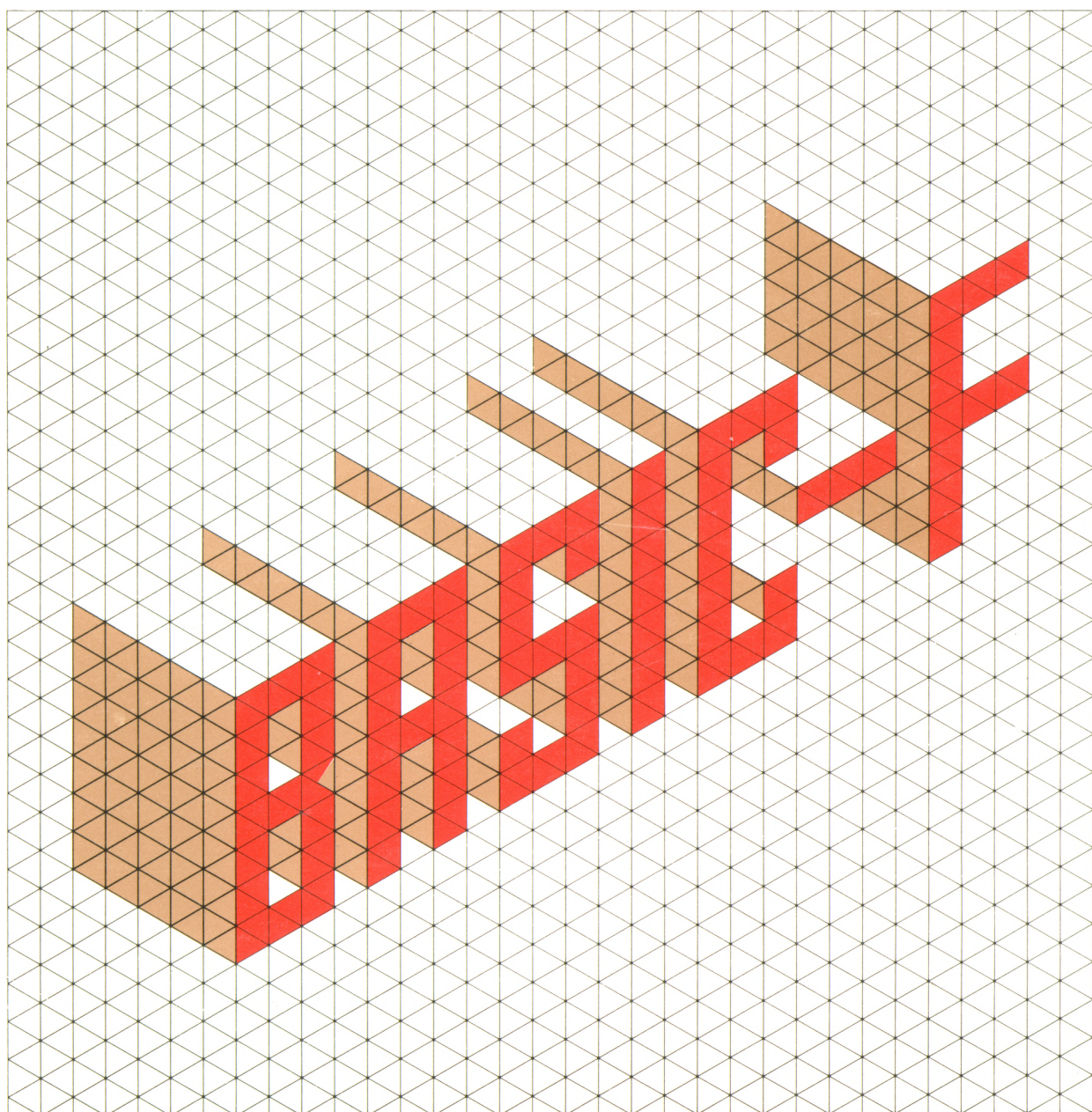


**SORD**

Creative Computer

# m5 BASIC-F Manual

2nd ED.



Easy BASIC for Science

**Copyright © 1984 by SORD COMPUTER CORPORATION**

All rights reserved. Printed in Japan.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SORD COMPUTER CORPORATION Japan.



# Contents

<b>Chapter 1</b>	<b>Introduction</b> .....	2
1-1	Getting Started .....	2
1-2	BASIC-F and other BASICs.....	3
1-3	BASIC-F Variable Types and Numerical Representative .....	3
1-4	BASIC-F Operating Environment .....	3
1-5	BASIC-F Files and Disk System .....	4
1-6	Using Disk Files .....	5
1-7	Using Peripherals.....	6
<b>Chapter 2</b>	<b>General Description</b> .....	8
2-1	Screen Control.....	8
2-2	Screen Buffers and Graphics Modes.....	9
2-3	The M5 Keyboard.....	10
2-4	Editing .....	10
2-5	Graphics Modes .....	11
2-6	Programming Style.....	11
2-7	Statement Syntax.....	13
2-8	Sound.....	13
2-9	Use of Keyboard and Printer .....	14
<b>Chapter 3</b>	<b>Basic Commands</b> .....	16
3-1	Functions .....	102
<b>Chapter 4</b>	<b>Applications Section</b> .....	147
4-1	Loan Payments .....	148
4-2	Initial Investment.....	150
4-3	Regular Deposits.....	152
4-4	Future Value of Regular Deposits (Annuity) .....	154
4-5	Remaining Balance on a Loan .....	156
4-6	Prime Factors .....	158
4-7	Long Number Arithmetic .....	160
<b>Appendix</b>	<b>A Character Codes</b> .....	164
	<b>B M5 Color Codes</b> .....	167
	<b>C M5 Control Codes</b> .....	168
	<b>D M5 CRT Layout Sheets</b> .....	171
	<b>E Keyboard Codes</b> .....	173
	<b>F Port Assignment Table</b> .....	176
	<b>G Memory Maps</b> .....	177
	<b>H Error Codes</b> .....	180
	<b>Commands and Function Index</b> .....	184

## Preface

This manual will be read by individuals with a wide range of skills. It is intended primarily for those who are familiar with computers and have used a BASIC language before. We have nevertheless tried to anticipate the needs of as many people as possible. At times this manual will appear too simple and detailed and at others too complex. Please bear with us when the manual doesn't match your personal expertise exactly. We hope that this manual will be helpful and easy to use. Please feel free to send us your comments and suggestions.

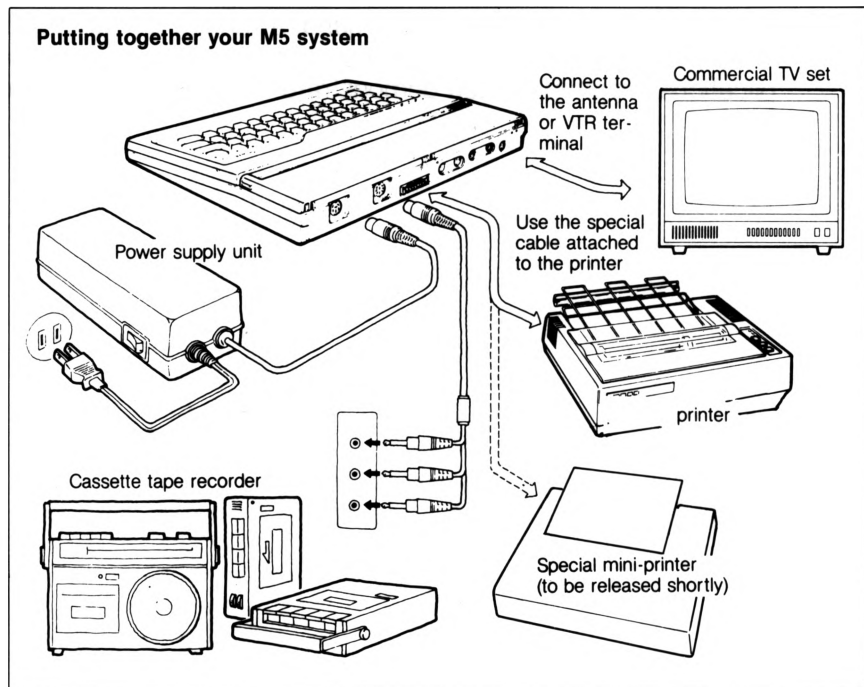
Parts one and two of this manual cover the basic capabilities of the M5 and BASIC-F, such as screen functions, files, and arithmetic. Part three contains individual explanations of each BASIC-F statement, function, and system command, arranged alphabetically each with a short example. The fourth section contains a set of useful applications programs which demonstrate the various features of the M5.

## 1. Introduction

Welcome to SORD's BASIC-F. In this section we will describe the way to set up your M5 to run with BASIC-F. We will also give a brief description of the overall difference between BASIC-G and BASIC-F.

### 1.1. Getting Started

If your M5 is not assembled and running look at the figure below. Connect the various components as shown and power up your computer! Make sure that the BASIC-F cartridge is properly inserted before turning the power on.



Your M5 is now ready for use. The top left corner of your screen should display:

```
BASIC-F  
READY  
L
```

If it does not display this message, turn the power off and check all the connections between components in your system.



## 1.2. BASIC-F and other BASICs

BASIC-F is an enhancement of BASIC-I, which also contains some of the important graphics features of BASIC-G. The most important difference is the inclusion of floating-point arithmetic. With BASIC-F you can perform all kinds of mathematical calculations.

## 1.3. BASIC-F Variable Types and Numerical Representation

BASIC-F has integer, real and string variable types. Constants may also be expressed in hexadecimal. Integers are represented internally by sixteen bits in two's complement form, giving a range of  $-32768$  to  $32767$ . Floating-point numbers range from  $\pm 8.6E-78$  to  $\pm 7.2E-75$ .

Arithmetic follows this operator priority:

(,) comma  
 functions  
 exponent  
 sign  
 \*, /  
 MOD  
 +, -  
 relational operators  
 logical operators

We recommend that programs enforce all operations with parentheses. This removes confusion, improves readability, and prevents errors when programs are transported to and from other systems where operator precedence may be different or unknown.

Thus, example A is recommended over B even though they are arithmetically equivalent:

A)  $X = (2 * C) + 4$   
 B)  $X = 2 * C + 4$

## 1.4. BASIC-F Operating Environment

BASIC-F has two operating environments: a program creation /edit mode and a program execution mode. Programs are executed, in the operating mode, simply by typing RUN.

Programs written in BASIC-G can be transferred to BASIC-F by using the LIST command to create a tape of the program listing. To access the stored program, use the INT command, followed by RETURN, then transfer the program with the OLD command. When there are no reserved words in BASIC-F an error will occur and tape read-out will end.

Once a program is transferred it can be run in both BASIC-G and BASIC-F, and can be saved in any form. Note, however, that language-dependent commands such as floating-point operations and graphics will have to be modified after transfer.

## 1.5. BASIC-F Files and Disk System

BASIC-F can use data files to store and retrieve information. The file system is similar to that found on larger SORD computers with advanced operating systems, but, since the disks used are 3-1/2 inch media, the files are not interchangeable with these machines.

When you use SORD's FD-5 Floppy Disk Drive Unit with the M5 and BASIC-F, you need a number of commands which are not strictly speaking BASIC-F commands. These are not included in the list in Section 2 of this manual, but are described in detail in the CF-5/S Operating System Manual. A brief description is given below.

The disks are used to store both programs and data in units called files. A list of which files are on a disk is itself stored on the disk. When you wish to see what files are stored on a disk you must request that this list be displayed on the screen of your M5. To do this the command used is RUN"LIST. When you enter RUN"LIST the M5 will request that you enter the unit number of the disk that you wish listed. After you give the number, the list of names will be displayed on the screen. You will also be shown how much space on each disk is in use and how much remains. Along with each file name some file attributes will be displayed, such as the file type, and the file size.

Before a new disk can be used, the computer must prepare it. This process is called disk initialization. The command to prepare a new disk for use is RUN"INIT. RUN"INIT clears the disk of any existing material and creates an empty list to receive the names of the files that you will place on the disk.

Since disks may be damaged by accident or by errors in your programs, it is wise to copy disks so that you have extra copies. It is also necessary to copy files for other reasons, and so BASIC-F allows you to copy one disk to another. The command is RUN" COPY.

Each file that is stored on a disk has a name. It may be desirable to change the name of a given file from time to time. A command called RUN"RENAME is supplied to do this. When you enter this command, you must supply the old name of the file and then the new name. The computer will prompt you for each of these names separately. Each name may be up to 9 letters long.

Each file also has several attributes. These attributes change the way that the computer can use the files. These attributes are represented by the letters A, P, R, and W. The command to change attributes is RUN"CHATR. The attributes are fully explained in the CF-5/S Manual.

To use any of these SEVEN commands, just type it as it has been typed above. Then enter RETURN. The M5 will prompt you for each input required to complete the command.

## 1.6. Using Disk Files

With the disk system it is possible to use the disk for storing data as well as programs. The procedure is essentially the same as inputting from the keyboard and outputting to the display, but a few descriptors are needed to specify the file. We will now describe how to use data files from within a program.

Data files can be thought of as large arrays which are stored on the disk instead of in memory as regular arrays. Instead of just numbers or strings, each cell of a file can contain a block of information, and each block can be different in form from others. Before we can use a disk file, we must tell BASIC-F that we intend to do so. This is similar to using the DIM statement for arrays. The command to do this is OPEN, and it tells BASIC-F to open access to a file of a given name. It also associates the file with a channel number. Channel numbers are always used to refer to the file after it is OPENed. This allows you to write programs without knowing what the files are called, and it allows one program to process many different files. The OPEN statement may be used with a variable name for the file name so that it can be changed at run time. The channel numbers are 0 to 15 so up to 16 files may be accessed at once. Channel 0 is usually used by the console. When processing is complete, you must CLOSE the file with the CLOSE command.

To use OPEN with the screen display, keyboard, or printer, the format is:

OPEN "descriptor" as #channel no.

For outputting characters to a graphic display or printer see elsewhere in this manual.



To use OPEN with a cassette unit or floppy disk drive, the format is:

OPEN "descriptor" for  $\left\{ \begin{array}{l} \text{input} \\ \text{output} \\ \text{append} \end{array} \right\}$  as # channel no.

'Input' is used when you want to read data from a file and 'output' is used when you want to write data to a file. 'Append' is used when you want to add data to an already existing file, but is only applicable with the disk system.

To use OPEN with random files on disk, the format is:

OPEN "descriptor" as # channel no. record record length

The format for the CLOSE statement is:

CLOSE # channel no.

Many channels can be closed at the same time by specifying their numbers, separated by commas.

If CLOSE only is executed, all OPEN files will be CLOSED. Be sure to CLOSE all files when you are finished using them. After CLOSEing disk files perform a disk update.

When accessing a cell of an array, we use a special notation: square brackets with an index number following the name of the array. There is a special notation for files as well. To get the information out of a file cell, we use the GET command. To put information into a cell of a file we use the PUT command. Cells in a file are normally referred to as records. Each file in BASIC-F can have up to 65535 records.

## 1.7. Using Peripherals

Besides the floppy disk drive, you can access a variety of peripherals using BASIC-F. The following table gives the standard abbreviations for your peripherals.

Name	Output Device	Input Device
CNS (Console)	Character display	Keyboard
GRP (Graphic)	Graphic display	
PRT (Printer)	Character printer	
PRI (Image)	Image printer	
CMT (Cassette)	Audio cassette tape Floppy disk drive	
FX (File)		

To operate your cassette tape, printer, or display the procedure is much the same as the procedure for using the disk drive as outlined in the previous section. You use the OPEN and CLOSE commands, and if appropriate to the peripheral, you can utilize PRINT, INPUT, SAVE, OLD, CHAIN, and LIST.

In BASIC-F all data inputs and outputs are carried out at the level of files. To operate the peripheral you need to use a special file name called a descriptor. The form of the descriptor is as follows:

Device Name Device Number Drive Number: File Name

A drive number need only be used with a disk drive; device number is significant only with the disk drive. Channels available range from 0 to 15, but as it is usual for the console to use channel 0, in practice 1 to 15 are available. File names must be no more than 9 characters in length. Thus to use OPEN with your printer, for example, the format is as follows:

Open "Descriptor" as # Channel Number

With a disk drive or cassette tape the option to add to your data exists, so the format is a little different:

OPEN <file name> [for {input|output|append}] as # <CH>  
[Record <record length>] {CR|:}

The format for the PRINT command is as follows:

Print [# <channel number>] <expression>,... {CR|:}

With disks or cassette tapes only files which have been OPENed in the manner described above can be PRINTed. The format for the INPUT command is as follows:

INPUT [<prompt> 1 # <channel number>] {,|} <variable> [...]  
{CR|}

You can LIST programs on the printer with the format:

List "PRT" Return

This format uses the characters integral to the printer; to print out just like on the screen use:

List "PRI:" Return

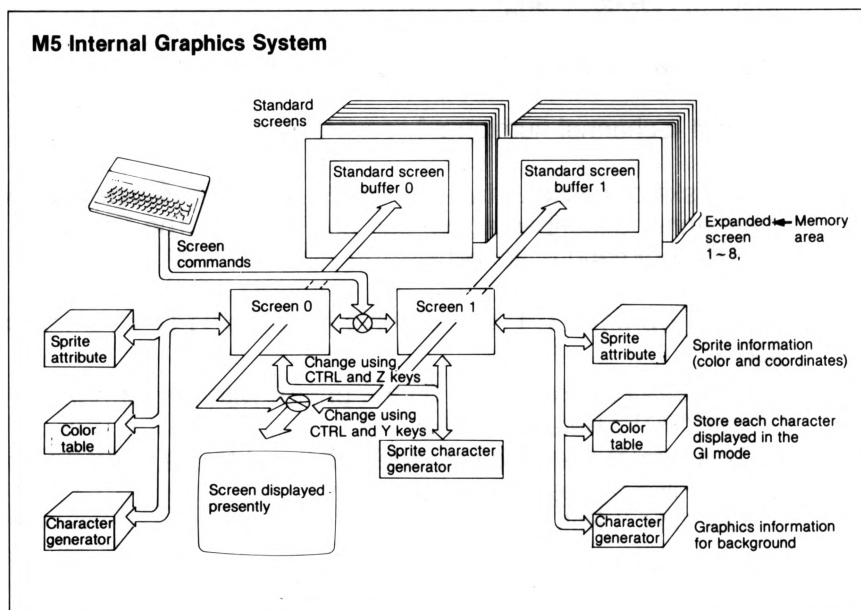
## 2. General Description

In this section we will describe the special features of the M5. The M5 has many special features which are different from other computers. These include the M5's special graphics capabilities and music and sound generation. Used together, these features allow you to create advanced programs on the M5.

### 2.1. Screen Control

The M5 computer uses memory mapping for the display screen. The computer display screen is represented by a complete block of memory. When you type a character, the M5 places the character in the video memory and the circuitry of the M5 places the image on the screen, where you can see it. A powerful feature of the M5 is that it has two such video memory buffers. This allows you to 'type' on two screens or to write to two different places. A simple command will tell the M5 to change which of the two buffers will actually be shown on the screen. When one is displayed the other video buffer is preserved exactly. The computer can switch between the two displays quickly without any screen flicker or noise.

This feature is extremely powerful and allows the development of very useful interaction environments. For instance, a program could, when requested, display help information on a second screen allowing a user to request help, receive it, and continue using the program, all without disturbing the display of the original program!





## 2.2. Screen Buffers and Graphics Modes

### DISPLAY CONTROL

The M5's two screen buffers are referred to as 'screen 0' and 'screen 1'. When the computer is turned on, screen 0 is displayed. At any time screen 0 may be displayed either by PRINTing a 'control U' or by typing it from the keyboard.

To change to the 'other' screen type PRINT CTRL-V. CTRL-V always causes the screen to change. If the screen now displayed is screen 0 then it will change to screen 1. If it is screen 1 then it will change to screen 0.

### WRITE CONTROL

On most computers with only one display screen, any text typed from the keyboard will go directly to the screen. On the M5, text entered from the keyboard can go to either of the screens. This can result in some confusing situations, so be careful!

When the M5 is turned on, screen 0 is displayed, and all text is displayed on that screen, like a normal computer. The M5 can be instructed to put text on the hidden screen by PRINTing or typing a CTRL-Z. CTRL-Z is the WRITE TO HIDDEN SCREEN command. If screen 0 is visible, the text will go to screen 1 and if screen 1 is visible the text will go to screen 0. The M5 maintains a cursor position for each screen. If text is written alternately to one screen and then the other and back again, it will be as if the text had been entered separately on each screen.

The M5 also has the ability to flip screens and write control at the same time. This allows the user to look at the hidden screen and continue to type on the previous screen. For example, if the user is typing on screen 0 but needs some information from screen 1, the user can display screen 1 but continue typing on screen 0. This is done with CTRL-Y. Notice that this command is actually redundant, the same effect being achieved with CTRL-V, CTRL-Z.

#### TABLE OF WRITE CONTROL COMMANDS

CODE	DEC	HEX	effect
CTRL-Q	20	14	multicolor
CTRL-R	18	12	GII
CTRL-S	19	13	GI
CTRL-T	20	14	enter TEXT mode
CTRL-U	21	15	display screen 0
CTRL-V	22	16	display alternate screen
CTRL-Y	25	19	display alternate screen and write to hidden screen
CTRL-Z	26	1A	write to hidden screen

## 2.3. The M5 Keyboard

The M5 has several features which make programming easier. The keyboard has several control functions which save typing time. The following table summarises these functions:

CTRL-C	-shift screen down
CTRL-D	-shift screen left
CTRL-E	-shift screen up
CTRL-F	-shift screen right
CTRL-H	-delete previous character
CTRL-I	-tab
CTRL-J	-move cursor down
CTRL-K	-cursor home
CTRL-O	-exit insert mode
CTRL-P	-enter insert mode
CTRL-X	-clear to end of line
CTRL-up	-move cursor up
CTRL-down	-move cursor down
CTRL-left	-move cursor left
CTRL-right	-move cursor right
CTRL-reset	-interrupt program execution
SHIFT-reset	-stop program execution permanently

These functions can be utilized by typing them into the M5 from the keyboard directly, or by a program using the PRINT statement. When used in a PRINT statement, they must be enclosed in double quotes as part of a string. Furthermore, you must indicate to the M5 that you wish to use the control code but not execute it immediately. This is done by typing 'SHIFT-CTRL-letter,' all at once. This tells the M5 that you wish the PRINT statement to use the correct code but that it should not be used now. When this is done, the controlled character will be displayed in reverse video.

## 2.4. Editing

These commands are often useful when editing. Naturally, when typing in programs, errors will occur. The M5 has several screen editing functions which facilitate error correction. When an error is discovered it is not necessary to retype the entire line. When the line containing the error is displayed on the screen, the arrow keys can be used to move the cursor directly to the position of the error. The error can then be corrected right on the screen. When the corrections have been made, entering a 'CR' will tell BASIC-F to enter the new, corrected line into the program.

The other control keys are also useful when editing. For instance, CTRL-X will delete the characters all the way to the end of the line. CTRL-N will move the cursor to the beginning of the next line. CTRL-H will delete the character under the cursor, anywhere in the line. CTRL-P will allow you to insert new characters into the line and CTRL-O will exit insert mode. At first these functions may seem confusing and time consuming, but with practice they speed up programming considerably. The screen also has cursor wrap around. This means that if you move past the edge of the screen the cursor will reappear on the opposite edge. This can save time if you have to move the cursor a long distance.

## 2.5. Graphics Modes

Under BASIC-F some of the graphics capabilities of BASIC-G have been retained. Graphics are facilitated through the inclusion of several commands such as DRAW, PLOT, PAINT, and COLOR. These are described in the command section of this manual. Here we will only describe the characteristics of graphics in the M5.

The M5 implements graphics through the use of different screen modes. Recall that the M5 has two screens. Each of these screens can be set to different modes. Under BASIC-F there are four such modes. Each screen mode has different properties which facilitate different tasks.

In TEXT MODE (G1) the screen can display 24 lines of 40 characters at once. Each character is composed of 48 dots in a 6x8 font. TEXT MODE is set for either screen by PRINTing or typing a CTRL-T.

An alternate screen mode is G2. Graphics cannot be displayed, but more characters will fit on the screen. The M5 is in G1 mode when first powered up.

mode	font	
G1	8x8	32x24
GII	8x8	32x24
Text	6x8	40x24
Multi	8x8	32x24

## 2.6. Programming Style

BASIC-F includes features which can make your programs easier to understand and debug. Programs in BASIC-F can be more clearly written, easier to debug and more efficient. They can also be stylistically superior. Features include:



- labels for branching
- indenting of code
- a REPEAT command

BASIC-F has the ability to indent lines of program text. In the following example the inside of the loop is indented.

Labels can be attached to line numbers. This allows routines also to be given mnemonic names. A mnemonic name relates a name to the function of the routine itself. This allows a reader to see quickly what a section of a program is supposed to do. Labels can be used with GOTO, GOSUB, and RESTORE in BASIC-F.

BASIC-F has the ability to indent lines of program text. In the following example the inside of the loop is indented.

Labels can be attached to line numbers. This allows routines also to be given mnemonic names. A mnemonic name relates a name to the function of the routine itself. This allows a reader to see quickly what a section of a program is supposed to do. Labels can be used with GOTO, GOSUB, and RESTORE in BASIC-F.

Within REPEAT and FOR loops, code should be indented two or three spaces to make programs more legible. This shows which instructions are within the scope of a given loop. For example:

```
100 for I=1 to 100
110   gosub $GETVAL
120   gosub $DOPROC
130 next I
```

This clearly shows which commands belong inside the loop and where the loop ends. This is a great aid to debugging and understanding programs. For this reason, BASIC-F does not remove the extra blanks as do some other BASICs.

The REPEAT command should be used whenever a counter is not required for a loop. This reduces the number of variables used, and thereby improves readability and saves both memory and memory accesses. REPEAT loops are always executed at least once, since the test for looping occurs at the end of the loop. Code should also be indented in the REPEAT loop. For example:

```
100 repeat
110   gosub $GETINPUT
120   gosub $PROCESS
130 until DONE$="TRUE"
```

Older BASICs allowed variables to have one letter and one number as names. Modern BASICs such as BASIC-F no longer have this restriction. It is good programming practice to make your variable names meaningful. Using mnemonics helps debugging and understanding programs. In BASIC-F variables may have names consisting of up to 32 alphanumeric characters. Of course, names that are too long also increase the likelihood of typing and reading errors, remember to keep a balance.

## **2.7. Statement Syntax**

Each line of a BASIC-F program begins with a line number. These line numbers range from 1 to 32,767. Each line may contain several BASIC statements separated by full colons, ':'. Each line may be up to 252 characters in length.

After a line number any BASIC statement may occur. Also, after a line number, but not a colon, a label should occur. The label must begin with a dollar sign '\$', and no blanks should occur between the line number and the label name. The label can then be used by other BASIC commands to refer to the line number. When referring to the label the dollar sign must be included.

A special statement, called a comment, may also appear on a line. When a comment appears, no other BASIC statements should occur after it on the line. A comment and anything after it is always ignored by BASIC-F. A comment is indicated by the keyword REM, an exclamation mark '!', or an apostrophe '''

The syntax of all other statements and commands is explicitly defined in this section of the manual.

## **2.8. Sound**

The M5 has a SN sound generation chip. This gives the M5 the ability to produce sound effects. These functions can be performed from BASIC-F for use in any program.

To produce sounds, the M5 contains three tone generators and one noise generator. These can be used singly or in combination to produce a wide variety of sounds. BASIC-F has one command to control all four of these sound generators. The command is SG and it has three parameters: the channel, the frequency and the volume of the sound to be produced. The second parameter describes the type of noise to be produced when accessing the noise generator. The command is as follows:

SG channel, frequency, volume

The sound channels are 0,1, and 2. The noise channel is number 3. The frequency value is specified as a value from 1 to 1023. For any given value the frequency is calculated as follows:

$$\text{frequency} = 111.86 \text{ (khz)}/\text{desired frequency}$$

For example, a middle C on a piano is 261.6 (Hz). Thus:

$$\begin{aligned}\text{code value} &= 111.86 \text{ (khz)}/256 \\ &= 428\end{aligned}$$

and:

SG 0, 428, 15

will produce a pure middle C from your TV.

When the channel specified is 3, the noise generator is accessed. The noise generator produces eight types of noise from 0 to 7. When used in combination with channel 2, interesting sounds can be produced.

See the command section for more information about SG.

## 2.9. Use of Keyboard and Printer

The printer port on the M5 can be used to connect several printers to your computer. With dot matrix printers, the M5 can print out graphics images from the screen as well as text and program listings.

Be sure to read the instructions with your printer before connecting it to your M5. If you have problems contact your dealer.

The M5 connects to printers via a parallel port. This requires a special cable for the connector on both the printer and the M5.

The printer is used for two main purposes: to print out program listings and to produce 'hard copy' output of programs written on the M5. Program listings are produced by the LIST command and its variations. For further information on these commands see the command section of this manual.

Two kinds of hard copy output can be produced by the M5. One is standard text, such as found in letters and reports. The other form of output is graphic images. Text is output directly to the printer with variations of the PRINT command. See the command section for further information.

Graphics images are sent to the printer with the GCOPY command. Again, see the command section for further details. Graphics images can only be printed on dot matrix printers.

### 3. Basic Commands

In this section we provide a list of all BASIC-F statements and system commands arranged alphabetically. Each statement has a small example shown in isolation; for more examples see the section of applications programs.

The FORMAT part of the statement descriptions is read as follows:

- {CRLF;} means that the statement can be ended with either a carriage return or a full colon and another statement.
- [ ] indicates that the component in braces is optional.
- [...] indicates that the previous component is optionally repeatable.
- | either of two options can be used.
- < > the word(s) inside the brackets refer(s) to a single object.

# \$

**FORMAT :**     \$<label name> {CR |;}

**FUNCTION :**   Remark statement or label-name.

**Comments :**   Up to 32 characters in length; the \$ must be situated next to the line number.

**Example :**     100 goto \$SUB  
                  :  
                  :  
                  1000\$SUB  
                  :  
                  :



# AUTO

**FORMAT :** AUTO [<first line number>][,<increment>]

**FUNCTION :** AUTOMatic line numbering when writing a program.

**Comments :** When entering a program, AUTO will put a line number on the screen after you complete each line. AUTO sometimes conflicts with the natural progression of programming by enforcing an artificial regime on the order in which code is entered. The use of this command is a matter of personal style.

**Example :**

```
AUTO 100,20
100 a=1
120 b=2
140 c=a+b
160 Print c
```

# BCOL

**FORMAT :** BCOL [<color code>] {CR |;}

**FUNCTION :** Sets the background color of the screen.

**Comments :** The default color code is 0, no color. If the color code is the same as that for characters (FCOL) then characters will be invisible. See Appendix B for the color codes.

**Example :**

```
10' BCOL TEST
20'
30  for I=0 to 10
40    bcol I
45      cls:Print cursor(15,13);I
50      sleep 1
60  next I
70 bcol 0
```

# CALL

**FORMAT :** CALL <address> [, <AF registers>] [, <BC registers>]  
[, <DE registers>] [, <HL registers>]

**FUNCTION :** CALL transfers program control to a specific machine address, after leaving the return address on the stack.

**Comments :** This command is used to call machine language routines. These can be part of the M5 monitor, or written by the user. When the machine routine executes a RET instruction, control is returned to the BASIC-F program. After the return, the following registers are stored in the following memory locations:

PSW : &7262  
A : &7263  
B : &7265  
C : &7264  
D : &7267  
E : &7266  
H : &7269  
L : &7268

Caution should be exercised with the use of CALL. Always save your program to disk or tape before testing a program with a CALL. Often the contents of the machine memory will be altered and your program will no longer even LIST properly. On the other hand, machine routines will run very fast, often 300 to 1000 times faster than a BASIC routine which performs the same function. Thus if speed is required, the CALL function may be necessary. (Also it can set the values to the registers; see the REG function.) You should however refer to a machine language manual for the proper use of this statement.

**Example :**

```
10 gosub $SETRoutine
20 call ADDRESS,,BCREG
30 end
40 $SETRoutine
50 ADDRESS=&0C97
60 BCREG=&0500
70 return
```

# CHAIN

**FORMAT :** CHAIN [<file-name>] [, all] {CR |:}

**FUNCTION :** Retrieves and executes a program stored on tape or disk.

**Comments :** A statement on the same line as a CHAIN command, separated by a ':' will never be executed. The variables and values of the current program are lost. And by setting [all], the variables and values will remain as is.

**Example :**

```
100 Print "Done"  
110 chain "PROG2"
```

# CLEAR

**FORMAT :** CLEAR [<work field>][,<last user field>]{CR |:}

**FUNCTION :** Clears a portion of memory for use as a PAINT and character buffer. Memory is freed in 256 byte blocks. The second parameter sets the highest memory location used by the program.

**Comments :** The top of the memory field is below the buffer, and prevents the program from running into it as it uses memory.

**Example :** 100 clear 512,&8FFF

# CLIST

**FORMAT :** CLIST[ <file name>][, <line number 1>]  
[, <line number 2>]{CR |;}

**FUNCTION :** Lists a program in upper case only.

**Comments :** This function is identical to LIST except that all lower case letters are converted to capitals before printing.

**Example :**

```
10 repeat
20   A$=inkey$
25   Print ascii(A$)
30 until A$=chr$(13)

clist
10 REPEAT
20   A$=INKEY$
25 PRINT ASCII (A$)
30 UNTIL A$=CHR$(13)
```



# CLOSE

**FORMAT :** CLOSE [# <channel no.> [...]] {CR |:}

**FUNCTION :** Ends usage of user file.

**Comments :** If the channel number is omitted, all channels will stop.

**Example :**

```
100 open "PRT:" as#2
110 print#2 "good"
120 close#2
130 end
```

# CLS

**FORMAT :** CLS [<initialize code>] {CR |:}

**FUNCTION :** Clears the screen.

**Comments :** The initialize code may specify the character code which will fill the screen. This is normally a null.

**Example :**

```
cls
110 print "This is the top of the
screen"
```

# COLOR

**FORMAT :** COLOR <character-code>, <color-code>

**FUNCTION :** Sets up the character color in the GI and GII modes.

**Comments :** The higher-order four bits indicate the character color and the lower-four bits designate the background color, or  
color-code = character-color  $\times$  16 + background-color  
or  
color-code = & HL (H = character-color, L = background-color: H, L are hexadecimal)

- The GI mode—each time a character is colored, it actually affects seven other characters with contiguous ASCII codes (If the 16  $\times$  16 ASCII table is split into upper and lower halves, each half contains columns of eight characters. These eight are colored identically).
- The GII mode—each character is colored individually.

**Example :**     color 65,880

# CONSOLE

**FORMAT :** CONSOLE [<A>][, <B>][, <C>][, <D>][, <E>][, <F>][, <G>]{CR |;}

**FUNCTION :** Enables/Disables keyboard function keys.

**Comments :**

Function	0	1
A Keyboard sounds	OFF	ON
B Generate key-board keywords	OFF	ON
C Display page	page 0	page 1
D Process page	page 0	page 1
E Display mode	O : M, 1 : G2, 2 : G1, 3 : T	
F Screen lock		
G ACMT Baud rate	33 = 2,000 Baud	

**Example :** console 0,,0,0,3

# CONT

**FORMAT :** CONT {CR |:}

**FUNCTION :** Restarts a program after it has been interrupted by a STOP command or a keyboard interrupt.

**Comments :** CONTinue will not work after an END command has been executed. By entering STOP commands while testing a new program, the programmer may check individual sections of a program for errors.

**Example :**

```
100 print "Here we are...."
110 stop
120 print "Here we go...."
130 end
run
Here we are....
Stop at 110
Ready
cont
Here we go....
Ready
```

# CURSOR

**FORMAT :**     CURSOR (X, Y)

**FUNCTION :**    Moves the cursor to the coordinates specified by (X, Y).

**Comments :**    A semicolon ";" is usually used after this keyword.

**Example :**     100 Print cursor(15,10); "GAME OVER"



# DATA

**FORMAT :** DATA <constant> [...] {CR |:}

**FUNCTION :** Stores constant information to be used by the program and accessed via READ.

**Comments :** Data may be numerical of any type, or character data. Collect DATA statements near the end of the program. When using disk systems keep DATA use to a minimum as they occupy valuable memory. Instead keep the information on a file.

**Example :**

```
10 cls
20 for I=0 to 7
30 read A$
40 print A$,
50 next I
100 data 12,24,1955,Smith
110 data 3,7,1942,Jones
```

# DEL

**FORMAT :** DEL [<line number 1>][,<line number 2>]{CR |:}

**FUNCTION :** DELetes a line or lines from the current program in memory.

**Comments :** Be careful when entering two line numbers not to delete entire blocks of code from a program.

**Example :**

```
100 rem ABCDE
110 rem fghij
120 rem klmno
130 rem pqrst
140 rem UVWXY z
```

```
Ready
del 110,130
```

```
Ready
list
100rem ABCDE
140rem UVWXY z
Ready
```

# DIM

**FORMAT :** DIM <array name> (<array size>[,...]) [,...] {CR |;}

**FUNCTION :** DIM allocates memory for an array.

**Comments :** A DIM must be executed before an array is used. Each DIM statement must only be executed once. It is not legal to change the DIMensions of an array after it has been declared. An array can use up a large amount of memory, so care should be taken to allocate only as much as is needed. (Max. 255 dimension is possible, it does not include 0 dimension).

**Example :**

```
100 dim A(5),A$(5)
110 for I=1 to 5
120     A(I)=I+64
130     A$(I)=chr$(A(I))
140 next I
150 for J=1 to 5
160     Print A(J),A$(J)
170 next
180 end
```

run

65	A
66	B
67	C
68	D
69	E

Ready

# DRAW

**FORMAT :** DRAW <GR-coordinates>[,<GR-coordinates>][,..]{CR |:}

**FUNCTION :** DRAWS a line on the screen.

**Comments :** If only one pair of coordinates are given then the line will be drawn from the current graphics-cursor position, and the graphics-cursor becomes the end of the line. If two pairs of coordinates are given they specify a line and the graphics-cursor does not change position.

**Example :**

```
100 Print "UR":ginit
110 draw 100,50,100,100
120 gmove 50,100
130 draw 100,150
140 draw 200,180;100,0,0,190
150 Print "L"
```

# END

**FORMAT :**     END {CR |;}

**FUNCTION :**   Indicates the end of a program and halts execution.

**Comments :**   Once the END instruction is executed, execution cannot be resumed with CONT. This instruction is not strictly necessary. (All of the channel will be closed)

**Example :**

```
1000 Print "This is a very short
Program"
1010 end

run
This is a very short  Program

Ready
```

# EVENT

**FORMAT :** EVENT <interrupt-interval> [, <delay-time>] {CR |;}

**FUNCTION :** Sets the interrupt interval accessed by the ON EVENT GOSUB statement.

**Comments :** The interrupt-interval is the interval between consecutive event timer interrupts that can take on values between 0 and 255 (number of 1/60 second units). Be careful since 0 is assumed to be 256 time units. The delay-time is the delay time until the first event timer interrupt and can take on values from 0 to 32767 (1/60 second units). 0 is assumed to be 32768. Unless otherwise specified, the first event timer interrupt occurs immediately after setting the event timer interrupt-interval. If negative values are specified, the event timer will stop.

**Example :**

```
10 cls
20 event 60,60
30 on event gosub $EU
40 event on
50 I=0:C=0
60 I=I+1
70 Print cursor(15,8);"I=";I;
80 goto 60
90 $EU
100 C=C+1
110 Print cursor(15,13);"C=";C;
120 return
```



# EVENT {ON/OFF}

**FORMAT :**     EVENT {ON/OFF} {CR |:}

**FUNCTION :**    Enables/Disables event timer interrupt.

**Comments :**    <ON/OFF>

ON = Allows calling of subroutine by an event timer interrupt set up by an ON EVENT GOSUB.. statement

OFF = Disables event timer interrupts

**Example :**

```
100 console,,0,0,2
110 cls
120 event 60,60
130 on event gosub $SUB
140 event on
150 H=0:I=0
160 $ADD
170 repeat
180     H=H+1
190     Print cursor(10,10);"H   =";H;
200     A$=inkey$
210 until A$<>" "
220 end
230 $SUB
240 I=I+1
250 Print cursor(10,13);"sec=";I;
260 return
```

# FCOL

**FORMAT :** FCOL [<color code>] {CR |:}

**FUNCTION :** In Text Mode, sets the color of the characters.  
In G2 mode or Multi-color mode, sets the color of the graphics display.

**Comments :** The default value for the color code is 14, grey. If the background color (BCOL) matches FCOL then characters will be invisible. See Appendix B for color codes.

**Example :**

```
10' FCOL TEST
20   for I=0 to 14
30   fcol I
40   sleep 1
50   next I
```

# FOR..TO..[STEP]

**FORMAT :** FOR <control variable> = <initial value> TO <final value> [STEP <step value>] {CR |;}

**FUNCTION :** FOR NEXT loops are used to perform many iterations of a section of the program. The control variable is set to the initial value and control transfers to the next statement. When a NEXT statement is reached with the same control variable then the control variable is incremented by the step value. When the control variable reaches or exceeds the final value then control is transferred to the statement immediately after the NEXT statement. Otherwise the statements between the FOR and the NEXT are executed once more.

**Comments :** The number of times that the program will iterate the commands in the loop is calculated as:

$$\text{iterations} = (\text{final value} - \text{initial value}) / \text{step value}$$

If the step value is not specified then it is given the default value of 1. The control variable should not be altered by the instructions in the loop. This is very dangerous and is poor programming style. When several loops occur one within the other they can take a long time to execute. Care should be taken to realize this when writing programs. Programs may contain complete loops within other loops but they may not overlap. Thus for each FOR statement encountered a NEXT must occur in the reverse sequence.

**Example :**

```
100 cls
110 for I=3 to 27 step 3
120     for J=1 to 9
130         locate I,J
140         Print right$(" "+num$(I/3*J),
150     next J
160 next I
170 Print
180 end
```

Note the use of indentation for readability.

# GCOPY

**FORMAT :** GCOPY [<format type>] {CR |;}

**FUNCTION :** Prints the current screen image on the printer.

**Comments :** The default format type is 0. Formats are defined as follows:

0 = 40 character image format

1 = 80 character image format

Before this command is executed, it is necessary to issue the GMODE command.

**Example :**

```
100' gcopy TEST
110 Print ""
120 ginit
130 gmode 4
140 ERASE
150 fcol 000 OF
160 GY1=30:GY2=5:H=0
170 Plot 228,0
180 for TH=0 to 6*180 step 8
190 H=H+1
200 X=cos(80-H/2,TH)
210 GY=GY1+1
220 GX=128+X
230 GY2=GY2+1
240 draw GX,GY1
250 gmove 128,GY1
260 draw GX,GY1
270 next TH
280 Print#2
290 '
300 gmode 0
320 for G=0 to 1
330 Print#1,cursor(1,21);"GCPY";G;
340 gcopy G
350 Print#2,""
360 next G
370 Print "";
380 end
```

# GET

**FORMAT :** GET [# <CH>][[, ]<variable> [...]]{CR |;}

**FUNCTION :** Reads data from a designated channel to a designated variable.

**Comments :** It is desirable that data obtained with GET be data created with PUT. With character variables it is necessary to use LEN before GET to control the size of the variable.

**Example :**     get #3 CODE%,NAME\$,TEL\$

# GINIT

**FORMAT :** GINIT {CR |:}

**FUNCTION :** Enter the graphics mode (applicable to the multicolor and Gll modes). The specified screen-clear-specifier is used for graphics (default is 255).

**Comments :** <screen-clear-request>  
0 = display character, font clear colors, graphics cursor and initialize graphics mode  
1 = display characters  
2 = font clear

In the multi-color mode, values greater than 1 clear the font and display characters.

If the screen is not cleared, go ahead and arrange characters on the displayed screen.

**Example :**

```
100 print "URL":ginit
110 console:,0,0,1
120 draw 10,10,200,180
130 ginit
140 fcol 0
150 draw 0,191,255,0
160 end
```

# GMODE

**FORMAT :** GMODE [<mode 1>][, <mode 2>]{CR |:}

**FUNCTION :** Sets up the graphics display mode.

**Comments :** Affects the PLOT, DRAW, and PAINT statements. It takes the color or pixel already on the screen and the newly specified color or pixel, and then applies one of the functions below to decide the resulting color or pixel, e.g. a boolean operation on the specified color code of a pixel ORed with the existing color code of that pixel yields the new pixel condition in GMODE 1.

<mode 1>

Color

- 0 = replace
- 1 = OR
- 2 = AND
- 3 = old display

Pixel

- 4 = replace
- 5 = AND
- 6 = XOR
- 7 = old display

The following relationship holds for the "GRP" device.

<mode 1>

Color

- 0 = replace
- 1 = OR
- 2 = AND
- 3 = old display

Image

- 4 = OR
- 5 = AND
- 6 = XOR
- 7 = Old display

When you specify "1" in mode 2, only the image will be processed.

**Example :**

```
100 console,,0,0,1
110 ginit
120 fcol 8
130 draw 100,0;100,100;0,100;0,0
140 Paint 50,50
150 fcol 3
160 draw 55,0;55,50;0,50;0,0
170 gmode 1
180 fcol 5
190 Paint 25,25,3
200 Print chr$(26)
210 end
```



# GMOVE

**FORMAT :** GMOVE <GR-coordinates> {CR |;}

**FUNCTION :** Moves the graphics cursor to the desired coordinates.

**Comments :** No line is drawn with this command; it is only used to position the graphics cursor.

**Example :**

```
100 console,,0,0,1
110 ginit
120 fcol B
130 draw 100,0;100,100;0,100;0,0
140 fcol 15
150 gmove 200,100
160 draw 100,0;100,100;0,100;0,0
170 Print chr$(26)
180 end
```

# GOSUB

**FORMAT :** GOSUB <destination> {CR |;}

**FUNCTION :** Transfers control to a subroutine so that control will be returned BACK to the point IMMEDIATELY AFTER the GOSUB.

**Comments :** Use the RETURN statement to transfer control back to the point where the subroutine was called. Notice that this allows the subroutine to be used from many places in the program. GOSUB should be used in place of GOTO whenever possible. The BASIC GOSUB has recursive capability: that is, a subroutine may GOSUB to itself, but as BASIC has no parameter passing abilities, all variables must be managed by the programmer. Destination = {line number|label name|numeric variable (line number)|string variable (label name)}

**Example :**

```
10 Print "USL"
20 gosub $STCHR
30 gosub $SP.SET
40 end
100 $STCHR
110 stchr "00183c66db7e2400" to &7F,0
120 return
200 $SP.SET
210 scod 0,&7F:scol 0,5
220 loc 0 to 128,96
230 return
```

# GOTO

**FORMAT :** GOTO <destination> {CR |:}

**FUNCTION :** GOTO transfers program control from the current line to the line number or label in the statement.

**Comments :** GOTO statements should be used as little as possible. Although this is difficult in BASIC their use can be limited. To make programs clear, write programs in blocks, with little use of GOTO within blocks. A statement appearing after a GOTO on the same line (with a : separator) will never be executed. Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 A=100
110 goto 1000
120 B=A*5
130 goto $DISP
.
.
.
1000 B=A/5
1010$DISP
1020 Print "a = ";A;" b = ";B
1030 end
```

# IF..THEN..ELSE

**FORMAT :** IF <conditional expression> THEN <statement> [...]  
[ELSE <statement> [...]] {CR |;}

**FUNCTION :** Evaluates the conditional expression. If it is true (the condition is satisfied) then the statements(s) after the THEN are executed. If it is false (the condition is not satisfied) then the statements after the ELSE are executed.

**Comments :** If there is no ELSE part and the condition is not satisfied then the statement has no effect. Conditional expressions are : =, <, >, >=, <=.

**Example :**

```
100 input "10 + 15=?":A
110 if A=25 then Print "good"
    else Print "again":goto 100
120 end
```

# INPUT

**FORMAT :** INPUT [<prompt> | # <channel number>] {, |;}  
<variable> [...]{CR|;}

**FUNCTION :** INPUT accepts alphanumeric data from the keyboard and assigns their values to variables.

**Comments :** The INPUT statement will place a question mark on the screen if the program does not include the <prompt> field. If multiple inputs are to be entered at one time then the variable names should be separated with commas. When multiple entries are to be made by the user running the program, then the inputs must be separated by commas. This format results in a problem inherent in BASIC: when the user is requested to input a string of alphabetic characters into a string variable, the string should not contain any commas. The only way to overcome this shortcoming is to write input routines using the INKEY function. For programs to be used by inexperienced users this may be preferable to INPUT, as 'bomb-proof' input routines can be created. A 'bomb-proof' input routine cannot cause a BASIC error, no matter what the user types in. With the INPUT statement, if the user enters the wrong number of parameters, or enters a comma in a string, BASIC will print an error message. These may not be understood by inexperienced users.

**Example :**

```
100 input "What is your age?";AGE
110 input "What is your name?";NAME$
120 print AGE;"is a good age ";NAME$
130 end
```

```
run
What is your age ? 4
What is your name ? Ronald
4 is good age Ronald
```

Ready

# KILL

**FORMAT :** KILL <file name> {CR |;}

**FUNCTION :** Deletes specified files.

**Comments :** Effective only with an external disk drive.

**Example :** kill "FXAO:OLDDATA"

# LEN

**FORMAT :**      LEN <character string length> {CR |:}

**FUNCTION :**    Resets the maximum length of string variables.

**Comments :**    The default value for string lengths is 18 characters. LEN can be used to set the value from 1 to 255.

**Example :**

```
100 A$="abcdefghijklmnopqrstu"
110 Print A$

run
Err 15 in 100

Ready

100 len 24:A$="abcdefghijklmnopqrstu"
110 Print A$

run
abcdefghijklmnopqrstu

Ready
```

# LET

**FORMAT :** [LET] <variable>[,<variable>] [,.....]= <expression>  
{CR |;}

**FUNCTION :** LET assigns the result of an expression to a variable. This result may then be referenced in other expressions.

**Comments :** The word LET is optional. A variable name may be followed by the assignment operator '=' and an expression. LET itself is a vestigial command.

**Example :**

```
100 let A=5
110 B=6
120 Print A,B
```



# LIST

**FORMAT :** LIST [<descriptor> | <file name>] [<line number-1>]  
[,<line number2>] {CR |:}

**FUNCTION :** Lists a file or portion of a file to another file, the printer, or on the screen.

**Comments :** The LIST command can be used instead of the SAVE command to store files. When a file that was LISTed to disk or tape is subsequently read, the program in memory will not be erased, unlike a file that was SAVED. However if the program being read has any line numbers identical to the memory program these lines will be replaced with the new lines. To distinguish these files from normal files, they are called "listing files."

**Example :**

```
list
100 A=1:B=2
110 Print A
120 Print B
:
:
```

Ready

# LOC

**FORMAT :** LOC <sprite-number> TO <GR-coordinates> {CR |:}

**FUNCTION :** Moves sprite-number to the specified GR-coordinates.

**Comments :** Sprite-numbers are 0 to 31, 0 has highest priority. The highest priority number at that location will be displayed.

**Example :**

```
10' LOC TEST
20  Print "USL"
30  stchr "00183c66db7e2400" to &7F,0
40  scod 0,&7F:scol 0,4
50  for I=0 to 255
60    loc 0 to I,90
70  next I
```

# LOCATE

**FORMAT :** LOCATE <column>, <line> {CR |:}

**FUNCTION :** Moves the cursor to the specified line and column on the screen.

**Comments :** This statement is identical to CURSOR(X,Y) except that because CURSOR is a function, it can be used in a PRINT statement and LOCATE cannot. The screen uses a 0 origin coordinate system. The legal range of each coordinate depends on the screen mode in use.

**Example :**

```
100 locate 9,11
110 Print,"Bottom corner"
```

# MAG

**FORMAT :** MAG [<sprite-modifier>] {CR |:}

**FUNCTION :** Change the sprite size and format.


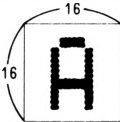
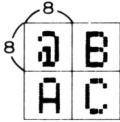
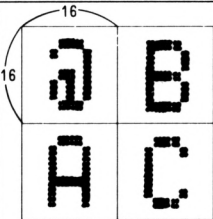
**Comments :** <sprite-modifier>

- 0 = 8×8 dot matrix
- 1 = 8×8 dot matrix (by 2)
- 2 = 16×16 dot matrix
- 3 = 16×16 dot matrix (by 2)

**Example :**

```

110 Print "USA"
120 stchr "00183c66db7e2400" to &7F,0
130 scod 0,&7F
140 scol 0,4
150 mag 1
160 loc 0 to 100,100
170 end
  
```

	Example of display for 'A', or & 41	Character dot matrix	Enlarged dot matrix	Notes
MAG 0		8×8 dots	8×8 dots	<ul style="list-style-type: none"> <li>• Default mode when power is first supplied</li> <li>• Sharp picture</li> </ul>
MAG 1		16×16 dots 16×16 dots	16×16 dots 16×16 dots	<ul style="list-style-type: none"> <li>• One large picture can be created by combining four smaller pictures</li> <li>• Hazy picture</li> </ul>
MAG 2		8×8 dots	16×16 dots	<ul style="list-style-type: none"> <li>• Large sprite can be easily created by combining four characters</li> <li>• Sharp picture</li> </ul>
MAG 3		16×16 dots	32×32 dots	<ul style="list-style-type: none"> <li>• One large picture can be created by combining four smaller pictures</li> <li>• Hazy picture</li> </ul>

# NEW

**FORMAT :** NEW {CR |:}

**FUNCTION :** Clears the current program and memory contents.

**Comments :** This command prepares the M5 for beginning a new program. MAKE SURE that the current program has been saved on disk or tape before issuing a NEW command.

**Example :**

```
10rem Program 1
20 dim A(100)
30 gosub $INIT
:
```

```
Ready
new
```

```
Ready
list
```

```
Ready
```

# NEXT

**FORMAT :**      NEXT [<control variable>] [...] {CR |:}

**FUNCTION :**    NEXT ends a section of a program started by a FOR statement which is to be executed repeatedly. The control variable indicates which FOR statement this NEXT matches.

**Comments :**    NEXT should always be used with explicit control variables as leaving them out makes the program very confusing. All FOR NEXT loops should be indented two or three spaces to show the extent of each loop graphically. Further, each loop should be a maximum of one screen or one page in length. Longer loops should call subroutines.

**Example :**

```
100 for I=0 to 21 step 3
110     print I
120 next I
```

# OLD

**FORMAT :** OLD [<file-name>] {CR |:}

**FUNCTION :** Reads a file from external storage.

**Comments :** When the file-name is omitted, the first file found is read into memory.

**Example :** old "CMT:PROG1" &g

# ON ERROR GOSUB..

**FORMAT :** ON ERROR GOSUB <destination> {CR |:}

**FUNCTION :** Transfers control to the line number when any BASIC-F error is detected during program execution.

**Comments :** This command allows the program to continue to run when an error occurs. The routine should display an error message which will explain the error that the user made. The program should then allow the user to correct the mistake. This will only work with run time errors, not program syntax errors. Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
10 on error gosub $ERR
20 input "INPUT NUMBER":A
30 print A
40 $ERR
50 if err=25 then resume 20
```



# ON EVENT GOSUB

**FORMAT :** ON EVENT GOSUB <destination> {CR |;}

**FUNCTION :** Calls subroutine beginning at line number when event timer interrupts—interrupt priority 2. (The event timer is initialized with the EVENT statement.)

**Comments :** Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 event 60,60
110 on event gosub $EU
120 event on
130 goto 130
140'
150 $EU
160 print chr$(7);
170 return
```

# ON.. GOSUB..

**FORMAT :** ON <expression> GOSUB <destination> [...]{CR |:}

**FUNCTION :** This is a multiway branch instruction. After evaluating 'expression', branches to the nth line number in the line number list. On executing a RETURN, returns to the next statement after this 'ON GOSUB.'

**Comments :** This type of instruction is also known as a CASE or SELECT statement. It allows one of many options to take place from one point in the program. Be sure that the range of results for the expression is limited to the number of line numbers in the line number list. It is suggested that labels be used for all line numbers. Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 input "Enter the command number  
    (1-5) "; C  
110 on C gosub $COM1,$COM2,$COM3  
    :  
    :  
1000 $COM1  
    :  
    :  
2000 $COM2  
    :  
    :  
3000 $COM3  
    :  
    :
```

# ON.. GOTO..

**FORMAT :** ON <expression> GOTO <destination> [...] {CR |:}

**FUNCTION :** This is a multiway branch instruction. After evaluating the expression, branch to the nth line number in the line number list.

**Comments :** For using subroutines the ON GOSUB statement will be more convenient. Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 input "No.(1~3) ? ";CASE.
110 if CASE<1 or CASE>3 then goto 100
120 on CASE goto $WHEN1,$WHEN2,$WHEN3
:
:

1000$WHEN1
:
:

2000$WHEN2
:
:

3000$WHEN3
:
:
```

# ON.. RESTORE..

**FORMAT :** ON <expression> RESTORE <destination> [...] {CR |:}

**FUNCTION :** Sets the data pointer to one of several data groups depending on expression. After evaluating the expression sets the data pointer to the data statements after the nth line number in the line number list.

**Comments :** Use labels instead of line numbers for each of the data groups. See the DATA and RESTORE statements. The expression must result in a number less than or equal to the number of labels in the line number list. Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 input "What character group ? ";T
110 if T<1 or T>2 then goto 100
120 on T restore $S,$L
130 for I=1 to 5
140     read A$
150     print A$;
160 next I
170 end
180$S
190 data a,b,c,d,e
200$L
210 data A,B,C,D,E
```

# OPEN

**FORMAT :** OPEN <file name> [for {input | output | append}] as#  
<CH> [Record <record length>] {CR |:}

**FUNCTION :** Opens user files, for further usage.

**Comments :** File names must have no more than 9 characters.

Input is used in conjunction with the INPUT # command, and is for reading data from a file. Output is used in conjunction with the PRINT # command, and is for writing data to a file.

Append is used like output, but only when you are adding data to an already existing file.

**Example :**

```
100 open "PRT:" as#2
110 Print#2 "good"
120 close#2
130 end
```

# PAINT

**FORMAT :** PAINT <GR-coordinates> [, <boundary-color> [...]]  
{CR |;}

**FUNCTION :** Paints an area delimited by the GR-coordinates using one of up to 16 colors indicated by boundary-color.

**Comments :** Even if the boundary-color is omitted, the appropriate area will not be colored transparent (invisible).

**Example :**

```
100 console,,0,0,1
110 cls
120 ginit
130 for I=1 to 10
140     X%=rnd(230%)
150     Y%=rnd(180%)
160     WX%=rnd(50%)+10%
170     WY%=rnd(50%)+10%
180     COL%=rnd(13%)+2%
190     fcol COL%
200     gmove X%,Y%
210     draw X%+WX%,Y%;X%+WX%,Y%+WY%;
           X%,Y%+WY%;X%,Y%
220     fcol rnd(13%)+2%
230     Paint X%+WX%/2%,Y%+WY%/2%,COL%
240 next I
250 console,,1
```

# PLOT

**FORMAT :** PLOT <GR-coordinates> [;,,] {CR |:}

**FUNCTION :** Displays the dot associated with the coordinates. Use the color set up by a FCOL statement.

**Comments :** After execution, the graphics cursor will reside at these coordinates.

**Example :**

```
100 console,,0,0,1
110 cls
120 ginit
130 for I=1 to 300
150     fcol rnd(13%)+2%
160     Plot rnd(255%),rnd(191%)
180 next I
190 console,,,1
```

# POKE

**FORMAT :** POKE <memory address> [<data> [...]] {CR |:}

**FUNCTION :** POKE writes data directly into specified locations in the computer memory.

**Comments :** The data may also be an expression to be evaluated before the rest of the statement. Care should be taken with memory addresses and data contents. Certain locations will destroy the current program in memory. Programs written for other computers will not work on the M5 if they have POKE (or PEEK) statements without special modifications to these statements.

**Example :** 100 P o k e &F000,&FE



# POKEW

**FORMAT :**     POKEW <memory-address> [, <data> [...]] {CR | ;}

**FUNCTION :**   Writes the data to the specified memory address in CPU memory.

**Comments :**   The data must be numeric. The lower-order byte is written in the specified address while the upper-order byte is written in address+1.

**Example :**

```
100 cls
110 B%=&FFFF
120 A%=varPtr(B%)
130 PokeW A%,&1000
140 Print hex$(B%)
150 end
```

# PRINT

**FORMAT :** PRINT [# <channel number>] <expression>,... {CR |:}

**FUNCTION :** Puts text in the screen display buffers.

**Comments :** PRINT is also used to send control characters which may not appear as characters but which have important effects on the screen buffers. Most characters sent by PRINT to the screen buffers are merely deposited and appear on the screen. Control characters, on the other hand, may clear the contents of a buffer, change which buffer will be displayed on the screen, erase a character from the buffer, or cause future PRINT commands to send characters to one or the other buffer. It is important to realize that the PRINT command can put characters into the buffer that is not currently visible.

**Example :** 100 print "Good day, eh?"

```
run
Good day, eh?
```

```
Ready
```

# PUT

**FORMAT :** PUT [# <CH>][[.] <expression> [...]]{CR |:}

**FUNCTION :** Assigns a binary form to the value of an expression and outputs it.

**Comments :** Binary form is integral in BASIC; the chief purpose of this command is to create records of fixed length. It ensures that when output, characters will be of constant length.

**Example :**     put#3 CODE%,NAME\$,TEL\$

# RANDOMIZE

**FORMAT :** RANDOMIZE {CR |;}

**FUNCTION :** Resets the seed for the random number generator.

**Comments :** This affects RND

**Example :**

```
10 randomize
20 for I=0 to 20
30 R=rnd(1)
40 Print R
50 next I
60 end
```

# READ

**FORMAT :**     READ <variable list> {CR |:}

**FUNCTION :**   READ loads data from DATA statements into variables.

**Comments :**   When a READ statement is issued, the variable gets the value of the data item at the current value of the data pointer. The data pointer is maintained by BASIC-F. When you type RUN the data pointer is set to the first data item in the first DATA statement wherever it appears in your program. After the first READ you must exercise care in using READs, to ensure that you are aware of the position of the data pointer. When modifying the program always ensure that the order of the DATA statements is preserved. Use the RESTORE command with labels, to ensure correct positioning of the data pointer.

**Example :**

```
100 read X,Y$
110 Print X,Y$
120 end
130 data 200,John

run
200      John

Ready
```

# RECORD

**FORMAT :** RECORD # <CH> ][,] <Record number> {CR |:}

**FUNCTION :** Specifies the record to be accessed next, and the execution order of reading and writing.

**Comments :** A record is a collection of files of uniform length. The record number may range from 0 to 65535, and the length is specified by OPEN. Random access is available. Normal execution order is write then read.

**Example :** record#3,CD%

# REM

**FORMAT :** {rem [|'|} <comments>

**FUNCTION :** Stores programmer comments within the program

**Comments :** REM is short for 'remark'. Remarks are not executed by the computer. It is recommended that remarks be placed liberally throughout all programs to help when debugging programs. Unfortunately REM statements can use up a lot of memory. A trade-off must be made between readability and ease of modification and debugging. No statements may occur after a REM in BASIC-F. This is unlike most other BASICs.

**Example :**

```
100 rem This Program was written b'y
    Mark Voumard
110 '
120 'the following section initializes
    the main variables
130 'and opens files
```

# RENUM

**FORMAT :**     RENUM [<new line number>][ , <old line number>][ , <step>]

**FUNCTION :**   Changes the line numbering of the program. Maintains all GOTO and GOSUB commands.

**Comments :**   As a program grows in size, more lines of code may need to be added than was anticipated. If room runs out, RENUM can be used to create more space and even out the line numbering.

**Example :**

```
100  A,B,C=1
113  for I=1 to 10
126      A=A+I
139      B=B-I
152      C=C*I
165  next I
178  Print A,B,C
191  end

Ready
renum 1000,100,20

list
1000  A,B,C=1
1020  for I=1 to 100
1040      A=A+I
1060      B=B-I
1080      C=C*I
1110  next I
1120  Print A,B,C
1140  end

Ready
```



# REPEAT

**FORMAT :** REPEAT {CR |:}

**FUNCTION :** Sets up a loop with a logical test at the end. The end of the loop is specified with an UNTIL.

**Comments :** The loop is used much like a FOR NEXT loop where no counter is required. A REPEAT loop is always executed at least once! The test to end the loop is in the UNTIL statement at the END of the loop. When a REPEAT instruction is encountered, BASIC-F marks the line and continues execution of the next and succeeding statements. When an UNTIL statement is encountered, the logical test is evaluated (see UNTIL). If it is true the last REPEAT mark is removed and execution continues on the line following the UNTIL. If it is false, execution continues on the line following the last marked REPEAT statement.

**Example :**

```
100 A=0
110 repeat
120     A=A+1
130     print A
140 until A=100
150 end
```

# RESTORE

**FORMAT :**     RESTORE [<destination>] {CR |:}

**FUNCTION :**   Resets the data pointer for a group of data items in DATA statements.

**Comments :**   This function is convenient when you need to extract several categories of information from the same group of data.  
Destination = {line number | label name | numeric variable  
(line number) | string variable ( label name)}

**Example :**

```
100 restore $DATA2
110 for I=1 to 5
120     read A$,A
130     print $A,A
140 next I
150 end
160$DATA1
170 data A=,100,B=,130,C=,300
180$DATA2
190 data D=,235,E=,432,F=,554,G=,123,
    H=,444
```

# RESUME

**FORMAT :** RESUME [<destination>] CR |;

**FUNCTION :** Bypasses an error and begins execution from line number.

**Comments :** When line-number is omitted, the next statement is executed.  
Destination = {line number | label name | numeric-  
variable (line number) | string variable (label name)}

**Example :**

```
100 on error gosub $ERR
110 A$="":B=65
120 A%=A%+chr$(B)
130 print A$
140 goto 120
150 end
160 $ERR
170 A$="":B=B+1
180 if B>90 then resume 150
190 resume 120
```

# RETURN

**FORMAT :** RETURN [ <destination> ] {CR | ;}

**FUNCTION :** Returns the program control to the point immediately after the point from where it was called by a GOSUB.

**Comments :** A RETURN must not be executed if it has not been preceded by a GOSUB. Thus a subroutine which ends in a RETURN must always be called by a GOSUB. GOSUB and RETURN should be used frequently to create a modular program. Using the optional line number version of RETURN generally defeats the purpose of GOSUB.  
Destination = {line number | label name | numeric variable (line number) | string variable (label name)}

**Example :**

```
100 gosub $EXAM
110 gosub $DISPLAY
.
.
.
1000$EXAM
.
.
.
1490 return
1500$DISPLAY
.
.
.
1320 return 100
```

# RUN

**FORMAT :** RUN [< destination > | < file name >] {CR | :}

**FUNCTION :** Executes the current program.

**Comments :** The optional line number can specify the starting point of the current execution. All variables are cleared when a RUN is executed.

**Example :**     run

# SAVE

**FORMAT :**     SAVE <file-name> [, <first-address>, <last-address>  
                  [, <start-address>]] {CR |:}

**FUNCTION :**   Writes to external memory.

**Comments :**   When a file name only is specified, or when an address is  
                  specified in a Basic-G program, this command will write  
                  from the CPU's optional memory area to tape or disk.

**Example :**     save "CMT:PROG1"

# SCOD

**FORMAT :** SCOD <sprite-number>, < character-code> {CR |:}

**FUNCTION :** Assigns character-code to sprite-number.

**Comments :** Sprite numbers can be from 0 to 31 only.

**Example :**

```
100 console,,,2
110 cls
120 stchr "00183c66db7e2400" to &7F,0
130 scod 0,&7F
140 scol 0,4
150 ma9 1
160 loc 0 to 100,100
170 end
```

# SCOL

**FORMAT :** SCOL <sprite-number>, <color> {CR |;}

**FUNCTION :** Colors sprite-number using color code.

**Comments :** Only the character color can be changed, not the pixel background color.

**Example :**

```
10 SCOL TEST
20 Print "USL"
30 stchr "00183c66db7e2400" to &7F,0
40   scod 0,&7F
50   for I=1 to 14
60     scol 0,I
70     loc 0 to 128,96
80     sleep 1
90   next I
```



# SG

**FORMAT :** \$G <channel-number>, [{<frequency> | <noise>}]  
[, <volume>] {CR |:}

**FUNCTION :** Turns the three tone generators and noise generator on and off as well as making them produce sound effects.

**Frequency :** <channel-number>  
0 = tone generator channel 0  
1 = tone generator channel 1  
2 = tone generator channel 2  
3 = noise generator

<frequency>  
Frequency value varies from 1 to 1023; 1 is the highest frequency and 1023 is the lowest. (1024 is the default.)

<noise>  
When the noise generator is used, channel 3, 0~7 specifies the type of noise. 0~3 are tone while 4~7 are variations of white noise. Notice noises 3 and 7 are dependent on the frequency of channel 2 (even if channel 2 is not on).

Noise	Frequency
0	N/512
1	N/1024
2	N/2048
3	dependent on channel 2
4	N/512
5	N/1024
6	N/2048
7	dependent on channel 2

<volume>  
Varies from 0~15 with 15 being the loudest.

# SLEEP

Example :

```
100 for I=1 to 10
110   for J=18 to 45 step 3
120     s9 1,30,J/10
130     s9 2,J,0
140     s9 3,7,J/3
150     sleep 8,1
160   next J
170   for K=45 to 18 step -3
180     s9 1,30,K/10
190     s9 2,J,0
200     s9 3,7,K/3
210     sleep 5,1
220   next K
230   s9 1,,0:s9 3,,0
240   sleep 5,1
250 next I
260 end
```

# SLEEP

**FORMAT :** SLEEP <sleep-count> [, <base-time>] {CR 1:}

**FUNCTION :** Stops execution for the specified sleep time. BASIC event interrupts will be ignored. However, machine language interrupts will be processed.

**Comments :** Sleep time is (sec):  
 $\text{sleep-count} \times \text{increment-time}/60$   
If the increment-time is omitted, 60 is assumed.

**Example :**

```
100 cls
110 for I=1 to 100
120     Print I,
130     if I>50 then sleep 30,1
140 next I
150 end
```

# STCHR

**FORMAT :** STCHR <pattern-code> TO <character-code>  
[, <character-set-number>] {CR |;}

**FUNCTION :** Assign a pattern-code to character-code (usually a hexadecimal number associated with an ASCII-code) to determine a character's shape.

**Comments :** A character can also be colored using the STCHR statement in the GII mode.

<Character-set-number>

- 0 = sprites
- 1 = for character patterns in other than the GII mode or character patterns in the top third of a GII screen
- 2 = for character patterns in the middle third of a GII screen
- 3 = for character patterns in the bottom third of a GII screen
- 4 = for color codes for characters in the top third of a GII screen
- 5 = for color codes for characters in the middle third of a GII screen
- 6 = for color codes for characters in the bottom third of a GII screen
- 7 = for character patterns in the entire screen
- 8 = for color codes for characters in the entire screen

Note: 2 to 8 correspond to the GII mode. The character 'A' is constructed below: its pattern code is to the right (use hexadecimal notation for its pattern code).

**Example :**

```
100 console,,0,0,2
110 cls
120 stchr "00183c66db7e2400" to ascii
    ("A"),1
130 repeat
140     locate rnd(31%),rnd(23%)
150     Print "A"
160     A$=inkey$
170 until A$<>" "
180 end
```

# STEP

**FORMAT :** <eb> STEP {on | off} {CR | :}

**FUNCTION :** STEP ON stops execution whenever the statements in the program change. STEP OFF resumes normal execution.

**Comments :** The number of the statement to be executed after STEP ON is indicated as follows:  
STOP AT 0000 where 0000 is the statement number.

**Example :**

step on

```
Ready
run
stop AT 100
read Y
control
stop AT 110
```

# STOP

**FORMAT :** STOP {CR |:}

**FUNCTION :** Halts execution of a program from within the program.

**Comments :** A program stopped by STOP can be restarted by CONT.  
This is useful for debugging programs.

**Example :**

```
100 Print "abcdefg"
110 stop
120 Print "hijklmn"

run
abcdefg
Stop at 110
Ready
```

# SWAP

**FORMAT :** SWAP <variable>, <variable> {CR |:}

**FUNCTION :** Transfers the values or contents of designated variables.

**Comments :** Both variables must be of the same form.

**Example :**

```
10 'Bubble sorting
20 on error gosub $ER
30 dim A(10)
40 gosub $INPUT
50 for I=0 to 10
60   for J=I to 10
70     if A(J)<=A(I) then swap A(J),A(I)
80   next J
90 next I
100 gosub $OUTPUT
110 end
120 $INPUT
130 for K=0 to 10
140   input A(K)
150 next K
160 return
170 $OUTPUT
180 for OP=0 to 10
190   Print A(OP)
200 next OP
210 end
220 $ER
230 if err=25 then resume
240 Print err,err1
```

# TAB

**FORMAT :** TAB (X)

**FUNCTION :** Moves the cursor X character spaces right.

**Comments :** TAB should be followed by a ';' in order to print material in particular columns on the screen.

**Example :**

```
100 Print tab(10); "This is column  
10"  
Run  
  
This is column 10  
  
Ready
```



# TAPE

**FORMAT :** TAPE {CR |}

**FUNCTION :** Accesses the assembler supplied on tape.

**Comments :** For use with external data tape recorder only.

**Example :**     t y p e

This will read the machine language from tape into the M5 memory.

# THETA

**FORMAT :** THETA {0|1} {CR |:}

**FUNCTION :** Sets the mode for trigonometric functions to degrees or radians.

**Comments :** 0 is radian mode and the default. 1 is degree mode.

**Example :**

```
100 theta 0
110 Print sin(30)
120 theta 1
130 Print sin(30)
140 end
```

```
run
-0.9880316240929
0.5
```

Ready

# TRACE

**FORMAT :** TRACE {ON|OFF} {CR|:}

**FUNCTION :** Displays a trace of executed line numbers while the program is executing.

**Comments :** TRACE allows the programmer to see which lines of the program are to be executed. This is useful only for debugging programs under development.

**Example :**

```
100 for I=1 to 2
110   Print "hi"
120   Print "done"
130 next I
140 Print "end"
150 end
trace on
run
Ready
```

# TYPE

**FORMAT :** TYPE {int | dbl | str} {CR | ;}

**FUNCTION :** Defines the type of a variable.

**Comments :** In cases where the type designation of a constant or a variable has been omitted, the TYPE command designates the type given to the variable or constant. Character-type designation is effective only with variables.

**Example :**

```
type str

Ready
list
10 A="ABC"
20 B%=3000#
30 C#=12.345#

Ready
type dbl

ready
list
10 A$="ABC"
20 B%=3000
30 C=12.345
```

# UNTIL

**FORMAT :** UNTIL < conditional expression > {CR |:}

**FUNCTION :** Marks the end of a REPEAT loop.

**Comments :** The REPEAT loop is demarcated by a REPEAT at the beginning and an UNTIL at the end. All the instructions between them are executed REPEATEDly until the expression in the UNTIL statement is TRUE. See REPEAT.

**Example :**

```
100 repeat
110   X%=rnd(10%)
120   Print X%
130 until X%=8%
```

This loop will print random integers between 0 and 10 until it hits an 8.

# VERIFY

**FORMAT :** VERIFY [<file-name>] {CR |:}

**FUNCTION :** Compares a program in memory to another stored on cassette tape.

**Comments :** When the file-name is omitted, the first file found on external memory is compared.

**Example :** VERIFY "CMT:PROG1"

# VIEW

**FORMAT :** VIEW [<x0>, <y0>, <x1>, <y1>] {CR |:}

**FUNCTION :** Creates a view-port on the display screen.

**Comments :** A view-port is a logical screen display. It can be used to temporarily reduce the amount of the screen used for display. When a window is in effect the cursor cannot be moved outside it. The top left corner of the view-port becomes cursor position 0,0. Thus CURSOR and LOCATE commands work relative to the new view-port. In general the M5 can be said to have a view-port of 0,0,39,23 when the power is turned on. X0,Y0 is the top left corner of the new view-port and X1,Y1 is the bottom left. X is the column and Y the row. VIEW does not clear the new view port. A view-port can be used to do relative character positioning for easily displaying graphs.

**Example :** 100 view 5,5,25,20

This example creates a window of four lines at the bottom of the screen.

# VPOKE

**FORMAT :** VPOKE <memory address>, <output data> {CR|:}

**FUNCTION :** Outputs data to the video memory.

**Comments :** The data may also take the form of an expression. The data may not be a list as in POKE.

**Example :**

```
vPoke &2000,&ff
Ready
Print vPeeK (&2000)
255
Ready
```



# VSAVE

**FORMAT :** <VSAVE <file-name>,<start-address>,<end-address>  
{CR |}

**FUNCTION :** Writes Video RAM data to external memory.

**Comments :** Writes VRAM data between these two addresses to cassette tape.

**Example :** vsave "cmt:PIC1"

# WAIT

**FORMAT :** WAIT <time out count>[, <base time>] {CR |:}

**FUNCTION :** WAIT limits the amount of time that the computer will wait for input from the keyboard.

**Comments :** The actual wait time in seconds is computed as:

$$\text{wait} = \text{time out count} * (\text{base time} / 60)$$

If the time out count is 0 then the timer will not function. If the base time is 0 then it is set to 256. If the base time is left out, then it is set to 60. This function is most useful in writing games in which response time is important.

**Example :**

```
10 rem wait for 0.5 second
20 wait 30,1
30 input "data";A$
40 Print A$
50 goto 20
```

## 3.1. Functions

In this section all the BASIC-F functions are defined. The layout is similar to the previous section. The FORMAT entry is slightly different; all functions may appear anywhere in an expression hence we do not specify the context syntax. Functions can not be executed directly without the use of a statement such as PRINT. Functions which take parameters are shown with sample variable names; constants may also be used. If more than one type is possible, variables of both types are shown.

# ABS

**FORMAT :** ABS (X)

**FUNCTION :** Returns the absolute value of X.

**Comments :** X must be a number.

**Example :** Print abs(-34)  
34

Ready

# ASCII

**FORMAT :** ASCII (X\$) {CR |:}

**FUNCTION :** Returns the ASCII code for the first character of string X\$.

**Comments :** As a matter of legibility X\$ should be one character long or replaced with ASCII(LEFT\$(A\$,1)). 'ASCII' stands for 'American Standard Code for Information Interchange'. The converse function is CHR\$(X).

**Example :**

```
100 A$="ABC"  
110 Print ascii(A$)  
120 end
```

# ATN

**FORMAT :** ATN (X)

**FUNCTION :** Returns the arc tangent of X.

**Comments :** Legal range for X is  $\pm 6.8 \text{ E } 74$   
Care should be taken to use either radians or degrees consistently.

**Example :**

```
100 A=atn(10)
110 Print A
120 end
```

# CALC

**FORMAT :** CALC (<string>)

**FUNCTION :** Performs a BASIC-F operation on an expression represented as a string.

**Comments :** This function allows BASIC-F to compute expressions unknown at the time the program is written. This is a very powerful function, but it may introduce problems in debugging. This function is useful in mathematical programs involving equations unknown to the programmer. See also EXE.

**Example :**

```
10 'Calculation
20 input "calc";A$
30 A=calc(A$)
40 Print A$;"=";A
45 Print
50 goto 20
```

# CDBL

**FORMAT :** CDBL (X)

**FUNCTION :** Converts an integer into a real number.

**Comments :** None.

**Example :**

```
100 X%=10%
110 Print XX/3%
120 Print cdbl(XX)/3%
130 end

run
3
3.33333333333333
Ready
```



# CHR\$

**FORMAT :** CHR\$ (X)

**FUNCTION :** Returns the character whose internal code is X.

**Comments :** This function is useful for accessing invisible or non-displayed characters. A complete list of character codes can be found in the appendices. X must be a valid ASCII code.

**Example :**

```
10 ' CHR$ TEST
20   for I=65 to 90
30     Print chr$(I);
40   next I
```

# CINT

**FORMAT :** CINT (R)

**FUNCTION :** Converts a real number to an integer.

**Comments :** Rounding occurs in the conversion.

**Example :**

```
100 Print cint(1.4)
110 Print cint(1.5)
120 end
```

# COS

**FORMAT :** COS (X)

**FUNCTION :** Returns the cosine of X.

**Comments :** As with all trigonometric functions, care should be taken to use either radians or degrees consistently. The range of values for X is  $\pm 2.8 \text{ E } 16$ .

**Example :**

```
100 A=cos(Pi/3)
110 Print A
120 end
```

# ERR

**FORMAT :** ERR

**FUNCTION :** Returns the error code of the most recent error.

**Comments :** This function, along with ERRL and ERRL\$, is useful for programs which trap errors and attempt to deal with them internally.

**Example :**

```
100 $ABC
110 A=B$
120 print A
130 return
```

# ERRL

**FORMAT :** ERRL

**FUNCTION :** Returns the line number of the most recent error.

**Comments :** See also ERR and ERRL\$.

**Example :**

```
100$ABC
110 A=B$
120 Print A

run
Err 13 in 110
Ready
print err, errl, errl$
      13      110      ABC
```

Ready

# ERRL\$

**FORMAT :** ERRL\$

**FUNCTION :** Returns the label of the line in which the most recent error occurred.

**Comments :** See also ERR and ERRL.

**Example :**

```
1000 $ABC
110 A=B$
120 print A
130 return

run
Err 13 in 110
Ready
print err, errl, errl$
13 110 ABC

Ready
```

# EXE

**FORMAT :** EXE <character string> {CR |:}

**FUNCTION :** Executes a BASIC-F statement which has a string representation.

**Comments :** This statement allows the program to execute BASIC-F statements which are unknown when the program is written. It allows the program to write program segments and execute them. The string must not contain another EXE command and an error will occur if there is a loop stack. This statement may also result in programs which are very difficult to debug. Since the program may execute another program, which does not exist in the original program, bugs may be difficult to find. This function is very useful for a BASIC-F program which can control other BASIC-F programs, and still maintain control of the computer. See also CALC.

**Example :**

```
100 on error gosub $ERR
110 input:P$
120 exe "gosub "+P$
130 goto 110
140$AAA
150 print "AAAAAAA"
160 return
170$BBB
180 print "BBBBBBB"
190 return
200$CCC
210 print "CCCCCCC"
220 return
230 $ERR
240 if err=0 then return
250 resume 110
```

# EXP

**FORMAT :** EXP (X)

**FUNCTION :** Returns e to the power of X.

**Comments :** The range of X is from - 175 to + 175 exclusive.

**Example :**

```
100 A=exp(2)
120 Print A
130 end
```



# FIX

**FORMAT :** FIX (X)

**FUNCTION :** Returns the truncated integer portion of X.

**Comments :**  $\text{FIX}(X) = \text{SGN}(X) * \text{INT}(\text{ABS}(X))$

**Example :**

```
100 X=1.732050807569
110 Y=fix(X)
120 Print Y
130 end
run
1
```

Ready

# FRE

**FORMAT :** FRE (X)

**FUNCTION :** Returns information about memory usage.

**Comments :** FRE returns five values according to the following table:

- 0 - maximum size of work area
- 1 - remaining user area
- 2 - remaining free work area
- 3 - remaining free user area and work area
- 4 - last address used by BASIC

**Example :** `Print fre(3)`  
`6849`

This represents the total free RAM available to the user.

# HEX\$

**FORMAT :** HEX\$ (X)

**FUNCTION :** Returns the hexadecimal equivalent of X in four hexadecimal places.

**Comments :** No Zero suppression.

**Example :** Print hex\$(65535)  
FFFF  
Ready

# INKEY\$

**FORMAT :** INKEY\$

**FUNCTION :** Reads in a character from the keyboard input buffer, without waiting for a keystroke.

**Comments :** This function takes a character without waiting, like INPUT does. If the user has typed a character, it will be read in; otherwise the character will be a null (00). This function is useful in games as well as for writing 'bomb' proof input function. With INKEY\$ it is impossible to create an error as with INPUT, so routines can be written which will accept input of any form without generating BASIC-F errors.

**Example :**

```
100 A$=inkey$
110 if A$(<>)chr$(13) then goto 100
120 print "Exit"
130 end
```

# INP

**FORMAT :** INP(X)

**FUNCTION :** Accepts one byte of input from port X.

**Comments :** X must be in the range 0-255. A port will always provide a byte without waiting for input. For example, a program to read a serial port connected to a modem will always have a byte ready even if the modem has not received a byte. To use a serial port correctly two actual ports must be read. The first port will have a bit to indicate when a byte has come to the other port from the sending device, modem or otherwise.

**Example :**

```
100rem MODEM = SIO Port address
110 repeat
120   X=inp(STATUS)
130 until X=2
140 X=inp(MODEM)
150 C$=chr$(X)
160 return
```

This loop will wait until it receives a byte from the port. For this loop to work STATUS must be assigned the number of the status port on your serial port and MODEM must be assigned the number of the serial port itself.

# INSTR

**FORMAT :** INSTR ( [X,] STR\$, SUB\$)

**FUNCTION :** Searches string STR\$ for the first occurrence of string SUB\$.

**Comments :** The search for SUB\$ starts after the Xth character in STR\$. If the substring is not found, then the result returned is zero. This function is very useful for some applications but consumes a great deal of time, especially if the strings involved are long.

**Example :**

```
100 A$="ABCDEFGH"
110 B$="D"
120 C=instr(1,A$,B$)
130 Print C
140 C=instr(5,A$,B$)
150 Print C
160 B$="H"
170 C=instr(1,A$,B$)
180 Print C
190 end
```

```
run
4
0
0
```

```
Ready
0
```

# INT

**FORMAT :** INT (X)

**FUNCTION :** Truncates the real and returns only the integer portion of X.

**Comments :** Use the FIX function to round a real number.

**Example :**

```
100 A=10:B=1.5
110 Print A*B
120 Print A*int(B)
130 end
```

run

15

10

Ready

# LEFT\$

**FORMAT :** LEFT\$ (X\$, Y)

**FUNCTION :** Returns the left substring of X\$ whose length is Y

**Comments :** Care must be taken to ensure that Y does not exceed the length of X\$.

**Example :**

```
100 A$="12345ABCDEvwxyz"
110 B$=left$(A$,5)
120 Print B$
130 end

run
12345

Ready
```



# LEN

**FORMAT :** LEN (X\$)

**FUNCTION :** Returns the length of character string X\$.

**Comments :** The maximum length of a character string is 18.

**Example :**

```
100 A$="ABCDEFGH"
110 B=len(A$)
120 Print B
run
7
```

Ready

# LN

**FORMAT :** LN (X)

**FUNCTION :** Returns the natural log of X.

**Comments :** The natural log is a logarithm with base e.

**Example :**

```
100 for I=1 to 10
110   A=ln(I)
120   Print I; " -> "; A
130 next I
140 end
```

# LOG

**FORMAT :** LOG (X)

**FUNCTION :** Returns the log of X to the base 10.

**Comments :** None.

**Example :**

```
10 for I=1 to 10
20   A=log(I)
30   Print I;"->";A
40 next I
50 end
```

# MID\$

**FORMAT :** MID\$ (X\$, X [,Y])

**FUNCTION :** Returns the substring of X\$ starting at character X and ending at character Y.

**Comments :** The maximum length of a character string is 18.

**Example :**

```
100 A$="12345ABCDEvwxyz"  
110 B$=mid$(A$,6,5)  
120 Print B$  
130 end
```

# NUM\$

**FORMAT :** NUM\$ (X)

**FUNCTION :** Converts the numeric value of X to its character equivalent.

**Comments :** Do not exceed maximum integer value.

**Example :**

```
100 A=999
110 B$="ABC"
120 C$=B$+num$(A)+"XY"
130 Print C$
140 end
```

```
run
ABC 999XY
Ready
```

# OUT

**FORMAT :** OUT <port number>, [<output data>[...]] {CR |:}

**FUNCTION :** Sends data byte by byte to a port.

**Comments :** The output data may be the result of an expression. Care should be taken with port numbers as both hexadecimal and decimal numbers are used to refer to both ports and data. Output to an incorrect port could damage a program saved on tape or disk.

**Example :** 100 out %20,%10

# PEEK

**FORMAT :** PEEK (X)

**FUNCTION :** Returns the contents of memory address X.

**Comments :** Returns 8 bits stored in CPU memory address X.

**Example :**

```
100 A=peek(&FFFF)
110 Print A
120 end
```

# PEEKW

**FORMAT :** PEEKW (X)

**FUNCTION :** Returns 16-bits from CPU memory address X and address X+1.

**Comments :** This will return both X and X+1.

**Example :**

```
100 poke &7200,&30
110 poke &7201,&40
120 A=peekw(&7200)
140 Print int(A/256)
150 Print A and 255
160 end
```



# PI

**FORMAT :** PI

**FUNCTION :** Returns the value of Pi.

**Comments :** Pi = 3.14159265359

**Example :**

```
100 A=sin(Pi/4)
110 Print A
120 end
```

# RDST\$

**FORMAT :** RDST\$ (X)

**FUNCTION :** Reads the statement indicated by the cursor from the beginning.

**Comments :** Characters to the right of position X will not be read. The end of the statement is indicated in the display by '00. If X does not exhaust the statement, the string CR ('OD) is attached.

**Example :**

```
10 cls
20 Print cursor(5,1); "Hello there!"
30 locate 5,1
40 for C=1 to 13
50   A$=rdst$(C)
60   Print cursor(5,C+5); A$;
70   locate 5,1
80 next C
90 end
```

# REG

**FORMAT :** REG (X)

**FUNCTION :** Returns the register value after a CALL statement has been executed.

**Comments :** X can signify various registers:

0	AF
1	BC
2	DE
3	HL

Data is stored "high-low." Thus if Ar contains &2E and Fr contains &30 then "NEWAF" below will have &2E30. EX-CHG (NEWAF) gives &302E and EXCHG (NEWAF) AND 255 gives &2E.

**Example :**

```
10 cls
20 gosub $SETCALL:'Prepare for call
30 call ROUTINE,AFREG,,HLREG
40 AF=reg(0)
50 HL=reg(3)
60 print cursor(0,2); "AFREG=";hex$(AF),
  "HLREG=";hex$(HL)
70 end
80 $SETCALL
90 ROUTINE=&14BD
100 AFREG=&4100
110 HLREG=&3800
120 return
```

# RIGHT\$

**FORMAT :** RIGHT\$ (X\$, <length>)

**FUNCTION :** Returns a substring from the right side of X\$ of specified length.

**Comments :** In general all strings can be done with just the MID\$ instruction.

**Example :**

```
100 A$="12345ABCDEvwxyz"
110 B$=right$(A$,5)
120 Print B$
130 end
```

```
run
vwxyz
```

Ready

# RND

**FORMAT :** RND (X)

**FUNCTION :** Returns a random number between 0 and X.

**Comments :** X is the seed. For most purposes, RND produces an acceptable distribution of numbers. For extremely sensitive applications, however, other methods of random generation should be used, even to the point of special hardware.  
When X is an integer, the value returned will be an integer between 0 and X. When X is a real number, the value returned will be a real number between 0 and X.

**Example :**

```
100 randomize
110 A=rnd(10)
120 B=rnd(10%)
130 Print "rnd(10)=";A
140 Print "rnd(10%)=";B
150 end
```

# RPT\$

**FORMAT :** RPT\$ (<repetitions>, X\$)

**FUNCTION :** Returns a string which consists of <repetitions> of X\$.

**Comments :** This is useful for building long strings from a small pattern.

**Example :**

```
100 len 30
110 X$=rpt$(30,"*")
120 Print X$
130 end
run
```

\*\*\*\*\*

# SGN

**FORMAT :** SGN (X) {CR |:}

**FUNCTION :** Returns the sign of X.

**Comments :** SGN (0) = 0; if  $X > 0$  then SGN (X) = 1, otherwise SGN (X) = -1. X may be real or integer or hex.

**Example :**

```
100 A=10
110 Print sgn(A)
120 A=0
130 Print sgn(A)
140 A=-10
150 Print sgn(A)
160 end
```

# SIN

**FORMAT :** SIN (X)

**FUNCTION :** Returns the sine of X.

**Comments :** The range of values for X is  $\pm 2.8 \text{ E } 16$ .

**Example :**

```
100 A=sin(pi/3)
110 Print A
120 end
run
0.8660254037844
Ready
```



# SQR

**FORMAT :** SQR (X)

**FUNCTION :** Returns the square root of X.

**Comments :** X can be a real number or an integer.

**Example :**

```
100 A=2
110 B=sqr(A)
120 Print "SQR(2) = " ; B
130 Print "SQR(2)*SQR(2) = " ; B*B
140 end
```

# TAN

**FORMAT :** TAN (X)

**FUNCTION :** Returns the tangent of X.

**Comments :** The range of values for X is  $\pm 2.8 \text{ E } 16$ .

**Example :**

```
100 X=23
110 A=tan(X)
120 Print A
130 end
```

# TIME

**FORMAT :** TIME

**FUNCTION :** Returns the amount of time since the system was powered up. Value is in seconds.

**Comments :** The value returned is in seconds.

**Example :**

```
100 MIN=int(time/60)
110 SEC=time mod 60
120 Print MIN;"MIN";SEC;"SEC"
130 end
```

# VAL

**FORMAT :** VAL (X\$)

**FUNCTION :** Converts a character string into its numeric equivalent.

**Comments :** The string must be a legal number in BASIC-F.

**Example :**

```
100 A$="100"  
110 B$="55"  
120 Print A$+B$  
130 Print val(A$)+val(B$)  
140 end
```

# VARPTR

**FORMAT :**     VARPTR ({X|X\$})

**FUNCTION :** Returns the actual memory address of a variable.

**Comments :** This function is required when machine language subroutines are to access BASIC-F variables. After the address has been obtained it can be passed to the subroutine by any of several methods.

**Example :**

```
100 A%=255%
110 B=varPtr(A%)
120 C=peek(B)
130 Print C
140 end
```

# VPEEK

**FORMAT :** VPEEK (X)

**FUNCTION :** Returns the contents of video memory at location X.

**Comments :** Displays the data stored in video memory address X.

**Example :**

```
100 Print ""
110 Print cursor(0,0);"A"
120 U=vpeek(&3800)
130 Print "ASCII(" chr$(U) )="
140 end
```

# XCHG

**FORMAT :** XCHG (X)

**FUNCTION :** Swaps the order of the upper and lower bytes of X.

**Comments :** Using XCHG moves original bits 0-7 (upper byte) to new bits 8-15 (lower byte) and original bits 8-15 (lower byte) to new bits 0-7 (upper byte)

**Example :**

```
100 X%=123
110 A%=xchg(X%)
120 Print A%
130 end
```

## 4. Applications Section

This section contains a set of useful programs which will run on your M5. They have all been tested by our staff. These programs make use of the special features of the M5 and BASIC-F. Each program has a statement of general function, a description of how to run the program with a sample run, a description of how the program runs and the program listing itself.



## 4.1. Loan Repayments

This program calculates the time needed to repay a loan. Given the amount of the loan, the interest rate, the number of payments made per year and the amount of each payment, the program will return the length of time that it will take until the loan is completely repaid. The program calculates the time according to the following equation:

$$Y = \left( \frac{\log \left( 1 - \frac{(p.i)}{(N.R)} \right)}{\log \left( 1 + \frac{i}{N} \right)} \right) \frac{1}{N}$$

where  $Y$  = term of payment in years

$P$  = principal

$i$  = interest rate

$N$  = number of payments per year

$R$  = amount of each payment.

This could be used to calculate the length of time it would take to pay off a mortgage. How many years would it take to pay off a mortgage of \$20,000 at 18% by quarterly payments of \$1,000?

```

10 cls
20 '
30 rem compute the length of time to
40 ' repay a loan
50 '
60 locate 5,5
70 Print "Term of a loan"
80 Print "Interest rate"
90 '
100 rem input the data for the calculation
110 '
120 input "regular Payment ";R
130 input "Principal ";P
140 input "annual interest rate ";I
150 input "number of Payments Per Year ";N
160 '
170 rem perform computations
180 '
190 Y=-((log(1-(P*(I/100)/(N*R)))/(log(1+I/
    100/N)*N))
200 '
210 rem compute years and months from years
220 '
230 M=int(Y*12+0.5):Y0=int(M/12)
240 M=M-Y0*12
250 Print "Term=";Y0;" years,"
260 Print "and";M;" months,"
270 Print
280 '
290 input "Once more? (y/n) ";Y$
300 if left$(Y$,1)="y" then goto 10
305 '
310 Print
320 Print "done"
330 end

```

## 4.2. Initial Investment

This program calculates the investment necessary to provide a stated future value in a specified time period. Suppose you want to earn \$10,000 from interest over 5 years, how much money would you have to put in the bank to start with? For any given investment, interest rate and time period the program calculates the required initial investment. The program requires that you enter the time period, the goal total, the interest rate and the number of times that the interest is compounded in one year. The program bases its calculations on the following equation:

$$P = \frac{T}{(1 + i/N)^{N \cdot Y}}$$

where P = initial investment  
T = future value  
N = number of times compounded per year  
Y = number of years  
i = interest rate

The interest rate must be entered as a whole value: for instance 8.5% is entered as 8.5. The number of years can include fractional parts, such as 5.5 for five and one half years.

```

10 cls
20 locate 5,5
30 print "Initial investment"
40 print
50
60 rem input data for computations
70
80 print
90 input "Amount of goal ";T
100 input "# of compounds per year ";N
110 input "# of years ";Y
120 input "Nominal interest rate ";I
130
140 rem Perform computations
150
160 I=I/100
170
180 P=T/(I+1)^(N*Y)
190
200 print
210 print "Initial investment required"
220 print "to achieve goal total is $";
230 print int(P*100+0.5)/100
240 print
250 print
260 input "One more time? (y/n) ";Y$
270 if left$(Y$,1)="y" then goto 10
280 print
290 print "done"
300 end

```

where:  
 $P$  = amount of regular deposit  
 $T$  = future value  
 $I$  = interest rate  
 $N$  = number of deposits per year  
 $Y$  = number of years

### 4.3. Regular Deposits

### 4.3. Regular Deposits

This program calculates the amount of each deposit required to reach a goal total within a specified time period. Let us assume that you wish to save \$5,000 to buy a car. You wish to buy the car at the end of one year and you want to know how much money you need to put in the bank each month. Or let's say that you wish to save up the money for a down payment of \$10,000 on your home. You want to save the money over three years. How much must you put in the bank each month in order to get the desired total at the right time?

To use this program you must enter the values of the total, the number of years and the interest rate. You must also specify the number of times per year that you will make deposits. This allows you to skip one or more months. For example, you may wish to make 11 deposits per year, skipping December as your other expenses may be high that month. Or you may decide not to make payments over a vacation month but spend the money on your holiday. The interest rate that you enter must be expressed as a whole value, such as '10' for 10%, or '4.5' for four and a half percent. Years may be expressed as fractions as well, for example six months would be entered as '.5' for half of one year. The calculation for regular deposits is based on the following equation:

$$R = T \left( \frac{i/N}{(1 + i/N)^{N \cdot Y} - 1} \right)$$

where:

- R = amount of regular deposit
- T = future value
- i = interest rate
- N = number of deposits per year
- Y = number of years.

```

10 cls
20 '
30 rem compute the deposits required
40 rem to achieve a desired total at
50 rem some future date
60 '
70 locate 0,5
80 Print "Regular Deposit schedule"
90 '
100 Print
105 '
110 rem input necessary data
120 '
130 input "What is the desired goal":T
140 input "What is the interest rate":I
150 input "How many deposits per year":N
160 input "How many years to goal date":Y
170 '
180 rem perform the computations
190 '
200 I=I/N*100
210 R=T*I/((I+1)^(N*Y)-1)
220 '
230 rem round off to cents
240 '
250 R=int(R*100+0.5)/100
260 '
270 Print
280 Print "Each deposit must be $":R
290 '
300 Print
310 input "Another run (Y/N)":Y$
320 if left$(Y$,1)="Y" then goto 10
330 '
340 end

```

## 4.4. Future Value of Regular Deposits (Annuity)

This program calculates the total amount which will be saved if deposits are made regularly. Assume that a payment is made to an interest bearing account each month or at some other interval. What is the total amount in the account at any given time? For example, assume that you put \$75 from your pay into a special 'rainy day' account each month. How much will you have saved after a year and a half, or after five years? Or assume that your company matches your \$50 savings with a benefit in a company savings plan at 8%, how much will you save in three years? This program can perform the necessary calculations. You must enter the amount of each deposit, the number of deposits per year, the number of years and the interest rate. The number of years may contain a fractional part: for example three and a half years is entered as '3.5' years. The interest rate is entered as a whole value. For example, seven and eight tenths of a percent is entered as '7.8' for the program.

This program assumes that interest is compounded with each deposit according to the following equation:

$$T = R \cdot \left( \frac{(1 + i/N)^{N \cdot Y} - 1}{i/N} \right)$$

where:

T = total value after Y years  
R = amount of regular deposit  
N = number of deposits per year  
Y = number of years  
i = nominal interest rate

As an exercise, modify this program to calculate the result with interest compounded over different time periods. Many accounts today compound interest on a daily basis, rather than monthly. This involves changing the value of Y for years and the way the interest value is used.

```

10 Print chr$(12)
20 '
30 locate 0,5
40 Print "Future value of regular deposits"
50 Print " "
60 Print " "
70 '
80 '
90 '
100 input "Amount of regular deposits": R
110 input "nominal interest rate %": I
120 input "number of deposits per year ": N
130 input "number of years ": Y
140 '
150 rem calculate the interest per deposit
160 '
170 I=I/N/100
180 '
190 rem calculate annuity
200 '
210 T=R*((I+1)^(N*Y)-1)/I
220 '
230 Print "Future value = $";
240 Print int(T*100+0.5)/100
250 Print
260 input "More data? (y/n) ": Y$
270 '
280 if left$(Y$,1)="y" then goto 10
290 Print
300 Print "done"
310 end

```



## 4.5. Remaining Balance on a Loan

This program calculates the balance remaining on a loan after a specified number of payments has been made. You must input the amount of each payment, the number of payments per year, the amount of the principal, the annual interest rate, and the payment number from which you wish to calculate the remaining balance. The interest rate is entered as a whole value, thus 17% is entered as 17 and 6.5% is entered as 6.5.

For example, if you have a loan of \$10,000 at 12.5% interest and your payments are \$200.00 per month, how much will you still have remaining to pay after the 11th payment in the third year?

```

10 cls
20 '
30 rem compute the remaining balance
40 '
50 '
60 '
70 '
80 '
90 '
100 '
110 '
120 '
130 '
140 '
150 '
160 '
170 '
180 '
190 '
200 '
210 '
220 '
230 '
240 '
250 '
260 '
270 '
280 '
290 '
300 '
310 '
320 '
330 '
340 '
350 '
360 '
370 '
380 '
390 '
400 '
410 '
420 '
430 '
440 '

```

## 4.6. Prime Factors

The following program calculates the prime factorization of a given number. Prime factorization is the expression of a number as the product of its prime factors. The method is to check all possible factors from 2 up to the square root of the number. In each case each factor is removed until the next factor is required or the factorization is complete. When this program detects a prime, it prints out that a prime has been found. The program, as given, does not make use of indentation. As an exercise, rewrite the program using indentation. Also, the program contains a branch at 230, out of the FOR NEXT loop. This is particularly bad programming style. As an exercise, rewrite this program without branching out of a loop.

To run the program, simply type RUN and enter the number that you want factored. After the factors have been printed, the program will ask for another number to factor. When you are done, ask for 0 to be factored, and the program will halt. As an exercise, modify the program to print out all the prime factors of the numbers from one to 1000. Another good exercise would be to change the program to display the current factor being tested at the top of the screen. Factors already found could be displayed near the middle of the screen. When all factors have been found, the program should wait till the user is done, then continue. This will require that you use the cursor control abilities of the M5, under BASIC-F.

The following short program is a stripped-down version of the previous prime factors program. This program is used to generate prime numbers only. As an exercise, see how many primes your M5 can generate using this program. Can you think of any ways to make this program faster? As an exercise modify this program to print out only twin primes, i.e. those pairs of primes with only one number between them, like 5 and 7 or 17 and 19.

```

100 rem prime factors
110'
120 cls
130'
140 locate 4,5
145 print "Prime Factors"
147 print
150 input "What number do you want factored
    ":N
160'
170 if N=0 then end
180 if N=2 then print "2 is prime": goto 300
190'
200 M=2: SW=0
210 IN=int(sqrt(N+.5))
220 for L=M to IN
230 if N/L=int(N/L) then goto 250
240 next L
245 if SW=0 then print N: " is prime ": goto
    300
247 print N: goto 300
250 SW=1
260 if IN<L then goto 290
270 print L: "*":
280 M=L: N=N/L: goto 210
290 print L
300 print:print: goto 150

120 cls
145 print "Primes"
147 N=1
150 N=N+2
210 IN=int(sqrt(N+.5))
220 for L=3 to IN
230 if N/L=int(N/L) then goto 150
240 next L
245 print N:
300 goto 150

```

## 4.7. Long Number Arithmetic

This program allows your M5 to do simple arithmetic on very large numbers. Normally the size of the numbers that can be handled by your M5 under BASIC-F is limited by the BASIC-F interpreter. Integers have a limited size. With this program you can represent numbers of arbitrary size and do addition, subtraction and multiplication with them. The only limit on the size of the numbers is the memory capacity of your computer. In other words, except for the space occupied by the program itself, you can use the entire memory of your M5 to store 3 numbers using this program. This means that you can store incredibly large numbers.

The program works by breaking the large numbers into components which are small enough to be held in normal integer variables. These small parts of your number are stored in arrays. The arrays are A, B, and C and are declared in line 210. Array C is twice as large as A and B in order to accept large products of  $A*B$ . You can change the size of A, B, and C by changing the values in lines 180 and 160. M must always be  $2*N + 1$ . Also, you can control the largest value in each part of the number by changing line 200. If you set this value (P) to the maximum power of 10 that can be represented on the M5 in BASIC-F, then you will be able to store the largest number of digits possible with this program. The program itself is extremely modular, and each section does only one task. The first routine is used to read in numbers from the keyboard. When using the program, zeros must be entered if the first part of the number is not used. For example, the current listing of the program requires numbers broken into 5 parts. If you want to enter the number 100, the first four parts must all be 0, and they must be entered. As an exercise you might try to modify the program so that only the last number need be entered, to save time. The program has four more subroutines. Each of them is called with a GOSUB. The first routine is used to print the long numbers on the screen. The second is used for addition, the third for subtraction and the fourth for multiplication. There is no routine for division in the program. Notice that division can be done by repeated subtraction. As an exercise add a new subroutine to perform division by calling the subtraction routine until the remainder is smaller than the divisor. Print out the quotient and the remainder. As another exercise, you may wish to convert the input routine to use DATA statements, to save typing. Another interesting addition would be the ability to perform several operations, by reading in an expression and then executing it one step at a time.

Remember, the numbers must each contain five groups of four digits. The possible operations are +, -, and \*.

```

100 rem number operations" print 0401
110 ' "done 0402
120 console,,,3 print 0403
130 cls 0404
140 ' 0405
150 locate 4,5 0406
160 Print "Long number " OPERATIONS=" 0407
170 ' 0408
180 L=4: N=4: M=9 0409
190 ' 0410
200 P=10000 0411
210 dim A(N),B(N),C(M) 0412
220 ' 0413
230 Print: Print 0414
240 Print "Enter the first number" 0415
250 gosub $LOAD: rem read a number 0416
260 for I=0 to N: A(I)=C(I):next I 0417
270 ' 0418
280 Print "Enter the second number" 0419
290 gosub $LOAD 0420
300 for I=0 to N: B(I)=C(I):next I 0421
310 ' 0422
320 input "What operator? ":OP$ 0423
330 ' 0424
340 for I=0 to N: C(I)=A(I):next I 0425
350 gosub $SHOW: rem Print a 0426
360 Print 0427
370 Print OP$: rem Print operator 0428
380 for I=0 to N: C(I)=B(I):next I 0429
390 gosub $SHOW: rem Print 0430
400 Print 0431
410 Print "=" 0432
420 ' 0433
430 for I=0 to M: C(I)=0:next I 0434
440 rem call appropriate op 0435
450 ' 0436
460 if OP$="+" then gosub $ADD 0437
470 if OP$="-" then gosub $SUB 0438
480 if OP$="*" then gosub $MUL 0439
490 gosub $SHOW 0440
500 Print: Print "done" 0441
510 ' 0442
520 end 0443
530 ' 0444
1000 $LOAD: C(I)=C(I)+A(I) 0445
1010 ' 0446
1020 rem read a number 0447

```

```

1040     Print "enter 5 groups of 4 digits
        each"
1050     Print
1060     for I=N to 0 step-1
1070         input C(I)
1080     next I
1090 return
1100 '
2000 $SHOW
2010     SW=0
2020     for K=M to 0 step-1
2030         if C(K)=0 and SW=0 then goto $I1
2040         if SW=0 then SW=1:Print mid$(num$(
            C(K)),2)::goto $I1
2050         if C(K)=0 then Print "0000":goto
            $I1
2060         Print right$("0000"+mid$(num$(C(
            K)),2),4)
2070     $I1
2080     next K
2090     if SW=0 then Print "0":
2100     return
2110 '
3000 $ADD
3010 rem addition
3020 '
3030     CARRY=0
3040     for I=0 to N
3050         C(I)=A(I)+B(I)+CARRY=1
3060         CARRY=0
3070         if C(I)>=P then C(I)=C(I)-P:CARRY=1
3080     next I
3090     K=N+1
3100     C(K)=CARRY
3110 return
3120 '
4000 $SUB
4010 '
4020 rem subtraction
4030 '
4040     BRRW=0
4050     for I=0 to N
4060         C(I)=A(I)-B(I)-BRRW
4070         BRRW=0
4080         if C(I)<0 then C(I)=C(I)+P:BRRW=1
4090     next I
4100 return
4110 '
5000 $MUL
5010 '

```

---

```

5020 rem multiply
5030 '
5040     CARRY=0
5050     for I=0 to N
5060         CARRY=0
5070         for J=0 to N
5080             C(I+J)=C(I+J)+A(I)*B(J)+CARRY
5090             CARRY=0
5100             if C(I+J)>P then CARRY=int(C(I+J)/P)
5110             C(I+J)=C(I+J)-CARRY*P
5120         next J
5130     next I
5140 return

```



# CODE INFORMATION

## CHARACTER CODES

Following are the ASCII representations of all characters stored and displayed on the M5 computer.

To use this appendix, find the character you want to display. Then look at the row of numbers and letters across the top and find the one that lines up with your character. Now look to the left at the leftmost column of numbers and letters for the corresponding number or letter. Combine these two numbers or letters. The one you found first is followed by the second.

Let's look at three examples. Verify they're correct in the table below.

Character	Character code
\$	&24
H	&48
+	&2B

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

DEC	(HEX)	CODE	DEC	(HEX)	CODE	DEC	(HEX)	CODE
32	(20)		71	(47)	G	110	(6E)	n
33	(21)	!	72	(48)	H	111	(6F)	o
34	(22)	"	73	(49)	I	112	(70)	p
35	(23)	#	74	(4A)	J	113	(71)	q
36	(24)	\$	75	(4B)	K	114	(72)	r
37	(25)	%	76	(4C)	L	115	(73)	s
38	(26)	&	77	(4D)	M	116	(74)	t
39	(27)	'	78	(4E)	N	117	(75)	u
40	(28)	(	79	(4F)	O	118	(76)	v
41	(29)	)	80	(50)	P	119	(77)	w
42	(2A)	*	81	(51)	Q	120	(78)	x
43	(2B)	+	82	(52)	R	121	(79)	y
44	(2C)	,	83	(53)	S	122	(7A)	z
45	(2D)	-	84	(54)	T	123	(7B)	{
46	(2E)	.	85	(55)	U	124	(7C)	
47	(2F)	/	86	(56)	V	125	(7D)	}
48	(30)	0	87	(57)	W	126	(7E)	~
49	(31)	1	88	(58)	X	127	(7F)	4
50	(32)	2	89	(59)	Y	128	(80)	■
51	(33)	3	90	(5A)	Z	129	(81)	○
52	(34)	4	91	(5B)	[	130	(82)	♠
53	(35)	5	92	(5C)	\	131	(83)	♣
54	(36)	6	93	(5D)	]	132	(84)	~
55	(37)	7	94	(5E)	^	133	(85)	⬆
56	(38)	8	95	(5F)	_	134	(86)	T
57	(39)	9	96	(60)	`	135	(87)	
58	(3A)	:	97	(61)	a	136	(88)	-
59	(3B)	;	98	(62)	b	137	(89)	⬆
60	(3C)	<	99	(63)	c	138	(8A)	⬆
61	(3D)	=	100	(64)	d	139	(8B)	+
62	(3E)	>	101	(65)	e	140	(8C)	r
63	(3F)	?	102	(66)	f	141	(8D)	L
64	(40)	@	103	(67)	g	142	(8E)	⬆
65	(41)	A	104	(68)	h	143	(8F)	⬆
66	(42)	B	105	(69)	i	144	(90)	-
67	(43)	C	106	(6A)	j	145	(91)	-
68	(44)	D	107	(6B)	k	146	(92)	■
69	(45)	E	108	(6C)	l	147	(93)	■
70	(46)	F	109	(6D)	m	148	(94)	l

# APPENDIX A

DEC	(HEX)	CODE	DEC	(HEX)	CODE	DEC	(HEX)	CODE
149	(95)	I	192	(C0)		235	(EB)	+
150	(96)	■	193	(C1)	⌘	236	(EC)	▲
151	(97)	■	194	(C2)	⌘	237	(ED)	▲
152	(98)	◀	195	(C3)	⌘	238	(EE)	▲
153	(99)	▼	196	(C4)	⌘	239	(EF)	▲
154	(9A)	▲	197	(C5)	⌘	240	(F0)	-
155	(9B)	▶	198	(C6)	⌘	241	(F1)	-
156	(9C)	⌘	199	(C7)	⌘	242	(F2)	■
157	(9D)	⌘	200	(C8)	⌘	243	(F3)	■
158	(9E)	⌘	201	(C9)	⌘	244	(F4)	I
159	(9F)	⌘	202	(CA)	⌘	245	(F5)	I
160	(A0)	I	203	(CB)	⌘	246	(F6)	■
161	(A1)	⌘	204	(CC)	⌘	247	(F7)	■
162	(A2)	⌘	205	(CD)	⌘	248	(F8)	■
163	(A3)	⌘	206	(CE)	⌘	249	(F9)	■
164	(A4)	⌘	207	(CF)	⌘	250	(FA)	⌘
165	(A5)	⌘	208	(D0)	⌘	251	(FB)	⌘
166	(A6)	⌘	209	(D1)	⌘	252	(FC)	▲
167	(A7)	⌘	210	(D2)	⌘	253	(FD)	▲
168	(A8)	⌘	211	(D3)	⌘	254	(FE)	▲
169	(A9)	⌘	212	(D4)	⌘	255	(FF)	▲
170	(AA)	U	213	(D5)	⌘			
171	(AB)	⌘	214	(D6)	⌘			
172	(AC)	U	215	(D7)	⌘			
173	(AD)	⌘	216	(D8)	⌘			
174	(AE)	+	217	(D9)	⌘			
175	(AF)	■	218	(DA)	⌘			
176	(B0)	i	219	(DB)	⌘			
177	(B1)	⌘	220	(DC)	⌘			
178	(B2)	⌘	221	(DD)	⌘			
179	(B3)	⌘	222	(DE)	⌘			
180	(B4)	⌘	223	(DF)	⌘			
181	(B5)	⌘	224	(E0)	⌘			
182	(B6)	⌘	225	(E1)	⌘			
183	(B7)	⌘	226	(E2)	⌘			
184	(B8)	+	227	(E3)	⌘			
185	(B9)	⌘	228	(E4)	⌘			
186	(BA)	⌘	229	(E5)	⌘			
187	(BB)	⌘	230	(E6)	⌘			
188	(BC)	⌘	231	(E7)	⌘			
189	(BD)	⌘	232	(E8)	⌘			
190	(BE)	⌘	233	(E9)	⌘			
191	(BF)	⌘	234	(EA)	⌘			






























DEC → denotes base 10  
 HEX → denotes base 16

## COLOR CODES

Color	Color code
No color	0
Black	1
Green	2
Light green	3
Deep blue	4
Light blue	5
Deep red	6
Cyan	7
Red	8
Light red	9
Deep yellow	A
Light yellow	B
Deep Green	C
Purple	D
Gray	E
White	F

## CONTROL CODES

These are functions that control the screen, cursor and a few other specialized functions. When using control functions directly after a READY prompt, press the CTRL key and the control key simultaneously. But when using control codes in a program (for example, in a PRINT statement), first press the CTRL and SHIFT keys before pressing the control key. Also enclose the control character in double quotes.

Keyboard Key	Base 10	Base 16	Function	Program Usage Display
	0	00	Ignore	
A	1	01	Ignore	
B	2	02	Return cursor to beginning of current line	
C	3	03	Scroll screen display down	
D	4	04	Shift screen display left	
E	5	05	Scroll screen display up	
F	6	06	Shift screen display right	
G	7	07	Bell	
H	8	08	Backspace	
I	9	09	Tab the cursor eight spaces	
J	10	0A	Move cursor down one line	
K	11	0B	Move cursor to home position	
L	12	0C	Clear screen display	
M	13	0D	Same as RETURN key	
N	14	0E	Move cursor to beginning of next line	
O	15	0F	Change to standard mode	
P	16	10	Change to insert mode	
Q	17	11	Change to multi-color mode	
R	18	12	Change to GII graphics mode	
S	19	13	Change to GI graphics mode	
T	20	14	Return to text mode	
U	21	15	Change to visible screen	
V	22	16	Alternates between the visible and invisible screens, input is sent to the displayed screen	
W	23	17	Same as RETURN key	
X	24	18	Delete characters to the right of cursor	
Y	25	19	Alternates between the visible and invisible screens only	
Z	26	1A	Writes input to the alternate screen	
[	27	1B	Ignore	
	28	1C	Right arrow	
]	29	1D	Left arrow	
	30	1E	Up arrow	
	31	1F	Down arrow	

## Coding sheet for 8×8 pixel pattern code

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								


	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								


	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								


	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								


Coding sheet for 16×16 pixel pattern code

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
A				
B				
C				
D				
E				
F				

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
A				
B				
C				
D				
E				
F				

# CRT LAYOUT COORDINATES

CRT Screen Layout Sheet (Characters)

<div><div>x</div><div>y</div></div>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	<div><div>x</div><div>y</div></div>
0																									0
1																									1
2																									2
3																									3
4																									4
5																									5
6																									6
7																									7
8																									8
9																									9
10																									10
11																									11
12																									12
13																									13
14																									14
15																									15
16																									16
17																									17
18																									18
19																									19
20																									20
21																									21
22																									22
23																									23
<div><div>y</div><div>x</div></div>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	<div><div>y</div><div>x</div></div>

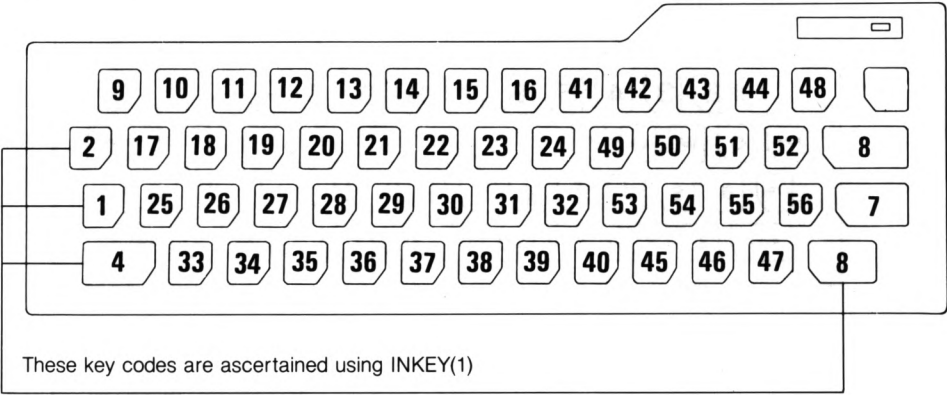


CRT Screen Layout Sheet (Pixels)

X		Y	
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
100	101	102	103
104	105	106	107
108	109	110	111
112	113	114	115
116	117	118	119
120	121	122	123
124	125	126	127
128	129	130	131
132	133	134	135
136	137	138	139
140	141	142	143
144	145	146	147
148	149	150	151
152	153	154	155
156	157	158	159
160	161	162	163
164	165	166	167
168	169	170	171
172	173	174	175
176	177	178	179
180	181	182	183
184	185	186	187
188	189	190	191
192	193	194	195
196	197	198	199
200	201	202	203
204	205	206	207
208	209	210	211
212	213	214	215
216	217	218	219
220	221	222	223
224	225	226	227
228	229	230	231
232	233	234	235
236	237	238	239
240	241	242	243
244	245	246	247
248	249	250	251
252	253	254	255
256	257	258	259
260	261	262	263
264	265	266	267
268	269	270	271
272	273	274	275
276	277	278	279
280	281	282	283
284	285	286	287
288	289	290	291
292	293	294	295
296	297	298	299
300	301	302	303
304	305	306	307
308	309	310	311
312	313	314	315
316	317	318	319
320	321	322	323
324	325	326	327
328	329	330	331
332	333	334	335
336	337	338	339
340	341	342	343
344	345	346	347
348	349	350	351
352	353	354	355
356	357	358	359
360	361	362	363
364	365	366	367
368	369	370	371
372	373	374	375
376	377	378	379
380	381	382	383
384	385	386	387
388	389	390	391
392	393	394	395
396	397	398	399
400	401	402	403
404	405	406	407
408	409	410	411
412	413	414	415
416	417	418	419
420	421	422	423
424	425	426	427
428	429	430	431
432	433	434	435
436	437	438	439
440	441	442	443
444	445	446	447
448	449	450	451
452	453	454	455
456	457	458	459
460	461	462	463
464	465	466	467
468	469	470	471
472	473	474	475
476	477	478	479
480	481	482	483
484	485	486	487
488	489	490	491
492	493	494	495
496	497	498	499
500	501	502	503
504	505	506	507
508	509	510	511
512	513	514	515
516	517	518	519
520	521	522	523
524	525	526	527
528	529	530	531
532	533	534	535
536	537	538	539
540	541	542	543
544	545	546	547
548	549	550	551
552	553	554	555
556	557	558	559
560	561	562	563
564	565	566	567
568	569	570	571
572	573	574	575
576	577	578	579
580	581	582	583
584	585	586	587
588	589	590	591
592	593	594	595
596	597	598	599
600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619
620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639
640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659
660	661	662	663
664	665	666	667
668	669	670	671
672	673	674	675
676	677	678	679
680	681	682	683
684	685	686	687
688	689	690	691
692	693	694	695
696	697	698	699
700	701	702	703
704	705	706	707
708	709	710	711
712	713	714	715
716	717	718	719
720	721	722	723
724	725	726	727
728	729	730	731
732	733	734	735
736	737	738	739
740	741	742	743
744	745	746	747
748	749	750	751
752	753	754	755
756	757	758	759
760	761	762	763
764	765	766	767
768	769	770	771
772	773	774	775
776	777	778	779
780	781	782	783
784	785	786	787
788	789	790	791
792	793	794	795
796	797	798	799
800	801	802	803
804	805	806	807
808	809	810	811
812	813	814	815
816	817	818	819
820	821	822	823
824	825	826	827
828	829	830	831
832	833	834	835
836	837	838	839
840	841	842	843
844	845	846	847
848	849	850	851
852	853	854	855
856	857	858	859
860	861	862	863
864	865	866	867
868	869	870	871
872	873	874	875
876	877	878	879
880	881	882	883
884	885	886	887
888	889	890	891
892	893	894	895
896	897	898	899
900	901	902	903
904	905	906	907
908	909	910	911
912	913	914	915
916	917	918	919
920	921	922	923
924	925	926	927
928	929	930	931
932	933	934	935
936	937	938	939
940	941	942	943
944	945	946	947
948	949	950	951
952	953	954	955
956	957	958	959
960	961	962	963
964	965	966	967
968	969	970	971
972	973	974	975
976	977	978	979
980	981	982	983
984	985	986	987
988	989	990	991
992	993	994	995
996	997	998	999
1000	1001	1002	1003
1004	1005	1006	1007
1008	1009	1010	1011
1012	1013	1014	1015
1016	1017	1018	1019
1020	1021	1022	1023
1024	1025	1026	1027
1028	1029	1030	1031
1032	1033	1034	1035
1036	1037	1038	1039
1040	1041	1042	1043
1044	1045	1046	1047
1048	1049	1050	1051
1052	1053	1054	1055
1056	1057	1058	1059
1060	1061	1062	1063
1064	1065	1066	1067
1068	1069	1070	1071
1072	1073	1074	1075
1076	1077	1078	1079
1080	1081	1082	1083
1084	1085	1086	1087
1088	1089	1090	1091
1092	1093	1094	1095
1096	1097	1098	1099
1100	1101	1102	1103
1104	1105	1106	1107
1108	1109	1110	1111
1112	1113	1114	1115
1116	1117	1118	1119
1120	1121	1122	1123
1124	1125	1126	1127
1128	1129	1130	1131
1132	1133	1134	1135
1136	1137	1138	1139
1140	1141	1142	1143
1144	1145	1146	1147
1148	1149	1150	1151
1152	1153	1154	1155
1156	1157	1158	1159
1160	1161	1162	1163
1164	1165	1166	1167
1168	1169	1170	1171
1172	1173	1174	1175
1176	1177	1178	1179
1180	1181	1182	1183
1184	1185	1186	1187
1188	1189	1190	1191
1192	1193	1194	1195
1196	1197	1198	1199
1200	1201	1202	1203
1204	1205	1206	1207
1208	1209	1210	1211
1212	1213	1214	1215
1216	1217	1218	1219
1220	1221	1222	1223
1224	1225	1226	1227
1228	1229	1230	1231
1232	1233	1234	1235
1236	1237	1238	1239
1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259
1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279
1280	1281	1282	1283
1284	1285	1286	1287
1288	1289	1290	1291
1292	1293	1294	1295
1296	1297	1298	1299
1300	1301	1302	1303
1304	1305	1306	1307
1308	1309	1310	1311
1312	1313	1314	1315
1316	1317	1318	1319
1320	1321	1322	1323
1324	1325	1326	1327
1328	1329	1330	1331
1332	1		

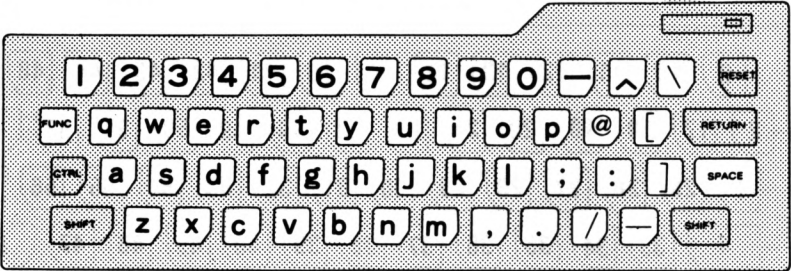
# KEYBOARD KEY CODES

## Key Codes

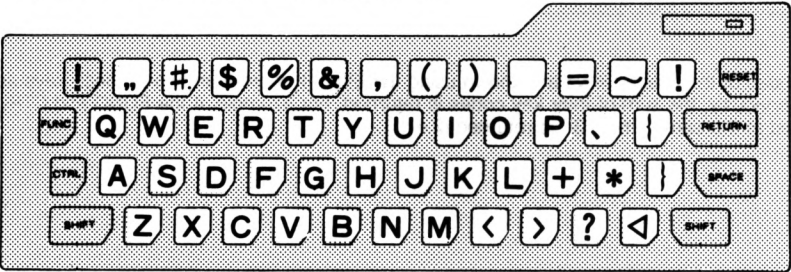


The codes of the shaded keys are known using INKEY(1). If more than two of these keys are pressed simultaneously, the sum of their key codes is returned. For example, if the FUNC and CTRL keys are pressed at the same time, "3" is returned (2 + 1).

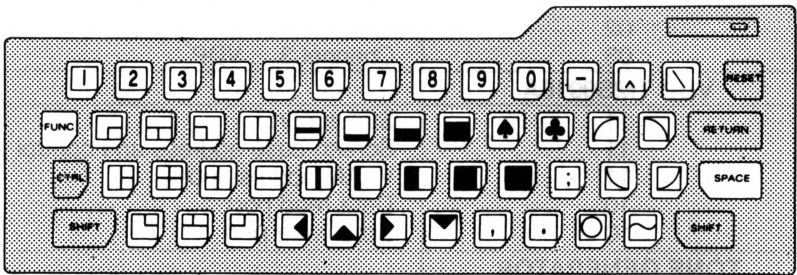
Alphabetic characters with SHIFT key not pressed



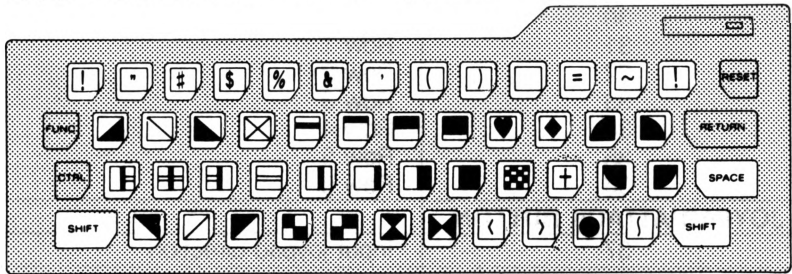
Alphabetic characters with SHIFT key pressed



Graphic characters available with SHIFT key not pressed



Graphic characters available with SHIFT key pressed



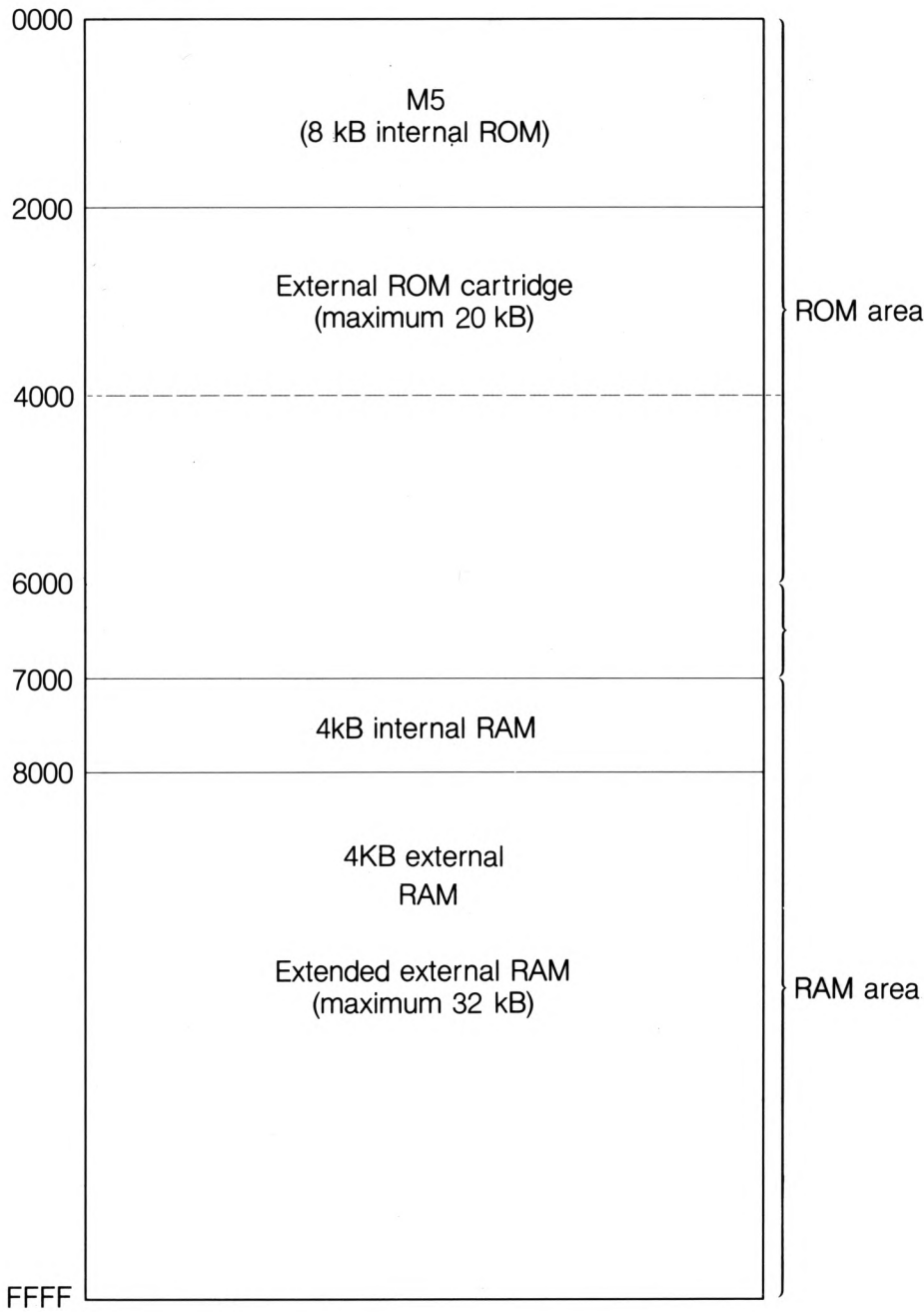
# POUT I/O TABLE

POUT Number	Summary
<b>Z80 CTC</b>	
01	Channel # 1 ... peripheral timer
02	Channel # 2 ... I/O clock
03	Channel 3 ... VDP
<b>VDP TMS9918A</b>	
11	Status port
11	Screen base address and control port
11	VRAM address port
10	Data read port
10	Data write port
<b>tone generator</b>	
20	Tone generator control
<b>KEYBOARD</b>	
30	Row 0
31	Row 1
32	Row 2
33	Row 3
34	Row 4
35	Row 5
36	Row 6
<b>JOYPAD/ATTACK BUTTON</b>	
37	Input joystick direction
<b>RESET/HALT KEY</b>	
50	Reset/Halt key data port (bit 7)
<b>CASSETTE RECORDER</b>	
50	Output port
50	Input port
50	Output port
<b>PERIPHERAL I/O</b>	
40	Data output
50	Strobe output
50	Printer busy

# MEMORY MAP

## Memory map

Online main memory



**VRAM (video RAM) memory map****Layout I**

0000	Free area
2000	Sprite pattern code table (2048 bytes—both screen buffers 0 and 1)
2800	Character pattern code table (2048 bytes—screen buffer 0)
3000	Character pattern code table (2048 bytes—screen buffer 1)
3800	Character to ASCII relationship table (768 bytes— screen buffer 0)
3B00	Sprite attribute table (128 bytes—screen buffer 0)
3B80	Character color table (32 bytes—screen buffer 0)
3C00	Character to ASCII relationship table (768 bytes— screen buffer 1)
3F00	Sprite attribute table (128 bytes—screen buffer 1)
3F80	Character color table (32 bytes—screen buffer 1)
4000	

Layout 1 remarks—applicable when the 8 expanded screens are used (uses addresses &0000 to &1FFF)

Layout II

0000	GII mode color table (6 kB)	
1800	Sprite pattern code table (2 kB)	
2000	GII mode pattern code table (6 kB)	
3800	Pattern code table (screen buffer 0)	
3B00	Sprite attribute table	
3B80	Character color table	*
3C00	Pattern code table (screen buffer 1)	
3F00	Sprite attribute table	
3F80	Character color table	*
4000		

Note: \* signifies color table in other than GII mode



## ERROR CODES

<b>ERROR CODE</b>	<b>ERROR SUMMARY</b>	<b>REASON</b>
<b>ERR 1 ERRNF</b>	FOR .. NEXT error	<ul style="list-style-type: none"> <li>• FOR ~ NEXT does not correspond</li> </ul>
<b>ERR 2 ERRSY</b>	Syntax error	<ul style="list-style-type: none"> <li>• Non-existent command</li> </ul>
<b>ERR 3 ERRRG</b>	Subroutine error	<ul style="list-style-type: none"> <li>• CLEAR used in the subroutine</li> <li>• Jumped to subroutine using a GOTO</li> <li>• GOSUB ~ RETURN does not correspond</li> </ul>
<b>ERR 4 ERROD</b>	READ error in DATA statement	<ul style="list-style-type: none"> <li>• Insufficient data</li> <li>• Missing DATA statement</li> </ul>
<b>ERR 5 ERRIF</b>	Variable type mismatch	<ul style="list-style-type: none"> <li>• Wrong type of value given for statement variable</li> </ul>
<b>ERR 6 ERROV</b>	Overflow	<ul style="list-style-type: none"> <li>• When multiplying — answer is correct but beyond the negative limit.</li> </ul>
<b>ERR 7 ERROM</b>	Memory exhausted	<ul style="list-style-type: none"> <li>• Program is too long</li> <li>• Too many variables (reduce the number)</li> <li>• Too many subroutines (reduce the number)</li> </ul>
<b>ERR 8 ERRUL</b>	Missing line number	<ul style="list-style-type: none"> <li>• Missing destination for GOTO or GOSUB</li> </ul>
<b>ERR 9 ERRBS</b>	Array variable error	<ul style="list-style-type: none"> <li>• Error in array statement</li> <li>• Letter accompanying array variable outside scope of the statement</li> </ul>
<b>ERR 10 ERRDD</b>	Array variable error	<ul style="list-style-type: none"> <li>• Same variable set twice</li> </ul>
<b>ERR 11 ERRDZ</b>	Division by 0	<ul style="list-style-type: none"> <li>• Divided by zero</li> </ul>
<b>ERR 12 ERRID</b>	Inappropriate direct execution statement	<ul style="list-style-type: none"> <li>• Wrong direct execution command (execute wrong program)</li> </ul>
<b>ERR 13 ERRTM</b>	Inappropriate data item	<ul style="list-style-type: none"> <li>• Characters were provided when numerics were expected, or vice versa</li> </ul>
<b>ERR 14 ERROS</b>	Stack overflow	<ul style="list-style-type: none"> <li>• Stack space exhausted</li> <li>• No stack area left for the PAINT statement</li> </ul>

<b>ERROR CODE</b>	<b>ERROR SUMMARY</b>	<b>REASON</b>
<b>ERR 15 ERRST</b>	Character string length error	<ul style="list-style-type: none"> <li>• Character string too long or becomes too long during calculation</li> <li>• Substituted a character string larger than the left hand variable</li> </ul>
<b>ERR 16 ERRUD</b>	Array variable error	<ul style="list-style-type: none"> <li>• Used an array variable which has not yet been allocated</li> </ul>
<b>ERR 17 ERRDL</b>	Redundant label	<ul style="list-style-type: none"> <li>• Used the same label more than once</li> </ul>
<b>ERR 18 ERRTR</b>	Tape read error	<ul style="list-style-type: none"> <li>• Tape read error</li> <li>• Reset during tape read operation</li> </ul>
<b>ERR 19 ERRDM</b>	Wrong screen display mode	<ul style="list-style-type: none"> <li>• Wrong screen mode chosen from GI, GII, text or multi-color.</li> </ul>
<b>ERR 20 ERRSP</b>	Sprite error	<ul style="list-style-type: none"> <li>• Tried to see or move a sprite that has been erased</li> </ul>
<b>ERR 21 ERRNS</b>	Stack error	<ul style="list-style-type: none"> <li>• Does not occur.</li> </ul>
<b>ERR 22 ERRUR</b>	REPEAT..UNTIL error	<ul style="list-style-type: none"> <li>• REPEAT..UNTIL does not match</li> </ul>
<b>ERR 23 ERRTO</b>	Timeout error	<ul style="list-style-type: none"> <li>• INPUT statement timed out</li> <li>• Response not received from floppy disk in time.</li> </ul>
<b>ERR 24 ERRRE</b>	RESUM error	<ul style="list-style-type: none"> <li>• Executed RESUM when no error occurred</li> </ul>
<b>ERR 25 ERRDF</b>	INPUT error	<ul style="list-style-type: none"> <li>• Pressed RETURN key without keying in any data</li> </ul>

<b>ERROR CODE</b>	<b>ERROR SUMMARY</b>	<b>REASON</b>
<b>Error 30</b>	SIO communications error	<ul style="list-style-type: none"> <li>• Wrong input data</li> </ul>
<b>Error 100</b>	Channel unavailable	<ul style="list-style-type: none"> <li>• All channels in use—appears when OLD, VERIFY, SAVE, or LIST are attempted</li> </ul>
<b>Error 101</b>	Specified channel not open	<ul style="list-style-type: none"> <li>• Channel may not be OPENed for data input</li> </ul>
<b>Error 102</b>	Channel already in use	<ul style="list-style-type: none"> <li>• You have tried to allocate more than one I/O device to a channel</li> </ul>
<b>Error 103</b>	Specified device already in use	<ul style="list-style-type: none"> <li>• You have tried to use multiple channels with the ACMT</li> </ul>
<b>Error 104</b>	Improper file name	<ul style="list-style-type: none"> <li>• Non-existent device</li> <li>• File name too long</li> </ul>
<b>Error 105</b>	Improper access	<ul style="list-style-type: none"> <li>• PUT or GET used with a device which cannot PUT or GET</li> <li>• You have randomly accessed a device which only handles sequential access</li> </ul>
<b>Error 106</b>	Wrong file	<ul style="list-style-type: none"> <li>• A file name already used cannot be reused on the same disk</li> </ul>
<b>Error 107</b>	Communications process error	<ul style="list-style-type: none"> <li>• Information transfer to disk not proceeding correctly</li> </ul>
<b>Error 131</b>	Improper drive number	<ul style="list-style-type: none"> <li>• The FD-5's drives are numbered 0 and 1</li> </ul>
<b>Error 132</b>	Incorrect file name	<ul style="list-style-type: none"> <li>• Zero-character file names are not permitted with the FD-5</li> </ul>
<b>Error 151</b>	Exceeded record	<ul style="list-style-type: none"> <li>• You have tried to read or write across more than one record</li> </ul>
<b>Error 152</b>	Data finished	<ul style="list-style-type: none"> <li>• You have attempted to read out data which has not been input</li> </ul>
<b>Error 154</b>	No space on the disk	<ul style="list-style-type: none"> <li>• There is no empty field on the disk; file may not be expanded. Unnecessary files may be erased with KILL</li> </ul>
<b>Error 155</b>	Too many files	<ul style="list-style-type: none"> <li>• The disk file capacity has been exceeded. Capacity is 108 files</li> </ul>
<b>Error 156</b>	Record not finished	<ul style="list-style-type: none"> <li>• You have randomly accessed a record before access to the previous record has been completed</li> </ul>

<b>ERROR CODE</b>	<b>ERROR SUMMARY</b>	<b>REASON</b>
<b>Error 160</b>	No empty channel	<ul style="list-style-type: none"> <li>• No more than four files may be simultaneously OPENed with the FD-5</li> </ul>
<b>Error 170</b>	No relevant file	<ul style="list-style-type: none"> <li>• The specified file is not on the disk</li> </ul>
<b>Error 171</b>	File already in use	<ul style="list-style-type: none"> <li>• Specified file is in use on another channel</li> </ul>
<b>Error 172</b>	File already exists	<ul style="list-style-type: none"> <li>• The file is already on the disk</li> </ul>
<b>Error 180</b>	Readout prohibited	<ul style="list-style-type: none"> <li>• You have tried to read a protected file</li> </ul>
<b>Error 181</b>	Entry prohibited	<ul style="list-style-type: none"> <li>• You have tried to enter to a protected file</li> </ul>
<b>Error 182</b>	Erase prohibited	<ul style="list-style-type: none"> <li>• You have tried to erase a protected file</li> </ul>
<b>Error 190</b>	Wrong disk	<ul style="list-style-type: none"> <li>• The file has been transferred to another disk</li> </ul>
<b>Error 191</b>	Panic	<ul style="list-style-type: none"> <li>• Unexpected fault during normal operations. Transfer from the disk has possibly gone haywire</li> </ul>
<b>Error 214</b>	Abnormal sounds	<ul style="list-style-type: none"> <li>• Mechanical problem with reading to or writing from the disk</li> </ul>

Command or Function	Definition	Abvr.	Page
\$	Remark statement or label name .....		17
ABS (F)	Returns absolute value of X	a. ....	103
ASCII (F)	Returns ASCII code for first character of a string	as. ....	104
ATN (F)	Returns arc tangent of X	at. ....	105
AUTO	Automatic line numbering	a. ....	18
BCOL	Sets screen background color	b. ....	19
CALC (F)	Performs BASIC-F operations on an expression given as a string	ca. ....	106
CALL	Transfers program control to a specific machine address	ca. ....	20
CDBL (F)	Converts an integer to a real number	cd. ....	107
CHAIN	Retrieves program from tape or disk and executes	ch. ....	21
CHR\$ (F)	Returns character with internal code X	ch. ....	108
CINT (F)	Converts a real number to an integer	ci. ....	109
CLEAR	Clears section of memory for PAINT/character buffer	cle. ....	22
CLIST	Lists in upper case letters	cli. ....	23
CLOSE	Ends file usage	cl. ....	24
CLS	Clears screen .....		25
COLOR	Sets up character color (GI and GII modes)	col. ....	26
CONSOLE	Enables/disables keyboard function keys	cons. ....	27
CONT	Restarts program after STOP or keyboard interrupt	c. ....	28
COS (F)	Returns cosine of X	co. ....	110
CURSOR (F)	Moves cursor to specified co-ordinates	c. ....	29
DATA	Stores constant information used by program and accessed by READ	d. ....	30
DEL	Deletes lines from current program in memory	de. ....	31
DIM	Allocates memory for an array	di. ....	32
DRAW	Draws a line on the screen	dr. ....	33
END	Indicates the end of a program and halts execution	e. ....	34
ERR (F)	Returns error code of the most recent error .....		111
ERRL (F)	Returns the line number of the most recent error .....		112
ERRL\$ (F)	Returns line label of most recent error .....		113
EVENT	Sets event timer interrupt interval	ev. ....	35
EVENT ON/OFF	Enables/disables event timer interrupt .....		36
EXE (F)	Executes a BASIC-F statement having string representation	ex. ....	114
EXP (F)	Calculates the function e*	ex. ....	115
FCOL	Sets character color or graphics display color	fc. ....	37
FIX (F)	Returns integer portion of X	fi. ....	116
FOR..TO.. [STEP]	Performs many iterations of a section of the program	f. ~ to ~ s. ....	38
FRE (F)	Returns information on memory usage	fr. ....	117
GCOPY	Prints the current screen image on the printer	gc. ....	39
GET	Reads data from channel to variable	ge. ....	40
GINIT	Enters graphic mode	gi. ....	41
GMODE	Sets up graphics display mode	gmod. ....	42
GMOVE	Moves graphics cursor	gm. ....	44
GOSUB	Transfers control to a subroutine	gis. ....	45
GOTO	Transfers program control to line number/label in statement	g. ....	46
HEX (F)	Returns hexadecimal equivalent of X	h. ....	118
IF..THEN..ELSE	Evaluates the conditional expression	if ~ t. ~ e. ....	47
INKEY\$	Returns current character from keyboard	ink. ....	119
INP (F)	Inputs a byte from a port .....		120
INPUT	Assigns alphanumeric data from keyboard to variables	i. ....	48
INSTR (F)	Searches string STR\$ for first occurrence of string SUB\$	ins. ....	121
INT (F)	Returns integer portion of a variable	i. ....	122
KILL	Deletes files	k. ....	49
LEFT\$ (F)	Returns left substring of X\$	lef. ....	123
LEN	Resets the maximum length of string variables	le. ....	50
LEN (F)	Returns length of character string	l. ....	124
LET	Assigns the result of an expression to a variable .....		51
LIST	Lists a file to another file, the printer or screen	l. ....	52
LN (F)	Returns the natural log of X .....		125
LOC	Moves sprite-number to specified GR co-ordinates .....		53
LOCATE	Moves cursor to specified line and column	lo. ....	54
LOG (F)	Returns log X to the base 10	lo. ....	126
MAG	Changes sprite size and format	m. ....	55
MID\$ (F)	Returns substring of X\$ between X and Y	m. ....	127
NEW	Clears the current program and memory contents .....		56

Command or Function	Definition	Abvr.	Page
NEXT	Ends repeated executions initiated by FOR within a program	n.....	57
NUM\$ (F)	Converts numeric value of X to character equivalent	nu.....	128
OLD	Reads file from external memory	o.....	58
ON ERROR GOSUB..	Transfers control to line number on error detection.....		59
ON EVENT GOSUB..	Calls subroutine when event timer interrupt occurs.....		60
ON..GOSUB..	Evaluates expression and branches to nth line number.....		61
ON..GOTO..	Evaluates expression and branches to nth line number.....		62
ON..RESTORE..	Sets data pointer to data group depending on expression.....		63
OPEN	Opens user files	op.....	64
OUT	Sends data byte by byte to a port	ou.....	129
PAINT	Paints area indicated by GR co-ordinates	pa.....	65
PEEK (F)	Returns contents of memory address X.....		130
PEEKW (F)	Returns 16 bits from CPU memory address	pe.....	131
PI (F)	Returns the value of Pi.....		132
PLOT	Displays dot at specified GR coordinates	pl.....	66
POKE	Writes data directly into specified memory locations.....		67
POKEW	Writes data to CPU memory	po.....	68
PRINT	Puts text in screen display buffers	p.....	69
PUT	Assigns binary form to a variable and executes it	pu.....	70
RANDOMIZE	Resets the seed for the random number generator	ra.....	71
RDST\$ (F)	Reads statement indicated by cursor	rd.....	133
READ	Loads data from DATA statements into variables	rea.....	72
RECORD	Record to be next assigned	rec.....	73
REG (F)	Returns register value after CALL is executed.....		134
REM	Stores programmer comments within the program.....		74
RENUM	Changes the line numbering of the program	ren.....	75
REPEAT	Sets up a loop ending in a logical test	rep.....	76
RESTORE	Resets data pointer for data item groups in DATA statements	res.....	77
RESUME	Bypasses an error	resu.....	78
RETURN	Returns program control after GOSUB has been called	re.....	79
RIGHT\$ (F)	Returns substring from right side of X\$	ri.....	135
RND (F)	Returns a random number between 0 and X	rn.....	136
RPT\$ (F)	Returns a string of repetitions of X\$	rp.....	137
RUN	Executes the current program	r.....	80
SAVE	Writes to external memory	sa.....	81
SCOD	Assigns numeric code to sprite-number	sc.....	82
SCOL	Colors sprite-number.....		83
SG	Activates tone and noise generators.....		84
SGN (F)	Returns sign of X	sp.....	138
SIN (F)	Returns the sine of X.....		139
SLEEP	Stops execution for specified sleep time	sl.....	86
SQR (F)	Returns square root of X	sq.....	140
STCHR	Assigns pattern-code to character-code	stc.....	87
STEP	Stops/resumes execution at statement change	ste.....	88
STOP	Halts execution of a program from within a program	s.....	89
SWAP	Transfers contents of variables	sw.....	90
TAB	Tab over X characters	ta.....	91
TAN (F)	Returns tangent of X.....		141
TAPE	Accesses assembler supplied on external memory	ta.....	92
THETA	Sets mode for trigonometric functions to degrees/radians	th.....	93
TIME (F)	Returns time since powering up in seconds	ti.....	142
TRACE	Displays trace of executed line numbers during program execution	t.....	94
TYPE	Defines the type of a variable	ty.....	95
UNTIL	Marks the end of a repeat loop	u.....	96
VAL (F)	Converts a character string to its numeric equivalent	v.....	143
VARPTR (F)	Returns the actual memory address of a variable	var.....	144
VPEEK (F)	Returns contents of video memory at X	vp.....	145
VERIFY	Compares programs in memory	v.....	97
VIEW	Creates a viewport on the display screen	vi.....	98
VPOKE	Outputs data to the video memory	vp.....	99
VSAVE	Writes video RAM data to tape	vs.....	100
WAIT	Limits time computer will wait for input from keyboard	w.....	101
XCHG (F)	Swaps order of upper and lower bytes of X	xc.....	146

**SORD**  
**SORD COMPUTER CORPORATION**  
SAITO BLDG. 2F, 14-6, KYOBASHI 3-CHOME,  
CHUO-KU, TOKYO 104, JAPAN  
PHONE: (03) 562-6061  
TELEX: 2522745 (SORD J)