVENTIONS:

CMD TWO MUST BE PRECEEDED WITH CMD 3 OR CMD 6 ONCE TO LOG
THE DISK IN, THEN READS OR WRITES CAN BE PERFORMED UNTIL
THE DISK IS CHANGED.

INPUT:

HOST MUST TRANSFER BURST DATA.

OUTPUT:

ONE BURST STATUS BYTE FOLLOWING EACH WRITE OPERATION.

BURST CMD THREE - INQUIRE DISK

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	S	0	1	0	N

SWITCHES:

S - SIDE SELECT (MFM ONLY)

N - DRIVE NUMBER

OUTPUT:

ONE BURST STATUS BYTE FOLLOWING THE INQUIRE OPERATION.

BURST CMD FOUR - FORMAT MFM

BYTE	BIT 7	6	5	4	3	2	1	0	
00	0	1	0	1	0	1	0	1	
01	0	0	1	1	0	0	0	0	
02	P	I	D	S	0	1	1	N	
03	M=1	T	LOGICAL STARTING SECTOR						
04	INTERLEAVE								(OPTIONAL DEF-0)
05	SECTOR SIZE								* (OPTIONAL DEF-01,256 BYTE SECTORS)
06	LAST TRACK NUMBER								(OPTIONAL DEF-39)
07	NUMBER OF SECTORS								** (OPTIONAL DEPENDS ON BYTE 05)
08	LOGICAL STARTING TRACK								(OPTIONAL DEF-0)
09	STARTING TRACK OFFSET								(OPTIONAL DEF-0)
0A	FILL BYTE								(OPTIONAL DEF-$E5)
0B-??	SECTOR TABLE								(OPTIONAL T-BIT SET)

```
*  00 - 128 BYTE SECTORS        ** DEF  26 - 128 BYTE SECTORS
   01 - 256 BYTE SECTORS                16 - 256 BYTE SECTORS
   02 - 512 BYTE SECTORS                 9 - 512 BYTE SECTORS
   03 - 1024 BYTE SECTORS                5 - 1024 BYTE SECTORS
```

SWITCHES:

P - PARTIAL FORMAT (1=PARTIAL)

I - INDEX ADDRESS MARK WRITTEN (1=WRITTEN)

D - DOUBLE SIDED FLAG (1=FORMAT DOUBLE SIDED)

S - SIDE SELECT

T - SECTOR TABLE INCLUDED (1=INCLUDED, ALL OTHER PARMS MUST BE INCLUDED)

N - DRIVE NUMBER

PROTOCOL:

CONVENTIONAL.

CONVENTIONS:

CMD FOUR MUST BE FOLLOWED WITH CMD 3 OR CMD 6 ONCE TO LOG THE DISK IN.

OUTPUT:

NONE. STATUS WILL BE UPDATED WITHIN THE DRIVE.

BURST CMD FOUR - FORMAT GCR (NO DIRECTORY)

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	X	0	1	1	N
03	M=0							
04	ID LOW							
05	ID HIGH							

SWITCHES:

N - DRIVE NUMBER

X - DON'T CARE.

PROTOCOL:

CONVENTIONAL.

CONVENTIONS:

CMD FOUR MUST BE PRECEEDED WITH CMD 3 OR CMD 6 ONCE TO LOG THE DISK IN. THIS COMMAND DOES NOT WRITE THE BAM OR THE DIRECTORY ENTRIES (DOUBLE SIDED FLAG WILL NOT BE ON TRACK 18 SECTOR 0). IT IS SUGGESTED THAT THE CONVENTIONAL FORMAT COMMAND BE USED.

OUTPUT:

NONE. STATUS WILL BE UPDATED WITHIN THE DRIVE.

BURST CMD FIVE - SECTOR INTERLEAVE

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	W	X	X	0	1	0	0	N
04	INTERLEAVE							

SWITCHES:

W - WRITE SWITCH

N - DRIVE NUMBER

X - DON'T CARE

PROTOCOL:

CONVENTIONAL.

CONVENTIONS:

THIS IS A SOFT INTERLEAVE USED FOR MULTI-SECTOR BURST

READ AND WRITE.

OUTPUT:

NONE (W=0), INTERLEAVE BURST BYTE (W=1).

BURST CMD SIX - QUERY DISK FORMAT

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	F	X	X	S	1	0	1	N
03	OFFSET (OPTIONAL F-BIT SET)							

SWITCHES:

F - FORCE FLAG (F=1, WILL STEP THE HEAD WITH THE OFFSET SPECIFIED IN BYTE 03.

N - DRIVE NUMBER

X - DON'T CARE.

PROTOCOL:

CONVENTIONAL.

CONVENTIONS:

THIS IS A METHOD OF DETERMINING THE FORMAT OF THE DISK ON ANY PARTICULAR TRACK, IT ALSO LOGS NON-STANDARD DISKS (IE. MINIMUM SECTOR ADDRESSES OTHER THAN ZERO).

OUTPUT:

* BURST STATUS BYTE (IF THERE WAS AN ERROR OR IF THE FORMAT IS GCR NO BYTES WILL FOLLOW)

** BURST STATUS BYTE (IF THERE WAS AN ERROR IN COMPILING MFM FORMAT INFORMATION NO BYTES WILL FOLLOW)

NUMBER OF SECTORS (THE NUMBER OF SECTORS ON A PARTICULAR TRACK)

LOGICAL TRACK (THE LOGICAL TRACK NUMBER FOUND IN THE DISK HEADER)

MINIMUM SECTOR (THE LOGICAL SECTOR WITH THE LOWEST VALUE ADDRESS)

MAXIMUM SECTOR (THE LOGICAL SECTOR WITH THE HIGHEST VALUE ADDRESS)

CP/M INTERLEAVE (THE HARD INTERLEAVE FOUND ON A PARTICULAR TRACK)

* STATUS FROM TRACK OFFSET ZERO.

** IF F BIT IS SET STATUS IS FROM OFFSET TRACK.

BURST CMD SEVEN - INQUIRE STATUS

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	W	C	X	0	1	1	0	N
03	NEW STATUS (W-BIT CLEAR)							

SWITCHES:

W - WRITE SWITCH

C - CHANGE (C=1 & W=0 - LOG IN DISK, C=1 & W=1 - RETURN WHETHER DISK WAS LOGGED IE. $B ERROR OR OLD STATUS)

N - DRIVE NUMBER

X - DON'T CARE

PROTOCOL:

BURST.

CONVENTIONS:

THIS IS A METHOD OF READING OR WRITING CURRENT STATUS.

OUTPUT:

NONE (W=0), BURST STATUS BYTE (W=1)

BURST CMD EIGHT - BACKUP DISK

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	?	?	?	?	1	1	1	?

SWITCHES:

? - UNKNOWN

BURST CMD NINE - CHGUTL UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	X	X	X	1	1	1	1	0
03	UTILITY COMMANDS: 'S', 'R', 'T', 'M', 'H', #DEV							
04	COMMAND PARAMETER							

SWITCHES:

X - DON'T CARE.

UTILITY COMMANDS:

'S' - DOS SECTOR INTERLEAVE.

'R' - DOS RETRIES.

'T' - ROM SIGNATURE ANALYSIS.

'M' - MODE SELECT.

'H' - HEAD SELECT.

#DEV - DEVICE #.

NOTE: BYTE 02 IS EQUIVALENT TO A '>'

EXAMPLES:

```
        "U0>S"+CHR$(SECTOR-INTERLEAVE)
        "U0>R"+CHR$(RETRIES)
        "U0>T"
        "U0>M1"=1571 MODE, "U0>M0"=1541 MODE
  *     "U0>H0"=SIDE ZERO, "U0>H1"=SIDE ONE (1541 MODE ONLY)
        "U0>"+CHR$(#DEV), WHERE #DEV = 8-30
```

* FOR DUAL DRIVE ADD ,1 FOR DRIVE ONE AND ,0 FOR DRIVE ZERO.

BURST CMD TEN - FASTLOAD UTILITY

BYTE	BIT 7	6	5	4	3	2	1	0
00	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	0	0
02	P	X	X	1	1	1	1	1
03	FILE NAME							

SWITCHES:

P - SEQUENTIAL FILE BIT (P=1, DOES NOT HAVE TO BE A PROGRAM FILE)

X - DON'T CARE.

PROTOCOL:

BURST.

OUTPUT:

BURST STATUS BYTE PRECEEDING EACH SECTOR TRANSFERED.

STATUS IS AS FOLLOWS:

```
      0000000X ..... OK
   *  00000010 ..... FILE NOT FOUND
      00011111 ..... EOI
```

* VALUES BETWEEN THE RANGE 3-15 SHOULD BE CONSIDERED A FILE READ ERROR.

STATUS BYTE BREAK DOWN

BIT 7	6	5	4	3	2	1	0
MODE	DN	SECTOR SIZE		[CONTROLLER	STATUS		]

MODE - 1=MFM, 0=GCR

DN - DRIVE NUMBER

SECTOR SIZE - (MFM ONLY)

00 .... 128 BYTE SECTORS

01 .... 256 BYTE SECTORS

10 .... 512 BYTE SECTORS

11 .... 1024 BYTE SECTORS

CONTROLLER STATUS (GCR)

000X .... OK

0010 .... SECTOR NOT FOUND

0011 .... NO SYNC

0100 .... DATA BLOCK NOT FOUND

0101 .... DATA BLOCK CHECKSUM ERROR

0110 .... FORMAT ERROR

0111 .... VERIFY ERROR

1000 .... WRITE PROTECT ERROR

1001 .... HEADER BLOCK CHECKSUM ERROR

1010 .... DATA EXTENDS INTO NEXT BLOCK

1011 .... DISK ID MISMATCH/ DISK CHANGE

1100 .... RESERVED

1101 .... RESERVED

1110 .... SYNTAX ERROR

1111 .... NO DRIVE PRESENT

CONTROLLER STATUS (MFM)

000X .... OK

0010 .... SECTOR NOT FOUND

0011 .... NO ADDRESS MARK

0100 .... RESERVED

0101 .... DATA CRC ERROR

0110 .... FORMAT ERROR

0111 .... VERIFY ERROR

1000 .... WRITE PROTECT ERROR

1001 .... HEADER BLOCK CHECKSUM ERROR

1010 .... RESERVED

1011 .... DISK CHANGE

1100 .... RESERVED

1101 .... RESERVED

1110 .... SYNTAX ERROR

1111 .... NO DRIVE PRESENT

CHAPTER 12

RELATED DOCUMENTATION

1. 6510 Chip Specification

2. 6567 Chip Specification

3. 8563 Chip Specification

4. C/64 Programmer's Reference Guide

5. CP/M PLUS Reference Manual

6. C/128 Hardware Specification

7. C/128 MMU Specification

8. SY6545E CRT Controller

9. IC, LSI, Sound Interface Device - 8580

10. IC, LSI, Video Controller - 8564

11. The C128 Fast Serial Disk Software Specification