

C 1 6   a n d   P L U S / 4

P R O G R A M M E R ' S

G U I D E

Copyright (C) 1985    COMMODORE BUSINESS MACHINES (U.K.) LTD.







## TABLE OF CONTENTS

## SECTION ONE - INTRODUCTION

1.1	Introduction.....	1	-	1
1.2	What is Included?.....	1	-	1

## SECTION TWO - PROGRAMMING BASIC

2.1	Introduction.....	2	-	1
2.2	Command and Statement Format.....	2	-	1
2.3	BASIC Commands.....	2	-	3
2.3.1	AUTO.....	2	-	3
2.3.2	BACKUP.....	2	-	3
2.3.3	COLLECT.....	2	-	4
2.3.4	CONT.....	2	-	4
2.3.5	COPY.....	2	-	4
2.3.6	DELETE.....	2	-	5
2.3.7	DIRECTORY.....	2	-	5
2.3.8	DLOAD.....	2	-	6
2.3.9	DSAVE.....	2	-	6
2.3.10	HEADER.....	2	-	7
2.3.11	HELP.....	2	-	7
2.3.12	KEY.....	2	-	8
2.3.13	LIST.....	2	-	8
2.3.14	LOAD.....	2	-	9
2.3.15	NEW.....	2	-	10
2.3.16	RENAME.....	2	-	10
2.3.17	RENUMBER.....	2	-	10
2.3.18	RUN.....	2	-	11
2.3.19	SAVE.....	2	-	11
2.3.20	SCRATCH.....	2	-	12
2.3.21	VERIFY.....	2	-	12
2.4	BASIC Statements.....	2	-	13
2.4.1	BOX.....	2	-	13
2.4.2	CHAR.....	2	-	14
2.4.3	CIRCLE.....	2	-	14
2.4.4	CLOSE.....	2	-	15
2.4.5	CLR.....	2	-	15
2.4.6	CMD.....	2	-	15
2.4.7	COLOR.....	2	-	16
2.4.8	DATA.....	2	-	16
2.4.9	DEF FN.....	2	-	17
2.4.10	DIM.....	2	-	17
2.4.11	DO (LOOP) WHILE (UNTIL EXIT).....	2	-	18
2.4.12	DRAW.....	2	-	19
2.4.13	END.....	2	-	19
2.4.14	FOR ... TO ... STEP.....	2	-	19
2.4.15	GET.....	2	-	20
2.4.16	GETKEY.....	2	-	21
2.4.17	GET#.....	2	-	21
2.4.18	GOSUB.....	2	-	21
2.4.19	GOTO or GO TO.....	2	-	22
2.4.20	GRAPHIC.....	2	-	22
2.4.21	GRAPHIC CLR.....	2	-	23

# Table of Contents

2.4.22	IF ... THEN ... ELSE.....	2	-	23
2.4.23	INPUT.....	2	-	23
2.4.24	INPUT#.....	2	-	24
2.4.25	LET.....	2	-	24
2.4.26	LOCATE.....	2	-	25
2.4.27	MONITOR.....	2	-	25
2.4.28	NEXT.....	2	-	25
2.4.29	ON.....	2	-	26
2.4.30	OPEN.....	2	-	26
2.4.31	PAINT.....	2	-	27
2.4.32	POKE.....	2	-	28
2.4.33	PRINT.....	2	-	28
2.4.34	PRINT#.....	2	-	29
2.4.35	PRINT USING.....	2	-	30
2.4.36	PUDEF.....	2	-	33
2.4.37	READ.....	2	-	33
2.4.38	REM.....	2	-	34
2.4.39	RESTORE.....	2	-	34
2.4.40	RESUME.....	2	-	34
2.4.41	RETURN.....	2	-	34
2.4.42	SCALE.....	2	-	35
2.4.43	SCNCLR.....	2	-	35
2.4.44	SOUND.....	2	-	35
2.4.45	SSHAPE/GSHAPE.....	2	-	36
2.4.46	STOP.....	2	-	37
2.4.47	SYS.....	2	-	37
2.4.48	TRAP.....	2	-	37
2.4.49	TRON.....	2	-	37
2.4.50	TROFF.....	2	-	38
2.4.51	VOL.....	2	-	38
2.4.52	WAIT.....	2	-	38
2.5	Additional Graphic Statement Information.....	2	-	38
2.6	FUNCTIONS.....	2	-	39
2.6.1	Numeric Functions.....	2	-	39
2.6.1.1	ABS(X) (absolute value).....	2	-	39
2.6.1.2	ASC(X\$).....	2	-	39
2.6.1.3	ATN(X) (arctangent).....	2	-	40
2.6.1.4	COS(X) (cosine).....	2	-	40
2.6.1.5	DEC (hexadecimal-string).....	2	-	40
2.6.1.6	EXP(X).....	2	-	40
2.6.1.7	FNxx(X).....	2	-	40
2.6.1.8	INSTR.....	2	-	40
2.6.1.9	INT(X) (integer).....	2	-	41
2.6.1.10	JOY(n).....	2	-	41
2.6.1.11	LOG(X) (logarithm).....	2	-	41
2.6.1.12	PEEK(X).....	2	-	41
2.6.1.13	RCLR(N).....	2	-	42
2.6.1.14	RDOT(N).....	2	-	42
2.6.1.15	RGB(X).....	2	-	42
2.6.1.16	RLUM(N).....	2	-	42
2.6.1.17	RND(X) (random number).....	2	-	42
2.6.1.18	SGN(X) (sign).....	2	-	43
2.6.1.19	SIN(X) (sine).....	2	-	43
2.6.1.20	SQR(X) (square root).....	2	-	43
2.6.1.21	TAN(X) (tangent).....	2	-	43
2.6.1.22	USR(X).....	2	-	43

# Table of Contents

2.6.1.23	VAL(X\$).....	2	-	44
2.6.2	String Functions.....	2	-	44
2.6.2.1	CHR\$(X).....	2	-	44
2.6.2.2	ERR\$(N).....	2	-	44
2.6.2.3	HEX\$(N).....	2	-	44
2.6.2.4	LEFT\$(X\$,X).....	2	-	44
2.6.2.5	LEN(X\$).....	2	-	45
2.6.2.6	MID\$(X\$,N,X).....	2	-	45
2.6.2.7	RIGHT\$(X\$,X).....	2	-	45
2.6.2.8	STR\$(X).....	2	-	45
2.6.3	Other Functions.....	2	-	45
2.6.3.1	FRE(X).....	2	-	45
2.6.3.2	POS(X).....	2	-	46
2.6.3.3	SPC(X).....	2	-	46
2.6.3.4	TAB(X).....	2	-	46
2.6.3.5	$\pi$ (PI).....	2	-	46
2.7	VARIABLES AND OPERATORS.....	2	-	46
2.7.1	Variables.....	2	-	46
2.7.1.1	VARIABLE NAMES.....	2	-	47
2.7.1.2	RESERVED VARIABLE NAMES.....	2	-	47
2.7.2	BASIC OPERATORS.....	2	-	48
2.8	BASIC Abbreviation and Reference Chart.	2	-	50

## SECTION THREE - PROGRAMMING MACHINE CODE

3.1	What is Machine Language?.....	3	-	1
3.2	What does Machine Code Look Like?.....	3	-	1
3.3	Simple Memory Map of the C16 and PLUS/4	3	-	2
3.4	The Registers Inside the 7501 Microprocessor.....	3	-	3
3.4.1	THE ACCUMULATOR.....	3	-	3
3.4.2	THE X INDEX REGISTER.....	3	-	3
3.4.3	THE Y INDEX REGISTER.....	3	-	3
3.4.4	THE STATUS REGISTER.....	3	-	3
3.4.5	THE PROGRAM COUNTER.....	3	-	3
3.4.6	THE STACK POINTER.....	3	-	4
3.4.7	THE INPUT/OUTPUT PORT.....	3	-	4
3.5	Writing Machine Language Programs.....	3	-	4
3.5.1	TEDMON COMMANDS.....	3	-	5
3.5.2	USING TEDMON.....	3	-	6
3.5.3	COMMAND DESCRIPTIONS.....	3	-	6
3.6	HEXADECIMAL NOTATION.....	3	-	14
3.7	ADDRESSING MODES.....	3	-	14
3.7.1	ZERO PAGE.....	3	-	14
3.7.2	THE STACK.....	3	-	15
3.8	INDEXING.....	3	-	15
3.8.1	INDIRECT INDEXED.....	3	-	16
3.8.2	INDEXED INDIRECT.....	3	-	16
3.8.3	BRANCHES AND TESTING.....	3	-	17
3.9	SUBROUTINES.....	3	-	19
3.10	7501 MICROPROCESSOR INSTRUCTION SET - ALPHABETIC SEQUENCE.....	3	-	20
3.11	THE KERNAL.....	3	-	21
3.11.1	HOW TO USE THE KERNAL.....	3	-	22
3.11.2	USER CALLABLE KERNAL ROUTINES.....	3	-	24
3.11.3	KERNAL ROUTINE DESCRIPTIONS.....	3	-	25

Table of Contents

3.11.4	ERROR CODES.....	3 - 52
3.12	C16 AND PLUS/4 MEMORY MAP.....	3 - 53

APPENDIX A - SCREEN DISPLAY CODES

APPENDIX B - ASCII AND CHR\$ CODES

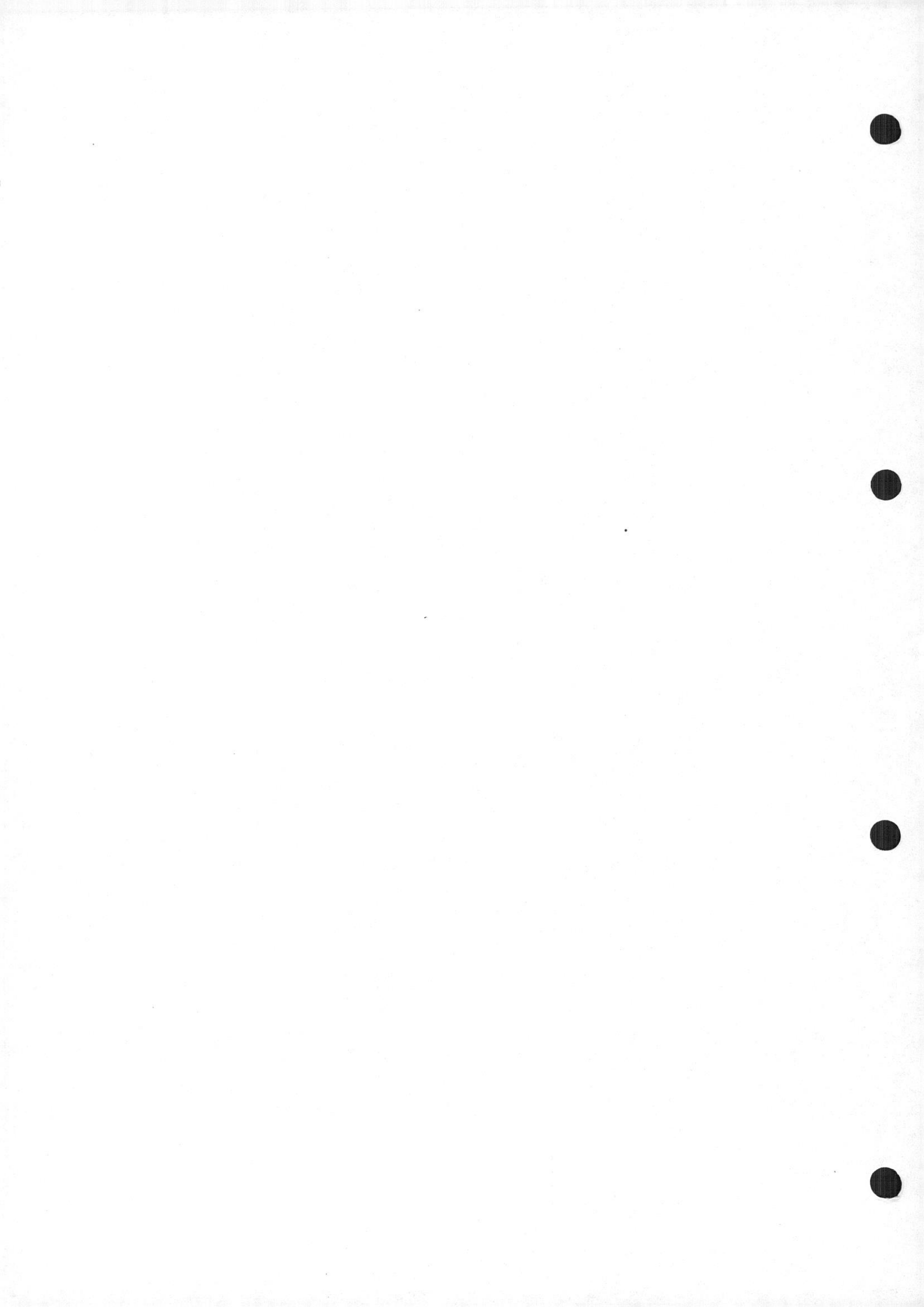
APPENDIX C - SCREEN AND COLOUR MEMORY MAPS

APPENDIX D - DERIVING MATHEMATICAL FUNCTIONS

APPENDIX E - MUSICAL NOTE TABLE

APPENDIX F - ERROR MESSAGES AND DOS ERROR MESSAGES

APPENDIX G - C16 AND PLUS/4 SCHEMATIC DIAGRAMS



## SECTION ONE

## INTRODUCTION

## 1.1 Introduction

The C16 and PLUS/4 PROGRAMMER'S GUIDE has been developed as a working tool and reference source for those who want to maximize their use of the built-in capabilities of your C16 or PLUS/4. This manual contains the information you need for your programs, it is designed so that everyone, from the beginner to the professional experienced in 6502 machine language, can find the information to develop his creative programs. The C16 and PLUS/4 PROGRAMMER'S GUIDE also shows you the capabilities, and limitations, of your C16 or PLUS/4.

This GUIDE is not designed to teach the BASIC programming language or the 6502 machine language. If you do not already have a working knowledge of BASIC and BASIC programming, COMMODORE recommends that you study the C16 or PLUS/4 USER'S GUIDE supplied with your computer. The USER'S GUIDE gives you an easy-to-read introduction to the BASIC programming language.

## 1.2 What is Included?

Section Two covers all aspects of programming in BASIC 3.5. This "BASIC Encyclopaedia" gives Commodore BASIC 3.5 language commands, statements and functions listed in alphabetical order. Included in that section is a "quick list" which contains all the words and their abbreviations.

The C16 or PLUS/4 computer has many powerful graphics features. These are also covered in Section Two.

Section Three gives information about machine code programming.

The Appendices contain technical information. Their contents are as follows:

Appendix A	Screen Display Codes
Appendix B	ASCII and CHR\$ Codes
Appendix C	Screen and Colour Memory Maps
Appendix D	Deriving Mathematical Functions
Appendix E	Musical Note Table
Appendix F	Error Messages and DOS Error Messages
Appendix G	C16 and PLUS/4 Schematic Diagrams





## SECTION TWO

### PROGRAMMING BASIC

#### 2.1 Introduction

This section provides formats, brief explanations and examples of the BASIC 3.5 commands and statements.

Commands and statements are listed in separate sections. Within each section, the commands and statements 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 they are prefixed by a line number. Many statements can be used as commands by using them in direct mode, i.e., without line numbers. If you are unsure where a term is located, refer to the BASIC Abbreviation and Reference Chart (see Section 2.8).

This chapter is organized as follows:

- \* **COMMANDS:** the commands used to work with programs, edit, store, and erase them.
- \* **STATEMENTS:** the BASIC program statements used in numbered lines of programs.
- \* **FUNCTIONS:** the string, numeric, and print functions.
- \* **VARIABLES AND OPERATORS:** the different types of variables, legal variable names, and arithmetic and logical operators.

#### 2.2 Command and Statement Format

For the sake of clarity, the commands and statements in this section are presented in standard format conventions. In most cases, there are several examples illustrating the command. The following gives an example of the format conventions used in the BASIC commands and statements in this manual:

EXAMPLE: `LOAD "program name",D0,D8`

		/
keywords	argument	additional arguments (possibly optional)

The parts of the command or statement given in upper case must be entered exactly as they appear in the format listing. Other words, such as "program name", are printed in lower case. When quote marks (") appear, usually around a program or file name, you include them in the command or statement, as in the format example.

- \* KEYWORDS appear in upper case letters. YOU MUST ENTER THESE WORDS EXACTLY AS THEY APPEAR. However, many keywords have abbreviations (see the Reference Chart).

Keywords are part of the BASIC language that your computer knows. They are the central part of a command or statement, and tell the computer what kind of action you want it to take. These words cannot be used as variable names.

- \* ARGUMENTS (also called parameters) appear in lower case letters. Arguments are the parts of a command or statement that you select; they complement keywords by providing specific information about the command or statement. For example, a keyword tells the computer to load a program, while an argument tells the computer which specific program to load and a second argument specifies which drive the program disk is in. Arguments include filenames, variables, line numbers, etc.
- \* SQUARE BRACKETS [] show OPTIONAL arguments. Select any or none of the arguments listed.
- \* ANGLE BRACKETS <> indicate that you MUST choose one of the arguments listed.
- \* VERTICAL BAR | separates items in a list of arguments when your choices are limited to those arguments listed. When the vertical bar appears in a list which is enclosed in SQUARE BRACKETS, your choices are limited to the items in the list, but you still have the option not to use any other arguments.
- \* ELLIPSIS ..., a sequence of three dots, means that an option or argument can be repeated.
- \* QUOTATION MARKS "" enclosing character strings, filenames, and other expressions. When arguments are enclosed in quotation marks in a format, you must include the quotation marks in your command or statement.
- \* PARENTHESES () When arguments are enclosed in parentheses in a format, you must include the parentheses in your command or statement. Parentheses are also required when they appear in a command or statement description.
- \* 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)

## 2.3 Basic Commands

### 2.3.1 AUTO

AUTO [line#]

Turns on the automatic line numbering feature. This eases the job of entering programs by typing the line numbers for you. As you enter each program line and press 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 lines in increments of ten

AUTO 50 automatically numbers lines in increments of fifty

AUTO turns OFF automatic line numbering

### 2.3.2 BACKUP

BACKUP Ddrive# TO Ddrive# [,ON Uunit#]

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

This command copies all the files on a diskette to another diskette on a dual drive system. You can copy onto a new diskette 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. Always BACKUP important diskettes in case the original is lost or damaged.

Because the BACKUP command also HEADERS diskettes, it destroys any information on the diskette onto which you are copying information. So if you are backing up onto a previously used diskette, make sure it contains no programs you wish to keep. See also the COPY command.

#### EXAMPLES:

BACKUP D0 TO D1                      Copies all files from the disk in  
drive 0 to the disk in drive 1

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

## 2.3.3 COLLECT

COLLECT [Ddrive#][,ON Uunit#]

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

EXAMPLE:

COLLECT D0

## 2.3.4 CONT

CONT (Continue)

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

## 2.3.5 COPY

COPY [Ddrive#,) "source file" TO [Ddrive#,) "other file" [,ON Uunit#]

COPYs a file on the disk in one drive (the source file) to the disk in the other drive on a dual disk drive, or creates a copy of a file on the same drive giving the copy a different file name.

EXAMPLES:

COPY D0,"NOON" TO D1,"NIGHT"	Copies NOON from drive 0 to drive 1, renaming it NIGHT
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 "CATS" TO "DOGS"	Copies CATS to the same drive giving it the name DOGS

## 2.3.6 DELETE

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 lines 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

## 2.3.7 DIRECTORY

DIRECTORY [Ddrive#][,Uunit#][,"filename"]

Displays a disk directory on the screen. Use <CTRL/S> to pause the display and any other key to restart the display after a pause. Use the <C=> key, the Commodore key, to slow it down. The DIRECTORY command cannot be used to print a hard copy. To do that you must LOAD the disk directory destroying the program currently in memory.

## EXAMPLES:

DIRECTORY	List all files on the disk
DIRECTORY D1,U9,"WORK"	Lists the file on disk drive unit 9 (8 is the default), drive 1, named WORK
DIRECTORY "AB*"	Lists all files starting with the letters "AB", like ABOVE, ABOARD, etc.
DIRECTORY D0,"FILE?.BAK"	The ? is a wild-card that matches any single character in that position: FILE1.BAK, FILE2.BAK, FILE3.BAK all match the string.

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

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

## 2.3.8 DLOAD

DLOAD "filename" [,Ddrive#][,Uunit#]

This command loads a program from disk into current memory. Use LOAD to load programs from tape. You must supply a program name.

## EXAMPLE:

DLOAD "DTRUCK"      Searches the disk for the program "DTRUCK" and LOADs it

DLOAD (A\$)          LOADs a program from disk whose name is the variable A\$. You get an error 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.

## 2.3.9 DSAVE

DSAVE "filename" [,Ddrive#][,Uunit#]

This command stores a program on disk. Use SAVE to store programs on tape. You must supply a program name.

## EXAMPLES:

DSAVE "DDAY"                  SAVES the program "DDAY" to disk

DSAVE (A\$)                  SAVES to disk the program whose name is in the variable A\$

DSAVE "PROG3",D0,U9      SAVES the program "PROG3" to the disk drive with a unit number of 9

### 2.3.10 HEADER

HEADER "diskname",Ddrive#[,Iid#][,ON Uunit#]

Before you can use a new disk for the first time you must format it with the HEADER command. If you want to erase an entire disk for re-use, you can use the HEADER command. This command divides the disk into sections called blocks, and creates on the disk a table of contents, called a directory or catalog. The diskname can be any name up to 16 characters long. The id number is any 2 characters. Give each disk a unique id number.

WARNING: Be careful when you HEADER a disk because the HEADER command erases all data previously stored on that disk.

Giving no id number allows you to perform a quick header. The old id number is used. You can only use the quick header method if the disk was previously formatted, since the quick header only clears out the directory rather than formatting the disk.

#### EXAMPLES:

HEADER "MYDISK",I23,D0

HEADER "THEBALL",I45,D1,U8

### 2.3.11 HELP

#### HELP

The HELP command is used when you get an error in your program. When you type HELP, the line where the error occurred is listed, with the portion containing the error displayed in flashing characters.



## 2.3.12 KEY

KEY [key#,string]

There are eight (8) function keys available to the user on the Commodore 16 and Plus/4 computers, four unshifted and four shifted. You can define what each key does when pressed. KEY without any parameter specified gives a listing displaying all the current KEY assignments. The data you assign to a key is typed out when that function key is pressed. The maximum length for all the definitions together is 128 characters. Entire commands or a series of commands can be assigned to a key. For example:

```
KEY 7,"GRAPHICØ"+CHR$(13)+"LIST"+CHR$(13)
```

causes the computer to select text mode and list your program whenever the "F7" key is depressed, in direct mode. The CHR\$(13) is the ASCII character for RETURN. Use CHR\$(34) to incorporate a double quote into a KEY string.

The keys may be redefined in a program. For example:

```
1Ø KEY 2,"TESTING"+CHR$(34):KEY3,"NO"
```

```
1Ø FORI=1TO8:KEYI,CHR$(I+132):NEXT
```

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

## 2.3.13 LIST

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

The LIST command lets you look at lines of a BASIC program that have been typed or LOADED into memory. When LIST is used alone (without any numbers following it), you get a complete LISTING of the program on your screen. This may be slowed down by pressing the <C=> key, paused by <CTRL-S> and unpaused by pressing any other key, or STOPPED by pressing the <RUN/STOP> key. If you follow the word LIST with a line number, only that line is displayed. If you type LIST with two numbers separated by a dash, all the lines from the first to the second number are shown. If you type LIST followed by a number and just a dash, it shows all the lines from that number to the end of the program. And if you type LIST, a dash, and then a number, you get all the lines from the beginning of the program to that line number. Using these variations, you can examine any portion of a program, or bring lines onto the screen for modification.



## 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

## 2.3.14 LOAD

LOAD ["filename"[,device#][,relocate flag]]

Use this command when you want to use a program stored on cassette tape or on disk. If you just type LOAD and press the <RETURN> key. Press PLAY on your cassette unit, and the computer starts looking for a program on the tape. When it finds one, the message FOUND "filename" is displayed on the screen. Press the <C=> key to LOAD that program. If you do not press that key, after a brief interval the computer resumes searching on the tape. Once the program is LOADED, you can RUN, LIST, or change it.

You can also type the word LOAD followed by a program name, which is either a name in quotes or a string variable. The name may be followed by a comma (outside of the quotes) and a number, or numeric variable, which is the number of the device where the program is stored, i.e., disk or tape. If there is no number given, your computer assumes device number 1, which is the cassette unit.

The other device commonly used with the LOAD command is the disk drive, which is device number 8.

## EXAMPLES:

LOAD                    Reads in the next program on tape

LOAD "BASES"           Searches tape for a program called BASES, and LOADS it if it is found

LOAD A\$                Looks for a program whose name is in the variable called A\$

LOAD "BRIDGES",8      Looks for the program called BRIDGES on the disk drive, and LOADS it if found

The LOAD command can be used within a BASIC program to find and RUN the next program on a tape. This is called chaining.

The RELOCATE FLAG determines where in memory a program is loaded. A relocate flag of 0 tells the computer 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 is generally used only when LOADING machine language programs.

### 2.3.15 NEW

#### NEW

This command erases the program in memory and clears any variables that have been used. Be careful when you use this command. Unless the program was stored on disk or cassette, it is lost until you type it in again.

The NEW command can also be used as a statement in a BASIC program. When this command is executed, the program is erased and execution stops.

### 2.3.16 RENAME

RENAME [Ddrive#,"old name"TO"new name"[,Uunit#]

Used to rename a file on a diskette.

#### EXAMPLE:

RENAME D0,"ASSET" TO "LIABILITY"      Changes the name of the  
file from ASSET to  
LIABILITY

### 2.3.17 RENUMBER

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 you to renumber a portion of your program. It defaults to the first line of your program.

This command can only be executed from direct mode.

## EXAMPLES:

RENUMBER 20,20,1      Renumbers the program, starting at line 1. Line 1 becomes line 20, and other lines are numbered in increments of 20

RENUMBER,,65          Renumbers in increments of 10, starting at line 65. Line 65 becomes line 10, unless there are already lines numbered 10-64, in which case the command is not carried out

## 2.3.18 RUN

RUN [line#]

Once a program has been typed into memory or LOADED, the RUN command causes it to be executed. This command clears all variables before starting program execution. If there is no number following this command, the computer starts at the lowest numbered program line. If there is a number following the RUN command, execution starts at that line. RUN may be used within a program.

## EXAMPLES:

RUN                    Starts program working from lowest line number

RUN 100                Starts program from line 100

## 2.3.19 SAVE

SAVE ["filename"[,device#[,EOT flag]]]

This command stores on cassette tape or disk a program currently in the computer's memory. If you just type the word SAVE and press <RETURN>, your computer attempts to store the program on the cassette. It has no way of checking if there is already a program on the tape in that location, so be careful with your tapes. If you type the SAVE command followed by a name in quotes or a string variable name, the computer gives the program that name when it SAVES it. In this way the program is more easily located and retrieved in the future.

If you want to specify a device number for the SAVE, place a comma after the quotes following the name. Then type a number or numeric variable. Device number 1 is the cassette unit, and number 8 is the disk. After the number on a tape command, there can be a comma and a second number, which is between 0 and 3. If this number is 2, the computer puts an END-OF-TAPE marker, i.e. EOT flag, after your program. If you are trying to LOAD a program and the computer finds one of these markers rather than the required program, you get a FILE NOT FOUND ERROR.

## EXAMPLES:

SAVE	Stores program to tape without a name
SAVE "MONEY"	Stores on tape with the name MONEY
SAVE A\$	Stores on tape with name in variable A\$
SAVE "YOURSELF",8	Stores on disk with name YOURSELF
SAVE "GAME",1,2	Stores on tape with name GAME and places an END-OF-TAPE marker after the program

## 2.3.20 SCRATCH

SCRATCH "filename" [,Ddrive#] [,Uunit#]

Deletes a file from the disk directory. As a precaution, you are asked "Are you sure?" before the operation is carried out. Type a Y to perform the SCRATCH or type N to cancel the operation. Use this command to erase unwanted files in order to create more space on the disk.

## EXAMPLE:

SCRATCH "MY BACK",D0 Erases the file MY BACK from the disk in drive 0

## 2.3.21 VERIFY

VERIFY "filename" [,device#] [,relocate flag]

This command checks a program on tape or disk against the one in memory. Use VERIFY after saving a program to ensure that nothing went wrong with the SAVE. This command can also be used to position a tape so that your computer resumes writing following the end of the last program on the tape. To do this, tell the computer to VERIFY the name of the last program on the tape. It does so, and tells you that the programs do not match. The tape is then positioned for storing the next program erasing an old one.

VERIFY with no arguments checks the next program on tape, regardless of its name, against the program currently in memory. VERIFY followed by a program name, in quotes, or a string variable, searches the tape for that program and then checks it against the one in memory. VERIFY followed by a name, a comma, and a number checks the program on that device number, 1 for tape, 8 for disk. The relocate flag is the same as in the LOAD command.

## EXAMPLE:

VERIFY Checks the next program on the tape

VERIFY "REALITY" Searches for REALITY on tape, checks against memory

VERIFY "ME",8,1 Searches for ME on disk, then checks

## 2.4 Basic Statements

## 2.4.1 BOX

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

colour source Colour source (0-3), default is 1 (foreground colour)

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 colour (0=off, 1=on), default is 0

This command allows you 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 centre 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, i.e. a diamond

BOX,30,90,,45,1 Draws a filled, rotated polygon

## 2.4.2 CHAR

CHAR [colour source#],x,y,"string"[,reverse flag]

colour source    Colour source (0-3)  
 x                Character column (0-39)  
 y                Character row (0-24)  
 "string"        Text to be printed  
 reverse         Reverse field flag (0=off, 1=on)

Text, i.e. alphanumeric strings, can be displayed on any screen at a given location using the CHAR command. Character data is read from the character ROM area. You supply the x and y coordinates of the starting position and the text string you wish to display. Colour source and reverse imaging are optional.

The string is continued on the next line if it prints past the right hand edge of the screen. When used in TEXT mode, the string printed by the CHAR command works in the same way that a PRINT string works, including reverse field, cursors, flash on/off, etc. These control functions inside the string do not work when the CHAR command is used to display text in a GRAPHIC mode.

## 2.4.3 CIRCLE

CIRCLE [cs],[a,b],xr[,yr][,[sa][,[ea][,[angle][,[inc]]]]]

cs            Colour source (0-3)  
 a,b          Centre 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

The CIRCLE command can be used to draw a circle, ellipse, arc, triangle or an octagon. The final coordinate is on the circumference of the circle at the ending arc angle. Any rotation is about the centre. Arcs are drawn clockwise from the starting angle 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 an oval
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

## 2.4.4 CLOSE

CLOSE file#

This command completes and closes any previously OPENned files. The number following the CLOSE command is the file number to be closed.

## EXAMPLE:

CLOSE 2 Logical file 2 is closed

## 2.4.5 CLR

This command erases any variables in memory, but leaves the program itself intact. This command is automatically executed when a RUN or NEW command is given, or any editing is performed.

## 2.4.6 CMD

CMD file#[,write list]

CMD sends the output which usually goes to the screen, e.g. PRINT statements, LISTS, to another device. Note that this does not include POKEs onto the screen. The other device can 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 referring to the file.



## EXAMPLE:

OPEN 1,4      OPENS device #4, i.e. the printer  
 CMD 1        All normal output now goes to the printer  
 LIST         The LISTing goes to the printer, not the screen.  
               This includes the word READY.  
 PRINT#1     Set output back to the screen  
 CLOSE1      Close the file

## 2.4.7 COLOR

COLOR source#,colour#[,luminance#]

Assigns a colour to one of the 5 colour sources:

Number	Source
0	background
1	foreground
2	multicolour 1
3	multicolour 2
4	border

The colours you can use are in the range 1-16. These are from your keyboard colour keys, i.e. 1 is black, 2 is white, 9 is orange, etc. As an option, you can include the luminance level 0-7, with 0 being lowest and 7 being highest. Luminance defaults to 7. This lets you select from eight levels of brightness for any colour except black.

## 2.4.8 DATA

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 words, and are separated by commas. Words need not be enclosed in quotation marks, unless they contain a SPACE, colon, and/or comma. If two commas have nothing between them, the value is READ as a zero for a number, or an empty string depending on the type of variable that you are READING data into. The DATA statement must be part of a program, otherwise it is not recognized. See also the RESTORE statement, which allows your computer to reread data.

## EXAMPLE:

DATA 100,200,FRED,"WILMA",,3,14,ABC123



## 2.4.9 DEF FN

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

This command allows you to define a complex calculation as a function. In the case of a long formula that is used several times within a program, this command can save a lot of space.

The name you give the numeric function begins with the letters FN, followed by any legal numeric variable name. First define the function by using the DEF statement followed by the name you have given the function. Following the name is a set of parentheses () enclosing a numeric variable. In the following example this is X. After the parentheses is an equal sign, followed by the formula you wish to define. You can "call" the formula, 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.

NOTE: DEF FN can only be used with standard numeric functions, not integer or string functions.

## 2.4.10 DIM

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

Before you can use an array of variables with more than 11 elements, that array must first be DIMensioned. The DIM statement is followed by the name of the array, which may be any legal variable name. After the variable name, enclosed in parentheses, you put the number, or numeric variable, of elements in each dimension. An array with more than one dimension is called a matrix. You may use any number of dimensions. Note that the whole list of variables you are creating takes up space in memory, and it is easy to run out of memory. To calculate the number of variables created with each DIM statement, multiply the total number of elements in each dimension of the array together, remembering that each array starts with element 0.

NOTE: Integer, i.e. single-digit, arrays take up 2/5ths of the space of floating point arrays.

## EXAMPLE:

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

You can DIMension more than one array with a DIM statement by separating the arrays with commas. If a program attempts to execute a DIM statement for any array more than once, a re'DIMed array error message is displayed. It is good programming practice to place DIM statements near the beginning of a program.

## 2.4.11 DO (LOOP) WHILE (UNTIL EXIT)

```
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. An example of a boolean argument is  $A=1$  or  $H \geq 57$ . 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 (see Sections 2.4.14, and 2.4.28).

If the UNTIL parameter is used, the program continues looping until the boolean argument is satisfied, i.e. becomes TRUE. The WHILE parameter is basically the opposite of the UNTIL parameter, i.e. the program continues looping as long as the boolean argument is TRUE.

## EXAMPLE:

```
DO UNTIL X=0 OR X=1
REM
LOOP
DO WHILE A$="":GET A$:LOOP
```

## 2.4.12 DRAW

DRAW [colour source#][,a1,b1][,TO a2,b2,][...]

With this command you can draw individual dots, lines, and shapes. You supply colour 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

## 2.4.13 END

END

When a program encounters an END statement, it stops RUNNING immediately. You may use the CONT command to re-start the program at the statement following the END statement (see Section 2.3.4).

## 2.4.14 FOR ... TO ... STEP

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

This statement is used in conjunction with the NEXT statement to set up a section of the program that repeats for a set number of times. This is useful if you want to pause a program, or perform an operation, e.g. printing, a certain number of times.

The loop variable 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 command NEXT, it adds the STEP increment to the value of the loop variable. The default value of the STEP increment is 1. The program then checks the loop variable to see if it is higher than the end of the loop value. If it is not higher, the next line executed is the statement 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. A STEP value can be positive or negative. See also the NEXT statement (see Section 2.4.28).

## EXAMPLE:

```

10 FOR L=1TO20
20 PRINT L
30 NEXT L
40 PRINT "BLACKJACK! L="L

```

This program prints the numbers from one to twenty on the screen, followed by the message BLACKJACK! L=21.

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 to the loop variable each time. This allows you to, for example, count backwards, or by fractions.

You can set up loops inside one another, i.e. nested loops. Note, you must ensure you nest loops so that the last loop to start is the first one to end.

## EXAMPLE OF NESTED LOOPS:

```

10 FOR L=1TO100
20 FOR A=5TO11STEP2   This FOR ... NEXT loop is "nested"
                      inside the larger one
30 NEXT A
40 NEXT L

```

## 2.4.15 GET

## GET variable list

The GET statement is a way to get data from the keyboard one character at a time. When it is executed, the character typed is received. If no character is typed, then a null, i.e. empty, character is returned, and the program continues without waiting for a key. The <RETURN> key is not pressed as that key can be received with a GET.

The word GET is followed by a variable name, usually a string variable. If a numeric variable 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, which waits for a key to be struck to continue. The GETKEY statement (see Section 2.4.16) could also be used in this case. This command can only be executed within a program.

## EXAMPLE:

```
10 GET A$<>"A"THEN10
```

This line causes the program to wait for the "A" key to be pressed before continuing.

## 2.4.16 GETKEY

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 command 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.

## 2.4.17 GET#

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 command can only be executed within a program.

## EXAMPLE:

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

## 2.4.18 GOSUB

GOSUB line#

This statement is similar to the GOTO statement, except that the program jumps back to the statement immediately following the GOSUB when a line with a RETURN statement is encountered. The target of a GOSUB statement is called a subroutine. A subroutine is useful if a particular routine is used at several different places in the program. Instead of duplicating the section of program, you can set it up as a subroutine, and GOSUB to it from the different parts of the program. See also the RETURN statement (see Section 2.4.41).

## EXAMPLE:

```

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

```

## 2.4.19 GOTO or GO TO

GOTO line#

When a GOTO statement is encountered, the program execution jumps to the line number specified. When used in direct mode, GOTO line# allows you to start execution of the program at the given line number without clearing the variables.

## EXAMPLE:

```

10 PRINT "REPETITION IS THE MOTHER OF LEARNING"
20 GOTO 10

```

The GOTO in line 20 causes line 10 to be executed until the <RUN/STOP> key is pressed.

## 2.4.20 GRAPHIC

GRAPHIC mode[,clear option]

This statement puts your computer into one of the 5 graphic modes:

Mode	Description
0	normal text
1	high-resolution graphics
2	high-resolution graphics, split screen
3	multicolour graphics
4	multicolour graphics, split screen

When executed, GRAPHIC 1-4 allocates a 10K bit-mapped area of memory for graphics, and moves the BASIC text area below the hi-res area. This area remains allocated even if you return to TEXT mode (GRAPHIC 0). If 1 is given as the second argument in the GRAPHIC statement, the screen is also cleared.

## EXAMPLES:

```

GRAPHIC 1,1      Selects hi-res graphic mode and clears the
                  screen

GRAPHIC 4,0      Selects multi-colour graphics with an area
                  for text, without clearing the screen

```

## 2.4.21 GRAPHIC CLR

## GRAPHIC CLR

This is a form of the GRAPHIC statement. This statement clears the 10K of memory allocated to the graphic area, and that memory space becomes available for BASIC once again.

## 2.4.22 IF ... THEN ... ELSE

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

IF ... THEN allows you to analyse the 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 statement may be any BASIC statement. If the expression is false, the program goes directly to the next line, unless an ELSE clause is present. The expression being evaluated may be a variable or formula, in which case it is considered true if non-zero, and false if zero. In most cases, it is an expression involving relational operators, i.e. =, <, >, <=, >=, <>, AND, OR, NOT.

The ELSE clause, if present, must be in the same line as the IF-THEN clause. When an ELSE clause is present, it is executed when the IF expression is FALSE.

## 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 is not. If X is not greater than 0, the ELSE clause is executed and the THEN clause is not.

## 2.4.23 INPUT

INPUT ["prompt string";]variable list

The INPUT statement allows the computer to ask for data and place it into a variable or variables. When an INPUT statement is encountered, the program stops, prints a question mark, i.e. ?, on the screen, and waits for the user to type the answer and press 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 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. If several variables are 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 you press the <RETURN> key without INPUTing values, the INPUT variables retain the values previously held for those variables. This statement can only be executed within a program.

EXAMPLE:

```
10 INPUT"WHAT'S YOUR NAME";A$
20 INPUT"AND YOUR FAVOURITE COLOUR";B$
30 INPUT"WHAT'S THE AIR SPEED OF A SWALLOW";A
```

#### 2.4.24 INPUT#

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 command can only be used in program mode.

EXAMPLE:

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

#### 2.4.25 LET

[LET] variable=expression

The word LET is hardly ever used in programs, since it is not necessary, but the statement itself is the heart of all BASIC programs. Whenever a variable is defined or given a value, LET is always implied. The variable name which is to receive the result of a calculation is on the left side of the equal sign, and the number or a formula is on the right side.

EXAMPLE:

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

LET is implied (but not necessary) in lines 20, 30 and 40.



## 2.4.26 LOCATE

LOCATE x-coordinate, y-coordinate

The LOCATE command lets you put the pixel cursor (PC) anywhere on the screen. The PC is the current location of the starting point of the next drawing. Unlike the regular cursor, you can't see the PC, but you can move it with the LOCATE command. For example:

```
LOCATE 160,100
```

positions the PC in the centre of the high resolution screen. You do not see anything until you use one the graphics commands to draw something. You can find out where the PC is at any time by using the RDOT(0) function to get the X-coordinate and RDOT(1) to get the Y-coordinate. The colour source of the dot at the PC can be found by PRINTing RDOT(2).

NOTE: In all drawing commands where a colour option is available, you may select a value from 0 to 3, corresponding to the background, foreground, multicolour 1, or multicolour 2 as the colour source.

## 2.4.27 MONITOR

MONITOR

This command takes you out of BASIC into the built-in machine language monitor program. The monitor lets you develop, debug, and execute machine language programs more easily than in BASIC. See the section on monitor commands for more information. When in the monitor, typing an "X" and pressing <RETURN> returns you to BASIC.

## 2.4.28 NEXT

NEXT [variable,...,variable]

The NEXT statement is used with the FOR statement. When the computer encounters a NEXT statement, it goes back to the corresponding FOR statement and checks the loop variable, (see Section 2.4.14 for more detail). If the loop is finished, execution proceeds with the statement after the NEXT statement. The word NEXT may be followed by a variable name, a list of variable names separated by commas, or no variable names. If there are no names listed, the last loop started is the one being completed. If the variables are given, they are completed in order from left to right.

EXAMPLE:

```
10 FOR L=1 TO 10:NEXT
20 FOR L=1 TO 10:NEXT L
30 FOR L=1 TO 10:FOR M=1 TO 10:NEXT M,L
```

## 2.4.29 ON

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

This command makes 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, then a list of line numbers separated by commas. If the result of the calculation of the formula, i.e. expression, is 1, the first line number in the list is executed. If the result is 2, the second line number is executed, and so on. If the result is 0, or larger than the number of line numbers on the list, the next line executed is the statement following the ON statement. If the number is negative, an ILLEGAL QUANTITY ERROR results.

## EXAMPLE:

```
10 INPUT X:IF X<0 THEN 10
20 ON X GOTO 50,30,30,70
25 PRINT"FELL THROUGH":GOTO 10
30 PRINT"TOO HIGH":GOTO 10
50 PRINT"TOO LOW":GOTO 10
70 END
```

When X=1, ON sends control to the first line number in the list, i.e. 50. When X=2, ON sends control to the second line, i.e. 30, etc. When X is greater than 4, execution "falls through" to line 25.

## 2.4.30 OPEN

OPEN file#[,device#[,secondary address[,"filename,type,mode"]]]

The OPEN statement allows your computer to access devices such as the Datassette recorder, the disk unit, a printer, or even the monitor screen. The word OPEN is followed by a logical file number, which is the number to which all other BASIC statements refer. This number is from 1 to 255. There is normally a second number after the first called the device number. Device number 0 is the computer keyboard, 3 is the screen, 1 is the Datassette recorder, 4 is the printer, 8 is usually the disk unit. A zero, i.e. 0, may be included in front of the device number digit, e.g. 08 for 8. The default value is 1. COMMODORE recommend that you use the same file number as the device number.

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 an end-of-tape marker at the end. In the case of the disk unit, the number refers to the channel number. With the printer, the secondary addresses are used to set the mode of the printer. See your printer manual for more information on secondary addresses. There may also be a string following the third number, which could be a command to the disk drive 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,"UP"	OPENS the cassette for reading, file to be searched for is named UP
OPEN 4,4	OPENS a channel to use the printer
OPEN 15,8,15	OPENS the command channel on the disk
5 OPEN 8,8,2,"TEST,SEQ,WRITE"	creates a sequential disk file for writing

See also: CLOSE, CMD, GET#, INPUT# and PRINT# statements and system variables ST, DS and DS\$ (see Sections 2.4.4, 2.4.6, 2.4.17, 2.4.24, 2.4.34, and 2.7.1.2).

## 2.4.31 PAINT

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

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

The PAINT command lets you fill an area with colour. It fills in the area around the specified point until a boundary of the same colour or any non-background colour, depending on which mode you have 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 colour source you name, or any non-background colour when mode 1 is used, the area to be PAINTed does not change colour.

EXAMPLE:

```
10 CIRCLE ,160,100,65,50  draws outline of circle
20 PAINT ,160,100        fills in the circle with colour
```

### 2.4.32 POKE

POKE address,value

The POKE command allows you to change a value in the computer's RAM, and lets you modify many of the Input/Output registers. POKE is always followed by two numbers or equations. The first number, i.e. the address, is a location inside the computer's memory. This can have any value from 0 to 65535. The second number is a value from 0 to 255. This is placed in the location given by the address, replacing any value currently in that location. This command can be used to control anything displayed on the screen, e.g. placing a character at a particular location and changing the colour at that location.

EXAMPLE:

```
10 POKE 16000,8      Sets the value at location 16000 to 8
20 POKE 16*1000,27   Sets the value at location 16000 to 27
```

NOTE: PEEK, a function related to POKE, is listed under FUNCTIONS (see Section 2.6.1.12).

### 2.4.33 PRINT

PRINT printlist

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 subtleties to be mastered here as well. The word PRINT can be followed by any combinations items in the printlist. The printlist is as follows:

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

The characters inside 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 any spaces. Either of these punctuation marks can be used as the last symbol in the statement. This results in the next PRINT statement acting as if it is continuing the current PRINT statement.

## EXAMPLES:

	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() and TAB() FUNCTIONS (see Sections 2.6.3.2, 2.6.3.3, and 2.6.3.4).

## 2.4.34 PRINT#

PRINT# file#,printlist

The PRINT# statement is similar to the PRINT statement, except that while PRINT is used to display data on the screen, PRINT# is used to send data to a device or file. 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 sends 10 spaces to most printers and can be used as a separator for disk files. Some devices may not work with TAB and SPC. The semicolon acts in the same manner for spacing as it does in the PRINT statement.

## EXAMPLE:

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

## 2.4.35 PRINT USING

```
PRINT [#filenumber,]USING format list;printlist
```

These statements allow you to define the format of string and numeric items you wish to print to the screen, printer, or another device. Put the format you require in quotes. This is the format list. Then add a semicolon and a list of items you want printed in the format, this is the print list. The list can be variables or the actual values you require printed. For example:

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

When you RUN this program, line 10 prints out:

```
$13.25 $32.00 $*****
```

NOTE: it prints \*\*\*\*\* instead of the Y value because Y has 5 digits and does not conform to the format list, as explained below in this section.

Line 20 prints this:

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

CHARACTER	NUMERIC	STRING
Hash sign (#)	X	X
Plus (+)	X	
Minus (-)	X	
Decimal point (.)	X	
Comma (,)	X	
Dollar sign (\$)	X	
Four carets (^^^^)	X	
Equal sign (=)		X
Greater than sign		X



The hash sign (#) reserves room for a single character in the output field. If the data item contains more characters than the number of # in your format field, the following occurs:

1. For a numeric item, the entire field is filled with asterisks (\*). No numbers are printed.

For example:

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

For these values of X, the format displays:

```
X=12.34      12
```

```
X=567.89    568
```

```
X=123456    ****
```

2. For a STRING item, the string data is truncated at the bounds of the field. Only as many characters are printed as there are hash 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 you use a minus sign and the number is positive, a blank is printed in the character position indicated by the minus sign.

If you use neither a plus nor minus sign in your 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 you can print one character more if the number is positive. If there are too many digits to fit into the field specified by the # 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. You can only have one decimal point in any format field. If you do not specify a decimal point in your format field, the value is rounded to the nearest integer and printed without any decimal places.

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



A comma (,) allows you to place commas in numeric fields. The position of the comma in the format list indicates the position of the comma in the 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 # must precede the first comma in a field.

If you specify commas 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
##.#+	-.01	0.01-	Leading zero added
##.#-	1	1.0	Trailing zero added
####	-100.5	-101	Rounded to no decimal places
####	-1000	****	Overflow because four digits and minus sign cannot fit in field
###	10	10.	Decimal point added
#\$##	1	\$1	Leading \$ sign

A dollar sign (\$) shows that a dollar sign will be printed in the number. If you want the dollar sign to float (always be placed before the number), you must specify at least one # before the dollar sign. If you specify a dollar sign without a leading #, the dollar sign is printed in the position shown in the format field.

If you specify commas and/or a plus or minus sign in a format field with a dollar sign, your program prints a comma or sign before the dollar sign.

The four up arrows or carets (^^^^) are used to specify that the number is to be printed in E+ format. You must use # in addition to the ^^^^ to specify the field width and the ^^^^ must appear after the #.

You must specify four carets (^^^^) when you want to print a number in E-format, i.e. scientific notation. If you specify more than one but fewer than four carets, you get a syntax error. If you specify more than four carets only the first four are used. The fifth, and subsequent, carets are interpreted literally as no text symbols.

An equal sign (=) is used to centre a string in the field. You specify the field width by the number of characters in the format field, the = is included in this count. If the string contains fewer characters than the field width, the string is centred in the field. If the string contains more characters than can fit in the field, the right-most characters are truncated and the string fills the entire field.

A greater than sign (>) is used to right justify a string in a field. You specify the field width by the number of characters in the format field. The = is included in this count. If the string contains fewer characters than the field width, the string is right justified in the field. If the string contains more characters than can fit into the field, the right-most characters are truncated and the string fills the entire field.

#### 2.4.36 PUDEF

PUDEF "1 through 4 characters"

PUDEF lets you redefine up to 4 symbols in the PRINT USING statement. You can change blanks, commas, decimal points and dollar signs 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 space. Place a new character here when you want another character to appear in place of spaces.

Position 2 is the comma character. Default is a comma.

Position 3 is the decimal point. Default is a decimal point.

Position 4 is the dollar sign. Default is a dollar sign.

#### EXAMPLES:

10 PUDEF "*"	PRINTs * in the place of blanks
20 PUDEF "&"	PRINTs & in place of commas
30 PUDEF ".,"	PRINTs decimal points in place of commas, and commas in place of decimal points
40 PUDEF ".,P"	PRINTs English pound sign in place of \$, decimal points in place of commas, and commas in place of decimal points

#### 2.4.37 READ

READ variable list

This statement is used to place information contained in DATA statements into the variables in the variable list. This allows the program to manipulate the data or perform calculations on it. 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. This produces an ERROR message.

#### EXAMPLE:

```
READ A$,G$,Y
DATA XXX,YYY,19
```

## 2.4.38 REM

REM message

The REMark statement allows notes and comments to be included in the program without affecting the operation of the program. Note that this statement adds to the program's length and, therefore, slows it down. It may be followed by any text, although use of graphic characters may produce strange results.

EXAMPLE:

```
10 NEXT X:REM THIS LINE IS UNNECESSARY
```

## 2.4.39 RESTORE

RESTORE [line#]

This command resets the pointer to the first item in a DATA statement list. This allows you to re-READ the information in a DATA statement(s). If a [line#] follows the RESTORE statement, the pointer is set to that line. Otherwise the pointer is reset to the first DATA statement in the program.

EXAMPLE:

```
RESTORE 200
```

## 2.4.40 RESUME

RESUME [line# | NEXT]

This is used to return to execution after TRAPPING an error (see Section 2.4.48). 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 statement containing the error. RESUME line# will GOTO the specific line and begin execution there.

## 2.4.41 RETURN

RETURN

This statement is always used with the GOSUB statement (see Section 2.4.18). 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 displayed, and program execution is stopped.



## 2.4.45 SSHAPE/GSHAPE

SSHAPE and GSHAPE are used to save and restore rectangular areas of multicolour 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 in which data is saved
a1,b1	Corner coordinate (scaled)
a2,b2	Corner coordinate opposite (a1,b1) (default is the PC)

Because BASIC limits string lengths to 255 characters, the size of the area you may save is limited. The string size required can be calculated using one of the following (unscaled) formulae:

$$L(\text{mcm}) = \text{INT}((\text{ABS}(a1-a2)+1)/4+.99)*(\text{ABS}(b1-b2)+1)+4$$

$$L(\text{h-r}) = \text{INT}((\text{ABS}(a1-a2)+1)/8+.99)*(\text{ABS}(b1-b2)+1)+4$$

(mcm) refers to multi-colour mode; (h-r) is high resolution

The shape is saved row by row. The last four bytes of the string contain the column and row lengths less one, i.e.  $\text{ABS}(a1-a2)$  in low/high byte format. If scaled, divide the lengths by 3.2 (X) and 5.12 (Y).

The command to display a saved shape on any area of the screen is:

GSHAPE string variable name [, [a,b] [,mode]]

string	Contains shape to be drawn
a,b	Top left coordinate of the position where the shape is to be drawn (scaled - 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

## EXAMPLES:

SSHAPE "VARIABLE\$",0,0 Saves screen area from the upper left corner to where the cursor is positioned. The saved area is given the name VARIABLE\$

GSHAPE "VARIABLE\$,,,1 Displays VARIABLE\$ shape with background and foreground colours reversed, with the top left of the shape positioned at the cursor

#### 2.4.46 STOP

STOP

This statement halts the program. A message, BREAK IN LINE #, where # is the line number containing the STOP, is displayed. The CONT command can be used to re-start the program at the statement following the STOP command. This statement is usually used while debugging a program.

#### 2.4.47 SYS

SYS address

The word SYS is followed by a decimal number or numeric variable in the range 0 to 65535. The program begins executing the machine language program starting at that memory location. This is similar to the USR function except that SYS does not pass a parameter to the machine language program. See Chapter 4 for more information about machine language programs.

#### 2.4.48 TRAP

TRAP [line#]

When turned on, TRAP intercepts all error conditions including the <RUN/STOP> key, except "UNDEF'D STATEMENT ERROR". In the event of an execution error, the error flag is set, and execution is transferred to the line number named in the TRAP statement. The line number in which the error occurred can be found by using the system variable EL (see Section 2.7.1.2). 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# argument turns off error TRAPPING.

#### 2.4.49 TRON

TRON

TRON is used in program debugging. This statement begins trace mode. When you are in trace mode, the line number of that statement is printed as each statement is executed.



#### 2.4.50 TROFF

TROFF

This statement turns trace mode, i.e. TRON, off.

#### 2.4.51 VOL

VOL volume level

Sets the current VOLUME level for SOUND commands. VOLUME may be set from 0 to 8, where 8 is maximum volume, and 0 is off. VOL affects both voices.

#### 2.4.52 WAIT

WAIT address,value 1 [,value 2]

The WAIT statement is used to halt the program until the contents of a location in memory changes in a specific way. The address must be in the range from 0 to 65535. Value 1 and value 2 must be in the range from 0 to 255.

The content of the memory location is first exclusive-ORed with value 2 (if present), and then logically ANDed with value 1. If the result is zero, the program checks the memory location again. When the result is non-zero, the program continues with the next statement.

### 2.5 Additional Graphic Statement Information

There are some concepts that apply to all of the bit map graphics statements. First is the concept of the Pixel Cursor (PC). The PC is similar to the cursor in text mode, it is the position where the next dot is to be drawn. Unlike the text cursor, the PC is invisible. All drawing commands use the PC. In addition, the locate command allows you to reposition the PC without drawing anything.





### 2.6.1.3 ATN(X) (arctangent)

Returns the angle, in radians, whose tangent is X.

### 2.6.1.4 COS(X) (cosine)

Returns the value of the cosine of (X), where X is an angle measured in radians.

### 2.6.1.5 DEC (hexadecimal-string)

Returns decimal value of hexadecimal-string  
( $\emptyset$ <hexadecimal-string<FFFF)

EXAMPLE:

```
N=DEC("F4")
```

### 2.6.1.6 EXP(X)

Returns the value of the mathematical constant e (2.71828183) raised to the power of X.

### 2.6.1.7 FNxx(X)

Returns the value of the user-defined function xx created in a DEF FNxx statement.

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

Returns position of string 2 in string 1 at or after the [starting position]. The starting position defaults to the beginning of string 2. If no match is found, a value of  $\emptyset$  is returned.

EXAMPLE:

```
PRINT INSTR("THE CAT IN THE HAT","CAT")
```

the result is 5, because CAT starts at the fifth character in string 1.

## 2.6.1.9 INT(X) (integer)

Returns the integer portion of X, with all decimal places to the right of the decimal point removed. The result is always less than or equal to X. Thus, a negative number with decimal places becomes the integer less than its current value, e.g. INT(-4.5)=-5.

If the INT function is to be used for rounding off, the form is INT(X+.5) or INT(X-.5).

EXAMPLE:

X=INT(X\*100+.5)/100 rounds to the next highest number

## 2.6.1.10 JOY(n)

When n = 1 position of joystick #1  
 n = 2 position of joystick #2

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

Fire = 128 +		UP	
		1	
		8    2	
	LEFT	7    0    3	RIGHT
		6    4	
		5	
		DOWN	

EXAMPLE:

JOY(2) with a value of 135 fires joystick #2 to the the left

## 2.6.1.11 LOG(X) (logarithm)

This function returns the natural log of X. The natural log is log to the base e (see EXP(X), Section 2.6.1.6). To convert to log base 10, divide by LOG(10).

## 2.6.1.12 PEEK(X)

This function gives the contents of memory location X, where X is located in the range of 0 to 65535, returning a result from 0 to 255. PEEK is often used in conjunction with the POKE statement.

### 2.6.1.13 RCLR(N)

Returns current colour assigned to source N, where N is in the range  $0 \leq N \leq 4$ .

0=background, 1=foreground, 2=multicolour 1, 3=multicolour 2, 4=border

### 2.6.1.14 RDOT(N)

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

N = 0 for XPOS  
1 for YPOS  
2 colour source

### 2.6.1.15 RGB(X)

Returns current graphic mode (X is a dummy argument and can be any value).

### 2.6.1.16 RLUM(N)

Returns current luminance level assigned to colour source N.

### 2.6.1.17 RND(X) (random number)

This function returns a random number between 0 and 1. The X is a dummy argument and can be any value. This is useful in games, to simulate dice rolls and other elements of chance. It is also used in some statistical applications. The first random number should be generated by the formula `RND(-TI)`, to give a different random number each time the program is RUN. After this, the number in X should be a 1, or any positive number, (X represents the seed, or what the Random number is based on). If X is zero, RND is re-seeded from the hardware clock every time RND is used. A negative value for X seeds the random number generator using X and gives a random number sequence. The use of 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.

To simulate the rolling of a die, use the formula  $\text{INT}(\text{RND}(1)*6+1)$ . First, a random number from 0-1 is multiplied by 6, this expands the range to 0-6, i.e.  $0 < n < 6$ . Then 1 is added, making the range  $1 \leq n < 7$ . The INT function chops off the decimal places, leaving the result as a digit from 1 to 6.

To simulate 2 dice, add together two of the numbers obtained using the above formula.

EXAMPLE:

100	X=INT(RND(1)*6)+INT(RND(1)*6)+2	Simulates 2 dice
110	X=INT(RND(1)*1000)+1	Number from 1-1000
120	X=INT(RND(1)*150)+100	Number from 100-249

#### 2.6.1.18 SGN(X) (sign)

This function returns the sign, i.e. positive, negative, or zero, of X. The result is +1 if positive, 0 if zero, or -1 if negative.

#### 2.6.1.19 SIN(X) (sine)

This is the trigonometric sine function. The result is the sine of X, where X is an angle in radians.

#### 2.6.1.20 SQR(X) (square root)

This function returns the square root of X, where X is a positive number or 0. If X is negative, an ILLEGAL QUANTITY ERROR results.

#### 2.6.1.21 TAN(X) (tangent)

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

#### 2.6.1.22 USR(X)

When this function is used, the program jumps to a machine language program whose starting point is contained in memory locations 1281 and 1282. 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 you to exchange a variable between machine code and BASIC. See Chapter 4 for more information on the USR function.

## 2.6.1.23 VAL(X\$)

This function converts the string X\$ into a number, and is essentially the inverse operation from STR\$. The string is examined starting at the left-most character, converting only characters which are in recognizable number format. If your computer finds any illegal characters, i.e. which it does not recognize as being in number format, it converts only the portion of the string up to that character.

## EXAMPLE:

```

10 X=VAL("123.456")      X=123.456
20 X=VAL("3E03")        X=3000
30 X=VAL("12A13B")      X=12
40 X=VAL("RIU017*")     X=0
50 X=VAL("-1.23.23.23") X=-1.23
60 A$="123":X=VAL(A$)   X=123

```

NOTE: 3E03 is scientific notation for 3000.

## 2.6.2 String Functions

String functions differ from numeric functions in that they return characters, graphics or numbers from a string instead of a number. A string is a group of characters enclosed in quotation marks.

## 2.6.2.1 CHR\$(X)

This function returns a string character whose ASCII code is X.

## 2.6.2.2 ERR\$(N)

Returns the string describing error condition N (see TRAP).

## 2.6.2.3 HEX\$(N)

Returns a four character string containing the hexadecimal representation of value N, where N is in the range  $0 < N < 65535$ .

## 2.6.2.4 LEFT\$(X\$,X)

This returns a string containing the leftmost X characters of X\$.

## 2.6.2.5 LEN(X\$)

Returns the number of characters (including spaces and other symbols) in the string X\$.

## 2.6.2.6 MID\$(X\$,N,X)

This returns a string containing X characters, starting with the Nth character in X\$. MID\$ can also be used on the left side of an assignment statement as a pseudo-variable. MID\$(string variable, starting position, length) = source string, see the example below.

This function reassigns values of positions (starting position) through (starting position + length) of source string to the characters of string variable in corresponding locations. Length defaults to the length of string variables, and an error results if (starting position + length) is greater than the length of the source string.

## EXAMPLE:

```
10 A$="THE LAST GOODBYE"
20 PRINT A$           Prints "THE LAST GOODBYE"
30 MID$(A$,6,3)="ONG"
40 PRINT A$           Prints "THE LONG GOODBYE"
```

## 2.6.2.7 RIGHT\$(X\$,X)

This returns the right most X characters in X\$.

## 2.6.2.8 STR\$(X)

This returns a string which is identical to the PRINTed version of X.

## EXAMPLE:

```
X=123
A$=STR$(X)
```

## 2.6.3 Other Functions

## 2.6.3.1 FRE(X)

This function returns the number of unused bytes available in memory. X is a dummy argument.



### 2.6.3.2 POS(X)

This function returns the number of the column (0-39) where the next PRINT statement begins on the screen. X is a dummy argument.

### 2.6.3.3 SPC(X)

This is used in the PRINT statement. It allows you to skip over X spaces. X can have a value from 0-255.

### 2.6.3.4 TAB(X)

This is used in the PRINT statement. The next item to be printed is in column number X. X can have a value from 0 to 255.

### 2.6.3.5 $\pi$ (PI)

The PI symbol, when used in an equation, has the value of 3.14159265.

## 2.7 VARIABLES AND OPERATORS

### 2.7.1 Variables

Your computer uses three types of variables in BASIC. These are normal numeric, integer numeric and string (consisting of alphanumeric and other characters) variables.

Normal NUMERIC VARIABLES, also called floating point variables, can have any value from  $10^{-38}$  to  $10^{+38}$ , with up to nine digits including the decimal point. When a number becomes larger than nine digits can show, as in  $10^{-10}$  or  $10^{+10}$ , the computer displays it in scientific notation, 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 in the range -32768 thru +32767, with no fractional portion, i.e. no decimal places. An integer variable is a number like 5, 10, or -100. Integers take up less space than floating point variables when used in an array.

STRING VARIABLES are those used for character data. They can contain numbers, letters and any other character that the computer can display. An example of a string variable is "COMMODORE PLUS/4".

### 2.7.1.1 VARIABLE NAMES

VARIABLE NAMES may consist of a single letter, a letter followed by a number, or two letters. Although variable names may be longer than 2 characters, only the first two are significant.

An integer variable is specified by using the percent (%) sign after the variable name. A string variable has the dollar sign (\$) after its name.

#### 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 variables in the list.

EXAMPLES: A(7), BZ%(11), A\$(87)

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

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

### 2.7.1.2 RESERVED VARIABLE NAMES

There are seven variable names which are reserved for use by your computer, and may not be used for another purpose. These are the variables DS, DS\$, ER, EL, ST, TI and TI\$. You also cannot use KEYWORDS such as TO and IF, or any names that contain KEYWORDS, e.g. SRUN, RNEW or XLOAD are not allowed as variable names.

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 input/output operation. See the READST KERNAL routine (Section 4.11.3) for more information on STATUS.

TI and TI\$ are variables that relate to the real-time clock built into your computer. The system clock is updated every 1/60th of a second. It starts at 0 when your machine is turned on, and is reset only by changing the value of TI\$. The variable TI gives you 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 in 24-hour format. 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 required numeric value, and is 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 your computer is turned off. It starts at zero when your computer is turned on and is reset to zero when the value of the clock exceeds 235959, i.e. 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 display the disk drive status, 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 which allows your program to print one of the BASIC error messages. PRINT ERR\$(ER) prints out the proper error message.

### 2.7.2 BASIC OPERATORS

The ARITHMETIC operators include the following signs:

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

In a statement containing more than one operator, there is a set order in which the operations are carried out - first, exponentiation, then multiplication and division, and last, addition and subtraction. If two operations have the same priority, then calculations are performed in the order they occur in the statement, from left to right. If you want these operations to occur in a different order, BASIC 3.5 allows you to give a calculation a higher priority by placing parentheses around it. Operations enclosed in parentheses are calculated before any other operation. You must ensure that your equations have the same number of left parentheses as right parentheses or a SYNTAX ERROR message is displayed when you RUN your program.

There are also operators for equalities and inequalities, these are called RELATIONAL operators. They are listed below. 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 used most often to join multiple formulas in IF...THEN statements. When they are used with arithmetic operators, they are evaluated last, i.e. after + and -.

#### EXAMPLES:

```

IF A=B AND C=D THEN 100      requires both A=B and 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>B           displays a value of -1
PRINT 123 AND 15:PRINT 5 OR 7 displays 11 and 7

```

## 2.8 BASIC Abbreviation and Reference Chart

KEYWORD	ABBREVIATION	TYPE
ABS	a <SHIFT> B	function - numeric
ASC	a <SHIFT> S	function - numeric
ATN	a <SHIFT> T	function - numeric
AUTO	a <SHIFT> U	command
BACKUP	b <SHIFT> A	command
BOX	b <SHIFT> O	statement
CHAR	ch <SHIFT> A	statement
CHR\$	c <SHIFT> H	function - string
CIRCLE	c <SHIFT> I	statement
CLOSE	cl <SHIFT> O	statement
CLR	c <SHIFT> L	statement
CMD	c <SHIFT> M	statement
COLLECT	col <SHIFT> L	command
COLOR	co <SHIFT> L	statement
CONT	c <SHIFT> O	command
COPY	co <SHIFT> P	command
COS	none	function - numeric
DATA	d <SHIFT> A	statement
DEC	none	function - numeric
DEF FN	d <SHIFT> E	statement
DELETE	de <SHIFT> L	command
DIM	d <SHIFT> I	statement
DIRECTORY	di <SHIFT> R	command
DLOAD	d <SHIFT> L	command
DO	none	statement
DRAW	d <SHIFT> R	statement
DSAVE	d <SHIFT> S	command
END	e <SHIFT> N	statement
ERR\$	e <SHIFT> R	function - string
EXP	e <SHIFT> X	function - numeric
FOR	f <SHIFT> O	statement
FRE	f <SHIFT> R	function - numeric
GET	g <SHIFT> E	statement
GETKEY	getk <SHIFT> E	statement
GET#	none	statement
GOSUB	go <SHIFT> S	statement
GOTO	g <SHIFT> O	statement
GRAPHIC	g <SHIFT> R	statement
GSHAPE	g <SHIFT> S	statement
HEADER	he <SHIFT> A	command
HEX\$	h <SHIFT> E	function - string
IF...GOTO	none	statement
IF...THEN...ELSE	none	statement
INPUT	none	statement
INPUT#	i <SHIFT> N	statement
INSTR	in <SHIFT> S	function - numeric
INT	none	function - numeric
JOY	j <SHIFT> O	function - numeric
KEY	k <SHIFT> E	command

LEFT\$	le	<SHIFT> F	function - string
LEN		none	function - numeric
LET	l	<SHIFT> E	statement
LIST	l	<SHIFT> I	command
LOAD	l	<SHIFT> O	command
LOCATE	lo	<SHIFT> C	statement
LOG		none	function - numeric
LOOP	lo	<SHIFT> O	statement
MID\$	m	<SHIFT> I	function - string
MONITOR	m	<SHIFT> O	statement
NEW		none	command
NEXT	n	<SHIFT> E	statement
ON...GOSUB	on...go	<SHIFT> S	statement
ON...GOTO	on...g	<SHIFT> O	statement
OPEN	o	<SHIFT> P	statement
PAINT	p	<SHIFT> A	statement
PEEK	p	<SHIFT> E	function - numeric
POKE	p	<SHIFT> O	statement
POS		none	function - numeric
PRINT	?		statement
PRINT#	p	<SHIFT> R	statement
PRINT USING	?us	<SHIFT> I	statement
PUDEF	p	<SHIFT> U	statement
RCLR	r	<SHIFT> C	function - numeric
RDOT	r	<SHIFT> D	function - numeric
READ	r	<SHIFT> E	statement
REM		none	statement
RENAME	re	<SHIFT> N	command
RENUMBER	ren	<SHIFT> U	command
RESTORE	re	<SHIFT> S	statement
RESUME	res	<SHIFT> U	statement
RETURN	re	<SHIFT> T	statement
RGR	r	<SHIFT> G	function - numeric
RIGHT\$	r	<SHIFT> I	function - string
RLUM	r	<SHIFT> L	function - numeric
RND	r	<SHIFT> N	function - numeric
RUN	r	<SHIFT> U	command
SAVE	s	<SHIFT> A	command
SCALE	sc	<SHIFT> A	statement
SCNCLR	s	<SHIFT> C	statement
SCRATCH	sc	<SHIFT> R	command
SGN	s	<SHIFT> G	function - numeric
SIN	s	<SHIFT> I	function - numeric
SOUND	s	<SHIFT> O	statement
SPC(	s	<SHIFT> P	function - special
SQR	s	<SHIFT> Q	function - numeric
SSHAPE	s	<SHIFT> S	statement
Status		none	reserved - numeric
STOP	s	<SHIFT> T	statement
STR\$	st	<SHIFT> R	function - string
SYS	s	<SHIFT> Y	statement

TAB (	t	<SHIFT> A	function - special
TAN		none	function - numeric
TI		none	reserved - numeric
TI\$		none	reserved - string
TRAP	t	<SHIFT> R	statement
TROFF	tro	<SHIFT> F	statement
TRON	tr	<SHIFT> O	statement
UNTIL	u	<SHIFT> N	statement
USR	u	<SHIFT> S	function - special
VAL		none	function - numeric
VERIFY	v	<SHIFT> E	command
VOL	v	<SHIFT> O	statement
WAIT	w	<SHIFT> A	statement
WHILE	w	<SHIFT> H	statement



## SECTION THREE

### PROGRAMMING MACHINE CODE

#### 3.1 What is Machine Language?

At the heart of every microcomputer is a central microprocessor, sometimes known as the central processing unit (C.P.U.). The C.P.U. is a very special microchip which is the computer's "brain". Almost everything that the computer does is controlled by the C.P.U. Every microprocessor understands its own language of instructions which are called machine language instructions. Machine language is the ONLY programming language that your C16 or PLUS/4 understands, it is the NATIVE language of the machine.

COMMODORE BASIC V3.5 is not the machine language of the C16 or PLUS/4. In order that the computer can understand the COMMODORE BASIC V3.5 programming language the computer contains a machine language program stored in a read only memory, i.e. ROM. This machine language program is called the OPERATING SYSTEM, i.e. OS. When the computer is switched on, it is automatically "RUN".

The OPERATING SYSTEM "organizes" all the memory in your machine for the various tasks the computer performs.

All of the commands that are available in COMMODORE BASIC V3.5 are simply recognized by another huge machine language program built into your C16 or PLUS/4. This program "RUNS" the appropriate piece of machine language depending on which BASIC command is being executed. This program is called the BASIC INTERPRETER, because it interprets each command.

#### 3.2 What does Machine Code Look Like?

Each memory location has its own number which identifies it. This number is known as the "address" of a memory location. If you imagine the memory in your C16 or PLUS/4 as a street of buildings, then the number on each door is, of course, the address.

### 3.3 Simple Memory Map of the C16 and PLUS/4

The following table introduces you to the computer's "street" and the functions of the various addresses in that "street".

ADDRESS	DESCRIPTION
0 & 1	7501 registers
2 to 2045	Start of memory Memory used by the operating system
2048 to 3071	Colour RAM
3072 to 4095	Screen memory
4096	Start of BASIC text area
8192	Start of BASIC when HIRES is on
53248	Beginning of character ROM

If you do not understand what the description of each part of memory means, do not worry, this becomes clear as you work through this section of the manual.

Machine language programs consist of instructions which may or may not have operands (parameters) associated with them. Each instruction takes up one memory location, any associated operand is contained in one or two locations following the instruction.

In BASIC programs, words like PRINT and GOTO only take up one memory location, rather than one for each character of the word. The contents of the location that represents a particular BASIC keyword is called a token. In machine language, there are different tokens for different instructions, which also take up just one byte (one memory location = one byte).

Machine language instructions are very simple. Each individual instruction carries out one small step in a program such as changing the contents of a memory location, or changing one of the internal registers, i.e. special storage locations, inside the microprocessor. The internal registers form the basis of machine language.

### 3.4 The Registers Inside the 7501 Microprocessor

#### 3.4.1 THE ACCUMULATOR

This is the most important register in the microprocessor. Various machine language instructions allow you to copy the contents of a memory location into the accumulator, copy the contents of the accumulator into a memory location, modify the contents of the accumulator or some other register directly, without affecting any memory. The accumulator is the only register that has mathematical instructions.

#### 3.4.2 THE X INDEX REGISTER

This is a very important register. There are instructions for nearly all of the transformations you can make to the contents of the accumulator. However, there are other instructions for things that only the X register can do. Various machine language instructions allow you to copy the contents of a memory location into the X register, copy the contents of the X register into a memory location, and modify the contents of the X, or some other register, directly.

#### 3.4.3 THE Y INDEX REGISTER

This register has instructions for nearly all of the transformations you can make to the contents of the accumulator and the X register. There are also other instructions for things that only the Y register can do. Various machine language instructions allow you to copy the contents of a memory location into the Y register, copy the contents of the Y register into a memory location, and modify the contents of the Y, or some other register directly.

#### 3.4.4 THE STATUS REGISTER

This register consists of six "flags". A flag lets you know whether something has, or has not, occurred. These flags give you information about the current "status" of the processor.

#### 3.4.5 THE PROGRAM COUNTER

This contains the address of the current machine language instruction being executed. Because the operating system is always "RUNning" in your C16 or PLUS/4, the program counter is constantly changing. It can only be stopped by halting the microprocessor in some way.

### 3.4.6 THE STACK POINTER

This register contains the location of the first empty place on the stack. The stack is used for temporary storage by machine language programs and by the computer.

### 3.4.7 THE INPUT/OUTPUT PORT

This register is an 8-bit input/output port. It is at memory location 0, for the DATA DIRECTION REGISTER, and 1, for the actual PORT.

## 3.5 Writing Machine Language Programs

The C16 and PLUS/4 both contain a machine language program called TEDMON which enables you to easily write machine language programs. TEDMON includes a machine language monitor, a mini assembler, and a disassembler.

Machine language programs written using TEDMON can run by themselves, or be used as very fast "subroutines" for BASIC programs since TEDMON has the ability to coexist with BASIC.

## 3.5.1 TEDMON COMMANDS

A	ASSEMBLE	Assemble a line of 7501 code
C	COMPARE	Compare two sections of memory and report any differences
D	DISASSEMBLE	Disassemble an area of memory
F	FILL	Fill an area of memory with the specified byte
G	GO	Start execution at the specified address
H	HUNT	Hunt through memory for all occurrences of certain bytes
L	LOAD	Load a file from tape or disk
M	MEMORY	Display the hexadecimal values of particular memory locations
R	REGISTERS	Display the 7501 registers
S	SAVE	Save to tape or disk
T	TRANSFER	Transfer code from one section of memory to another
V	VERIFY	Compare memory with tape or disk
X	EXIT	Exit TEDMON
.	(period)	Assembles a line of 7501 code
>	(greater than)	Modifies memory
;	(semi-colon)	Modifies 7501 register displays

NOTE: In the PLUS/4 only, location \$07F8 controls whether TEDMON looks at ROM or RAM above \$8000. If this location is set to \$00 when TEDMON is commanded to do a disassembly or memory dump above \$8000, it displays BASIC and the KERNAL. If this location is set to \$80, TEDMON displays the RAM under BASIC and KERNAL. This is often convenient for machine language program development. Note that location \$07F8 does not affect the GO command. The GO command starts execution in the current memory map (ROM on or RAM on) regardless of the setting of location \$07F8.

### 3.5.2 USING TEDMON

Enter TEDMON by typing:

MONITOR

TEDMON responds by displaying the 7510 registers and the flashing cursor. The cursor is the prompt that lets you know that TEDMON is waiting for your commands.

### 3.5.3 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 where the opcode is to be placed.

<opcode mnemonic> a standard MOS Technology assembly language mnemonic, e.g. LDA, STX, ROR, etc.

<operand> the operand, when required, can be any of the legal addressing modes. For zero-page modes, a 2-digit hex number is required 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, and the cursor moves to the next line. The screen editor can then be used to correct those errors.

When a line of code is successfully assembled, the assembler prints a prompt containing the next legal memory location which can be used for an instruction. This means that A and the address number do not have to be typed more than once when typing assembly language programs into the C16 and PLUS/4.

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 address of the first area of memory

<address 2> is a hexadecimal number indicating the end address of the first area of memory

<address 3> is a hexadecimal number indicating the start address of the area of memory to be compared with the first area of memory

If the two areas of memory are the same, then TEDMON prints a <RETURN>, indicating that the second area of memory is the same as the first. The addresses of any bytes in the two areas which are different 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 at which disassembly is to start

<address 2> an optional hexadecimal ending address of the 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 period rather than an A, this is 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 operand on the screen, then press the <RETURN> key. This enters the line and calls the assembler for further modifications.

A disassembly can be paged. Typing a D on its own causes the next 20 bytes of code to be disassembled to the screen.

**EXAMPLE:**

```
D 3000 3004
. 3000 A9 00      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 be filled with the <byte>

<address 2> the last location to be filled with the <byte>

<byte> a 1 or 2-digit hexadecimal number

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

**EXAMPLE:**

F 0400 0518 EA

This fills memory locations from \$0400 to \$0518 with \$EA which is a NOP instruction.

**COMMAND: G**

**PURPOSE:** Begin execution of a program at a specified address

**SYNTAX :** G [<address>]

<address> is an optional argument specifying the new value of the program counter and the address where execution is to start. When <address> is left out, execution begins at the current location of the PC, i.e. Program Counter. The PC can be viewed using the R command.

The GO command restores all registers and begins execution at the specified starting address. The registers can be displayed using the R command. Caution is recommended in using the GO command. To return to TEDMON after executing a machine language program, use the BRK instruction.

**EXAMPLE:**

G 140C

Execution begins at location \$140C.

**COMMAND: H**

**PURPOSE:** Hunt through memory within a specified range for all occurrences of a set of bytes

**SYNTAX :** H <address 1> <address 2> <data>

<address 1> beginning address for the hunt

<address 2> ending address for the hunt

<data> is the data set to search for. Data may be hexadecimal or an ASCII string. An ASCII string is specified by preceding the first character with a single apostrophe, e.g. 'STRING. Data may be a single or multiple element argument. When multiple and in hexadecimal, each number must be separated by a space.

**EXAMPLES:**

H C000 FFFF 'READ            Search for ASCII string "READ" from  
                                 \$C000 to \$FFFF

H A000 A101 A9 FF 4C        Search for data \$A9, \$FF, \$4C, from  
                                 \$A100 to \$A101

**COMMAND: L**

**PURPOSE:** Load a file from cassette or disk

**SYNTAX :** L <"filename">,<device>

<"filename"> is any legal C16 or PLUS/4 filename enclosed in quotes

<device> is a hexadecimal number indicating the device to load from. 1 is cassette, 8 is disk.

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

**EXAMPLE:**

L "SCREEN",1        Reads the file called SCREEN from cassette

L "TANK",8         Reads the file called TANK from disk

**COMMAND: M**

**PURPOSE:** To display memory within the specified address range as a hexadecimal and ASCII dump

**SYNTAX :** M [<address 1>] [<address 2>]

<address 1> first address of memory dump. This is optional. If it is omitted, then one page is displayed starting from the last address specified

<address 2> last address of memory dump. Optional. If omitted, then one page is displayed starting from <address 1>

Memory is displayed in the following format:

```
>0310 8B 8C 42 CE 0E CE 4C F4 :..BN. NLI
```

Memory content may be edited using the screen editor. Move the cursor to the data to be modified, type the desired correction and press <RETURN>. If there is a bad RAM location or an attempt to modify ROM has occurred, a question mark (?) is displayed.

An ASCII dump of the data is displayed in REVERSE colours to the right of the hex data. The REVERSE colours are to differentiate the dump from other data displayed on the screen. When a character is not printable, it is displayed as a reverse period (.).

As with the DISASSEMBLY command, you can page down. This is done by typing M and <RETURN>.

**EXAMPLE:**

```
>1C00 41 00 AA AA 00 98 56 45 :A.**..VE
>1C10 41 00 AA AA 00 98 56 45 :A.**..VE
>1C10 41 00 AA AA 00 98 56 45 :A.**..VE
>1C18 41 00 AA AA 00 98 56 45 :A.**..VE
>1C20 41 00 AA AA 00 98 56 45 :A.**..VE
>1C28 41 00 AA AA 00 98 56 45 :A.**..VE
>1C30 41 00 AA AA 00 98 56 45 :A.**..VE
>1C38 41 00 AA AA 00 98 56 45 :A.**..VE
>1C40 41 00 AA AA 00 98 56 45 :A.**..VE
>1C48 41 00 AA AA 00 98 56 45 :A.**..VE
>1C50 41 00 AA AA 00 98 56 45 :A.**..VE
>1C58 41 00 AA AA 00 98 56 45 :A.**..VE
```

COMMAND: >

PURPOSE: Can be used to set 1 to 8 memory locations at a time

SYNTAX : >address data byte 1 <data byte 2> <data byte 3>  
...8

address - first memory address to be set

data byte 1 - data to be put at address

<data byte 2...8> - data to be placed in the successive memory locations following the first address. Optional.

EXAMPLES:

>2000 08            places a \$08 at location \$2000

>3000 23 45 65    places a \$23 at location \$3000, a \$45 at  
\$3001, and a \$65 at \$3002

COMMAND: R

PURPOSE: Show 7501 registers. The Program Counter, Status Register, Accumulator, X and Y registers, and Stack Pointer are displayed.

SYNTAX : R

EXAMPLE:

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

NOTE: the semi-colon (;) can be used to modify register displays in the same way that > is used to modify memory.

**COMMAND: S**

**PURPOSE:** Save the contents of memory onto tape or disk

**SYNTAX :** S <"filename">,<device>,<address 1>,<address 2>

<"filename"> any legal C16 or PLUS/4 filename enclosed in quotes

<device> two possible devices are cassette and disk. To save onto cassette use a device number of 1, to save onto disk use a device number of 8.

<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 using the L command.

**EXAMPLE:**

S "GAME",8,0400,0C00

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

**COMMAND: T**

**PURPOSE:** Transfer an area of memory to another location

**SYNTAX :** T <address 1> <address 2> <address 3>

<address 1> the start address of data to be moved

<address 2> the end address of data to be moved

<address 3> start address of new location to which the data is to be transferred

Data can be moved from low memory to high memory or vice-versa. Additional memory segments of any length can be moved forward or backward any number of bytes.

**NOTE:** the value of <address 3> must not fall within the range of <address 1> to <address 2>.

**EXAMPLE:**

T 1401 1600 1400      Shifts the data from \$1401 up to and including \$1600 one byte down in memory

**COMMAND: V**

**PURPOSE:** Verify a file on cassette or disk with the memory contents

**SYNTAX :** V <"filename">,<device>

<"filename"> is any legal C16 or PLUS/4 filename enclosed in quotation marks

<device> is a hexadecimal number indicating which device the file is on, cassette is 1 or 01, disk is 8 or 08

The VERIFY command compares a file with the contents of memory. The C16 or PLUS/4 responds by displaying the message VERIFYING. If an error is found, the word ERROR is displayed. If the file is successfully verified, the flashing cursor reappears.

**EXAMPLE:**

V "WORKLOAD",8

**COMMAND: X**

**PURPOSE:** Exit to BASIC

**SYNTAX :** X

When the X command is given, the machine stack pointer is set to the current stack pointer value (see the R command). If this is modified in any way, use the BASIC CLR command to reset the pointers after exiting to BASIC.

### 3.6 HEXADECIMAL NOTATION

Hexadecimal is the notation usually used by machine language programmers when they talk about a number or address in a machine language program.

By looking at decimal, i.e. base 10, numbers, you can see that each digit falls somewhere in the range 0 thru 9, i.e. in the range zero through to a number equal to the base less one. This is true of all number bases. Binary, i.e. base 2 numbers have digits ranging from zero to one, i.e. one less than the base. Similarly, hexadecimal, i.e. base 16, numbers have digits ranging from zero to fifteen. As there are no single digit figures for the numbers ten to fifteen, the first six letters of the alphabet are used. This is shown in the following table:

DECIMAL	HEXADECIMAL	BINARY
0	0	00000000
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

### 3.7 ADDRESSING MODES

#### 3.7.1 ZERO PAGE

Absolute addresses are expressed in terms of a high and a low order byte. The high order byte is often referred to as the page of memory. For example, the address \$1637 is in page \$16, i.e. decimal 22, and \$0277 is in page \$02, i.e. decimal 2. There is, however, a special mode of addressing known as zero page addressing and is, as the name implies, associated with the addressing of memory locations in page zero. These addresses, therefore, ALWAYS have a high order byte of zero. The zero page mode of addressing only expects one byte to describe the address, rather than two when using an absolute address. The zero page addressing mode tells the microprocessor to assume that the high order address is zero. Therefore, zero page can reference memory locations whose addresses are between \$0000 and \$00FF.



### 3.7.2 THE STACK

The 7501 microprocessor has a temporary storage area, known as the stack which is used by both the programmer and the microprocessor. It is also used to remember a particular order of events. When a GOSUB statement is encountered in a program, the BASIC interpreter "pushes", i.e. places, its current position in the program onto the stack before going to do the subroutine. When a RETURN is executed, the interpreter "pulls", i.e. takes, this information off the stack so that the program continues executing at the correct point. The assembly language instructions for doing this are PHA, which pushes the contents of the accumulator onto the stack, and PLA, i.e. the reverse, which pulls a value off the stack and places it in the accumulator. The status register can also be pushed and pulled with the PHP and PLP instructions respectively.

The stack is 256 bytes long, and is located in page one of memory, from \$0100 to \$01FF. It is organized backwards in memory, the first position in the stack is at \$01FF, and the last is at \$0100.

Another register in the 7501 microprocessor is called the stack pointer. This always points to the next available location in the stack. When a value is pushed onto the stack, it is placed at the location to which the stack pointer is pointing, and the stack pointer is moved down, i.e. decremented, to the next position. When a value is pulled off the stack, the stack pointer is incremented.

NOTE: the X register is referred to as X from now on. Similarly A (for the accumulator), Y (for the Y index register), S (for the stack pointer), and P (for the processor status).

### 3.8 INDEXING

Indexing plays an extremely important part in the running of the 7501 microprocessor. It can be defined as "creating an actual address from a base address plus the contents of either X or Y registers".

For example, if X contains \$05, and the microprocessor executes an LDA instruction in the "absolute X indexed mode" with base address \$9000, then the actual location that is loaded into the A register is  $\$9000 + \$05 = \$9005$ . The mnemonic format of an absolute indexed instruction is the same as an absolute instruction except that an "X" or "Y" denoting the index is added to the address.

## EXAMPLE:

```
LDA $9000,X
```

There are absolute indexed, zero page indexed, indirect indexed, and indexed indirect addressing modes available on the 7501 microprocessor.

## 3.8.1 INDIRECT INDEXED

This only allows usage of the Y register as the index. The actual address can only be in zero page. The mode of instruction is called indirect because the zero page address specified in the instruction contains the low byte of the actual address, and the next byte contains the high order byte.

## EXAMPLE:

Location \$02 contains \$45, and location \$03 contains \$1E. If the instruction to load the accumulator in the indirect indexed mode is executed and the specified zero page address is \$02, then the actual address is:

```
Low order = contents of $02
High order = contents of $03
Y register = $00
```

Thus the actual address = \$1E45 + Y = \$1E45.

The title of this mode implies an indirect principle. To look at it another way, "a letter is to be delivered to the post office at address \$02, MEMORY ST., and the address on the letter is \$05 houses past past \$1600, MEMORY ST.". This is equivalent to the code:

```
LDA #$00    ; load low order actual base address
STA $02    ; set the low order of the indirect base address
LDA #$16    ; load the high order indirect address
STA $03    ; set the high byte of the indirect address
LDY #$05    ; set the indirect index (Y)
LDA ($02),Y ; load indirectly indexed by Y
```

## 3.8.2 INDEXED INDIRECT

Indexed indirect only allows usage of the X register as the index. This is the same as indirect indexed, rather than the actual base address. Therefore, the actual base address IS the actual address because the index has already been used for the indirect. Indexed indirect would also be used if a table of indirect pointers were located in zero page memory, and the X register could then specify which indirect pointer to use.

## EXAMPLE:

Location \$02 contains \$45, and location \$03 contains \$10. If the instruction to load the accumulator in the indexed indirect mode is executed and the specified zero page address is \$02, then the actual address is:

Low order = contents of (\$02 + X)  
 High order = contents of (\$03 + X)  
 X register = \$00

Thus the actual pointer is in \$02 + X = \$02, and the actual address is the indirect address contained in \$02 which is \$1045.

The title of this mode does, in fact, imply the principle underlying it. Look at it this way: "a letter is to be delivered to the fourth post office at address \$02, MEMORY ST., and the address on the letter is \$1600, MEMORY ST.". This is equivalent to the code:

```
LDA #$00      load low order actual base address
STA $06      set the low byte of the indirect address
LDA #$16     load high order indirect address
STA $07     set the high byte of the indirect address
LDX #$04     set the indirect index (X)
LDA ($02,X)  load indirectly indexed by X
```

NOTE: of the two indirect methods of addressing, the first (indirect indexed) is far more widely used.

## 3.8.3 BRANCHES AND TESTING

Another very important principle in machine language is the ability to test, and detect certain conditions in a similar fashion to the "IF...THEN, IF...GOTO" structure in BASIC.

The various flags in the status register are affected in different ways by different instructions. For example, there is a flag that is set when an instruction has caused a zero result, and is reset when a result is non-zero. For example, the instruction:

```
LDA #$00
```

causes the zero result flag to be set, because that instruction has resulted in the accumulator containing a zero.

There are a set of instructions that, given a particular condition, branch to another part of the program. An example of a branch instruction is BEQ, which means Branch if result Equal to zero. These instructions branch if the condition is true. If it is not, the program continues to the next instruction, as if nothing had occurred. The branch instructions branch by internally examining the status register, not as a result of the previous instruction(s). For example, the BEQ instruction branches if the zero result flag (Z) is set. The BEQ instruction has an opposite instruction BNE, which means Branch on result Not Equal to zero, i.e. Z not set. Every branch instruction has an opposite branch instruction.

The index registers have a number of associated instructions which modify the contents of those registers. For example, the INX instruction INcrements the X index register. If the X register contained \$FF, which is the maximum number the X register can contain, before it was incremented, it "wraps around" back to zero. If you require a program to continue to do something until you perform the increment of the X index that pushes it around to zero, you can use the BNE instruction to continue "looping" around, until X becomes zero.

The reverse of INX, is DEX, which is DEcrement the X index register. If the X index register contains zero, DEX wraps around to \$FF.

Similarly, there are the INY and DEY instructions for the Y index register.

If you do not want your program to wait until X or Y reaches, or does not reach, zero, then use the comparison instructions CPX and CPY. These allow you to test the index registers, or the contents of memory locations with specific values. For example, if you wish to see if the X register contains \$40, you would use the instruction:

```
CPX #$40      compare X with the value $40
BEQ $????    branch to somewhere else in the program if this
              condition is "true"
```

The compare, and branch instructions play a major part in any machine language program.

The operand specified in a branch instruction when using TEDMON is the address of the section that the branch goes to when the conditions are met. However, the operand is only an offset, which gets you from where the program currently is to the address specified. This offset is a single byte, and therefore, the range that a branch instruction can use is limited. It can branch from 128 bytes backward to 127 bytes forward. Note that this is a total range of 255 bytes, the maximum that a single byte can contain.

TEDMON tells you if you try to "branch out of range" by refusing to "assemble" that particular instruction. The branch is a "quick" instruction by machine language standards because it uses the "offset" principle as opposed to an absolute address. TEDMON allows you to type in an absolute address and it calculates the correct offset. This is one of the advantages of using an assembler.

### 3.9 SUBROUTINES

In machine language you can call subroutines in much the same way as you do in BASIC. The instruction to call a subroutine is JSR (Jump to SubRoutine), followed by the absolute address of the subroutine.

Incorporated in the operating system is a machine language subroutine that PRINTs a character to the screen. The CBM ASCII code for the character must be in the accumulator before calling the subroutine. The address of this subroutine is \$FFD2.

The following program prints "HI" on the screen:

```
A 3000 LDA #$48      load the CBM ASCII code of "H"
A 3002 JSR $FFD2     print it
A 3005 LDA #$49      load the CBM ASCII code of "I"
A 3007 JSR $FFD2     print that too
A 300A LDA #$0D      load a carriage return
A 140C JSR $FFD2     print it
A 140F BRK          return to TEDMON
```

```
G 3000                prints "HI" and returns to TEDMON
```

This "PRINT-a-character" routine is part of the KERNAL jump table. The instruction similar to GOTO in BASIC is JMP, which means JUMP to the specified absolute address. The KERNAL is a long list of "standardized" subroutines that control ALL input and output operations. Each entry in the KERNAL jump table Jumps to a subroutine in the operating system. This "jump table" is found between memory locations \$FF81 to \$FFF3 in the operating system.

The following program displays the alphabet using the KERNAL PRINT routine. The only instruction in this program which you have not yet been introduced to is TXA. This Transfers the contents of the X index register into the Accumulator.



```

A 3000 LDX #$41      X = CBM ASCII of "A"
A 3002 TXA          A = X
A 3003 JSR $FFD2    print character
A 3006 INX          bump count
A 3007 CPX #$5B     have we gone past "Z"?
A 3009 BNE $3002    if no, go back and do more
A 300B BRK          if yes, return to TEDMON

```

To see your C16 or PLUS/4 print the alphabet, type:

```
G 3000
```

The comments that are beside the program explain its flow and logic. If you are writing a program, COMMODORE recommends that you write it on paper first, and then test small parts of it at a time.

### 3.10 7501 MICROPROCESSOR INSTRUCTION SET - ALPHABETIC SEQUENCE

```

ADC    add memory to accumulator with carry
AND    "AND" memory with accumulator
ASL    shift left one bit (memory or accumulator)

BCC    branch on carry clear
BCS    branch on carry set
BEQ    branch on result zero
BIT    test bits in memory with accumulator
BMI    branch on result minus
BNE    branch on result not zero
BPL    branch on result plus
BRK    force break
BVC    branch on overflow clear
BVS    branch on overflow set

CLC    clear carry flag
CLD    clear decimal mode
CLI    clear interrupt disable bit
CLV    clear overflow flag
CMP    compare memory and accumulator
CPX    compare memory and index X
CPY    compare memory and index Y

DEC    decrement memory by one
DEX    decrement index X by one
DEY    decrement index Y by one

EOR    "EXCLUSIVE-OR" memory with accumulator

INC    increment memory by one
INX    increment index X by one
INY    increment index Y by one

```

JMP    jump to new location  
JSR    jump to new location saving return address

LDA    load accumulator with memory  
LDX    load index X with memory  
LDY    load index Y with memory  
LSR    shift right one bit (memory or accumulator)

NOP    no operation

ORA    "OR" memory with accumulator

PHA    push accumulator onto stack  
PHP    push processor status onto stack  
PLA    pull accumulator off stack  
PLP    pull processor status off stack

ROL    rotate one bit left (memory or accumulator)  
ROR    rotate one bit right (memory or accumulator)  
RTI    return from interrupt  
RTS    return from subroutine

SBC    subtract memory from accumulator with borrow  
SEC    set carry flag  
SED    set decimal mode  
SEI    set interrupt disable status  
STA    store accumulator in memory  
STX    store index X in memory  
STY    store index Y in memory

TAX    transfer accumulator to index X  
TAY    transfer accumulator to index Y  
TSX    transfer stack pointer to index X  
TXA    transfer index X to accumulator  
TXS    transfer index X to stack pointer  
TYA    transfer index Y to accumulator

### 3.11 THE KERNAL

The KERNAL is a standardized JUMP TABLE to the 39 input, output, and memory management routines in the operating system.

The location of each routine in ROM may change as the system is upgraded, and the KERNAL jump table is always changed to match. If your machine language routines only use the system ROM routines through the KERNAL, it takes much less work to modify them, should that need ever arise.

The KERNAL jump table is located in the last page of memory, in read-only memory (ROM) and allows you to simplify the machine language programs you write.



To use the KERNAL jump table, first set up the parameters that the KERNAL routine needs in order to work. Then JSR, i.e. Jump to SubRoutine, to the proper place in the KERNAL jump table. After performing its function, the KERNAL transfers control back to your machine language program. Depending on which KERNAL routine you are using, certain registers may pass parameters back to your program. The particular registers for each KERNAL routine may be found in the individual descriptions of the KERNAL subroutines (see Section 4.11.3).

You can JSR directly to the required KERNAL subroutine. However, using the jump table ensures that machine language programs still work if the KERNAL or BASIC is changed. Future operating systems or machines may have the memory locations of routines in different positions in the memory map, but the jump table routines in existing programs will still work correctly in spite of any changes.

### 3.11.1 HOW TO USE THE KERNAL

When writing machine language programs it is convenient to use the routines which are already part of the operating system for input/output, access to the system clock, memory management, and other similar operations. It is an unnecessary duplication of effort to write these routines over and over again, and easy access to the operating system helps speed machine language programming.

To use a KERNAL routine you must first make all of the preparations that the routine demands. If one routine says that you must call another KERNAL routine first, then that routine must be called. If the routine expects you to put a number in the accumulator, then that number must be there. If these guidelines are followed, the routines work as expected.

After all preparations are made, you must call the routine by means of the JSR instruction. All KERNAL routines you can access are structured as SUBROUTINES, and therefore end with an RTS instruction. When the KERNAL routine has finished its task, control is returned to your program at the instruction after the JSR.

Many of the KERNAL routines return error codes in the status word or the accumulator. This is in case you have problems in the routine. Good programming practice and the success of your machine language programs demand that you handle these error codes properly. If you ignore an error return, the rest of your program might fail.

Summary: the three steps involved when using a KERNAL routine are:

1. Set up
2. Call the routine
3. Error handling

The KERNAL routines are described in Section 3.11.3.

The following conventions are used in these descriptions:

FUNCTION NAME: name of the KERNAL routine.

CALL ADDRESS: this is the call address of the KERNAL routine, given in hexadecimal and decimal.

COMMUNICATION REGISTERS: registers listed under this heading are used to pass parameters to and from the KERNAL routines.

PREPARATORY ROUTINES: certain KERNAL routines require that data be set up before they can operate. The routines needed are listed here.

ERROR RETURNS: a return from a KERNAL routine with the CARRY set indicates that an error was encountered in processing. The accumulator contains the number of the error.

STACK REQUIREMENTS: this is the actual number of stack bytes used by the KERNAL routine.

REGISTERS AFFECTED: all registers used by the KERNAL routine are listed here.

DESCRIPTION: a short tutorial on the function of the KERNAL routine.

## 3.11.2 USER CALLABLE KERNAL ROUTINES

NAME	ADDRESS		FUNCTION
	HEX	DECIMAL	
ACPTR	: \$FFA5	: 65445	: Input byte from serial port
CHKIN	: \$FFC6	: 65478	: Open channel for input
CHKOUT	: \$FFC9	: 65481	: Open channel for output
CHRIN	: \$FFCF	: 65487	: Input character from channel
CHROUT	: \$FFD2	: 65490	: Output character to channel
CIOUT	: \$FFA8	: 65448	: Output byte to serial port
CINT	: \$FF81	: 65409	: Initialize screen editor
CLALL	: \$FFE7	: 65511	: Close all channels and files
CLOSE	: \$FFC3	: 65475	: Close specified logical file
CLRCHN	: \$FFCC	: 65484	: Close input and output
	:	:	: channels
GETIN	: \$FFE4	: 65508	: Get character from keyboard
	:	:	: queue (keyboard buffer)
IOBASE	: \$FFF3	: 65523	: Returns base address of I/O
	:	:	: devices
IOINIT	: \$FF84	: 65412	: Initialize input/output
LISTEN	: \$FFB1	: 65457	: Command devices on serial bus
	:	:	: to LISTEN
LOAD	: \$FFD5	: 65493	: Load RAM from a device
MEMBOT	: \$FF9C	: 65436	: Read/set the bottom of memory
MEMTOP	: \$FF99	: 65433	: Read/set the top of memory
OPEN	: \$FFC0	: 65472	: Open a logical file
PLOT	: \$FFF0	: 65520	: Read/set X,Y cursor position
RAMTAS	: \$FF87	: 65415	: Initialize RAM, allocate tape
	:	:	: buffer, set screen \$0400
RDTIM	: \$FFDE	: 65502	: Read real time clock
READST	: \$FFB7	: 65463	: Read I/O status word
RESTOR	: \$FF8A	: 65418	: Restore default I/O vectors
SAVE	: \$FFD8	: 65496	: Save RAM to device
SCNKEY	: \$FF9F	: 65439	: Scan keyboard
SCREEN	: \$FFED	: 65517	: Return X,Y organization of
	:	:	: screen
SECOND	: \$FF93	: 65427	: Send secondary address after
	:	:	: LISTEN
SETLFS	: \$FFBA	: 65466	: Set logical, first and second
	:	:	: addresses
SETMSG	: \$FF90	: 65424	: Control KERNAL messages
SETNAM	: \$FFBD	: 65469	: Set file name
SETTIM	: \$FFDB	: 65499	: Set real time clock
SETTMO	: \$FFA2	: 65442	: Set timeout on serial bus
STOP	: \$FFE1	: 65505	: Scan stop key
TALK	: \$FFB4	: 65460	: Command serial bus device to
	:	:	: TALK
TKSA	: \$FF96	: 65430	: Send secondary address after
	:	:	: TALK
UDTIM	: \$FFEA	: 65514	: Increment real time clock
UNLSN	: \$FFAE	: 65454	: Command serial bus to
	:	:	: UNLISTEN
UNTLK	: \$FFAB	: 65451	: Command serial bus to UNTALK
VECTOR	: \$FF8D	: 65421	: Read/set vectored I/O

## 3.11.3 KERNAL ROUTINE DESCRIPTIONS

Function name: ACPTR

Purpose: Get data from the serial bus  
Call address: \$FFA5 (hex) 65445 (decimal)  
Communication registers: A  
Preparatory routines: TALK, TKSA  
Error returns: See READST  
Stack requirements: 13  
Registers affected: A, X

Description: Use this routine when you require information from a device, e.g. disk, on the serial bus. This routine gets a byte of data from the serial bus using full handshaking. The data is returned in the accumulator. To prepare for this routine the TALK routine must be called to command the device on the serial bus to send data through the bus. If the input device needs a secondary command, it must be sent, before calling this routine, using the TKSA KERNAL routine. Errors are returned in the status word which is read using the READST routine.

How to use:

1. Command a device on the serial bus to prepare to send data to your C16 or PLUS/4, (use the TALK and TKSA KERNAL routines).
2. Call this routine (using JSR).
3. Store or otherwise use the data.

EXAMPLE:

```
; GET A BYTE FROM THE BUS  
JSR ACPTR  
STA DATA
```

Function name: CHKIN

Purpose: Open a channel for input  
Call address: \$FFC6 (hex) 65478 (decimal)  
Communication registers: X  
Preparatory routines: (OPEN)  
Error returns: See READST  
Stack requirements: NONE  
Registers affected: A, X

Description: Any logical file that has already been opened by the KERNAL OPEN routine can be defined as an input channel by this routine. The device on the channel must be an input device, otherwise an error occurs and the routine aborts.

If you are obtaining data from anywhere other than the keyboard, this routine must be called before using either the CHRIN or the GETIN KERNAL routines for data input. If you wish to use the input from the keyboard, and no other input channels are opened, then the calls to this routine and to the OPEN routine are not needed.

When this routine is used with a device on the serial bus, it automatically sends the talk address, together with the secondary address if one was specified by the OPEN routine, over the bus.

How to use:

1. OPEN the logical file if necessary (see description above).
2. Load the X register with the number of the logical file to be used.
3. Call this routine (using a JSR command).

Possible errors are:

- #3: File not open
- #5: Device not present
- #6: File not an input file

EXAMPLE:

```
; PREPARE FOR INPUT FROM LOGICAL FILE 2
LDX #$02
JSR CHKIN
```

Function name: CHKOUT

Purpose: Open a channel for output  
Call address: \$FFC9 (hex) 65481 (decimal)  
Communication registers: X  
Preparatory routines: (OPEN)  
Error returns: 0,3,5,7 (See READST)  
Stack requirements: 4+  
Registers affected: A, X

Description: Any logical file number that has been created by the KERNAL OPEN routine can be defined as an output channel. The device to which you intend opening a channel must be an output device, otherwise an error occurs and the routine is aborted.

This routine must be called before any data is sent to any output device unless you wish to use the C16 or PLUS/4 screen as your output device. If screen output is desired, and there are no other output channels already defined, then calls to this routine, and to the OPEN routine, are not required.

When used to open a channel to a device on the serial bus, this routine automatically sends the LISTEN address specified by the OPEN routine, together with a secondary address if specified.

How to use:

1. Use the KERNAL OPEN routine to specify a logical file number, a LISTEN address, and a secondary address (if needed).
2. Load the X register with the logical file number used in the OPEN statement.
3. Call this routine (by using the JSR instruction).

NOTE: This routine is NOT NEEDED to send data to the screen.

EXAMPLE:

```
LDX #$03 ; DEFINE LOGICAL FILE 3 AS AN OUTPUT CHANNEL
JSR CHKOUT
```

Possible errors are:

- #3: File not open
- #5: Device not present
- #7: Not an output file

Function name: CHRIN

Purpose: Get a character from the input channel  
Call address: \$FFCF (hex) 65487 (decimal)  
Communication registers: A  
Preparatory routines: (OPEN, CHKIN)  
Error returns: 0 (See READST)  
Stack requirements: 7+  
Registers affected: A, X

Description: This routine gets a byte of data from a channel already set up as the input channel by the KERNAL routine CHKIN. If CHKIN has NOT been used to define another input channel, then all the data is expected from the keyboard. The data byte is returned in the accumulator and the channel remains open after the call.

Input from the keyboard is handled in a special way. The cursor is turned on, and blinks until a carriage return is typed on the keyboard. All characters on the line, up to a maximum of 88 characters, are then stored in the BASIC input buffer. These characters can be retrieved one at a time by calling this routine once for each character. When the carriage return is retrieved, the entire line has been processed.



How to use:

From the keyboard:

1. Retrieve a byte of data by calling this routine.
2. Store the data byte.
3. Check if it is the last data byte (is it a CR?).
4. If not, go to step 1.

EXAMPLE:

```
LDY #$00      ; PREPARE THE Y REGISTER TO STORE THE DATA
RD JSR CHRIN
STA DATA,Y   ; STORE THE YTH DATA BYTE IN THE YTH
              ; LOCATION IN THE DATA AREA
INY
CMP #$0D      ; IS IT A CARRIAGE RETURN?
BNE RD        ; NO, GET ANOTHER DATA BYTE
```

From other devices:

1. Use the KERNAL OPEN and CHKIN routines.
2. Call this routine (using a JSR instruction).
3. Store the data.

EXAMPLE:

```
JSR CHRIN
STA DATA
```

Function name: CHROUT

Purpose: Output a character

Call address: \$FFD2 (hex) 65490 (decimal)

Communication registers: A

Preparatory routines: (CHKOUT, OPEN)

Error returns: 0 (See READST)

Stack requirements: 8+

Registers affected: A

Description: This routine outputs a character to an already opened channel. Use the KERNAL OPEN and CHKOUT routines to set up the output channel before calling this routine. If this call is omitted, data is sent to the default output device, i.e. number 3, the screen. The data byte to be output is loaded into the accumulator, and the CHROUT routine is called. The data is then sent to the specified output device. The channel is left open after the call.

NOTE: Care must be taken when using this routine that you send data to a specific serial device, as data is sent to all open output channels on the bus. Unless this is desired, all open output channels on the serial bus, other than the intended destination channel, must be closed by a call to the KERNAL CLRCHN routine.



How to use:

1. Use the CHKOUT KERNAL routine if necessary (see description above).
2. Load the data to be output into the accumulator.
3. Call this routine.

EXAMPLE:

```

                ; DUPLICATE THE BASIC INSTRUCTION CMD 4,"A";
LDX #$04       ; LOGICAL FILE #4
JSR CHKOUT     ; OPEN CHANNEL OUT
LDA #$41       ; HEX CODE FOR ASCII "A"
JSR CHROUT    ; SEND CHARACTER

```

Function name: CIOUT

Purpose: Transmit a byte over the serial bus  
 Call address: \$FFA8 (hex) 65448 (decimal)  
 Communication registers: A  
 Preparatory routines: LISTEN, [SECOND]  
 Error returns: See READST  
 Stack requirements: 5  
 Registers affected: None

Description: This routine is used to send information to devices on the serial bus. A call to this routine puts a data byte onto the serial bus using full serial handshaking. Before this routine is called, the LISTEN KERNAL routine must be used to command a device on the serial bus to get ready to receive data. If a device needs a secondary address, it must also be sent by using the SECOND KERNAL routine. The accumulator is loaded with a byte to handshake as data on the serial bus. A device must be listening or the status word returns a timeout. This routine always buffers one character, i.e. the routine holds the previous character to be sent back. This means that when a call to the KERNAL UNLSN routine is made to end the data transmission, the buffered character is sent with an End Or Identify (EOI) set. The UNLSN command is then sent to the device.

How to use:

1. Use the KERNAL LISTEN routine (and the SECOND routine if needed).
2. Load the accumulator with a byte of data.
3. Call this routine to send the data byte.

EXAMPLE:

```

                ; SEND AN X TO THE SERIAL BUS
LDA #$58       ; HEX CODE FOR ASCII "X"
JSR CIOUT     ; SEND IT

```

Function name: CINT

Purpose: Initialize screen editor  
Call address: \$FF81 (hex) 65409 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: None  
Stack requirements: 4  
Registers affected: A, X, Y

Description: This routine initializes the KERNAL screen editor.

NOTE: On the COMMODORE 64 the CINT KERNAL routine also resets the (6567) video chip. CINT does not alter the state of the video chip on the C16 and PLUS/4.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR CINT  
JMP RUN ; BEGIN EXECUTION
```

Function name: CLALL

Purpose: Close all files  
Call address: \$FFE7 (hex) 65511 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: None  
Stack requirements: 11  
Registers affected: A, X

Description: This routine closes all previously OPENed files. When this routine is called, the pointers into the open file table are reset, closing all files. The CLRCHN routine is also automatically called to reset the I/O channels.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR CLALL ; CLOSE ALL FILES AND SELECT DEFAULT I/O CHANNELS  
JMP RUN ; BEGIN EXECUTION
```

Function name: CLOSE

Purpose: Close a logical file  
Call address: \$FFC3 (hex) 65475 (decimal)  
Communication registers: A  
Preparatory routines: None  
Error returns: 0, 240 (See READST)  
Stack requirements: 2+  
Registers affected: A, X, Y

Description: This routine is used to close a logical file after all I/O operations have been completed on that file. It is called after the accumulator is loaded with the number of the logical file to be closed, this must be the same number used when the file was opened using the OPEN routine.

How to use:

1. Load the accumulator with the number of the logical file to be closed.
2. Call this routine.

EXAMPLE:

```
                ; DUPLICATE THE BASIC INSTRUCTION CLOSE 15  
LDA #$0F      ; LOAD ACCUMULATOR WITH HEX OF 15  
JSR CLOSE    ; CLOSE THE FILE
```

Function name: CLRCHN

Purpose: Clear I/O channels  
Call address: \$FFCC (hex) 65484 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: 9  
Registers affected: A, X

Description: This routine is called to clear all open channels and restore the I/O channels to their original default values. It is usually called after opening other I/O channels, such as a tape or disk drive, and using them for input/output operations. The default input device is 0, i.e. the keyboard, and the default output device is 3, i.e. the screen.

If one of the channels to be closed is on the serial port, an UNTALK signal must first be sent to clear the input channel or an UNLISTEN sent to clear the output channel. If the UNLISTEN routine is not called, thus leaving listeners active on the serial bus, several devices can receive the same data from the Cl6 or PLUS/4 at the same time. One way to take advantage of this is to command the printer to LISTEN and the disk to TALK. In this way, with the aid of a routine to get data bytes from the disk drive and send them to the printer, you can print a disk file directly.

This routine is automatically called when the KERNAL CLALL routine is executed.

How to use:

1. Call this routine using the JSR instruction.

EXAMPLE:

```
JSR CLRCHN
```

Function name: GETIN

Purpose: Get a character

Call address: \$FFE4 (hex) 65508 (decimal)

Communication registers: A

Preparatory routines: CHKIN, OPEN

Error returns: See READST

Stack requirements: 7+

Registers affected: A (X, Y)

Description: If the channel is the keyboard, this subroutine removes one character from the keyboard queue and returns it in the accumulator as an ASCII value. If the queue is empty, the value returned in the accumulator is zero. Characters are put into the queue automatically by an interrupt driven keyboard scan routine which calls the SCNKEY routine. The keyboard buffer can hold up to ten characters. If the buffer is full, additional characters are ignored until at least one character has been removed from the queue. If the channel is RS-232, then only the A register is used and a single character is returned. See READST to check validity. If the channel is serial, cassette, or screen, then call the CHRIN routine.

How to use:

1. Call this routine using a JSR instruction.
2. Check for a zero in the accumulator (empty buffer).
3. Process the data.

EXAMPLE:

```

                                ; WAIT FOR A CHARACTER
WAIT JSR GETIN                 ; GET A CHARACTER
CMP #$00                       ; COMPARE IT WITH ZERO
BEQ WAIT                       ; IT'S A ZERO! GO BACK AND GET ANOTHER

```

Function name: IOBASE

Purpose: Return location of start of I/O  
 Call address: \$FFF3 (hex) 65523 (decimal)  
 Communication registers: X, Y  
 Preparatory routines: None  
 Error returns: See READST  
 Stack requirements: 2  
 Registers affected: X, Y

Description: This routine sets the X and Y registers to the address of the memory section where the memory mapped I/O devices are located. This address can then be used with an offset to access the memory mapped I/O devices in the C16 and PLUS/4. The offset is the number of locations from the beginning of the page on which the required I/O register is located. The X register contains the low order address byte, and the Y register contains the high order address byte.

This routine exists to provide compatibility between COMMODORE machines. If the I/O locations for a machine language program are set by a call to this routine, they remain compatible with other versions of the KERNAL and BASIC.

How to use:

1. Call this routine by using the JSR instruction.
2. Store the X and Y registers in consecutive page zero locations.
3. Load the Y register with the offset.
4. Access that I/O location.

EXAMPLE:

```

JSR IOBASE
STX POINT                     ; SET BASE REGISTERS
STY POINT + 1                 ;
LDY #$02                      ; LOAD Y WITH OFFSET
LDA #$00                      ; LOAD A WITH DATA
STA (POINT),Y                 ; ACCESS MEMORY MAPPED I/O

```

Function name: IOINIT

Purpose: Initialize I/O devices  
Call address: \$FF84 (hex) 65412 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: None  
Registers affected: A, X, Y

Description: This routine initializes all input/output devices and routines.

EXAMPLE:

```
JSR IOINIT
```

Function name: LISTEN

Purpose: Command a device on the serial bus to listen  
Call address: \$FFB1 (hex) 65457 (decimal)  
Communication registers: A  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: None  
Registers affected: A

Description: This routine commands a device on the serial bus to receive data. The accumulator must be loaded with a device number between 0 and 31 before calling the routine. LISTEN ORs the number bit by bit to convert it to a listen address, then transmits this data as a command on the serial bus. The specified device then goes into listen mode ready to accept information.

How to use:

1. Load the accumulator with the number of the device to be commanded to LISTEN.
2. Call the LISTEN routine using the JSR instruction.

EXAMPLE:

```
                ; COMMAND DEVICE #8 TO LISTEN  
LDA #$08       ; LOAD ACCUMULATOR WITH DEVICE NO.  
JSR LISTEN     ; SEND COMMAND
```

Function name: LOAD

Purpose: Load RAM from device  
Call address: \$FFD5 (hex) 65493 (decimal)  
Communication registers: A, X, Y  
Preparatory routines: SETLFS, SETNAM  
Error returns: 0, 4, 5, 8, 9 (See READST)  
Stack requirements: None  
Registers affected: A, X, Y

Description: This routine LOADs data bytes from any input device directly into the memory of the C16 and PLUS/4. It can also be used to perform a verify operation, comparing data from a device with the data already in memory, while leaving the data stored in RAM unchanged.

The accumulator (A) must be set to 0 for a LOAD operation, or 1 for a verify. If the input device is OPENed with a secondary address (SA) of 0, the header information from the device is ignored. In this case, the X and Y registers must contain the starting address for the load. If the device is addressed with a secondary address of 1, then the data is loaded into memory starting at the location specified by the header. This routine returns the address of the highest RAM location loaded.

Before this routine can be called, the KERNAL SETLFS and SETNAM routines must be called.

NOTE: You can NOT LOAD from the keyboard (0), RS-232 (2), or the screen (3).

How to use:

1. Call the SETLFS and SETNAM routines. If a relocated load is desired, use the SETLFS routine to send a secondary address of 0.
2. Set the A register to 0 for load, 1 for verify.
3. If a relocated load is desired, the X and Y registers must be set to the start address for the load.
4. Call the routine using the JSR instruction.



## EXAMPLE:

```

; LOAD A FILE FROM TAPE
LDA #$01 ; SET DEVICE NUMBER (1 FOR TAPE)
LDX #FILENO ; SET LOGICAL FILE NUMBER
LDY #SA ; SET SECONDARY ADDRESS
JSR SETLFS
;
LDA #NAME1 - NAME ; LOAD A WITH NUMBER OF CHARACTERS
; IN FILE NAME
LDX #<NAME ; LOAD X WITH LOW ORDER BYTE OF
; FILE NAME ADDRESS
LDY #>NAME ; LOAD Y WITH HIGH ORDER BYTE OF
; FILE NAME ADDRESS
JSR SETNAM
;
LDA #$00 ; SET FLAG FOR A LOAD
LDX #$FF ; ALTERNATE START
LDY #$FF ; (NOT REQUIRED FOR THIS LOAD)
JSR LOAD ; LOAD FILE
;
STX VARTAB ; STORE ENDING ADDRESS IN POINTER
STY VARTAB + 1 ; FOR START OF BASIC VARIABLES
JMP START ; BEGIN EXECUTION
;
NAME .ASC "FILE NAME" ; FILE NAME IN ASCII BYTES
NAME1 NOP ; LENGTH OF FILENAME (NAME1 - NAME)

```

Function name: MEMBOT

Purpose: Read/set bottom of memory  
 Call address: \$FF9C (hex) 65436 (decimal)  
 Communication registers: X, Y  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: None  
 Registers affected: X, Y

Description: This routine is used to set the bottom of memory. If the accumulator carry bit is set when this routine is called, a pointer to the lowest byte of RAM is returned in the X and Y registers. On the C16 and PLUS/4 the initial value of this pointer is \$1000, i.e. 4096 in decimal. If the accumulator carry bit is clear, i.e. = 0, when this routine is called, the values of the X and Y registers are transferred to the low and high bytes respectively of the beginning of RAM pointer.

How to use:

TO READ THE BOTTOM OF RAM

1. Set the carry.
2. Call the MEMBOT routine.

TO SET THE BOTTOM OF RAM

1. Clear the carry.
2. Call the MEMBOT routine.

EXAMPLE:

```

                ; MOVE BOTTOM OF MEMORY UP 1 PAGE
SEC             ; SET CARRY
JSR MEMBOT     ; READ MEMORY BOTTOM
INY            ; INCREMENT Y (HIGH ORDER BYTE OF POINTER)
CLC            ; CLEAR CARRY
JSR MEMBOT     ; SET MEMORY BOTTOM TO NEW VALUE

```

Function name: MEMTOP

Purpose: Read/set top of memory

Call address: \$FF99 (hex) 65433 (decimal)

Communication registers: X, Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: X, Y

Description: This routine is used to set the top of RAM. When this routine is called with the carry bit of the accumulator set, the pointer to the top of RAM is loaded into the X and Y registers. When this routine is called with the accumulator carry bit clear, i.e. = 0, the contents of the X and Y registers are loaded into the top of memory pointer, changing the top of memory.

EXAMPLE:

```

                ; BRING DOWN THE TOP OF MEMORY
SEC             ; SET CARRY
JSR MEMTOP     ; READ TOP OF MEMORY
DEX            ; DECREMENT X REGISTER
CLC            ; CLEAR CARRY
JSR MEMTOP     ; SET NEW TOP OF MEMORY

```

Function name: OPEN

Purpose: Open a logical file  
 Call address: \$FFC0 (hex) 65472 (decimal)  
 Communication registers: None  
 Preparatory routines: SETLFS, SETNAM  
 Error returns: 1, 2, 4, 5, 6, 240 (See READST)  
 Stack requirements: None  
 Registers affected: A, X, Y

Description: This routine is used to OPEN a logical file. Once the logical file is set up, it can be used for input/output operations. Most of the I/O KERNAL routines require this routine to create the logical files on which they operate. No arguments are needed to set up this routine, but both the SETLFS and SETNAM routines must be called before using this routine.

How to use:

1. Use the SETLFS routine.
2. Use the SETNAM routine.
3. Call this routine.

EXAMPLE:

The following routine emulates the BASIC statement: OPEN 15,8,15,"I/O"

```
LDA #NAME1 - NAME ; LENGTH OF FILE NAME FOR SETLFS
LDX #<NAME
LDY #>NAME      ; ADDRESS OF FILE NAME
JSR SETNAM
LDA #$0F
LDX #$08
LDY #$0F
JSR SETLFS
JSR OPEN

NAME .ASC "I/O"
NAME1 NOP
```

Function name: PLOT

Purpose: Read/set cursor location  
 Call address: \$FFF0 (hex) 65520 (decimal)  
 Communication registers: A, X, Y  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: A, X, Y

Description: A call to this routine, with the accumulator carry flag set, loads the current position of the screen cursor, in X, Y coordinates, into the Y and X registers. The Y register contains the column that the cursor is in (0-39), and the X register contains the row that the cursor is on (0-24). A call with the carry bit clear positions the cursor at the X, Y location held in the Y and X registers.

How to use:

#### READING CURSOR LOCATION

1. Set the carry flag.
2. Call this routine.
3. Get the X and Y position from the Y and X registers, respectively.

#### SETTING CURSOR LOCATION

1. Clear the carry flag.
2. Set the Y and X registers to the desired cursor location.
3. Call this routine.

EXAMPLE:

```

                ; MOVE THE CURSOR TO ROW 10, COLUMN 5 (5,10)
LDX #$0A      ; LOAD X REGISTER WITH 10
LDY #$05      ; LOAD Y REGISTER WITH 5
CLC           ; CLEAR CARRY
JSR PLOT
  
```

Function name: RAMTAS

Purpose: Perform RAM test  
 Call address: \$FF87 (hex) 65415 (decimal)  
 Communication registers: A, X, Y  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: A, X, Y

Description: This routine is used to test RAM and set the top and bottom of memory pointers accordingly.

EXAMPLE:

```

JSR RAMTAS
  
```

Function name: RDTIM

Purpose: Read system clock  
Call address: \$FFDE (hex) 65502 (decimal)  
Communication registers: A, X, Y  
Preparatory routines: None  
Error returns: None  
Stack requirements: 2  
Registers affected: A, X, Y

Description: This routine is used to read the system clock. The clock's resolution is 1/60th of a second. Three bytes are returned by the routine. The accumulator contains the most significant byte, the X index register contains the next most significant byte, and the Y index register contains the least significant byte.

EXAMPLE:

```
JSR RDTIM
STY TIME
STX TIME + 1
STA TIME + 2
```

Function name: READST

Purpose: Read status word  
Call address: \$FFB7 (hex) 65463 (decimal)  
Communication registers: A  
Preparatory routines: None  
Error returns: None  
Stack requirements: 2  
Registers affected: A

Description: This routine returns the current status of the I/O devices in the accumulator. The routine is usually called after new communication to an I/O device. The routine gives you information about device status, or errors that have occurred during the I/O operation.

The bits returned in the accumulator contain the following information:

ST BIT POSITION	ST NUMERIC VALUE	CASSETTE READ	SERIAL/RW	TAPE VERIFY + LOAD
0	1		Time out write	
1	2		Time out read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	-128	End of tape	Device not present	End of tape

How to use:

1. Call this routine.
2. Decode the information in the A register

EXAMPLE:

```

                ; CHECK FOR END OF FILE DURING READ
JSR READST
AND #$40      ; AND WITH 64 TO CHECK EOF (END OF FILE) BIT
BNE EOF      ; BRANCH ON EOF

```

Function name: RESTOR

Purpose: Restore default system and interrupt vectors  
 Call address: \$FF8A (hex) 65418 (decimal)  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: A, X, Y

Description: This routine restores the default values of all system vectors used in KERNAL and BASIC routines and interrupts (see the Memory Map in Section 3.12 for a list of the vectors). The KERNAL VECTOR routine is used to read and alter individual system vectors.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR RESTOR
```

Function name: SAVE

Purpose: Save memory to a device  
 Call address: \$FFD8 (hex) 65496 (decimal)  
 Communication registers: A, X, Y  
 Preparatory routines: SETLFS, SETNAM  
 Error returns: 5, 8, 9 (See READST)  
 Stack requirements: None  
 Registers affected: A, X, Y

Description: This routine saves a section of memory to a device. Memory is saved from an indirect address on page zero specified by the accumulator contents, to the address stored in the X and Y registers. It is then sent to a logical file on an input/output device. The SETLFS and SETNAM routines must be used before calling this routine. Note that a file name is not required when saving to device 1, the cassette unit. An attempt to save to any other device without using a file name results in an error.

NOTE: You cannot SAVE to device 0 (the keyboard), device 2 (RS-232), or device 3 (the screen). If this is attempted, an error occurs, and the SAVE is stopped.

How to use:

1. Use the SETLFS and SETNAM routines.
2. Load two consecutive locations in page zero with a pointer to the start of the save, low byte first, high byte next.
3. Load the accumulator with the single byte page zero offset to the pointer.
4. Load the X and Y registers with the low byte and high byte, respectively, of the location of the end of the end of the save.
5. Call the SAVE routine.

EXAMPLE:

```
LDA #$01      ; DEVICE = 1, CASSETTE
JSR SETLFS
LDA #$00      ; NO FILE NAME
JSR SETNAM
LDA PROG      ; LOAD START ADDRESS OF SAVE
STA TXTTAB    ; (LOW BYTE)
LDA PROG+1
STA TXTTAB+1  ; (HIGH BYTE)
LDX VARTAB    ; LOAD X WITH LOW BYTE OF END OF SAVE
LDY VARTAB+1  ; LOAD Y WITH HIGH BYTE OF END OF SAVE
LDA #<TXTTAB ; LOAD ACCUMULATOR WITH PAGE ZERO OFFSET
JSR SAVE
```



Function name: SCNKEY

Purpose: Scan the keyboard

Call address: \$FF9F (hex) 65439 (decimal)

Communication registers: None

Preparatory routines: IOINIT

Error returns: None

Stack requirements: 5

Registers affected: A, X, Y

Description: This routine scans the Cl6 and PLUS/4 keyboard and checks for pressed keys. This is the routine which is called by the interrupt handler. If a key is down, its ASCII value is placed in the keyboard queue. This routine is called only if the normal IRQ interrupt is bypassed.

How to use:

1. Call this routine.

EXAMPLE:

```
GET JSR SCNKEY ; SCAN KEYBOARD
    JSR GETIN  ; GET CHARACTER
    CMP #$00   ; IS IT NULL?
    BEQ GET    ; YES...SCAN AGAIN
    JSR CHROUT ; PRINT IT
```

Function name: SCREEN

Purpose: Return screen format

Call address: \$FFED (hex) 65517 (decimal)

Communication registers: X, Y

Preparatory routines: None

Stack requirements: 2

Registers affected: X, Y

Description: This routine returns the format of the screen, e.g., 40 columns in X and 25 lines in Y. The routine can be used to determine what machine a program is running on. This function has been included on the Cl6 and PLUS/4 to help upward compatibility of your programs.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR SCREEN
STX MAXCOL
STY MAXROW
```

Function name: SECOND

Purpose: Send secondary address for LISTEN  
Call address: \$FF93 (hex) 65427 (decimal)  
Communication registers: A  
Preparatory routines: LISTEN  
Error returns: See READST  
Stack requirements: 8  
Registers affected: A

Description: This routine is used to send a secondary address to an I/O device after a call to the LISTEN routine is made. This routine can NOT be used to send a secondary address after a call to the TALK routine.

A secondary address is usually used to give setup information to a device before I/O operations begin. When a secondary address is to be sent to a device on the serial bus, the address must first be ORed with #\$60.

How to use:

1. Load the accumulator with the secondary address to be sent.
2. Call this routine.

EXAMPLE:

```
                ; ADDRESS DEVICE #8 WITH COMMAND  
                ; (SECONDARY ADDRESS) #15  
LDA #$08  
JSR LISTEN  
LDA #$0F  
JSR SECOND
```

Function name: SETLFS

Purpose: Set up a logical file  
Call address: \$FFBA (hex) 65466 (decimal)  
Communication registers: A, X, Y  
Preparatory routines: None  
Error returns: None  
Stack requirements: 2  
Registers affected: None

Description: This routine sets the logical file number, device address, and secondary address, i.e. command number, for other KERNAL routines.

The logical file number is used by the system as a key to the file table created by the OPEN file routine. Device addresses can range from 0 to 31. The codes used by the C16 and PLUS/4 for the CBM devices are listed below:

ADDRESS	DEVICE
0	Keyboard
1	Cassette unit
2	RS-232C device
3	CRT display
4	Serial bus printer
8	CBM serial bus disk drive

Device numbers greater than 3 automatically refer to devices on the serial bus.

The device number is sent during the serial attention handshaking sequence, then a command to the device is sent as a secondary address on the serial bus. If no secondary address is to be sent, the Y index register must be set to 255.

How to use:

1. Load the X index register with the logical file number.
2. Load the accumulator with the device number.
3. Load the Y index register with the command.

EXAMPLE:

```

                                ; FOR LOGICAL FILE 32, DEVICE #4, NO COMMAND
LDA #$20                        ; HEX FOR 32
LDX #$04
LDY #$FF                        ; HEX FOR 255
JSR SETLFS

```

Function name: SETMSG

Purpose: Control system message output  
 Call address: \$FF90 (hex) 65424 (decimal)  
 Communication registers: A  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: A

Description: This routine controls the printing of error and control messages by the KERNAL. Either error messages or control messages can be selected by setting the accumulator when the routine is called. FILE NOT FOUND is an example of an error message, PRESS PLAY ON TAPE is an example of a control message.

Bits 6 and 7 of this value determine whether control or error messages are displayed. If bit 7 is set, the KERNAL error messages are printed. If bit 6 is set, control messages are printed.

How to use:

1. Set accumulator to desired value.
2. Call this routine.

EXAMPLE:

```
LDA #$40
JSR SETMSG ; TURN ON CONTROL MESSAGES
LDA #$80
JSR SETMSG ; TURN ON ERROR MESSAGES
LDA #$00
JSR SETMSG ; TURN OFF ALL KERNAL MESSAGES
```

Function name: SETNAM

Purpose: Set file name

Call address: \$FFBD (hex) 65469 (decimal)

Communication registers: A, X, Y

Preparatory routines: SETLFS

Stack requirements: None

Registers affected: A, X, Y

Description: This routine is used to set up the file name for OPEN, SAVE, or LOAD routines. The accumulator must be loaded with the length of the file name. The X and Y registers must be loaded with the address of the file name, low byte first, high byte next. This address can be any valid memory address in the system. If no file name is desired, the accumulator must be set to 0, representing a zero file length.

How to use:

1. Load the accumulator with the length of the file name.
2. Load the X index register with the low order address of the file name.
3. Load the Y index register with the high order address of the file name.
4. Call this routine.

EXAMPLE:

```
LDA #NAME1 - NAME      ; LOAD LENGTH OF FILE NAME
LDX #<NAME             ; LOAD ADDRESS OF FILE NAME
LDY #>NAME
JSR SETNAM

;
NAME .ASC "FILENAME" ; FILE NAME IN ASCII FORMAT
NAME1 NOP
```

Function name: SETTIM

Purpose: Set the system clock  
 Call address: \$FFDB (hex) 65499 (decimal)  
 Communication registers: A, X, Y  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: None

Description: A system clock is maintained by an interrupt routine that updates it every 1/60th of a second, i.e. one "jiffy". The clock is three bytes long, which gives it the capability of counting up to 5,184,000 jiffies, i.e. 24 hours, at which point the clock resets to zero. Before calling this routine ensure that the accumulator contains the most significant byte, i.e. MSB, the X index register the next most significant byte, and the Y index register the least significant byte, i.e. LSB, of the initial time setting. These values must be in jiffies.

How to use:

1. Load the accumulator with the MSB of the 3-byte number to set the clock.
2. Load the X register with the next byte.
3. Load the Y register with the LSB.
4. Call this routine.

EXAMPLE:

```

                ; SET THE CLOCK TO 10 MINUTES = 36000 JIFFIES
LDA #$00      ; MOST SIGNIFICANT
LDX #$8C
LDY #$A0      ; LEAST SIGNIFICANT
JSR SETTIM
  
```

Function name: SETTMO

Purpose: Set IEEE bus card timeout flag  
 Call address: \$FFA2 (hex) 65442 (decimal)  
 Communication registers: A  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: 2  
 Registers affected: None

NOTE: This routine can only be used if you have an IEEE add-on card.

Description: This routine sets the timeout flag for the IEEE bus. When the timeout flag is set, the C16 or PLUS/4 waits for 64 milliseconds for a response from a device on the IEEE port. If the device does not respond to the Data Address Valid, i.e. DAV, signal within that time, the computer recognizes an error condition and aborts the handshake sequence. To enable timeouts, call this routine with bit 7 of the accumulator clear. To disable timeouts, call this routine with bit 7 of the accumulator set.

How to use:

TO SET THE TIMEOUT FLAG

1. Clear bit 7 of the accumulator.
2. Call this routine.

TO RESET THE TIMEOUT FLAG

1. Set bit 7 of the accumulator.
2. Call this routine.

EXAMPLE:

```

                ; DISABLE TIMEOUT
LDA #$00
JSR SETTMO

```

Function name: STOP

Purpose: Check if STOP key is pressed  
 Call address: \$FFE1 (hex) 65505 (decimal)  
 Communication registers: A  
 Preparatory routines: None  
 Error returns: None  
 Stack requirements: None  
 Registers affected: A, X

Description: If the STOP key is pressed during a UDTIM call, a call to the STOP routine returns with the Z flag set. In addition, the channels are reset to their default values. All other flags remain unchanged. If the STOP key is not pressed then the accumulator returns a byte representing the last row of the keyboard scan.

How to use:

1. Call UDTIM routine.
2. Call this routine.
3. Test for the zero flag.

EXAMPLE:

```

JSR UDTIM
JSR STOP ; SCAN FOR STOP
BNE NO ; KEY NOT DOWN
JMP READY ; STOP

```

Function name: TALK

Purpose: Command a device on the serial bus to TALK  
Call address: \$FFB4 (hex) 65460 (decimal)  
Communication registers: A  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: 8  
Registers affected: A

Description: To use this routine the accumulator must first be loaded with a device number between 0 and 31. The device number is ORed bit by bit to create a talk address. The data is then transmitted as a command on the serial bus.

How to use:

1. Load the accumulator with the device number.
2. Call this routine.

EXAMPLE:

```
                ; COMMAND DEVICE #4 TO TALK  
LDA #$04  
JSR TALK
```

Function name: TKSA

Purpose: Send secondary address to device commanded to TALK  
Call address: \$FF96 (hex) 65430 (decimal)  
Communication registers: A  
Preparatory routines: TALK  
Error returns: See READST  
Stack requirements: 8  
Registers affected: A

Description: This routine transmits a secondary address for a TALK device on the serial bus. It must be called with a number between 0 and 31 in the accumulator, the routine then sends this number as a secondary address over the serial bus. TKSA can only be called after a call to the TALK routine, it does not work after a LISTEN.

How to use:

1. Use the TALK routine.
2. Load the accumulator with the secondary address.
3. Call this routine.

EXAMPLE:

```
                ; TELL DEVICE #4 TO TALK WITH COMMAND #7  
LDA #$04  
JSR TALK  
LDA #$07  
JSR TKSA
```



Function name: UDTIM

Purpose: Update the system clock  
Call address: \$FFEA (hex) 65514 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: None  
Stack requirements: 2  
Registers affected: A, X

Description: This routine updates the system clock. This routine is usually called by the normal KERNAL interrupt routine every 1/60th of a second. However, if your program processes its own interrupts, this routine must be called to update the time. In addition, the STOP routine must be called if the STOP key is to remain functional.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR UDTIM
```

Function name: UNLSN

Purpose: Send an UNLISTEN command  
Call address: \$FFAE (hex) 65454 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: 8  
Registers affected: A

Description: This routine commands all devices on the serial bus to stop receiving data. A call to this routine sends an UNLISTEN command on the serial bus, only devices previously commanded to LISTEN are affected. This routine should be used when you have finished sending data to external devices.

How to use:

1. Call this routine.

EXAMPLE:

```
JSR UNLSN
```

Function name: UNTLK

Purpose: Send an UNTALK command  
Call address: \$FFAB (hex) 65451 (decimal)  
Communication registers: None  
Preparatory routines: None  
Error returns: See READST  
Stack requirements: 8  
Registers affected: A

Description: This routine transmits an UNTALK command on the serial bus. All devices previously set to TALK stop sending data when this command is received.

How to use:

1. Call this routine.

EXAMPLE:

JSR UNTALK

Function name: VECTOR

Purpose: Manage RAM vectors  
Call address: \$FF8D (hex) 65421 (decimal)  
Communication registers: X, Y  
Preparatory routines: None  
Error returns: None  
Stack requirements: 2  
Registers affected: A, X, Y

Description: This routine manages all system vector jump addresses stored in RAM. Calling this routine with the accumulator carry bit set, causes the current contents of the RAM vectors to be stored in a list pointed to by the X and Y registers. When this routine is called with the carry bit clear, the user list pointed to by the X and Y registers is transferred to the system RAM vectors. The RAM vectors are listed in the memory map (see Section 3.12).

NOTE: Use this routine with caution. COMMODORE recommend that you use it by transferring the entire vector contents into the user area, altering the desired vectors, and then copying the contents back to the system vectors.

How to use:

READ THE SYSTEM RAM VECTORS

1. Set the carry bit.
2. Set the X and Y registers to the required address.
3. Call this routine.

LOAD THE SYSTEM RAM VECTORS

1. Clear the carry bit.
2. Set the X and Y registers to the address of the vector list in RAM.
3. Call this routine.

EXAMPLE:

```

                                ; CHANGE THE INPUT ROUTINES TO NEW SYSTEM
LDX #<USER
LDY #>USER
SEC
JSR VECTOR ; READ OLD VECTORS
LDA #<MYINP ; CHANGE INPUT
STA USER+10
LDA #>MYINP
STA USER+11
LDX #<USER
LDY #>USER
CLC
JSR VECTOR ; ALTER SYSTEM
                                ;
USER
```

### 3.11.4 ERROR CODES

The following is a list of error messages which can occur when using the KERNAL routines. If an error occurs during a KERNAL routine, the carry bit of the accumulator is set and the number of the error message is returned in the accumulator.

NOTE: some KERNAL I/O routines do not use these codes for error messages. Instead, errors are identified using the KERNAL READST routine.

NUMBER	MEANING
0	Routine terminated by the STOP key
1	Too many open files
2	File already open
3	File not open
4	File not found
5	Device not present
6	File is not an input file
7	File is not an output file
8	File name is missing
9	Illegal device number
240	Top of memory change RS-232 buffer allocation/deallocation

## 3.12 C16 AND PLUS/4 MEMORY MAP

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
PDIR	\$0000	0	7510 on-chip data-direction register (DDR)
PORT	\$0001	1	7510 on-chip 8-bit input/output register
SRCHTK	\$0002	2	Token 'search' looks for (run-time stack)
ZPVEC1	\$0003-0004	3-4	Temp (renumber)
ZPVEC2	\$0005-0006	5-6	Temp (renumber)
CHARAC	\$0007	7	Search character
ENDCHR	\$0008	8	Flag: scan for quote at end of string
TRMPOS	\$0009	9	Screen column from last TAB
VERCK	\$000A	10	Flag: 0 = load 1 = verify
COUNT	\$000B	11	Input buffer pointer / No. of subscripts
DIMFLG	\$000C	12	Default array DIMension
VALTYP	\$000D	13	Data type: \$FF = string \$00 = numeric
INTFLG	\$000E	14	Data type: \$80 = integer \$00 = floating
DORES	\$000F	15	Flag: DATA scan/LIST quote/garbage collect
SUBLEG	\$0010	16	Flag: subscript ref / user function call
INPFLG	\$0011	17	Flag: \$00 = INPUT, \$40 = GET, \$98 = READ
TANSGN	\$0012	18	Flag: TAN sign / comparison result
CHANNL	\$0013	19	Flag: INPUT prompt
LINNUM	\$0014-0015	20-21	Temp: integer value
TEMPPT	\$0016	22	Pointer: temporary string stack
LASTPT	\$0017-0018	23-24	Last temp string address
TEMPST	\$0019-0021	25-33	Stack for temporary strings
INDEX1	\$0022-0023	34-35	Utility pointer area
INDEX2	\$0024-0025	36-37	Utility pointer area
RESHO	\$0026	38	
RESMOH	\$0027	39	
RESMO	\$0028	40	
RESLO	\$0029	41	
	\$002A	42	
TXTTAB	\$002B-002C	43-44	Pointer: start of BASIC text
VARTAB	\$002D-002E	45-46	Pointer: start of BASIC variables
ARYTAB	\$002F-0030	47-48	Pointer: start of BASIC arrays
STREND	\$0031-0032	49-50	Pointer: end of BASIC arrays (+1)

## Section Three

## Programming Machine Code

FRETOP	\$0033-0034	51-52	Pointer: bottom of string storage
FRESPC	\$0035-0036	53-54	Utility string pointer
MEMSIZ	\$0037-0038	55-56	Pointer: highest address used by BASIC
CURLIN	\$0039-003A	57-58	Current BASIC line number
TXTPTR	\$003B-003C	59-60	
FNDPNT	\$003D-003E	61-62	
DATLIN	\$003F-0040	63-64	Current DATA line number
DATPTR	\$0041-0042	65-66	Pointer: current DATA item address
INPPTR	\$0043-0044	67-68	Vector: INPUT routine
VARNAM	\$0045-0046	69-70	Current BASIC variable name
VARPNT	\$0047-0048	71-72	Pointer: current BASIC variable data
FORPNT	\$0049-004A	73-74	Pointer: index variable for FOR/NEXT
OPPTR	\$004B-004C	75-76	
OPMASK	\$004D	77	
DEFPNT	\$004E-004F	78-79	
DSCPNT	\$0050-0051	80-81	
	\$0052	82	
HELPER	\$0053	83	
JMPER	\$0054	84	
SIZE	\$0055	85	
OLDOV	\$0056	86	
TEMPFL	\$0057	87	
HIGHDS	\$0058-0059	88-89	
HIGHTR	\$005A-005B	90-91	
	\$005C	92	
LOWDS	\$005D-005E	93-94	
LOWTR	\$005F	95	
EXPSGN	\$0060	96	
FACEXP	\$0061	97	Floating-point accumulator #1: exponent
FACHO	\$0062	98	Floating accum. #1: mantissa
FACMOH	\$0063	99	
FACMO	\$0064	100	
FACLO	\$0065	101	
FACSGN	\$0066	102	Floating accum. #1: sign
SGNFLG	\$0067	103	Pointer: series evaluation constant
BITS	\$0068	104	Floating accum. #1: overflow digit
ARGEXP	\$0069	105	Floating-point accumulator #2: exponent
ARGHO	\$006A	106	Floating accum. #2: mantissa
ARGMOH	\$006B	107	
ARGMO	\$006C	108	
ARGLO	\$006D	109	
ARGSGN	\$006E	110	Floating accum. #2: sign
ARISGN	\$006F	111	Sign comparison result: accum. #1 vs #2

## Section Three

## Programming Machine Code

FACOV	\$0070	112	Floating accum. #1 low order (rounding)
FBUFPT	\$0071-0072	113-114	Pointer: cassette buffer
AUTINC	\$0073-0074	115-116	Increment value for AUTO (0 = off)
MVDFLG	\$0075	117	Flag if 10K hires allocated
KEYNUM	\$0076	118	
KEYSIZ	\$0077	119	
SYNTMP	\$0078	120	Used as temp for indirect loads
DSDESC	\$0079-007B	121-123	Descriptor for ds\$
TOS	\$007C-007D	124-125	Top of run-time stack
TMPTON	\$007E-007F	126-127	Temps used by music (tone and volume)
VOICNO	\$0080	128	
RUNMOD	\$0081	129	
POINT	\$0082	130	
GRAPHM	\$0083	131	Current graphic mode
COLSEL	\$0084	132	Current colour selected
MCL	\$0085	133	Multicolour1
FG	\$0086	134	Foreground colour
SCXMAX	\$0087	135	Maximum # of columns
SCYMAX	\$0088	136	Maximum # of rows
LTFLAG	\$0089	137	Paint-left flag
RTFLAG	\$008A	138	Paint-right flag
STOPNB	\$008B	139	Stop paint if not BG (not same colour)
GRAPNT	\$008C-008D	140-141	
VTEMP1	\$008E	142	
VTEMP2	\$008F	143	
STATUS	\$0090	144	Kernal I/O status word: ST
STKEY	\$0091	145	Flag: STOP key / RVS key
SPVERR	\$0092	146	Temp
VERECK	\$0093	147	Flag: 0 = load 1 = verify
C3PO	\$0094	148	Flag: serial bus - output char buffered
BSOUR	\$0095	149	Buffered character for serial bus
YSAV	\$0096	150	Temp for basin
LDTND	\$0097	151	# of open files / index to file table
DFLTN	\$0098	152	Default input device (0)
DFLTO	\$0099	153	Default output (CMD) device (3)
MSGFLG	\$009A	154	Flag: \$80 = direct mode \$00 = program
SAL	\$009B	155	Tape pass 1 error log
SAH	\$009C	156	Tape pass 2 error log
EAL	\$009D	157	
EAH	\$009E	158	
T1	\$009F-00A0	159-160	Temp data area
T2	\$00A1-00A2	161-162	Temp data area
TIME	\$00A3-00A5	163-165	Real-time jiffy clock (approx) 1/60 sec

## Section Three

## Programming Machine Code

R2D2	\$00A6	166	Serial bus usage
TPBYTE	\$00A7	167	Byte to be written/read on/off tape
BSOUR1	\$00A8	168	Temp used by serial routine
FPVERR	\$00A9	169	
DCOUNT	\$00AA	170	
FNLEN	\$00AB	171	Length of current file name
LA	\$00AC	172	Current logical file number
SA	\$00AD	173	Current secondary address
FA	\$00AE	174	Current device number
FNADR	\$00AF-00B0	175-176	Pointer: current file name
ERRSUM	\$00B1	177	
STAL	\$00B2	178	I/O start address
STAH	\$00B3	179	
MEMUSS	\$00B4-00B5	180-181	Load RAM base
TAPEBS	\$00B6-00B7	182-183	Base pointer to cassette base
TMP2	\$00B8-00B9	184-185	
WRBASE	\$00BA-00BB	186-187	Pointer to data for tape writes
IMPARM	\$00BC-00BD	188-189	Pointer to immediate string for primms
FETPTR	\$00BE-00BF	190-191	Pointer to byte to be fetched in bank f routine
SEDSAL	\$00C0-00C1	192-193	Temp for scrolling
RVS	\$00C2	194	RVS field flag on
INDX	\$00C3	195	
LSXP	\$00C4	196	X position at start
LSTP	\$00C5	197	
SFDX	\$00C6	198	Flag: shift mode for print
CRSW	\$00C7	199	Flag: INPUT or GET from keyboard
PNT	\$00C8-00C9	200-201	Pointer: current screen line address
PNTR	\$00CA	202	Cursor column on current line
QTSW	\$00CB	203	Flag: editor in quote mode \$00 = no
SED1	\$00CC	204	Editor temp use
TBLX	\$00CD	205	Current cursor physical line number
DATA	\$00CE	206	Temp data area
INSRT	\$00CF	207	Flag: insert mode, >0 = # INSTs
	\$00D0-00D7	208-215	Area for use by speech software
	\$00D8-00E8	216-232	Area for use by application software
CIRSEG	\$00E9	233	Screen line link table / editor temps
USER	\$00EA-00EB	234-235	Screen editor colour IP



## Section Three

## Programming Machine Code

KEYTAB	\$00EC-00ED	236-237	Key scan table indirect
TMPKEY	\$00EE	238	
NDX	\$00EF	239	Index to keyboard queue
STPFLG	\$00F0	240	Pause flag
TO	\$00F1-00F2	241-242	Monitor ZP storage
CHRPTR	\$00F3	243	
BUFEND	\$00F4	244	
CHKSUM	\$00F5	245	Temp for checksum calculation
LENGTH	\$00F6	246	
PASS	\$00F7	247	Which pass we are doing str
TYPE	\$00F8	248	Type of block
USEKDY	\$00F9	249	(B.7=1)=> for wr, (B.6=1)=> for rd
XSTOP	\$00FA	250	Save xreg for quick stopkey test
CURBNK	\$00FB	251	Current bank configuration
XON	\$00FC	252	Char to send for a x-on
XOFF	\$00FD	253	Char to send for a x-off
SEDT2	\$00FE	254	Editor temporary use
LOFBUF	\$00FF	255	
FBUFR	\$0100-010F	256-271	
SAVEA	\$0110	272	Temp locations for...
SAVEY	\$0111	273	...save
SAVE	\$0112	274	...restore
COLKEY	\$0113-0122	275-289	Colour/luminance table in RAM
SYSSTK	\$0124-01FF	291-511	System stack
BUF	\$0200-0258	512-600	BASIC/monitor buffer
OLDLIN	\$0259-025A	601-602	BASIC storage
OLDTXT	\$025B-025C	603-604	BASIC storage
	\$025D-02AC	605-684	BASIC/DOS INTERFACE AREA
XCNT	\$025D		DOS loop counter
FNBUFR	\$025E-026D		Area for filename
DOSFIL	\$026E		DOS filename 1 length
DOSDS1	\$026F		DOS disk drive 1
DOSF1A	\$0270-0271		DOS filename 1 addr
DOSF2L	\$0272		DOS filename 2 length
DOSDS2	\$0273		DOS disk drive 2
DOSF2A	\$0274-0275		DOS filename 2 addr
DOSLA	\$0276		DOS logical address
DOSFA	\$0277		DOS physical addr
DOSSA	\$0278		DOS secondary address
DOSDID	\$0279-027A		DOS disk identifier
DIDCHK	\$027B		DOS DID flag

## Section Three

## Programming Machine Code

DOSSTR	\$027C		DOS output string buffer
DOSSPC	\$027D-02AC		Area used to build DOS string
Graphics Variables			
XPOS	\$02AD-02AE	685-686	Current x position
YPOS	\$02AF-02B0	687-688	Current y position
XDEST	\$02B1-02B2	689-690	X coordinate destination
YDEST	\$02B3-02B4	691-692	Y coordinate destination
XABS	\$02B5-02B6	693-694	
YABS	\$02B7-02B8	695-696	
XSGN	\$02B9-02BA	697-698	
YSGN	\$02BB-02BC	699-700	
FCT1	\$02BD-02BE	701-702	
FCT2	\$02BF-02C0	703-704	
ERRVAL	\$02C1-02C2	705-706	
LESSER	\$02C3	707	
GREATR	\$02C4	708	
ANGSGN	\$02C5	709	Sign of angle
SINVAL	\$02C6-02C7	710-711	Sine of value of angle
COSVAL	\$02C8-02C9	712-713	Cosine of value of angle
ANGCNT	\$02CA-02CB	714-715	Temps for angle distance routines
Start of multiply defined area #1			
	\$02CC	716	Placeholder
BNR	\$02CD	717	Pointer to begin no.
ENR	\$02CE	718	Pointer to end no.
DOLR	\$02CF	719	Dollar flag
FLAG	\$02D0	720	Comma flag
SWE	\$02D1	721	Counter
USGN	\$02D2	722	Sign exponent
UEXP	\$02D3	723	Pointer to exponent
VN	\$02D4	724	# of digits before decimal point
CHSN	\$02D5	725	Justify flag
VF	\$02D6	726	# of pos before decimal point (field)
NF	\$02D7	727	# of pos after decimal point (field)
POSP	\$02D8	728	+/- flag (field)
FESP	\$02D9	729	Exponent flag (field)
ETOF	\$02DA	730	Switch
CFORM	\$02DB	731	Char counter (field)
SNO	\$02DC	732	Sign no.
BLFD	\$02DD	733	Blank/star flag
BEGFD	\$02DE	734	Pointer to beginning of field
LFOR	\$02DF	735	Length of format
ENDFD	\$02E0	736	Pointer to end of field

## Section Three

## Programming Machine Code

XCENTR	\$02CC-02CD	716-717	
YCENTR	\$02CE-02CF	718-719	
XDIST1	\$02D0-02D1	720-721	
YDIST1	\$02D2-02D3	722-723	
XDIST2	\$02D4-02D5	724-725	
YDIST2	\$02D6-02D7	726-727	
	\$02D8-02D9	728-729	Placeholder
COLCNT	\$02DA	730	Characters column counter
ROWCNT	\$02DB	731	Characters row counter
STRCNT	\$02DC	732	
			Start of multiply defined area #2
XCORD1	\$02CC-02CD	716-717	
YCORD1	\$02CE-02CF	718-719	
BOXANG	\$02D0-02D1	720-721	Rotation angle
XCOUNT	\$02D2-02D3	722-723	
YCOUNT	\$02D4-02D5	724-725	
BXLENG	\$02D6-02D7	726-727	Length of a side
XCORD2	\$02D8-02D9	728-729	
YCORD2	\$02DA-02DB	730-731	
XCIRCL	\$02CC-02CD	716-717	Circle center, x coordinate
YCIRCL	\$02CE-02CF	718-719	Circle center, y coordinate
XRADIUS	\$02D0-02D1	720-721	X radius
YRADIUS	\$02D2-02D3	722-723	Y radius
ROTANG	\$02D4-02D5	724-725	Rotation angle
ANGBEG	\$02D8-02D9	728-729	Arc angle start
ANGEND	\$02DA-02DB	730-731	Arc angle end
XRCOS	\$02DC-02DD	732-733	X radius * cos (rotation angle)
YRSIN	\$02DE-02DF	734-735	Y radius * sin (rotation angle)
XRSIN	\$02E0-02E1	736-737	X radius * sin (rotation angle)
YRCOS	\$02E2-02E3	738-739	Y radius * cos (rotation angle)
			Start of multiply defined area #3
	\$02CC	716	Placeholder
KEYLEN	\$02CD	717	
KEYNXT	\$02CE	718	
STRSZ	\$02CF	719	String length
GETTYP	\$02D0	720	Replace string mode
STRPTR	\$02D1	721	String position counter
OLDBYT	\$02D2	722	Old bit map byte
NEWBYT	\$02D3	723	New string or bit map byte
	\$02D4	724	Placeholder

## Section Three

## Programming Machine Code

XSIZE	\$02D5-02D6	725-726	Shape column length
YSIZE	\$02D7-02D8	727-728	Shape row length
XSAVE	\$02D9-02DA	729-730	Temp for column length
STRADR	\$02DB-02DC	731-732	Save shape string descriptor
BITIDX	\$02DD	733	Bit index into byte
SAVSIZ	\$02DE-02E1	734-737	Temporary work locations
CHRPAG	\$02E4	740	High byte addr of char ROM for CHAR
BITCNT	\$02E5	741	Temp for gshape
SCALEM	\$02E6	742	Scale mode flag
WIDTH	\$02E7	743	Double width flag
FILFLG	\$02E8	744	Box fill flag
BITMSK	\$02E9	745	Temp for bit mask
NUMCNT	\$02EA	746	
TRCFLG	\$02EB	747	Flags trace mode
T3	\$02EC	748	
T4	\$02ED-02EE	749-750	
VTEMP3	\$02EF	751	Graphic temp storage
VTEMP4	\$02F0	752	
VTEMP5	\$02F1	753	
ADRAY1	\$02F2-02F3	754-755	Ptr to routine: convert float to integer
ADRAY2	\$02F4-02F5	756-757	Ptr to routine: convert integer to float
BNKVEC	\$02FE-02FF	766-767	Vector for function cartidge users
IERROR	\$0300-0301	768-769	Indirect Error (output error in X)
IMAIN	\$0302-0303	770-771	Indirect Main (system direct loop)
ICRNCH	\$0304-0305	772-773	Indirect Crunch (tokenization routine)
IQFLOP	\$0306-0307	774-775	Indirect List (char list)
IGONE	\$0308-0309	776-777	Indirect Gone (character dispatch)
IEVAL	\$030A-030B	778-779	Indirect Eval (symbol evaluation)
IESCLK	\$030C-030D	780-781	Escape token crunch
IES CPR	\$030E-030F	782-783	
IESCEX	\$0310-0311	784-785	
ITIME	\$0312-0313	786-787	
CINV	\$0314-0315	788-789	IRQ RAM vector
CBINV	\$0316-0317	790-791	Brk instr RAM vector
IOPEN	\$0318-0319	792-793	Indirects for code
ICLOSE	\$031A-031B	794-795	
ICKIN	\$031C-031D	796-797	
ICKOUT	\$031E-031F	798-799	
ICLRCH	\$0320-0321	800-801	
IBASIN	\$0322-0323	802-803	
IBSOUT	\$0324-0325	804-805	
ISTOP	\$0326-0327	806-807	

## Section Three

## Programming Machine Code

IGETIN	\$0328-0329	808-809	
ICLALL	\$032A-032B	810-811	
USRCMD	\$032C-032D	812-813	
ILOAD	\$032E-032F	814-815	
ISAVE	\$0330-0331	816-817	
TAPBUF	\$0333-03F2	819-1010	Cassette tape buffer
WRLLEN	\$03F3-03F4	1011-1012	Length of data to be written to tape
RDCNT	\$03F5-03F6	1013-1014	Length of data to be read from tape
INPQUE	\$03F7-0436	1015-1078	RS-232 input queue
ESTAKL	\$0437-0454	1079-1108	
ESTAKH	\$0455-0472	1109-1138	
CHRGET	\$0473-0478	1139-1144	
CHRGOT	\$0479-0484	1145-1156	
QNUM	\$0485-0493	1157-1171	
INDSUB	\$0494-04A1	1172-1185	Shared ROM fetch sub
ZERO	\$04A2-04A4	1186-1188	Numeric constant for BASIC
INDTXT	\$04A5-04AF	1189-1199	Txtptr
INDIN1	\$04B0-04BA	1200-1210	Index & Index1
INDIN2	\$04BB-04C5	1211-1221	Index2
INDST1	\$04C6-04D0	1222-1232	Strngl
INDLOW	\$04D1-04DB	1233-1243	Lowtr
INDFMO	\$04DC-04E6	1244-1254	Facmo
PUFILL	\$04E7	1255	Print using fill symbol
PUCOMA	\$04E8	1256	Print using comma symbol
PUDOT	\$04E9	1257	Print using D.P. symbol
PUMONY	\$04EA	1258	Print using monetary symbol
TMPDES	\$04EB-04EE	1259-1262	Temp for instr
ERRNUM	\$04EF	1263	Last error number
ERRLIN	\$04F0-04F1	1264-1265	Line # of last error
TRAPNO	\$04F2-04F3	1266-1267	Line to go on error
TMTTRP	\$04F4	1268	Hold trap no. temporarily
ERRTXT	\$04F5-04F6	1269-1270	
OLDSTK	\$04F7	1271	
TMPTXT	\$04F8-04F9	1272-1273	
TMPLIN	\$04FA-04FB	1274-1275	
MTIMLO	\$04FC-04FD	1276-1277	Table of pending jiffies (2's comp)
MTIMHI	\$04FE-04FF	1278-1279	
USRPOK	\$0500-0502	1280-1282	
RNDX	\$0503-0507	1283-1287	

## Section Three

## Programming Machine Code

DEJAVU	\$0508	1288	'Cold' or 'warm' start status
LAT	\$0509-0512	1289-1298	Logical file numbers
FAT	\$0513-051C	1299-1308	Primary device numbers
SAT	\$051D-0526	1309-1318	Secondary addresses
KEYD	\$0527-0530	1319-1328	IRQ keyboard buffer
MEMSTR	\$0531-0532	1329-1330	Start of memory
MSIZ	\$0533-0534	1331-1332	Top of memory
TIMOUT	\$0535	1333	IEEE timeout flag
FILEND	\$0536	1334	File end reached = 1, otherwise 0
CTALLY	\$0537	1335	# of chars left in buffer (for R & W)
CBUFVA	\$0538	1336	# of total valid chars in buffer (R)
TPTR	\$0539	1337	Ptr to next char in buffer (for R & W)
FLTYPE	\$053A	1338	Contains type of current cass file
COLOR	\$053B	1339	Active attribute byte
FLASH	\$053C	1340	Character flash flag
	\$053D	1341	FREE
HIBASE	\$053E	1342	Base location of screen (top)
XMAX	\$053F	1343	
RPTFLG	\$0540	1344	Key repeat flag
KOUNT	\$0541	1345	
DELAY	\$0542	1346	
SHFLAG	\$0543	1347	Shift flag byte
LSTSHF	\$0544	1348	Last shift pattern
KEYLOG	\$0545-0546	1349-1350	Indirect for keyboard table setup
MODE	\$0547	1351	
AUTODN	\$0548	1352	Auto scroll down flag (0 = on, 0 <> off)
LINTMP	\$0549	1353	
ROLFLG	\$054A	1354	
FORMAT	\$054B	1355	Monitor non-zpage storage
MSAL	\$054C-054E	1356-1358	
WRAP	\$054F	1359	
TMPC	\$0550	1360	
DIFF	\$0551	1361	
PCH	\$0552	1362	
PCL	\$0553	1363	
FLGS	\$0554	1364	
ACC	\$0555	1365	
XR	\$0556	1366	
YR	\$0557	1367	
SP	\$0558	1368	
INVL	\$0559	1369	
INVH	\$055A	1370	



## Section Three

## Programming Machine Code

CMPFLG	\$055B	1371	Used by various monitor routines
BAD	\$055C	1372	
KYNDX	\$055D	1373	Used for programmable keys
KEYIDX	\$055E	1374	
KEYBUF	\$055F-0566	1375-1382	Table of P.F. lengths
PKYBUF	\$0567-05E6	1383-1510	P.F. key storage area
KDATA	\$05E7	1511	Temp for data write to Kennedy
KDYCMD	\$05E8	1512	Select for Kennedy read or write
KDYNUM	\$05E9	1513	Kennedy's dev#
KDYPRS	\$05EA	1514	Kennedy present = \$FF, else \$00
KDYTYP	\$05EB	1515	Temp for type of open for Kennedy
SAVRAM	\$05EC-06EB	1516-1771	1 page used by banking routines
PAT	\$05EC-05EF	1516-1519	Physical address table
LNGJMP	\$05F0-05F1	1520-1521	Long jump address
FETARG	\$05F2	1522	Long jump accumulator
FETXRG	\$05F3	1523	Long jump x register
FETSRG	\$05F4	1524	Long jump status register
AREAS	\$05F5-065D	1525-1629	RAM areas for banking
ASPECH	\$065E-06EB	1630-1771	RAM area for speech
STKTOP	\$06EC-07AF	1772-1967	BASIC run-time stack
WROUT	\$07B0	1968	Byte to be written on tape
PARITY	\$07B1	1969	Temp for parity calc
TT1	\$07B2	1970	Temp for write-header
TT2	\$07B3	1971	Temp for write-header
RDBITS	\$07B5	1973	Local index for READBYTE routine
ERRSP	\$07B6	1974	Pointer into the error stack
FPERRS	\$07B7	1975	Number of first pass errors
DSAMP1	\$07B8-07B9	1976-1977	Time constant
DSAMP2	\$07BA-07BB	1978-1979	Time constant
ZCELL	\$07BC-07BD	1980-1981	Time constant
SRECOV	\$07BE	1982	Stack marker for stopkey recover
DRECOV	\$07BF	1983	Stack marker for dropkey recover
TRSAVE	\$07C0-07C3	1984-1987	Params passed to RDBLOK
RDETMP	\$07C4	1988	Temp stat save for RDBLOK



## Section Three

## Programming Machine Code

LDRSCN	\$07C5	1989	# consec shorts to find in leader
CDERRM	\$07C6	1990	# errors fatal in RD countdown
VSAVE	\$07C7	1991	Temp for VERIFY command
TPIPE	\$07C8-07CB	1992-1995	Pipe temp for Tl
ENEXT	\$07CC	1996	Read error propagate
For RS-232			
UOUTQ	\$07CD	1997	User character to send
UOUTFG	\$07CE	1998	0 = empty, 1 = full
SOUTQ	\$07CF	1999	System character to send
SOUNFG	\$07D0	2000	0 = empty, 1 = full
INQFPT	\$07D1	2001	Pntr to front of input queue
INQRPT	\$07D2	2002	Pntr to rear of input queue
INQCNT	\$07D3	2003	# of chars in input queue
ASTAT	\$07D4	2004	Temp status for ACIA
AINTMP	\$07D5	2005	Temp for input routine
ALSTOP	\$07D6	2006	FLG for local pause
ARSTOP	\$07D7	2007	FLG for remote pause
APRES	\$07D8	2008	FLG to indicate presence of ACIA
KLUDS	\$07D9-07E4	2009-2020	Indirect routine downloaded
SCBOT	\$07E5	2021	
SCTOP	\$07E6	2022	
SCLF	\$07E7	2023	
SCRT	\$07E8	2024	
SCRDIS	\$07E9	2025	
INSFLG	\$07EA	2026	
LSTCHR	\$07EB	2027	
LOGSCR	\$07EC	2028	
TCOLOR	\$07ED	2029	
BITABL	\$07EE-07F1	2030-2033	
SAREG	\$07F2	2034	Registers for SYS command
SXREG	\$07F3	2035	
SYREG	\$07F4	2036	
SPREG	\$07F5	2037	
LSTX	\$07F6	2038	Key scan index
STPDSB	\$07F7	2039	Flag to disable CTL-S pause
RAMROM	\$07F8	2040	MSB for monitor fetches from RAM = 0, ROM = 1
COLSW	\$07F9	2041	MSB for colour/luminance table in RAM = 0, ROM = 1
FFRMSK	\$07FA	2042	ROM mask for split screen
VMBMSK	\$07FB	2043	VM base mask for split screen
LSEM	\$07FC	2044	Motor lock semaphore for cassette

## Section Three

## Programming Machine Code

PALCNT	\$07FD	2045	PAL tod
TEDATR	\$0800-0BFF	2048-3071	TED attribute bytes (colour)
TEDSCN	\$0C00-0FFF	3072-4095	TED character pointers (screen)
BASBGN	\$1000-	4096-	Start of BASIC text area
GRBASE	\$2000-	8192-	Start of BASIC when HIRES is on
BMLUM	\$1800-1BFF	6144-7167	Luminance for bit map screen
BMCOLR	\$1C00-1FFF	7168-8191	Colour for bit map
CHRBAS	\$D000	53248	Beginning of character ROM

## BANKING JUMP TABLE

\$FCF1	64753	JMP to cartridge IRQ routine
\$FCF4	64756	JMP to PHOENIX routine
\$FCF7	64759	JMP to LONG FETCH routine
\$FCFA	64762	JMP to LONG JUMP routine
\$FCFD	64765	JMP to LONG IRQ routine

## UNOFFICIAL JUMP TABLE

\$FF49	65353	JMP to define function key routine
\$FF4C	65356	JMP to PRINT routine
\$FF4F	65359	JMP to PRIMM routine
\$FF52	65362	JMP to ENTRY routine
\$FF80	65408	Release # of KERNAL (MSB 0 = NTSC, 1 = PAL)

## KERNAL JUMP TABLE

CINT	\$FF81	65409	Initialize screen editor
IOINIT	\$FF84	65412	Initialize I/O devices
RAMTAS	\$FF87	65415	RAM test
RESTOR	\$FF8A	65418	Restore vectors to initial values
VECTOR	\$FF8D	65421	Change vectors for user
SETMSG	\$FF90	65424	Control O.S. messages
SECND	\$FF93	65427	Send SA after LISTEN
TKSA	\$FF96	65430	Send SA after TALK
MEMTOP	\$FF99	65433	Set/read top of memory
MEMBOT	\$FF9C	65436	Set/read bottom of memory
SCNKEY	\$FF9F	65439	Scan keyboard

## Section Three

## Programming Machine Code

SETTMO	\$FFA2	65442	Set timeout in DMA disk
ACPTR	\$FFA5	65445	Handshake serial bus or DMA disk byte in
CIOUT	\$FFA8	65448	Handshake serial bus or DMA disk byte out
UNTLK	\$FFAB	65451	Send UNTALK out serial bus or DMA disk
UNLSN	\$FFAE	65454	Send UNLISTEN out serial bus or DMA disk
LISTN	\$FFB1	65457	Send LISTEN out serial bus or DMA disk
TALK	\$FFB4	65460	Send TALK out serial bus or DMA disk
READST	\$FFB7	65463	Return I/O STATUS byte
SETLFS	\$FFBA	65466	Set LA, FA, SA
SETNAM	\$FFBD	65469	Set length and FN address
OPEN	\$FFC0	65472	Open logical file
CLOSE	\$FFC3	65475	Close logical file
CHKIN	\$FFC6	65478	Open channel in
CHKOUT	\$FFC9	65481	Open channel out
CLRCH	\$FFCC	65484	Close I/O channels
BASIN	\$FFCF	65487	Input from channel
BSOUT	\$FFD2	65490	Output to channel
LOADSP	\$FFD5	65493	Load from file
SAVESP	\$FFD8	65496	Save to file
SETTIM	\$FFDB	65499	Set internal clock
RDTIM	\$FFDE	65502	Read internal clock
STOP	\$FFE1	65505	Scan STOP key
GETIN	\$FFE4	65508	Get character from queue
CLALL	\$FFE7	65511	Close all files
UDTIM	\$FFEA	65514	Increment clock
SCRORG	\$FFED	65517	Screen org
PLOT	\$FFF0	65520	Read/set X,Y coord of cursor
IOBASE	\$FFF3	65523	Return location of start of I/O



APPENDIX A  
SCREEN DISPLAY CODES

The following chart lists all of the characters built into the C16 and PLUS/4 character sets. It shows which numbers should be POKed into screen memory, locations 3072 to 4071, to display a desired character.

Two character sets are available, but only characters from one set can be displayed at any one time. The sets are switched by holding down the <SHIFT> key and pressing the <C=> key.

From BASIC, PRINT CHR\$(142) switches to upper-case/graphics mode, and PRINT CHR\$(14) switches to upper/lower-case mode.

Any character on the chart may also be displayed in reverse. The reverse character code is obtained by adding 128 to the values shown.

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
"		0	T	t	20	(		40
A	a	1	U	u	21	)		41
B	b	2	V	v	22	.		42
C	c	3	W	w	23	+		43
D	d	4	X	x	24	,		44
E	e	5	Y	y	25	-		45
F	f	6	Z	z	26	:		46
G	g	7	[		27	/		47
H	h	8	£		28	0		48
I	i	9	]		29	1		49
J	j	10	↑		30	2		50
K	k	11	←		31	3		51
L	l	12	<b>SPACE</b>		32	4		52
M	m	13	!		33	5		53
N	n	14	"		34	6		54
O	o	15	#		35	7		55
P	p	16	\$		36	8		56
Q	q	17	%		37	9		57
R	r	18	&		38	:		58
S	s	19			39	;		59

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
		60		T	84			108
		61		U	85			109
		62		V	86			110
		63		W	87			111
		64		X	88			112
	A	65		Y	89			113
	B	66		Z	90			114
	C	67			91			115
	D	68			92			116
	E	69			93			117
	F	70			94			118
	G	71			95			119
	H	72	<b>SPACE</b>		96			120
	I	73			97			121
	J	74			98			122
	K	75			99			123
	L	76			100			124
	M	77			101			125
	N	78			102			126
	O	79			103			127
	P	80			104			
	Q	81			105			
	R	82			106			
	S	83			107			

Codes from 128-255 are reversed images of codes 0-127.











## APPENDIX B

## ASCII AND CHR\$ CODES

This appendix shows the characters that appear if you PRINT CHR\$(X) for all possible values of X. It also shows the values obtained by typing PRINT ASC("X"), where X is a character. This is useful for evaluating the character received in a GET statement, converting upper/lower-case, and printing character based commands, like switch to upper/lower-case, that can not be enclosed in quotes.

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	0	↓	17	"	34	3	51
	1	RVS ON	18	#	35	4	52
	2	CLEAR HOME	19	\$	36	5	53
	3	DNST DEL	20	%	37	6	54
	4		21	&	38	7	55
WRT	5		22	'	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES SHIFT Cx	8		25	*	42	;	59
ENABLES SHIFT Cx	9		26	+	43	<	60
	10	ESCAPE	27	.	44	=	61
	11	RED	28	-	45	>	62
	12	→	29	_	46	?	63
RETURN	13	GRN	30	/	47	@	64
SWITCH TO LOWER CASE	14	BLU	31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
D	68		97		126	LT GREEN	155
E	69		98		127	PUR	156
F	70		99		128		157
G	71		100	ORANGE	129	YEL	158
H	72		101	FLASH ON	130	CYN	159
I	73		102		131	SPACE	160
J	74		103	FLASH OFF	132		161
K	75		104		133		162
L	76		105		134		163
M	77		106		135		164
N	78		107		136		165
O	79		108		137		166
P	80		109		138		167
Q	81		110		139		168
R	82		111		140		169
S	83		112	SHIFT RETURN	141		170
T	84		113	SWITCH TO UPPER CASE	142		171
U	85		114		143		172
V	86		115	BLK	144		173
W	87		116		145		174
X	88		117	SVS OFF	146		175
Y	89		118	CLEAR HOME	147		176
Z	90		119	INST DEL	148		177
	91		120	BROWN	149		178
£	92		121	YEL/GREEN	150		179
	93		122	PINK	151		180
†	94		123	BL/GREEN	152		181
†	95		124	LT BLUE	153		182
	96		125	BLUE	154		183

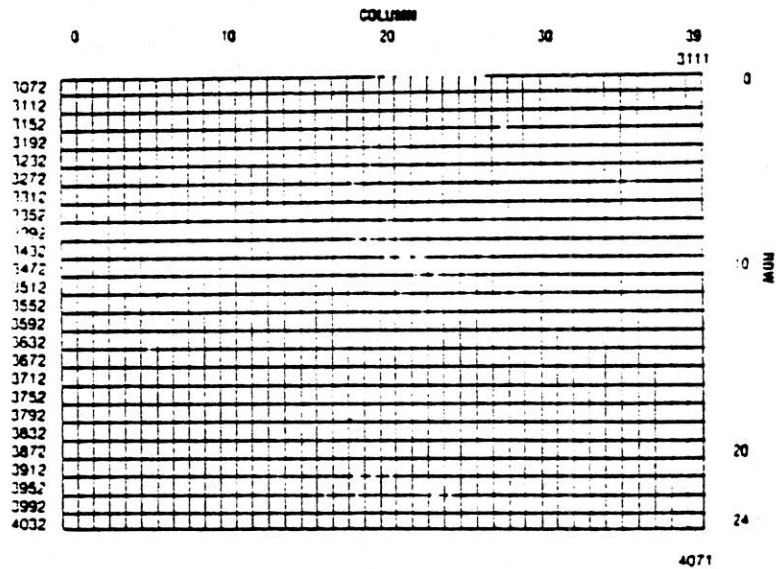
PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$
	184		186		188		190
	185		187		189		191
<b>CODES</b>	<b>192-223</b>	<b>SAME AS</b>	<b>96-127</b>				
<b>CODES</b>	<b>224-254</b>	<b>SAME AS</b>	<b>160-190</b>				
<b>CODE</b>	<b>255</b>	<b>SAME AS</b>	<b>126</b>				

APPENDIX C

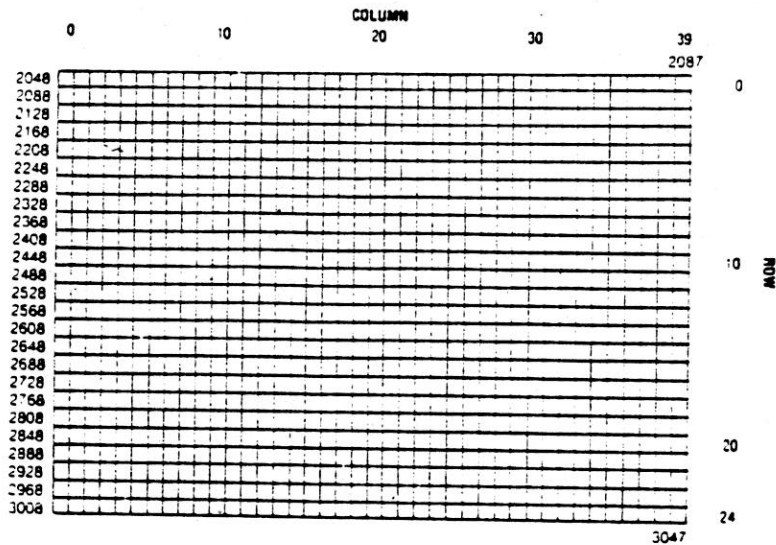
SCREEN AND COLOUR MEMORY MAPS

The following charts show the memory locations used by the screen, and the locations used to change individual character colours. A list of the character colour codes is also given in this appendix.

SCREEN MEMORY MAP



## COLOUR MEMORY MAP



The values to change a character's colour are:

0 BLACK	8 ORANGE
1 WHITE	9 BROWN
2 RED	10 YELLOW-GREEN
3 CYAN	11 PINK
4 PURPLE	12 BLUE-GREEN
5 GREEN	13 LIGHT BLUE
6 BLUE	14 DARK BLUE
7 YELLOW	15 LIGHT GREEN

The luminance of the colour is selected by multiplying the luminance value (0-7) by 16, and adding it to the colour number. To make a character flash, increase the colour value by 128.

## APPENDIX D

## DERIVING MATHEMATICAL FUNCTIONS

Functions that are not intrinsic to BASIC V3.5 may be calculated as follows:

FUNCTION	BASIC EQUIVALENT
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X*X+1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X*X+1)) + \text{PI}/2$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/\text{SQR}(X*X-1))$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X*X-1)) + (\text{SGN}(X) - 1) * \text{PI}/2$
INVERSE COTANGENT	$\text{ARCOT}(X) = \text{ATN}(X) + \text{PI}/2$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = \text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X)) * 2 + 1$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X)) * 2 + 1$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X*X+1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X*X-1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1+X)/(1-X))/2$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X*X+1)+1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X) * \text{SQR}(X*X+1))/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X+1)/(X-1))/2$

APPENDIX E  
MUSICAL NOTE TABLE

NOTE	SOUND REGISTER VALUE	ACTUAL FREQUENCY (HZ)
A	7	110
B	118	123.5
C	169	130.8
D	262	146.8
E	345	164.7
F	383	174.5
G	453	195.9
A	516	220.2
B	571	246.9
C	596	261.4
D	643	293.6
E	685	330
F	704	349.6
G	739	392.5
A	770	440.4
B	798	494.9
C	810	522.7
D	834	588.7
E	854	658
F	864	699
G	881	782.2
A	897	880.7
B	911	989.9
C	917	1045
D	929	1177
E	939	1316
F	944	1398
G	953	1575

The above table contains the sound register values for four octaves of notes. These values are used as the second parameter of the SOUND command (see Section 2.4.44). To play the first note in the table, use 7 as the second number in the SOUND command, i.e. SOUND 1,7,30. Use VOL 8 first to turn on sound.

The following formula allows you to calculate the sound register values for frequencies other than those in the table:

$$\text{SOUND REGISTER VALUE} = 1024 - (111860.781 / \text{FREQUENCY})$$

Both the table of sound register values and the above formula are for NTSC computers. This is the television standard used throughout the United States and Canada. If you are in a country where PAL is the television standard, use the following formula to calculate new sound register values:

$$\text{SOUND REGISTER VALUE} = 1024 - (111840.45 / \text{FREQUENCY})$$



APPENDIX F  
ERROR MESSAGES

These error messages are printed by BASIC. You can PRINT these messages by using the ERR\$( ) function.

ERROR#	ERROR NAME	
1	TOO MANY FILES	There is a limit of 10 files OPEN at one time
2	FILE OPEN	Attempt 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	Required I/O device not available
6	NOT INPUT FILE	Attempt made to GET or INPUT data from a file that was specified as output only
7	NOT OUTPUT FILE	Attempt made to send data to a file that was specified as input only
8	MISSING FILE NAME	An OPEN, LOAD, or SAVE to disk drive generally requires a file name
9	ILLEGAL DEVICE NUMBER	Attempt made to use a device improperly (SAVE to the screen, etc.)
10	NEXT WITHOUT FOR	Either loops are nested incorrectly, or the variable in the NEXT statement does not correspond with the one in the FOR statement
11	SYNTAX	Statement is unrecognizable by BASIC. This could be because of missing or extra parentheses, misspelled keywords, etc.

12	RETURN WITHOUT GOSUB	RETURN statement encountered when no GOSUB was active
13	OUT OF DATA	READ statement encountered with insufficient data in the program
14	ILLEGAL QUANTITY	A number used as the argument of a function or statement is outside the allowed 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 the program and program variables, or there are too many DO, FOR, or GOSUB statements in effect
17	UNDEF'D STATEMENT	A line number referenced does not 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. If an array is referenced before that array is DIM'd, an automatic DIM (to 10) is performed
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 by splitting it up, or using 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	UNDEF'D FUNCTION	A user defined function referenced does not exist in the program
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 pressed to halt program execution
31	CAN'T RESUME	RESUME statement encountered with 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 statement encountered without a DO statement active
34	DIRECT MODE ONLY	This command is allowed only in direct mode, not from a program
35	NO GRAPHICS AREA	A graphics command (DRAW, BOX, etc.) encountered before the GRAPHIC command was executed
36	BAD DISK	An attempt failed to HEADER a diskette, either because no ID was specified, or because the diskette is bad

## DESCRIPTION OF DOS ERROR MESSAGES

These error messages are returned through the DS and DS\$ reserved 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.

- |    |  |  |
|----|--|--|
| 20 | READ ERROR<br>(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 if the header has been destroyed.   |
| 21 | READ ERROR<br>(no sync character)            | The disk controller is unable to detect a sync mark on the desired track. Caused by misalignment of the read/write head, if no diskette is present, or an unformatted or improperly sealed diskette. Can also indicate a hardware failure. |
| 22 | READ ERROR<br>(data block not present)       | The disk controller has been requested to read or verify a data block that was not properly written. This error message occurs in conjunction with the BLOCK commands and indicates an illegal track and/or sector request.                |
| 23 | READ ERROR<br>(checksum error in data block) | This message indicates 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 wrong. This message may also indicate grounding problems.                                  |
| 24 | READ ERROR<br>(byte decoding error)          | The data or header has been read into the DOS memory, but a hardware error has occurred due to an invalid bit pattern in the data byte. This message may also indicate grounding problems.   |
| 25 | WRITE ERROR<br>(write-verify error)          | This message is generated when 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 is requested to write a data block while the write protect switch is depressed. Typically, this is caused by using a diskette with a write protect tab over the notch.
- 27 READ ERROR  
(checksum error in header)  
The controller has detected an error in the header of the requested data block. The block has not been read into the DOS memory. This message may also indicate grounding problems.
- 28 WRITE ERROR  
(long data block)  
The controller attempts to detect the sync mark of the next header after writing a data block. If the sync mark does not appear within a pre-determined time, the error message is generated. The error is caused by a bad diskette format (the data extends into the next block), or by hardware failure.
- 29 DISK ID MISMATCH  
This message is generated when the controller is 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 illegal patterns. For example, two files names 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 used incorrectly in the OPEN or SAVE command.

- 34 SYNTAX ERROR  
(no file given) The file name was left out of a command or the DOS does not recognize it as such. Typically, a colon (:) has been left out of the command.
- 39 SYNTAX ERROR  
(invalid command) This error may result if the command sent to the command channel, secondary address 15, is not recognized by the DOS.
- 50 RECORD NOT PRESENT Result of disk reading past the last record through INPUT#, or GET# commands. This message also occurs after positioning to a record beyond the end of 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 or 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 occurs if the total number of 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 an attempt is made to OPEN an unCLOSEed write file for reading.
- 61 FILE NOT OPEN This message is generated when an attempt is made to access an unOPENed file. Sometimes, a message is not generated, the request is simply ignored.
- 62 FILE NOT FOUND The requested file does not exist on the specified drive.
- 63 FILE EXISTS The file name of the file being created already exists on the diskette.

- 64 FILE TYPE MISMATCH The file type does not match the file type in the directory entry for the requested file.
- 65 NO BLOCK This message occurs in conjunction with the B-A command. It indicates that the block has already been allocated. The parameters indicate the track and sector available with the next highest number. If the parameters are zero (0), then all blocks higher in number are in use.
- 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 to 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 The BAM does not match the internal count. There is a problem in the BAM allocation or the BAM has been overwritten in DOS memory. To correct this problem, reinitialize the diskette to restore the BAM in memory. Some active file may be terminated by this corrective action. NOTE: BAM is the Block Availability Map.
- 72 DISK FULL Either the blocks on the diskette have been used up, or the directory is at its entry limit. DISK FULL is sent when two blocks are available on the 1541 to allow the current file to be closed.



- 73    DOS MISMATCH (73,  
      CBM DOS V2.6 1541)    DOS 1 and 2 are read, but not  
                              write compatible. Disks may be  
                              read with either DOS, but a disk  
                              formatted on one version cannot  
                              be written to with the other  
                              version because the format is  
                              different. This error is  
                              displayed whenever an attempt is  
                              made to write to a disk which has  
                              been formatted in a non-  
                              compatible format. This message  
                              may also appear after power up.
- 74    DRIVE NOT READY    An attempt has been made to  
                              access the floppy disk drive  
                              without a diskette present.

APPENDIX G

C16 AND PLUS/4 SCHEMATIC DIAGRAMS

=

THIS PART HAS NOT  
BEEN FINISHED!  
BAD QUALITY