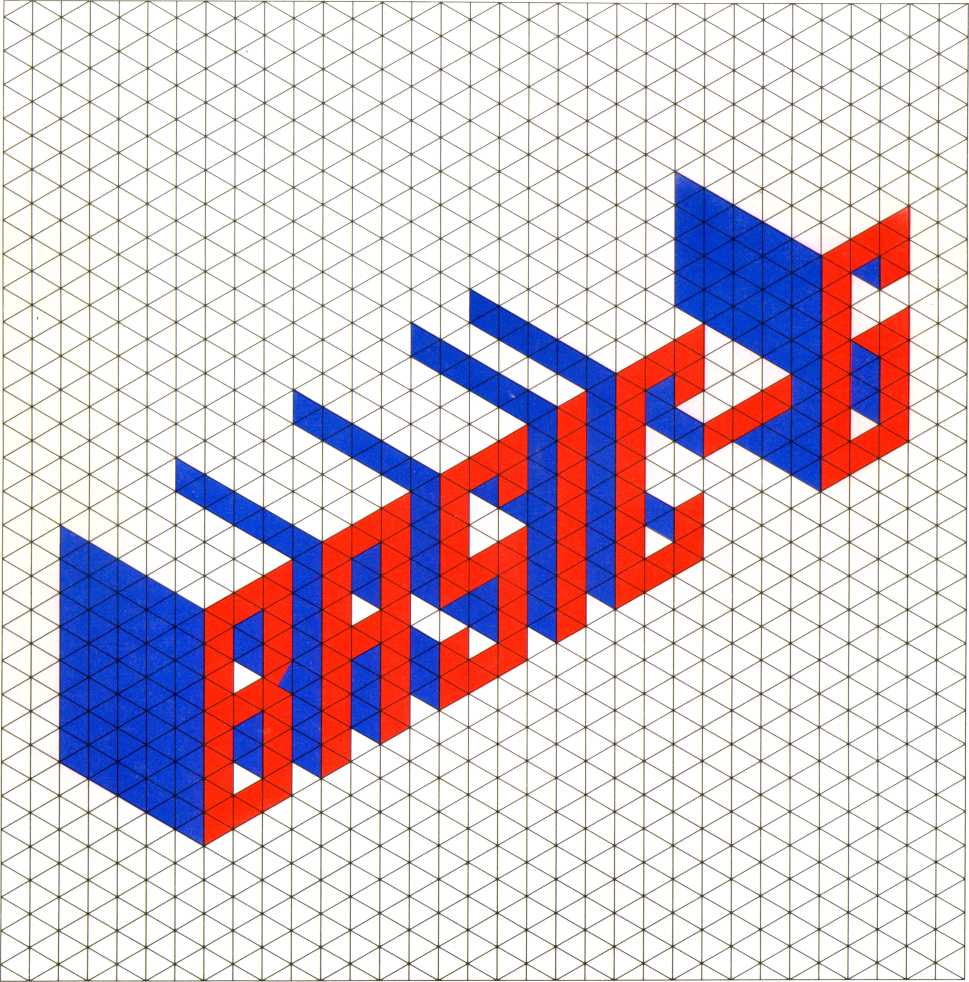
L



Easy BASIC for Games

**INDEX**

[Introduction 5](#bookmark0)

[GETTING STARTED 7](#bookmark1)

**CHAPTER 1 CHAPTER 2**

[WHAT IS BASIC-G? 11](#bookmark2)

1. [What Can BASIC-G Really Do? 12](#bookmark3)
2. [BASIC-G and BASIC-1 13](#bookmark4)
3. [The Keyboard 16](#bookmark5)
4. Displaying Characters on the Screen ...17
5. [Scrolling Function 18](#bookmark7)
6. [Repeat Function 19](#bookmark8)
7. [Using the CTRL Key 19](#bookmark9)
8. [Cursor Control Functions 20](#bookmark10)
9. [CTRL J—Cursor Down Function 21](#bookmark11)
10. CTRL E, F, C, D—Screen Movement

Functions 21

1. [CTRL DEL—Delete Function 22](#bookmark13)
2. CTRL X—Delete Characters to

the Right of the Cursor 24

1. CTRL P and CTRL O—Insert Function .24
2. CTRL B—Cursor Moves to

Leftmost Character 25

1. [CTRL K—Cursor to Home Position 25](#bookmark17)
2. CTRL'N—Cursor to Beginning

of Next Line....... 26

1. [CTRL I—Tab Function 26](#bookmark19)
2. [Control Codes Using the CTRL Key 27](#bookmark20)
3. Using the FUNC Key ,..2iB
4. [Keyboard Modes 28](#bookmark22)
5. [Running Some Simple Programs 31](#bookmark25)
6. [Renumbering Line Numbers 33](#bookmark26)
7. [Constants and Variables 33](#bookmark27)
8. [Operators 36](#bookmark28)
9. [Labels 36](#bookmark29)
10. [Array Variables 37](#bookmark30)

TWO SCREENS AND

**CHAPTER 3**

FOUR SCREEN MODES 41

1. [CTRLU—Display Screen 0 42](#bookmark32)
2. [CTRL V—Display Alternate Screen 42](#bookmark33)
3. CTRL Y—Display Alternate Screen,

Write to Hidden Screen 43

1. [CTRL Z—Write to Hidden Screen 44](#bookmark35)
2. [Interaction 44](#bookmark36)
3. [Four BASIC-G Screen Modes 45](#bookmark37)
4. [Gl—Graphics Mode 1 45](#bookmark38)
5. [Gil—Graphics Mode 2 45](#bookmark39)
6. [Multi-color Mode 46](#bookmark40)
7. [Text Mode 46](#bookmark41)
8. [Screen Modes and Control Codes 46](#bookmark42)
9. [Screens and Their Modes 47](#bookmark43)
10. Characters, Pixels, and Screen

Coordinates 47

1. [Dividing the Screen 48](#bookmark45)
2. [Game Planes 50](#bookmark46)

[FILES ON CASSETTE TAPE 53](#bookmark48)

**CHAPTER 4**

1. Using a Remote Function Recorder—

SAVE, VERIFY Commands 53

1. Using a Recorder Without the Remote Function—SAVE, VERIFY

Commands 55

1. Retrieving a File Saved On Tape—

OLD, CHAIN Commands 57

1. [Using the LIST Command 57](#bookmark53)
2. [Using the SKIP Command 58](#bookmark52)
3. Saving and Retrieving of the Screen—

VSAVE Command 59

[WHAT IS A CHARACTER? 61](#bookmark55)

**CHAPTER 5 CHAPTER 6**

1. [Different Numbering Systems 61](#bookmark56)
2. Finding the Pattern Code (Dot Matrix)...63

[CHARACTERS IN THE Gl MODE 67](#bookmark58)

1. Setting the Pattern Code (Dot Matrix) ...67
2. [Deleting a Pattern Code 68](#bookmark60)
3. [Coloring a Character 68](#bookmark61)
4. [Changing the Screen Color 70](#bookmark62)
5. [Changing the Backlight Plane Color 70](#bookmark63)
6. [Moving the UFO Character 71](#bookmark64)
7. Moving the UFO Using Expanded

Screen Buffers 72

CHARACTERS IN THE Gil MODE 77

**CHAPTER 7 CHAPTER 8**

* 1. [Gil Mode Screen Characteristics 77](#bookmark67)
  2. Setting the Pattern Code (Dot Matrix) ...77
  3. [Coloring a Character 79](#bookmark69)

CHARACTERS IN THE TEXT MODE 83

1. Setting the Pattern Code (Dot Matrix) ...83
2. [Coloring a Character 84](#bookmark72)
3. [Coloring the Backlight Plane 84](#bookmark73)

[MULTI COLOR MODE 87](#bookmark74)

[WHAT IS A SPRITE 89](#bookmark75)

**CHAPTER 9 CHAPTER 10**

1. [Sprites and Viewports 90](#bookmark76)
2. [Hierarchy of Sprites 90](#bookmark77)
3. [Sprites and the Four Screen Modes 91](#bookmark78)
4. [Let’s Make and Use a Sprite 91](#bookmark79)
5. [Create and Number a Sprite 92](#bookmark80)
6. [Pixel Coordinates 93](#bookmark81)
7. [Coloring Sprites 95](#bookmark82)
8. [Erasing Sprites 95](#bookmark83)

[10.9 UFO Sprite 96](#bookmark84)

1. [Increasing the Size of a Sprite 97](#bookmark85)
2. [Moving Sprites Around 99](#bookmark87)
3. [Sprite Coordinates 104](#bookmark88)
4. Creating a Jumbo Size Sprite

Using a Pattern Code 105

1. Creating a Jumbo Sprite Using the

JOI NT Statement 109

1. Detaching a Jumbo Sprite Joined

with the JOINT Statement 111

[WHAT IS GRAPHICS? 117](#bookmark92)

**CHAPTER 11**

1. [The Graphics Cursor 118](#bookmark93)

[11.2 Let’s Draw Something 119](#bookmark94)

1. [Using the PLOT Statement 120](#bookmark95)
2. [Using the Cl RCLE Statement 121](#bookmark96)
3. [Using the BOX Statement 125](#bookmark98)
4. [Using the BAR Statement 126](#bookmark97)
5. [Using the PAI NT Statement 127](#bookmark99)
6. [Coloring Graphics Pixels 128](#bookmark100)
7. [Coloring the Graphics Background 131](#bookmark101)
8. [Using Different Colors 132](#bookmark102)
9. [Dividing a Graphics Screen 133](#bookmark103)
10. [Characters on a Graphics Screen 134](#bookmark104)
11. [Chasing a Sprite 135](#bookmark105)
12. [Patterns 136](#bookmark106)
13. [GMODE Statement 137](#bookmark107)
14. [Multi-color Mode Graphics 143](#bookmark108)

[ANIMATION 147](#bookmark109)

**CHAPTER 12**

1. [Constructing the Environment 148](#bookmark110)
2. [Still Picture 148](#bookmark111)
3. [Animated Picture 154](#bookmark112)

COMPOSING YOUR OWN MUSIC .161

1. [Using the PLAY Statement 161](#bookmark115)

**CHAPTER 13**

1. [Sharps and Flats 162](#bookmark116)
2. [Octaves 163](#bookmark117)
3. [Duration of Music Notes 164](#bookmark118)
4. [Dotted Notes 165](#bookmark119)
5. [Sound Length Request —Triplets 166](#bookmark120)
6. [Music Rests 168](#bookmark121)
7. [Harmony 169](#bookmark122)
8. [Volume Level of a Music Note 169](#bookmark123)
9. [Tempo 170](#bookmark124)
10. [Sound Alteration 171](#bookmark125)
11. [Staccato 172](#bookmark126)
12. Using Numbers to Denote

Music Notes' 174

1. [Altering the Pitch 176](#bookmark128)
2. [Default Note Length 177](#bookmark129)
3. [Getting Ready for a Performance 177](#bookmark130)

[SOUND EFFECTS 181](#bookmark131)

**CHAPTER 14**

1. [Precision Tone Generator 181](#bookmark132)
2. [Noise Generator 183](#bookmark133)
3. [Live Sound Effects 184](#bookmark134)

[INTERRUPT MANAGEMENT 189](#bookmark135)

**CHAPTER 15**

1. [Practical Interrupt Management 191](#bookmark136)
2. [ON EVENT GOSUB Interrupts 191](#bookmark137)
3. [ON ALARM GOSUB Interrupts 193](#bookmark138)
4. [ON COINC GOSUB Interrupts 195](#bookmark139)
5. ON KEY'GOSUB and ON

JOY GOSUB Interrupts 196

1. [ON ERROR GOSUB Interrupts 198](#bookmark141)

[CONDITIONAL STATEMENTS 201](#bookmark142)

**CHAPTER 16**

1. [Conditional Jump 201](#bookmark143)
2. [Conditional Call 201](#bookmark144)
3. [Conditional RESTORE 203](#bookmark145)
4. Hints When Using Conditional

Statements 203

[ACCURATE TIMER 205](#bookmark147)

**CHAPTER 17**

**CHAPTER 18 CHAPTER 19**

**CHAPTER 20 CHAPTER 21**

**APPENDIX**

1. [TIMES Statement 205](#bookmark148)
2. [WAIT Statement 206](#bookmark149)
3. [SLEEP Statement 207](#bookmark150)

[MORE PRINT FEATURES 209](#bookmark152)

1. [MPRINT, MRCRTS Statements 209](#bookmark153)

[MORE SPRITE FEATURES 213](#bookmark154)

1. [DIST Statement 213](#bookmark155)
2. [DRCT Statement 215](#bookmark156)

KEYBOARD FEATURES 219

1. IN KEY Statement 219

[PERIPHERALS 223](#bookmark158)

1. [The Printer 223](#bookmark159)
2. [Before Using a Printer 223](#bookmark160)
3. [Printing a Program Listing 225-](#bookmark161)
4. [Printing Directly to a Printer 227](#bookmark162)
5. [Printing Graphics Displays 228](#bookmark163)
6. [PMODE Statement 234](#bookmark164)
7. [P-EDIT 235](#bookmark165)

A—CODE INFORMATION

Character Codes 239

[Control Codes 242](#bookmark167)

[Color Codes 243](#bookmark168)

[B—BASIC-G STATEMENT TYPES 244](#bookmark169)

[C-SCREEN MODE SUMMARY 271](#bookmark170)

[D—CONSTRUCTING YOUR OWN CHARACTER OR SPRITE 272](#bookmark171)

[E-CRT LAYOUT COORDINATES 275](#bookmark172)

[F—KEYBOARD KEY CODES 277](#bookmark173)

G—OPERATORS 278

H—POUT I/O TABLE 279

I—MEMORY MAP 280

J—ERROR CODES 283

**CHAPTER 3 CHAPTER 4 CHAPTER 5 CHAPTER 6 CHAPTER 7 CHAPTER 8**

**CHAPTER 12**

CHAFT^^^

**16**

**CHAPTER 20**

INTRODUCTION

Welcome to BASIC-G, an enhancement of BASIC-1 (BASIC-Introduction), designed especially for creating action packed video games.

BASIC-G differs from BASIC-1 since it includes many powerful capabilities in addition to all BASIC-1 features. Being a subset of BASIC-G and upwardly compatible, BASIC-1 programs are easily executed on a BASIC-G system. (The inverse is not true, however—BASIC-G programs will not run on a BASIC-1 system.) The result is a language medium to create effective animation and interesting gameboard backgrounds for mind boggling games! Thinking about the sounds at your favorite arcade? Sophisticated music can be composed and played using BASIC-G, rivaling even the cream of the crop at the arcade! You can even sing along with your M5!

Take enough time to become familiar with BASIC-G. Go leisurely through this manual from the beginning. It’s written in everyday English. Read a bit, play with your M5, and read a little more. In no time at all you can be composing action packed games limited only by your imagination.

Look into BASIC-F if you’re interested in scientific applications, or perhaps FALC if you’re searching for an electronic spread sheet.

Before we start, one word of caution. Some elementary BASIC concepts lightly glossed in this manual are explained in the BASIC-1 manual. BASIC-1 may be a nice stepping stone to understanding BASIC-G.

CHAPTER

1. GETTING STARTED

OK. We’re ready to get started. Or are we? If your system is not already put together, look at the figure below. Be certain the connector plugs are inserted properly. Don’t force them, but they should be firmly inserted.

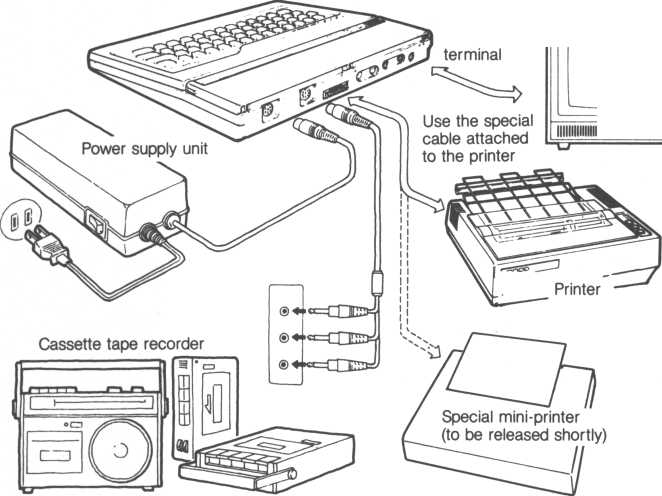
Your system consists of the M5 console, power supply unit, TV set, and optional cassette tape recorder and printer.

Connect to the antenna or VTR

Commercial TV set

000000000000 □ □

Putting together your M5 system



To avoid what is known as electrical noise, keep your cassette recorder away from the television and power supply unit (as well as other power sources).

Look at the type of cartridge inserted into your M5 console. You must have the BASIC-G program cartridge inserted into the console. To protect your M5, first turn off power to your M5. Be careful when inserting or pulling out a cartridge because the connector can be damaged. Be sure the cartridge label is facing you and push firmly. Initially, it will be a snug fit, don’t force it.

Label facing towards you

Indentation

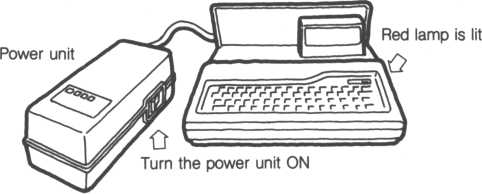


**CHAPTER**



1. WHAT IS BASIC-G?

After connecting the console to your television and inserting the BASIC-G program cartridge, turn the power unit ON.

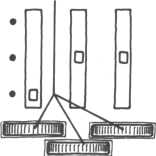


Turn on the attached TV set and adjust the channel selector until a “BASIC-G Ready” prompt is displayed. When the TV is connected at its VTR terminal, be sure to switch to "VTR.”

Turn the channel selector to the left or right. Fine tuning may be necessary.



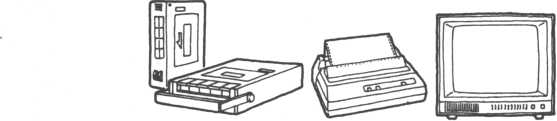
Select a channel button and adjust the tuning control to the left or right



For televisions with For televisions with channel selector knobs push button tuning

Turn on the other components of your system. The other parts, also known as peripherals, are the cassette recorder, printer and perhaps another monitor.

*ll*



Your M5 system is now ready to input and run programs. The “BASIC-G Ready” message will be displayed at the upper left hand corner of the screen. But if the system isn’t connected properly, it will not work. If the Ready message is not displayed, immediately turn off the power unit and check the connection between the television and M5 console. Also check that your BASIC-G cartridge is plugged in correctly.

**B** 3 :5 **1** C **- G** Ready

**in**

Basic-G Ready message

The Basic-G Ready message is your initial contact with the BASIC-G system. We’re finally ready. But before we get our feet wet, what can BASIC-G REALLY do?

* 1. What Can BASIC-G Really Do?

You’ve may have gone through the BASIC-1 manual where software for several introductory games was explored. BASIC-G runs with the ball much further to introduce techniques to create fascinating animated games.

• Drawing animated figures

Animated figures can be described by what are known as “sprites.” Sprites are figures created by us and moved freely against a background. The spacemen and UFOs you see at the arcade are also sprites. Out of a total 34 sprites, 32 are of the movable variety. And because sprites are sequentially numbered, it’s possible to create pseudo 3-dimensional effects, such as a UFO flying away.

* Music and sound effects

The M5 can play up to six octaves and three notes per chord using a statement named PLAY. Surprised? And not only can the M5 play music, it’s also possible to create impressive sound effects like wind, cannon shots, chirping birds.

* Games are easily designed and composed

An example. The distance between two sprites (which can be UFOs, race cars, or whatever) is easily calculated using the DIST statement. Their directions are also easily determined using the DRCT statement.

Players’ actions are quickly retrieved and processed. All are important when designing dynamic games. They’re easy to learn and use.

* 1. BASIC-G and BASIC-1

BASIC-G is an enhanced superset of BASIC-1. Some capabilities built into BASIC-G are not available under BASIC-1; other BASIC-1 functions have been extended. Some BASIC-1 statements we given new nuances to aid in readability, making it easier to correct erroneous programs. Let’s look at some highlights.

* BASIC-1 processed almost everything in upper case. The exceptions were character strings within double quotes, REMarks and data in DATA statements. BASIC-G allows lower and upper case in almost any situation.
* LIST command—list BASIC-G keywords in lower case, everything else is in upper case. LIST can also be used to save a file on cassette tape.
* LISTC command—similar in BASIC-1.
* ELIST and ELISTC commands—works similar to LIST and LISTC except it clears the screen before listing. [[1]](#footnote-1)

• LET is not automatically inserted into assignment statements. However, if you type them in, they will be retained. This saves a long program from being unnecessarily long (saves memory). The following programs function identically, but their syntax is different. Notice the LET is missing from the BASIC-G version.

10 A$="BASIC-G":B=?800 20 print A$> B

BASIC-G

30 end

BASIC-1 io LET A$= "BASIC-G" : LET D=9800 20 PRINT A$,B 30 END

• You can indent program lines for better readability.

The following example makes the second FOR-TO-NEXT loop more conspicuous.

10 c l 5

20 for A=0 to 4 30 for B-0 to 100 40 print A\* B 50 next B 60 next A 70 end

• Remark statements and labels (Slabel-name) are set off by being situated next to the line number. This makes it easier to see how your program works.

lOOrem adding two numbers 110 els

120 A= rnd(9) + 1s B = rnd(9) + 1 130 print A;" +"■B\*" = "•

140 i nput C

150 if C< >A+B then goto $BAD 160 pri nt "Ri ght answer!"

170 goto 110 180SBAD

190 print "Sorry. One more time. 200 goto 130 210 end

* Remarks can also be specified by inserting an exclamation mark instead of the REM keyword.

100! **adding** two numbers

is identical to line 100 in the previous program.

* Unlike BASIC-I, BASIC-G checks the type of data input for an INPUT statement. If an unexpected type of data is input, an error message is output and program execution stops. For example, when an INPUT statement expects a number, inputting a character string will prompt an error message.
* A “PRINT #2” statement in BASIC-I forced an automatic line feed. Using BASIC-G, it works similar to a PRINT to the screen and does not force a line feed.
* Various messages are displayed when retrieving a file from cassette tape.

SKIP indicates an unrequested file was found and skipped, reading of the tape continues

FIND indicates the desired file has been found and loaded into

• • memory. The periods denote the approximate length of the

file—one period every 256 bytes. The periods are also used for the OLD, VERIFY and SKIP commands. An example sequence is:

Ready

OLD"test3"

You may then see the following displayed on the screen (providing these files are actually stored on a cassette tape in the proper sequence).

SKIP"-test 1. BG"

SKIP"test2.BG"

SKIP"tests.BG"

testl and test2 are found and skipped while searching for test3. The two periods indicate test3 is between 256 and 512 bytes (1 period for each 256 bytes). ‘BG’ denotes a BASIC-G file; a summary of this field follows:

BASIC-I file BASIC-G file CPU memory data Screen data Video RAM data Listing file

Bl

BG

CM

SC

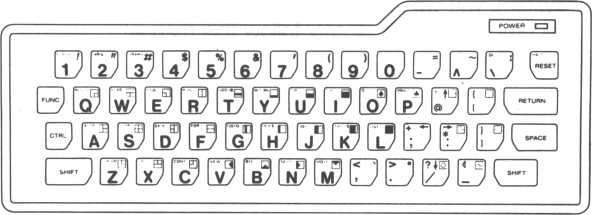
VM

LS

* 1. The Keyboard

Look at the M5 console. What you’ll notice right away is the typewriterlike keyboard. But don’t assume it’s exactly identical to a typewriter. There are several keys that make an M5 uniquely flexible and powerful.

Most keys on the keyboard either display a letter, number, character symbols, graphics symbols, functions, or a combination of these. We’ll go through and learn how all of these symbols and characters are used. We’ll take it nice and easy. Before you realize it, you’ll be composing your own custom made programs and taking advantage of the M5’s processing power.



M5 Keyboard

* 1. Displaying Characters on the Screen

What’s a cursor? Look on the screen and you’ll see the character ‘L’ blinking. That’s the cursor. It’s used to indicate the place on the screen where new characters are input.

Let’s try pressing some letter and number keys on the keyboard. Type in the following (the RETURN key is at the top right of the keyboard).

print 3+5 RETURN

You’ll then see the answer “8” immediately displayed. Pretty handy, right? Each time you press a key, the blinking cursor moves one position to the right and a “beep” is heard.

Try pressing a couple more letters and numbers. How about PRINT and another equation followed by the RETURN key? You’ll see all the keys you pressed sequentially displayed on the screen. The cursor stays ahead of your input by one position. It marks where your next input character will be displayed. Be careful when typing in your equation. If it’s too long, an error will be displayed.

Let’s learn more about M5 fundamentals, and then we’ll go onto composing a computer program. Don’t let any seemingly high powered words fool you. A computer program is only a group of many statements like the ones you just typed in.

The M5 system has many powerful functions. If you accidently get lost, turn the power OFF and then back ON. This resets the system and allows you to start over. This is not always good practice since the system forgets what it was previously doing. But when you become familiar with the system, you won’t have to reset the system this way. In the meantime, it’s recommended you closely follow this manual.

* 1. Scrolling Function

The numbers zero through nine are displayed on the uppermost line of the keyboard while the letters are arranged in the three lower rows, just like a typewriter. Press all of these keys. When enough keys have been pressed to fill up one line, the cursor automatically falls off the end of the current input line and ends up at the leftmost position of the next line. This happens for every new input line.

Obviously, since there are 24 input lines, the screen quickly runs out of space. In this case, the cursor never leaves the screen. Instead, the cursor falls off the end of the current input line as before. But the uppermost line moves up one line and disappears from the screen. You’ll see the cursor displayed at the leftmost position of the new input line which is now the bottom line of the screen. The same thing will occur every time a new line is input. This function is called “scrolling.”

Eventually, press the RETURN key. Don’t worry if an “Err 12 in 0” error message is displayed. In this exercise, we’re just exploring the system’s capabilities. The error message means the system only resets itself and is ready for your next input. It’ll be useful later.

The figure below shows a screen that has been completely filled with input. The cursor is sitting at the lower right hand corner of the screen.

**a a a a a a** b b b b b b

bbbbbbbbbbbbbbbbb

ddddddddddddddddd

**a a a a a a a** b b b b b b b

dddddd

|  |  |  |
| --- | --- | --- |
| 1 1 1 111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 rn rn rn rn rn m m rr» rn m m m m m m m m m m m m m m m m m m m m mm m  nrinnnrinnnnririnnnnnnnnnnnnnnnnnnnn | | |
| p p p P P p q q q q q q  r r r r r r s s s S S 2 t t t t t t  U U U U U L  V V V V k,  w u U u u u  XX X X X > | ppppppppppp. ppp qqqqqqqqqqqqqq rrrrrrrrrrrrrri  tttttttttttttt  JUUUUUUUUUUUUUU'  JWWWWWWW  JWWWWWUIWWWWWWWWI  cxxxxxxxxxxxxxx: | PPPPPKKKKrr-  qqqqqqqqqqq\*  r r r r r r r rr r r r  tttttttttttt  uuuuuuuuuuuu  UJUUUWWWWWUW w  XX X X X X x X x X XJ& |

Full screen of input

After inputting one more key, the topmost line scrolls off the screen and the cursor moves to the leftmost position of the next input line. The next input line then becomes the current input line.

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

dddddddddddddddddddddddddddddddd

eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee

99999999999999999999999999999999

hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii

jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj

kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk

11111111111111111111111111111111

nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn

PPPPPPPPPPPPPPPPPPPPtrrrr ■ \_

qqqqqqqqqqqqqqqqqqqqqqqqqqqqqq99

**rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr t t t t t t t t.t t t t t t t t t t t t t t t t t t t t t t \* J**

UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUW xx xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Automatic scrolling

* 1. Repeat Function

Next, try pressing the same key again and again. Then keep your finger pressed down on the same key. The effect is the same. In other words, constant pressure on a key is identical to repeatedly pressing it. It results in a constant stream of the same key displayed on the screen. It’ll stop once you take pressure off the key.

This function, for obvious reasons, is called the “repeat function.”

* 1. Using the CTRL Key

Let’s explore usage of the CTRL key situated at the left side of the keyboard. It’s one of several important keys that increase the power of your keyboard.

* 1. Cursor Control Functions

What do you do if you find a wrong character many positions back on the screen? How do we backtrack? Look to the right side of the keyboard. In the lower right corner of the keyboard, four keys known as “cursor control keys” are situated. Look for the keys marked —, —, t, and l. Try pressing the t key. (Don’t forget to keep the CTRL key pressed.) Looking at the cursor,, notice it’s moved one line up and remains on the same column. Next, try pressing the — key while maintaining pressure on the CTRL key. The cursor moves to the left of the same line, but the character that sits where the cursor was previously displayed is still there.

To sum up cursor control keys, they allow you to easily move the cursor in any direction while not changing any of the displayed information. However, the purpose of moving the cursor is usually to change character text.

Take your left hand off the CTRL key and press a letter key. You’ll see the letter displayed in place of the last displayed character. It’s easy to see how any character displayed on the screen can be quickly and easily corrected.

This is known as the “editing function.”

**0B00000B0D0DD 0**

**BBBBBBBOBBDBOQ**

**ED0BOG3BBBDDB5BGD**

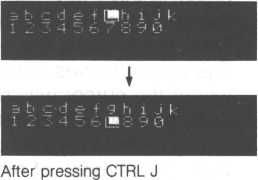
V "" J

Invoking the editing function with the CTRL and I keys

* 1. CTRL J—Cursor Down Function

Keeping the previous discussion of cursor control keys in mind, pressing the CTRL and J keys at the same time is identical to the CTRL and cursor down key combination.

Before pressing CTRL J



* 1. CTRL E, F, C, D—Screen Movement Functions

We’ve learned how to move the cursor within the displayed screen. How about moving the entire screen without moving the cursor? Easily done.

The key combinations below shift the screen one of four directions.

CTRL E—Shifts the screen up one line

CTRL F—Shifts the screen one character to the right

CTRL C—Shifts the screen down one line

CTRL D—Shifts the screen one character to the left

Type any information on the screen and place the cursor somewhere near the middle of the text (use the cursor control keys). Press the CTRL and F keys; notice the entire screen shifts one character to the right, the cursor remains in the same location. To shift it back, press the CTRL and D keys. The CTRL E and CTRL C key combinations are just as easy to use, try them both.

Before pressing CTRL F

be d e t 9 h i J k 2 3 4 56 711 6 0



After pressing CTRL F

* 1. CTRL DEL—Delete Function

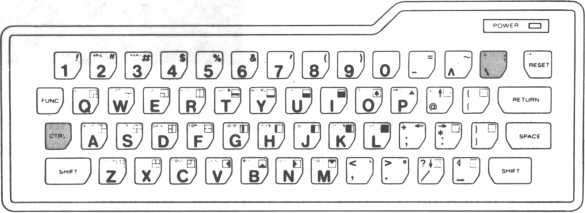
Up to this point, we’ve only discussed how to display letters and numbers on the screen. Now we’ll describe how to fix incorrect letters and numbers already displayed on the screen.

For this, we’ll conveniently use the “delete function.” Press the CTRL key at the lefthand side of the keyboard with your left hand. Keeping the CTRL key pressed, press the DEL key situated at the upper right side of the keyboard with your right hand. (If you have trouble finding the DEL key,

DEL is indicated at the upper left of the key in small capital letters.) Pressing this combination of keys at the same time results in the cursor stepping one position to the left and erasing the character previously displayed there. To delete several characters, press the DEL key for each character you want to delete.

Now stop pressing the CTRL and DEL keys and type in the correct information. Each time you input a new character, the cursor will then move to the right as before and the new character is displayed on the screen.

The CTRL H key combination works identically.



The delete function using the CTRL and DEL keys (or CTRL H key combination)

**abcdef90**



After deleting several characters

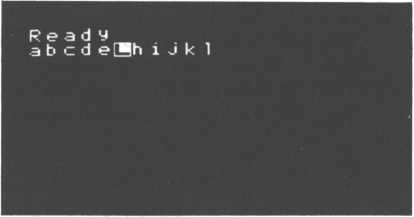
Before deletion from end of text

The delete function can also be used in the middle of a line of characters. In this case, the cursor remains where it is and all characters to the right move one position left. The text is shortened.

Ready

a **b c d e Hs h i** j k **1**

Before deletion from middle of text



Text closes up

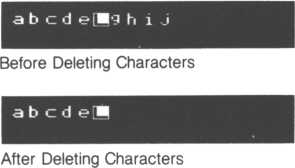
We’ve gone over the technique to delete one character in the middle of a line using the CTRL DEL key combination. You can also place the cursor at the beginning of the area you wish to delete and push the SPACE key as many times as desired. This, however, will leave spaces where the key was pressed. This latter method doesn’t really delete any characters, it actually substitutes the SPACE character for the old character. So what you see (or don’t see, depending on your perspective) is a blank character. If this isn’t what you want, use the CTRL DEL key combination. Notice you can also use the CTRL DEL keys to delete spaces since they’re treated identically to other visible characters.

|  |  |
| --- | --- |
| Ready |  |
| a b c d e | Hi J k 1 |

Using the SPACE key to delete characters

* 1. CTRL X—Delete Characters to the Right of the Cursor

You may sometimes want to delete many characters in one handy stroke. Move the cursor to a position anywhere on the input line. Keep the CTRL key pressed and press the X key at the same time. All characters to the right of the cursor are deleted. Likewise, a complete line can be deleted by positioning the cursor to the input line’s leftmost position and using the CTRL and X key combination.



* 1. CTRL P and CTRL O—Insert Function

You may have been frustrated if you deleted too many characters and had to type the rest of the line to restore it. The solution is the insert function. Use the cursor control keys and place the cursor in the middle of any line. Now press the CTRL and P keys simultaneously. Type several characters. Yes, the newly typed characters are inserted in your text. Call this the insert mode. Your keyboard will stay in this mode until a CTRL O key combination is pressed, which reverts your keyboard back to the normal mode. By the way, when power is first supplied to your keyboard, it’s initialized to the normal mode.

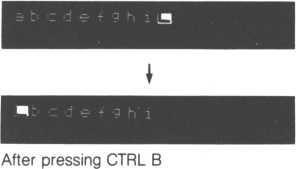


New characters are written over old characters.

* 1. CTRL B-—Cursor Moves to Leftmost Character

Move the cursor to the leftmost postion of the current input line. If done repeatedly, it can be pretty tedious, especially if the cursor is near the right margin. But there’s an easier way. For this exercise, move the cursor somewhere near the middle of the current input line. While keeping the CTRL key pressed, press the B key. The cursor is automatically moved to the leftmost position of that input line. You can now type in new characters for that line if you like.

Before pressing CTRL B



* 1. CTRL K—Cursor to Home Position

The upper left corner of the screen is also known as the “home” position. Similar to the CTRL and B key combination which moves the cursor to the leftmost position of the current input line, the CTRL and K key combination moves the cursor to the home position. Don’t forget to press both keys at the same time. Can’t think of any uses for this esoteric function? Don’t worry. The home position can be useful for interactive programs (interactive means programs that need user dialogue), it may become handy later.

[1



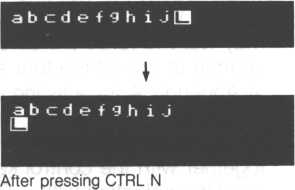
After pressing CTRL K

J

* 1. CTRL N—Cursor to Beginning of Next Line

Pressing the CTRL and N keys simultaneously is similar to pressing the RETURN key, the cursor moves to the beginning of the next line.

Before pressing CTRL N



* 1. CTRL **I—Tab** Function

Think about your typewriter “tab” function. The CTRL I key combination is similar, pressing both simultaneously will cause the cursor to tab 8 spaces. If any characters are in the path of the tab, they are deleted.

characters

Tabbing past the end of line causes the cursor to move to the next line

J

* 1. Control Codes Using the CTRL Key

Rather than let BASIC-G perform automatic cursor and screen movement, there may be occasions where you want to directly manipulate the screen or cursor, or perhaps access a special control function.

Obviously, the visible characters that we've been playing with will not do the job. We need what are known as "control codes.” For example, we may want to force the screen not to scroll when it would normally do so, or change to one of the four screen modes. There are many possibilities. Look at Appendix A for a listing of control codes and their corresponding functions.

To use control codes, the CTRL and SHIFT keys need to be pressed together with the control key to denote the control code. Control codes are used in conjuction with a PRINT statement.

double quote marks



print"G"

^ Press the CTRL, SHIFT and G keys

simultaneously. The key denoting the control code will be displayed within the double quote marks, a G in this case.

Control codes can also be combined. If control codes designated by the R and L keys are used as below, the screen will change to the Gil mode— discussed later—(because of control code R) and the screen is cleared (because of control code L).

print"RL"

The PRINT statement may also be usefully inserted into programs (discussed later) to access control codes during program execution.

* 1. Using the FUNC Key

The FUNC key at the upper left of the keyboard assigns other uses for keyboard keys. Let’s take an example. Look at the P key. Also printed on the P key is PRINT. Pressed alone, the P key produces a lowercase ‘p’. If the FUNC key is pressed at the same time as the P key, the word PRINT is produced. Remember our exercise PRINT 3 + 5 in section 2.4? Instead of typing in each character individually for PRINT, press the FUNC key and the P key simultaneously. This function can be utilized at any time while responding to a BASIC-G Ready prompt or typing in a BASIC-G program.

Some precautions.

* When using the CONSOLE command (which will be explained later), the FUNC function can be temporarily disabled.
* The FUNC and CTRL keys are often mistakenly used interchangeably. They’re two separate keys with two distinct functions.
  1. Keyboard Modes

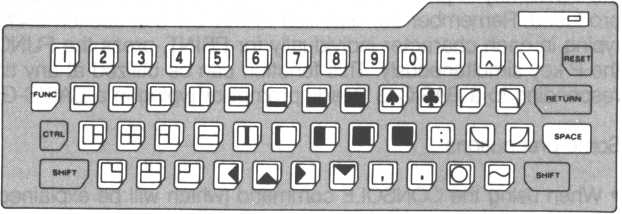
Up to now, you’ve learned how to input, correct and delete letters, numbers, symbols and signs—this is called the letter mode. The M5 also has the capacity to draw graphics.

Graphics mode

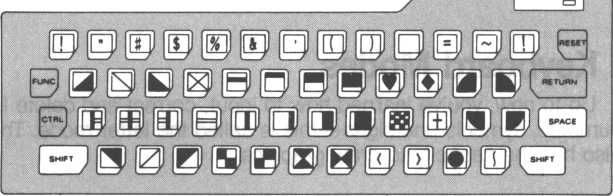
To get into the graphics mode, push the FUNC key and the ‘3’ key together. Notice GRAPH written in small letters above the number ‘3’. The cursor now blinks with the letter ‘G’. In this mode, you can draw graphics using the symbols highlighted in yellow on the keyboard. Other symbols can also be used if the SHIFT key is also pressed. Try drawing some pictures. It will probably be easier if you use the —, t, and i keys as described in

the editing function section.

Graphic characters available with SHIFT key not pressed



Graphic characters available with SHIFT key pressed



LETTER mode

When the M5 BASIC-G system is first powered up, your M5 keyboard is in the letter mode. In the previous section, we explored the graphics mode of the keyboard. To get back into the alphabet mode, press both the FUNC key and the ‘1’ key together. (LETTER is written in small letters on the top part of the ‘1’ key.) The cursor now blinks with the character L ? Recognize it?

You can flip back and forth between keyboard modes as often as you like.

Alphabetic characters with SHIFT key not pressed

/"

**0®000B000@B®[DQ** B 0 0 ® B © 0 0 0 0 0 0 0 CEE) **BT)0000Ei0©DDE)0CE]**

Alphabetic characters with SHIFT key pressed

/ " i °i

**0®©©® B®©0®© DBS 0®©®(E)©®©®[gi±)00iB]**

You’ve learned how to type letters, numbers, graphic symbols, and signs easily on the console. You’ve also learned how to correct and delete them. These techniques are very important and basic to programming in BASIC- G. Be sure you become familiar with all operations described so far.

1. Running Some Simple Programs

Remember the PRINT 3 + 5 statement you typed in earlier? We’re now going to take that one statement and package it as a one line computer program. Type in:

**10 print 3+5**

What happened after the RETURN key was pressed? Right, nothing. That’s because you haven’t requested this simple program be “run” yet—also known to many computer hobbyists as “executing” a program. To run this program, type in

run

followed by a RETURN key. The result is identical to your earlier PRINT 3 + 5 input. When you typed RUN, the M5 extracted PRINT 3 + 5 from your program and performed the calculation.

What is the 10 to the left of the PRINT? It’s a label for the PRINT 3 + 5 statement. It’s obvious this BASIC-G statement is about as simple as one can get. Most programs consist of many such sequentially labeled (numbered) statements. Some statements perform calculations, others perform other tasks.

The three BASIC-G statements below show another simple BASIC-G program.

**10 print 3+5 20 ft i n t 2+4 30 print 9/3**

Type in RUN followed by the RETURN key. That’s right, the last statement with label 20 is executed prior to statement number 30. BASIC-G accesses all statements sequentially depending on their labels (statement numbers). If we include a statement labeled with the number 5, it would be executed before the three statements above. Line numbers can range from 1 to 32767. You can type statements in any order, BASIC-G will order them for you.

A line can contain more than one statement. All that’s needed is a colon, separating each statement. Press the RETURN key after typing in each of the following lines.

10 clss i input A

20 if A>0 and A<10 then print "under 10"

30 if A>10 then print "over 10"

40 end

line 10—clear the screen and request input of a number. Press the RETURN key after keying in a number.

line 20—if your input number is greater than zero and less than 10, print the character string “under 10”

line 30—if your input number is greater than 10, print “over ten”

1. Renumbering Line Numbers

There may be occasions where you want to renumber a program so that all line numbers are equally spaced. For example, before you decide to publish a nice program you’ve composed, you may decide to renumber your program to make it easier to read. The RENUM command does this easily.

Assuming we have the following program,

10 ! c on ve r t d ec i ma l t o h exa d ec i nia l

20 print Hiugn

30 for 1=0 to 233

40 print 15 " ; hex\* (I) >

50 next I 60 end

using the following RENUM command will renumber it as shown below.

RENUM 100 RETURN

Y

first program line number

|  |  |  |
| --- | --- | --- |
| 100 | ! conv | ert dec i mal t o hexadeci mal |
| 110 | pr i nt | "UIBil" |
| 120 | for I | =0 to 255 |
| 130 | pr i nt | 15" ="; hex\* (I).. |
| 140 | next | I |
| 150 | end |  |

1. Constants and Variables

Type in the short program below and RUN it. Follow each statement with a RETURN key. Does it behave the way you expected by looking at the program? Statement 10 sets both A and B to zero as well as clearing the screen. Statement 20 adds one to A and two to B, statement 30 then prints them both. Statement 40 causes the program to go back to statement 20 and the cycle starts anew. This will go on literally forever. When you get tired of it, press the SHIFT and RESET (at upper right of keyboard) keys at the same time to RESET BASIC-G.

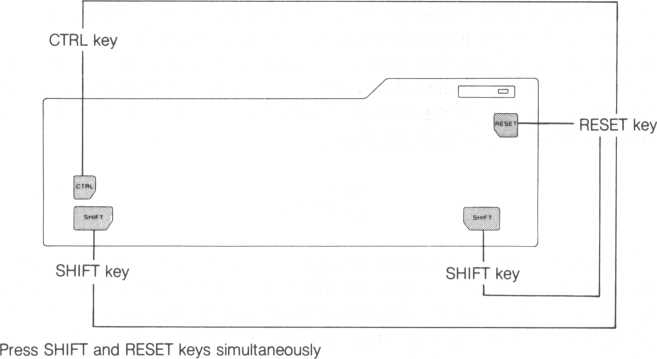
10 cls:A = 0 i B = 0 20 A=A+1:B=B+2 30 print A,B 40 goto 20

Use the SHIFT and RESET key combination to stop execution. If you want to run this program again, type in RUN one more time. However, if you want to stop execution and know the line number where execution stops, press the CTRL and HALT keys together (same key as RESET key—HALT is written in small upper case letters):

"t O psi "C X

will be displayed, xx is the line number executed last. All variables set during execution remain as is, (not reinitialized during a HALT); if a program is left in this state, the OLD, CHAIN, VERIFY and SKIP commands prompt an error message.

Press CTRL and HALT keys simultaneously



Let’s go into some terminology. In this four line program, the number 0 in line 10 as well as the ‘1’ and ‘2’ in line 20 are called ’’constants.” The A and B used throughout the program are called “variables” since they can take on varying values.

Constants are values that do not vary, and so are called “constants.” Constants may be numeric constants, such as 1, 2, -3 or 100, with a limitation that they must be greater than —32767 and less than 32767. Similarly, character constants consist of a combination of characters, such as “X”, “A123”, “BELL”, or even no characters (a character constant of no characters is known as a “null string”). A character constant may consist of at. most 255 characters—also known as a “character string.”

Unlike a constant, a variable can take on varying values. And since there are two types of constants, it makes sense to have two types of variables. One type of variable stores numeric constants and the other contains character strings.

A variable storing numeric constants can be set to 1, and when necessary, changed to another numeric constant, such as 2 or -1000.

The method to specify this type of variable is with a character string up to 32 characters long. These characters may be letters, numbers, graphics symbols, as well as some miscellaneous symbols. But remember the first character must be a letter. Examples:

ABC=20 CAT 1=-100

A variable holding character strings like “A”, “CAT”, or even character constants containing numbers, as with “55” (not equal to numeric constant 55), is specified similar to variables containing numeric constants, but can only store character strings. They also differ because a ’$’ (dollar sign) must be the last character.

ABC$ ="BASIC"

TEN$ ="10 dollars"

NOTE: The function LEN will give you the length of a character string stored in a variable. Look in Appendix B.

1. Operators

The addition sign is part of a family of “arithmetic operators” while the equal sign is a “relational operator.” Another class of operators, known as "boolean operators,” is also available, but will not be dealt with here. Refer to Appendix G for a complete summary.

|  |  |  |
| --- | --- | --- |
| Functions Operators | | Examples |
| Arithmetic  operators | + . —, +:, /, , MOD | PRINT A+B PRINT 2 + 3 |
| Relational | II  A  A  II  o  II  A  V | PRINT A = 3 |
| operators | A  II  o  II  V  A   * o * A | IF A<3 THEN GOTO 30 |
| Boolean  operators | AND, OR, XOR | IF A>10 OR A<4 THEN B = 1 (If A is greater than 10, or less than 4, set B equal to 1) |

The operator precedence follows (the order in which they will be executed).

1. \ >

2. \* , /

1. MOD
2. + , —
3. relational operators
4. AND
5. OR
6. XOR
7. Labels

Instead of jumping to line 20 as we did in line 40 of our last program, we can jump to a character string representation of our jump destination. For example, we can also call our jump destination “$ADD” instead of line 20. We’ll then execute

40 goto 4ADD

which jumps to line 20. Labels are similar to variables since they are also composed of up to 32 letters and numbers. Labels differ in that they must begin with a dollar sign and a letter. So let’s change our program.

10 cls!A = 0 s B = 0 20\*ADD:A=A+l:B=B+2 30 print A,B 40 goto fADD

Rather than "jumping” to line 20, our new program jumps to a statement labeled $ADD which just happens to be statement 20. Alas, the resulting program functions identically. Labels are independent of line numbers. We could renumber our program and it would still function identically.

Other valid character string labels include:

$A 123

SCAT

$FOX

Labels are distinguished from variables containing character strings because the dollar sign is on opposite ends.

1. Array Variables

An array variable is a particular type of variable. The attributes assigned to variables in general also apply to array variables, including storing of numeric constants and character strings. Array variables are unique because they combine related information. For example, your calorie intake for January can be stored in an array variable named CALORIE consisting of 31 separate fields, as shown below.

|  |  |
| --- | --- |
| Array variable | Remarks |
| CALORIE(1)=2000 CALORIE(2)=2550 | 2000 calories on January 1 2550 calories on January 2 |
| CALORIEC 31)=2300 | 2300 calories on Januray 31 |

Before using array variables, we must allocate them at the beginning of a program with a DIM statement. In our calorie example,

dim CALORIE(31)

suffices.

Character strings are also stored similarly; the days of the week can be kept in an array of seven fields DAY$(1) TO DAY$(7),

DAY(l)="Monday" DAY(2)="Tuesday"

DAY(7)=“Sunday"

and declared with the following DIM statement.

dim DAY\*(7)

Type in and execute the next program. It illustrates the usage of both types of array variables. Line 30 allocates our array variables.

lOrem DICE 20 els

30 d i m A $(6)> DIE(6)

4-0 for 1 = 1 to 50

50 MUM = rnd (5) + 1

60 DIE(MUM>= DIE(MUM)+1

70 next I

80 for 1=1 to 6

90 read A$(I)

100 print A\*(I);"was rolled"» DIE(I)5" t i mes"

110 next I

120 data "one ”, "two ", "three ",Hfour ", “five %"six

This program randomly rolls a die 50 times and stores the number of times each number comes up in its corresponding array variable field.



A$(1) stores "ONE"

A$(2) stores "TWO”

A$(6) stores “SIX"

DIE(1) stores the number of times 1 is rolled DIE(2) stores the number of times 2 is rolled

DIE(6) stores the number of times 6 is rolled

**CHAPTER**

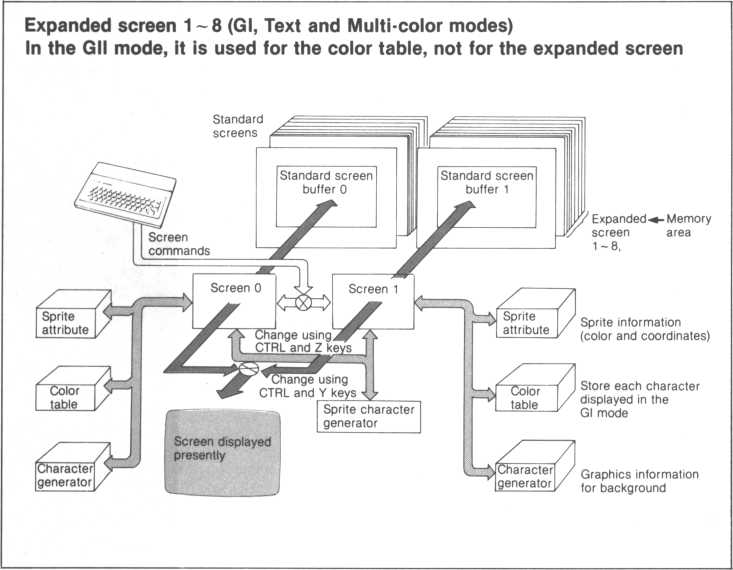


1. TWO BASIC-G SCREEN BUFFERS AND FOUR SCREEN MODES



BASIC-G has two screen buffers, named “screen 0” and “screen 1”. And since your television can only display one screen buffer at a time, one screen buffer is always hidden from view. This isn’t a problem since changing screens is easy. The CTRL key is used to flip back and forth between screen buffers.

Here’s an illustration to help us out. This figure may be giving you more information than is reasonable. Don’t try to take it all in at one sitting. Remember it’s here when we begin discussing screen modes, expanded screens, etc.



* 1. CTRL U—Display Screen 0

When power is first supplied to your M5 console, screen 0 is displayed. Pressing the CTRL U key combination will always insure screen 0 is displayed. |f screen 0 is displayed, nothing will happpen by pressing the CTRL U key combination since screen 0 is already displayed. However, if screen 1 is displayed, pressing the CTRL and U keys simultaneously will return you to screen 0. In other words, pressing CTRL U will always ensure screen 0 is displayed.

This function may be useful if you’ve forgotten which screen buffer is displayed. Simply display screen 0 to orientate yourself.

* 1. CTRL V—Display Alternate Screen

Screen 1 (the alternate screen) is displayed by pressing the CTRL V key combination. Let’s go through a short exercise.

1. Be sure screen 0 is displayed by pressing the CTRL and U keys at the same time.
2. Type in some text. Anything will do.
3. Press the CTRL V key combination. Screen 0 disappears and screen 1 magically appears.
4. Type some text into screen 1. Again, anything will do.
5. Press the CTRL V key combination. Notice screen 0 is displayed again.
6. Press the CTRL V key combination again. Yes, screen 1 is again displayed.
7. Steps 5 and 6 can be repeated as often desirable to alternate between screens 0 and 1.

This exercise showed us the CTRL V combination will flip back and forth between our two screen buffers.

* 1. CTRL Y—Display Alternate Screen, Write to Hidden Screen

What if we wanted to peek at the alternate screen buffer in order to accurately copy text onto our displayed screen buffer? Among other things, that’s what a CTRL Y can do for us. An exercise will illuminate it for us.

1. Use CTRL V to type text into screen 1. Anything will do.
2. Press CTRL U or CTRL V to return to screen 0.
3. Press CTRL Y to display screen 1.
4. Type miscellaneous text. Notice the cursor and your typed text is not displayed on screen 1. You got it, it’s going onto screen 0.
5. Press CTRL Y. Screen 0 is now displayed. It contains the text you just typed in. Newly typed text is still inserted into the now displayed screen 0.
6. Steps 4 through 6 can be repeated as desired.

Some pointers.

* When CTRL Y is used and the cursor resides on the hidden screen, text is not inserted into the displayed screen.
* A second CTRL Y reverts back to the previous screen which now contains the newly typed text.
* Screen 1 can be substituted for screen 0 in the above exercise. Rather than displaying screen 0 in step 2, omit step 2 and substitute screen 0 for screen 1, and vice versa, in steps 3 through 6.
* It may be somewhat confusing initially, so let’s summarize the general

case:

1. Change to the screen that you want to contain newly typed text. Ensure the cursor is displayed on this screen.
2. Use CTRL Y to display the previously hidden screen.
3. Type text. It’s inserted into the screen displayed in step a.
4. Use CTRL Y to return to the screen displayed in step a.
   1. CTRL Z—Write to Hidden Screen

What if we want to simply write text to the alternate screen buffer? It’s easily done with the CTRL Z function. The CTRL Z function is easier than the CTRL Y function, so let’s state the general case right away.

1. Press the CTRL Z key combination. The cursor disappears to the hidden screen buffer.
2. Type in miscellaneous text.
3. Use CTRL Z again to force the cursor to return to the screen displayed in step 1.
4. Use CTRL U or CTRL V to look at the alternate screen. See your newly typed text?
5. Steps 1 through 4 can be repeated as much as desired.
   1. Interaction

The interaction between the CTRL U, V, Y and Z functions is straightforward and predictable, but there are quite a few implications. If you find yourself getting lost, use the screen control functions to find out which screen buffer is displayed. Then look at the cursor to ascertain which screen buffer is being displayed, then change the screen and/or cursor if incorrect.

It’s handy to know a CTRL T key combination pressed after a CTRL U key combination will display screen buffer 0 in the text mode. Several others follow.

|  |  |
| --- | --- |
| Combination | Function |
| CTRL T after CTRL U | return to text mode screen 0 |
| CTRL V after CTRL U | display screen 1 |
| CTRL Y after CTRL U | display screen 1, cursor goes to screen 0 |
| CTRL Z After CTRL U | display screen 0, cursor goes to screen 1 |
| CTRL Z after CTRL Y | same as CTRL V |
| CTRL Y after CTRL Z | same as CTRL V |

Don’t worry if you haven’t mastered these functions. When necessary, review this section and try it again.

* 1. Four BASIC-G Screen Modes

Apart from having two distinct screens, BASIC-G allows four screen modes that determine the characteristics of a displayed screen. The characteristics differ in the number of characters displayed per screen, the quality (resolution, or sharpness) of each character, color and a few graphics capabilities.

* Gl—Graphics mode 1
* Gil—Graphics mode 2
* Multi-color mode
* Text mode
  1. Gl—Graphics Mode 1

To enter the Gl screen mode, press the CTRL S keys simultaneously. Important characteristics of this mode include:

* 32 characters in the horizontal direction and 24 characters in the vertical direction for a total of 768 characters
* each character consists of an 8 by 8 dot matrix (8x8 equals 64 dots, also known as “pixels”)
* this mode is assumed when power is first supplied to the M5 console
* graphics is realized by entering graphics characters on the keyboard (while in the graphics mode)
  1. Gil—Graphics Mode 2

To enter the Gil screen mode, press the CTRL R keys simultaneously. Important characteristics of this mode include:

* 32 characters in the horizontal direction and 24 characters in the vertical direction for a total of 768 characters
* each character consists of an 8 by 8 dot matrix (8x8 equals 64 pixels)
* graphics is realized by entering graphics characters on the keyboard (while in the graphics mode)
* full graphics capability
* designation of detail colors
  1. Multi-color Mode

To enter the multi-color screen mode, press the CTRL Q keys simultaneously. Important characteristics of this mode include:

* 32 characters in the horizontal direction and 24 characters in the vertical direction for a total of 768 characters
* each character consists of an 8 by 8 dot matrix (8x8 equals 64 pixels)
* mosaic creations are feasible
  1. Text Mode

To enter the text mode, press the CTRL T keys simultaneously. Important characteristics of this mode include:

* 40 characters in the horizontal direction and 24 characters in the vertical direction for a total of 960 characters
* each character consists of 6 horizontal and 8 vertical dots (48 pixels per character)
* text mode screen contains more characters than other screen modes
* graphics capability is poor
* some graphics functions are not accessible
  1. Screen Modes and Control Codes

Screen modes can be accessed using control codes embedded in programs.

|  |  |
| --- | --- |
| CONTROL CODE | SCREEN MODE |
| R | Gil mode |
| S | Gl mode |
| T | Text mode |
| Q | Multi-color mode |

* 1. Screens and Their Modes



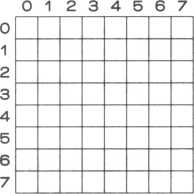
Both screens 0 and 1 can each be set to different screen modes. An exception—one screen cannot be in the multi-color mode while the other screen is in the Gil mode.

* 1. Characters, Pixels, and Screen Coordinates

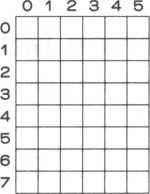
The Gl, Gil and multi-color modes can each contain 768 characters (24 lines of 32 characters), each of which is assigned a unique set of two numbers. (The text mode contains 980 characters since it contains 24 lines of 40 characters.) This set of two numbers (start from zero) is known as a character’s coordinates. Similar to geometry, a coordinate consists of a value for its horizontal location (x coordinate) and a value for its vertical location (y coordinate). The upper left corner of the screen is (0,0). Thus used, a character’s coordinates indicate a unique location on the screen. For example, since the text mode consists of 24 tines of 40 characters, it’s center is indicated by CURSOR (19,12)—a BASIC-G statement used to manipulate the cursor.

(15,10) from the home position is called “absolute coordinates” since there is no ambiguity. There are also "relative coordinates” that are marked off from another arbitrary set of coordinates. If say absolute coordinate (100,50) is marked off from (0,0) and then moved again to relative coordinates of (20,10), the target coordinate is absolute coordinates (120,60).

A character is made by combining many small dots, also known as pixels. A character in the Gl, Gil and multi-color modes have 64 pixels configured in an 8 by 8 square. Likewise, the text mode has 48 pixels configured in a 6 by 8 rectangle.



Gl, Gil and multi-color mode dot matrix

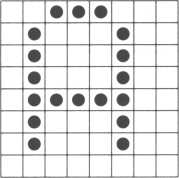


Text mode dot matrix

Once the appropriate pixels are filled in, a dot matrix can represent a character. For example, the letter A in the Gl mode is represented by the dot matrix below.

0

0 1 2 3 4 5 6 7



1

2

3

4

5

6

7

(Pixel representing the character ‘A’)

If the light pixels are colored in while the dark pixels are changed to white, the dot matrix represents the same character against a dark background. In fact, the alphabet mode cursor is constructed in this way.

The text mode can contain more characters per line by omitting the rightmost two columns of the dot matrix above.

* 1. Dividing the Screen

Using the screen coordinates from the previous coordinates discussion, we can divide the screen into different blocks. Type in the following short program after the Ready prompt is given. Press the RETURN key after each line.

10 p r i n t " S" press SHIFT CTRL S keys

20 view 10> 10» 20 ? 20 simultaneously

Now type RUN followed by the RETURN key to run our short program. The Ready prompt and cursor moved to the center of the screen. Why? The 10,10,20,20 in statement 20 above marked off a block, otherwise known as a “viewport.” The 10,10 is the upper left corner of our viewport while the 20,20 indicates the lower right corner. The viewport is now our new screen. Once a viewport is defined, the cursor cannot move out of the viewport.

Use the CTRL K keys to place the cursor at the home position. Rather than going to our old home position, the coordinate (10,10) is now our new home position. In other words, we now have a screen within a screen.

Think of the new home position as (0,0), coordinates will be relative to these new coordinates.

Since the Gl, Gil and multi-color modes have 24 lines of 32 characters, VIEW 0,0,31,23 (count from zero) is assumed when power is first supplied to your M5 console.

To return to our original screen, type in VIEW followed by a return key.

VIEW X,Y,X,Y 1122

corresponds to

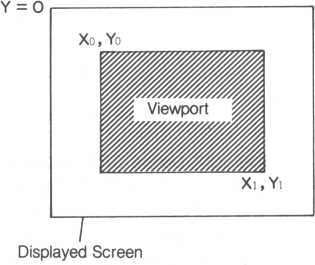
VIEW Xo, 'Yo, I , I

Xi, Yi I I

Upper left corner of viewport.

Lower right corner of viewport

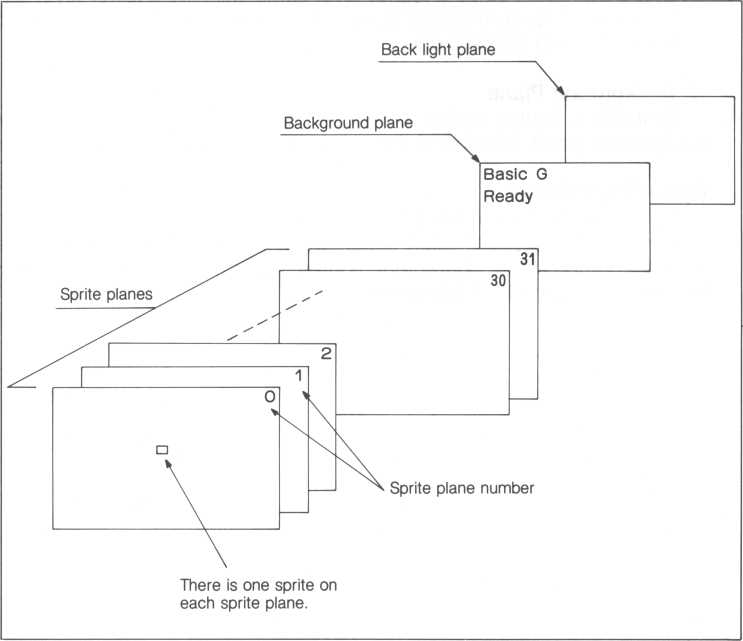
X = 0



Screen divided by a VIEW statement

* 1. Game Planes

You may have wondered how some computer games define the shape of their spacecraft or battleships. With your M5, these can be described by what are known as “sprites” residing on “sprite planes.” Sprites are nothing more than a graphic representation of a UFO, battleship or whatever your imagination conjures up. Your M5 has a maximum of 32 sprite planes as well as a “background plane” and a “backlight plane.”



1. Sprite Planes

Sprites, which can be as small as an 8 by 8 dot matrix or as large as four 16 by 16 dot matrices, are used in different ways depending on the screen mode. Usually, sprites are moved around the screen either by game user actions or software. The 32 sprite planes can move virtually at the same time and are numbered hierarchically from 0 to 31; 0 is the highest level. It’s significant since a higher level sprite will hide a lower level sprite. For example, if sprite 0 passes over sprite 5 and sprite 0 is the same size or larger, sprite 5 will be hidden.

1. Background Plane

Characters, figures and simulated graphics can be represented on the background plane. Division of the screen into viewports can also be done.

1. Backlight Plane

The backlight plane is the final plane containing only a backlight color. Using a BCOL (backlight color) statement, its color can be changed. Unlike the background plane, characters, figures, and simulated graphics cannot be specified on the backlight plane.

4

**CHAPTER**

1. FILES ON CASSETTE TAPE

After composing and typing in a program file into your M5’s memory, it’s cumbersome to have to retype it every time we want to use it. Your M5 provides the capability to save and verify programs on cassette tape with the SAVE command.

Before proceding any further, be sure the connections between your cassette recorder and M5 console are correct and snug. Then, if your recorder has the remote function, step through section 4.1. Otherwise, go through section 4.2.

If you have difficulty controlling your tape recorder, refer to its owner manual.

* 1. Using a Remote Function Recorder—SAVE, VERIFY Commands

The remote function on your recorder will allow the M5 to automatically stop and start the tape transport to facilitate recording and playback.

• Using the SAVE command to save a file

1. Think up a name for your file, (it may consist of up to 9 characters). Try to think up names that describe your file, e.g., CALORIES to describe a file for calculating calorie intake; it’ll be easier to remember later.
2. When the BASIC-G Ready prompt is displayed, type in the SAVE command with the file name enclosed between double quote marks. Do not press the RETURN key yet.

save"f ile-name"

1 up to 9 letters

The SAVE command will not work unless the file name is enclosed in

double quotes.

3.Set your tape recorder to RECORD. (The tape transport should not yet move.)

1. Press the RETURN key. You’ll hear a soft “click”, the tape will begin to move and the cursor will disappear from the screen. This means your file is now being copied to the tape.
2. When your file has been completely copied, you’ll again hear a soft click and the tape transport will stop. The Ready prompt will reappear.

• Using the VERIFY command to confirm whether a recorded file is correctly saved on cassette tape.

I.Set the playback volume on your recorder to a reasonable playback level, a 7 or 8 is probably sufficient.

2.When the Ready prompt is displayed, type in

ski p

followed by the RETURN key.

1. Rewind the tape.
2. Type in

veri fy"fit e-name"

As with the SAVE command, enclose the name of the file you want to check within double quotes. Don’t press the RETURN key yet.

5.Set your recorder to PLAY. The tape transport will not yet move.

1. Press the RETURN key. You’ll hear a soft click, the tape will begin to move and the cursor will disappear from the screen.
2. If a file is found that precedes the desired file, the following message will be displayed on the screen if it’s a BASIC-1 file.

SKIP"file-name.BI"

Or, if it’s a BASIC-G file, a

SKIP"f ile-name, BG"

message is displayed; the suffix designates the type of file. Notice the cursor has not yet reappeared, that’s because it’s still looking for your file.

1. When the desired file is found,

FIND"file-name.BG"...

is displayed. The dots indicate give a rough measure of the file’s length—one dot for every 256 bytes.

Note: If a VERIFY command is typed in without a file-name and double quotes, the M5 searches for the most recently saved file. If this file is not found, an “Err 8” message is displayed.

* 1. Using a Recorder Without the Remote Function—SAVE, VERIFY Commands

If you. don’t have the remote feature on your tape recorder, it’s still easy to save and verify your files on cassette tape. The only significant difference is that you have to directly control tape movement.

• Using the SAVE command to save a file

1. Think up a name for your file, (it may consist of up to 9 characters). Try to think up names that describe your file, e.g., CALORIES to describe a file for calculating calorie intake; it’ll be easier to remember later.
2. When the BASIC-G Ready prompt is displayed, type in the SAVE command with the file name enclosed between double quote marks. Do not press the RETURN key yet.

save"fi t e-name"

1 up to 9 letters

3.Set your tape recorder to RECORD. (The tape will begin to move.)

1. Press the RETURN key. You’ll hear a soft “click”, and the cursor will disappear from the screen. This means your file is now being copied to the tape.
2. When your file has been completely copied, you’ll again hear a soft click and the Ready prompt will reappear.
3. Turn off your tape recorder.

• Using the VERIFY command to confirm whether a (recorded) file is correctly saved on cassette tape.

I.Set the playback volume on your recorder to a reasonable playback level, 7 or 8 is probably sufficient.

2. Rewind the tape.

veri f y"f ile-name"

3.Type in

As with the SAVE command, enclose the name of the file you want to check within double quotes.

4.Press the RETURN key. When a soft click is heard, set your recorder to PLAY. The tape will begin to move and the cursor will disappear from the screen.

5.If a file is found that precedes the desired file, the following message will be displayed on the screen if it’s a BASIC-I file.

**SKIP"ti1 e-name.B I "**

Or, if it’s a BASIC-G file, a

**SKIP"f i1 e-name.BG"**

message is displayed; the suffix designates the type of file. Notice the cursor has not yet reappeared, that’s because it’s still looking for your file.

1. When the desired file is found,

**FIND** "fit **e-name.**BG" **. ..**

is displayed. The dots indicate give a rough measure of the file’s length—one dot for every 256 bytes.

Note: If a VERIFY command is typed in without a file-name and double

quotes, the M5 searches for the most recently saved file. If this file is not found, an “Err 18”message is displayed.

* 1. Retrieving a File Saved On Tape—OLD, CHAIN Commands

Three commands are available to retrieve a file saved on cassette tape. They differ in the actions subsequent to retrieval. Their usage is virtually identical to the VERIFY command (refer to section 4.1 or 4.2, whichever applies to your tape recorder type). However, if the file-name is not specified, the first file found on tape is used. All three commands should be used with care since they can erase or alter any programs currently in memory.

|  |  |
| --- | --- |
| COMMAND | CHARACTERISTICS |
| OLD | • Erase the file in memory (if any) |
|  | • Retrieve the desired file from cassette tape |
|  | • Load it into memory. |
| CHAIN | • Erase the file in memory (if any) |
|  | • Retrieve the desired file from cassette tape |
|  | • Load it into memory |
|  | • Execute the file (must be a BASIC file) |

* 1. Using the LIST Command

Files to be retrieved and loaded by the OLD or CHAIN command need to be saved on tape with the LIST command. Its syntax is

L I ST"lit e-name "

later retrieved by  
the OLD or CHAIN command

Usage of the LIST command is identical to, but takes a bit longer than, the SAVE command (refer to section 4.1 or 4.2).

* 1. Using the SKIP Command

Being similar to the VERIFY command, the SKIP command skips over any files on tape having no relationship to the desired file. For example, suppose “TEST3” is to saved on tape. But files “TEST1” and “TEST2” currently take up space at the beginning of the tape. The SKIP command is used to look for the end of “TEST2”, where “TEST3” is to be saved.

Type in

S K I P " T E S T "

followed by the RETURN key to find the end of “TEST2” below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tape | T | "test 1” | "test 2” | |
|  | |  | end of test2 — | save test3 here |

SKIP" TEST 1 "

When “TEST1” is found, a

message is displayed. After the end of “TEST2” is found, a

F I N D " T E S T 2 "

message is displayed and a soft click is heard. If you have a recorder with the remote feature, its tape transport will stop. If you don’t have this type of recorder, STOP the tape after you hear a click. “TEST3” can now be saved on tape with either the SAVE or LIST commands.

Note: If a file-name is not specified, SKIP skips the first file encountered and returns you to the Ready prompt. So if you want to skip several files, be sure to indicate the appropriate file between double quotes.

* 1. Saving and Retrieving of the Screen—VSAVE Command

Characters and pictures displayed on the screen can be saved on tape by using the VSAVE (Video SAVE) command.

The VSAVE command is used in the same way as the SAVE command.

But note that the alternate screen buffer is saved, not the screen buffer currently displayed. So if you want to save the displayed screen, press the CTRL V keys simultaneously to display the alternate screen and use the VSAVE command.

VSAVE saves all data from VRAM (Video RAM), therefore the time required depends on the screen mode, usually one to two minutes. Refer to Appendix J for the memory map of VRAM.

Retrieving a file from tape containing VRAM data is similar to retrieving text files, both file types use the OLD command. However, before retrieving a VRAM data file, be sure the screen buffer is set to the screen buffer and mode the retrieved file expects. Otherwise, the screens or their modes differ and an “Err 19” message is displayed. So when saving a VRAM screen, remember to jot down its screen mode and buffer in a notebook or on its cassette tape cartridge.

When the applicable screen is retrieved, use the CTRL Y key combination to display the retrieved screen (on the alternate screen buffer). If you want the retrieved screen to be displayed immediately after being retrieved, type in

pr i n t" y "sol d “ 'VRAM-f i l e-name "

1 1 1

1—name of VRAM data file

— important semicolon

1—Control code

(press CTRL, SHIFT, Y keys at the same time)

**CHAPTER**



1. WHAT IS A CHARACTER?

In this section, we’ll go in depth into the concept of a character and how it’s represented in the M5 system. This is important if we are to understand how characters are stored and represented and ultimately affect the outcome of our software—if we’re to create exciting video game software, this is something we’ve got to know.



Characters represent letters, numbers, signs and graphics symbols. Ihternal to the M5, each character is kept in what is commonly referred to as its ASCII representation. Look at Appendix A. The ASCII code is made up of 256 code combinations (numbered from zero) representing different characters.For example, 65 corresponds to the letter ‘A’.

A BASIC-G statement named CHR$ will take the numeric equivalent of a character and produce the character. The statement

**print** chr\*(65)

prints the character ‘A’. Type it in when the Ready prompt is displayed followed by a RETURN key.

* 1. Different Numbering Systems

For the character ‘A’, we found we could represent it with the number 65, which is part of the decimal numbering system. That’s the system most of us use in our daily lives, counting from 0 to 10, then from 11 to 20, and so on. Well, unfortunately or fortunately depending on your perspective, there are other numbering systems.

The system most commonly used for computer programming is not the decimal system, but rather two systems called the “hexadecimal” number system and the binary number system. Rather than go into an involved discussion on number systems, simply look at the following table. Refer to an appropriate textbook for a complete description.

|  |  |  |
| --- | --- | --- |
| Binary | Decimal | Hexadecimal |
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | A |
| 1011 | 11 | B |
| 1100 | 12 | C |
| 1101 | 13 | D |
| 1110 | 14 | E |
| 1111 | 15 | F |

Given a binary number, the numbers to the right represent the decimal and hexadecimal systems respectively. For example, the number 11 (decimal) is expressed as 1011 in binary and OB in hexadecimal.

Go back to the ASCII chart. The character ‘A’ is represented by 65, decimal. Similarly, in the hexadecimal numbering system, 41 also represents the character ‘A’. Whenever the hexadecimal system is used in BASIC-G, always insert the character before the number. This informs BASIC-G that the following number is in the hexadecimal system format. So,

print chr$(41) also prints an ‘A’. Notice the in front of the 41.

1. Finding the Pattern Code (Dot Matrix)

Do you remember our discussion of pixels and characters in section 3.13? Let’s create our own characters. But before we do that, let’s look at some existing characters and how they are represented in the M5 computer.



Type in:

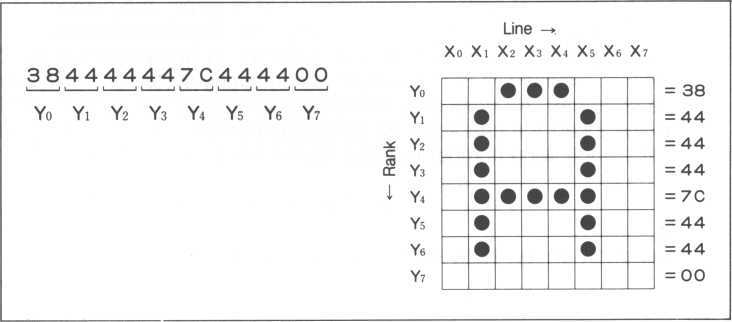
p r i n t r d c h r $ (65 > 1)

The 65 within the parentheses tells BASIC-G we want to find out how the character ‘A’ (65 in decimal notation) is represented in the M5 computer. The M5 responds with a string of 16 hexadecimal numbers:

384444447C444400

which is the pattern code for ‘A’. But what does it mean? How do we decipher it?

The following figure illustrates the relationship between ‘A’ and its pattern code specified down the right side. The aggregate of all these numbers form our pattern code.



Let’s call the X coordinates “lines” and the Y coordinates “ranks.” Each rank has up to eight pixels. If we halve each rank, we have two fields of four pixels each.

Xo Xi X2 x3 X4 Xs Xe X?

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Yo |  |  | • | • | • |  |  |  |
| One rank is encoded | i | i | 1 | 1 | 1 | 1 | i | 1 |
| with two hexadecimal —► | O | 0 | 1 | 1 | 1 | 0 | O | 0 |
| digits | I— | ~r~ | | J | I | ~T~ | | 1 |
| Four dots translate into -\*• |  | 3 | |  |  | 8 | |  |

one hexadecimal digit

We then split each half into four digits, XO to X3 and X4 to X7 for the left and right halves respectively. If we substitute a ‘1’ for each colored pixel and a ‘0’ for other pixels, we come up with two sets of binary numbers, one for each of the two four-pixel fields.

left half right half

XO X1 X2 X3 X4 X5 X6 X7

0 0 1 1 1 0 0 0

Refer to our binary/decimal/hexadecimal table in section 5.1. For each four digit half, look down the table and search for its binary number. Find its corresponding hexadecimal number. The left half’s 0011 (binary) is equivalent to 3 (hexadecimal). Likewise, the right half’s X4 to X7, or 1000 (binary) yields 8 (hexadecimal).

2-digit hexadecimal number = 38

I— right half left half

38 is our 2-digit hexadecimal number. Repeating this process for all eight ranks, Y0 to Y7, results in eight two-digit hexadecimal numbers, what we were striving for. The long string of numbers we saw before can be traced back to these eight numbers. Refer to Appendix D for some aids. Make copies of Appendix D and explore several sprite designs.

38 44 44 44 7C 44 44 00 Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7

we just finished discussing the technique to ascertain this hexadecimal number

**CHAPTER**

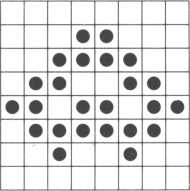


1. CHARACTERS IN THE Gl MODE
   1. Setting the Pattern Code (Dot Matrix)

After learning how a dot matrix is evaluated for its pattern code, let’s learn how to make our own pattern codes (for our own characters) using the STCHR statement. Simply make a 8 by 8 draft of your proposed dot matrix. Look in Appendix D for a sample form. Translate all eight ranks of your dot matrix into eight two-digit hexadecimal numbers and enclose them in double quotes in a STCHR statement.

How about a simple graphics character? A UFO.

= oo = 18 = 3C = 66 = DB = 7E = 24 = 00



The character above translates to:

s t ch r " 0 0183c 66d b 7 ©240 0 “ t o S<7 f ? pattern code—I

“—character set number

I ASCII representation

associated with this pattern code (set by the user)

When using the Gl screen mode (your current mode), always use character set number 1 (since the Gl mode only has one character set). After this particular STCHR statement is executed, the UFO character is accessed by referring to its ASCII representation assigned in the STCHR statement, a &7F in this case. Running the short program below will display the UFO at the center of the screen. The CURSOR statement in line 30 uses the character coordinates we discussed in section 3.13.

10 print "BIBB" ■+ control codes

20 stchr "00183c66db7e24-00" to -\*<7F, 1 30 pr i nt cursor(15? 11)'chr$C&7F)

* 1. Deleting a Pattern Code

Characters developed and encoded using the STCHR command are valid as long as the screen mode remains the same. When the screen mode changes, the newly programmed ASCII code character will change to its ASCII representation for that screen.

Another way to delete all pattern codes for a particular screen mode is to enter STCHR without any parameters.

STCHR RETURN

* 1. Coloring a Character

Add the following line to the three line program in section 6.1.

25 color &7F> &81 pattern code 1 ^—color code

If you have a color television, the UFO will change to red (because of the ’8’ in &81). Look in Appendix A for a listing of the colors and their codes. They’re all listed in hexadecimal notation, so be sure to include the ’&’ character.

The color code can be split up into two fields.

&

character color

*background* color

The character color highlights the character (color of darkened pixels). The background color (alternate pixels colored a different color) provides the background color for the character. If &18 is used instead of &81 in line 25, the background color and character color are swapped. So instead of red, the UFO will be black and its background color will be red. Notice color code 0, (the absence of color), can be treated just like the other colors; however, if the background is color code 0, it will be invisible to us.

When a character is colored in the Gl mode, seven other characters are colored identically. Refer to the next table, notice how it’s split into an upper and lower half. Look across the top of the table until the T is found, then map down until the ‘F’ line is found. Since the 7F combination found us at the bottom half of the table, all characters in the bottom half of row 7 are also colored by statement 25, i.e. the x, y, z, {, | ,), ~ and <l characters are also colored by statement 25.

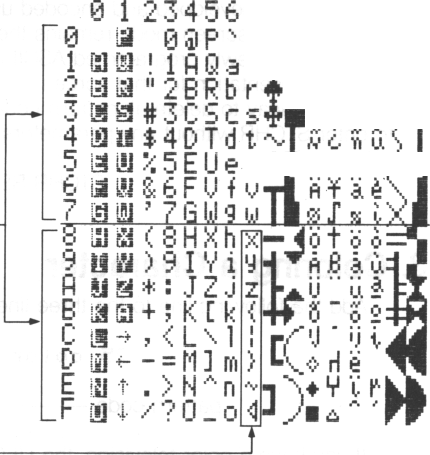
Some pointers on coloring characters.

Divided into two halves. Changing the color of a character also changes the color of the other seven characters along its vertical line.

789ABCDEF Pi-I i

u-H I l '/.\*

The < character is set as the UFO. Changing the UFO color changes the color of all eight enclosed characters.



* If the same color is used for the character and the background, you will not be able to recognize which character is being displayed.
* If black is used for both the character and the background, the screen will appear blank. If this occurs, change the mode. Then LIST the program and change the color code.
* Decimal numbers may also be used to color characters, but this makes it a bit more difficult for us to decipher the foreground and background colors.
  1. Changing the Screen Color

Changing colors is not only limited to one or & few characters. The entire screen’s background color and character color can be changed in one stroke. Type in the next short program.

10 print "Liieti" control codes



20 for 1=0 to 255 step 8 30 color Ij &12 40 next I

hexadecimal notation may also be used (&FF)

The beginning portion of line 20 is used to take care of all possible characters. But since only one character needs to be accessed every eight ASCII characters, the STEP 8 clause accomplishes this nicely. This program results in a green background and black characters. If the &12 in line 30 is substituted with a &21 instead, the screen would be green characters against a black background..Experiment with line 30. Practice changing the color codes, there are 256 combinations (16x16).

* 1. Changing the Backlight Plane Color

The background plane can be colored with the BCOL statement. But before experimenting with it, change your screen to the Gil, multi-color or test mode to delete any old pattern codes, then return to the Gl mode. Type in the below followed by a RETURN key:

b C O L O 3

The screen color will change to blue (color code &05). Now try changing the screen color code to &02.

* 1. Moving the UFO Character

Type in and run the following program.

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | print | "111011 | ii |
| 20 | stchr | "00183c66 | |
| 30 | c o l o r | &7F ? & | :5 1 |
| 4 0 | for X = | ■ 0 t o | 31 |
| 50 | P r i n t | cur so | r (X ? ! |
| 60 | els |  |  |
| 70 | next |  |  |

Line 50 will shoot the UFO across the screen at Y coordinate equal to 5. The X coordinate varies between 0 and 31 because of the FOR-TO-NEXT statement between lines 40 and 70, inclusive. Each time the UFO is displayed, line 60 erases it so the next one can be displayed to show a semblance of movement. If line 60 is omitted, a line of 32 UFO’s will be displayed across the Y = 5 coordinate. The ’<’ character is the UFO assigned in line 20; it doesn’t look like our UFO until line 20 is executed, designating &7F as our UFO. If the UFO moves too quickly, try inserting a SLEEP statement within the FOR-TO-NEXT statement. The SLEEP statement forces the program to suspend execution for up to two seconds. Try inserting line 55 as follows,



55 sleep 12?1

The SLEEP statement is frequently used to stall movement to allow the human eye to catch all the action (otherwise you’d see the UFO at its origin and then, all of a sudden, at its destination).

Similar to the CURSOR statement in line 50, the LOCATE statement can also control the cursor by locating it at the prescribed X-Y coordinates. Replace line 50 with

50 locate x>5 52 print "4"

and our program will still function identically. Did you notice that line 52 contains our UFO between the two double quotes? That’s because line 20, executed earlier on our last pass, set ‘7F’ as our UFO until we change it again.

* 1. Moving the UFO Using Expanded Screen Buffers

In BASIC-G, eight expanded screen buffers can be displayed in every mode except the Gil mode.

Expanded screen buffers can be envisioned as eight extra screen buffers added to the two already mentioned, ten screen buffers in all. If a UFO is displayed on the eight screen buffers only a small distance apart in succeeding screens, an illusion of motion can be achieved, not unlike turning animated cartoon book pages.

While VRAM (memory map page 0) is used frequently in the Gil mode, those 8K bytes are used in the Gl and multi-color modes to construct eight 1K byte screen buffers. So if either primary screen is set to the Gil mode, expanded screen buffers cannot be utilized. Likewise, changing to the Gil mode will destroy any expanded screen buffers. Let’s go through an example. Type the following program (it’s similar to the program we just reviewed).

|  |  |  |
| --- | --- | --- |
| to | P r i n t | " LIMB" |
| 20 | st chr | "00183 |
| 30 | c o L o r\* | &7F., 5 |
| 40 | f o r X = | 0 to 8 |
| 50 | L ocate | X,5 |
| 60 | pr i nt | " <j " |
| 70 | escrn | X, 1 |
| 80 | next X |  |
| The format of | line 70 is |  |

70 escrn X>

'—clear screen after copying  
to expanded screen X  
— expanded screen number

The ESCRN (Expanded SCReeN) statement above takes the displayed screen, copies it to expanded screen X and clears the screen. Since succeeding expanded screens then contain our UFO at slightly different positions, we can redisplay the expanded screens to gain the semblance of motion. Obviously, if anything else is displayed on the screen when an ESCRN copy statement is executed or a screen mode change is actuated, it will unconditionally copy that screen to the desired expanded screen, perhaps ruining what you wanted to copy.

Let’s replay the screens.



|  |  |  |
| --- | --- | --- |
| 10 | print | "ISMS |
| 20 | for X- | : 0 t o |
| 30 | escrn | X» 0 |
| 40 | s t eep | 12, 1 |
| 50 | next ) |  |

where line 30’s function is explained with:

30 escrn X, 0

^— display expanded screen X (omitting this value and the comma is optional in this case, since the default is 0)

expanded screen number

The general format of the ESCRN statement parameters is:

0: display expanded screen X 1: clear the screen after copying the screen to expanded screen X 2: copy the present screen to expanded screen X expanded screen number

Specifying ESCRN without any parameters displays the standard screen.

Let’s look into a more'compact sample program that does basically the same thing as the previous two programs.

1 0 ! GI ino d e esc r n u f o

**20 print** "liliiaiiiSiBlilSi“

30 gosub f UFO.ST 40 gosub f EF'R I NT 50 gosub fESC.IND 60 goto 50 190 end 40OfUFO.ST

410 stchr "001S3c66db7e2400" to

420 return

500f EF'R I NT

510 f o r I = 0 t o 8

520 escrn I> 1

530 pr i n t cur sor <I+3, 10)5"4 " ?

540 color ?:50

550 next I

560 print "LULi"

570 return 60OfESC.IND 610 for 1=0 to 8 620 escrn I

630 s L eep 12? 1

640 next I 650 return

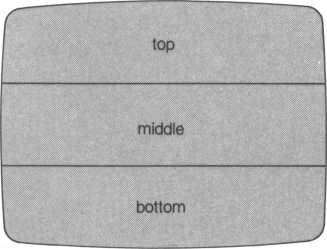
*15*

**CHAPTER**



1. CHARACTERS IN THE Gil MODE
   1. Gil Mode Screen Characteristics

Although there are similarities between the Gl and Gil modes, probably the most striking difference is that the Gil mode is divided into three screen parts, the upper, middle and lower parts. Naturally, all ASCII characters can be displayed in all three parts.



Gil mode screen partitioning (numbers denote character set number)

* 1. Setting the Pattern Code (Dot Matrix)

Just as with the Gl mode, the STCHR statement is used in the Gil mode to construct a character. Basically, the same rules apply. But since the Gil screen is partitioned into three parts, the character set number must reflect the correct portion, indicated by 1, 2 or 3 above. Therefore, to construct a UFO on the Gil screen, each screen portion is initialized separately, as shown below.

s t c h r "00183c 66 d b 7 e240 0" to &7f,l - top part of screen s t chr" 001 83c66db7e2400" to &7f, 2 — middle part of screen s t chr “ 00183c66db7e2400" t o 3 \*- bottom part of screen

I—character set  
number

Look at the next FOR-TO-NEXT loop beginning on line 20 and ending on line 40. These three lines are functionally identical to the three statements above.

As with the Gl mode, moving the UFO in the Gil mode is also very easy with the following. How about moving the UFO from the bottom of the screen to the top? Let's see how it’s done.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | pr i nt | "isjiaii" | ◄ | - control | codes |
| 20 | for 1 = | 1 to | 7, |  |  |
| 30 | stchr | "001.8 | 3 c66d b7e24 0 0" | | to &7F,I |
| 4\*0 | next I |  |  |  |  |
| 50 | for Y- | 23 to | 0 step- | -1 |  |
| 60 | P r i n t | c u r s o | r(12,Y)? |  |  |
| 70 | s l eep | 6, 1 |  |  |  |
| 30 | c l s |  |  |  |  |
| 90 | next Y |  |  |  |  |

Another method, far more efficient, replacing lines 20 to 40 follows:

stchr "00183c66db7e2400" to .?,7f,7

I—sets the pattern code for all three screen parts

You may be wondering why the character characteristics of all three Gil screen portions must be initialized. The advantage is to be able to construct three different characters (one per screen portion) using the same code. This immediately triples the number of constructable characters to 768. There are many potentially exciting applications. An example game application that quickly comes to mind is decreasing the size of a falling meteorite as it passes from the upper through the middle part and then into the lower portion of the screen.

* 1. Coloring a Character

The COLOR and STCHR statements work well together to color characters and, as with the Gl mode, the BOOL statement is used to color the backlight plane.

• Using the COLOR statement

One notable difference when coloring characters in the Gil mode is that, unlike the Gl mode, each character in each screen portion can be colored separately. (Do you remember the Gl mode colors eight characters at the same time for the entire screen?) Naturally, all characters can also be colored the same color using the following program.

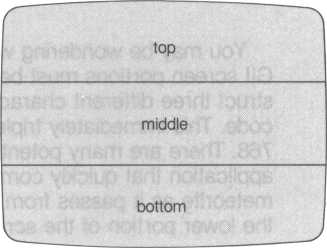
10 print "ili 13!?" ■+ control codes

20 for 1=0 to 255 30 color I , **.\*80 40** next 1

red characters

• Using the STCHR statement

The STCHR statement accesses each character set within each of the three screen portions. But when specifying the color of a character within a screen portion, the following values apply for the character set number.



Gil screen mode and character set numbers 4,5,6

The syntax of the STCHR statement when coloring characters in a particular part of the screen is summarized here.

STCHR “xxxxxxxxxxxxxxxx” TO &NN,

4: top part 5: middle part 6: bottom part 8: entire screen ASCII code of character color code of character

The character’s ASCII code is by now a familiar concept.

The last value, I, is also quite simple; its value is dependent on the screen portion that a particular character is to be displayed on. For example, if a character is defined for the middle part of the screen, it uses character set 2; using the STCHR statement to color it would then use 5 (middle part of screen) for its I value; an ‘8’ colors characters associated with the ASCII code in all three screen parts.

The concept for the color code is a bit more involved. Let’s try an example. Assume we construct a UFO and assign it ASCII &7F in the top portion of the screen (character set 1 —the ‘4’ below signifies coloring of a character in the top third of the screen). We’ll again use STCHR, except this time we’ll use it to color our UFO.

stchr \*• **008a808057505000" to 67f , 4**

If we divide our UFO into our familiar 8 by 8 dot matrix and look at the following illustration, it’s easily discernable that each rank can be individually colored. The first ‘00’ makes rank 0 invisible, the '8A' colors rank 1, the first ‘80’ colors rank 2, and so on up to rank 7. Try it out. Try other colors, virtually allowing any hue can be painted by mixing rank colors.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | r° |  |  |  |  |  |  |  |  | = 00 |
|  | -1 |  |  |  | • | • |  |  |  | = 80 |
|  | -\*2. |  |  | • | • | • | • |  |  | = 80 |
|  | -+3 |  | • | • |  |  | • | • |  | = 80 |
|  | -►4 |  |  |  |  |  |  |  |  | = 57 |
|  | -\*5 |  |  |  |  |  |  |  |  | = 50 |
|  | -►6 |  |  | • |  |  | • |  |  | = 50 |
|  | -\*7 |  |  |  |  |  |  |  |  | = 00 |
|  |  | | |  |  | color of character -J ^-character background color | | | | |
| color is set for each rank | | | |  | (color of dark pixels) (color of light pixels) | | | | | |

A rank may also be referred to as an “image,” each image consisting of a 1 by 8 dot matrix.

0 1 2 3 4 5 6 7

t

size of one image  
(1x8 dots)



1. CHARACTERS IN THE TEXT MODE

As mentioned earlier, the text mode is entered with the CTRL T key combination. Since the size of a text mode character is only 6 by 8 dots, more characters can be displayed per screen. But because of this, the text mode is not appropriate for graphics since the right two columns of dots of each pixel have been removed to accomodate more characters. (Although this does not prevent us from using the text mode for graphics.)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 1 2 3 4 5 6 7 | | | | | | | |  |
| 0  1  2  3  4  5   1. 7 |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| i | | | | | |  | |
| t—Two columns of dots omitted Size of text mode characters ^rom \*ext m°de characters | | | | | | | | | |

* 1. Setting the Pattern Code (Dot Matrix)



The STCHR statement is used virtually identically to the Gl mode although the rightmost two columns of pixels will not be displayed (6 by 8 dot matrix rather than an 8 by 8 dot matrix). However, it’s good policy to specify the rightmost two columns in your pattern codes anyway. The reason is that you can retain one pattern code for all screen modes. Easier to remember, harder to make mistakes.

* 1. Coloring a Character

Characters can be colored in the text mode with the FCOL statement. But the background color of characters cannot be changed, it’s always assumed to be transparent (color code 0).

Enter the text mode with a CTRL T key combination. Then type in the statement below followed by a RETURN key.

t'col s-05

All characters on the text mode screen will change to blue (color code 5). There is no provision for several colors in the text mode.

* 1. Coloring the Backlight Plane

As with the Gl and Gil modes, use the BCOL statement to designate the backlight plane color. The following sets up black characters with a blue background.

i 0 p r i n -t " si" control code T

20 f CO l ■:?<() 1 30 bcol &05



1. MULTI-COLOR MODE

The multi-color mode, entered by pressing the CTRL and Q keys simultaneously, cannot display text characters. Instead, it displays tessellated graphics patterns given the patterns' corresponding ASCII codes 32(&20) to 255(&FF).

A graphics character in the multi-color mode is called a semi-graphics character whose size is a 4 by 4 dot matrix (versus a full-graphics character in the Gil mode).

After beginning execution, give the next program about ten seconds. Beautiful (if you’re using a color monitor) line by line mosaics will be created on your screen. Each distinct tessellated pattern consists of four lines and six patterns. Be sure everything is spelled correctly and nothing is missing from this program or it may not work.



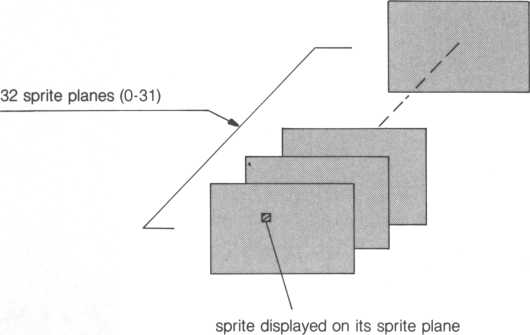
|  |  |  |
| --- | --- | --- |
| 10 | P r i n t | " IMS" control codes |
| 20 | for I | =32 to 255 |
| 30 | for Y | =0 to 23 |
| 40 | for X | =0 to 31 |
| 50 | pr i nt | cursor <X>Y)‘chr$<1)5 |
| 60 | n ex t | x |
| 70 | next | Y |
| 80 | next | I |
| 90 | P r i n t | "9" |

**CHAPTER**



1. WHAT IS A SPRITE?

Sprites are special characters especially suitable for games since they are easily manipulated by software to do a variety of stunts. The M5 system has a maximum of 32 movable sprites. Each rides independently on its own sprite plane, taking full advantage of their own coordinate systems.

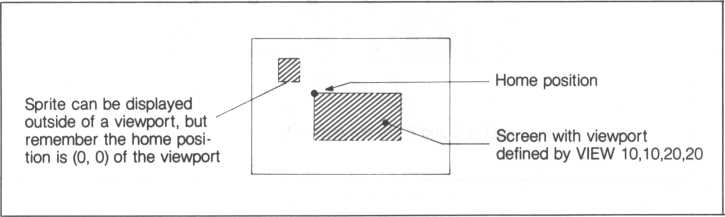


Sprites are numbered consecutively and hierarchically from 0 to 31, with sprite 0 having the highest priority. This is important since a higher priority sprite sharing a screen location with a lower priority sprite causes the lower priority sprite to be hidden.

The size of each sprite is usually an 8 by 8 dot matrix. However, various BASIC-G techniques (MAG and JOINT statements) enable construction of 16 by 16 dot matrix or even larger sprites.

* 1. Sprites and Viewports

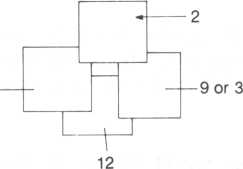
A VIEW statement has no effect on sprites. So sprites can be displayed either within or outside of a viewport. The backlight plane, however, is hidden wherever a viewport is allocated.



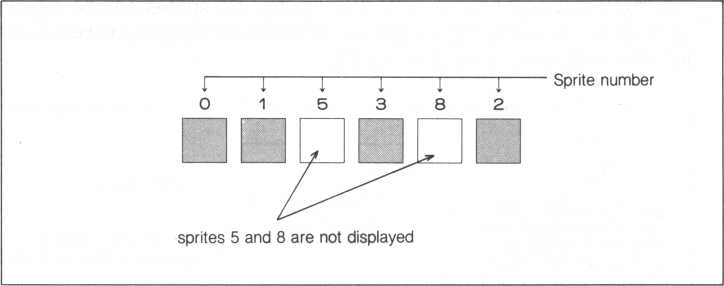
* 1. Hierarchy of Sprites

When sprites overlap, lower priority sprites will not be displayed. For example, if four sprites overlap as below, sprites 3 and 9 cover sprite 12 where they slightly overlap. Sprite 2, being the highest priority sprite, in this case, covers portions of sprites 3 and 10.

3 or 9 (doesn’t matter for our discussion here)



If move than five sprites are set contigously side by side, only the four highest priority sprites are displayed. In the next figure, sprites 5 and 8 will not be displayed.



* 1. Sprites and the Four Screen Modes

Sprites can be displayed in the Gl, Gil and multi-color modes. They cannot, however, be displayed on text mode screens.

The following examples are carried out in the Gl mode. When writing BASIC-G statements, a:

ft i n t "IsjgLi"\*\*-press CTRL SHIFT and U, S and L keys at the same time

statement changes the screen to the Gl mode. Look in Appendix A for a summary of control code functions. For more exercise, go onto the next few examples in the Gil and multi-color modes.

* 1. Let’s Make and Use a Sprite

To create a sprite, several steps need to addressed.

1. Create and assign a sprite number to the sprite (actually its ASCII code)
2. Specify a location for the sprite
3. Color the sprite
   1. Create and Number a Sprite

Let’s define a sprite and assign it an equivalent ASCII code. We also need to assign it a “sprite code” (from 0 to 31) using the SCOD statement.

**s c o d 0 ? 2:41**

I equivalent ASCII code

' sprite number (0 to 31)

The SCOD statement above assigns sprite code 0 to the sprite having ASCII code 41 (hexadecimal), which in ASCII is the way to describe the alphabet ‘A’.

Once a sprite is assigned with the SCOD statement, its ASCII representation is longer used for the keyboard character set. For instance, the above SCOD statement assigns ASCII &41 to sprite 0; the letter ‘A’ (which is also represented\* by ‘41’) is not altered .to the sprite character, its keyboard representation is retained. The result is two different characters, one is the sprite and the other the keyboard character. An exception is the Gil mode— since video RAM is shared by sprites and characters, restrain from using the same ASCII representation frequently. The program below assigns sprite 0 (UFO) ASCII &41.

10 stchr "00183c66db7e2400" to &41..0 20 seed 0> **&4-1**

After executing this program, &41 represents sprite 0. Type the keyboard character ‘A’, notice it really is an ‘A’ and not the UFO sprite. Both the ‘A’ and the sprite share ASCII &41 in different character sets.

* 1. Pixel Coordinates

Without an explicit BASIC-G statement to display a sprite, it’ll remain hidden. The LOC (LOCation) statement works nicely to display sprite 0.

toe 0 to 120?100

1 location (pixel coordinates)

I sprite number (0 to 31)

Sprite number 0, which is actually the alphabet ‘A’ (&41 is the standard representation of ‘A’), is displayed near the middle of the screen since the (120,100) pixel coordinates designate a unique pixel near the middle of the screen.

A pixel is the smallest accessible display on the screen. Once colored, pixels can be combined to form characters dr pictures. And since each screen can display 24 lines of 32 characters (each consisting of a 8x8 dot matrix), our new coordinate system using pixels turns out to be 256 pixels by 192 pixels. (Each character is an 8x8 dot matrix.) But since we number from zero, pixel coordinates can vary up to 255 x 191.



Remember our PRINT CURSOR statement that used character dependent coordinates? As it turns out, the PRINT CURSOR statement tends to be a bit ragged when moving or locating sprites because it positions sprites directly on character coordinates.

The LOC statement, on the other hand, permits locating a sprite on any pixel address, resulting in complete movement and location flexibility. Looking at the figure below, notice pixel coordinates can take on values from -32768 to 32767 in any direction. Since the size of the screen is only 256 by 192 dots, anything located outside of this range cannot be seen.

(-32768, -32768)

\_ sprites may be located outside of the screen with the LOC statements, but they will not be visible

(32767, -32768)

|  |  |  |
| --- | --- | --- |
| (-32, | -32) | (255, -32) |
|  | (0, 0) | (\*55^0) J |
|  |  | |
|  | (0, 191) | (255, 191) |
| (-32, | 207) | (255, 207)  t |

Screen displayed , when not cut by a viewport

Sprite displayed at these coordinates is not visible

( 32768, 32767) Sprites may be displayed in this (32767, 32767)

range using the LOC statement



1. Coloring Sprites

The SCOL statement colors sprites. Our example sprite, the letter ‘A’, is colored red with the following (since we earlier assigned it to be sprite 0). For sprites, only the character color can be changed (the pixel background color cannot be altered).

% c o l 0.' & 0 3

I color code (&08 is red)

number of sprite to be colored

1. Erasing Sprites

Erasing sprites is even easier than creating them. One simple command is all you need.

erase 0

I sprite number

Type in and run the following program. It’ll help us to better understand sprites.

10 print "sIMI"

|  |  |  |  |
| --- | --- | --- | --- |
| 20 | erase |  |  |
| 30 | f o r I | = 0 | to 7 |
| 40 | s c o d | I ? & | 41 + 1 |
| 50 | SCO l | I ? | 07+1 |
| 60 | loc I | to | 100 |
| 70 | next | I |  |
| 80 | f o r I | = 0 | to 7 |

90 sleep 60?1 100 erase I 110 next I 120 end

But what do all these lines do?

line 20—erases all sprites

lines 30 to 70—creates, assigns colors, and positions 8 sprites numbered 0 to 7. Since ASCII codes &41 to &48 are used (line 40), the sprites are the letters ‘A’ to ‘H’. Also note each sprite is assigned different colors (line 50) and located at different locations (line 60). Even if you have a black and white television (or green monitor), don’t omit the SCOL statement, otherwise your sprite will be colored transparent by default (you won’t be able to see it).

lines 80 to 110—waits a fraction of a second (line 90), erases the first sprite (line 100), waits a fraction of a second, erases the next sprite and so on until all sprites are erased.

Some more examples.

erase 4 Erases sprite 4

erase 0.- 3 Erases sprites 0 and 3

1. UFO Sprite

In this section, we’ll create a sprite identical to our UFO character.

So, let’s:

1. Define the pattern code for the sprite. Our dot matrix for the UFO is ‘00183C66DB7E2400’. If you’ve forgotten why, look at section 6.1.
2. Assign an ASCII code to the sprite. Since we used &7F before, let’s use it again.
3. Using the STCFIR statement, combine steps 1 and 2 (assign &7F to our UFO’s dot matrix pattern code).

STCHR “00183C66DB7E2400’’ TO &7F,0

i 1

Whenever defining sprites,

always use 0 for the character set number

The following program fits our needs nicely. Type it in and try it. Is our UFO sprite displayed where you expected?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | P | r | int "l! | ilSIi" |
| 20 | s |  | chr " | 00183c 66 d b 7 e24- 0 0 " |
| 30 | s | c | o d 0 - | &7F |
| 40 | h | c | o l 0 - | M>8 |
| 50 | l | 0 | c 0 t | o 120-100 |
| 60 | 8 | n | d |  |

1. Increasing the Size of a Sprite

The sprites we’ve been exercising are 8 by 8 dot matrices. Using the MAG statement, they can be expanded twofold to 16 by 16 dot matrices. If we add line 45 to our program in section 10.9, our UFO sprite’s size is doubled. Try it out.

45 mag t

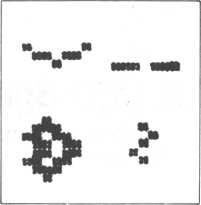
I— signifies a 16 by 16 dot matrix

Our enlarged UFO does not have high resolution. Imagine our original UFO is enlarged with a magnifying glass and you have a MAG 1 sprite.

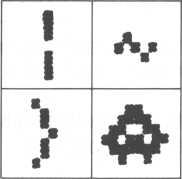
The MAG statement can alter a sprite one of four ways. This time, use:

45 m3g 3

After running our program again, it turns out we actually see four figures that look similar to the figures below. Why?



If we remember we assigned the ASCII code &7F to our sprite, it soon becomes evident there is a relationship between our UFO sprite (which is the character at the lower right) and the other three figures. The numbers in parentheses are the decimal number equivalents of our hexadecimal ASCII codes.



|  |  |
| --- | --- |
| S7C  (124) | a 7 E (126) |
| a 7 d (125) | a 7 F (127) |

or, put another way:

|  |  |
| --- | --- |
| a 7 c  (124) | a 7 e (126) |
| a 7 d (125) | a 7 f (127) |

»

Sequential

relationship

0

2

**1 3**

How do we determine which square to put what? The answer lies in the MOD function. What is the MOD function? Basically, the MOD function divides one number by another, the remainder is the MOD function. For example, if we divide the value of our ASCII code, &7F, by 4, the remainder is 3, which is our answer. Therefore square number 3 will contain our sprite. BASIC-G provides a handy function to calculate the MOD of any number.

print 127 mod 4 which is identical to:

Pr i nt ■S:7f mod 4

The answer, 3, is printed on your screen. Try it.

The other squares contain figures corresponding to their own ASCII codes. To ascertain the other ASCII codes, assume square number 3 corresponds to our sprite’s ASCII code. Decrease this ASCII code by one; the answer corresponds to the ASCII code for square 2. Then decrement square 2’s ASCII code by one which results in square 1’s ASCII code. Likewise, square 0’s ASCII code is one less than square 1’s.

This relationship between the MAG and MOD function applies to ASCII codes between 32 and 255 (decimal), or &20 and &FF (hexadecimal), inclusive.

Once used, the MAG statement changes the size of all sprites. For example, we cannot assign one size to sprite 0 and another to sprite 1.

We’ve used MAG 1 and MAG 3, what else can we do with the MAG function? The table below lists the possibilities.

**Example of display  
for ‘A’, or & 41**

**Notes**

Character dot matrix

Enlarged dot matrix

MAG 0

N

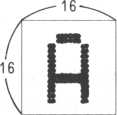
8x8

dots

8x8 dots

* Default mode when power is first supplied
* Sharp picture

MAG 1



16x16

dots

16x16

dots

16 x 16 dots 16x 16 dots

* One large picture can be created by combining four smaller pictures
* Hazy picture

MAG 2

|  |  |
| --- | --- |
| 3 | B |
| fl1 | C |

8x8

dots

16x16

dots

* Large sprite can be easily created by combining four characters
* Sharp picture

|  |  |  |
| --- | --- | --- |
| .\*  \*1 | ) | B |
| fl | | C |

16

MAG 3

16x16

dots

32 x 32 dots

* One large picture can be created by combining four smaller pictures
* Hazy picture

1. Moving Sprites Around

Several methods are available to move sprites.

1. Moving sprites with the LOC statement

Look below to move a sprite using the LOC statement.

10 print "fJtaiK"

**20 stchr "00183c66db7e2400" to &7F? 0 30 scod 0»&7F**

40 scol 0,&03 45 rrtag 1

47 for X = 0 to 255 50 loc 0 to X?100 60 sleep 1 ? 1 70 next X

30 end

When run, X in line 50 increments by one each time the FOR-TO-NEXT statement is executed. From this, the first number of the dot matrix address (horizontal direction) also increases by one, up to 255. It’s easy to see that since the horizontal component of the address changes from 0 to 255, our UFO moves left to right across the screen. The SLEEP in line 60 suspends our program (SLEEP) inbetween moves for about half a second. Its purpose is to slow down the movement of our UFO to allow us to see its movement. Sprites can move a greater deal faster than we can perceive on the screen.

One note of caution—if (0,0) is altered with a VIEW statement (to create a viewport), the coordinate system above shifts up, down, left or right depending of the new home position.

1. Moving sprites with the MOVE-IN-TO statement

Let’s- reuse our most recent program using the LOC statement to move the UFO sprite. This time, remove lines 47, 60 and 70. Then change the ‘X’ in line 50 to a ‘0’ (zero) and add a new line 60.

10 print :'L\*J£31S"

**20 stchr "00183c66dt«7e2400" to &7F, 0 30 seed** Q>&7P **40 scot 0, 7:08** 45 mag 1

50 loc 0 to X,100

60 move on-\* turn on MOVE capability

70 move 0 in 0 to 200?100?1

1. 1 1 I speed (1 is the fastest)

' destination pixel address

post number

sprite number

When run, this program turns on the capability to move a sprite in line 60, and the UFO is set in motion in line 70 to pixel coordinates (200,100). Our program ends and the BASIC-G Ready prompt is displayed, even before our UFO stops moving.

To verify this, add the following statements to our program.

80 I -• 0

**90 1=1+1**

This loop increases and prints I during each interaction

100 pri nt cursor(10,10)‘I \* 110 goto 90

Run this program and watch the counter. The UFO sprite will start moving before the counter begins. It will continue to move while the counter increases, regardless of the counter value. It’ll stop while the counter continues. Clearly, execution of our program and the continued movement of sprites are independent processes. To stop the counter and’return to the Ready prompt, press the SHIFT and RESET keys simultaneously.

One important point to keep in mind.

• When setting the destination pixel address, be sure the difference between the cursor’s coordinates and the sprite’s destination coordinates is less than 255 (in both the x- and y-directions). For instance,

10 loc 1 to -1,3 20 move 1 to **255,** 191

The difference in the x-coordinate is 255-( -1) = 256, which should not happen.

The post number can be thought of as giving a sprite the ability to move. Without having an assigned post number, a sprite cannot move. There are 12 posts (0 ~ 11); each post can have assigned only one sprite. Assigning a sprite to a post number greater than 11 yields an error.

move 1 in 2 to **1.00, 120**

post number

sprite number

The above statement assigns post 2 to sprite 1. If “IN post-number” is omitted from a MOVE statement, a sprite is automatically assigned the post number equal to its sprite number. For example,

move 0 in 0 to 140>20

is identical to

move 0 to 140>20

I post number omitted

1. Moving sprites with the MOVE-IN-STEP statement

The MOVE-IN-TO statement allows movement of a sprite toward a target address. The MOVE-IN-STEP, on the other hand, moves a sprite along an (X,Y) vector specified by us. Unlike the MOVE-IN-TO statement, the MOVE- IN-STEP does not provide us with the capability to stop a sprite at a particular address. Once given the vector, it’ll continue to move (out of sight).

move i in j step x-\*d i rect i on> y-d i rect i on> s

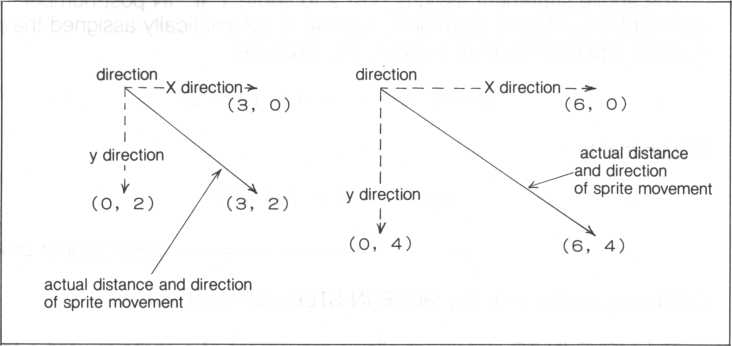
speed (1 to 255)

(X, Y) vector

post number (0 to 11)

sprite number (0 to 31)

As in geometry, specifying a MOVE-IN-STEP vector is done with two coordinates (x direction, y direction). Therefore, given the origin of the sprite when the MOVE-IN-STEP statement is executed, the vector defines the direction and the traveling distance simultaneously. For example, the vectors (3,2) and (6,4) point in the same direction but their traveling distances are different.



Don’t discard our program yet. Modify it so that it’s identical to the following. Note lines 25 and 55 have been added to provide a FOR-TO-NEXT loop. To take advantage of this, lines 30 and 40 are changed to alternate between two sprites, numbered 0 and 1. The color in line 40 is also modified as is the y coordinate in line 50. Obviously, for our example, line 60 has been replaced by a MOVE-IN-STEP statement to control sprite 0’s movement; line 70 controls movement of sprite 1. Notice the speed settings in statements 60 and 70-are identical.

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | FT i nt | "ilHS" |  |
| 20 | s t c h r | 1100183c66 d b 7 s?240 0 " t | o t<7F> 0 |
| •~> cr | for I | = 0 t o 1 1 |  |
| 30 | s c o d | I, ?<7F | loop sets ASCII |
| 40 | SCO l | I,1+1+5 | code &7F to |
| 45 | ma g 1 |  | sprites 0 and 1 |
| 50 | l DC I | t o 0 .< 0 |  |
| 55 | next | I 1 |  |
| 60 | move | 0 in 0 step 6> 4?1 | sprite 0 movement |
| 70 | move | 1 in 1 step 3i 2> 1 | — sprite 1 movement |

speed (sprites 0 and 1 are identical)

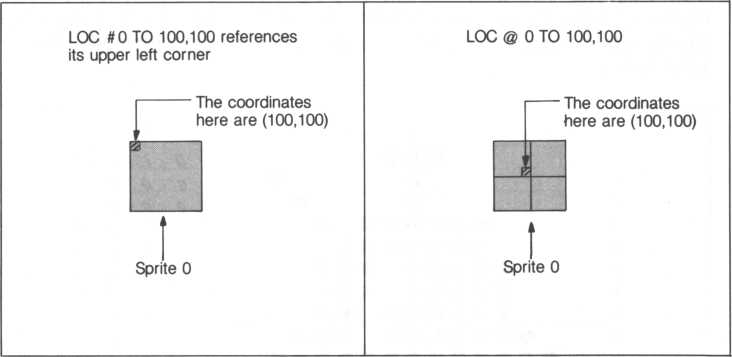
When running, the blue (sprite 0) and yellow (sprite 1) UFO sprites travel in the same direction as specified by their vectors (vectors point in the same direction). The sprite speeds are also set identically, but one appears to move faster since the vectors are different. If the speed setting are omitted (including the preceding comma), it is set to 4, the default speed. Also keep in mind that when many sprites are set to speed 1, the processing speed of our software will slow down a bit.

The color assignment in line 40 looks a bit complicated. It’s not. When I equals 0, sprite 0 is assigned color code 5 (blue), or 1 + 0x5. Likewise, when I equals 1, sprite 1 is assigned color code 10 (yellow), or 1 + 1x5.

Line 45 (the MAG statement) can actually be moved out of the FOR-TO- NEXT loop since a MAG statement affects all sprites until a different MAG statement is executed. We could have inserted it prior to the loop as line 22. A reminder, the MOVE-IN-STEP cannot stop a sprite at a specified location. We can only specify the direction. If the sprite should be stopped somewhere, use the MOVE-IN-TO statement.

1. Sprite Coordinates

The sprite’s coordinates can be referenced at two places using the LOC statement, (1) at its upper left corner, and (2) at its center.



When a ‘ # ’ is used, the sprite’s upper left corner is placed at the specified coordinates. Using a will place the center of a sprite squarely on the target coordinates.

A hint—when setting a sprite in motion with a MOVE statement, be sure the differences between the target and destination coordinates are less than 255, otherwise sprite movement may be awkward. For example,

l o c 1 "t o 3 ? 3 move 1 to 259-191

In the above two statements, the difference between the x-coordinates is 256 (259-3). This should be avoided.

1. Creating a Jumbo Size Sprite Using a Pattern Code



We’ve seen how to create a large 16 by 16 dot sprite with the MAG 3 statement. But alas, using the MAG function can be a bit unnerving to create a good looking large sprite. So let’s construct a good resolution jumbo size UFO using a familiar method (its pattern code).

Refer back to section 5.2 if you’ve forgotten the significance of pattern codes. Creating 16 by 16 pattern codes is almost identical to constructing 8 by 8 dot matrix characters. The only significant difference is that the dot matrix is larger. In fact, four times larger since it consists of four squares. We therefore require four pattern codes. Look at the semblance of a UFO below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 01 23456789ABCDE F | | | | | | | | | | | | | | | |  |
| 0  1  2  3  4  5  6  7   1. 9 A B C D E F |  |  |  |  |  |  |  | • |  | • |  |  |  |  |  |  | 0 <7 i / i 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | • |  | 1 o, oS’ , |
|  |  |  |  |  |  |  |  |  |  |  |  |  | • |  |  | 2 P , <7 , 8i tL. |
|  |  |  |  |  |  |  |  |  |  |  |  | • |  |  |  | 3 o , t , e. , g |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 4 o, 7 , ? , 0 |
| — |  |  |  |  | • | • |  |  |  | • |  |  |  |  |  | 5 o, F S ,8 |
|  |  |  |  | • | • |  |  |  | • |  |  |  |  |  | 6 <? . F 0 i 8 |
|  |  |  |  |  | • | • |  |  | • |  |  |  |  |  |  | 7 0 , t, P , 8 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 8 / . F o . c |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 9 3i f i p, e |
|  |  | • | • |  |  |  |  |  | • |  |  |  | • | • |  | A $ , 9 , C , E |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 8 3 1 f, F 1 £ |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | c / , P , V 1 C |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D Q if ,f ,2  6 o 1 ^ , / 1 o |
|  |  |  |  |  |  | • |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | • | • |  |  |  | • |  |  |  |  |  | F (>,6,3,0 |
|  |  | | | | | | | | | | | | | | | | |

Its pattern code is listed below (spaces have been inserted making it easier to read—take them out of your BASIC-G programs). We’ve blocked out four distinct pattern codes corresponding to squares 0 to 3 of our 16 by 16 dot matrix.

1. ... “01 00 00 07 07 OF OF OF”
2. ... “1F 3F 39 3F 1F OF 02 06”
3. ... “CO 82 84 E8 F0 B8 B8 D8”

3... “DC FE CE FEFCF8 10 30”

Since our jumbo UFO displaces four-squares, let’s allocate four ASCII codes, &7C to &7F for squares 0 to 3 respectively.

We’re going to depart from our previous program, so it’s a good opportunity to take a break here.

To start fresh, type in NEW followed by a RETURN key to delete our old program. Then type in ERASE followed by a RETURN key to erase sprites 0 and 1 (the ERASE in line 30 below is redundant in this situation, but it’s a good idea to include it in all your games).

Now we’re ready to enter our program. Like it? It randomly moves our jumbo UFO by manipulating only the sprite assigned ASCII code &7C.

**lOrem jumbo (mag 3) ut'o**

**20 p r i n t "** SIMS**"** control codes

**30 erase**

1. **gosub $STCHR 50 gosub $SP.SET 60 gosub $SP.MOVE 70 end**

**100$STCHR**

**110 s-tchr " 01000007070 f Of Of " to &7C, 0 120 s t ch r " 1 f 3 f 393f 1 f 0 f 0206 " t o** &7D**> 0 130 stchr "c08284e8f0b8b8d8" to** &7E,**0 14 0 s t c h r " d c f** e **c e f e f c f 81.030" t o -&7F0 150 return 200\*SP.SET 210 mag 3**

**220 seed 0>** &7C

**230 scot (V 205 240 loc 0 to 100>100 250 return 400$SP.MOVE**

1. **0 X = r n d (2:55) s Y = r n d (191)**

**420 move 0 to X>Y>1**

**430 if status(0)=0 then goto $SP.MOVE 440 goto 430 450 return**

Some main points of this program follow.

lines 10 to 70—main body of our program

lines 100 to 150—subroutine SSTCHR sets up our jumbo size UFO

lines 200 to 250—subroutine SSP.set sets up several sprite parameters

lines 400 to 450—subroutine SSP.move moves our sprite

lines 430 and 440—if our sprite has stopped moving, STATUS(0) = 0, then force it to move again. But if it’s still moving, don’t disturb it (execute line 440 which brings us back to 430) and check again later.

1. Subroutine SSTCHR

Subroutine SSTCHR constructs our jumbo UFO by setting up the four necessary squares.

1. Subroutine SSP.SET

Subroutine SSP.SET defines necessary parameters for our jumbo UFO sprite. Look at line 220, it defines the sprite we constructed in subroutine SSTCHR to be sprite number 0. And since we will be using MAG 3 to define the sprite size, we only need to define square 0 (ASCII code &7C).

it is necessary to onjy specify the ASCII code of square 0

|  |  |
| --- | --- |
| O  (azc) | 2  (8 7E) |
| 1  (a 7D) | 3  (a 7 f) |

1. Subroutine SSP.MOVE

Using the MOVE statement, we’ll move our sprite in random directions on the screen. The vector is set by the random number generator in line 410. (RND, a random number generator generates random numbers—the numbers within the parentheses define the the maximum desired random number. The X coordinate and Y coordinate maximums reflect the bounds of the displayed screen.)

After the RND function generates the random (X,Y) coordinates, line 420 moves the UFO sprite along the vector defined by the random coordinates. And as we saw previously, a MOVE starts the sprite moving. The next BASIC-G line is then executed while the sprite is moving. In this case, line 430 checks whether\*the sprite is still moving (if it’s reached the target coodinates). If it’s still in transit, STATUS(O) will equal 1, causing execution of line 440. Line 440 returns to line 430 to check STATUS(0) again. This condition is repeated until STATUS(O) equals 0 (sprite has stopped moving). When the sprite stops moving, STATUS(O) equals 0 and the GOTO in line 430 jumps back to SSP.MOVE, line 400.

Once subroutine SSP.MOVE is entered at line 400, you may have guessed the RETURN in line 450 will never be executed. It’s been included to avoid an error (ERR 8) since BASIC-G expects a RETURN at the end of every subroutine.

This will continue indefinitely until the SHIFT and RESET keys are pressed simultaneously.

10.14 Creating a Jumbo Sprite Using the JOINT Statement

Change the program from section 10.13 as follows. Added or modified parts are delimited by enclosed boxes.

**10 r em j urnb o (ma g 3) u f o**

20 print " LiiHH" control codes

**■30 erase 40 gosub TSTCHR 50 gosub $3P.**SET **55 g o s u b $ SP. J' OI NT 60 gosub $SP.MOVE 70 end 100$STCHR**

110 stchr "01000007070f0f0f" to ?.-7C, 0 1\*20 stchr " 1 f3f393f 1 f Of0206" to &7B> 0 130 stchr "c08284e8fObSbSdB" to -!<7E, 0 140 stchr "dcf ecef et"cf81030" to .&7F, 0 150 return 200$SP.SET

|  |  |  |
| --- | --- | --- |
| no | ma g | 1 |
| ;i5 | for : | S = 0 t o |
| ?20 | scod | O ? ~7c |
| J30 | SCO l | S? A 05 |
| i 3 T j | next |  |

240 toe 0 to 100? 100 250 return 300$SP.JOINT 310 .j o i n t 1 t o 3 > 4 320 j o i nt 3 t o 2 > 3 330 joint- 2 to 0? 2 340 return 400$SP.MOVE

41 0 X - r n d (255> ’• Y = r n d (191)

420 move 0 to X?Y> 1

430 if status(0)=0 then goto $8P.MOVE 44O g o t o 430 450 return

Some main points of this program follow.

lines 10 to 70—main body of our program. Since we’ve added a new subroutine, line 55 is added to call it.

lines 100 to 150—subroutine SSTCHR sets up our jumbo size UFO

lines 200 to 250—subroutine SSP.SET sets up several sprite parameters. Notice this modified subroutine defines all four squares Also, notice we’re using MAG 1.

lines 300 to 340—subroutine SSP.JOINT joins the four 8 by 8 dot sprites defined by subroutine SSTCHR

lines 400 to 450—subroutine SSP.MOVE moves our sprite

1. Subroutine SSP.SET

Since MAG 1 is set (MAG 0 is also applicable), we cannot easily create a UFO by simply accessing square 0 as before. When using the JOINT statement, we must explicitly define each square (each sprite). Look at the FOR-TO-NEXT loop of lines 215 through 235. As S changes from'0 to 3, lines 220 and 230 use S to define the sprite number (0 to 3), ASCII code (&7C to &7F) and color (&05) of each sprite.

<2) Subroutine SSP.JOINT

The three JOINT statements join the four squares together.

|  |  |  |  |
| --- | --- | --- | --- |
|  | joint number 2 | |  |
| the last sprite is joined implicitly with the MOVE ~> statement in line 420 | 0  (a 7 c) | \* 2 (&7E)  I | joint number 3 |
|  | 1  (a 7 d) | $  - 3  (S7F) |
|  | joint number 4 | |  |

The general format of the JOINT statement is:

JOINT sprite-number TO next-square joint-number

Let’s start from square 0. Going counterclockwise from square 0 (sprite 1), we see it’s joined to square 2 on joint number 3’s side. So, line 310 is appropriate. Likewise, square 2 is joined to square 1 on joint number 2’s side (line 320). Square 1 is treated similarly. Notice square 0 is not yet defined, it’ll be done implicitly when the MOVE statement is executed in line 420.

We could have started at squares 0, 2 or 3. Or we could have also gone clockwise. The important point to stress is the last square is joined by a MOVE statement.

1. Detaching a Jumbo Sprite Joined with the JOINT Statement

The JOINT statement is not only used to construct a jumbo sprite, it can also be used to detach a jumbo sprite. For our example, we’ll create a jumbo sprite made up of four small UFOs (the group of four small UFOs can be treated as if they were one large sprite).

Type in NEW and an ERASE after the BASIC-G Ready prompt. Then type in the next program.

**10 rem d et ach j umbo(joint) u f o 20 els:erase 30 dim X(3)?Y(3)**

**40 gosub $STCHR 50 gosub $SP.SET 60 gosub iSF'. JOINT 70 gosub $SP.MOVE SO gosub $SP. CUT 190 end 200SSTCHR**

**210 st c h r "00183c66d b 7 e2400" t o &7C,0**

**220 return**

**300$SP.SET**

**310 rna g 1**

**320 for SP = 0 to 3**

**330 scod SP? &7C**

**340 scol SP?&Q5+SP\*3**

**350 loc 0 to 0?100**

**360 next SP**

**370 return 400\*SP.JOINT 410 j oi n t 3 t o 2 ,2**

**420 j o i n t 2 t o i j** 2.

**430 joint 1 to 0>2**

**440 return 500$8P.NOME**

**510 move 0 to 255, 100, 2**

**520 if spri te(0, 1)- 100 then return**

**530 got o 52 0**

**600$8P« IJU I**

**610 j o i n t 0,7**

**620 for 1=0 to 3**

**630 X (I) = r n d (255) \*• Y < I) = r n d (191)**

**640 move I t o X(I),Y(I),1 650 next I 660 return**

Some main points of this program follow, lines 10 to 190—main body of our program.

lines 200 TO 220—subroutine SSTCHR constructs our familiar small UFO sprite

lines 300 to 370—subroutine SSP.SET sets up several sprite parameters. Notice all four sprites use the same pattern code (line 330) since all four UFOs are identically shaped.

lines 400 to 440—subroutine SSP.JOINT joins our four 8 by 8 dot matrix UFO sprites horizontally

lines 500 to 530—subroutine SSP.MOVE moves our jumbo sprite until our target coordinates are reached. When reached, the RETURN in line 520 is executed.

lines 600 to 660—subroutine SSP.CUT detaches all sprites from each other (line 610) and sends them off in random vectors.

(1) Subroutine SSTCHR

Subroutine SSTCHR constructs the pattern code for our familiar semblance of a small UFO and assigns it ASCII code &7C.

1. Subroutine SSP.SET

As with subroutine SSP.SET in section 5.10, a FOR-TO-NEXT statement is used to assign necessary parameters for sprites 0 through 3. In this section, every sprite is assigned the same ASCII code &7C (since they have the same shape). However, each sprite is given a different color (line 340).

1. Subroutine $SP.JOINT

Subroutine SSP.JOINT sets up a jumbo sprite by lining up our four small UFO sprites horizontally using the JOINT statement.

|  |  |  |  |
| --- | --- | --- | --- |
| l  0 | i  1 | 1  2 | 3 |
| ,S7C | a7C | a7c | a7C <- |
| - | -> - |  |  |
|  |  |  |  |

ASCII code of each sprite

2

t

sprite number

2

**JL**

2

**JL**

-joint number

first sprite joined using the joined statement

1. Subroutine SSP.MOVE

The MOVE of line 510 moves our jumbo sprite toward coordinates (255,191). The format of the SPRITE statement in line 520 is:

**spri te(x> y)= z**

I— value tested for

1 0 : check y-coordinate value

1. : check x-coordinate value
2. : check ASCII code
3. : check color code

sprite number

Since line 520 contains SPRITE(0,1)= 100, we know we’re waiting for the x-coordinate of sprite 0 to reach 100. Until this condition is satisfied, the loop between lines 530 and 520 is repeated indefinitely. Once satisfied, the RETURN in line 520 returns to line 80 in the main body of our program. In other words, we’re waiting for sprite 0 to reach its destination.



1. Subroutine SSP.CUT

The JOINT statement in line 610 detaches all sprites from each other. The T of JOINT 0,7 indicates detachment of all our allocated sprite joints. If the 7 is changed to a ‘0’ (zero), the sprites are detached one at a time. Try it out for yourself.

After being detached, the sprites are sent off toward random coordinates.

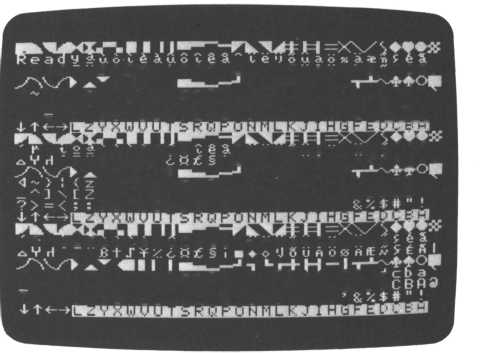
|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |



CHAPTER

11- WHAT IS GRAPHICS?



Your M5 is endowed with a complete graphics capability. But what is graphics? Simply put, graphics draws pictures on your screen in 16 different colors. But how? Remember our discussion of pixels and coordinates in section 3.13? Using the same concept, graphics produces nice pictures.

Type in NEW followed by a RETURN key to erase any leftover programs. Then type in and execute the following program.

10 print "I8MS" Changes the screen to Gil dome

20 g i n i t (press the CTRL and SHIFT keys

simultaneously with the U, R and L keys)

Since we need to be in the Gil mode to use graphics, line 10 changes your screen to the Gil mode; GIN IT in line 20 is necessary to change it to the graphics mode (graphics initialization).

Graphics also works in the multi-color mode. All that’s need is to change to the multi-color mode and execute a. GIN IT statement. Graphics commands are identical for both screen modes, but since the resolution is limited to a 4x4 dot matrix in the multi-color mode, it is not suitable for quality graphics. This discussion will focus primarily on the Gil mode.

Look at your screen. Our familiar cursor is blinking. There’s also another character that looks like the cursor, but it’s not blinking. We’ll discuss how to use these in a moment. But before we do that, press the CTRL and S keys at the same time to momentarily change to the Gl mode. Interesting? Displayed are three copies of the ASCII code (from &00 to &FF) at the top, middle and lower portions of the screen. We’ll take advantage of many of these displayed characters for effective graphics.

**rf&Vrl " B + J¥^cQ36§ i ■♦O'JSuftOfSfilEA/Hfil**

<\*\* 1 I -« ,T+H~I

4 > ! < z y x w m u t s r <3 P o n m T k j i h 9 ♦ e d c 6 a

**\_A]\C2VXWUUTSRQP0Nt1LKJIHGFEDCBfi3 ■'?>—<? :9876543210/.-,+\*)('&** y-AJL'LL

U.O o 6 £. ij. 8 ~ ie'.l0UA05SA\*ijj||

**BtJ¥/. « ■♦o'JSuAbj3afw5| ft I**

’’«■! I -i L r+ H -1

yxwvutsrqponinlkj lhHedcba

**VXWUUT SRQPONMLKJIHGFEDCBP3 9876543210/. -,+\*>< 7**

Y L 2 ^ uo iS^u8t§Si/'teiJou4<)Ka\*tt

4 ^ > 5 <zyxwMUtsr4PonmTkJih9fe« ^IxCZVXWUUTSRQPONMLKJIHGFED

**?>=<; =9876543210/.-,+\*> <**

The Gil screen can also display these characters. However, in the Gil mode, these characters are “colored” transparent (invisible—color code 0) when the GIN IT statement is executed in line 20 above. (It automatically changes to color &0E in the Gl mode. That’s why we are able to see these characters in the Gl mode.)

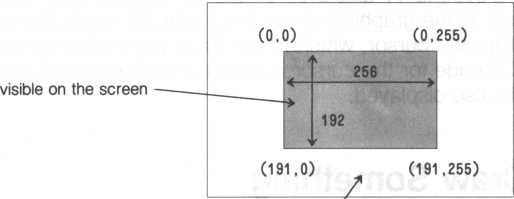
1. The Graphics Cursor

What were the two characters on the screen earlier? The blinking ‘A’ is the graphics cursor. We’ll use it later to describe coordinates on the screen so that we can color and combine dots to make up cohesive pictures.

Since coordinates are central to understanding graphics, let’s review. Look at the figure below.

(32767, -32768)

(-32768,-32768)



(-32768, 32767) not visible on the screen, but (32767, 32767)

the graphics cursor can still be set to these coordinates

As shown above, the screen is divided into an X-Y grid of 256 dots across (x-direction) and 192 dots down (y-direction). By starting at the (0,0) point and specifying both the x- and y-directions, a unique dot can be addressed; this is known as an absolute coordinate since there is no ambiguity about the dot. In addition, relative coordinates can also be indicated by specifying a reference coordinate and then specifying a vector away from the reference coordinate.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Characteristics | Designation Note 1 | Examples |
| Absolute  coordinates | Home position is (0,0) | X,Y | GMOVE 120,100 GMOVE 0,0 |
| Relative  coordinates | Graphics cursor is (0,0) | STEP X,Y | GMOVE STEP 10,20 GMOVE STEP -5,8 |
| Sprite  coordinates | Coordinates of a sprite’s upper left corner | # N | GMOVE #0 |
| Sprite  coordinates | Coordinates of a sprite’s center | @N | @2 |

Note 1: X ... x-coordinate Y ... y-coordinate N ... sprite number

But why is a second ‘A’ displayed on the Gil screen? Well, the Gil screen displays all the graphics characters in the Gl mode. But all are invisible except for the cursor, which needs to be colored to be seen. And since the ASCII code for the cursor is identical to the second, non-blinking character, it is also displayed.

1. Let’s Draw Something

Press the RETURN until you see some Ready prompts. (You’re now in the Gl screen mode, but it’s not a problem.) Add lines 30 and 40 to your program (input in section 11) and try it out.

|  |  |
| --- | --- |
| 10 print " tiliail" •+  20 g i n i t | control codes  rn/^\/p thfi nnnhicr\* oi ir^nr to /n n\ |
| 40 draw 120,100 | 11 lUVt? lilt? yi dpi IlOb L/UloUi IU \U,U/ |

draw a line from the cursor located at (0,0) to (120,100)

A line is automatically drawn from (0,0) to the coordinates (120,100) expressed in line 40. Each time a DRAW statement is executed, a line is drawn from the cursor location to the expressed coordinates. The cursor is then positioned at the end of the drawn line.

A portion of the line is omitted near the top of the line. It’s because the Ready prompt is blocking the part of the line displayed there (the Ready prompt is colored transparent—it’s still a color, albeit invisible). To avoid this situation, let’s change our program to move the cursor to our alternate screen. Press the CTRL and V keys simultaneously to display the alternate screen and add line 50 to our program. (If you lose your cursor to the alternate screen, use the CTRL Y key combination.)

50 print "ST shifts the cursor to the

alternate screen (press the CTRL and SHIFT keys at the same time as the Z key)

Do you remember the CTRL and Z key combination moves the cursor to the alternate screen? Running our program again produces an unbroken line since the cursor is now at the alternate screen.

DRAW forces the graphics cursor to move each time a line is drawn.

If line 40'is changed to

4- 0 d r a w 10 > 20120 > 10 0

1 end point of line

starting point of line

a line is drawn without affecting the location of the graphics cursor.

1. Using the PLOT Statement

A single pixel can be displayed anywhere on your screen with the PLOT statement. The graphics cursor is then moved to the colored pixel’s coordinates. Try the program below.

10 print "HM§" \*\* control codes

20 g i ni t 30 plot 12Q>100 40 end

I

coordinates of dot to be displayed

See it? The small dot in the middle of the screen? Now, instead of displaying only one dot, we’ll display many dots at random coordinates.

|  |  |  |
| --- | --- | --- |
| 10 | pr | i n t " tob " control codes |
| 20 | 9 i | n i t |
| 30 | X - | rnd<255):Y=rnd(191> |
| 40 | pi | at. X,Y |
| 50 | go | t o 30 |

Do you like the outer space effect? Since this program will loop indefinitely between lines 30 and 50, more and more dots will continue to fill the screen. If we let our program execute for an extended period, theoretically, all the dots on the screen will be colored resulting in a solid color screen.

Press the SHIFT and RESET keys concurrently to stop execution of this pfogram.

1. Using the CIRCLE Statement



The CIRCLE statement, naturally, draws a circle. It can also draw an ellipse or a polygon.

1. Drawing a circle

The only information necessary to draw a circle is its center and radius (expressed in number of dots). The following program draws a circle at coordinates (128,96) with a radius of 60 dots. The CIRCLE statement cannot designate the cursor location, so line 30 sets the cursor location and line 40 draws the circle. Drawing a circle doesn’t affect the cursor position

|  |  |  |
| --- | --- | --- |
| 10 | Print | — control codes |
| 20 | g i n i t |  |
| 30 | gmove 128?96 ◄ | — moves the graphics cursor to (128,96) |
| 40 | circle 60 |  |
| 50 | end 1 | — radius of circle (number of dots) |

If you don’t like the cursor displayed on the same screen, line 55 can be added (discussed earlier).

55 print "gj" control codes

1. Drawing an ellipse

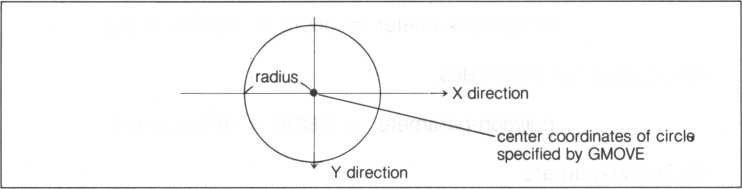
Drawing a circle is nearly as easy as a circle. Rather than describing one value for the radius (as with a circle), specify a radius for both the x and y directions. Change line 40 to:

4-0 c i re l e tb0j 30

I— radius in the y direction



The general case is:



radius in the x direction

1. Drawing a polygon

The next parameter in a CIRCLE statement easily creates a polygon.

Substitute line 40 with the next statement.

40 c i PC le 60» .- 120

I polygon parameter

I— omitted parameter (when omitted, the radius in the y-direction is assumed equal to the radius of in the x-direction)

A triangle is created. How did that happen?

The number of sides in a polygon is calculated by

polygon-parameter = 360/number-of-sides

For example, a triangle has three sides. Thus,

polygon-parameter = 360/3 = 120 (degrees)

An octagon has eight sides.

polygon-parameter = 360/8 = 45 (degrees)

1. Drawing an arc

Do you like pie charts? Change line 40 and rerun our program. It will draw the outer curve of a potential pie chart.

40 c i rct e 60 **??>30? 280**

ending bngle (degrees)

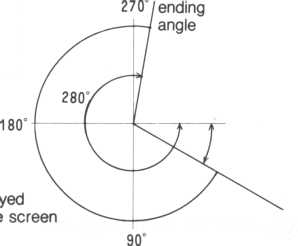
starting angle (degrees)



two omitted parameters

1 radius in the x-direction (dots)

By omitting the radius in the y-direction, we imply the same radii in both the x- and y-directions (since a circle has equal radii along the x- and y- axes). The figure below shows nicely the relationship of the starting and ending angles.



displa on the

— starting point 30°

starting

angle

Since a circle is 360 degrees, the next statement obviously draws a complete circle.

**40 circle 60 ? > ?** 0?360

can be omitted

1. Drawing fans (easy pie charts)

Line 40 can be changed again to easily make a slice (fan like shape) of a pie chart.

**40 ci rele 60? ? j 30?230? ? 1**

The ‘1’ above turns on the function to draw a fan like shape; if omitted or set to ‘O’, the fan function is disabled.

Several statements like line 40 above can create a complete pie chart.

1. Drawing an oblique circle or ellipse

Change line 40 again.

40 c i rcle 60?? ? ? ?20

L oblique angle (degrees)

The result is a circle obliqued 20 degrees. For an interesting effect, try the next program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | pr i | nt | "ISMS" -- | control codes |
| 20 | g i n | i t |  |  |
| 30 | gmo | ve | 128?96 |  |
| 35 | f or | 1 = | 0 t o 360 | step 10 |
| 40 | c i r | c l e | 60? ? ? ? ? | I |
| 50 | nex | t I |  |  |
| 55 | pr i | nt | "Hn | control codes |

1. CIRCLE Statement Summary

CIRCLE statement parameters, separated by commas obviously have a  
necessary sequence. When omitted, the comma separator must be included.

**c i r c l e**

fan parameter oblique angle ending angle starting angle polygon angle radius in y-direction radius in x-direction

1. Using the BOX Statement



Drawing squares and rectangles is easy with the BOX statement.

10 p rint " i!MS'" control codes

20 g i n i t

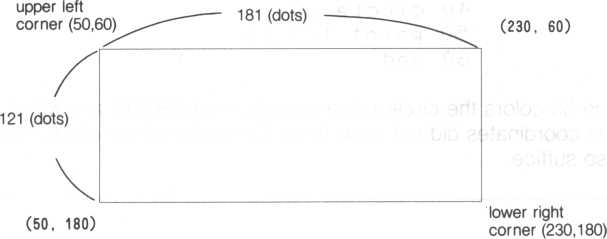
30 box 50>60,230> 180

lower right corner of rectangle

upper left corner

of rectangle

The rectangle encompassing the coordinates (50,60) and (230,180) is displayed. The figure below shows the relationships between the coordinates and rectangle sides.



1. Using the BAR Statement

Let’s draw a solid rectangle using the BAR statement.

**10 pr i n +. "iiMI"** control codes

20 gi ni t

**30 bar 50j60>230>t30**

lower right corner of rectangle - upper left corner of rectangle

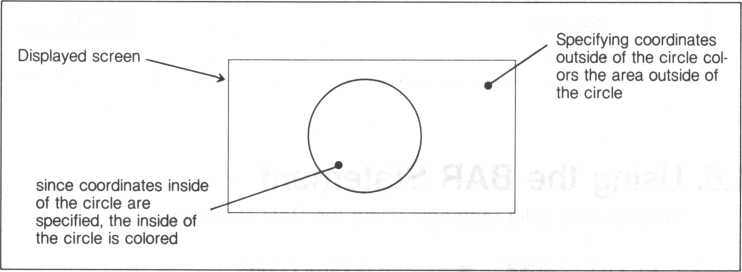
1. **Using the PAINT Statement**

The PAINT statement is used to color (paint) an enclosed area. All we  
need to do is specify a set of coordinates within the area we want to color.

control codes

|  |  |  |
| --- | --- | --- |
| 10  20 | pr i nt g i n i t | "mm" \* |
| 30 | gfnove | 128?100 |
| 40 | circle | 60 |
| 50 | pa i nt | 128,100 |
| 60 | end |  |

Line 50 colors the circle since coordinates (128,100) are within the circle. The coordinates did not need to be the center of the circle, (100,100) will also suffice.

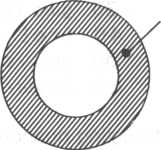


What do you think would happen if (10,10) is the parameter for a PAINT statement? Try it. Change line 50 to

**50 paint 10,10**

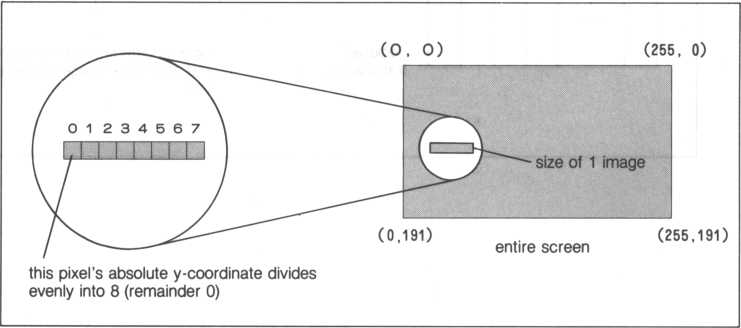
One point to keep in mind when using the PAINT statement is not to color in a donut shaped area. Except in the case of GMODE 0, it is sometimes not possible. In addition, the PAINT statement may go into what is known as an infinite loop, never finishing its task, The result is your program will be stopped at the PAINT statement. Avoid this as much as possible.

avoid coloring donut shaped areas as much as possible



1. **Coloring Graphics Pixels**

The graphics screen is composed of thousands of contiguous “images," which are merely blocks of 1x8. pixels. The x-coordinate of a pixel in an image is its x-coordinate on the screen. Dividing its absolute y-coordinate by 8 derives its position within the image. For example, if the absolute y- coordinate of a pixel divides evenly into 8 (remainder 0), its coordinates relative to that image are (0,0).



Using graphics, a single pixel can be colored any one of 16 possible colors using an FCOL statement. However, an image can only contain, at most, two colors. (Since the resolution-in the multi-color mode is 4x4 pixels, coloring is done in 4x4 pixel units.) A red line is colored in the next program.

10 ft ini: " iliiaffl" — control codes

20 g i n i t 30 ginode 40 fcol &QQ 50 groove 0> 100 60 draw 255>100

Let’s look into some interesting nuances of coloring graphics images with some examples.

• Designate a pixel in an image to be blue (2); only (2) is colored blue.

The other pixels are “colored” transparent (remember, the lack of color is also a color). If a different pixel within this image is colored a different color, say red for example, the whole image becomes red.

1 image

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 1 | 2 | 3 4 5 | 6 | 7 |
| I |  | I I |  |  |

® when a different

color is used

01 234567 on a different pixel

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A > | ^ A > | \* > | < > |  | k A |
|  | I  blue |  |  |  |  |

"colored”

transparent

when a different pixel is colored red, the entire image is colored red

• When the same color is used for a different pixel within the same image, only the designated pixel is colored.

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

T t I

blue blue blue

any pixel can be colored in an image if the same color is used

• If a pixel that has already been colored is colored a different color, only that pixel’s color changes. The rest of the image is not affected.

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

although blue at first, another color alteration changes the pixel to the new color

• Using two different colors for two contiguous dots belonging to different images is possible.

1 image 1 image

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

If two different colors are specified for two contiguous pixels in different images, neither of the two images are completely colored.

1. **Coloring the Graphics Background**

Initially the graphics background is colored transparent when a GINIT statement is executed. If the background of a graphics screen is to be colored, use a BCOL statement. Another method, a bit more tedious, is to color every image to the same color using an FCOL statement. Add line 35 to our program from section 11.8 to color our background blue.

|  |  |  |
| --- | --- | --- |
| 10 | P r i n t | " iJISli" — control codes |
| 20 | g i n i t |  |
| 30 | giYtode |  |
|  | bcol & | 05 |
| 40 | fcol & | :08 |
| 50 | gmove | 0, 1 00 |
| 60 | draw 2 | 55,100 |

All the dots of a graphics screen can also be colored using a FCOL and a BAR statement. Add the next few lines to our program.

1. fcol ?<0A
2. ba r 0 0 ? 255 ? 191

line 37—designates the color for the BAR statement

line 38—draws a solid square which is really the entire screen

Try changing the coordinates in line 38 and run our program again.

38 bar 10,10,150,150

Another statement that can be used is the PAINT statement. Change line 38. Coordinates (0,0) is where painting is started using the color specified by FCOL.

38 pa i nt 0, 0

1. **Using Different Colors**

We’ve seen how to color the inside of an enclosed area. We can also specify the inside of an area to be one color while the boundary color is another. Delete our old program from memory and try the next one.

**10!paint test**

20 print " ajjfiiii" — control codes

**30 g i n i t**

**40 bcol -101**

**50 fc?l \*<0F**

**60 box 24.- 24? 176? 17'6**

70 t'col .\*<02

**80 box 40.-40.-80.-80**

**90 f co l .\*<03**

**100 box 120.- 40.- 160? 80**

**110 fcol .\*<04**

**120 box 120?40?160?80**

**130 fcol \*<05**

**140 box 40?120?80?160**

**150 fcol \*<0A**

**160 g m o v e 150.-1 0 0**

**170 paint step 0?0?15?2?3?4?5**

l I

' designates boundary line color

Our program produces a large white square (color code &0F) and four smaller squares using color codes &02, &03, &04 and &05. The white square is then painted from its center in yellow (&0A); color codes &0F(15), &02, &03, &04 and &05 are designated as the boundary line colors in the PAINT statement of line 170.

Trying it again, let’s leave out yellow (&02) from the list of boundary line colors.

**170 paint step 0?0?15?3?4?5**

I color code &02 is omitted

since it’s the same as the color of the square

The box at the upper left is colored. In this way, its easy to see the boundary color designation serves as a boundary for PAINT statements.

1. **Dividing a Graphics Screen**

A graphics screen can be divided into different areas, called “viewports” with a VIEW statement.

10 print " iSlld" — control codes

20 v i ew 30 c l s

40 v i ew 0.- 0? 9? 23 50 g i ni t

**60 for 1=0 to 20 7 0** X **= r n d (9 0) ! Y =** r **n d (191)**

80 fcol AOF 90 draw X,Y 100 next I

line 20—initialize viewport usage

line 40—allocate a viewport whose base corner coordinates are (0,0) and opposite corner coordinates are (9,23)

lines 70 to 90—draw 20 random lines

The area in which all the lines are drawn are in the viewport allocated with the VIEW statement in line 40. After execution, change the screen to the Gl mode. You’ll see characters only in the area set aside as the viewport. In other words, the viewport has become the graphics screen.

If we cut out two other viewports of the same size and perform a

g i n i t ? 1

we can copy the screen from the previous viewport into our new viewports. Add the next few lines to our program.

**110 vi aw 10 ?0 ? 19? 23 120** g **i** n i t ? **1 130** view **20?0?29?23 140** g i n!+ ? **1**

lines 110 and 130—allocate another viewport of the same size and set it contiguous to our other viewport

line 120—designates a screen copy

The picture drawn earlier into our first viewport is copied into the other two viewports. Switch to the Gl mode and confirm it. The pictures of all three viewports are identical.

1. **Characters on a Graphics Screen**

When you want to display characters on a graphics screen, add a “# 1” to a PRINT statement.

10 print " IMS" •\*— control codes

20 v i ew 30 q i n i t

40 cri n t #1» "BASIC-G"?

I don’t forget the semicolon

line 40—The “# 1” designates displaying on the graphics screen. Note that the CURSOR and TAB functions can also be used here. Don’t forget the semicolon, otherwise your picture may be ruined.

The next program displays a mosaic using characters. Try changing the &EC in line 60 to another value and try it again for other effects.

|  |  |  |
| --- | --- | --- |
| 10 | print " ms" \*\*— control codes | |
| 20 | g i n i t |  |
| 30 | gmode |  |
| 40 | for 1=0 to 768 |  |
| 50 | fcol rnd(13)+2 |  |
| 60 | print#!.- chr$ (&EC+rnd (3) ) ’ | |
| 70 | next I |  |
| 80 | print "H" | - control code |

1. **Chasing a Sprite**

The coordinates of a sprite can be used to chase a sprite. Is your imagination already envisioning new game usages for this capability?

The coordinates of a sprite can either be the set of coordinates at its upper left (indicated with a “ # ”) or at its center (indicated with a In the next program, the coordinates where a sprite stopped is used as the coordinates for a DRAW statement. The end result are lines that trace the movement of a sprite.

10 p pint " fflisHB" — control codes 20 g i n i t 30 gmode 40 scod 0.&E1 bo loc 0 to 123?V6 60 loc 0 to 123?96 70 X=rnd(255):Y = rnd Cl91)

80 move 0 to X? V? 1

90 if status CO)=0 then goto 110

100 goto 90

110 f col &0F

120 drawSi0



130 goto 70

lines 90 and 100—loops until the sprite stops

line 120—draws a line to the center coordinates of where the sprite stops

Changing the program with the next couple of lines produces a circle using the coordinates of the stopped sprite as its center.

120 H=rndC20)+10 130 gmove30 140 circle H 150 goto 70

line 130—the graphics cursor is moved to the center of the stopped sprite

1. **Patterns**

Spectacular patterns can be drawn using geometric functions. For example, the sin (sine) and cos (cosine) functions produce dazzling pictures.

10 pr i n t " M!" — control codes 20 g i n i t 30 gmode 40 bcol 1 50 fcol 15

60 for Y=0 to 380 step 5

1. 0 **f** or X = — 100 0 t o 3000 step 10
2. 0 SX = s **i n** (10 0.- **X**)

90 SY=sin(60»Y)

100 Plot X/10+SY,SX/15+Y/2+10

110 next X

120 next Y

130 print " $" — control code

line 80—produces a swirl in the x-direction line 90—produces a swirl in the y-direction

How about the cos function? The next swirl gets smaller as it nears the bottom of the screen.

10 print " HUSH" ■+— control codes

20 g i n i t 30 gmode 40 erase

|  |  |  |
| --- | --- | --- |
| 50 | for I= 2 t o | 15 |
| 60 | H = 0 |  |
| 70 | f co l I |  |
| 80 | GY 1 =30 '• GY2 | = 5 |
| 90 | Plot 228,0 |  |
| 100 | for TH=0 | t o 6 \* 18 0 |
| 110 | H=H+1 |  |
| 120 | X = cos (80- | H/2,TH) |
| 130 | GYl= GY1rl |  |
| 1 40 | GX=128+X |  |
| 150 | GY2=GY2+1 |  |
| 160 | draw GX« G | Y2 |
| 170 | gmove 128 | , GY 1 |
| 180 | draw GX>G | Y2 |
| 190 | amove GX> | GY2 |
| 200 | next TH |  |
| 210 | next I |  |
| 220 | goto 50 |  |

line 110—forces the swirl to get smaller until h equals 80 line 120—designates the size of the swirl

1. **GMODE Statement**

We’ve seen how to utilize colors on a graphics screen. There are other variations we haven’t explored. Sometimes one color is specified (let’s call this the “specified color”), but another color is actually used (and we’ll call this the “displayed color”). This type of relationship also applies to whether if and how pixels and images are displayed.

GMODE 0~3 specifies the relationship between the specified color and the color actually used.

GMODE 4~7 deals with the pixels or images themselves, whether they are turned on or off. This may sound a little strange. How can a pixel or image be turned on or off? Turning a pixel or image off means it becomes invisible; likewise, turning it on means it is visible.

The commands that apply here include:

PLOT DRAW BOX BAR

CIRCLE PAINT PRINT #

Several tables will explain our point.

GMODE/specified color/displayed color relationship

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command | MODEO New color | GMODE1 Color OR | GMODE 2 Color AND | GMODE 3 Old color |
| PLOT  DRAW  BOX  BAR  CIRCLE  PAINT | relationship between specified color and displayed color | | | |
| specified color is the displayed played color | OR of the specified and displayed colors’ codes produce displayed color | AND of the specified and displayed colors’ codes produce displayed color | displayed color remains |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command | MODE 4 Pixel OR | GMODE 5 Pixel AND | GMODE 6 Pixel XOR | GMODE 7 Old pixel |
| PLOT  DRAW | relationship between specified pixel and existing pixel | | | |
| BOX  BAR  CIRCLE  PAINT | specified pixel is displayed | OR of the specified and displayed pixells’ conditions | AND of the specified and displayed pixels’ conditions | displayed pixels remain |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command | MODEO  Replace | GMODE1 OR | GMODE 2 AND | GMODE 3 Old |
|  | relationship between specified image and displayed image | | | |
| PRINT # | specified image is displayed | OR of the specified and displayed images’ is displayed | AND of the specified and displayed images’ is displayed | displayed image remains |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command | MODE 4 OR | GMODE 5 AND | GMODE 6 XOR | GMODE 7 Old |
|  | relationship between specified pixel and existing pixel | | | |
| PRINT # | same as GMODE1 | same as GMODE 2 | XOR of the specified and displayed images’ is displayed | same as GMODE 3 |

The following program checks the function of GMODE 0.

10 p r i n \ " liMl ■' m— control codes

20 g i n i t

**30 ginode 0**

**40 fcol A07**

**50 bar 30,30, 130- 130**

**60 fcol \*0C**

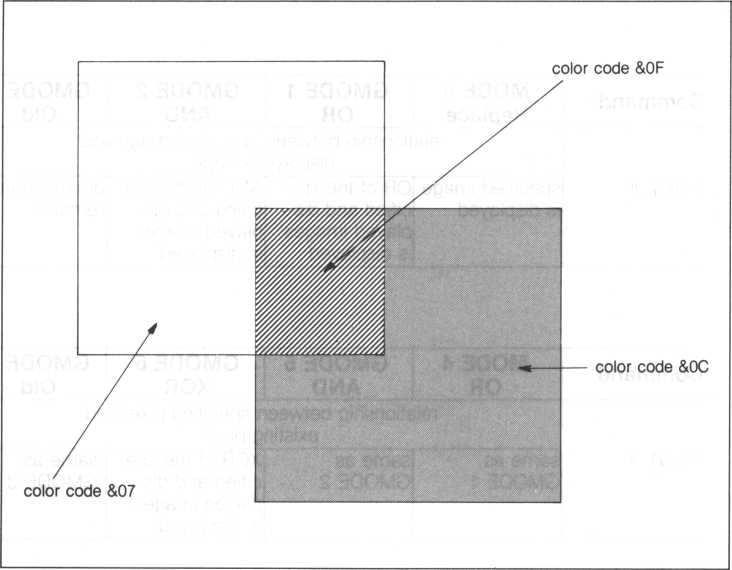
**7 0 bar 30,80\* 130, 180**

**30 end**

When run, the specified color is used regardless of the displayed color. Change line 30 to the following and run the program again.

**30 gmode 1**

An OR of the color codes &07 and &0C is displayed where they overlap, or color &0F.

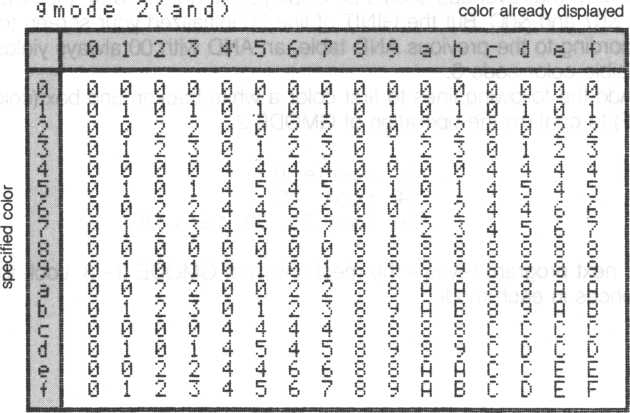
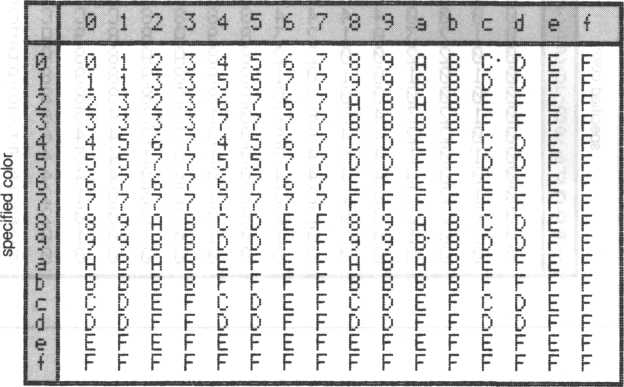


If you don’t know what an OR means, type in the next two PRINT statements followed by RETURN keys.

**print \*07 or &0c print 7 or 12**

The answer is 15, or &0F. The tables below lists summaries of the relationship between the specified color, the color already displayed and the resulting color.

9 rn O d e 1 O r ) color already displayed



9 Ifi O d 8 3 ( O Id ) color already displayed

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | l | o  JLm. | •J\* | 4 | 5 | 6 | ( | 8 | 9 | a | b | c | d | 0 | i |
| 0 | 0 | i | 2. | \*7 | 4 | 5 | 6 | ■J  i | O | 9 | ft | B | c | D | E | F |
| 1 | 0 | 0 | jL |  | 4 | 4 | b | 6 | O | 9 | ft | fi | c | C | E | E |
| 2 | 0 | 1 | 0 | 1 | 4 | 5 | 4 | 5 | a | 9 | 8 | 9 | c | D | c | D |
| \*y  o | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 0 | 8 | 3 | 8 | c | C | c | c |
| 4 | 0 | 1 | 2 | 3 | 0 | 1 | v | 7  •J | ;5 | 9 | a | B | 8 | Q | fi | B |
| 5 | 0 | 0 | •“  JLm |  | 0 | 8 | V |  | «-.  O | 6 | a | fi | 8 | 8 | fi | fi |
| 6 | 0 | i | fi | i | 0 | 1 | 0 | 1 | O | 9 | 8 | 9 | 8 | 9 | O | 9 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 8 | 0 | 1 | V | .J | 4 | 5 | 6 | -n | 0 | 1 | •“«  JLm | •-» | 4 | 5 | b | ■j  »’ |
| 9 | 0 | 0 | JLm | JLm | 4 | 4 | 6 | 6 | 0 | 0 | O  JLm |  | 4 | 4 | 6 | 6 |
| a | 0 | 1 | 0 | 1 | 4 | 5 | 4 | 5 | 0 | 1 | 0 | 1 | 4 | 5 | 4 | 5 |
| b | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 |
| r | 0 | 1 | O | 7 | 0 | 1 | JLm | 7 | 0 | i | 9  2. | 3 | 0 | 1 | 2 | 3 |
| d | 0 | 0 | o  JLm | •"i | 0 | 0 | 9  o\_ | 9  Am | 0 | 0 | 2 | \*9 | 0 | 0 | 2 | \*9 |
| e | 0 | 1 | 0 | t | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| t | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Change line 30 again and rerun our program.

**30 gmode 2**

The area that overlaps should be displayed with color &04, the AND of colors &07 and &0C. But the GINIT of line 20 initialized your screen to color 0. According to the previous AND table, an AND with 00 always yields 0, thus invisible color code 0.

Add the following lines to first color a white background box (color code &0F) to confirm the operation of GMODE 2.

22 gmode 0 24 t col 15

**26 bar 30,30,180,180**

The next program reveals the mechanism of GMODE 0~3. Look for the differences in each mode.

|  |  |  |  |
| --- | --- | --- | --- |
| 100 | ! qfiTode test (0’"'3 | | ) |
| 110 | print "liM&iS" -— | | - control codes |
| 120 | vi ew:c l s |  |  |
| 130 | bcol 3:01 |  |  |
| 140 | 1 |  |  |
| 150 | f 0r GM-0 | t«:» 3 |  |
| 160 | for F'A = | 0 10 15 | |
| 170' | I T I 0 s S 9{- |  |  |
| 180 | v i euj 0 ? | 0,31,2 |  |
| 190 | 1 ocat e | 3, 1 |  |
| 200 | print " | gmode"? | GM? |
| 210 | locate | 3,2 |  |
| 220 | print “ | color"? | TS  J> |

**230! paint box 240 v i ew 3, 3- 23- 2-3 250 g i n i t 260 gitiode**

**270 fcol A:OF: box 0,0,152,152**

**280 locate 2., 17**

**290 p pint #1 -"pa int i ng"?**

**300 locate 2, 18**

**310 p r i n t # 1» " color"?PA?**

**320 fcol PA:bar 1,1,151-151**

**330 !**

**340 gn'iode gM**

**350! vert i cal line**

**360 for X = 0 to 15**

**37 0 g Us o d e ? f c o l 15**

**380 locate X+2,1**

390 p r i n t #1 rig h t$(hex $ <X), 1)

**400 gniode GM**

**410 fcol X**

**420 bar(X+2)+8,16,(X+2)+8+7,143**

**430 next X**

**440! hor i zon t a l line**

**450 for Y = 0 to 15**

**460 grriode! fcol 15**

**470 locate 1,Y+2**

**480 pr i nt#l r i ght$ (hiex$ (Y) , 1)**

**490 g ir'i o d e G M**

|  |  |  |
| --- | --- | --- |
| 500 | fcol Y |  |
| 510 | bar 16? (Y+2)\*8? | 143? (Y+2)\*8+ |
| 520 | next Y |  |
| 530 | sleep 1 |  |
| 540 | next F'A |  |
| 550 | next GM |  |
| 560 | P r i n t "fa" : end — | - control code |

1. **Multi-color Mode Graphics**

As mentioned earlier, graphics is also possible in the multi-color mode. But the resolution is limited to a large 4x4 pixel block, thus there are only 3072 (64x48) distinct locations. Other than this, graphics in the multi-color mode is done similar to the Gil mode. Be aware that since the resolution is limited to a 4x4 dot matrix, coordinate matching and sprite.interactions (which are based on being able to use a single pixel afe coordinates) are a bit different.

The next program draws a circle but, as can be seen, it is not suitable for graphics.



10 print " mm" \*\* control codes

**20 g i n i t**

**30 gwode**

**40 groove 32 ? 24**

**50 for 1=0 to 6**

6} O t c O t i- + I +2

**7 0 circle 5 +I \*3**

**80 next I**

line 10—changes to the multi-color mode lines 20 and 30—identical in the Gil mode

line 40—since the number of addressable locations is small, (32,24) is the center of the screen

Another mosaic is drawn in the next bit of software.

10 pr i n.t "IMS" — control codes

**20** g **i n** i **t**

**30 gmode**

40 bcol 1

50 for Y=0 to 48

60 for X = 0 to 64

70 t'co l rnd (13) +2

**80 plot X? Y**

90 next X

100 next V

1 i 0 pp I n t "9" 5 and control codes

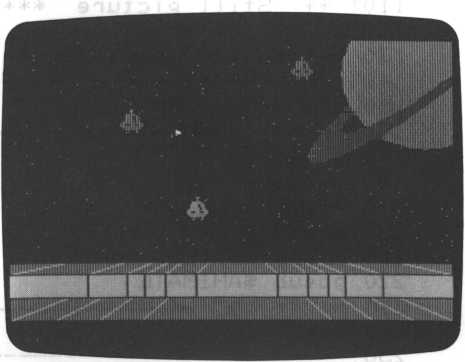
**CHAPTER**

1. **ANIMATION**

Animation, the crux of exciting game software. We’ll look into adding movement to sprites constructed using BASIC-G graphics. How should we go about learning animation? Follow along.

1. Draw a sprite freehand—it’s no exaggeration to say that at this stage the success or failure of the animated ambience is decided
2. Use what we learned in sections 3.13 and 5.2 to construct the sprite in BASIC-G by specifying its pattern code. As an aid, make a copy of Appendix D and use it to mark the pixels of your sprite. From this, the pattern code is easily ascertained.
3. Figure out the coordinates of where your sprite is to be displayed on the screen. Use the CRT layout in Appendix E to easily plot the screen.
4. Write a BASIC-G program using the above data.
5. **Constructing the Environment**

How about animating the following picture?

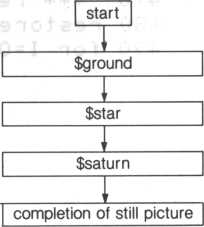


This picture can be divided into two distinct parts, (1) a still graphics picture and (2) an animated sprite.

**12.2 Still Picture**

Input the following program to draw the still picture. Remember, the lines with an exclamation point (!) are remarks and are not processed by the M5 computer. They are merely for our sake, to help us understand the flow of the program better.

This program contains two main subroutines, SSTILL (line 280) draws the background still picture, $ANIMATE, naturally, will move the sprite against the still picture. This section deals only with the still picture. The remaining software dealing with animation has been separated for the sake of clarity and is included in the next section. So, don’t execute this program yet, wait until the software for subroutine SAN I MATE has been appended (added) in section 12.3—if you’re really itching to try out the still picture software, you can omit line 220 and run this program. The still picture is made up of three subroutines, SGROUND, SSTAR and SSATURN. Its flow is described in the following flow chart.



110! **\* \* Still** picture \*\*+ **120! .?<**

**130 ! \*\* sprite an i ma t i on** ■\*\*

**15 0 d i i'fi X (3) » Y (3) , P (3)**

**160 print "HUMS"**

**170 g i n i t**

**180 gmode**

**190 view**

**200 erase**

**210 g os ub $STILL**

**220 g o s ub $ ANIMATE**

**230 print "0"**

**240 end**

**260! \*\*\*\* Still picture \* \* +**

**280\*STILL 290 gosub $GROUND 300 g o sub $3TAR 310 gosub s&SATURN 320 return 330$GROUND**

**340! + + •+■+ yellow ground \*\*\***

**350 f co l** S:**0A**

**360 bar 0,152,255,191**

**370! + \* read (dat.al diag. I**

**380 nest ore 640**

**390 for 1=0 to 11**

**4 0 0 r ea d X 0, Y 0, X1» Y1**

**410 fcol C**

**420 draw X0,Y0,X1,Y1**

**430 next I**

**440! + +\* white ground \*\*\* 450 fcol** &QF

**460 bar 0, 160,255, 176 470! \*\*\* read (data2,3 vert 480 restore 770 490 for 1=0 to 12**

control codes

control code

i ne)

**. Ii ne?)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 500 | read XO | | , YO? | XI? Y | i,e |  |
| 510 | f c | o l C |  |  | / |  |
| 520 | dr | auj XO | ? YO? | XI ? Y | i |  |
| 530 | next | I |  |  |  |  |
| 540 | return | |  |  |  |  |
| 550\*STAR | |  |  |  |  |  |
| 560 | f o r | 1=0 t | o 300 | |  |  |
| 570 | f c | o! AO | “TT  f |  |  |  |
| 580 | Xs | rnd(255): | | Y = rnd (15 | | 0) |
| 590 | Pi | ot X? | Y |  |  |  |
| 600 | next | I |  |  |  |  |
| 610 | return | |  |  |  |  |
| 620 1 | ++\* | \* dat | a \*\* | \*\* |  |  |
| 630 | 1 \*\* | data 1 | (d i | a g. | l i ne) | |
| 640 | data | 32? 152? 0 | | ? 16 0 | ? 15 |  |
| >;j50 | da t a | 64? 1 | 52? 0 | > 176 | ? 15 |  |
| 660 | data | 80? 1 | 52? 0 | ? 192 | ? 15 |  |
| 670 | data | 88? 1 | 73 ju- ? y. | 4? 19 | 2? 15 | i |
| 680 | data | 96? 1 | 52?48?19 | | 2? 15 |  |
| 690 | data | 112? | 152? | 72? 1 | 92? 1 | 5 |
| 700 | da t a | 144? | 152? | 134, | 192? | 15 |
| 710 | data | 160? | 152? | 208? | 192? | 15 |
| 720 | data | 168 ? | 152? | 2 3 <1 ? | 192? | 15 |
| 730 | data | 176? | 152? | r;; <r ^ | 192? | 15 |
| 740 | data | 192? | 152? | 255? | 176? | 15 |
| 750 | data | 2$4, | 152? | 255? | 160? | 15 |
| 760 1 | \* + | da. ta2 | (ho | r i z. | 1 i n | e) |
| 770 | data | 0? 17 | 6?255? 17 | | 6? 6 |  |
| 780 | data | 255? | 160? | 0? 160? 6 | |  |
| 790 | d a t a | 0? 15 | 2? 255? 15 | | 2 ? 6 |  |
| 800 1 | + + | data3 | (vert. | | l i ne | ) ■ |

+ + +

+ \* +

**810 dat a 42? 160?42? 175? 6 820** data **64?160?64?175?6 830 data 74?160?74, 175, 6 840 da** t **a 86 ? 160,86? 175? 6 850 data 104? 160?104?175?6 860** data **152?160?152?1/5?6 870 data 170,160?170,175?6 880** data **182?160?182?175?6 890** data **192?160,192?175?6 900** data **214,160,214? 175? 6 910\*SATURN**

920 ! + + + + s a t u r n \* + \* \*

**930 groove 240?30 940 fcol A07**

**950 circle 50**

**960 paint step 0? 0**

**970! \*\*\*\* saturn’s rings \*\*\***

**980 f col £05**

**990 c i rcle 70?60? ? 0? 158? 70 1000 c i rcle 50? 40?? 0? 140?70 1010 pai n** t s **t ep-60**? **40** ? **5? 1** 1020 return

line 150—variables used for animated sprite. This is not used yet, notice

that this is only concerned with drawing the still picture and has not yet begun dealing with the sprite.

line 220—this subroutine call deals with the sprite and its animation. If you only want to draw the still picture, take out this line.

line 360—draw yellow ground using color &0A from line 350

line 380—initialize DATA statement beginning from line 640 for reading

line 390 to 430—loop 12 times and read data for lines 1 to 12

line 460—draw white ground using color &0F

line 480—initialize DATA statement beginning from line 770 for reading

lines 490 to 530—loop 13 times and read data for lines 13 to 25

lines 560 to 600—loop 300 times and draw 301 background stars at random coordinates with the PLOT statement

line 580—generate two random numbers for the stars’ coordinates

lines 640 to 900—coordinate data used for DRAW and FCOL statements. There are 25 lines to be drawn on the screen, one data statement takes care of one display line. Display lines 1 to 12 are read from lines 370 to 430 while display lines 13 to 25 are read from lines 470 to 530 with READ statements. A DRAW statement then draws the lines.

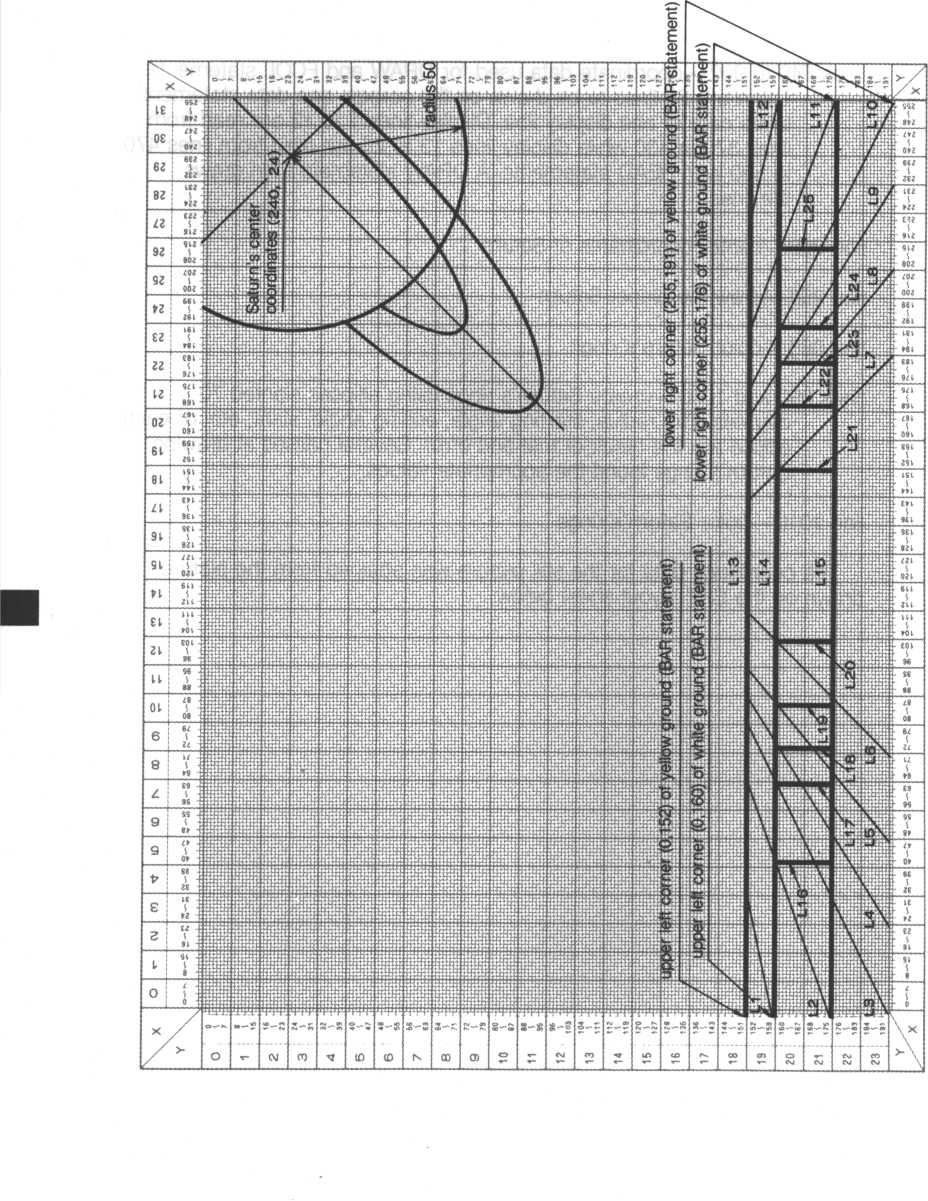
line 930—center coordinates of Saturn line 950—draw the planet Saturn line 980—color Saturn’s rings

lines 990 and 1000—create Saturn’s rings using ellipses slanted 70

degrees. The radius, starting angle, ending angle, slant, etc. of each ellipse are the result of trial and error. Several passes at drawing Saturn’s rings found the appropriate values.

line 1010—color Saturn’s rings

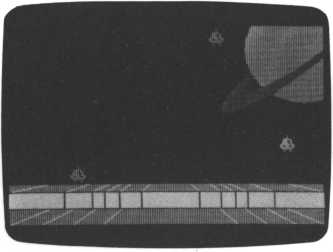
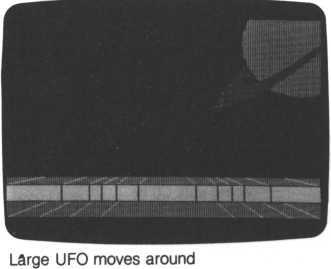
The coordinate data for the still picture was attained from the picture drawn on the CRT layout sheet below.



1. **Animated Picture**

We’ve seen how a still picture can be drawn. Let’s now animate a UFO against this backdrop. How about the following animation sequence? (It’ll be repeated twenty times.)

* Move a large UFO around
* Divide the UFO into four smaller UFOs, then move each independently
* Combine the four UFOs into a large UFO



Divide the UFO into four smaller UFOs

As said before, the program of section 12.2 combines very nicely with the software in this section. The program in section 12.2 ended at line 1020; the following software picks up at line 2000. If you only want to deal with the software in this section, leave out the program in section 12.2 and add lines 1900 and 1910 below (otherwise, leave them out); also change line 2110 to:

2110 end

The animation is composed of six subroutines, $STCHR, SSP.SET, SSP.- JOINT, SSP.MOVE, SSP.CUT and $SP.UNITE. Its flow is described in the following flow chart.

I start I

**~zz**

SSTCHR

♦

SSP.SET

*l*

$SP. JOINT

SSP.MOVE

*l*

SSP.CUT

I

SSP.UNITE

repeat until CO is greater than 19 r

end

SSTCHR (lines 2130 to 2180)—set the UFO pattern code and assign ASCII codes &7C&7F



SSP.SET (lines 2190 to 2330)—assign the UFO patterns to sprites 4 to 7 and move to coordinates (110,180)

SSP.JOINT (lines 2340 to 2390)—use JOINT to make a large UFO

SSP.MOVE (lines 2410 to 2480)—move a large UFO to random coordinates using MOVE-TO.

SSP.CUT (lines 2490 to 2670)—using JOINT in line 2510, disjoin the

UFO. The separated UFOs are enlarged to 16 by 16 dot sprites with MAG in line 2540. Furthermore, these sprites are moved to their own random coordinates.

SSP.UNITE (lines 2680 to 2920)—randomly select a sprite and move the other three sprites to the coordinates of this sprite. The sprite information of the randomly selected sprite is kept in the array variable P(0)~P(3). (P(0) contains the y-coordinate and P(1) contains the x-coordinate)

1900 print " iMSISii ": e r a s e

19 1 0 d i m X (3) , Y (3) , P (3)

2 0 10 ! \* \* S p r i t e A n i ma t i o n + \* \* \* 20 30$ANIMATE

|  |  |  |  |
| --- | --- | --- | --- |
| 2040 | C0"0i ! | (i f c0 >19 end p rog r | am) |
| 2050 | go sub | $STCHR |  |
| 2060 | gosub | \*SP.SET |  |
| 2070 | go sub | $SP. JOINT |  |
| 2080 | gosub | $SP.MOVE |  |
| 2090 | gosub | TSP.CUT |  |
| 2100 | gosub | $SP.UNITE |  |
| 21 1 0 | return |  |  |
| 2120! | SpT i t e pa X t 0P H CC'd 0 | |  |
| 2130$ | STCHR |  |  |
| 2140 | s tchr | "0100 00 0707 0f 0f 0 f" | to &7C»0 |
| 2150 | stchr | "1f 3f 393f1f0f0206" | to &7D»0 |
| 2160 | •stchr | "c 03f 393f1fOf0206" | to T.7E, 0 |
| 2170 | stchr | "dcf ecef efc f 81030" | to A7F\*0 |
| 2180 | return | |  |

21.90$SP. yET

|  |  |  |
| --- | --- | --- |
| 2200! | ++ set | UFO spri t es |
| 221 0 | ma g 1 |  |
| 2220 | f o r S = 0 | t o 3 |
| 2230 | 5 C O d S | +4 ? TT’C+S |
| 2240 | SCO l S | +4\*T04 |
| 2250 | next S |  |
| 2260 | loc 4 to | 110,180 |
| 2270 ! | \* \* set | invisible sprites |
| 2280 | for S=0 | to 3 |
| 2290 | s c o d S | , 7:80 |
| 2300 | sco l S | , 700 |
| 2310 | l o c S | to 120,176 |
| 2320 | next S |  |
| 2330 | r e t u r n |  |
| 2340! | \*;+: j o i n | spr i t es |
| 2330$ | SP.JOINT |  |
| 2360 | joint 5 | to 7,4 |
| 2370 | joint 7 | to 6, 3 |
| 2380 | joint 6 | to 4, 2 |
| 2390 | r e t u r n |  |

**2400! \* \* move large UFO 241 0 $SP. MOVE**

**2420 Cl=0s! cl =times moved 2430 X = rnd(230)sY-rnd(160)**

**2440 move 4 to X,Y,1 2450 C1=C1+1**

**2460 if Cl>4 then return**

**2470 if status (4-) =0 then goto 2430**

**2480 goto 2470**

**2490! \*\* break up large UFO**

**2300\*8P.CUT**

**2510 joint 0? 7**

**2520! \*\* move small sprites**

**2530 C2 = 0: ! c2 =1 i mes small UFOs move**

**2540 mag 2**

**2550 for 1=0 to 3**

**2560 X(I)= rn d(230):Y(I)= r n d(160)**

**2570 scol 1+4,404+1\*5 2580 move I+4 to XCI)>Y(I), 1 12590 next I 2600 F=0**

**2610 for K = 0 to 3 2620 F=status(K+4)+F 2630 next K**

**2640 if F>0 then goto 2600 2650 C2=C2+1**

**2660 if C2>4 then return**

**2670 goto 2550**

**2680\* \*\* unite small UFOs**

**2690$SP.UNITE**

**270 0 N = r n d (3) + 4**

**2710 for S=0 to 3**

**2720 P(S)= s prite(N,S)**

**2730 next S**

**2740 for 1=0 to 3**

**2750 move 1+4 to P(1),P(2),S+1**

**2760 next I**

**2770 F=0**

**2780 for K = 0 to 3 2790 F=status(K+4)+F**

**2800 next K**

**2810 it' F>0 then goto 2770**

**2820! +\* join sprites togather**

**2830 jo i nt 5 to 7? 4**

**284-0 joint 7 to 6? 3**

**2850 joint 6 t o 4- > 2**

**2860 for S = 0 to 3**

**2870 scol £.+4,8(3)**

**2880 next S 2890 mag 1 2900 C0=C0+1**

**2910 if CO=19 then mag 3 s return**

**2920 g o** t **o 2080**

lines 1900 and 1910—omit these lines when combining this software with lines 100 to 1020 of section 12.2.

line 2040—CO is the counter for the number of times the entire animation sequence is performed (initialized to zero)

lines 2040 to 2100—main body of subroutine SANIMATE

line 2110—change as follows if lines 1900 to 2920 are not combined with the program from section 12.2

lines 2140 to 2170—UFO pattern code, identical to the large UFO created in section

lines 2220 to 2250—assign ASCII codes to the 4th to 7th sprites

lines 2280 to 2320—color sprites numbered 0 to 3 invisible (line

2300)—allows the still backdrop to be seen since the four sprites are not yet joined and are all at (120,176). If you don’t understand, change the color code in line 2300 and run the program. It’ll help you to appreciate the “invisible” color.

line 2360 to 2380—join sprites 4~7

lines 2430 to 2480—move the large UFO in random directions line 2470—check if sprite is moving line 2510—disjoin the large sprite

lines 2550 to 2590—move the four small UFOs in separate random directions

lines 2600 to 2630—wait for the four small UFOs to stop moving

line 2700—randomly determine sprite number

lines 2710 to 2730—substitute sprite data into P(0)~P(3) from data of sprite determined randomly in line 2700

line 2740 to 2760—move sprites together to the coordinates of the sprite determined in line 2700

line 2770 to 2800—wait for the four small UFOs to stop moving

lines 2830 to 2850—join the four small UFOs together

lines 2860 to 2880—using SCOL, color the joined UFO to the color of the sprite randomly determined in line 2700 and assigned in lines 2710 to 2730

line 2890—create a large UFO

**CHAPTER**

1. COMPOSING YOUR OWN MUSIC

Although it is sometimes open for debate, it is generally agreed music is not a collection of random sounds. If you know very little about the concept of a note and how it relates to music, check out a music textbook or nearest music store. But even if you are of the uninitiated, try and follow along anyway. It can still be fun.

* 1. Using the PLAY Statement

Your M5 uses the designations C, D, E, F, G, A, and B to describe music. When the Ready prompt is displayed, type in the following.

p l a y" c"

The C note is played after the RETURN key is pressed. Another way to

designate the C note is ‘do’. This latter system is not used by BASIC-G. It’s

mentioned here since quite a few people use it to verbally describe music. The relationship between the two systems is:

C D E F G A B

do re mi fa so la ti

The next input plays from ‘do’ to ‘ti’, or C to B. This is the most basic form of the PLAY statement. The alphabets within the double quotes can be either upper or lower case.

piay"cdefgab"

* 1. Sharps and Flats

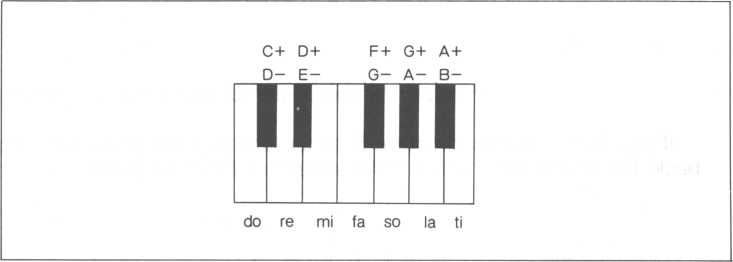
The notes C through B are the primary designations for notes. There are also notes halfway between these primary notes, also called “half steps.” Going half of a step up yields a sharp (#) of that particular note; half of a step down yields a flat ((?) of that note. BASIC-G uses a plus sign ‘ + ’ to denote a sharp and a minus sign ‘ ’ to indicate a flat.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Function | BASIC-G  Designation | Example |
| sharp (#) | move up half | add a ‘ + ’ after | C + (C sharp) |
|  | a step | the note name | D + (D sharp) |
| flat (b) | move down | add a ‘ - ’ after | B- (B flat) |
|  | half a step | the note name | E- (E flat) |

The next input line plays a scale that includes the notes C through B and their corresponding sharps. The line following plays the same scale using their corresponding flat designations. A piano keyboard representation of these same notes follows. The shaded areas denote half steps. As you can see from the figure, E sharp is F, F flat is E, B sharp is C, and C flat is actually B.

P l a y " c c + >.:l d + e f t + g g+a a + b "

P l a y " c d -- d e - e t" g - g -- a **b—b"**



* 1. Octaves

It’s obvious our seven notes (twelve including their sharps or flats) cannot handle very many songs. We need more notes, which brings us to the concept of an octave. We can consider our seven notes to be an octave. If we repeat the same concept for another set of seven notes (either higher or lower), we have two octaves. Repeating this another time gives us three octaves. BASIC-G provides up to six octaves, which is only one less than a full scale piano.

The octave we played earlier is octave 5 shown below. When power is first supplied to your M5 console, the default octave number is 5.

middle C

|  |  |  |
| --- | --- | --- |
| 1 | | 1 |
| -g> ill" j J J | J J \*\*' | T r 1 -riz |
| J 1 \*  1  C D E F G A B | rO — O m  TI  o  > | 3CDEFGAB |

octave 4 octave 5 octave 6

If we want to play a note in the next higher octave, precede the note(s) with the letter ‘o’ and the octave number. The next example also plays the C which is one note higher than the B in octave 5.

**P l a** y " c **d e f g** a **b o 6** c"

octave number (letter ‘o’ and a number from 3 to 8)

If you want to return to octave 5, either turn your M5 power OFF and back ON or change it with a PLAY statement as shown below.

**P I** a**y** "cego7cego5ceg"

return to octave 5

change to octave

plays octave 6 from previous PLAY statement

* 1. Duration of Music Notes

To play interesting music, we need to be able to specify the note duration. BASIC-G allows up to six different durations. When your M5 is first powered ON, it plays quarter notes. In fact, our last exercise used quarter notes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ty | | | pe of note | | | |
|  | Whole  note | Half  note | Quarter  note | Eighth  note | 1/16  note | 1/32  note |
| Music  sheet  symbol | o | J | J | J | J> |  |
| BASIC-G  value | 1 | 2 | 4 | 8 | 16 | 32 |

The duration is denoted with its BASIC-G value after the note name (without the ‘o’ which denotes the octave). For example, the following plays a half note C. Omitting the duration assumes a quarter note.

play "c2"

* 1. Dotted Notes

If that were not complicated enough, music also contains dotted notes which last in duration equal to its normal value plus half again as long. For example, a dotted quarter note equals the duration of a quarter note plus an eighth note.

Notice the periods (dots) in the BASIC-G duration notation below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ty | | | pe of note | | | |
|  | Dotted  whole  note | Dotted  half  note | Dotted  quarter  note | Dotted  eighth  note | Dotted  1/16  note | Dotted  1/32  note |
| Music  sheet  symbol | O' | J. | J. |  | J | J |
| Duration |  | J+J | J+J> |  |  |  |
| BASIC-G  value | 1. | 2. | 4. | 8. | 16. | 32. |

Practically speaking, the difference between a 1/32 note and a dotted 1/32 note (duration of a 1/32 note plus duration of a 1/64 note) can hardly be discerned since the duration of a 1 /64 note is so short. One point, omitting a number assumes a dotted quarter note.

The next PLAY statement gives an example of dotted notes. Assign it line 10 and execute it as a program.

**10 play "cba.g8de8c8dega.b8b2"**

* 1. Sound Length Request—Triplets

Up to this point, the duration of notes were based on the premise that a whole note equals 1. Other notes are then a simple fraction of a whole note. (A quarter note is 1/4 of a whole note, so ‘4’ denotes a quarter note.) What do we do about triplets? A triplet, which is three notes having the time value of two notes of the same kind, doesn’t follow this philosophy.

Let’s walk through an example, a triplet lasting the duration of a half note.

We already know that when we divide a half note by 2, we get two quarter notes. Another way to think about it is

1 1 1

2 4 4

where 112 is the value of the’ half note equal to 1 + 1 /4, or two quarter

notes. And since an equivalent triplet is three notes lasting the same duration, let’s divide the half note by 3.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Note duration | | | | Calculation is as follows |
| two quarter notes lasting the time value of a half note | J | L. | u |  | 1/2 = 1/4+1/4 |
| triplet lasting the time value of a half note | J | LJ | jj |  | 1/2 = 1/6+1/6+1/6 |

As it turns out, the duration of a note can be expressed very accurately, accurate enough to approximate the sound of a triplet with uncanny precision by specifying the duration of each note. In practice, this is quite necessary since harmony requires both the melody and the accompaniment to be in sync. This prevents one from overtaking the other because of inexact triplet durations. The trick is in symbolically expressing the duration of a note followed by an exclamation point. For example

Play "c24! "

plays a C note whose duration is about a dotted quarter note. Where did the “24” come from? 24 could have been any number, the reference values follow. Notice 24 is halfway between a half note and a quarter note, or a dotted quarter note.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type of note | | | | | |
|  | Whole  note | Half  note | Quarter Eighth note note | 1/16  note | 1/32  note |
| BASIC-G  value | 64! | 32! | 16! 8! | 4! | 2! |

A set of triplets can be set up so they approximate very closely the desired affect.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Masic  sheet  symbol | calculation | Example using the C note |
| Quarter note triplet | JJJ | 32 = 11 + 11 + 10 | cm cm cioi |
| Eighth note triplet | m | 16 = 5 + 5 + 6 | C51 C51 C61 |
| Sixteenth note triplet | m | 8=2+3+3 | C21 C31 C31 |

* 1. Music Rests

Music requires resting for predetermined durations as well as playing notes. As with notes, there are corresponding rest periods. But since rests, by definition, are not sounds, they only require one denotation, an ‘FT.

**Type of rest**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Whole  rest | Half  rest | Quarter  rest | Eighth  rest | 1/16  rest | 1/32  rest |
| Music  sheet  symbol | - | - | \* | y | 1 |  |
| BASIC-G  value | r1 | r2 | r4 | r8 | r16 | r32 |

We’ll put some rests in our earlier melody.

**10 p l a y " c b a. g 8 d e8 c 8 d r 4 e g a. b 8 b 2 r 4"**

U U quarter rests

* 1. Harmony

Three notes can be played simultaneously to created harmony. Sophisticated music with a melody and accompaniment can be composed and played on your M5 musical instrument.

To play more than one note concurrently, separate them with commas in the same PLAY statement.

p l a y " c “» " e" plays a C and E together

p lay "c", "e", " g" plays a C, E and G together

Harmony is limited to three notes. If more than three notes are specified, only three notes are played, an Err 2 is encountered, and program execution is stopped.

Let’s add some harmony.

10 play "egagegag", "cba. g8de8'c8dr4ega. b8b2r2"

i i

harmony

Add line 20 for another effect.

20 goto 10

* 1. Volume Level of a Music Note

A note can take on one of fifteen volume levels using the V (Volume) setting. When your M5 is first turned ON, it is always set to volume 15 (V15).

V statement volume settings

|  |  |
| --- | --- |
| V statement setting | VO -—- V15 |
| Volume | soft-—- loud |

Note: Volume setting 0 (zero) is silent.

After selecting a certain volume, by pressing the SHIFT and RESET keys or by setting the V setting, the volume (V15) is always available.

10 piay "v5cdev10cdevl5ced"

* 1. Tempo

The tempo is the speed of a song. It’s usually indicated at the upper left of the music score; an example is “Moderato.”

BASIC-G alters the tempo with a T setting between 1 and 255. Look at the the table showing the relationship between the T value and common tempos.

|  |  |  |  |
| --- | --- | --- | --- |
| Tempo | Meaning | Quarter note equals | T |
| Presto | Rapidly | 184 | 184 |
| Allegro | Lightly and quickly | 134 | 134 |
| Allegretto | Rather quickly | 108 | 108 |
| Moderato | Mild pace | 92 | 92 |
| Andantino | Faster than andante | 80 | 80 |
| Andante | Walking speed | 74 | 74 |
| Adagio | Slowly | 58 | 58 |
| Largo | Easy and slow | 46 | 46 |

When your M5 is first turned on, it is initialized to T equals 116. Once the tempo is altered, it must be set explicitly to another tempo or by pressing the SHIFT and RESET keys (which resets it to 116). Try the next statement.

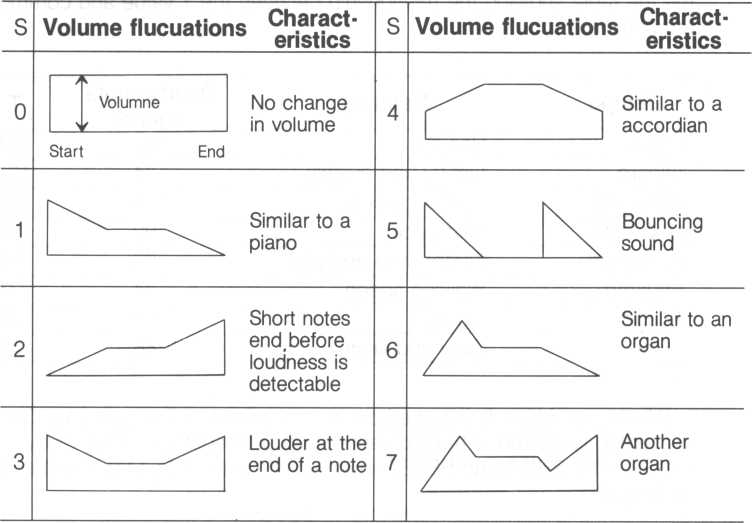
**play "t**100**"**

tempo setting of 100



* 1. Sound Alteration

Using BASIC-G, the volume changes within a note can be altered one of eight ways. The duration of the changes depend on the duration of a note.



When power is first supplied to your M5, 0 is used for the S setting.

Once set, the S setting must be set explicitly to another S setting or by pressing the SHIFT and RESET keys. Therefore, if S8 is set, it will remain at S8 until set again.

Type in and execute the following. It plays each note of one octave using all eight S settings. The second C is two octaves lower than the rest of the notes to make it easier to “sound out” the beginning of the octave.

lOrem 5 test 20 restore 30 for 1=0 to 7 40 read 8$

50 play 8$

60 pla y "o5cdefgabo3c"

70 next I 80 goto 20

vo data sO? si ? s2? s-\*\*? s4? s5’ s7

The S setting can be made using two methods. One method is shown above using the DATA statement. The other, specified within our familiar PLAY statement double quotes, is shown below.

10 p l a y " s2 PLAY is used to change S setting

20 p lay " cdesSf gs2ab S setting can be embedded

within a music passage

When S is set to a value not zero, the volume setting (V) is not longer applicable.

* 1. Staccato

Staccato shortens the duration of a note without shortening the intended duration of the note. For example, a staccato quarter note sounds very short and terse, but the time allocated to the quarter note is still used (the rest of the duration is silent).



When power is first supplied to your M5, H is set to 7. To change H, H must be reset explicitly or pressing the SHIFT and RESET keys concurrently.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| H | Note duration (amount of staccato) | | H | Note duration (amount of staccato) | |
| n | start | | J end | C |  | v/////m. \ |
| u |  | | O |  |  |
| 1 |  | ^ I | a |  | W77/77W/. 1 |
| i |  | | D |  |  |
|  |  | |  |  |  |
| o |  | 'm | 7 |  | WW 1 |
|  |  | | I |  |  |
|  |  | |  |  |  |
| Q | i | | Q |  | 77777777777777?. |
| o |  | | O |  |  |
| 4 |  | i |  |  | |

To clarify usage of the H statement, modify lines 10, 40, 50 and 90 of the program we just used from section 13.11 and run the program again. Instead of hearing each note flucuate sound volumes, you’ll hear each note affected by the nine types of staccato, H0~ H8.

10 r <3 h i h test

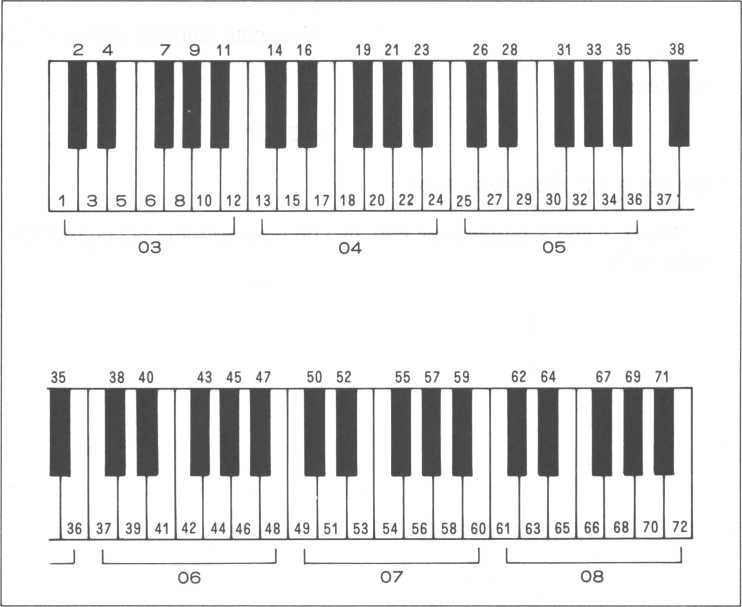
|  |  |  |
| --- | --- | --- |
| 20 | rest | ore |
| 30 | f or | 1=0 to 8 |
| 40 | read | H$ |
| 50 | Play | m |
| 60 | P lay | " o 5 c d e t‘ g a b o 6 c r " |
| 70 | nex t | I |
| 30 | g o t o | 20 |
| 90 | data | HO >hijh2»h3fh4,h5-h6\*h7 |

As with the S setting, the H statement can be used with two methods. One method is shown above using the DATA statement. The other specifies the H value within PLAY statement double quotes.

H8 is identical to playing the note without staccato; HO is the same as a rest since the note is not played for its duration.

* 1. Using Numbers to Denote Music Notes

In addition to the C through B jargon we’ve been discussing, music notes can also be designated with numbers. BASIC-G can play six octaves of twelve notes each (seven notes plus their sharps or flats), resulting in 72 notes. Starting with the lowest note, each note is sequentially numbered from 1 to 72.



An example of this type of notation follows.

Play "n25n27n29rn25n27n29rn30n31n32"

A potential handicap when using this technique is that the sheer volume of numbers within a music passage enclosed by double parentheses can be cumbersome. Also how can we specify its length? To avoid conflicting notation, use semicolons as special delimiters between the note identifier and its qualifier.

| necessary semicolon delimiter

play "n4;32"

duration of note (1/32 note)

note identifier (same as 03e + )

whereas,

Play "n43;2" defines a half note equal to o6f +.

The V, T and S parameters can also be used with the N command to alter notes.

* 1. Altering the Pitch

With BASIC-G, you can alter the pitch up or down by up to 11 half-steps. Say you want to sing along with your M5, and the tune is too high or low to be comfortable for your voice, the entire song can be moved up or down until it’s just right.

Using the oa\* command, where ‘a’ equals 1 — 11, changing the pitch is easy. The ‘a’ signifies the half-step position referenced to middle C.

Once set, the pitch is available until you change the pitch, or press the SHIFT and RESET keys or turn the power OFF.

Half-step positions using middle C as a reference

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j |  |  |  |  | # A |  |  | c | k-^s | — |
| 1 1 | 6 | If \* | —\* |  |  | o | 9 |  |  |  |

a=0123 45 6 78 9 10 11

Type in and execute the next program.

**10 play**

20 play

**"oG\*cdefgab"**

**"o3\***

**def gab"**

sets

middle C equal to D#

The scale in line 20 is played 3 half-steps higher than the scale in line 10.

* 1. Default Note Length

If a note doesn’t have a number following it, it’s assumed to be a quarter note. But if we have many notes that have identical durations, do we need to specify each duration each time? No. Once set, any notes following an L statement without a length indicator are played the duration specified in the L statement. This continues until another L statement is executed. It saves time and makes the notes within double quotes easier to read.

In other words,

**play "LI6"**

changes the system default from a quarter note to a 1/16 note (until another L statement is executed or the M5 system is turned OFF).

**Play "LScde"**

plays three eighth notes (C, D and an E). There is no need to separately designate each eighth note. The same thing is also true if a music rest is encountered. The rest in the next statement is an eighth rest since it’s treated identically to the C, D and E. The following F is a half note, but the next G is an eighth note.

P l a y “ L 8 c d <? r f 2 g"

I eighth rest

* 1. Getting Ready for a Performance

OK. So you have a song written on music sheets and you want to play it on your M5. What sort of things should you be aware of?

* There are two ways to program a music performance.

(both have been discussed)

1. directly in double quotes of a PLAY statement
2. using a READ to read the music data after specifying the music in DATA statements

* Your M5 can play six octaves.
* Your M5 can play three notes at the same time.
* Code both a song’s melody, accompaniment and tempo.
* Decide the type of notes for your song, e.g. whether

your song would sound better on an organ. Try several styles.

* Do you want to change the pitch of the song?
* Do you want to vary the volume in some passages?
* Do you want to have a powerful or quiet ending passage?
* Using the note name, e.g., C, D, E, is easier to read than using the note number.

• Encore!

**CHAPTER**



1. SOUND EFFECTS

You walk into a video arcade and all of a sudden you’re inundated with sound effects, missiles, explosions, bouncing ping pong balls. Your M5 can match these easily! You may even think of some sound effects that’ll blow those arcade games away.

The secret is three tone generators and one noise generator!

* 1. Precision Tone Generator

Three tone generators, channels 0~2, can be accessed independently to generate tones at precision frequencies.

**s g 0? 200? 15**

volume (0~15)

frequency (1 ~ 1023)

I channel number (0~2)

Type in the previous command after the Ready prompt is displayed.

You’ll hear frequency value 200 coming from channel 0 at volume level 15 (the loudest). Press the SHIFT and RESET keys at the same time to stop the sound.

The frequency value can vary from 1 to 1023; the frequency gets higher as the frequency value decreases. So 1 is the highest frequency and 1023 is the lowest. (O outputs a frequency that is lower than 1023).



|  |  |  |
| --- | --- | --- |
| Frequency value | 1 —1023 | When set to 0 |
| Frequency | 111.86 (kHz)—109.35 (Hz) | 109.24 (Hz) |
| Tone | High tone —Low Tone |  |

To calculate the frequency value for a particular frequency, use the next formula.

frequency value = 111.86 (kHz)/frequency

For example, the TV time signal is actually two tones, one is 440 Hz (hertz, or cycles per second) and the other is 880 Hz. The calculations are:

frequency value = 111.86 (kHz)/440 (Hz) = 254 frequency value = 111.86 (kHz)/880 (Hz^ = 127

^ frequency

Listen and compare the performance of both time tones.

100! time signal 110 for 1=0 to 2

120 sg 0,254,15

130 sleep 10, 1

140 sg 0,,0

150 sleep 50,1

160 next I

170 sg 0,127,15

180 sleep 1

190 sg 0,,0

200 .end

lines 120 and 170—notice the calculated frequency values

lines 140 and 170—the last 0 stops channel 0. Also notice the frequency is not included in this case.

sg 0? ? 0

1. turns channel 0 off

channel number

The sound volume can vary from 0—15; 15 is the loudest, 0 turns the tone generator off.

More than one tone generator can be played concurrently.

10!sg test 20 for F=1 to 1023 30 s g 0, F, 15

40 sg 1,1023—F,15

50 next F

60 sg 0,,0 70 sg l,,0 80 end

line 30—frequency varies from high to low line 40—frequency varies from low to high

* 1. Noise Generator

A tone generator, channel 3, generates eight types of noise (0~7). Type in the following, follow it with a RETURN key.

59 **3,7,15**

I volume (0~ 15)

noise type number (0~7)

noise channel number

To stop the sound, press the SHIFT and RESET keys simultaneously, or type in:

sg 3.i , 0

I volume 0 (silent)

The noise varies from types 0-7. 0-3 are different pitches while 4~7 are varieties of white noise. The next program plays every noise for a short time.

**i0 ! no i se -test:**

**20 for N = 0 to 7 30 sg 3iNi 15**

**40 print "... noise number ="?N 50 sleep 30>1 60 next N 70 sg 3>;0 80 end**

You may have noticed noises 2 and 3, and 6 and 7 are identical. This is because noises 3 and 7 are controlled by channel 2. And since channel 2 is not specified in this program, the default specifies that two noises from channel 3 are duplicated. In the following program, as the frequency value of channel 2 is varied from 1-150, the frequency of noise 7 also varies.

|  |  |
| --- | --- |
| 10 ! | no i se 7 t |
| 20 | for F=1 to |
| 30 | sg 2,F,0 |
| 40 | - sg 3, 7? 1! |
| 50 | slcep 12 |
| 60 | next |
| 70 | sg 3> > 0 |
| 80 | end |

150

line 30—set the frequency of channel 2. But since its purpose is actually to change the frequency of noise 7, the volume of channel 2 is turned off.

* 1. Live Sound Effects

A variety of sound effects are possible with one or a combination of SG statements. We’ll try several things.

• Vary the frequency

By varying the frequency dynamically, we can mimic a semblance of the sound of a UFO or a mechanical chicken (who really knows what a UFO sounds like?)

**10' sg ufo test 20 for.1=1 to 60 30 sg 2\*1,15 40 next I 50 sleep 6? 1 60 goto 20**

lines 20 to 40—change the frequency value dynamically

line 50—suspend execution for 1/10 seconds to allow separation of sounds. While execution is suspended, a sound with frequency value 60 is played.

line 60—go back to line 20 which plays the sound again. This will continue until you press the SHIFT and RESET keys at the same time.

If the SLEEP statement in line 50 is changed, it can alter the ambience. Try

50 sleep 30**j 1**

* Vary the volume

Changing the volume can also create some interesting sound effects. How about a UFO floating in mid air?

10! sg volume test 20 for 1=1 to 60 30 sg 2,200,1/4 40 next I 50 5leep 6? 1 60 goto 20

line 30—vary the volume of the frequency whose frequency value is 200

And if the frequency also varies, add some chirping birds to your morning.

10! sg volume/frequency test

20 for 1=1 to 60

30 sg 2,1,1/4

40 next 1

50 sleep 6,1

60 goto 20

line 30—the frequency varies as its volume changes

* Using the noise generator

Interesting sounds can be enjoyed by changing the frequency and volume of the noise generator. Anyone for androids tramping through their backyards?

10! sq noise generator test

2 0 f o r I = 0 t o (b 0 30 so 2,I,0 40 sq 3,7,15 50 next I 60 goto 20

line 20—change the frequency of channel 2 which in turn alters the frequency of the noise from channel 3

Changing line 40 to the following statement alters the volume continuously.

**40 sg 3,7,15-1/4**

• Use the noise generator and tone generator simultaneously

**10! sg test** 20 sg 2,10,0 **30 s g 3,7,15 40 for 1=1 to 60 50 sg 2,1,15-1/4 60 sg 3,7,15 70 next I 80 goto 40**

Now you’ve got the audio backdrop for a galaxial game.

You’ve seen how to create different sounds by effortlessly changing only a few things. Challenge yourself to come up with some of your own mind blowing sound effects.



**CHAPTER**



1. INTERRUPT MANAGEMENT

By now you should be getting a good feel for what BASIC-G can do.

Well, there’s more. In this section, we’re going to learn some BASIC-G commands and nuances that elevate you above normal users.

Interrupt management is basically a fancy term for suspending work on one thing while something else is done. In our case, it means we stop execution of our program while we do something else. After our “interrupt” is taken care of, we return execution to our program. An analogy—say your mom asks you to wash dishes while you’re watching TV. So you wash the dishes. Afterwards, you resume watching TV. In this case, the request is the interrupt and watching TV is the program executing before and after the interrupt.

BASIC-G has six statements that take care of different types of interrupts. Do you remember the DIM statement that creates an array variable? Interrupt management statements work the same way. They’re not actually executed as such; they initialize the system. Look at the next program.

10 print "tSMi"-« control codes

1. 0 o n c o i n c g o 5 u h 2 0 0

30 coi nc on 100 end

200 (subroutine to take care of the interrupt) 300 return

Let’s discuss it.

line 20—the “ON COINC GOSUB” statement takes care of any collisions between sprites. When a collision occurs, the subroutine beginning at line 200 is called (gosub 200). You may be assuming line 20 is executed when the main program is first executed, however this is not the case. The first time around, BASIC-G looks at it, checks its function and records its existence. The subroutine is not called this time. When an interrupt occurs, BASIC-G recalls that line 20 has something to do with sprite collisions and immediately calls the Subroutine. Likewise, if execution of the main program finishes before an interrupt is detected, line 20 is not executed. The main program must be active to process interrupts.

line 30—the ability to detect interrupts, in this case the collision of sprites, can be turned on and off. Before line 30 is executed, this capability is turned off; line 30 turns it on. Likewise, sprite collision interrupts are easily ignored with:

60 coinc off

lines 200 to 300—when a sprite interrupt occurs, execution of the main program is suspended, line 20 of the main program is immediately executed and the subroutine is called. After the RETURN is executed in line 300, execution of the main program is resumed where it left off. NOTE: Don’t call interrupt subroutines yourself using “GOSUB”, or even “GOTO” statements, from the main program, your program won’t work correctly.

Like line 20 above, there are several other types of BASIC-G interrupt management statements, listed below.

**INTERRUPT COMMANDS**

|  |  |  |  |
| --- | --- | --- | --- |
| BASIC-G  Statement | Characteristics | Corresponding ON/OFF Statement [see note] | Remarks |
| ON EVENT GOSUB | Takes care of EVENT timer interrupts | EVENT ON/OFF | * Priority 2 * Interrupt at set intervals |
| ON ALARM GOSUB | Takes care of ALARM timer interrupts | ALARM ON/OFF | • Priority 1 |
| ON COINC GOSUB | Takes care of sprite collision interrupts | COINC ON/OFF | • Priority 5 |
| ON KEY GOSUB | Takes care of key interrupts | KEY ON/OFF | • Priority 3 |
| ON JOY GOSUB | Takes care of joystick direction change interrupts | JOY ON/OFF | • Priority 4 |
| ON ERROR GOSUB | Takes care of detected errors |  | • Priority 0 (highest) |

NOTE: All interrupt types are initially turned off. When you want to enable a  
certain type of interrupt (turn in on), you must do it explicitly.

1. Practical Interrupt Management

The concept of interrupt management is not as trivial as first supposed. Let’s go through examples of each type of interrupt management statement.

1. ON EVENT GOSUB Interrupts



The M5 has a interval timer which can be set by a user. Each time it expires, it causes an interrupt.

10 print " SSil" control codes

**20 event 6> 60 30 on event gosub 200 40 event on 50 H=0:I=0**

**60 I iYia i n program-add i t i on 70 H=H+1**

**80 print cursor<10>10)?“ha " 5 H5 90 goto 70**

**200' Interrupt subrouti ne-subtraction 210 1=1-1**

**220 pr i nt cursor (10? 20) \* 11 i = " ; I;**

**230 return**

line 20—establishes the interval of the timer. Its format is:

**20 event 6> 60**

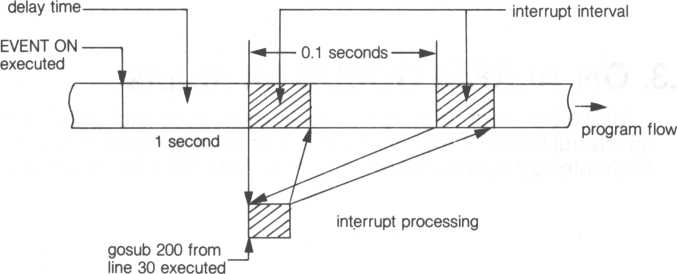
I delay time

interrupt interval

To find the actual time values (seconds) of both values, divide both by 60.

Delay time = 1 second (60/60)

Interrupt time = 0.1 seconds (6/60)



(EVENT 6,60)

The delay time is the wait time until the timer begins marking time. In this case, the timer waits 1 second, and begins counting. If the ‘60’ is changed to ‘600’, it’ll wait 10 seconds before timing. Try it out. The interrupt interval is the time interval of each new event time interrupt, every 0.1 seconds. If the ‘6’ is changed to ‘60’, it’ll interrupt every 1 second.

line 30—the function is very similar to the “on coinc gosub” statement from the previous section. In this situation, it calls the subroutine beginning at line 200 when an event timer interrupt is detected, every .1 seconds.

line 40—turns on the capability to detect event timer interrupts.

lines 60 to 90—this is our main program. It’ll continue to increment H and display the result at (10,10) on the screen. When an event timer interrupt is detected every .1 seconds, execution of this loop is suspended and the interrupt is serviced. After the interrupt is taken care of, this loop resumes where it left off. The next time an event timer interrupt is detected (.1 seconds later), this loop is again suspended.

lines 200 to 230—this is the subroutine called each time an event timer interrupt is detected. Before being executed, main program execution is suspended; when this subroutine finishes (the RETURN in line 230), execution of the main program is resumed. A point of caution—if the time taken to service an interrupt exceeds the interval between interrupts, the RETURN to the main program will never take place since the interrupt subroutine takes up all the time trying in vain to service all interrupts.

1. ON ALARM GOSUB Interrupts



Unlike the event timer, the alarm timer is a one shot timer that evokes an interrupt when it expires. It’s useful to set a game’s time limit. An alarm timer interrupt is serviced by an “ON ALARM GQSUB” statement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | pr i n + |  | -control codes | |
| 20 | t i me$ = | "00500500” |  |  |
| 30 | a l arm$ | r "00 5 02“ |  |  |
| 40 | o n a l a | rm gosub 1 | 50 |  |
| 50 | a t arm | on |  |  |
| 60 ! | lYia i n | program—ti | !Tie |  |
| 70 | print | cursor<10? | 10 > | ;t i me$; |
| 80 | goto 7 | 0 |  |  |
| 150 | ! i n t e | rrupi subr | out | i rt e-stop |
| 1 6 0 | p r i n t | c u p s o r (1 0 | , 10 | )?" END |
| 170 | 1 end |  |  |  |

line 20—sets the BASIC-G tinner to time 00:00:00 where:

TIMES = hh:mm:ss ALARMS = hh:mm

hh: hours mm: minutes ss: seconds

line 30—sets the alarm timer to go off and yield an interrupt after two minutes.

line 40—when an alarm timer is detected, call the interrupt subroutine beginning from line 150.

line 50—enables your system to detect alarm timer interrupts

lines 60 to 80—main loop of our program. It continually prints the timer value on your screen.

lines 150 to 170—interrupt subroutine. When executed, the “end” in line 170 forces our program to stop execution.

1. ON COINC GOSUB Interrupts

When the capability to detect sprites colliding is enabled (COIN ON statement), the ON COINC GOSUB statement eloquently takes care of these collisions. This ability is, to detect sprites colliding, is what most game software is all about.

"Bias"\*

-control codes

|  |  |  |  |
| --- | --- | --- | --- |
| 40 | o n c o | i nc | gosub SSI |
| 50 | co i nc | on |  |
| 6 0 | move | o n |  |
| 70 | gosub | $SP | ST |
| 80 | gosub | $SP | . MO |
| 90 | gosub | SPRINT | |
| 1 90 | end |  |  |
| 200$SP.S | | T |  |
| 21 0 | f o r | 1=0 | to 1 |
| 220 | 5C | od I | , &E1. |
| 230 | sc | ot I | , .&05+ I \*3 |
| 240 | io | c I | to 0+1\*2' |
| 250 | next | I |  |

,10 prrnt 20 erase 30 sleep

1, 1

**260 return 300\*SP.MO**

**310 move 0 step 1»0»2 320 move 1 step-1»0>2 330 return 400SPRINT 410 H=H+1**

**420 pr i nt cursor(10? 10)\* H; 430 goto 400 500\*SP.CO 510 move off**

**520 print " ! ! !co**ll**i** s **ion 1 1 1 530 coinc off 540 return 190**

line 40—calls subroutine $SP.CO when a sprite collision is detected

line 50—turns on the capability to detect sprite collision interrupts

line 200 to 260—sets up two sprites

line 300 to 330—set the two sprites in motion

line 400 to 430—execute this loop until the two sprites collide

line 500 to 530—when the two sprites collide, this subroutine is called, "collision” is printed on the screen, line 540 then forces the program to stop by returning back to the "end” in line 190.

1. ON KEY GOSUB and ON JOY GOSUB Interrupts

The mechanism of "ON KEY GOSUB” and “ON JOY GOSUB” are similar to the interrupt statements we’ve already discussed. They, naturally, service two other types of interrupts.

ON KEY GOSUB services keyboard interrupts (when a key is pressed)

ON JOY GOSUB services joypad interrupts (including the joypad ATTACK key)

Each time a key is depressed or the joypad is controlled, an interrupt occurs. When that same key is released or joypad pressure abates, another interrupt is produced. That’s right, two interrupts are generated each time.

110 pr i nt "ISIBLi" control codes

120 erase

130 on joy gosub $JOY

1. 0 on k gy g os ub $KEY
2. 0 j o y o n '■ k e y o n: m o v e o n 1601mai n program

170 gosub $SP.ST 180 gosub $SF'. MO'-.'E 190 end 200$SP.ST 210 scod 0.-&E1 220 scol 0? &05 230 toe 0 to 0?0 240 mag 0 250 return 260$SP.MOVE

270 **fo'r** 1=0 to **360** step 10 280 X = s i n (50 >I):Y = cos(50> I)

290 move 0 to X+128>Y+98>1

300 if status(0)=0 then goto 320

310 goto 300

320 next I

330 goto 260

340$JOY

350 J = jo y(0>s S = 2

360 on J + 1 go t o $JN, $ J1, **$J2s** $J3» $J4, $J5, $J6> $J7, $ J8 370$JN:return

380$J1smove 0 step 0?-S> 1 • return

390**$** J 2 s m o v e 0 s t e **p** S**—**S **>** 1 s ret u r n

400$J3:move 0 step **SjOjI:**return

410$J4:move 0 step S? S? is return

**420$J5!** iTiove 0 step 0>S> 1: return

430$J6:move 0 step-S>S>1 ' return

440$J7smove 0 step-S?0»1sreturn

4504J8:move 0 step-S?~S>1:return

460$KEY

470 move 0on 0 **-** 0 430 return

line 130 and 140—sets up handling of joypad and keyboard interrupts

line 150—turns on the capability to accept joypad and keyboard interrupts as well as moving sprites

lines 160 to 190—main program

lines 340 to 450—interrupt subroutine to process joypad interrupts. Depending on the joypad action, the sprite is moved a different direction.

line 360—if J +1 is equal to 1, a jump is made to $JN; if equal to 2, a jump is made to $J1, and so on up to 7 for $J8.

lines 460 to 480—interrupt subroutine to process keyboard interrupts.

When a key is pressed, the sprite is moved to (0,0).

* 1. ON ERROR GOSUB Interrupts

As with the other interrupt statements, the “ON ERROR GOSUB” statement is used to service interrupts. Every time an error occurs while drawing graphics pictures, the interrupt subroutine specified by the “ON ERROR GOSUB” statement is called. A typical interrupt subroutine might be used to print an error message on the screen.

One significant difference between an error interrupt and the others is that error interrupts cannot be turned on and off, they cannot be ignored.

10 on error gosub $ERR

20 p rint " mm" control codes

30 g i n i t 990 end

10000\*ERR

10 010 p r i n t " lilill" control codes

10020 print err>errl>errl $

line 10—an error interrupt causes interrupt subroutine $ERR to be called

lines 20 to 990—drawing a graphics picture

lines 10000 to 10020—interrupt subroutine

**CHAPTER**



1. CONDITIONAL STATEMENTS

Conditional statements are BASIC-G statements that perform different tasks depending on a particular condition. If one condition is satisfied, one task is done; if another condition is satisfied, another task is done, and so on.

* 1. Conditional Jump

A conditional jump is similar to the unconditional GOTO we’ve already seen. The difference is that the jump destination is dependent on a value; if the value equals 0, a jump is made to one destination, but equaling 1 causes a jump to another destination. An example will clear up any clouds.

on A goi o 100, »ABC> 250

^—if A = 3, jump to line number 250

if A = 2,jump to label $ABC

1 if A= 1, jump to

line number 100

I value to determine the jump destination

Depending on the value of ‘A’, either line 100, $ABC or line 250 is the jump destination. Only one of the jump destinations is used; execution continues from there.

* 1. Conditional Call

A conditional subroutine call is used identically to the preceding conditional jump statement. The difference, obviously, is that subroutines are called rather than jumping to a line number or label. After a RETURN is executed in the subroutine, execution resumes at the statement following the conditional subroutine statement.



The next program calls either the subroutines beginning at $A1, $A2 or $A3 depending on the value of A (input by a user).

100! condi ti onal subrouti ne ca Ll 110 ft i n t " mm" control codes

120 view

130 v i ew 0 ? 8 ? 31**j23**

140 gi n i t

150 gmode

160 view 0»0>31>9

170 fcol &0F

180 iriput A

**190 if A<<1 or A>>3 then clssqoto 180**

2 0 0 v i ew 0 > 831 > 23

210 gmove **128>** **50**

**220** on A gosub **$A** 1 .< **$A2,** **$A3**

**230** sleep 1

240 goto 110

**250SA1**

**260** circle 10 270 return **2804A2**

**290 circle** 20 300 return 3104A3

320 **circle** 30 330 return

line 190—if A is not 1,2 or 3, ask the user to input ‘A’ again

line 220—call subroutine depending on value of ‘A’

A = 1 call subroutine beginning at label $A1

A = 2 call subroutine beginning at label $A2 A = 3 call subroutine beginning at label $A3

lines 270, 300 330—return from subroutine. The next statement executed is line 230.

* 1. Conditional RESTORE

The “ON RESTORE” statement is used in the same way as the other conditional statements except that the pointer to a data statement is set instead of affecting the flow of a program.

The program following will reset different data statements depending on the value of A (input by a user).

100! condi i»onal RESTORE 110 print "mail11 120 input A

**130 if A<<1 or A>>3 then qoto 110**

140 on A restore $DATA1,\*DATA2,\*DATA3 150 read 1$

160 print cursor(10>10)\* 1$

170 sleep 1 180 goto 110 190$DATA1 200 data "one"

210$DATA2 220 data"two"

230$DATA3

240 data "three"

line 140—conditional RESTORE statement.

A = 1 set data pointer to SDATA1 A = 2 set data pointer' to $DATA2 A = • 3 set data pointer to $DATA3

lines 190 to 240—contains the data statements alluded to by line 140

* 1. Hints When Using Conditional Statements

When the value that determines the jump, call or restore destination is 0, negative, or greater than the number of destinations, an error is displayed and execution stops.

on A gosub $A1 ,$A2,$A3

^ 1

1—three subroutine destinations

I if A equals 0, is negative or greater

than 3 (in this case), an error occurs

**CHAPTER**

1. ACCURATE TIMER

Your M5 system possesses an accurate clock that can be used for a variety of applications. Combined with interrupt management statements, such as “ON EVENT GOSUB”, the timer can prove very useful.

* 1. TIMES statement

The TIMES statement sets the clock time. An example:

**t i me$ = "12s 15: 03" \***

Don’t forget the double quotes and the semicolons between the timer values. Its general format is:

TIMES = ”hh:mm:ss”

hh: hour

mm: minute

ss: second

The next program, actually very trivial, transforms your TV into a digital  
clock! All you need to do is enter the time when program execution begins.

|  |  |  |
| --- | --- | --- |
| 10 1 | ! TV digital cloc | k |
| 20 | P rint " liiiSli " -\* | control codes |
| 30 | input "Enter the | time hh:mm!ss |
| 40 | t i me$ = T$ |  |
| 50 | print cursor<10? | 10 > 51 ime$ |
| 60 | goto 50 |  |

* 1. WAIT Statement

The WAIT statement waits a prescribed time; if allowed to time out, an interrupt is generated and the appropriate interrupt subroutine is called.

wai i 30? 1

I increment (equals 60 when omitted)

I time out counter (does not wait if 0)

wait time = (time out counter) x (increment) x 1/60 Substituting the above formula for our example, wait time = 30 x 1 x 1/60 = 0.5 seconds Another example.

wa i i 2

wait time = 2x60x1/60 = 2 seconds

The next program waits 10 seconds for “BASIC-G” to be typed in from the keyboard. If this condition is not met, “TIME OUT!!” is printed.

|  |  |  |  |
| --- | --- | --- | --- |
| 100 1 | ! wa i t | iest |  |
| 110 | on er | ror gosub | $ERR |
| 120 | wa i i: | 10 |  |
| 130 | P rint | ■ " HJfeil" -•— | -control codes |
| 140 | repea | t |  |
| 150 | i np | ut A$ |  |
| 160 | uni i l | A$="BASIC | -i3" |
| 170 | els |  |  |
| 180 | pr i ni | cursor(10 | , 10);Af |
| 190 | A$ = "" |  |  |
| 200 | s l eep | jL |  |
| 210 | g o t o | 130 |  |
| 220$ERR | |  |  |
| 230 | pr i nt | "TIME OUT | i i» |
| 240 | resume | |  |

line 120—sets the wait timer to 10 seconds

lines 140 to 160—asks a user to continually type in character strings until “BASIC-G” is typed in. If a user enters something that is not “BASIC-G”, it’ll ask again. The UNTIL is useful for checkirlg character string conditions.

line 220 to 240—interrupt subroutine. If the wait time is allowed to time out, “TIME OUT!!” is printed on the screen.

line 240—the RESUME cleans up various error conditions and returns to

where the main program left off. It is also possible to return to a specific line number if specified in a RESUME statement.

* 1. SLEEP Statement

A SLEEP statement suspends execution of a program for the time prescribed in its parameters. The function of the parameter values are identical to the WAIT statement from section 17.2.

s t eep 30**j1**

^—increment (equals 60 when omitted)

I sleep counter

sleep time = (sleep counter) x (increment) x 1/60

The next program rests (sleeps) for 0.5 seconds in line 50. If line 50 is taken out, the counter from 0 to 100 is counted in a hurry. Take it out and find out for yourself.



|  |  |  |
| --- | --- | --- |
| 10 1 | ! sleep | tes |
| 20 | c l s |  |
| 30 | for 1=0 | to |
| 40 | pr i nt | I |
| 50 | s l eep | 30 |
| 60 | next I |  |
| 70 | end |  |

**CHAPTER**

18

1. MORE PRINT FEATURES

For those of us more advanced, we may need more print features. Let’s take full advantage of the M5’s system capabilities.

* 1. MPRINT, MRCRT$ Statements

The MPRINT statement allows a user to separate a character string and print different segments on different lines.

10 c l s

20 pri nt cursorC10? 10)? 30 mp r i n t (2 ? 3) ?" abc d e f11

character string

characters/line-\* ■-number of lines

The following program cuts out an 8 by 8 character viewport from screen buffer 0, reads characters displayed from the screen (MRCRTS—works similar to MPRINT except that it reads characters rather than printing them) and displays the same characters on screen buffer 1.

100 print " fiMiWUMilUJ"«\* control codes

**110 console 1?1?1**

120!

**130! page 0 print**

**140!**

**150 ten 64 160 view**

**170 print cursor(10?9)?"01234567"\***

**180 for Y = 0 to 7**

**190 pr i nt cursor <18?10+Y);Y?**

**200 next Y**

210 view 10?10? 17?17 220 els 230 stop

**240 !**

**250! page 0 mr c rt**

**260 !**

**270 pr i nt ""?cursor (0?0)?**

**280 M$ = mrcrt$(8? 8)**

290 !

**300!** page 1 print

310!

**320 p**ri n t "";cursor(10 >10)5 330 mpr i nt (8> 8) 5 340 end

line 110—turn on screen lock

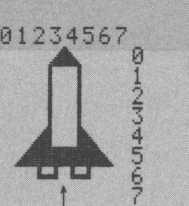
line 270—read the characters from screen buffer 0 copy the character string to MS

line 320—divide the character string into eight lines of eight characters, then print the result matrix of characters

When run, this program allots an 8 by 8 viewport surrounded by a matrix scale on screen 0. Draw a picture within this viewport. Try your hand at using the graphics characters—don’t press the RETURN key; if you do, your picture will be ruined. Here’s an example.

Screen 0 display

Draw using graphics keys

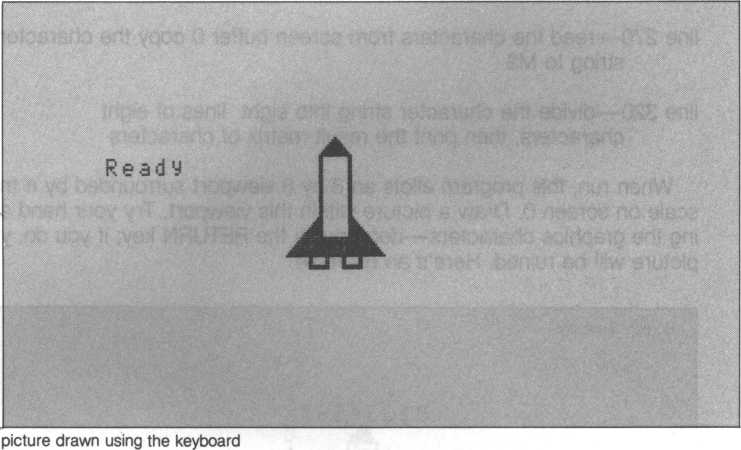


When your picture satisfies your aesthetic tastes, use the CTRL V combination to change to screen buffer 1. And type in the CONT command followed by the RETURN key.

CONT



Your creation is now displayed on screen buffer 1.



**CHAPTER**



1. MORE SPRITE FEATURES
   1. DIST Statement

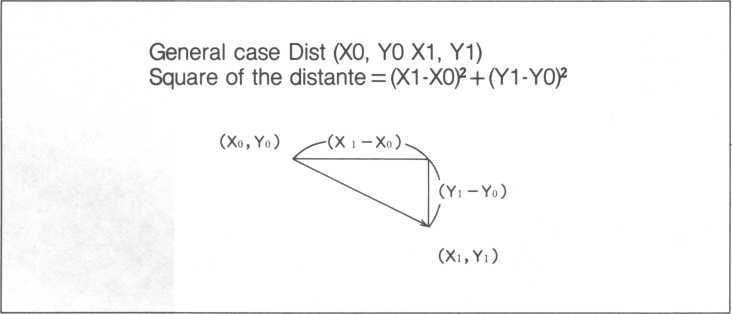
The DIST function returns the square of the distance between two sets of specified coordinates—very useful for calculating the distance between .two sprites. Type in the following followed by a RETURN key.

**print: d i st (100? 120? 200? 220)**

—r-—1 4—j

one set of coordinates 1 Lone set of coordinates

The square of the distance is displayed, 20000. How is this ascertained? Look below.



If the square of the distance is greater than 32767, a -.1 will be returned. The program directly following finds the distance between the center coordinates of two sprites.

100! distance test 110 print **"yyiai?"**

120 erase

130 g i n i t

140 for 1=0 to 1

150 seed I,&E1

16 0 s c o l I ? 02+ I \* 3

170 loc I to **128,96**

180 next I

190 for 1=0 to 1

20 0 X = r n d (2'5) s Y = r n d (191)

210 move I to X, Y, 1 220 next I 230 F=0

240 for K=0 to 1 250 F = s t a t us (K) +F-

260 next K

270 if F>0 then goto 230 280 draw30,31 290 D = d i s t C 3 0,31)

30 0 p r i n t c ur s o r (0, 0 > ; s p r i n t # 1 D;

310 sleep 2 320 g i n i t 330 goto 190

lines 140 to 180—sets up two sprites

lines 190 to 220—move the two sprites off in separate random directions

lines 240 to 260—this loop continues to be executed until both sprites stop moving

line 280—draw a line between the two sprites

line 290—calculate the square of the distance between the two sprites and put the result in the variable ‘D’

* 1. DRCT Statement

The DRCT statement returns the relationship between two sets of coordinates. Try the following command after the Ready prompt is displayed.

**Print** drct(128\*96>140,i 20)



**L**

one set of coordinates—

one set of coordinates

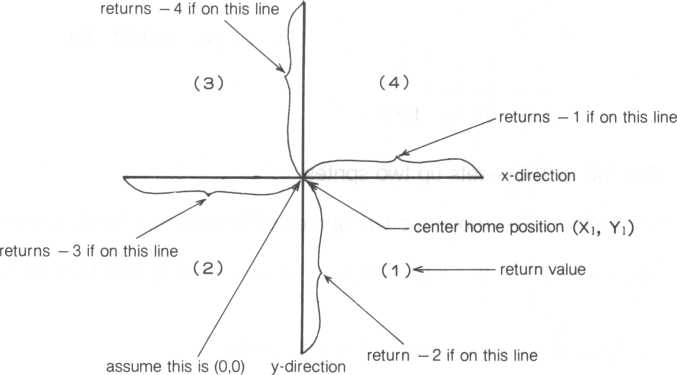
The relationship between the two sets of coordinates is displayed, T. What does the ‘1’ mean? Assume (X1,Y1)—(128,96) in this case—is the center of an X-Y grid. The quadrant of (X0,Y0)—(140,120) in this case—relative to (X1,Y1) determines the return value.

If (X1,Y1) is assumed to be (0,0), the position of (X0,Y0) determines the return value, illustrated below.

DRCT (Xn.Yo.Xi, Yi)

1—I—1

1 assume this is the center of the grid



The coordinates of two sprites can be inserted into (X0,Y0,X1,Y1) to find their relationship. The following listing shows a program to find the relationship between the center coordinates of two sprites, one is green and the other blue. Assume (128,96) is (0,0).

100! drct "test 110 print **"LUUiaiS"**

120 erase

130 for 1=0 to 1

140 seed I ? &E 1

150 scol I,£02+1\*3

160 loc I to 128?96

170 next

180! line

190 for X = 0 to 31

200 pri nt cursor(X? 12)\*‘

210 next

220 for Y=0 to 23

230 pr i nt cursor (16? V) \*"!■"\*

240 next

250 pr i nt cursor(16?12)?"+"?

**260!** move sprite **270** f **o r I = 0 t o 1 280 X = r n d (255)** '• Y = r n d (191)

290 move I to X?Y?2 300 next I 310!dret

320 DR0 = drct(30? **128?** **96)**

**330** DR1= d r c t(31? 128 ? 96)

340 pr i nt cursor (0? 1)5"green";DR1’ 350 pr i nt cursor(0?0)'"blue"? DR0? **360!**

**370** F=0

**330** for K = 0 to 1 **390** F = status (K)+F 400 next K

410 if F>0 then goto 320 420 g o t o 27 0

lines 130 to 170—set up two sprites, one is green (sprite 0) and the other is blue (sprite 1)

lines 180 to 240—draw the grid lines 260 to 300—move the sprites

lines 320 and 330—assume (128,96) is the center of the grid. The relationships of sprites 0 and 1 are stored in DR0 and DR1, and later displayed on the screen.



1. KEYBOARD FEATURES 20.1. INKEY Statement

The INKEY statement is used to ascertain if a key on the keyboard is pressed, and if so, which key is pressed. For a complete list of keys and their corresponding codes, refer to Appendix F.

100! inkey test 110 print **"ISJSIS"**

**120** console 1 - 0

130 erase

140 g os u b **$3P.ST**

**150 gosub $KEY**

**160** **end**

170$SP.ST

**180** scod 0 > **.?•€ 1**

|  |  |  |
| --- | --- | --- |
| 190 | sco l 0 < 2:05 |  |
| 200 | return |  |
| 210$KEY | |  |
| 220 | N=5 sX =128!Y = 96 |  |
| 230 | if inkey(0)=51 | then Y=Y-N |
| 240 | if i nk ey(0)=55 | then X=>X+N |
| 250 | if inkey(0)=46 | then Y=Y+N |
| 260 | if inkey(0)=54 | then X=X—N |
| 270 | if X<0 then X-0 |  |
| 280 | if X>245 then X | = 245 |
| 290 | i.f Y< 0 then Y = 0 |  |
| 300 | if Y>180 then Y | = 180 |
| 310 | loc 0 to XjY |  |
| 320 | goto 230 |  |

line 120—disables some of the automatic functions of the keyboard. For example, pressing the FUNC key and the R key no longer generate a RUN command. If you want to set your keyboard back, type in

console 1?1

followed by the' RETURN key. See the statement summary in Appendix B for a description of all CONSOLE parameters.

lines 230 to 260—using the cursor control keys, the sprite can be moved either up, down, to the left or right depending on the cursor control key pressed



1. PERIPHERALS

There is optional equipment to expand the repertoire of possibilities for your M5 system. For instance, a printer, a joy pad and P-EDIT (an application program for graphics) extend your system to capabilities previously enjoyed only by larger and more expensive systems.

21.1. The Printer

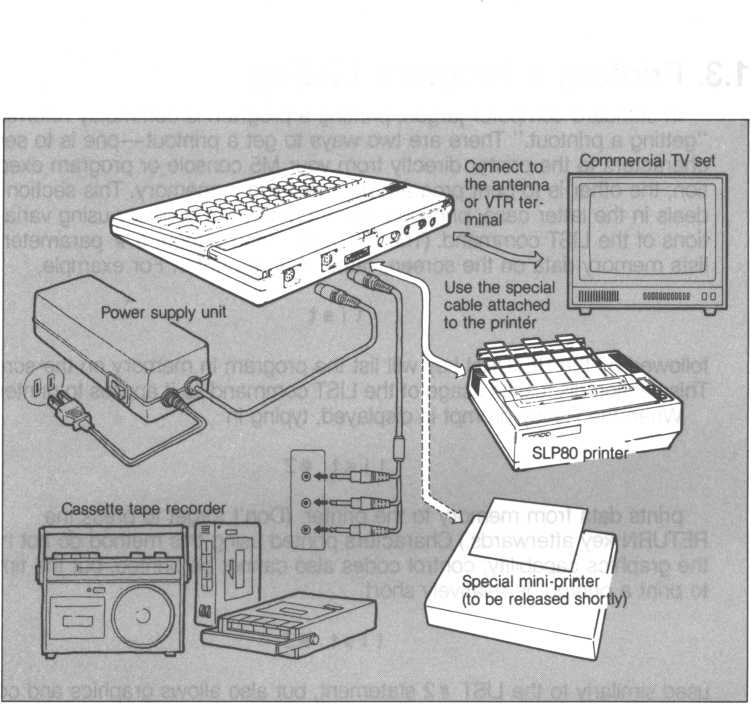
A printer can be attached to the M5 through the parallel port.

NOTE: Before using your printer, use the PMODE command to initialize your system for printer operation (refer to section 21.5). This setting will be effective until it is reset or power is turned off.

* 1. Before Using a Printer

A printer is a precision machine. A version of the familiar maxim applies here, “Take care of your printer, and it’ll give you many years of faithful service.” Before using it, read the instruction and maintenance manuals specifically for your printer.

Connecting the printer and your M5 computer together is simple. Follow the next illustration. The cable runs from the printer to the M5 console. If you have trouble getting your printer to work, the independent dealer for your printer can provide advice.



Before turning the printer’s power switch on, turn on power to your M5 console first. The reverse is true. Turn off the printer before turning off power to your M5 console.

Then go through the next sections to learn about printing data on your printer. Try the examples, make up some of your own.

* 1. Printing a Program Listing

In standard computer jargon, printing a program is commonly referred to “getting a printout.” There are two ways to get a printout—one is to send characters to the printer directly from your M5 console or program execution; the other is to print program data residing in memory. This section deals in the latter case, printing program data from memory using variations of the LIST command. (The LIST command without a # parameter lists memory data on the screen, rather than a printer. For example,

list

followed by the RETURN key will list the program in memory on the screen. This section discusses usage of the LIST command as it applies to printers.) When the Ready prompt is displayed, typing in

list #2

prints data from memory to the printer. (Don’t forget to press the RETURN key afterwards.) Characters printed using this method do not have the graphics capability; control codes also cannot be printed, but the time to print a printout is relatively short.

**list #3**

used similarly to the LIST # 2 statement, but also allows graphics and control codes to be printed. Other commands, LISTC, ELIST, ELISTC, are also available.

LIST [ft2or #3]

BASIC-G reserved words, such as “print” and ”for-to-next” are printed in lower-case while user specified names are printed in upper-case characters. Character strings are printed as specified by the user (upper- and lower-case).

LISTC [#2 or #3]

Character strings are printed as specified by the user. Other characters are printed in upper-case characters.

ELIST, ELISTC [#2 or #3]

This functions similar to the LIST and LISTC commands. However, the screen is cleared before printing.

To print the program in memory, use L i st #2

or,

**list #3**

To print one line of the program in memory, use

**list #2,200,200**or,

**list #3,200,200** I I

1. line number you want to print out

When a range of line numbers is to be printed, use

**list #2,30,300**or,

**list #3,30**

last line number of the range (if omitted, it prints until the last line is printed) first line of the range (if omitted, it prints from the beginning of the program)

* 1. Printing Directly to a Printer

As we said earlier, there are two ways to get a printout—one is to send characters to the printer directly from your M5 console or program execution; the other is to print program data residing in memory. Section 21.3 dealt with the latter case, this time we’ll discuss sending characters directly to the printer from the M5 console or during program execution with a PRINT command. (Similar to the LIST command, executing a PRINT command without a ft parameter will send data to the screen and not to the printer.)

pr i nt#2> "BASIC-G Sy s tem"

1—printed character string enclosed by double quotes (upper- and lower-case characters)

prints the character string, "BASIC-G System” on the printer.

**pr i n t #2'A**

I—numeric variable ‘A’ (the printed value

varies according to the value of ‘A’)

prints the value of numeric variable ‘A’ on the printer. This type of PRINT statement is easily inserted into a program. The following program prints 0 to 255 and their hexadecimal equivalents to the printer.

**10! base 10 <—> base 16 20 for 1=0 to 255**

**30 pr i nt#2> I ' " (dec i nia l) " ; " = " ; hex$ (I) ? " (hsxadsc i ma l)" 40 next I 50 end**

Using ‘ft3’ with the PRINT statement is similar to the “LIST 3” command. The only significant difference is that “PRINT 3” permits printing of characters constructed with a STCHR statement as well as control characters.

* 1. Printing Graphics Displays

By using the GCOPY (graphics copy) command, you can print graphics images. How about an exercise? Type in and execute the following program.

10 print " Liiaa" •« control codes

20 for 1=32 to 255 30 print chr$(I);

40 next I 50 end

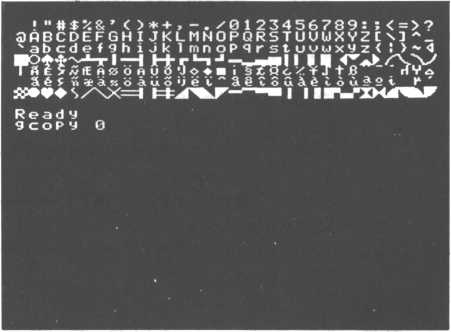
The characters represented by ASCII &20-&FF are displayed at the top half of the screen. (Don’t forget the semicolon in line 30).

When the Ready prompt is displayed, type in

**g c o p y 0**

I image number

followed by the RETURN key. The image copy of the screen is displayed.



Another program.

100! grriode "test

110 p rint " iliUlg"«« control codes

**120 ginit**

**130 gmode 4**

**140 erase**

**150 fee l A-OF**

**160 GY1= 30:GY2 = 5sH = 0**

**170 Plot 228, 0**

**180 for TH = 0 to 6\*180 step 8 190 H=H+1**

**200 X = cos C80--H/2, TH)**

**210 GY 1=GY1 + 1 220 GX=128+X**

**230 GY2 = GY2 +1**

**240 draw GX,GY2**

**250 giyio v e 128, GY1**

**260 draw GX,GY2 270 next TH 280 print#2,**

**290!**

300 gmode 0

**320 for G=0 to 7**

**330 p r i n t # 1, c u r s o r (1,21 > 5 " GCOPY " ; G 5**

**340. gcopy G**

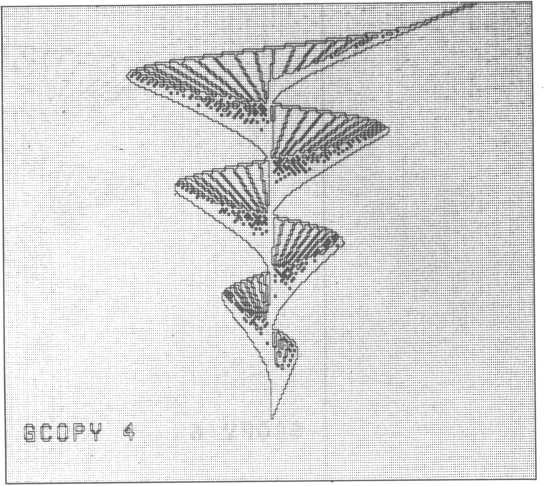
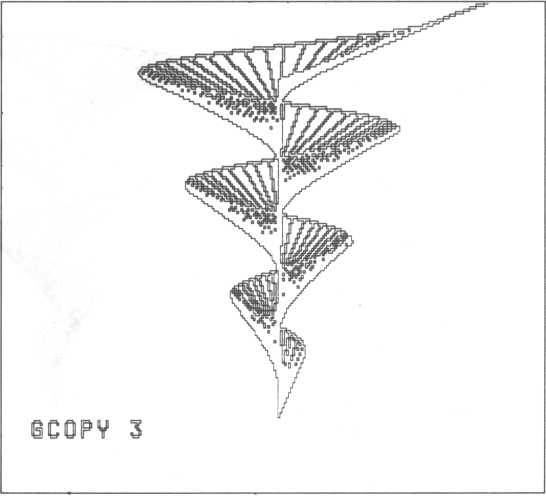
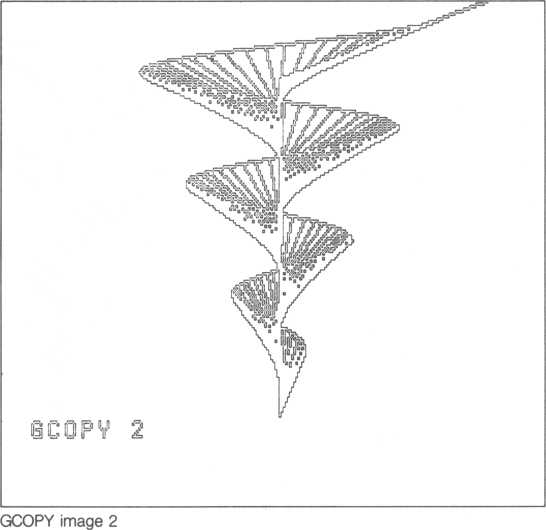
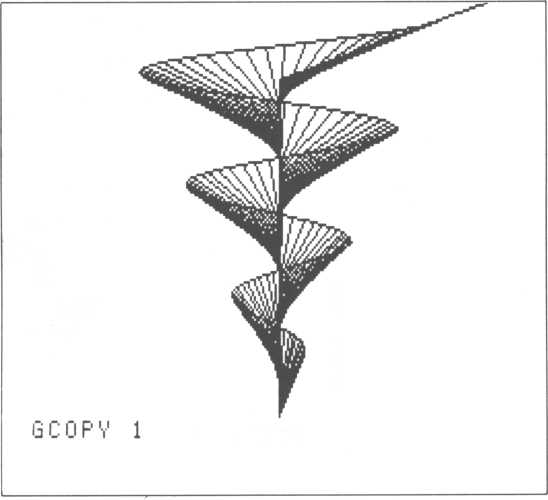
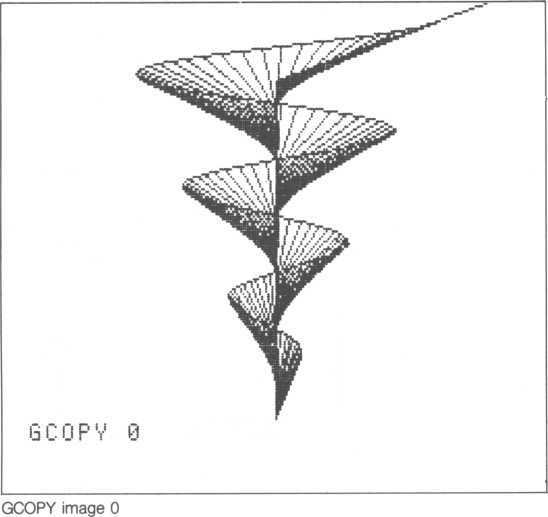
350 pr i nt#2, "11"; control code L

**360 next G**

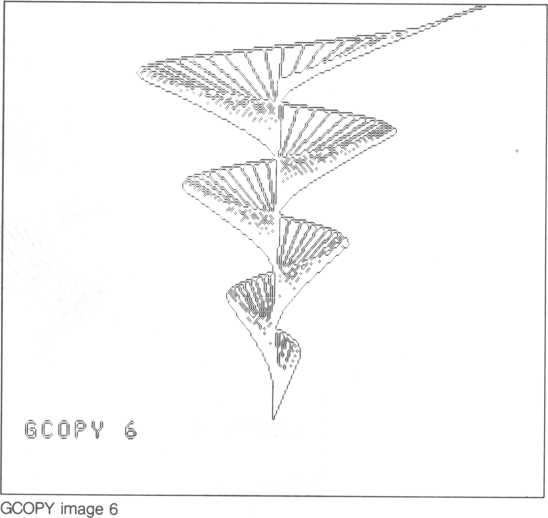
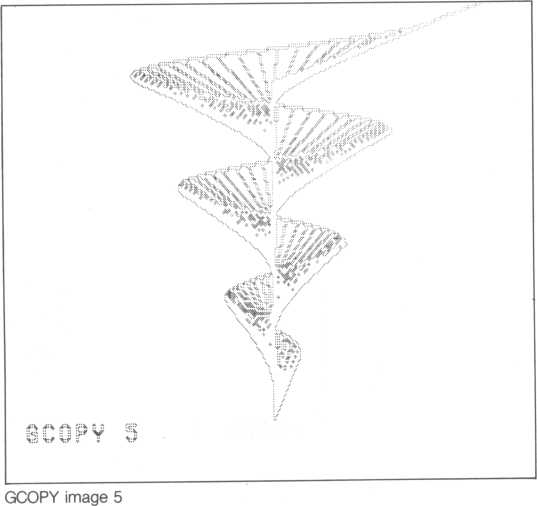
370 pr i n t "0" 5 control code Z

**380 end**

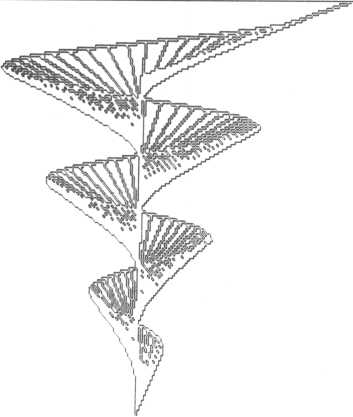
This program displays eight (0~7) different GCOPY images. “GCOPY 0” is shown nearly the same size, while “GCOPY 1 ~7” is almost double the size of “GCOPY 0” with each image having its own characteristics.



GCOPY image 4



acopv ?



* 1. PMODE Statement

The PMODE statement is used to specify the characteristics of a printer, for example, the maximum number of characters per output line. Type in the next statement when the Ready prompt is displayed to specify the maximum number of characters per output line at 80.

pm ode 80»3>0

I image characteristic

' POUT characteristic

longest output line

Refer to Appendix B for a complete listing of PMODE parameters.

Before printing images with the PRINT #3 statement, specify the number of characters with a PRINT #2 statement.

**pr i rvt#2> "4-K" [[2]](#footnote-2)chr$<AF8) ? chr\*<400)**

escape sequence

1 number of bytes to be

printed (higher-order byte)

number of bytes to be printed (lower- order byte)

The escape sequence is a clue to the printer that it will shortly receive some printer specific information, or two bytes, &f8 and &00. If we combine the lower- and higher- order bytes, we get &00F8, or 248 (decimal) bits per line. But one character is actually composed of eight bits; dividing 248 by 8 yields 31. Therefore, 32 characters, (0~31), can be printed. The next program prints out 32 image characters numbered from 0~31.

|  |  |  |
| --- | --- | --- |
| 10 | pr i nt#2> 11 i- K"" c h r $ (&F 8) 5 c h r $ < | .?<00) ; |
| 20 | for 1=0 to 31 |  |
| 30 | pr i nt#3> chr$(1)5 |  |
| 40 | next I |  |
| 50 | pr i nt#2 |  |
| 60 | end |  |

When printing out images, specify the number of characters which will be printed in advance.

NOTE: If the number of characters printed out with a PRINT #3 is less than the number previously specified, the printer sometimes stops functioning. Likewise, when the number of characters printed out is greater than the number specified, random characters are sometimes printed. If either of these situations occur, turn the printer OFF and then back ON.

* 1. P-EDIT

P-EDIT is an optional application program that supplies several specialized drawing capabilities for graphics pictures. Three typical applications include:

* Draw pictures using P-EDIT

Graphics can be drawn directly from the keyboard without using BASIC-G.

* Background P-EDIT pictures

A picture drawn by P-EDIT can be saved on cassette tape. When retrieved, the picture can be part of a backdrop for animated sprites. •

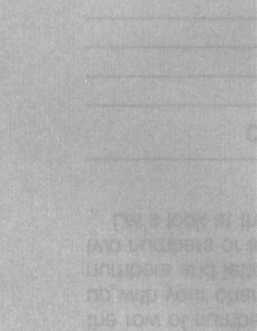
When loading pictures drawn using P-EDIT, use the following program.

|  |  |  |
| --- | --- | --- |
| 10 | c o n s o | le, , ,0, 0 |
| 20 | P r i n t | " ia" control code |
| 30 | ' o l d s o | Id |
| 40 | g i n i t | 1 |
| 50 | c o n s o | (.€\*>>> 0 > 1 |

237



**XIQNBddV**



APPENDIX A—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

2345 0 3 P ! IRQ " 2 B R

|  |  |
| --- | --- |
| CScsi | m9Q9 of |
| DT d t- | v| w c vi U. S |
| EUeuJ | 4 it 7! S k/ |
| FUf m-i | r|fi¥ a£\ |
| ij hj g 1.1.1 | \MbSx iX |
| HXh x— 0= | |
| I v i y H |  |
| JZJz | r Jk, U \_ U. - : |
| KCk H | hr & \* 2+ |
| LM ! 1 | P/''J ' iA |
| M ] rii 1 | LWHe 1 |
| NAn\*! >VVry | |
| O-o <|J "r | |

0

1

\*“i

*jL*

*7*

4

5

1. I
2. 9 fi B C D E F

13

mm mm **m** a an ia iy 13 UJ

mm

JJS\*

11 *n* y *m mm*

Li \*\* lil <- !-il t

10 4

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DEC | (HEX) | CODE | DEC | (HEX) | CODE | DEC | (HEX) | CODE |
| 32 | (20) |  | 71 | (47) | G | 110 | (6E) | n |
| 33 | (210 | i | 72 | (48) | H | 11 1 | (6F) | 0 |
| 34 | (22) | II | 73 | (49) | I | 1 12 | (70) | P |
| 35 | (23) | # | 74 | (4A) | J | 1'13 | (71) | q |
| 36 | (24) | \* | 75 | (4B) | K | 114 | (72) | r |
| 37 | (25) | X | 76 | (40 | L | 115 | (73) | 5 |
| 38 | (26) | <§c | 77 | (4D) | M | 116 | (74) | t |
| 39 | (27) | \*> | 78 | (4E) | N | 117 | (75) | U |
| 40 | (28) | ( | 79 | (4F) | 0 | 1 18 | (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 | (20 | > | 83 | (53) | S | 122 | (7A) | 2 |
| 45 | (2D) | - | 84 | (54) | T | 123 | (7B) | < |
| 46 | (2E) | . | 85 | (55) | U | 124 | (70 | i  i |
| 47 | (2F) | / | 86 | (56) | V | 125 | (7D) | > |
| 48 | (30) | 0 | 87 | (57) | w | 126 | (7E) |  |
| 49 | (31) | 1 | 88 | (58) | X | 127 | (7F) | < |
| 50 | (32) | 2 | 89 | (59) | Y | 128 | (80) | m |
| 51 | (33) | 3 | 90 | (5A) | z | 129 | (81) | o |
| 52 | (34) | 4 | 91 | (5B) | [ | 130 | (82) | ♦ |
| 53 | (35) | 5 | 92 | (50 | \ | 131 | (83) | \* |
| 54 | (36) | 6 | 93 | (5D) | ] | 132 | (84) |  |
| 55 | (37) | 7 | 94 | (5E) | /s | 133 | (85) | JL |
| 56 | (38) | 8 | 95 | (5F) |  | 134 | (86) | T |
| 57 | (39) | 9 | 96 | (60) | l | 135 | (87) | 1 |
| 58 | (3A) | •  • | 97 | (61) | a | 136 | (88) |  |
| 59 | (3B) |  | 98 | (62) | b | 137 | (89) | H |
| 60 | (30 | < | 99 | (63) | c | 138 | (8A) | h |
| 61 | (3D) | = | 100 | (64) | d | 139 | (8B) | + |
| 62 | (3E) | > | 101 | (65) | e | 140 | (80 | r |
| 63 | (3F) | ? | 102 | (66) | f | 141 | (8D) | L |
| 64 | (40) | <D | 103 | (67) | g | 142 | (8E) | -1 |
| 65 | (41) | A | 104 | (68) | h | 143 | (8F) | J |
| 66 | (42) | B | 105 | (69) | i | 144 | (90) | — |
| 67 | (43) | C | 106 | (6A) | j | 145 | (91) | — |
| 68 | (44) | D | 107 | (6B) | k | 146 | (92) | ■i |
| 69 | (45) | E | 108 | (60 | l | 147 | (93) | ■ |
| 70 | (46) | F | 109 | (6D) | m | 148 | (94) | 1 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DEC | (HEX) | CODE | DEC | (HEX) | CODE | DEC | (HEX) | CODE |
| 149 | (95) | 1 | 192 | (CO) |  | 235 | (EB) | \* |
| 150 | (96) | 1 | 193 | (Cl) | % | 236 | (EC) | A |
| 151 | (97) | ■ | 194 | (C2) | 6 | 237 | (ED) |  |
| 152 | (98) | ◄ | 195 | (C3) | f | 238 | (EE) |  |
| 153 | (99) | yr | 196 | (C4) | S | 239 | (EF) | r |
| 154 | (9A) | jkk | 197 | (C5) | \* | 240 | (FO) | — |
| 155 | (9B) | ► | 198 | (C6) | 4 | 241 | (FI) | — |
| 156 | (90 | S' | 199 | (C7) | 8 | 242 | (F2) | a. |
| 157 | (9D) | V | 200 | (C8) | 0 | 243 | (F3) | ■ |
| 158 | (9E) | ■\*\ | 201 | (C9) | 4 | 244 | (F4) | 1 |
| 159 | (9F) | J | 202 | (CA) | ij. | 245 | (F5) | 1 |
| 160 | (AO) | \ | 203 | (CB) | 0 | 246 | (F6) | 1 |
| 161 | (Al) | 8 | 204 | (CC) | »j | 247 | (F7) | ■ |
| 162 | (A2) | t | 205 | (CD) | H | 248 | (F8) | % |
| 163 | (A3) | 5 | 206 | (CE) | X | 249 | (F9) | ■■ |
| 164 | (A4) | .V | 207 | (CF) |  | 250 | (FA) | X |
| 165 | (A5) | if | 208 | (DO) |  | 251 | (FB) | H |
| 166 | (A6) | H | 209 | fDl) | e | 252 | (FC) | d |
| 167 | (A7) |  | 210 | (D2) | A  l | 253 | (FD) |  |
| 168 | (A8) | ti | 211 | (D3) | A  1\*1 | 254 | (FE) | fc |
| 169 | (A9) | A | 212 | (D4) | A  U. | 255 | (FF) | r |
| 170 | (AA) | ij | 213 | (D5) | k |  |  |  |
| 171 | (AB) | S | 214 | (D6) |  |  |  |  |
| 172 | (AC) | ■J | 215 | (D7) | i |  |  |  |
| 173 | (AD) |  | 216 | (D8) | 0 |  |  |  |
| 174 | (AE) | ♦ | 217 | (D9) | u. | DEC-\*- | denotes base 10 | |
| 175 | (AF) | ■ | 218 | (DA) | 4 | HEX-\* | denotes base 16 | |
| 176 | (BO) | 1 | 219 | (DB) | o |  |  |  |
| 177 | (Bl) | § | 220 | (DC) | i |  |  |  |
| 178 | (B2) | i | 221 | (DD) |  |  |  |  |
| 179 | (B3) | a | 222 | (DE) | F- |  |  |  |
| 180 | (B4) | 6 | 223 | (DF) |  |  |  |  |
| 181 | (B5) | '/ | 224 | (EO) |  |  |  |  |
| 182 | (B6) | ¥ | 225 | (El) | • |  |  |  |
| 183 | (B7) | I | 226 | (E2) |  |  |  |  |
| 184 | (B8) | t | 227 | (E3) | ♦ |  |  |  |
| 185 | (B9) | JB | 228 | (E4) | S |  |  |  |
| 186 | (BA) |  | 229 | (E5) | / |  |  |  |
| 187 | (BB) |  | 230 | (E6) | \ |  |  |  |
| 188 | (BC) |  | 231 | (E7) | X |  |  |  |
| 189 | (BD) | ri | 232 | (E8) | zz |  |  |  |
| 190 | (BE) | V | 233 | (E9) |  |  |  |  |
| 191 | (BF) | a | 234 | (EA) |  |  |  |  |

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 | 0 |
| B | 2 | 02 | Return cursor to beginning of current line | 13 |
| C | 3 | 03 | Scroll screen display down | is |
| D | 4 | 04 | Shift screen display left | la |
| E | 5 | 05 | Scroll screen display up | 13 |
| F | 6 | 06 | Shift screen display right | i3 |
| G | 7 | 07 | Bell | (3 |
| H | 8 | 08 | Backspace | 13 |
| 1 | 9 | 09 | Tab the cursor eight spaces | If |
| J | 10 | 0A | Move cursor down one line | y |
| K | 11 | 0B | Move cursor to home position | la |
| L | 12 | OC | Clear screen display | IS |
| M | 13 | 0D | Same as RETURN key |  |
| N | 14 | 0E | Move cursor to beginning of next line | 121 |
| 0 | 15 | OF | Change to standard mode | (3 |
| P | 16 | 10 | Change to insert mode | 13 |
| Q | 17 | 11 | Change to multi-color mode | (SI |
| R | 18 | 12 | Change to Gil graphics mode | 13 |
| S | 19 | 13 | Change to Gl graphics mode | a |
| T | 20 | 14 | Return to text mode | u |
| U | 21 | 15 | Change to visible screen | is |
| V | 22 | 16 | Alternates between the visible and invisible screens, input is sent to the displayed screen | IS |
| w | 23 | 17 | Same as RETURN key | IS |
| X | 24 | 18 | Delete characters to the right of cursor | Si |
| Y | 25 | 19 | Alternates between the visible and invisible screens only | IS |
| z | 26 | 1A | Writes input to the alternate screen | a |
| [ | 27 | 1B | Ignore |  |
|  | 28 | 1C | Right arrow |  |
| ] | 29 | 1D | Left arrow | «r- |
|  | 39 | 1E | Up arrow | T |
|  | 40 | 1F | Down arrow |  |

COLOR CODES

**Color**

**No color**

**Black**

**Green**

**Light green** Deep blue **Light blue Deep red**

**Cyan**

**Red**

**Light red** Deep yellow **Light** y**ellow Deep Green** Purple

**Gray**

White

**Color code** 0 **1 2**

**3**

**4**

**5**

6

**7**

1. **9** A B **C** D E F

APPENDIX B—BASIC-G STATEMENT TYPES

Keep in mind these capabilities when using BASIC-G.

* Maximum program size - 16K bytes
* Decimal, hexadecimal and letter characters

base 10 10,-25,-1-1001

base 16 &0A013, &0F, &205

character ‘&30’ = “0”, ‘&21 ’ = “!”

character string ‘‘ABC’’, “0123”

* Variable names up to 32 letters and numbers consisting of letters and numbers (the first character must be a letter)
* Integer and character string variables—default is a maximum of 18 characters in a character string; but it’s possible to extend it to 255 characters with the LEN statement
* Different types of variables

A integer variable

A(0) array variable

AS character string variable

A$(1) character string array variable

* Multiple number of statements on one line
* Indentation
* Only integer arithmetic
* Array up to 255 elements (integer and character arrays)
* Most statements can be executed in Ready prompt mode
* Music
* Graphics
* Noise generator
* Interrupt processing for faster programs—6 levels
* User control of sprites
* Interval timer
* 256 by 192 dot matrix graphics
* Characters can be printed on paper
* Screen hardcopy
* Line numbers (1 through 32767) should precede BASIC-G statements followed by a RETURN key

line-number statement RETURN

* Statement labels allow easy branching without referencing line-numbers (up to 32 characters)
* Graphics coordinates use (x,y) matrix system for graphics and sprites
* Different sized sprites

8x8 dot matrix 16x 16 dot matrix 32 x 32 dot matrix

* Easy sprite movement
* Four methods to express graphics coordinates

X,Y ... relative to the base corner (upper left corner) of the coordinates’ viewport

STEP X,Y ... relative to the graphics cursor # N ... sprite base corner coordinates @N ... sprite coordinates (center of a sprite)

These.four types of graphics coordinates are called “GR coordinates.”

* Legend of this BASIC-G statement list

{CR | : } follow with RETURN key or semicolon,

\* also available with BASIC-1 [ ] • optional

and so on

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| $ | $< label name> |CR | : ) Up to 32 characters (\*) |
| DIM | DIM<array-name> (<array-size> (OR | :) Sets up an array of up to 255 elements, 1 -255 (\*) |
| GOTO | GOTO < line-number > |CR | :) Transfer execution to line-number (\*) |
| Gosub | GOSUB < line-number> (OR | : )  Call subroutine beginning at line-number (executed after return address from the subroutine is pushed onto the stack (\*) |
| RETURN | RETURN [<line-number>] (OR | : j Return to the calling program (line-number after the GOSUB). If the optional line-number is included, branch to line-number. It can also be used to exit from the middle of a FOR-TO-NEXT loop. (\*) |
| (LET) | LET <variable> [,..] = <expression> (OR | : j Assign the result of the expression to variable. An expression can also consist of a variable. (\*) |
| PRINT | PRINT [# <ID>] [<expression> [{,|;)--] ] |CR | : ) Print data to  <ID number>   1. = CRT (Default) 2. = CRT (Image) 3. = POUT 4. = POUT (Image)   When the ID is expressed in BASIC-I, be sure to insert a comma between the #ID and the expression. Data between double quotes is printed as is. (\*) |
| IF..THEN..ELSE.. | IF < conditional -expression >  THEN <expression > [:••]  [ELSE <expression> [:■■]] |CR | : )  If the conditional-expression is satisfied the THEN expression is executed, otherwise the ELSE expression is executed. (\*)  Conditional-expressions are:  = ,<>(<>),>,>=(=>),<,<=(=<) |
| FOR..TO..(STEP..) | FOR < control-variable > = < initial-value >  TO < final-value >  [STEP < step-value > ] |CR | : )  Control variable is incremented from initial-value to final-value in steps of .step-value (looping).  Used in conjuction with the NEXT statement. (\*) |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| NEXT | NEXT [<control-variable> [,..]] [OR | : j Last statement of a FOR..TO..(STEP) statement. Control-variable allows matching more than one FOR-TO-NEXT loop with one NEXT statement. (\*) |
| OUT | OUT < port-number>, [< output-data > [,..]] fCR | :}  Send output-data to port-number, an expression can also be evaluated and sent. More than one byte can be output if separated by commas. (\*) |
| POKE | POKE <memory-address> [<data> [,..]] (OR | :) Write data into CPU memory-address, an expression can also be evaluated and written. If more than one data item is specified and separated by commas, they are written in contiguous increasing addresses. (\*) |
| VPOKE | VPOKE <memory-address>, <output-data> (CR'|:}  Write data into VIDEO memory-address, an expression can also be evaluated and written (\*) |
| READ | READ <variable> [,..] (OR |: J Assign data specified in bATA statement. More than one data item can be read if data items are separated by commas. (\*) |
| DATA | DATA < constant > [,..] {OR |:)  Specify data read by a READ statement, data can be characters as well as decimal and hexadecimal numbers (\*) |
| CHAIN | CHAIN [<file-name>] {OR | :)  Retrieve and execute a program stored in external memory (\*) |
| INPUT | INPUT [<prompt>] (,|;) <variable> [,..]  (CR | : J  Read a character from the keyboard. (An input time limit can be set up with the WAIT statement. If a timeout occurs, a timeout error message is displayed which is detected by an INPUT statement.) A prompt can be supplied by the user, it’s suggested the prompt be a coherant character string—in this case, the ’?’ prompt is no longer given. (\*) |
| REM | REM <character-string > |CR)  User remarks—not executed by your M5 computer. (\*) |
| ! | ! < character-string > (OR)  An exclamation point T, in the column directly after the line number also designates that line as a remark. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| RANDOMIZE | RANDOMIZE [<numeric-value>] (OR | : | Initialize the random number generator (\*) |
| RESTORE | RESTORE [< line-number>] (OR |:)  Set up data pointer to READ statement (\*) |
| CLEAR | CLEAR [<work-field >] [,< last-user-field >]  (CR | :)  Set up work-field up to last-user-field used as a character buffer and a PAINT coordinate work area. Increments of 256 bytes can be set up. The user can use CPU RAM after the last address used by BASIC-G. (\*) |
| CLS | CLS [<initialize-code>] [CR | : )  Clears the screen. When the optional numeric- expression is given, the corresponding character also clears the screen. (\*) |
| VIEW | VIEW [<X0>,<Y0>,<X1 >,<Y1 >] (CR | : j Creates viewport below. (\*)  (0,0) of screen  (X0,Y0)  Viewport  (X1.Y1)  (31 ,23)1(39  The screen is not cleared and (X0,Y0) is used as the home position for the viewport, i.e., (X0,Y0) is assumed to be (0,0). |
| MAG | MAG [<sprite-modifier>] [CR | : ) Change the sprite size and format (\*)  < sprite-modifier >   1. = 8x8 dot matrix 2. = 8x 8 dot matrix (by 2) 3. = 16x 16 dot matrix 4. = 16x 16 dot matrix (by 2) |
| SCOL | SCOL <sprite-number>,<color> [CR | : J Color sprite-number using color code (\*) |
| SCOD | SCOD < sprite-number >, < numeric-code> (CR | :)  Assign numeric-code to sprite-number (\*) |

**BASIC-G Statement Types**

**Statement Type** **Function**

STCHR STCHR <pattern-code> TO <character-code>

[, < character-set -number > ]

**ICR | : )**

Assign a pattern-code to character-code (usually a hexadecimal number associated with an ASCI I-code) to determine a character's shape.

A character can also be colored using the STCHR statement in the Gil mode. (\*)

< Character-set-number >

1. = sprites
2. = for character patterns in other than the Gil

mode or character patterns in the top third of a Gil screen

1. = for character patterns in the middle third of a

Gil screen

1. - for character patterns in the bottom third of a

Gil screen

1. = for color codes for characters in the top third

of a Gil screen

1. = for color codes for characters in the middle

third of a Gil screen

1. = for color codes for characters in the bottome

third of a Gil screen

1. = for character patterns in the entire screen
2. = for color codes for characters in the entire

screen

Note: 2 to 6 correspond to the Gil mode The character ‘A’ is constructed below,\_ its pattern code is to the right (use hexadecimal notation for its pattern code).

STCHR STCHRE [< charcter-set-selection >] (CR | : )

Select the character set to be used as specified by character-set-selection. If 0, initialize for sprite operation. If 1, initialize background plane display.

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| LOC | LOC < sprite-number> TO <GR-coordinates>  (OR | : |  Move sprite-number to the specified GR-coordinates (\*) |
| CALL | CALL <assembler-program-address> |CR |: | Execute assembler program after putting the return address into a register. A RET (&C9) returns control back to the BASIC calling routine. At this time, registers other than IY can be used. (\*)  Address/register relationship PSW...& 7262 ACC...& 7263 B ...& 7265 C ...& 7264 D ...& 7267 F ...& 7266 H ...& 7269 1 ...& 7268 |
| STOP | STOP (CR | : )  Stop execution of program. (A program can be continued usinq a CONT statement.) (\*) |
| END | END (CR |: )  End of program. (A program can then no longer use the CONT statement.) (\*) |
| SAVE | SAVE <file-name> [,<start-address>,  < last-address > [,< start-address >] ] fCR-| : J  Write to external memory. In case of files, memory files are written to external memory. When the start- address and last-address specify a memory block in CPU RAM, data block from CPU RAM is written to cassette tape. (\*) |
| OLD | OLD [<file-name>] (CR | : )  Read a file from external memory. When the filename is omitted, the first file found is read into memory. (\*) |
| VERIFY | VERIFY [<file-name>](CR |:)  Corhpare a program in memory to another on external memory. When the file-name is omitted, the first file found on external memory is compared. (\*) |
| RUN | RUN [<line-number>] (CR |: j  Execute a file. You can begin executing from a line  number by including it as a parameter. (\*) |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| LIST | LIST [# <ID> | <file-name>] [<line-number-1 >] [,< line-number-2 >] |CR | :)  List a file or a portion of a file from line-number-1 to line-number-2 (optional). When a file-name is specified instead of an ID number, the listing will be output to external memory. (\*)  < # ID> (Don’t forget the ‘ # ’)  0 = CRT   1. = POUT 2. = POUT (Image) |
| LISTC | LISTC [# <ID> | <file-name>] [,<line-number-1 >] [,< line-number-2>] |CR | :)  Execute a listing of the desired data in upper case letters. Otherwise, it’s identical to the LIST statement. |
| ELIST | ELIST [# <ID> j <file-name>] [,<line-number-1 >] [,<line-number-2>] (OR | :)  Execute a listing of the desired data after clearing (erasing) the screen. Otherwise, it’s identical to the LIST statement. |
| ELISTC | ELISTC[# <ID> | <file-name>][,<line-number-1 >] [,< line-number-2>] (OR | :)  Execute a listing of the desired data in upper case letters after clearing the screen. Otherwise, it’s identical to the LIST statement. |
| RENU'M | REN U M [ < new-line-number > ] [, < old-line-number > ] [,<added-line-number>] (OR | :)  Renumber the line numbers of the program in CPU memory. Change the old-line-numbers to new-line- numbers, also put newly added-line-numbers in order. |
| CONT | CONT (OR | : )  Resume execution of the stopped program (\*) |
| AUTO | AUTO [<first-line-number>] [,<increment>] (CR | :)  Automatically give line-numbers (\*) |
| DEL | DEL [< line-number-1 >] [,< line-number-2>] {CR | : ]  Delete specified line numbers) (\*) |
| NEW | NEW {CR | :) Erase memory. (\*) |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| TAPE | TAPE|CR | : )  Access the assembler supplied on external memory.  \_J±) |
| ON ERROR GOSUB.. | ON ERROR GOSUB <line-number> [CR |:)  Call subroutine beginning at line-number when an error is detected, allows user to execute error processing routines. Interrupt priority = 0. |
| ON COINC GOSUB.. | ON COINC GOSUB <line-number> |CR |: ) Call subroutine beginning at line-number when sprites collide—interrupt priority 5. |
| COINC | COINC ION/OFF) (CR |: )  <ON/OFF>  ON = Allow calling of subroutine by an ON COINC GOSUB.. statement via sprite interrupts  OFF = Disable sprite collision interrupts |
| ON EVENT GOSUB.. | ON EVENT GOSUB <line-number> |CR | : j Call subroutine beginning at line-number when event timer interrupts—interrupt priority 2. (The event timer is initialized with the EVENT statement). |
| EVENT | EVENT (ON/OFF) (CR | : )  <ON/OFF>  ON = Allow calling of subroutine by an event timer interrupt set up by an ON EVENT GOSUB.. statement OFF = Disable event timer interrupts |
| ON ALARM GOSUB.. | ON ALARM GOSUB < line-number> (CR | : )  Call subroutine beginning at line-number when the alarm is set off—interrupt priority 1. (The alarm is set with the ALARM statement). |
| ALARM | ALARM (ON/OFF) (OR | : }  <ON/OFF>  ON = Allow calling of subroutine by an alarm interrupt set up by an ON ALARM GOSUB.. statement OFF = Disable alarm interrupts |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| ON KEY GOSUB.. | ON KEY GOSUB <line-number> (OR | : )  Call interrupt subroutine beginning at line-number when a keystroke is detected—interrupt priority 3. Keys other than RESET (HALT) apply. |
| KEY | KEY (ON/OFF) (OR | : J <ON/OFF>  ON = Allow calling of subroutine by a  keystroke interrupt set up by an ON KEY GOSUB.. statement OFF = Disable keystroke interrupts |
| ON JOY GOSUB.. | ON JOY GOSUB line-number (OR | : )  Call subroutine beginning at line-number when a joy pad direction change is detected—interrupt priority 4. |
| JOY | JOY (ON/OFF) (OR | : )  <ON/OFF>  ON = Allow calling of Subroutine by a joy pad interrupt set up by an ON JOY GOSUB.. statement OFF = Disable joypad interrupts |
| ON.. GOTO.. | ON < expression > GOTO < line-number> [,..]  (OR | : )  When the result of the expression is 1, execution jumps to the first line-number. When the result of the expression is 2, execution jumps to the optional second line-number, and so on ... If there are not enough jump destinations, an error will be issued. |
| ON.. GOSUB.. | ON <expression > GOSUB <line-number> [,..] (OR | : )  When the result of the expression is 1, call the subroutine beginning at the first line-number. When the result of the expression is 2, the subroutine beginning at the optional second line-number is called, and so on ... |
| ON.. RESTORE.. | ON <expression> RESTORE <line-number> [,..] (OR | : )  Using the same statement syntax as the ON.. GOTO., and ON.. GOSUB.. statements, RESTORE the appropriate DATA statements. |
| RESUME | RESUME [< line-number>] (OR | :)  Bypass an error and begin execution from line- number. When line-number is omitted, the next line is executed. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| REPEAT | REPEAT [CR | : }  First line of a repeat block (used with the UNTIL statement). It differs from the FOR statement in that it checks for a condition, rather than incrementing /decrementing a variable. |
| UNTIL | UNTIL < logical-expression > [CR | : )  Last line of a repeat block (used with the REPEAT statement). The logical-expression (e.g. X>10) defines a condition that must be met before execution exits the block delimited by REPEAT and UNTIL statements. If the condition is not satisfied, execution loops back to the corresponding REPEAT statement. |
| TRACE | TRACE [# 2] [ON/OFF) [CR | : )  <ON/OFF>  ON = list executing line numbers OFF = stop listing of executing line numbers #2 lists executing line numbers to POUT. |
| LOCATE | LOCATE <X>,<Y> [CR | : )  Move the cursor to the specified coordinates. Functions similar to CURSOR(X,Y). |
| ESCRN | ESCRN [<screen-number>] [,<switch>] [CR | : ) Change screen to expanded screen-number 0 to 8 (0 is the ordinary screen).   * screen number >  1. = standard screen buffer 2. = expanded screen buffer 1   8 = expanded screen buffer 8   * switch >  1. = display expanded screen (default) 2. = clear screen after copying screen   buffer to expanded screen   1. = copy screen buffer to expanded   screen |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| M PRINT | MPRINT (< rank-number>, < line-number >): <character-rank> (CR | : )  Divide the display data by rank and line, and display from the present cursor position. |
| FCOL | FCOL [<color-code>] (CR | :)  Set the character color in the text mode. Or in the multi-color and Gil modes, set the color of the graphics display. When omitted, color code 14 (gray) is assumed. |
| BCOL | BCOL [< color-code >] (OR | :)  Set up the background color. When omitted, color code 0 (no color) is assumed. |
| JOINT | JOINT [<sprite-number-1 > [TO < sprite-number-2 > ]  ,<link-position>]] (CR | : )  Join sprites together. If no parameters are included, all joints are released.  < link-position >   1. = release joints of one unit 2. = same position 3. = right edge 4. = lower edge 5. = left edge 6. = upper edge 7. = series release 8. = release all joints |
| EVENT | EVENT < interrupt-interval > [,< delay-time >]  (CR | :)  Set the interrupt interval accessed by the ON EVENT GOTO statement. 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. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| MOVE.. IN.. TO.. | MOVE <sprite-number> [IN <post-number>] TO |
|  | <GR-coordinates> [,< speed >] (OR | : J Move the sprite from its present position to its target GR-coordinates. Speed can vary from 0 to 255 (1 is the fastest). A sprite needs to be assigned to a postnumber (0~ 11) before motion is possible. A post can support one sprite at time. When the postnumber is omitted, the post-number equal to the sprite-number is used. |
| MOVE.. IN.. STEP.. | MOVE <sprite-number> [IN <post-number>] STEP <x-direction>,<y-direction> [,<speed>] (OR | : )  Move the sprite from its present position along the vector defined by the x-direction and y-direction coordinates. Speed can vary from 0 to 255 (1 is the fastest). When the post-number is omitted, a postnumber equal to the sprite-number is assumed. The coordinates should be from -128 to 127 pixels away from the cursor position. |
| MOVE.. IN.. ON.. | MOVE <sprite-number> [IN <post-number>] ON <GR-coordinates> jCR | : |  Move the sprite to the specified position (X,Y). This differs from the LOC statement in that it also moves a linked sprite joined by a JOINT statement. When the post-number is omitted, a post-number equal to the sprite-number is assumed. |
| MOVE | MOVE (ON/OFF) (OR | : )  <ON/OFF>  ON = Allow execution of MOVE statements OFF = Suspend execution of MOVE statements |
| ERASE | ERASE [< sprite-number >[,..] ] jCR | : )  Erase a sprite from the screen. If no parameters are included, all sprites are erased. |
| WAIT | WAIT < time-out -count > [, < increment -time > ]  [CR | :)  Set up a time-out for the keyboard (time to wait until a keystroke is detected). The actual time-out is (sec):  time-out-count\*increment-time/60  If the time-out is zero, the time-out timer will not function. If the increment-time is zero, it is assumed to be 256. If it is omitted, 60 is assumed. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| SLEEP | SLEEP <sleep-count> [,< increment-time>] [CRI:| Stop execution for the specified sleep time (however, interrupts are not ignored). Sleep time is (sec):  sleep-count x increment-time/60  If the increment-time is omitted, 60 is assumed. |
| COLOR | COLOR < character-code >, < color-code >  Set up the character color in the Gl and Gil modes. The higher-order four bits indicate the character color and the lower-four bits designate the background color, or  color = character-code x 16 + background-color or  color = & HL (H = character-color, L = background-color: H, L is hexadecimal)   * The Gl mode—each time a character is colored, it actually affects seven other characters with contiguous ASCII codes (If the 16x16 ASCII table is split into upper and lower halves, each half contains columns of eight characters. These eight are colored identically.) * The Gil mode—each character is colored individually. |
| PLAY | PLAY [<melody-1 >] [,<melody-2>] [,< melody - 3>]  (CR | :)  Play melodies 1, 2 and 3 concurrently to create harmony. At least one note should.be specified. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| GINIT | GINIT [<screen-dear-specifier>]  [, < screen-dear-request > ]  (CR | : )  Enter the graphics mode (applicable to the multicolor and Gil modes), the specified screen-dear- specifier is used for graphics (default is 255). |
|  | < screen-clear-request >   1. = 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. |
| GMODE | GMODE [<mode 1 >] [,<mode 2>] {CR |: }  Set up the graphics display mode—affects the PLOT, DRAW, BAR, CIRCLE, BOX and PAINT statements. It takes the color or pixel already on the screen and the newly specified color or pixel, it 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>  Color   1. = replace 2. = OR 3. = AND 4. = old display   Pixel   1. = replace 2. = AND 3. = XOR 4. = old display |
|  | The following relationship holds for the LIST #1 statement. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| GMODE | <mode>  Color   1. = replace 2. = OR 3. = AND 4. = old display Image 5. = same as GMODE 1 6. = same as GMODE 2 7. = XOR 8. = same as GMODE 3 |
| GMOVE | GMOVE <GR-coordinates> {CR | : j  Move the graphics cursor to the desired coordinates |
| PLOT | PLOT <GR-coordinates>[;..] {CR | :)  Display the dot associated with the coordinates. Use the color set up by a FCOL statement. After execution, the graphics cursor will reside at these coordinates. |
| DRAW | DRAW < GR-coordinates > [, < GR-coordinates > ] [;••] (CR | :)  When only one set of GR-coordinates is specified, draw a line from the cursor position to the GR-coordinates. After drawing the line, the cursor will reside at these coordinates. When two sets of GR-coordinates are specified, draw a line from the first set of coordinates to the second set of coordinates. The cursor position does not change. |
| CIRCLE | CIRCLE <x-axis-radius> [,<y-axis-radius>] [,<polygon>] [,<beginning-ahgle>]  [,< ending-angle >]  [, < oblique-angle > ] [, < fan -parameter > ]  {CR | :)  By only specifying the x-axis-radius, a circle is drawn, also specifying the y-axis-radius produces an ellipse. The beginning- and ending-angles can be specified to produce circles and ellipses perfect for the outer curve of a pie chart. Polygons are also easily created with the polygon parameter. Always express angles in degrees. |

**BASIC-G Statement Types**

**Statement Type Function**

BAR BAR <GR-coordinates-1 >,<GR-coordinates-2>

(CR | :)

Draw a solid rectangle whose base corner corresponds to the first set of GR-coordinates. Its opposite corner is the second set of GR-coordinates. The direction to draw a rectangle begins from its base corner to the opposite corner. The graphics cursor does not change positions after the rectangle is drawn.

(X0,Y0) (X1.Y1)

JJ l\_L

(X1-Y1) (XO.YO) (XO.YO) (X1.Y1)

(X1.Y1) (XO.YO)

The above pictures show the direction a rectangle is filled in relative to its base and opposite corners.

BOX BOX <GR-coordinates-1 >,<GR-coordinates-2>

(CR | :)

Draw a rectangle whose base corner corresponds to the first set of GR-ooordinates. Its opposite corner is the second set of GR-coordinates. The graphics cursor does not change positions after the rectangle is drawn.

(XO.YO) (X1.Y1)

1

2 2

(XTYI) (XO.YO) (X°4Y0)

(X1.Y1)

2

1 3

4

(XO.YO)

1. 1

2

(X1.Y1)

The above pictures show the direction (by the side number) the box is drawn given its base point (XO.YO) and opposite point (X1,Y1).

**BASIC-G Statement Types**

< format-type >

|  |  |
| --- | --- |
| Statement Type | Function |
| PAINT | PAINT <GR-coordinates> [,<boundary-color>  U ] (CR | : |  Paint an area delimited by the GR-coord'nates using one of up to 16 colors indicated by boundary-color. Even if the boundary-color is omitted, the appropriate area will not be colored transparent (invisible). |
| GCOPY | GCOPY [<format-type>] |CR-1 : J Print the screen image on the printer. |

1. = 40 character image format
2. - 80 character image format
3. = line format (by color)

|  |  |
| --- | --- |
|  | 7 = : : : |
|  | Default is 0. |
| POKEW | POKEW <memory-address> [,<data> [,..]]  ICR | : ]  Write the data to the specified memory address in CPU memory. 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. |
| CONSOLE | CONSOLE [<A>] [,<B>] [,<C>] [,<D>] [,<E>]  ICR | : ) |

|  |  |  |
| --- | --- | --- |
| Function | 0 | 1 |
| A Keyboard sounds | OFF | ON |
| B Generate keyboard keywords | OFF | ON |
| C Screen switch | OFF | ON |
| D Display page | page 0 | page 1 |
| E Process page | page 0 | page 1 |

|  |  |
| --- | --- |
| OUTW | OUTW <l/0-port-number> [,<data> [,..] ] (CR | : j Output data to the specified port; the lower-order byte is output first. The data must be numeric. |
| SKIP | SKIP [<file-name>] (CR | : )  Skip reading of the specified file and find the end of the file. |

**BASIC-G Statement Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| VSAVE | VSAVE <file-name> [<start-address>]  [,< end-address >] {CR | :)  When a file name is specified, save the video RAM data on cassette tape. If two addresses are specified, write VRAM data between these two addresses to cassette tape. |
| PUSH | PUSH <expression > [„..] {CR | :)  Push the data specified by the expressions onto the stack. Do not use character data. |
| POP | POP variable [„..] {CR | : )  Pop the FILO stack and set the data into variable. (Data was pushed using the PUSH statement.) |
| LEN | LEN <character-string-length >  Define the length of the character string variable (from 1 - 255) accessed after execution of this LEN statement (only applicable to this particular character string variable). |
| SG | SG <channel-number>,[{<frequency> | <noise>|] [,< volume >] {CR | :)  Turn the three tone generators and noise generator on and off as well as making them produce sound effects.   * channel-number >  1. = tone generator channel 0 2. = tone generator channel 1 3. = tone generator channel 2 4. = noise generator  * frequency >   Frequency value varies from 1 to 1023; 1 is the highest frequence 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). |

**BASIC-G Statement Types**

**Statement Type Function**

|  |  |
| --- | --- |
| 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.

PMODE PMODE [longest-output-line]

[.POUT-characteristic] [.image-characteristic]

[,image-list-pitch]

[,image-copy-pitch]

[,image-mode-set-sequence]

(CR | :)

PMODE allows interfacing between the M5 system and your printer (POUT).

* longest-output-line >

Specifies the longest line to be sent to and printed by POUT.

<POUT characteristic >

Select whether a CR (carriage return)/LF (line feed) is applicable for your printer.

|  |  |  |
| --- | --- | --- |
| POUT | CR + LF | Automatic-  new-line |
| 0 | No | No |
| 1 | Yes | No |
| 2 | No | Yes |
| 3 | Yes | Yes |

CR + LF means a LF (ASCII 10) is output after a CR (ASCII 13). An automatic-new-line function automatically inserts a CR + LF every line (less than the longest-output-line).

**BASIC-G Statement Types**

**Statement Type Function**

PMODE < image-characteristic >

Select the format of printed data

|  |  |  |
| --- | --- | --- |
| IMAGE CHAR | COUNTER | DATA |
| 0 | Lw-Hi | T-MSB |
| 1 | Hi-Lw | T-MSB |
| 2 | Lw-Hi | B-MSB |
| 3 | Hi-Lw | B-MSB |

Lw-Hi means to send the lower order byte first; Hi-Lw means to send the upper order byte first.

T-MSG means the most significant bit is held high; B-MSG means the least significant bit is held high.

* image-list-pitch >

Determines the pitch of a line when using LIST #3.

* image-copy-pitch >

Pitch of the printer output data used by GCOPY.

* image-mode-set-sequence >

Changes a printer into the image mode. This allows the image data number to be sent, the image data follows.

**BASIC-G Function Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| ABS(X) | Return the absolute value of X (\*) |
| SGN(X) | Return the sign of X (\*) |

X > 0 ... SGN(X) = 1 X = 0 ... SGN(X) = 0 X < 0 ... SGN(X) = -1

|  |  |
| --- | --- |
| TIME | Return the amount of time since the system has been turned on (in seconds). (\*) |
| CURSOR(X,Y) | Move the cursor to the coordinates specified by (X,Y). A semicolon, is usually used after this keyword. (\*) |
| TAB(X) | Tab over X characters. An automatic RETURN is |

done if TAB(X) forces the current line to be exceeded. A semicolon, is usually used after this keyword. (\*)

RND(X) Return a random number between 0 and X. (\*)

|  |  |
| --- | --- |
| IN KEYS | Read one character from the keyboard input buffer. (\*) |
| PEEK(X) | Return 8-bits stored in CPU memory address X. (\*) |
| VPEEK(X) | Return the data stored in video memory address X. (\*) |
| INP(X) | Read one byte from port X. (\*) |
| LEN(X$) | Return the length of character string X$. (\*) |
| ASCII(X$) | Return the ASCII equivalent of the first character in character string variable X$. (\*) |
| LEFT$(X$,Y) | Return Y characters from the left side of character string X$. (\*) |
| RIGHT$(X$,Y) | Return Y characters from the right side of character string X$. (\*) |
| MID$(X$,Y [,Z]) | Return Z characters starting from character number Y (from the left) in character string X$. (\*) |
| VAL(X$) | Convert the value of character string X$ to its numeric equivalent. (\*) |
| NUM$(X) | Convert the numeric value of X to its character equivalent. (\*) |

**BASIC-G Function Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| CHR$(X) | Return the character having ASCII code X. (\*) |
| HEX$(X) | Return the hexadecimal equivalent of X in four hexadecimal places (no zero suppression). (\*) |
| ERR | Return the number of the most recent error. (\*) |
| ERRL | Return the erroneous line number. (\*) |
| ERRL$ | Return the label of the erroneous line. (\*) |
| FRE(X) | Return several work area or user area parameters. (\*) |
|  | X = 0 ... maximum size of work area X = 1 ... remaining free user area X = 2 ... remaining free work area X = 3 ... remaining free user and work area X = 4 ... last address used by BASIC |
| Additional BASIC-G Function Types | |
| Statement Type | Function |
| POS(X) | Supply the cursor position. |
|  | X = 0 ... x coordinate X = 1 ... y coordinate X = 2 ... x coordinate of graphics cursor X = 3 ... y coordinate of graphics cursor |
| INSTR([X],Y$,Z$) | Check for the existence of character string Z$ after the Xth character in character string Y$. If found, return its position; otherwise return zero. |
| RPT$(X,Y$) | Return X repetitions of character string Y$. |
| VARPTR(X/X$) | Return a pointer to variable X/X$. |
| TIMES | Return the elapsed time since turning on the system. The format is “hh:mm:ss” where: |
|  | hh = hour mm = minute ss = second |
| RCRT(X,Y) | Return the ASCII code of the displayed character on the screen at the coordinates (X,Y). |
| RDCHR$(X,Y) | Return the pattern code of character X in character set Y (hexadecimal numbers). |

**Additional BASIC-G Function Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| MRCRT$(X,Y) | Return the displayed character from the (X,Y) coordinates using the present cursor position on the screen. |
| COLOR(X) | Specify color of the present character. Colors are related to the screen mode and value X.  Text mode  X = 0 ... character color X = 1 ... background color of character  Gl mode  Character color of ASCII code X  Multi-color mode, Gil mode  Ignore X and return color code |
| SPRITE(X,Y) | Return sprite information (attributes) of sprite X as specified by Y.  <Y>   * = 0 ... y coordinate * = 1 ... x coordinate * = 2 ... ASCII code * = 3 ... color |
| LINK(Xt[Y]) | Return linkage information of sprite X as specified by Y.  <Y>   * = 0 ... number of linked sprites * = 1 ... linked position |
| DIST(X0,Y0,X1,Y1) | DIST (GR-coordinates,GR-coordinates)  Return the square of the distance between the set of two GR-coordinates. If the result is over 32767, return -1. |

**Additional BASIC-G Function Types**

**Statement Type** **Function**

POINT(X,Y) POINT (GR-coordinates)

Used for graphics in the multi-color and Gil modes. Multi-color mode—return the color code of a point when given a set of coordinates Gil mode—return the color code which also indicates its existence

< return value >

M-mode Oto 15 Gil-mode

1. to 15 ... color code indicate^ the dot does not exist

16 to 32 ... color code indicates existence of the dot

to find the color of the dot, use point (x-coordinate, y-coordinate) AND & OF

DRCT(X0,Y0,X1,Y1) DRCT (GR-coordinates,GR-coordinates)

Return the relationship of (X0,Y0) assuming (X1,Y1) is (0,0) of an x-y grid.

< return value>

-4 ... **X0** = **X1, Y0** < **Y1** -3 ... **X0 < X1, Y0 = Y1** -2 ... **X0 = X1, Y0 > Y1 -1 ... X0 > X1, Y0 = Y1**

1. ... same point
2. ... (X0,Y0) is in the first quadrant
3. ... (X0,Y0) is in the second quadrant
4. ... (X0,Y0) is in the third quadrant
5. ... (X0,Y0) is in the fourth quadrant

|  |  |
| --- | --- |
| quadrant 3 | quadrant 4 |
|  | (X1.Y1) |
| quadrant 2 | quadrant 1 |
| + Y | |

STATUS(X) Check whether post number X is presently used.

< return value >

1. = not used
2. = used

POST[([M],[N])] Return a post number presently not being used in the range M to N. If M is omitted, return a post number between 0 and N. If N is omitted, return a post number up to 11.

**Additional BASIC-G Function Types**

**Statement Type Function**

INKEY(X) Return the address of the key currently being

pressed.

1. = key-address
2. = supplemental key’s information < supplemental key’s information >

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| result | CTRL | FUNC | SHIFT  (left) | SHIFT  (right) |
| 0 | OFF | OFF | OFF | OFF |
| 1 | ON | OFF | OFF | OFF |
| 2 | OFF | ON | OFF | OFF |
| 3 | ON | ON | OFF | OFF |
| 4 | OFF | OFF | ON | OFF |
| 5 | ON | OFF | ON | OFF |
| 6 | OFF | ON | ON | OFF |
| 7 | ON | ON | ON | OFF |
| 8 | OFF | OFF | OFF | ON |
| 9 | ON | OFF | OFF | ON |
| 10 | OFF | ON | OFF | ON |
| 11 | ON | ON | OFF | ON |
| 12 | OFF | OFF | ON | ON |
| 13 | ON | OFF | ON | ON |
| 14 | OFF | ON | ON | ON |
| 15 | ON | ON | ON | ON |

JOY(X) Return joypad position information given the joypad

number.

<X>

1. = left joypad (#1)
2. = right joypad (#2)

< return value>

1. = neutral

(middle)

1. = up
2. = upper right
3. = right
4. = lower right
5. = bottom
6. = lower left
7. = left
8. = upper left

8

7

6

1

5

2

3

4

<X>

**Additional BASIC-G Function Types**

|  |  |
| --- | --- |
| Statement Type | Function |
| ASW(X) | Return 4-bit information on an attack button given |

the corresponding joypad number.

1. = left attack button (joypad 1)
2. = right attack button (joypad 2)

XCHG(X) Swap the upper- and lower-order bytes of X.

|  |  |
| --- | --- |
| PEEKW (X) | Return 16-bits from CPU memory address X and address X +1. |
| INPWPQ | Read 16-bits from port number X. The first byte read is the lower order byte. |
| SIN(R,X) | Return the positive value of R\*sin(x). |
| COS(R,X) | Return the positive value of R\*cos(x). |
| COINC(X [,M] [,N]) | Return the number of the sprite colliding into sprite number X. M is the starting retrieved sprite number while N is the finishing retrieved sprite number. |
| PLAY(X) | Play music enclosed in double quotes. A melody as well as harmony can be played. |

**APPENDIX C—SCREEN MODE SUMMARY**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CHARAC1 | | rER | COLOR | | | EXPANDED  SCREEN |
| \ Maximum # \ different \ characters  MODE\ | | Usage of STCHR | Background  plane | | Backlight  plane  color  statement |
| Char  color | Back  ground  color |
| Gl | 256 | one screen character set “1” | COLORS 8 at the same time | COLORS 8 at the same time | BCOL | ESCRN  1-8 |
| &!  char\_  color | 31  back  ground  color |
| Gil | 786(256 for the top,middle and bottom thirds) | differs according to part of the screen  top "1” middle “2” bottom “3” | COLORS  each  char  sepa  rately  STCHR  colors  images  sepa  rately | COLORS  each  char  separately  STCHR  colors  images  sepa  rately | BCOL | — |
| &81  char\_l Lback'  color ground color | |
| Text | 256 | one screen character set “1” | FOOL |  | BCOL | ESCRN 1 ~8 |
| Multi  color | — | — | — | — | BCOL | ESCRN  1-8 |

**APPENDIX D—CONSTRUCTING YOUR OWN CHARACTER OR SPRITE**

Hex code to pixel relationship

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

0

1

2

3

4

5

6

7

0

1

2

3

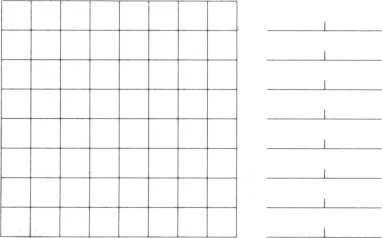
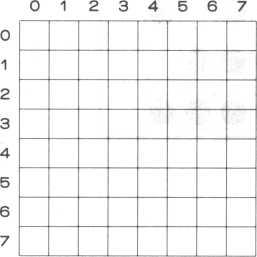
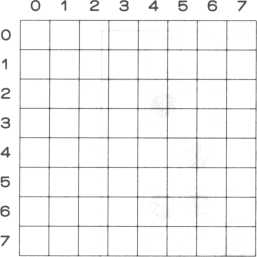
4

5

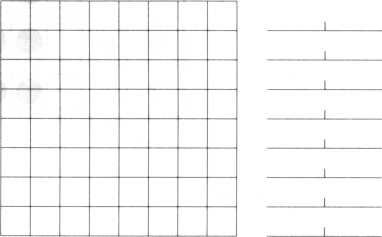
6

7

Coding sheet for 8x8 pixel pattern code

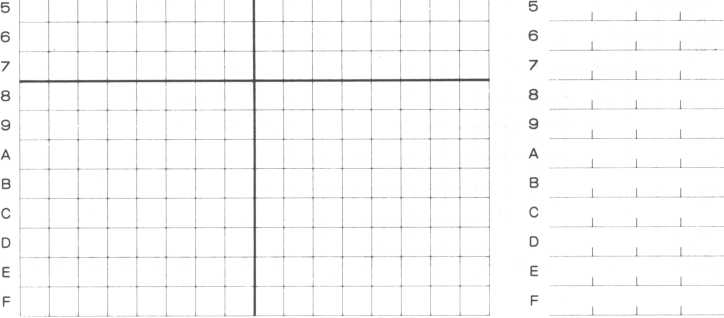


0 1 2 3 4 5 6 7



Coding sheet for 16 x 16 pixel pattern code

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |  |  |  |  |
| 0 |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  | 0 |  | \_J | L |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 | 1 | \_J | L |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 2 |  | \_J | L |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 3 | I | \_l | L |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 4 | r | \_1 | L |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | i | \_i | L |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 6 | i | \_J | L |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 7 | i | \_J | L |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 8 | i | \_J | L |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 9 | I | \_l | 1 |
| A |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | A | L | \_J | L |
| B |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | B | 1 | \_1 | L |
| C |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | C | 1 | \_1 | L |
| D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D | 1 | L | L |
| E |  |  |  |  |  |  |  |  |  |  |  |  |  | L , - |  |  | E | 1 | \_L | L |
| F |  |  |  |  | —  d L | | | I  l |  | J | |  |  | I |  |  | F | 1 | \_J | L |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |  |  |  |  |

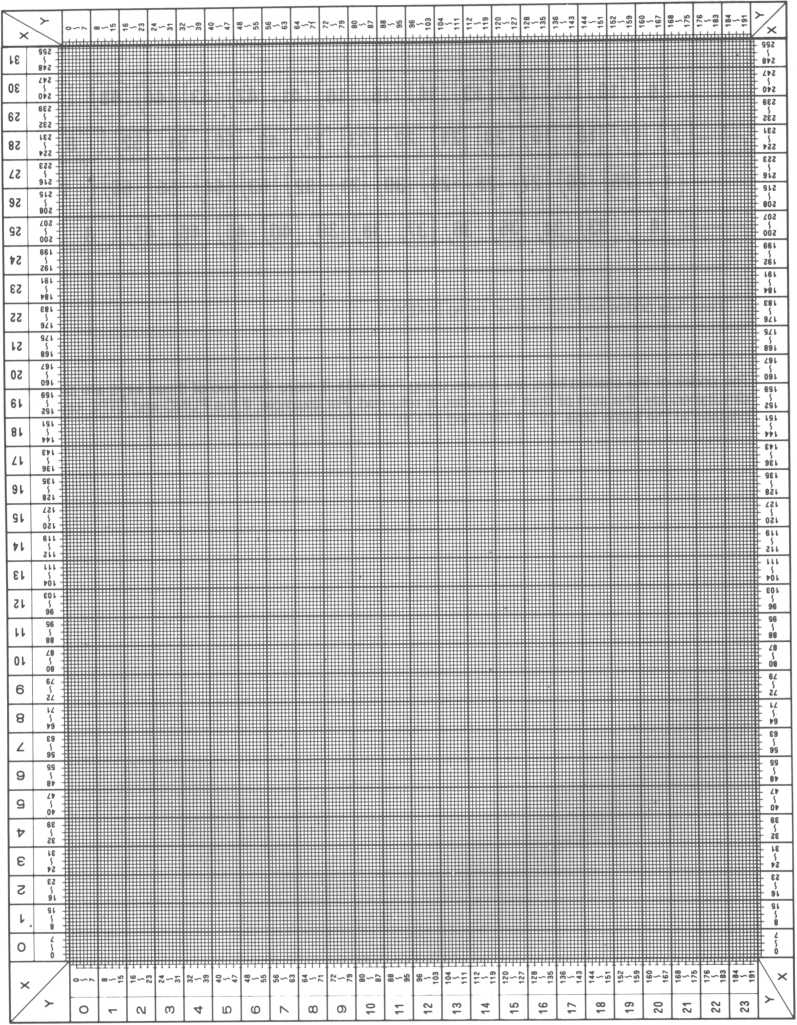


**APPENDIX E—CRT LAYOUT COORDINATES**

CRT Screen Layout Sheet (Characters)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | O |  | CVJ | CD |  | ID | CD | N | 00 | 0) | o | - | CM | CO | •M- | in | CD | Cm | CO | 03 | o  CM | CM | CM  CM | CO  CM | >- / / < |
| m |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | eo |
| o  CO |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | o  CO |
| 03  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 03  CM |
| CO  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CO  CM |
| r-\*  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | I'M  CM |
| CD  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CO  CM |
| in  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | m  CM |
| CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | •\*r  CM |
| ,co  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CO  CM |
| CM  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CM  CM |
| CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CM |
| O  CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | o  CM |
| 03 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 03 |
| CO |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CO |
| r\*\* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | f-v |
| CD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CD |
| m |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | m |
| ■M- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | •M- |
| CO |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CO |
| CM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CM |
| - |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | - |
| o |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | o |
| 0) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | \ |  |  |  | 0) |
| CD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 00 |
| N |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | N |
| CD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |
| ID |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |
| \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| CD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CD |
| CVJ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | CVI |
| r- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | - |
| o |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | O |
| x/  /3- | o | - | CVJ | CD |  | ID | CD | N | 00 | 0) | o | £ | CM | CO | 2 | in | CD | 1^ | CO | 03 | o  CM | CM | CM  CM | CO  CM | \x  >-\ |

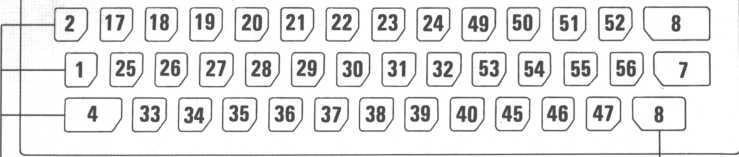
CRT Screen Layout Sheet (Pixels)



**APPENDIX F—KEYBOARD KEY CODES**

Key Codes

X



These key codes are ascertained using INKEY(1)

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).

APPENDIX G—OPERATORS

from page 1 and 2 of Japanese appendix modifications

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbol | | Usage | Example | | Priority |
| Calcu-  lation | ✓V | Exponentiaton | A- B | A to the power of B | 1 |
| ★ | Multiplication | A\*B | Ax B | 2 |
| / | Division | A/B | A divided by B | 2 |
| MOD | Mod function | A mod B | remainder of A divided by B | 3 |
| + | Addition | A + B | A + B | 4 |
| — | Subtraction | A-B | A-B | 4 |
| Compar-  ators | zz | Equal | A = B | A = B | 5 |
| < | Less than | A<B | A less than B | 5 |
| > | Greater than | A>B | A greater than B | 5 |
| <>, >< | Not equal | AoB A> <B | A not equal to B | 5 |
| A  II  II  A | Less than or equal | >> II A A II CD CD | A less than or equal to B | 5 |
| * II   II  V | Greater than equal | >> II V V II  CD CD | A greater than or equal to B | 5 |
| Boolean | AND | AND | A AND B | A B AND 11 = 1 10 = 0 0 1=0 00 = 0 | 6 |
| OR | OR | A OR B | A B OR 11=1 1 0= 1 0 1=1 | 7' |
| XOR | Exclusive OR | A XOR B | A B XOR 11=0 1 0= 1 0 1=1 00= 0 | 8 |
| NOT | NOT | NOT A | A NOT A  0 1 1 0 | 0 |

If part of an equation is enclosed in parentheses (nesting of parentheses is also possible), its precedence requires that it be processed before the priorities listed here.

The precedence of negative numbers and variables exceeds everything listed above.

**APPENDIX H —POUT I/O TABLE**

|  |  |
| --- | --- |
| POUT Number | Summary |
| Z80CTC |  |
| 01 | Channel # 1 ... peripheral timer |
| 02 | Channel #2 ... I/O clock |
| 03 | Channel 3 ... VDP |
| VDP TMS9918A |  |
| 11 | Status port |
| 11 | Screen base address and control port |
| 11 | VRAM address port |
| 10 | Data read port |
| 10 | Data write port |
| TONE GENERATOR |  |
| 20 | Tone generator control |
| KEYBOARD |  |
| 30 | Row 0 |
| 31 | Row 1 |
| 32 | Row 2 |
| 33 | Row 3 |
| 34 | Row 4 |
| 35 | Row 5 |
| 36 | Row 6 |
| JOYPAD/ATTACK BUTTON |  |
| 37 | Input joypad direction |
| RESET/HALT KEY |  |
| 50 | Reset/Halt key data port (bit 7) |
| CASSETTE |  |
| RECORDER |  |
| 50 | Output port |
| 50 | Input port |
| 50 | Output port |
| PERIPHERAL |  |
| I/O |  |
| 40 | Data output |
| 50 | Strobe output |
| 50 | Printer busy |

**APPENDIX I—MEMORY MAP**

Memory map

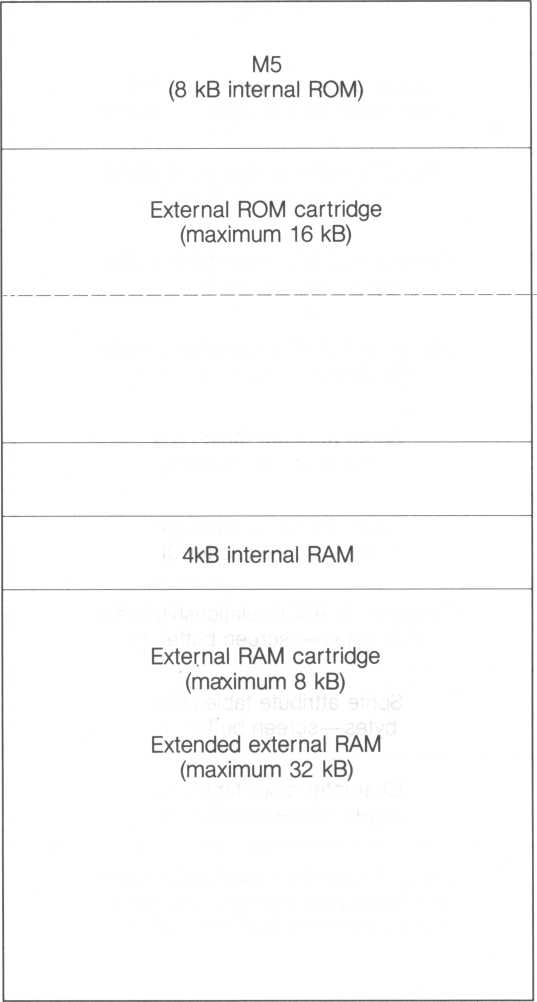
Online main memory

0000

ROM area

Reserved

RAM area



2000

4000

6000

7000

8000

FFFF

VRAM (video RAM) memory map Layout I

0000 2000 2800 3000 3800 3B00 3B80 3C00 3F00 3F80 4000

Free area

Sprite pattern code table (2048  
bytes—both screen buffers 0 and 1)

Character pattern code table (2048  
bytes—screen buffer 0)

Character pattern code table (2048  
bytes—screen buffer 1)

Character to ASCII relationship table  
(768 bytes— screen buffer 0)

Sprite attribute table (128 bytes—screen buffer 0)

Character color table (32 bytes—screen buffer 0)

Character to ASCII relationship table  
(768 bytes— screen buffer 1)

Sprite attribute table (128 bytes—screen buffer 1)

Character color table (32 bytes—screen buffer 1)

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

**Layout II**

0000

1800

2000

3800

3B00

3B80

3C00

3F00

3F80

4000

Gil mode color table (6 kB)

Sprite pattern code table (2 kB)

Gil mode pattern code table (6 kB)

Pattern code table (screen buffer 0)

Sprite attribute table

Character color table

\*

Pattern code table (screen buffer 1)

Sprite attribute table

Character color table

\*

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

**APPENDIX J—ERROR CODES**

|  |  |  |
| --- | --- | --- |
| ERROR  CODE | ERROR SUMMARY | REASON |
| ERR 1 ERRNF | FOR .. NEXT error | • FOR - NEXT does not match |
| ERR 2 ERRSY | Syntax error | • Bad BASIC-G grammer |
| ERR 3 ERRRG | Subroutine error | * Used CLEAR in the subroutine * Jumped to subroutine using a GOTO * GOSUB-RETURN does not match |
| ERR 4 ERROD | READ error in DATA statement | * Short DATA statement * Missing DATA statement |
| ERR 5 ERRIF | Variable type mismatch | • Wrong type of value given for statement variable |
| ERR 6 ERROV | Overflow | • Answer is too large to be kept in the M5 (check for large multiplications) |
| ERR 7 ERROM | Memory exhausted | * Program is too large * Too many variables (reduce the number) * Too many subroutines (reduce the number) |
| ERR 8 ERRUL | Missing line number | • Missing destination for GOTO or GOSUB |
| ERR 9 ERRBS | Array variable error | * Array variable error * Variable types differ |
| ERR 10 ERRDD | Array variable error | • Same variable set twice |
| ERR 11 ERRDZ | Division by 0 | • Divided by zero |
| ERR 12 ERRID | Illegal statement | • Bad request |
| ERR 13 ERRTM | Bad data item | • Characters were provided when numerics were expected, or vice versa |
| ERR 14 ERROS | Stack overflow | * Stack space exhausted * Too many data items PUSFIed on to the stack * No stack area left for the PAINT statement |

|  |  |  |
| --- | --- | --- |
| ERROR  CODE | ERROR SUMMARY | REASON |
| ERR 15 ERRST | Character string length error | * Character string too long * Over 19 characters * Attempt to substitute a character string into a variable not capable of holding a string of that length |
| ERR 16 ERRUD | Array variable error | • Used an array variable which has not yet been allocated |
| ERR 17 ERRDL | Redundant lable | • Used the same label more than once |
| ERR 18 ERRTR | Tape read error | * Tape read error * Reset during tape read operation |
| ERR 19 ERRDM | Wrong screen display mode | • Wrong screen mode, either Gl, Gil, text or multi-color. |
| ERR 20 ERRSP | Sprite error | • Tried to use or move a sprite that has been erased (deleted) |
| ERR 21 ERRNS | Stack error | • PUSH ~ POP does not match match correctly |
| ERR 22 ERRUR | REPEAT..UNTIL error | • REPEAT..UNTIL does not match |
| ERR 23 ERRTO | Timeout error | • INPUT statement timed out |
| ERR 24 ERRRE | RESUM error | • Executed RESUM when no error occurred |
| ERR 25 ERRDF | INPUT error | • Pressed RETURN key without keying in any input data |

**Copyright © 1983 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



SORD COMPUTER CORPORATION

KYOBASHI K-1 BLDG., 7-12, YAESU 2-CHOME, CHUO-KU, TOKYO 104, JAPAN PHONE: (03)281-8118 TELEX: 2224225 (SORDIN J)

1. Many commands are now more efficient (saves execution time). [↑](#footnote-ref-1)
2. Amend pictures drawn with BASIC-G

   The graphics capability of BASIC-G is higher quality, but it sometimes takes a lot of data.

   Sometimes a quick fix is more appropriate than changing a BASIC-G program—easily done with P-EDIT. [↑](#footnote-ref-2)