Graphics Book for the COMMODORE 64

A Data Becker Book from First Publishing Limited



1st
FIRST PUBLISHING LTD

GRAPHICS BOOK

FOR THE

COMMODORE 64

BY: Axel Plenge

A DATA BECKER BOOK

PUBLISHED BY: FIRST PUBLISHING LTD
UNIT 20B
Horseshoe Road
Horseshoe Industrial Est.
Pangbourne
Berkshire, England

Copyright 1983, 1984

Data Becker GMBH
Merowingerstraße 30
4000 Düsseldorf
West Germany

Copyright 1983, 1984

Abacus Software
P. O. Box 7211
Grand Rapids
MI 49510

Copyright 1984

First Publishing Ltd
Unit 20B Horseshoe Rd
Horseshoe Trading Est
Pangbourne, Berkshire
England

This book is copyrighted. 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 First Publishing Ltd.

ISBN 0-948015-047

Preface

The Commodore 64's graphics are one of its primary strengths, but Commodore has hidden this strength very carefully. The only awareness the BASIC beginner has of the graphics capabilities is from commercial ready-to-run programs and games, which naturally have appropriate routines (usually in machine language) to make use of these capabilities. This book is intended to give every Commodore 64 user the opportunity to make use of the capabilities which this computer offers.

The author of this book, Axel Plenge, knows the Commodore 64, particularly its graphics side, like few others as is made clear in his program SUPERGRAPHICS. He found writing this book to be so much fun and discovered so many interesting things that the Graphics Book is almost 50 pages longer than planned. This has many advantages for you as reader, and we hope you have fun trying out the suggestions and programs in this book.

Dr. Achim Becker

Table of Contents

Chapter 1
Introduction.....1

Chapter 2
Bits and Bytes.....4
2.1 Decimal system.....5
2.2 Binary system.....7
2.3 Hexadecimal system.....10
2.4 Logical operations.....12

Chapter 3
Hardware Foundations.....15
3.1 The VIC registers.....16
3.2 VIC operating modes.....24
3.3 Memory management.....30
3.3.1 Memory access of the 6510.....33
3.3.1.1 Reading a byte.....33
3.3.1.2 Writing a byte.....38
3.3.2 Memory access of the VIC.....38
3.3.2.1 VIC memory functions.....39
3.3.2.2 VIC memory control.....41
3.3.2.3 Moving screen memory.....43
3.4 Bit-mapped graphics.....49
3.4.1 Colors.....49
3.4.2 High-resolution graphics (HGR).....50
3.4.3 Multi-color graphics (MC).....56
3.5 Sprites.....59
3.5.1 Organization and color of normal sprites...60
3.5.2 Definition and color of multi-color sprites.....61
3.5.3 Sprite definition - Color.....63

3.5.4 Additional sprite properties.....67
3.5.4.1 Positioning.....67
3.5.4.2 Enlargement.....70
3.5.4.3 Priority.....71
3.5.4.4 Collisions.....72
3.6 Text and character set management.....74
3.6.1 Text page organization.....74
3.6.1.1 Normal text.....75
3.6.1.2 Multi-color mode.....75
3.6.1.3 Extended-color mode.....75
3.6.2 Character set organization.....78
3.7 IRQ capabilities.....84
3.7.1 Screen raster lines.....87
3.7.2 Light pen.....91
3.7.3 Sprite collisions.....93

Chapter 4
Fundamental Graphics Programming.....95

4.1 Text and graphics on the low-res screen.....96
4.2 Programming bit-mapped graphics.....112
4.2.1 Initializing graphics.....114
4.2.1.1 Turning graphics on.....115
4.2.1.2 Clearing the graphics screen.....116
4.2.1.3 Clearing the color RAM.....117
4.2.1.4 Turning graphics off.....119
4.2.2 Simple figures in bit-mapped graphics.....120
4.2.2.1 Points.....120
4.2.2.2 Lines.....126
4.2.2.3 Circles and Ellipses.....132
4.3 Sprite programming.....137
4.3.1 Creating sprites 137
4.3.2 Programming using the sprite features.....167

4.4 Character set programming.....181
4.4.1 Modifying characters.....182
4.4.2 Modifying a character set.....187
4.5 Input/Output of graphics and character sets.....213
4.5.1 Saving/loading.....213
4.5.2 Hardcopy.....216
4.6 IRQ handling.....220
4.6.1 Raster line IRQ.....221
4.6.2 Light pen.....228
4.7 A small graphics package.....237

Chapter 5
Applications.....266

5.1 Graphics applications.....267
5.1.1 Graphing functions.....267
5.1.2 Three-dimensional graphics.....281
5.1.2.1 Parallel projection.....282
5.1.2.2 Central projection.....291
5.1.2.3 3-D functions.....293
5.1.2.4 Moving pictures in 3D.....300
5.1.3 Graphing statistics.....303
5.2 Moving messages.....315
5.3 The secrets of the games.....321
5.3.1 Animation.....321
5.3.2 Scrolling.....326

Chapter 6 Reference Information

6.1 Program optimization.....335
6.2 Graphics memory construction.....338

6.2.1 Graphics memory.....339
6.2.2 Video ram.....340
6.3 Color table.....341
6.4 Screen codes.....342
6.5 Dec-Hex-Binary conversion table.....344
6.6 Sprite development sheet.....345
6.7 Character development grid.....346
6.8 VIC register overview.....347
6.9 Bibliography.....351

Chapter 1
Introduction

"64K RAM; 8 independent and freely movable sprites; high-resolution graphics with 320x200 pixels; multi-color graphics (160x200 pixels); 16 colors; 40 characters; 25 lines, user-defined character sets; sensational interrupt capabilities, ..., a computer which everyone should be acquainted with, no, own!..."

Such has been the praise for our good old Commodore 64 And rightly so! No one who knows anything about computers can ignore such a list of features--your '64 is really a very capable machine.

But soon after purchasing it and studying the user's guide, you get a peculiar feeling in the pit of your stomach. Not one word about the high-resolution graphics, silence about creating new character sets and no mention of the interrupt capabilities. Except for the guide's relatively detailed chapter on programming sprites, the true capabilities of the '64 remain a secret.

We do not wish to make too harsh a judgement in this introduction. Instead, our purpose is to describe the features of this computer in detail and to demonstrate their use.

So at this point we begin The Graphics Book of the Commodore 64. It is intended to be a comprehensive source of information about the inner workings of graphics in the Commodore 64.

The book is divided into three major parts. The capabilities of the computer are described from three different viewpoints and illustrated with many examples. These three sections describe:

- hardware foundations for graphics
- fundamentals of graphics programming
- applications using graphics

In Chapter 3 you'll learn everything (and we mean everything) there is to know about the '64 hardware and the VIC (Video Interface Chip) in particular. Here, examples will show you the purpose of each of the VIC registers, what sprites are and how they are organized, what you must do to enable, create, and use graphics. After reading this section, you'll know how to do such diverse things as creating moving graphics, designing new character sets, or switching screen memory. In general, you'll know how the entire memory of the '64 is put together and what graphics possibilities this construction offers.

In Chapter 4 you'll learn about programming graphics on '64. You'll learn how to draw simple figures using points, lines, circles and ellipses on screen. We'll present two easy-to-use editor programs that let you create or modify sprites and characters with little effort. Additional sections introduce you to loading, saving, and printing graphic screen images and explain the interrupt techniques. Finally, we'll present a graphics package written in machine language which will allow you to quickly and easily make use of the high-resolution graphic features of your '64.

In Chapter 5, you'll be introduced to some practical applications that use the '64's graphics capabilities. After reading this section, you'll be able to answer the question: "What can a computer do for me?"

We hope to turn every '64 user into a "graphics freak," an expert who is consulted when no one else knows the answer. If some detail should slip your mind, a quick flip through the reference information in Chapter 6 will refresh your memory. A comprehensive bibliography of additional literature rounds the book off.

One more thing: We know that not everyone has the time or the desire to type in all of the programs (especially the long ones) in this book. But because essential information will be lost without these routines, we also offer a diskette containing all of the program in this book plus additional useful routines, already typed in and ready to use. See the back of the book for ordering information, or consult your local dealer.

Chapter 2
Bits and Bytes

In order to take advantage of the many graphic capabilities of the '64, we'll need to know a little about the memory structure of the computer and the various registers of the VIC controller. Don't be afraid! We won't bombard you with electronic jargon. Understanding these things requires only some relatively simple math concerning the representation of numbers using the binary system.

For a normal computer, consisting of a large number of electronic connections and building blocks, there are only two conditions out of which its entire little world is made: current on and current off. But because we people demand a good deal more from a computer, we must come up with a way of dealing with this limitation. If we want to represent a number in the computer, we cannot make ends meet with only these two conditions. We could assign the number 0 to the condition "current off" and the number 1 to the condition "current on", but this won't get us far since we want to represent many other numbers besides 0 and 1.

2.1 Decimal system

In the decimal system we use only 10 digits (0-9). But we can represent numbers greater than this range by using a familiar technique. We simply group several digits together to form a larger number. When we to count to 1000, for example, we have already reached the end of our digit supply by the digit 9. Everyone knows that after 9 we simply start over at the beginning (at 0) and write a 1 before the last digit to show that we have already reached 9. The next numbers after 9 are of course 10 (one zero), 11 (one one), 12 (one two), and so on. At 19 we encounter a similar problem. Again we begin with 0 and simply increment the left-most digit by one. The result is 20 (two zero). When the left-most digit reaches 9 (99) we start with zeroes in the digits which have reached 9 and place the number 1 before them. Using this principle we can represent all whole numbers. The individual digits of a number are called places and these places are denoted by ones, tens, hundreds, thousands, etc. A number n whose digits (d_1, d_2, ...) are known can be calculated as follows:

$$n = d_0*10^0 + d_1*10^1 + d_2*10^2 + \ldots$$

where d denotes the ones, tens, hundreds, ... places (note: A number to the 0 power always yields 1, therefore 10 to 0 power equals 1, as does 5 to the 0 power--Exception: 0 to the 0th power is not defined). We can illustrate this differently as follows (the number 3124 is arbitrary):

10^4	10^3	10^2	10^1	10^0
0	3	1	2	4

The top line gives the value of the individual places, the lower line the factor d.

2.2 Binary system

As we said before, we have 10 digits to represent our numbers. Our poor computer on the other hand can represent only 2 digits (0 and 1) directly. How then can it count to 1000? Quite simply: exactly as we do! It starts:

0, 1

and its digit supply is exhausted so it starts again with 0, but places a 1 in front and gets:

10

It continues with:

11,100,101,110,111,1000,1001,1010,1011,1100,1101,...

As you can see, the binary system operates in much the same way as our familiar decimal system. Correspondingly, the value of a binary number can be calculated though the following formula:

$$d = b_0*2^0 + b_1*2^1 + b_2*2^2 + \ldots$$

where the parameters $b_{0,1,2,\ldots}$ represent the individual digits, starting with the first (one's place) up to the highest binary place. If you know the digits of a binary number, it is possible to convert it to a decimal number with the help of this formula. As above, this can be achieved through the following table (here as example the number 10110100):

2^7 =	2^6 =	2^5 =	2^4 =	2^3 =	2^2 =	2^1 =	2^0 =	
128	64	32	16	8	4	2	1	
1	0	1	1	0	1	0	0	
128		+32	+16		+4			= 180

A binary place is called a bit (binary digit) in computer jargon. A bit has two possible conditions yes or no, 1 or 0. In your computer, 8 such bits (or binary places) make up an entity called a byte. With these 8 bits, the decimal numbers 0-255 can be represented. The numbers 0-65536 can be represented with two bytes (=16 bits). If we want to convert a byte to a decimal number (which is necessary if we want to make changes to specific memory locations in BASIC), we use our table or formula.

If we want to go the other way and convert a decimal number to binary, we proceed as follows (using the number 180 as an example):

```
180 / 128 = 1 rem 52
 52 /  64 = 0 rem 52
 52 /  32 = 1 rem 20
 20 /  16 = 1 rem  4
  4 /   8 = 0 rem  4
  4 /   4 = 1 rem  0
  0 /   2 = 0 rem  0
  0 /   1 = 0 rem  0
180(d)    = 10110100(b)

          * rem = remainder
```

Here the number 180 is successively divided by the powers of 2 from the table. The result of each division is one digit of the binary number and the remainder is used for

the next calculation.

We can do the same thing using a different method. This method is well suited for computer programs because it is recursive and quite simple and fast:

```
180 / 2 = 90 rem 0
 90 / 2 = 45 rem 0
 45 / 2 = 22 rem 1
 22 / 2 = 11 rem 0
 11 / 2 =  5 rem 1
  5 / 2 =  2 rem 1
  2 / 2 =  1 rem 0
  1 / 2 =  0 rem 1
```

Here the decimal number is continuously divided by 2. The remainders represent the binary number starting with the one's place. The result of the division is used for the next division until it is equal to 0.

In order to indicate that a number is a binary number, we place a percent sign (%) in front of the number. This is the usual convention and will be used throughout the book.

As you can see from these examples, converting numbers between binary and decimal is a somewhat time-consuming but relatively simple task. Although the computer must work with binary numbers and does so with ease, we would prefer our good old decimal system. Because of the tedious conversions necessary, another number system is frequently used with computers, the hexadecimal system.

2.3 Hexadecimal system

The hexadecimal or base 16 system offers several advantages over decimal when working with computers. As you might have guessed from the name it uses 16 digits. Since we have only 10 in our decimal system, we must find 6 more. The hexadecimal digits are:

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F

The extra digits are symbolized through letters. In order to designate a hexadecimal number as such, it is a convention to place a dollar sign ($) in front of the number. Conversion between hex and the other systems is done in the usual manner. With the help of the following table you can convert a hexadecimal number into a decimal number (as example, $FE2A):

16^3 = 4096	16^2 = 256	16^1 = 16	16^0 = 1	
F	E	2	A	
15*4096	+14*256	+2*16	+10*1	= 65066

This conversion can also be made recursive, as demonstrated below.

```
65066 / 4096 = 15 rem 3626
 3626 /  256 = 14 rem   42
   42 /   16 =  2 rem   10
   10 /    1 = 10 rem    0
    655066(d) = FE2A(h)
```

You can see that this conversion follows the same pattern as that described for binary numbers. The recursive method mentioned there is also usable. Simply divide continuously 16instead of 2 to obtain the digits.

What advantages does the hexadecimal system offer? First, it is possible to represent numbers clearly and with fewer digits than is possible with either the decimal or binary systems. Second, the conversion between binary and hexadecimal numbers is extremely simple. Exactly four binary digits correspond to one hexadecimal digit. A byte can therefore be represented by exactly two hexadecimal digits. For example:

```
% 1101 0110
$    D    6
```

You see how simple it is to represent otherwise long binary numbers. In section 6.4 you'll find a decimal-hexadecimal-binary conversion table for your use.

2.4 Logical operations

Logical operations are useful in order to later change or manipulate the contents of specific bytes or memory locations. These operations are also available in Commodore BASIC: AND and OR. Both are used to combine two binary numbers with each other.

a) **AND:**

Assume that we want to set a certain bit within a byte only when the corresponding bit is set in another byte. The AND operations does this for us. The numbers are compared bit by bit, the result determined by the truth table below:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

The result is 1 only if the corresponding bits of both operands are 1, otherwise the result is 0. If we want to "AND" two complete bytes (8 bits) with each other, the process looks like this:

```
    10110010
AND 01100111
    00100010
```

Here each individual bit of the first byte is compared to the corresponding bit in the second byte and the result determined by the truth table.

This operation is used to change a single bit of byte without affecting the others, as is the next operation.

b) **OR:**

The OR operation can also be illustrated using a truth table:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

The result is 1 if the corresponding bit in either the first or second operand is 1. A byte can be ORed in the same way:

```
   10110010
OR 01100111
   11110111
```

Chapter 3 : Hardware Foundations

Your Commodore 64 offers tremendous capabilities for creating and controlling graphics. But since Commodore 64 BASIC offers no direct commands for making use of these graphics capabilities, all of the programming must be done directly in machine language or using POKEs in BASIC, unless you have a BASIC expansion which adds these commands to BASIC. No expansion can possibly cover all of the capabilities of the Commodore 64. For this reason, a good working knowledge of your computer is essential in order to make full use of all its potential. In addition, it is quite interesting to see how everything works together.

Before we start programming graphics and applications which use graphics, we'll present a detailed description of the '64 hardware involved in generating graphics.

3.1 The VIC registers

First we'll describe the 47 registers of the VIC - the Video Interface Chip, the processor which controls all of the screen output, among other things. These registers control all of the graphic and text functions. Understanding the VIC chip is fundamental to the operation of graphics on the '64, so careful attention should be given to this chapter.

This overview will be supplemented and explained in greater detail in later sections. We'll be working with these concepts throughout the book, so we have included a summary of the registers in section 6.8 The following pages describe the purpose of each register of the Video Interface Chip.

Following the decimal and hexadecimal offset of each individual register (the offset is always added to the base address to get the address of the register) are the name of the register and its initial contents. The initial content is the value written to the register as an initialization value immediately after the computer is turned. The value is given in both decimal and binary.

VIC registers --- Base address: 53248 ($D000)

Reg. dec	hex	Description	Initial value dec	binary
00	$00	X-coordinate, sprite 0	00	%0000 0000

This register contains the 8 least-significant bits (0-255) of the x-coordinate of the first sprite (sprite 0). The most-significant (ninth) bit is stored in register 16. This is necessary because an x-coordinate can be greater than 255.

01	$01	Y-coordinate, sprite 0	00	%0000 0000

As above, only y-coordinate, and without carry.

02-15	$02-0F	Coordinates of the other 7 sprites		

(Same as above). Sprite 1: reg. 2/3; sprite 2: reg. 4/5, and so on.

16	$10	MSB of the X-coordinates	00	%0000 0000

Here are the carries (ninth bits) from the x-coordinate registers. Each bit is assigned to a sprite. Bit 0 is for sprite 0, bit 1 for sprite 1, and so forth.

17	$11	Control register 1	155	%1001 1011

Bits 0-2: screen scroll up/down
Bit 3: =0: 24 lines / =1: 25 lines
Bit 4: =0: screen off / =1: screen on

If the screen is off, the CPU will no longer be interrupted by the VIC (as can occur for up to 40 milliseconds when displaying sprites, for instance) and your

program runs somewhat faster.
Bit 5: =1: standard bit-map mode
Bit 6: =1: extended color mode
bit 7: carry from register 18 ($12)

18 $12 Raster line IRQ 55 %0011 0111
If this register is written to, the number sets the screen raster line at which the VIC will generate an interrupt request to the CPU.
If read, the number returned is the actual raster line which the VIC is currently working on. The carry of this register is placed in register 17.

19 $13 Light pen x-coordinate 00 %0000 0000
X-coordinate (raster coordinate) of the screen position that was being written when a signal came from the light pen (light pen line = 0).

20 $14 Light pen y-coordinate 00 %0000 0000
Same as register 19, except for the y raster coordinate.

21 $15 Sprite on/off 00 %0000 0000
Individual sprite on/off control. Each bit is assigned to a sprite (as in register 16).

22 $16 Control register 2 08 %0000 1000
Bits 0-2: screen scroll left/right
Bit 3: =0: 38 columns / =1: 40 columns
Bit 4: =1: multi-color mode
Bits 5-7: unused

23 $17 Sprite expansion, y direction 00 %0000 0000

Each sprite is assigned a bit. If a bit is set (1) the sprite is made doubly wide (see also reg. 29).

24 $18 VIC base addresses 20 %0001 0100

Here are stored the upper bits of the starting address of the video RAM and the character set memory. By changing these values it is possible to move this area (see also reg. 0 of CIA 2). The assignment is as follows:

Bit 0 : unused

Bits 1-3: address bits 11-13 of the character set (*2048)

Bits 4-7: address bits 10-13 of the video RAM (*1024)

The assignment of these bits given on page 158 of the CBM 64 user's guide is wrong!

25 $19 Interrupt Request Register (IRR) 15 %0000 1111

Allows the user to determine the source orf an IRQ.

Bit 0 = 1: source: raster line IRQ (reg. 18)

Bit 1 = 1: source: sprite-background col. (reg. 31)

Bit 2 = 1: source: sprite-sprite col. (reg. 30)

Bit 3 = 1: source: light pen sends impulse

Bits 4-6 : unused

Bit 7 = 1: at least one of the first 4 bits is 1.

This register must (if it is used) be cleared after the interrupt. This is done by reading the register and rewriting that same value.

26 $1A Interrupt Mask Register (IMR) 00 %0000 0000

Here the programmer can choose which events will be allowed to generate an interrupt. The assignment of

the bits is the same as for register 25. If the same bit in both this and register 25 is set, an IRQ to the CPU will be generated. Set bits in register 26 permit the generatation an interrupt through register 25.

27 $1B Priority 00 %0000 0000

Each sprite is assigned to a bit. If a bit=1, background characters have priority over the sprite. If a bit=0, the sprite has priority over the background.

28 $1C Multi-color sprites 00 %0000 0000

Each sprite is assigned to a bit. If a bit=1, the sprite is displayed in multi-color mode.

29 $1D Sprite expansion, x-direction 00 %0000 0000

Each sprite is assigned to a bit. If a bit=1, the sprite is expanded in the x-direction.

30 $1E Sprite-sprite collision 00 %0000 0000

Each sprite is assigned to a bit. If a sprite is coincident with another, the two bits corresponding to these sprites are set. At the same time, bit 2 in register 25 (IRR) is set. After the event, this register must be cleared because the bits do not automatically reset themselves.

31 $1F Sprite-background collision 00 %0000 0000

Each sprite is assigned to a bit. If a sprite collides with a background character, the bit assigned to that sprite is set. At the same time, bit 2 in register 25 (IRR) is set. After the event, this

register must be cleared because the bits do not automatically reset themselves.

32	$20	Border color	14 %0000 1110
33	$21	Background 0	06 %0000 0110
34-36	$22-24	Background colors 1-3	01 02 03
37/38	$25/26	Sprite multi-color 0/1	04 00
39-46	$27-2E	Color sprite 0-7	01 02 03 04 05 06 07

In addition to these registers, the '64 has other memory locations which are of interest to us when programming graphics. For the sake of completeness we include them here:

a) Sprite definition pointers:
In order to inform the VIC where in its address range it can find the 63-byte definition of a sprite, the last 8 bytes of the video RAM must contain pointers to these definitions (after power-up or reset--memory locations 2040-2047 or $07F8-$07FF). Multiplying the pointer by 64 gives the address of the start of a sprite definition relative to the address range of the VIC (determined by register 0, bits 1 and 0 of CIA 2 [see below]). Each of the 8 bytes is assigned to a sprite (from 0 to 7) and the pointers can accept values from 0-255 (16K addressing).

b) Most-significant address bits of the VIC address range.
In addition to VIC register 24, there is another memory

location with which the video RAM or character set (for example) can be moved. This is register 0 of the CIA 2 (Complex Interface Adapter 2) having address 56576 ($DD00), particularly bits 0 and 1. These two bits make up the two most-significant address bits (bits 14 and 15) of the base address of the VIC.

Caution! The bits active LOW, that is, if a bit=1 it counts as 0 and vice versa!

c) Joystick/paddle/keyboard

The first two registers of CIA 1 are used as follows:

```
Register 0 (address: 56320/$DC00)
Normal operation:
Bits 0-7: row selection of the keyboard
additional tasks:
Bits 0-4: joystick 0: bit 0=1: up
          (port 1)    bit 1=1: down
                      bit 2=1: left
                      bit 3=1: right
                      bit 4=1: fire button
Bits 6/7: paddle set selection: bit 6=1: set A
          (port 1)              bit 7=1: set B
          Only one of the two bits may = 1!
```

For joystick operation, register 0 must first be set up for input with a POKE 56322,224 (register 2 - $DC02). To reset, POKE 56322,255 (does not apply absolutely for register 1).

```
Register 1 (address: 56321/$DC01):
Normal operation:
Bits 0-7: keyboard column
additional tasks:
Bits 0-4: as register 0, except for port 2
          (joystick 1)
```

So much for our short description of the most important registers for '64 graphics. Now we can let you in on many of the secrets of your computer--secrets which for thousands of years have passed down from programmer to programmer under the oath of silence. You'll be astonished how worlds will suddenly open up at your feet and you will feel as if you had been transported to the seventh programmer heaven. So pay attention and prepare yourself!

3.2 VIC operating modes

A great number of things are posible with the Video Interface Chip. These can be divided into three major categories:

-high-resolution graphics (bit mapped mode)
-sprite graphics
-text mode (characters from a character set)

There are two modes which can be selected in addition to these three basic categories:

-normal color mode
-multi-color mode

and another possibility which applies only to text and cannot be used together with the multi-color mode, the:

-extended-color mode

We'll review these modes quickly in order to give you an overview of their operation and capabilities. A more detailed discussion will follow in later sections (for information on access to color RAM, video RAM, graphics storage, and character set storage, see section 3.3).

A. Bit-mapped mode
   a) High resolution
      In the normal high resolution mode, which is selected by setting bits 5 and 6 of VIC register 17 (register 22, bit 4=0), there is a direct connection

between graphics memory and the screen. A bit in the graphics memory corresponds to a point on the screen.

The resolution is 320x200 points (also called pixels), and requires graphics storage of about 8K. The color is determined by the 1K video RAM in which one byte of video RAM determines the color for an 8x8 point field on the screen. The following color assignment applies for each bit in the graphics memory:

bit=0: upper 4 bits of the video RAM (upper nybble)
bit=1: lower 4 bits of the video RAM (lower nybble)

b) Multi-color mode

In the multi-color mode, which is selected by bits 5 and 6 in VIC register 17 and the 4th bit of the 22nd register, two bits of the graphics storage correspond to a double-width point on the screen. The resolution is therefore 160x200 points. Four different colors are selectable for each 8x8-point field. The two bits assigned to each point determine which color that point will be. The color is determined as follows:

bits=00: background color register 0
bits=01: lower nybble of video RAM
bits=10: upper nybble of video RAM
bits=11: color RAM

B. Sprites

Sprites are user-definable objects of a determined resolution, but with alterable size and color. Eight sprites can be placed on the screen at the same time, each fully independent of all the others. Their priority

with respect to each other and the background characters as well as their position on the screen (movement range: 512x256, beyond the edges of the screen) can be specified. A sprite definition occupies 63 bytes of memory. There are two different types of sprites:

a) Normal sprites
   Normal sprites have a resolution of 24x21 points. Each bit in the sprite definition represents a point of the sprite matrix. The color is determined as follows:

bit=0: transparent
bit=1: sprite color register
(registers 39-46)

b) Multi-color sprites

For multi-color sprites, 2 bits in the definition determine a double-width point on the screen. It then follows that the resolution is limited to 12x21 points. These 2 bits determine the derivation of the color of a point in the following manner:

bits=00: transparent
bits=01: multi-color register 0
bits=10: multi-color register 1
bits=11: sprite color register

It is possible to display sprites of both types on the screen at the same time.

C. Character representation

For character representation, a character set (character generator) contains the patterns which specify which bits within the 8x8 matrix (8 bytes are required for each character) are displayed. The character set contains all of the letters, numbers, graphics symbols, and punctuation which can be displayed. A control byte in the video RAM indicates which of the 2 X 256 characters will be displayed on the screen at a given location.

a) Normal character mode:

In this display mode, the byte in the video RAM is used as a pointer to a pattern in the character generator. A maximum of 256 different characters can be

displayed on the screen at one time. A bit in the pattern determines the color of a single point of the character. The colors come from:

bit=0: background color register 0
bit=1: lower nybble of the color RAM

b) Multi-color character mode:

In this display mode, both normal and multi-color are possible. If bit three of the color RAM is 0, the character is displayed in the normal 8x8 matrix. The character is displayed as described above except that the color of the character is limited to one 8 colors. Bit three of the color RAM is always 0.

If bit three of color RAM is 1 however, each pair of bits in the character matrix determine the color of a double-width point on the screen. The character resolution is thereby limited to 4x8 points. Thus a character displayed in the multi-color mode can contain up to 4 colors. The source of these colors is determined by the two bits in the matrix:

bits=00: background color register 0
bits=01: background color register 1
bits=10: background color register 2
bits=11: lower three bits of the color RAM

Color 3 may be one of only 8 colors. The other three colors are the same for all characters.

c) Extended-color mode

In this display mode, each character on the screen can have one of 4 background colors. The character

representation corresponds to the normal mode (8x8 matrix). For bits in the character set that are set (bit=1), the color comes from the color RAM as in the normal mode. For bits in the character set that are reset (bit=0), there are four possible sources for the color. The two highest bits (bits 6 and 7) of the video RAM determine these sources:

bits=00: background color register 0
bits=01: background color register 1
bits=10: background color register 2
bits=11: background color register 3

Of the 8 bits in the video RAM, only 6 remain as a pointer to the character set. Therefore, only 64 different characters may be displayed on the screen at any one time in the extended color mode.

This concludes our overview of the various operating modes of the video controller. These modes will be examined in greater detail in sections 3.4 through 3.6.

3.3 Memory management

This section requires a knowledge of memory organization, addressing, and related topics and should therefore be skipped by beginners. The reader who knows nothing about the graphics capabilities or who knows nothing about graphics in general should read section 3.3.2.1 and then proceed directly to section 3.4.

A knowledge of these topics, particularly the areas pertaining to the memory access of the computer, is necessary for making full and effective use of the Commodore 64's graphics. In this section it will become clear that the most valuable functions can be accessed only in machine language. For this reason, a knowledge of machine language is very helpful.

There is often some hesitation in learning machine language. But it is relatively easy to learn, especially for experienced BASIC programmers. We recommend that you consult one of the books dealing with this topic such as The Machine Language Book for the Commodore 64, also available from Abacus Software. The tools necessary are an assembler and a machine language monitor. You'll soon find yourself programming in assembly language rather than the slow and clumsy BASIC. If, however, you do not want to learn machine language, but still want to work with graphics, we strongly recommend a graphics extension package. These packages offer simple commands which make graphics programming child's play. When looking for an extension package, remember that speed is very important in such a program.

A certain amount of memory is used for text and graphics to store the contents of the screen. For example, bitmapped graphics requires 8K plus an additional 1K for the

color memory or 2x1K for multi-color. This memory area may not be used for other purposes after it has been set aside for graphics use. For this reason, there are several ways for the user to choose this area of memory himself to make optimum use of available memory. At the same time, there are areas of memory which cannot be used or are very difficult to use (at least from BASIC).

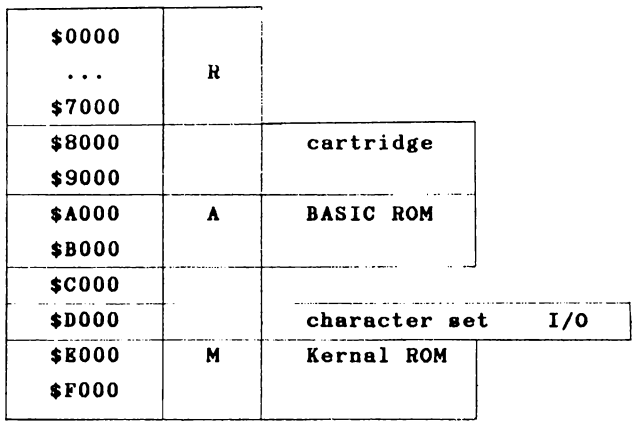
To understand what is to follow, a few things must first be said about the memory organization of your computer. The '64 has a 6510 main processor, also called a CPU (Central Processing Unit), which is required for all computing procedures. This 6510 has the same instruction set as its predecessor, the 6502, and is therefore machine instruction compatible with it. The 6510 differs from the 6502 in that it has additional lines which are used for memory management. It has two unique registers at addresses 0 and 1. Only register 1 is of interest to us here.

Another similarity of the 6510 to the 6502 is its address range. By address range we mean the size of the memory which the computer can control. The address range of the Commodore 64 is 64K. This range is limited by the 16 "address lines" that are connected to the 6510. These 16 address lines imply 16 bit addresses. Thus addresses 0-65536 can be controlled with 16-bits.

The '64 has 64K of RAM. RAM stands for random access memory which can be written to as well as read from. Its contents are lost when the computer is turned off. It serves therefore as working memory. In contrast, ROM (read only memory) can only be read from. Its contents cannot be changed, even when the computer is turned off. ROM on the '64 contains the kernal, the BASIC interpreter, the character generator, and so on.

In addition to this 64K of RAM, the '64 has various ROM areas and registers of peripheral devices (VIC, CIA, synthesizer, etc.) which take up a total area of 24K. How is all of this memory managed with only 64K of address space?

The answer is that the memory management is handled by a technique called overlapping. Several memory areas (such as RAM and ROM) may all have the same addresses, that is, they occupy the same place in memory. A map of the memory might look like this:



After power-up, all of the following memory areas are readable:

$0000-$9FFF: RAM
$A000-$BFFF: ROM
$C000-$CFFF: RAM
$D000-$DFFF: I/O
$E000-$FFFF: ROM

What the computer will access at a given address at any one time is determined by a number of factors. Furthermore, the VIC and the 6510 can access different areas of memory at practically the same time (or even the same addresses but one reads ROM while the other reads RAM). We will first examine the memory accessing of the 6510 and then move on to the VIC. Both are important when working with graphics.

3.3.1.1 Reading a byte

If you are reading a byte from a specific address (with PEEK, for example), the decision concerning which of the overlapping regions is accessed is made by the contents of the data register of the 6510 CPU (memory address 1). The first three bits (0-2) have the function of switching certain memory areas. The three bits are set to the configuration given previously after the device is turned on. They have the following functions:

A. Bits 0/1 - LORAM/HIRAM:

a) Bits 0/1=11:

If these bits are both 1, reading from the range $A000-$BFFF accesses the BASIC ROM. Reading from the range $E000-$FFFF accesses the kernal ROM. This is the normal condition. If a cartridge is inserted in the $8000 range, it is enabled instead of RAM. The memory configuration then looks like this:

$0000	R
...	A
$7000	M
$8000	cartridge
$9000	(ROM)
$A000	BASIC ROM
$B000	
$C000	
$D000	I/O range
$E000	Kernal ROM
$F000	

b) Bits 0/1=10:

If only bit 1 is set, the cartridge and the BASIC ROM are switched out and the underlying RAM will be read instead. We have the following layout:

$0000	R
...	
$7000	
$8000	A
$9000	
$A000	
$B000	M
$C000	
$D000	I/O range
$E000	Kernal ROM
$F000	

This configuration can be used (even from BASIC) to modify BASIC by copying the interpreter to the RAM, making changes and then switching the ROM out and RAM in. A BASIC program to do this might look like this:

```
10 FOR AD=10*4096 TO 12*4096-1
20 POKE AD, PEEK(X) : REM COPY BASIC ROM TO RAM
30 NEXT AD
40 REM POKE IN CHANGES HERE ...
50 POKE 1, PEEK(1) AND 254 : REM ENABLE RAM
```

It must be noted that if a cartridge is inserted, the range from $8000-$9FFF must also be copied before the switch is made because the cartridge ROM is switched out at the same time BASIC is.

Be careful when working with CPU register 1 because altering it has far-reaching consequences on the way in which memory is accessed and if an error is made, the power switch is often the only way to bring your computer "back to life" (don't worry--it is not harmed in any way. Programs are different matter altogether, however).

c) Bits 0/1=01:

If only bit 0 is set, the entire ROM range is switched out. The I/O range from $D000-$DFFF remains undisturbed and can be controlled without change. We get 60K of usable RAM this way:

$0000	R
...	
$7000	
$8000	A
$9000	
$A000	
$B000	M
$C000	
$D000	I/O range
$E000	RAM
$F000	

d) Bits 0/1=00:

If both bits are 0, all of the RAM is switched on. No ROM and no I/O functions may used (although the VIC and the other peripherals can still read their registers and the screen remains unchanged). The configuration is quite simple:

$0000	
...	R
$7000	
$8000	
$9000	
$A000	A
$B000	
$C000	
$D000	
$E000	M
$F000	

This is the only way of using the RAM underneath the I/O range and character set.

B. Bit 2 - CHAREN:

This bit serves to make the character generator (see below) accessible to the 6510, and thereby to the programmer, so that it can be copied and changed, etc.

a) Bit 2=1:

This controls the $D "super-page" ($D000-$DFFF) of the computer's memory. The other addresses are not affected. This bit is normally set. When it is set, the 4K character generator at $D000-$DFFF can be read only by the VIC. The I/O addresses, the VIC, SID (Sound Interface Device), CIA registers, and the color RAM are accessible to the 6510. This memory is arranged as follows:

$D000 ...	$D400 ...	$D800 ...	$DC00	$DD00	$DE00	$DF00
VIC regs.	SID regs.	Color RAM	CIA 1	CIA 2	I/O 0	I/O 1

b) Bit 2=0

If this bit is cleared, the programmer (actually the 6510) can read the contents of the character set storage. If you try this from BASIC, your computer will crash within 1/60th of a second because an interrupt routine is executed 60 times per second which manipulates the timer found in CIA 1, which is, of course, no longer accessible. In machine language, the interrupt flag must be set (command: SEI) in order to keep this routine from being called. The interrupt can be enabled later by executing CLI. The same thing must be done when the kernal ROM is switched off by clearing bit 1.

3.3.1.2 Writing a byte

If a write access is made to the memory (through POKE, for example), the relationship changes fundamentally:

Since it makes no sense to try to write something to ROM, all write accesses automatically access the RAM underneath the ROM regardless of the condition of register 1, with one exception: the range from $D000 to $DFFF. Here only the I/O range is normally accessible. If we want to make reference to the RAM at these locations, we have two options:

-turn off all of the ROM
-turn on the character set storage

The first is done by clearing the first two bits (0 and 1) of CPU register 1 to 0, the second condition is attained by clearing bit 2 of this memory location (see above).

By now you should be able to understand the small BASIC program presented earlier. The contents of the BASIC ROM are read in line 20 by the command POKE AD, PEEK(AD), and written back to the underlying RAM, not the ROM.

3.3.2 The memory access of the VIC

In addition to the main processor, which is necessary for running all of programs, the video processor must naturally access the memory in order to get screen information such as color or point settings. Because access to such things as the kernal ROM or BASIC interpreter does not make sense here, a different approach is taken than is used for

the CPU control. The VIC undertakes switching various memory ranges independently. For us this results in a more or less static picture in contrast to the many possibilities available to the programmer with the 6510. We need only know which memory areas the VIC controls for which purposes and which it doesn't. At the same time, the memory control of the VIC can appear extremely dynamic. It can itself determine various conditions, where which memory functions of the video controller should lie. In other words, you have the ability to move graphics storage, character set, video RAM, and sprite storage anywhere it seems appropriate to you.

Before we delve into this, we want to first say a little about the memory functions of the VIC in order to clarify the principle which will from now become our daily bread.

3.3.2.1 VIC memory functions

In order to carry out the various tasks to which the VIC is assigned (text, graphic, sprites, color, ...), it requires an extensive memory space with quite different functions:

- character generator
- video RAM
- color RAM
- graphics storage
- sprite definitions

In the following discussion we point out the significance and uses of each of these individual positions. The more

detailed functions and exact construction will be handled in sections 3.4 to 3.6. The listing found here is intended for orientation and to clarify some of the principles involved.

a) Character generator:

By character generator (also called character set or character set storage) we mean the memory which contains the patterns for each individual character, which we can display on the screen with a simple keypress. It occupies a total of 2x2K bytes and contains the information for 512 characters, although only a part of these may be displayed simultaneously (see sections 3.2, 3.6, 4.4).

b) Video RAM:

The video RAM occupies approximately 1K of memory and has various tasks. Normally, on power-up it serves as character storage for the screen code. Character storage should not be confused with "character set storage." The screen code is a value that serves as a pointer to a pattern in the character generator. The VIC then uses the pattern to display the chosen character on the screen. In graphics mode, video RAM has the function of color memory. It determines the point and background color or multi-color colors 1 and 2 for each 8x8 point field (in MC: 4x8) on the graphics screen (see Sections 3.4, 3.6, 4.4).

Additionally, the pointers to the eight sprite definitions are stored in the last eight bytes of video RAM.

c) Color RAM:

The color RAM occupies about 1K and lies in the range

$D800-$DFFF (55296-56319). It is readable and writable, but only the lower four bits are active. The upper four bits cannot be changed and are always set. In the normal mode it serves as storage for color of the text characters on the screen. In the graphics mode it has a function only for multi-color. There it represents the multi-color color 3 for a 4x8 point field on the graphics screen.

d) Bit-mapped graphics memory:

The bit-mapped graphics memory contains an image of the graphics screen. The state of each point on the screen is determined individually. With a resolution of 320x200 (in high-resolution graphics), there are a total of 64,000 points, for which a storage of almost 8K is required.

e) Sprite definitions:

A memory area 63 bytes long must be reserved for each sprite definition. These 63 bytes contain the 21x24 points of a normal sprite.

3.3.2.2 VIC memory control

The control of the various memory areas by the video controller is considerably simpler and clearer than the many other options which the CPU offers. RAM (even in the range from $D000-$DFFF = 53248-57343) is accessed by the video controller, even when only the ROM is available to the programmer (thereby independent of CPU register 1). This is extremely important to remember because the graphics memory can be placed under the ROM. Doing this does not use any BASIC storage space. The same applies to the following:

- video RAM
- graphics storage
- sprite definitions
- character generator

These memory areas are always taken from RAM--with two exceptions:

If the memory from $1000-$1FFF (4096-8191) or from $9000-$9FFF is accessed (for example if you place the graphics page at $8000-$9FFF or choose the blocks 64-127 [range $1000-$1FFF] for sprites in the power-up condition), the information is not read from the RAM in these areas, but from the character generator ROM which lies at $D000-$DFFF.

The first peculiarity can be explained by the fact that after turning on your computer, only the lower 16K is controllable by the VIC (see below). The character set needed to draw the define characters on the screen lies in the range $D000-$DFFF, beneath the I/O addresses. For this reason, addresses in the range $1000-$1FFF have a special status and the character set is logically (not physically) located there. The other functions of this area must be sacrificed, unfortunately. The limitation at $9000 results from this same reason.

Things are even simpler for the color RAM. Since it is not movable as are the other areas it is always located in the range $D800-$DBFF (55296-56319). Color RAM also has RAM lying underneath it which for the programmer lies at the same level as the I/O addresses (see above). The color RAM at $D800 may not be exchanged for the normal RAM at $D800.

3.3.2.3 Moving screen memory

One of the nicest and most practical things about the memory management of the video controller is the ability to move screen memory around. You can, for example, put the bit-mapped graphics memory at $E000 (41440) underneath the ROM instead of at $2000 (8192) where it is usually placed. Or perhaps you would like to maintain two or more graphics pages so that you can work on one while the other is being displayed.

You can also move the character set to other areas and create your own personal characters. So many possibilities are available that you probably don't have the time to pursue them all!

With each movement you must keep track of the memory area which the VIC can control. For example, you could not place a sprite definition in the RAM at $1000-$1FFF because the VIC reads not RAM but the character set ROM at $D000-$DFFF. For this reason, an understanding of the material in section 3.3.2.2 is absolutely necessary for the following discussion!

a) Changing the VIC's address range

The video controller can address only a 16K range ($0000-$3FFF). The '64's address range is 64K--an area four times as large. The VIC does does not have the top two address bits (bits 14 and 15). They must be provided from the outside. Register 0, already mentioned in section 3.1 is used for this:

register 0, bits 0/1 of CIA 2 ($DD00=56576)

These two bits represent the two most-significant bits for the memory address of the VIC (underlined in the following illustration):

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

You could simply set these and have the complete address. There is a catch, however. These two bits are active LOW, that is, if they are set, they are treated as if they were cleared for the purpose of figuring the address and vice versa. If we want to arrive at the correct address we must invert these bits. Once we do this, we get the range which the VIC can control. It automatically moves all of the screen functions which are controlled by the VIC (except for the color RAM) in 16K steps:

- video RAM
- graphics storage
- character generator
- sprite definitions

For your understanding, we present the memory ranges in tabular form which can be selected through a specific layout of these bits:

B 0/1	Addr B	Address range
11	00	$0000-$3FFF (0-16383)
10	01	$4000-$7FFF (16384-32767)
01	10	$8000-$BFFF (32768-49151)
00	11	$C000-$FFFF (49152-65535)

By "B 0/1" we mean the actual layout of the two bits in register 0 of CIA 2. "Addr B" gives the resulting bits 14

and 15 for VIC addressing. The original assignment is B 0/1=11, the first case in the table. The video RAM then lies at $0400-$07FF (1024-2047) and the character set (through the special layout of address $1000 [see above]) lies at $D000 (53248).

An example: Say you want to move the storage of characters to $C400 (50176 = 49152+4*256). For this purpose you need merely give the command:

```
POKE 56576, PEEK(56576) AND 253 OR 0
```

and the VIC takes all of its information from the memory in the range from $C000 to $FFFF, which in our case would lead to wild chaos on the screen (only the color can be properly changed because of its immobility). The command

```
POKE 56576, PEEK(56576) AND 253 OR 3
```

restores the screen (what you type in on the keyboard blindly will naturally appear on the screen once you press <RETURN>, assuming that you do not make an error).

b) Moving the video RAM:

In addition to moving the address range of the VIC, the video RAM, among other things, can be moved separately in small steps within this 16K addressing range. This movement is possible by changing the contents of VIC register 24 ($18), particularly bits 4-7 (see 3.1). These four bits determine bits 10-13 of the address of the video RAM. The following illustration should make this clear:

Address bits	$F E	D C B A	9 8 7 6 5 4 3 2 1 0
	CIA2 B0/1	reg. 24 bits 4-7	internal to the VIC

The video RAM can therefore be moved in 1K steps within the 16K address range. After power-up these four bits are assigned %0001. This is why our video RAM (in its function as text storage) lies at $400 (1024). It is important to note that except for the possibility given under a), this address applies only to the video RAM. Changing these bits leaves the other memory ranges unaltered.

An example: We want our video RAM, the text storage, which still lies at $400, moved to $800 (2048). This accomplished with the command:

```
POKE 53248+24, PEEK(53248+24) AND 15 OR 2*16
```

and we get nonsense since this is the start of the BASIC storage. To return the old video RAM, we enter:

```
POKE 53428+24, PEEK(53248+24) AND 15 OR 1*16
```

This movement possibility can be used, for example, when you have such large BASIC programs that they must use the memory area at $400-$7FF, or if you use two text pages, which are constantly alternated, or ... or ... or ...

c) Moving the character generator:

In addition to the video RAM, it is also possible to move the character generator within the 16K range of the VIC. Here also register 24 of the VIC serves as an intermediate storage for some address bits. This time there are only 3, which represent the address bits 11-13. This

means that we can move the 2x2K character generator only in 2K increments:

Address bits	$F E	D C B	A 9 8 7 6 5 4 3 2 1 0
	CIA2	reg.24	internal to the VIC
	B0/1	bit1-3	

It is interesting that the operating system of the computer also makes use of these three bits. When you switch to the alternate character set with the <SHIFT><C=> key combination, the computer changes the 11th bit of the character set address by changing the first bit of VIC register 24 (see also section 3.6).

It is important to recall what was said in section 3.3.2.2 about the special layout of the address range $1000-$1FFF (4096-8191). If you want to move the character generator, it would be helpful if you have also read section 3.3.1.

d) Moving the graphics storage:

The location of the graphics storage can also be selected. Because it is 8K long, there are only two places it can go in the 16K address range. For this reason, there is only one bit to select for it (in addition to the general movement, of course). It is the 3rd bit VIC register 24, which has two tasks:

- character set movement
- graphics storage movement

As for the character set, this determines the 13th bits of the graphics storage address:

```
Address bits  $F E D C  B A 9 8  7 6 5 4  3 2 1 0
              CIA2 B     internal to the VIC
              B0/1 3
```

If your VIC address range lies at $0000-$3FFF (0-16383), for instance, you can choose whether the graphics storage should begin at $0000 or $2000 (8192) (In this case, the latter is to be recommended, otherwise the zero-page, stack, and so on will be destroyed).

3.4 Bit-mapped graphics

After so many details about the registers and memory management, we will now discuss the organization of the high-resolution and multi-color graphics, also called bit-map graphics. This section and the four following it provide the hardware background for the programming capabilities presented in Chapter 4. You may find that you do not understand all of the material in this section, but you should read it through and do the best you can. Don't be discouraged. A book, especially a book such as this, should be read at least twice. You'll probably work daily with many passages if you really want to become proficient in graphics programming. But let's begin:

3.4.1 Colors

Your Commodore 64 has the ability to display 16 colors in both graphics operation as well as in text. These 16 colors each have something called a color code which is stored in the various registers as a binary number. If, for example, a 0 is POKEd into the 32nd register of the Video Interface Chip, the outer screen border assumes the color black. The following table lists the colors and their assigned color codes (a complete table can be found in Section 6.3.

Code		Color	Code		Color
Dec	Hex		Dec	Hex	
0	$00	black	8	$08	orange
1	$01	white	9	$09	brown
2	$02	red	10	$0A	light red
3	$03	cyan	11	$0B	grey 1
4	$04	purple	12	$0C	grey 2
5	$05	green	13	$0D	light green
6	$06	blue	14	$0E	light blue
7	$07	yellow	15	$0F	grey 3

The colors in this table are used for all types of graphics whether you are working with sprites, graphics, text, or anything else of that nature. Information about setting these is given in the appropriate sections.

3.4.2 High-resolution graphics (HGR)

Your computer has basically two different graphics modes:

- high-resolution graphics (HGR)
- multi-color graphics (MC)

The first offers you a graphics field of 320 points in the x-direction (horizontal) and 200 points in the y-direction (vertical). One speaks of a resolution of 320x200. This gives you 64,000 points of equal size on the screen.

Naturally, graphics must be stored just as text or color is. The VIC must renew the picture on the screen every 1/20th of a second. This is done with the help of the bit-mapped graphics memory. Each point is individually access-

ible and represented in graphics storage by a bit. As you know (see chapter 2) a bit is a single binary digit and can hold the value 1 or 0. Eight successive bits are called a byte. A byte can hold 2 to the 8th power = 256 different values. These come from the various combinations of set (1) and reset (0) bits.

A byte therefore represents 8 points on your screen. It follows that 64000/8 = 8000 bytes (almost 8K) are required to store the entire screen contents. As you may know from Section 3.3.2.3, this 8K can be placed anywhere in the 64K memory of the computer. Once you have done this (see section 4.2.1.1) you can begin. First, however, you must learn how the organization of the graphics picture proceeds from the memory information:

a) Graphics organization:

You can think of the graphics memory as a direct image of the screen. The actual situation is not so simple, however. The graphics memory has the same layout as the character generator described later because this is easier to do from a hardware standpoint. A character consists of 8x8 points. Each character therefore has an 8x8-point matrix. The foundation of the graphics storage is also an 8x8 matrix which is represented by 8 bytes in the graphics memory. Since such a matrix is 8 points high and 8 point wide, a total of 320/8 = 40 such matrices fit in a line and 200/8 = 25 in a column (exactly as the organization of the text mode). One byte yields the information for an 8-point wide row of such a package. Eight bytes below each other (naturally after each other in memory) represent this block. Thereby the corresponding bits of each of these 8 bytes--the bits having the same number or place value--form a column of points. This

can be clarified with a small illustration:

		Column 0								Column 1								....
	Bit:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	etc.
	Byte 0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
L	Byte 1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
i	Byte 2	.	.	.	.	.	.	.	.	.	.	.	.					
n	Byte 3	.	.	.	.	.	.	.	.									
e	Byte 4	.	.	.	.	.	.	.	.									
	Byte 5	.	.	.	.	.	.	.	.									
0	Byte 6	.	.	.	.	.	.	.	.									
	Byte 7	.	.	.	.	.	.	.	.									
	Byte 320	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
L	Byte 321	.	.	.	.	.	.	.	.	.	.	.	.					
.	Byte 322	.	.	.	.	.	.	.	.									
1	etc.																	

Each column consists of 8 vertical rows, each line of 8 horizontal rows. This allows 320 vertical and 200 horizontal rows on the screen (resolution of 40 columns x 25 lines). Each column, line, and row is numbered (starting with 0) and so can be exactly located. It is important to follow these arrangements in order to avoid mix-ups.

In order to have a simple method of addressing points, we give each point a coordinate. This consists of two values (x and y), the number of the horizontal (x-coordinate) line and vertical (y-coordinate) row in which the point is to be found. To calculate the position of the corresponding byte and bit from this coordinate, we need the formula found in section 4.2 under "Drawing a point."

In conclusion, you may have noticed that not all of the 8K bytes reserved for graphics storage are actually required; only 8000 ($1F40) are used. The remaining 192 ($C0) bytes can be used for other purposes.

b) Color organization:

For every high-resolution picture there is a corresponding color memory. This color memory is also used as video RAM in text mode. The video RAM is normally used to store text or general characters which appear on the screen so that the VIC can create the picture. If you do not move it, the places where characters were stored will appear as colored blocks when switched to graphics mode (see section 3.3.2.3 for information on moving the video RAM). This effect can be seen with the following program:

```
100 V = 53248 : REM BASE ADDRESS OF THE VIC ($D000)
110 REM
120 REM ************
130 REM ** PART 1 **
140 REM ************
150 REM
160 REM TURN GRAPHICS ON
170 POKE V+17, PEEK(V+17) OR 6*16 : REM SET BITS 5 AND 6
    OF VIC REGISTER 17
180 REM MOVE GRAPHICS STORAGE TO $2000 (8192)
190 POKE V+24, PEEK(V+24) OR 8 : REM SET BIT 3 OF VIC
    REGISTER 24
200 REM WAIT 198,255 : GOTO 220 : REM WAIT FOR KEY
210 END
220 REM
230 REM ************
240 REM ** PART 2 **
```

```
250 REM ************
260 REM
270 REM TURN GRAPHICS ON:
280 POKE V+17, PEEK(V+17) AND 9*16+15 : REM CLEAR BITS 5
    AND 6 OF VIC REGISTER 17
290 REM RESET CHARACTER GENERATOR:
300 POKE V+24, PEEK(V+24) AND 15*16 + 7 : REM CLEAR BIT 3
    OF VIC REGISTER 24
310 END
```

This program can be run in two variations:

1) as is
2) after removing the first REM in line 200

In the first case the program ends immediately after turning the graphics on and you can make additional input, although it will not appear on the screen but--as mentioned--will be shown as colored squares. If you want to switch back to normal text, enter:

```
RUN 220
```

In the second case, the program waits for a key press and then automatically switches the graphics off (see section 4.2).

The graphics/color relationship is clear in this example. A byte in the video RAM determines the color of a 8x8-point field on the HGR screen. In HGR, both the color of the background, the color of the reset points (bits=0), and the color of the set points can be selected from a choice of 16 different colors. The upper nybble (half--4 bits) of each byte determined the point color

and the lower nybble the background color. The color resolution is considerably inferior to that which the normal graphics allows. This is necessary because it would take far too much memory if each of the 64,000 points could be given its own color (in addition, an ordinary television would have problems displaying it). This would require 64,000 * 4 = 256,000 bits or 32K RAM. The operation speed necessary would also require a separate graphics processor!

The video RAM is organized somewhat more simply than the graphics memory. Here byte after byte and line after line are stored in sequential order. The organization looks like this:

	Column									
Line	0	1	2	3	4	5	6	7	...	39
0	$00	01	02	03	04	05	06	07	...	1D
1	$1E	1F	20	21	22	23	24	25	...	4F
2	$50	51	52	53	....					
...										
24	$3C0	3C1	...							

When using high-resolution graphics, each byte of the video RAM is assigned to a predefined, defined 8x8 field on the screen. If you know the memory address of a point (deducting the starting address of the graphics memory), you need only divide this by 8 to find the address of the corresponding video RAM byte, to which one must yet add the starting address of the video RAM.

3.4.3 Multi-color graphics (MC):

Before we start this section, we want to make sure that you have read the last few paragraphs (3.4.2) because they are important for understanding this section as well.

As discussed earlier, the Commodore 64 has quite a high graphics resolution. The color selection (despite 16 different shades) is rather limited, however. In order to balance this limitation (necessary because of space considerations), the designers decided to introduce a second graphics mode which allows a larger color selection, at the expense of a lower graphics resolution. This is the multi-color mode.

The multi-color mode makes it possible to use 4 colors in an 8x8 block instead of 2. Here too, 8K of graphics memory is used. Now, however, two bits of each byte are required in order to define a double-width point. The resolution becomes 160 double-width points in the x-direction (double-width because otherwise the screen window would only be half as large) and 200 points in the y-direction (resolution: 160x200).

The color comes not only from the video RAM but from background register 0 of the VIC and the color RAM. We divide these areas into 4 color channels, which are numbered 0-3. Each bit pair, pertaining to each point and containing the values 0-3 (%00-%11), tells the VIC from which channel it should get the color for that point (In HGR, the one bit pertaining to each point determined whether the color would come from the background color register or from the point-color channel). The graphics memory does not contain the color code of a point, but where this color can be found. The bit-pair/channel/memory relationship is shown in the following table:

Bit code	Channel no.	Memory location of the channel
00	0	register 33 of the VIC
01	1	lower nybble of video RAM
10	2	upper nybble of video RAM
11	3	color RAM

By bit code we mean the binary number which the two bits pertaining to a point represent. The memory location of the channel is the part of the memory where the color code is stored.

An example: In the graphics memory a point is represented by two bits with the assignments 0 and 1. The resulting bit code %01 refers to channel 1 and therefore to the lower 4 bits of the corresponding byte of the video RAM. This byte is used exactly as explained in section 3.4.2 (HGR).

New here is the use of the color RAM for channel 3. This color RAM is normally used for the color of the text in the video RAM. Because the color RAM is not movable, we have conflict when we want to use text and multi-color at the same time. The color RAM is organized the same way as the video RAM.

As in HGR, these colors can be determined for each 8x8 point (4x8 in this case) field. This allows us to give each such field its own color combination. Only channel 0, the background color which originates from the VIC background color register 0 applies to the entire contents of the graphics screen.

It is important to remember the distortion in the x-direction which results from doubled point-thickness when working with multi-color graphics.

For a better understanding we have included an illustration of the structure of the graphics memory in multi-color:

		Column 0								Column 1								....
	Bit:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	etc.
	Byte 0	<->		<->		<->		<->		<->		<->		<->		<->		
L	Byte 1	<->		<->		<->		<->		<->		<->		<->		<->		
i	Byte 2	<->		<->		<->		<->		<->		<->						
n	Byte 3	<->		<->		<->		<->										
e	Byte 4	<->		<->		<->		<->										
	Byte 5	<->		<->		<->		<->										
0	Byte 6	<->		<->		<->		<->										
	Byte 7	<->		<->		<->		<->										
	Byte 320	<->		<->		<->		<->		<->		<->		<->		<->		
L	Byte 321	<->		<->		<->		<->		<->		<->						
.	Byte 322	<->		<->		<->		<->										
1	etc.																	

In this diagram, the "<->" indicate the double-width points which appear on the screen.

3.5 Sprites

One of the most interesting attributes of your Commodore 64 is of course the sprites. Sprites are independent graphic designs which can be moved independently of each other and the rest of the screen contents in the text or graphics window. A total of 8 sprites may be displayed on the screen at one time.

Sprites can change their color, their size, and their priority over the background characters and with respect to each other. Collisions between two sprites and between sprites and the background are automatically detected. In addition, you have the opportunity to choose between two sprite modes

- normal
- multi-color

for each sprite. All of these functions can be easily realized with the help of the VIC (video controller 6567) and its registers. First, however, we want to concern ourselves with the construction of sprites. We have already gone over binary arithmetic and the various registers in Chapter 2 and section 3.1 and these should suffice for the following discussion although you may wish to reread them.

You can (as said before) display a total of eight different sprites simultaneously on the screen. Each sprite is assigned a specific number (0-7) which will acompany it throughout the section.

3.5.1 Organization and color of normal sprites

Every normal sprite consists of 504 points which can be individually set or reset. A 24x21 point matrix is used. Therefore, a sprite is 24 points wide and 21 points high. Within this matrix you can create various graphics or figures.

In order to save the definition of a sprite, a total of 504/8 = 63 bytes are required since each point requires exactly one bit and a byte contains 8 bits. Since each sprite has a width of 24 points, these fit into a row of exactly 24/8 = 3 bytes. This means that the first three bytes determine the first 24 points of the sprite. The rest of the rows are defined in the same way. Each three successive bytes determine one row of points. The next row then begins with the next byte. We can illustrate this with the following sketch:

		Column 0	Column 1	Column 2
Row/bit:		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
0	byte 0	. . . . .	. . . . .	. . . . .
1	byte 3	. . . . .	. . . . .	. . . . .
2	byte 6	. . . . .	. . . . .	. . . . .
3	byte 9	. . . . .	. . . . .	. . . . .
4	byte 12	. . . . .	. . . . .	. . . . .
			etc.	
20	byte 60	. . . . .	. . . . .	. . . . .

If you want to define a sprite, you would enter the proper bytes column by column and row by row in succession. You will learn how this is done in BASIC in Chapter 4.

Each of the eight sprites may be displayed in any of the 16 colors available. All of the set points in the sprite are given the color in the corresponding sprite color register (VIC registers 39-46). To display sprite 5 in white, you place the value 1 (for white) in VIC register 39+5 = 44, memory location $D02C (53292). All reset points will be transparent, meaning that what ever is "under" these points will be displayed.

3.5.2 Definition and color of multi-color sprites

Not all sprites have this 24x21 point definition. You can choose between high-resolution and multi-color sprites. The multi-color sprites are formed in the same way as multi-color graphics. Because of this, they have only half the resolution in the x-direction as the normal sprites. Each line has 12 double-width points. The sprites may be formed of four different colors, including the background color, while the high-resolution sprites may contain only one color. The four colors are comparable to the channels of the MC graphics.

In order to determine which color each of the 12x21=252 double-width points should have, two bits (bit code) are used, which can contain the values 0-3 (%00-%11) and thereby determine the color channel. The VIC then gets the color of the point from this channel. The assignment of the channels to the individual bit codes is demonstrated in the following table:

Channel no.	Bit code	Color storage
0	00	transparent
1	01	multi-color reg. 0 (VIC reg. 37)
2	10	multi-color reg. 1 (VIC reg. 38)
3	11	sprite color reg. (regs. 39-46)

As you see, only the color of channel 3 can be different for each the sprites because there is one color register for each sprite. The other colors (colors 1 and 2 in addition to the background color) are the same for all of the sprites because they are obtained from the same registers for all sprites. Contrary to the CBM 64 User's Guide, all 16 colors may be used for sprites in the multi-color mode.

In order for a sprite to appear on the screen in multi-color mode you must instruct the VIC to display it as such. Register 28 ($1C) is used for this, in which each of the 8 bits is assigned to one of the 8 sprites. If a bit is set here, the corresponding sprite will be displayed as a multi-color sprite. The bit assignment of this register is the following:

Bit:	b7	b6	b5	b4	b3	b2	b1	b0
Value:	128	64	32	16	8	4	2	1
Sprite:	s7	s6	s5	s4	s3	s2	s1	s0

To use sprite 4 as a multi-color sprite, for instance, simply set bit number 4 in register 28 of the VIC. If you are using BASIC, this is done as follows:

```
POKE 53248+28, 16
```

If you want to represent several sprites this way (for example, sprite 1, 5 and 7), enter the following:

```
POKE 53248+28, 1 OR 32 OR 128  or:
POKE 53248+28, 1 + 32 + 128
```

The organization of a multi-color sprite is identical to that of a normal sprite.

		Column 0								Column 1								Column 2							
Row/bit:		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	byte 0	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	
1	byte 3	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	
2	byte 6	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	
3	byte 9	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	
4	byte 12	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	
													etc.												
20	byte 60	<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->		<->	

A "<->" stands for a double-width point, coded with a bit pair.

3.5.3 Sprite definition - Color

When you want to display a sprite on the screen, there are a few things you must consider. First you must give some thought to the appearance of the object and convert the definition into the required 63 values. Section 4.3 will be of help to you in this. It includes an easy-to-use sprite editor, among other things.

Next you must decide where in memory these 63 bytes should be placed as per our discussion in Section 3.3.2. There are relatively few choice available using the power-up

configuration. There is space for only four different sprite definitions.

If you want to use more, you can do either of the following:

a) move the start of BASIC (normally at $0801 = 2049). You can do this by POKEing the new address into locations $2B/$2C (43/44). Remember to do this before loading or saving a BASIC program;

b) move the video RAM to another location. You can then use the original video RAM storage at $400-$7FF. You can move video RAM by POKEing memory location 648 with the high byte of the new video RAM starting address. Normally memory location 648 contains $04. The new contents of memory location 648 can be calculated with: INT(video RAM/ starting address. To move the video RAM to $0800 = 2048, for example, (which requires that you also move the start of BASIC as well) then enter:

```
POKE 648, INT(2048/256)
```

After this POKE a <SHIFT><CLR/HOME> or a PRINT CHR$(147) must be used to clear the screen.

In most cases, four different sprite definitions will suffice because multiple sprites may use the same definition, as you will soon see.

In order that the VIC be able to find your 63-byte sprite definition, you must divide the 16K addressing range of the VIC into 256 blocks of 64 bytes each. These blocks are numbered from 0-255. After power-up these blocks have the following starting addresses (if the VIC address range

is moved by changing bits 0/1 of register 0 of CIA 2, the new base address must naturally be added to these values):

Block	Start address
0	$0000 - 0
1	$0040 - 64
2	$0080 - 128
3	$00C0 - 192
4	$0100 - 256
etc.	
255	$3FC0 - 16320

One sprite definition may be placed in each of these blocks. The last byte of each block has no significance because only 63 are required for each sprite. Only the following blocks may be used under power-up conditions:

Block	Address range
11	$02C0-$02FE (704- 766)
13	$0340-$037E (832- 894)
14	$0380-$03BE (896- 958)
15	$03C0-$03FE (960-1022)

The last three areas are located within the cassette buffer and will be overwritten if the datasette is used. You can also store sprites beginning at $2000 (8192), block 128 (be careful with large BASIC programs!). But because the memory at $1000-$1FFF has a special status (see Section 3.3.2.2) it is not usable. Other limitations apply if the 16K address range is moved, of course.

In order to tell the VIC in which block it can find the sprite definition, you must place this block number in one

of the last 8 bytes of the video RAM (see section 3.1). Under the power-up configuration, these lie at $07F8-$07FF (2040-2047).

If you calculate how many bytes are actually required for storing the text-screen contents, you find that 40x25 = 1000 bytes are necessary. But the video RAM takes up 1K, or 1024 bytes. The remaining 24 bytes are normally unused and can be used freely, except for the last 8 bytes. These are required for the sprite pointers. Each byte is assigned in the following manner (the addresses shown apply to the position of the video RAM after power-up and moving the video RAM naturally moves these as well):

Register :	$07F8	07F9	07FA	07FB	07FC	07FD	07FE	07FF
	2040	2041	2042	2043	2044	2045	2046	2047
Sprite # :	0	1	2	3	4	5	6	7

An example: Assume you have placed a sprite definition in the range at $03C0-$03FE (960-1022 which is block 15). Now you want sprite number 2 and sprite number 6 to appear as you have defined them in block 15. In this case, you would POKE the value 15 into the corresponding registers as a pointer to this block:

```
POKE 2040+2, 15 : POKE 2040+6, 15
```

You can see that this allows several sprites to share the same definition by simply setting their pointers to the same block.

If you have defined another sprite in block 14 and want sprite 2 to look like it, for example, then

```
POKE 2040+2,14
```

will change the definition (see Section 4.3).

Now we want these two sprites to appear on the screen. First we must turn them on. The VIC register 24, "sprite on/off," performs this function. Each of the 8 bits in this register is assigned to a sprite:

Bit:	b7	b6	b5	b4	b3	b2	b1	b0
Value:	128	64	32	16	8	4	2	1
Sprite:	s7	s6	s5	s4	s3	s2	s1	s0

In our case we must enter the following for sprites 2 and 6:

```
POKE 53248+24, 64 OR 4    or:
POKE 53248+24, 64 + 4
```

The sprites are still not visible, however, because they must be first moved onto the screen. You will learn more about programming sprites in Chapter 4.

3.5.4 Additional sprite properties

With simply defining, sprites, selecting their color, and turning them on, we have not yet finished covering all of the possibilities which sprites offer us.

3.5.4.1 Positioning

The first and most important property is the screen position of the sprites. You can determine where on the video display your figures will be displayed. This is esp-

ecially important because it makes movement, animation, and other effects quite easy, as you will see in Chapter 4.

For positioning, the screen is divided into x and y coordinates which we have already discussed in for high-resolution graphics. The following should be noted:

The sprite coordinates always represent the position of the lower left corner of the sprite.

The sprite movement has a resolution of 512x256 points, considerably more than is representable on the screen. The raster or the size of the point is the same size as that for high-resolution graphics. This makes it possible for sprites to travel beyond the edge of the screen--the sprite will move smoothly over the border, disappearing bit by bit (see Chapter 4).

The zero-point of the sprite coordinates lies far outside the text or graphics window in the upper left corner. The first visible point of this coordinate system and therefore the zero-point of the normal graphics has the coordinates x=20 and y=30. This is the first coordinate at which the sprite can be seen in full. In order to convert from sprite coordinates to graphics coordinates, you must subtract 20 from the x-coordinate and 30 from the y (add these values when converting from graphics to sprite coordinates). At x=320+20=340 and y=200+30=230, the sprite is no longer visible.

The first 17 (0-16) registers of the VIC are used to inform it where the sprites are to appear on the screen. The 17th (number 16) register is different from the rest and will be discussed a bit later. The 16 memory addresses are assigned to the sprites in pairs. The first element of this register pair gives the x-coordinate and the second the y-coordinate of the corresponding sprite:

Sprite :	s0	s1	s2	s3	s4	s5	s6	s7
x-c. reg.:	0	2	4	6	8	10	12	14
y-c. reg.:	1	3	5	7	9	11	13	15

For example, if we want to set sprite 6 at the coordinates x=100, y=150, we need only enter:

```
POKE 53248 + 2*6     , 100
POKE 53248 + 2*6 + 1, 150
```

and the previously defined sprite will appear at this spot. As you may have already noticed, we can only access 256 of the 512 points of the sprite-movement resolution (a byte can hold a maximum of 256 values). For this reason, another register must be used to hold the highest bit (msb = most significant bit) of the x-coordinate, register 16. In this byte, each sprite is again assigned to one bit holding the desired information:

Bit:	b7	b6	b5	b4	b3	b2	b1	b0
Value:	128	64	32	16	8	4	2	1
Sprite:	s7	s6	s5	s4	s3	s2	s1	s0

If we want to move our sprite beyond x-coordinate 256, we simply set the appropriate bit in this register. This has the effect of adding 256 to the contents of the regular x-coordinate register.

3.5.4.2 Enlargement

Our sprite is looking pretty good now--we would be quite contented--but our computer is not yet. It offers you still more features and possibilities. The registers of the video controller include two previously unmentioned addresses. These are registers 23 and 29. With the help of these two bytes you can increase the size of your sprite. The first of the two mentioned is for enlargement in the y-direction and the second for enlargement in the x-direction. In both cases the enlargement factor is 2, that is, each point is twice as wide or twice as tall. You can choose to enlarge a sprite in either direction or in both directions together (enlarging in both the x and y directions). You are already familiar with the organization of the registers, in which each sprite is assigned a bit. If the corresponding bit is reset, the sprite is not expanded, otherwise if it is set, then the sprite is expanded in that direction. The relationship can be represented as follows:

Expansion	Expansion factor	Point matrix
none	1x1	24x21
x-direction	2x1	48x21
y-direction	1x2	24x42
x/y direction	2x2	48x42

The term "point matrix" represents the size of the matrix which appears on the screen (it is identical for multi-color sprites), the maximum number of points which the sprite will cover. It should be noted that an expanded sprite will no longer disappear at the left or upper borders (or both), even if the sprite coordinates are zero.

3.5.4.3 Priority

What happens when two sprites or a sprite and a letter, for example, overlap? Do they cover each other, and if so, what covers what? This will be clarified in this section.

a) Sprite-sprite overlapping:

In this case it is quite simple: The individual sprites are assigned numbers from 0-7. If two sprites overlap, the sprite with the lower number will appear "over" the other, covering it. This means that a sprite with number 0 will overlap sprite number 5, for example, wherever they overlap (sprite 0 will cover only sprite 5 where sprite 0 is not transparent [see sprite definition]).

b) Sprite-background character overlapping:

When a sprite overlaps a background character, things become a bit more complicated, but also more interesting. First a clarification of a term: In the following discussion, a background character is taken to mean any set point, whether letter, special character, or graphics--anything, other than a sprite, that appears on the screen.

We can choose whether a sprite will be covered by this background character, the various objects appearing "in front" of the sprite, or whether the sprite will cover the character, appearing in front of it. An additional register exists in the VIC for this function, register 27. The construction is familiar--see register 16, 21, 23, 28 or 29 for details. A bit is again assigned to each sprite. If a bit is reset, which is the case after power-up, the sprite appears to cover the other

characters, whereas if it is set, all of the background characters appear in front of the sprite.

With the help of these valuable properties it is possible to represent 3-dimensional graphics or motion. More will be said about this in Chapter 4.

3.5.4.4 Collisions

An invaluable capability, especially for games, is determining when collisions or contacts between sprites and between sprites and a background character take place. This is made very easy through various other VIC registers. Memory locations 30 and 31 of the VIC are use for this purpose.

The first of these two registers records the overlapping of two sprites. Each bit of this register pertains to one of the eight sprites (see above). If two sprites touch (overlap), the two corresponding bits in the register will be set. If sprites 2 and 6 collide, the contents of this register will be %0100 0100. These contents will remain until they are cleared by the programmer with

```
POKE 53248+30,0
```

At the same time these two bits are set, a third is also set. This is bit 2 of VIC register 25 (IRR), which we will not discuss until section 3.7. We will say this much: If register 26 permits it, this bit can result in an IRQ (interrupt request).

The second register, number 31, is also of interest to us here since it records collisions between sprites and

background characters. If such an event occurs, the bit assigned to the colliding sprite will be set (as for register 30). If, for example, sprite 2 encounters a background point on the screen, the contents of register 31 read: %0000 0100. After reading this register, it too must be erased as described above. Also, this time bit 1 of register 25 is set, causing an interrupt if permitted.

3.6 Text/Character set management

3.6.1 Text page organization

The computer places all of the characters in the video RAM so that it knows where all the output is to be placed on the screen (it must recreate this picture of your screen every 1/20th second [see section 3.7]). This occupies an area of about 1K (in actuality only 40x25=1000 bytes) and normally extends from the memory location $0400 (1024) up to $07FF (2047), although it can be moved, as explained already in section 3.3.2. In the high-resolution and multi-color graphics, this area is used to store color information.

The individual characters are assigned certain codes and these codes are placed in the video RAM at the memory location corresponding to the place on the video screen where the character is to be displayed. You are already familiar with assignment of codes from the ASCII coding. The screen codes are formed from a different system however. While the CBM ASCII table which is found in the appendix of your user's guide many times assigns different values to the same characters and control codes which display no character on the screen, the screen codes are assigned clearly and without breaks. In addition to normal characters, the reverse characters must be differentiated--how else is the computer to know that a character is to be displayed in reverse except by using a different code? The keyboard accomplishes this using two control codes (<RVS ON> and <RVS OFF> = CHR$(18) and CHR$(146)). You can find a table of the screen codes in the appendix. If you add 128 to the value found there, you get the same character in inverted form.

3.6.1.1 Normal text

In addition to the actual character, the color of that character must also be stored, because theoretically each character can have a different color. An additional color RAM exists for this purpose, having the same size as the video RAM, which contains the necessary information. This color RAM lies at $D800-$DFFF (55296-56295) and is also used in multi-color graphics for color storage. Each byte of this range determines the color of the corresponding character in the video RAM, whose construction is identical.

The background color of the text picture is, in contrast, determined by a single register in the VIC. This register (register 33) lies at memory location 53248+33 and can be changed with:

```
POKE 53248+33,0 : REM BACKGROUND COLOR = BLACK
```

for example.

3.6.1.2 Multi-color mode

See section 3.2 for information on the multi-color character mode.

3.6.1.3 Extended-color mode

In addition to the normal text mode, the '64 has an additional mode in which you can choose a different background color for each character (there are 4 background color registers, therefore there are 4 different colors at your disposal). This method of display is called extended-color mode.

As we said, you can choose a background color from one of 4 for each character. The colors can be changed by POKE-ing the appropriate value into the following registers:

background register 0: VIC register 33
background register 1: VIC register 34
background register 2: VIC register 35
background register 3: VIC register 36

As you can see, color register 0 corresponds to the normal register for background color. Naturally, you place the usual color codes 0-15 into these registers, a table of which can be found in the appendix.

In order to switch the extended-color mode on, you need only set bit 6 of VIC register 17 ($11).

```
POKE 53248+17, PEEK(53248+17) OR 4*16
```

It can be turned off again with

```
POKE 53248+17, PEEK(53248+17) AND 256 - 4*16
```

To select the color of an individual character in extended color mode, you must know the following:

The upper 2 bits of each byte of the video RAM determine the color of that character. Since these bits are normally used for coding the various characters, you can only use 64 characters in the extended-color mode (ECM).

All of the graphics and upper case characters in the upper/lower case mode are unavailable. If you try to display one of these, one of the permitted characters will appear on the screen with a different background color. The choice of characters which can be displayed and which are left out can

be best determined from the table of screen codes in the Section 6.4. The upper two bits determine the extended character color as follows:

msbs	Screen codes	Background color
00	000-063/$00-$3F	BC 0
01	064-127/$40-$7F	BC 1
10	128-191/$80-$BF	BC 2
11	192-255/$C0-$FF	BC 3

The term msb stands for the two upper bits of the video RAM, bits 6 and 7 (msb = most significant bit, MSB = Most Significant Byte). Compare this table with the screen code table in the Section 6.4.

For example, suppose you want to display a "B" on the screen with background color yellow. To do this you must POKE the value 7 (for yellow) into the desired background color register (1 in this case) and POKE the screen code for for "B" (2 in this case) + 64 (to set bit 6) to select color register 1 into the appropriate screen memory location. Alternatively you can use a PRINT statement which outputs the character <SHIFT><B>, CHR$(98) on the screen. In a program, this would look like this:

```
10 V = 53248 : REM VIC REG BASE ADDRESS
20 POKE V+17, PEEK(V+17) OR 4*16 : REM TURN ON ECM
30 POKE V+34, 7 : REM BACKGROUND COLOR 1 = YELLOW
40 POKE 1024, 2+64 : REM CHARACTER IN UPPER LEFT CORNER
```

or:

```
10 V = 53248 : REM VIC REG BASE ADDRESS
20 POKE V+17, PEEK(V+17) OR 4*16 : REM TURN ON ECM
```

```
30 POKE V+34, 7 : REM BACKGROUND COLOR 1 = YELLOW
40 PRINT CHR$(98) : REM CHARACTER AT CURSOR POSITION
```

After running either of these programs you will still be in the extended-color mode and can do a bit of experimenting.

3.6.2 Character set organization

Besides the unusually rich graphics variations which the Commodore 64 offers, it has additional features. One of these capabilities is the ability to change the character set. This is the basis for almost all game graphics and is one of the reasons (along with sprites) that games can be developed so quickly and yet with such good graphics and effects.

First a definition: A character set is the entire range of characters (letters and graphics characters) which can be displayed in the text mode by pressing a key or a combination of keys (such as <SHIFT> and <C=> [Commodore key]).

The pattern and appearance of these characters must be known to the computer and therefore stored somewhere. At the same time, the computer must have information telling it, for example, that it is supposed to display a "B" when you press the letter B on the keyboard. All of this information is stored in memory within the computer.

On the Commodore 64, this information is placed in memory, but this location is not fixed as explained in section 3.3.1. In addition, the '64 has the ability to change the memory address from which it will read the patterns of the individual characters (see section 3.3.2). Therefore you have the option of telling the Commodore 64

that it should now retrieve the character patterns from RAM at say the memory range beginning at $2000 (8192). We can change this area of memory ourselves.

If we had previously copied the original character set from ROM to this area, we will not notice any difference because all of the information is identical. If we change part of this memory area, however, we thereby change the pattern of certain characters as well.

In order to create our own characters, we must know how the patterns of the characters are arranged in memory.

The computer has a total of 4 character sets, each with 128 characters. Only two sets may be displayed on the screen at any one time. These four character sets are identified by the following:

```
Set  I/1 - normal  - upper case/graphics characters
Set  I/2 - reverse - upper case/graphics characters
Set II/1 - normal  - upper/lower case
Set II/2 - reverse - upper/lower case
```

You can switch between sets I and II by simultaneously pressing the keys <C=> and <SHIFT> by hand. The ASCII values 14 and 142 (note: 142 = 14+128) are used from within a program, that is you can switch to set II with

```
PRINT CHR$(14);
```

and switch to set I with

```
PRINT CHR$(142);
```

With

```
PRINT CHR$(8);
```

you can prevent the possibility of switching from the keyboard. To re-enable this ability, type:

```
PRINT CHR$(9);
```

See also CBM 64 User's Guide, pages 135-137.

Switching between sets 1 and 2 is accomplished through the use of <RVS ON> and <RVS OFF>.

Each of these 4 character sets must naturally be stored separately in the character set memory. You can therefore change 128x4 = 512 characters (as mentioned before, only 256 different characters may appear on the screen at any one time).

Each character on the screen consists of a matrix of 8x8 points. Correspondingly, these 8x8 = 64 points must be represented in the character set storage. This is done by representing each point of the character on the screen with a bit in memory, similar to the way high-resolution graphics is stored. A character pattern is composed of total of 8 bytes, each of 8 bits. Each byte represents one of 8 lines of the character. A set bit means a set point on the screen. We can imagine the pattern of a character as follows:

Bit	7	6	5	4	3	2	1	0
Byte 0	.	.	.	.	.	.	.	.
Byte 1	.	.	.	.	.	.	.	.
Byte 2	.	.	.	.	.	.	.	.
Byte 3	.	.	.	.	.	.	.	.
Byte 4	.	.	.	.	.	.	.	.
Byte 5	.	.	.	.	.	.	.	.
Byte 6	.	.	.	.	.	.	.	.
Byte 7	.	.	.	.	.	.	.	.

A full character set is composed of a total of 512 sequential definitions of 8 bytes each. It therefore requires a memory area of 4K (= 4096 bytes). This normally lies in ROM at $D000 - $DFFF (decimal: 53248 - 57344). (This memory area cannot be read directly from BASIC, however.)

As a demonstration we'll illustrate how a capital B is stored in the character set memory:

Bit	7	6	5	4	3	2	1	0
Byte 0	.	*	*	*	*	.	.	.
Byte 1	.	*	*	.	.	*	*	.
Byte 2	.	*	*	.	.	*	*	.
Byte 3	.	*	*	*	*	.	.	.
Byte 4	.	*	*	.	.	*	*	.
Byte 5	.	*	*	.	.	*	*	.
Byte 6	.	*	*	*	*	.	.	.
Byte 7	.	.	.	.	.	.	.	.

We get the following 8 bytes:

```
Byte 0: 0111 1000 = $78 = 120
Byte 1: 0110 0110 = $66 = 102
Byte 2: 0110 0110 = $66 = 102
Byte 3: 0111 1000 = $78 = 120
Byte 4: 0110 0110 = $66 = 102
Byte 5: 0110 0110 = $66 = 102
Byte 6: 0111 1000 = $78 = 120
Byte 7: 0000 0000 = $00 = 000
```

These eight values are found at the place in the character set storage which is reserved for the upper-case normal B.

For multi-color characters, 2 bits denote a double-width point of the character, whereby the resolution of the matrix is limited to 4x8 points per character. Because this procedure is very similar to the definition of multi-color sprites and graphics, we will only give the point-bit relationship here in a diagram:

Bit	7 6	5 4	3 2	1 0
Byte 0	<->	<->	<->	<->
Byte 1	<->	<->	<->	<->
Byte 2	<->	<->	<->	<->
Byte 3	<->	<->	<->	<->
Byte 4	<->	<->	<->	<->
Byte 5	<->	<->	<->	<->
Byte 6	<->	<->	<->	<->
Byte 7	<->	<->	<->	<->

Next we must determine the memory location of the character pattern to be displayed. To display a particular character on the screen, the screen code is placed into

video RAM (either by POKEing or by PRINT, which converts the literal to the screen code). It is this code that determines which character pattern is displayed. Because each character requires 8 bytes, we take the value of the screen code times 8 and add the address at which the character set begins. For the base addresses, remember that character sets I and II must be distinguished. A lower case "b" in the upper/lower case mode with screen code 2 lies 2K away from the capital "B" in the upper case/graphics mode with the code 2 (not to be confused with the upper case B of the upper/lower case mode). In the original character set, these base addresses are:

- upper case/graphics characters: $D000 (53248)
- upper/lower case : $D800 (55296)

The formula given above is:

address = base address + 8 * screen code

For the character B in the normal character storage this would be:

address = $D000 + 8 * 2 = $D010 = 53256

In Chapter 4 you will learn in detail how you can best make these changes to your own character set (among other ways, with a very easy-to-use character editor) and how you can make use of what you have learned.

3.7 IRQ Capabilities

One very powerful but frequently misunderstood capability of the '64 is interrupt management. An interrupt is a controlled interruption of a program, usually by some event occuring outside of the program. If an interrupt occurs, the 6510 stops what it is doing but remembers where is left off and begins another programming task at a predetermined place in memory where it executes something called an interrupt service routine. The address to which it jumps is addressed indirectly. At the end of this service routine the '64 returns to the place where it left off before and continues as though nothing had happened--until the next interruption.

Have no fear, you can read on in peace even if your machine language skills are a little rusty. All of the interrupt capabilities presented here will function even if you don't understand how why they work. If you aren't acquainted with interrupts, you should read through this section in order to get an overview of the applications and methods. At the conclusion we'll explain the interrupt capabilities which exist from within BASIC.

An interrupt is a planned and foreseen interruption (not a crash). Interrupts are fundamentally possible in BASIC, although the CPU interrupts discussed here are hardware-generated interrupts controlled by software. There are four different interrupts for the microprocessor used in your device:

- reset
- NMI (Non-Maskable Interrupt)
- BRK (BReaK)
- IRQ (Interrupt ReQuest)

a) Reset:

This first interrupt, which cannot be suppressed through software, is generated after power-up and initializes the computer. At the end of this procedure, the following message is displayed:

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
```

b) NMI:

The Non-Maskable Interrupt (that is, software cannot prevent the interrupt from being executed) is generated when you press the <RESTORE> key. It is also required to service the RS 232 interface. You can find or change the indirect jump address of the NMI in memory locations $318/$319 (792/793).

c) BRK:

This is the software interrupt which can be used in assembly language programs. It is generated when the processor encounters the BRK code $00 (indirect jump to the address in memory locations $316/$317 [790/791]).

d) IRQ:

Here it is at last! All of the above interrupts are of no further interest to us and have been mentioned only for the sake of completeness. The really interesting interrupt as far as we are concerned is this maskable interrupt. Maskable means that software can determine whether the interrupt will be executed or not. You can turn it off whenever you feel like it. Two assembly language commands exist for this purpose: SEI (SEt Inter-

rupt flag - disables interrupts) and CLI (CLear Interrupt flag - enables interrupts). At the same time, you can choose which sources may trigger an interrupt and which may not. For example, the timer of the CIA can be made to generate an interrupt which is used by the BASIC interpreter and kernal, interrupting the normal program execution every 1/60th of a second and jumps to the ROM IRQ routine (indirect address in $314/$315, decimal: 788/789). Here the cursor is flashed, the internal clock (TI$) is controlled, and the keyboard is polled (read). We can also permit interrupts from sources other than the timer. The most interesting to us are:

- raster line
- light pen
- sprite-sprite collision
- sprite-background collision

If we change the indirect address of the kernal IRQ routine to our own interrupt service routine, we can make use of these interrupt capabilities. The appropriate section in Chapter 4 illustrates how this can be done within a program. The advantage of using interrupts for these applications is that the event is reported immediately without waiting for the next reading of the conditions (besides using interrupts, the event could also be determined by a simple polling). This has special significance for the raster line IRQ, because the procedure must run very quickly. But now on to more details.

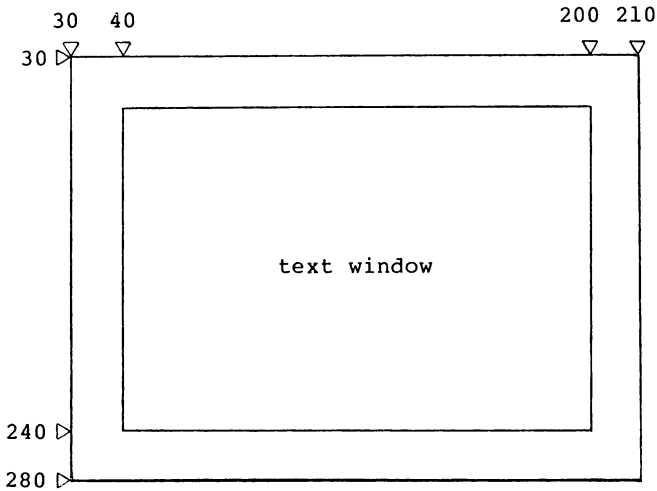
3.7.1 Screen raster lines

One of the least-understood but most interesting capabilities of your computer is the raster line interrupt. With this you can achieve a great number of effects: multiple background colors, more that 8 simultaneous sprites, mixed graphics and text, and so on, and so on. But before we go into further detail about this capability in Chapter 4, we must first clarify what we mean by a raster line.

We must understand how a picture is composed on the television or monitor. As you may know, the screen consists of a cathode plate together with a phosphorescent coating which glows when struck by a beam of electrons from an electron gun. This electron gun moves from line to line to each individual point and either illuminates it or skips over it. This occurs with great speed so that the each picture is recreated 20 times per second. This creates the impression of a moving picture. This electron gun is controlled by a complicated apparatus. The information as to whether a point is supposed to be illuminated or left dark must come from another source. With a normal television this is sent through the air and received by the antenna from the television station transmitter. In our case, the computer must create this signal. It must go through, row by row and point by point, and send an "on/off" signal. In the case of the Commodore 64, this task is assumed by the video controller (VIC).

Normally, this is all controlled internally without programmer involvement. It's different for the '64. Here the software has the ability to determine which raster line the VIC is creating. This can be done, among other ways, by reading VIC register 18.

Since the raster lines must be extended to the screen borders as well as the text or graphics window and the beam deflection extends somewhat beyond the screen, the coordinates for raster lines are somewhat different than for graphics or sprites. To illustrate what will be said in the following, consult this sketch of the screen which depicts this window.



The electron beam starts in the top row. This does not possess the number 0 or 1 as one might expect, but the number 30 ($1E) (the actual border value can vary from TV [monitor] to TV). It ends (lower edge) at about number 280 ($118). The actual screen window which can normally be used has an upper edge raster line value of about 40, while the lower is at number 240.

As you can see, the VIC divides the text window (exactly in accordance with the point resolution) into 200 lines (rows). You will see in the section "Joystick" that this is not so simple in the case of column resolution.

As we said, we can influence and be made aware of these events. What does this mean?

First, we can determine the number of the raster line on which the VIC is currently working from register 18. But since a register can only contain values from 0 to 255 and the VIC sends at least 280 rows, the missing bit (bit 9) comes from register 17. As you can gather from the table in section 3.1, bit 7 of this register 17 represents the most-significant bit of the raster line number. We can determine the exact raster line simply by reading these registers. Because a picture is renewed 20 times per second and at least 280 rows must be sent per picture, a row must be constructed in about 1 sec./(20*280) = 0.00018 seconds, 0.18 milliseconds or 180 microseconds (a microsecond is a millionth of a second, a millisecond is a thousandth of a second), meaning 5600 rows are constructed per second. When one recalls that a row consists of a great number of points, one can imagine how much time is left for each point (approximately 0.89 microseconds).

Such speed on the part of the video controller makes it impractical to read the raster line even from assembly language in order to control a specific row, since a machine language instruction takes at least one 500,000th of a second (2 microseconds).

This is why the VIC's interrupt capabilities are so useful. You can simply instruct the VIC to generate an IRQ (as described above) when it reaches a certain line on the screen (it does not make sense to try to address a single point because of the incredibly short point-generation time). To do this, write the desired line at which an interrupt is to be generated into the same register 18. Here too, bit 7 of register 17 serves as the high bit. We need only

inform the computer that it should generate an interrupt when it reaches that line. This is done using registers 25 and 26 ($19/$1A). The first is called the Interrupt Request Register (IRR). This register indicates the source of an interrupt generated by the VIC. The following is a description of the IRR:

```
Bit 0 = 1: raster line interrupt
Bit 1 = 1: interrupt by sprite/background collis.
Bit 2 = 1: interrupt by sprite/sprite collis.
Bit 3 = 1: interrupt by lightpen
Bit 4-6  : unused
Bit 7 = 1: interrupt has one of the four sources
```

By reading bit 7, you can determine if one of the other four bits is set. If this bit is set, then an interrupt was caused by one of the 4 sources on the VIC. After each read, this register must be reset or another interrupt will be generated after the interrupt service routine is done, and so on--"crash!" The register can be reset by reading it and then rewriting the same value back into the register.

In spite of this, the VIC still does not know that it is supposed to generate an interrupt since we have not made use of register 26 yet. Here we have the same assignment of the individual bits as we found in register 25 (except for bit 7). A set bit means that the corresponding event is permitted to trigger an interrupt. For example, if we want the VIC to interrupt the main processor and jump to the IRQ routine whose address is at $314/$315 (=788/789) when row 100 is reached, we first write 100 into register 18 (msb = 0!), clear register 25 by reading its contents and then writing these back to it directly, and setting bit 0 of register 26 to 1. How this is all brought together is discussed in the corresponding section in Chapter 4.

3.7.2 Light pen

Before you read these paragraphs, you should have at least read section 3.7.1 concerning the construction of the picture on the screen. The following explanations will build on this knowledge.

Your Commodore 64 has various capabilities for using control devices as peripherals. There are two control ports on the right side of the computer which are used for this purpose. These are the sockets for connecting joysticks, paddles, a light pen, or some home-brew device (thermometer, hygrometer, etc.). The pin layout is described in your CBM 64 User's Guide on page 141. You do not have to know this in order to connect and a use a joystick. All that is important for us here is that the procedure for the light pen is identical to that for the fire button of a joystick inserted in port 1. You can trigger an interrupt with the fire button on port 1, a property which can be utilized in more than just games. You can connect devices which interrupt the computer, a capability which can have valuable consequences!

One of the most interesting capabilities is the light pen.

A light pen is a hand-held "pen" which serves to determine and input a point on the screen and provides direct contact between the user and the television (monitor). It is possible to input a screen position simply by moving the light pen to that position on the screen.

How does this work? You point the light pen to a point on the screen. It does not matter whether this point lies inside or outside the text window. The computer has the capability to identify the coordinates of this point. From within a program you can, for example, determine whether or

not the light pen is positioned over a certain object (letter or graphics). You can also plot a point at the screen location, thereby drawing by hand with the light pen! Another idea is to use the light pen for easy cursor positioning. There are many possibilities for its use.

Now we move to the technical side. How does the computer determine where on the screen the light pen is positioned? You know that a picture on the screen is composed of many small dots which are illuminated every 1/20th of a second by an electron beam. The light pen, when pointed at the screen, can detect this short illumination and send an impulse to the computer. (Note that when using the light pen you must not select a black background and not turn the brightness control on the television or monitor down too low). This impulse reaches the VIC which immediately stores the current raster line and raster column in two registers as the x and y coordinates of the point. These two registers are VIC registers 19 and 20 (x and y, respectively). A program can read the two values and perform the desired operation (such as plotting a point at that location).

First something must be said about the coordinate system. You are already familiar with the y-division, the division of raster lines, from section 3.7.1. The x-division is somewhat more complicated. Here there are half as many addressable raster points as we are familiar with from graphics, meaning that a given raster point stands for two actual graphics points. It yields the following edge values: The left border of the screen has the edge value 30, the right is column number 210. The regular screen window lies at about 40 left and 200 right, whereby we get 160 = 320/2 raster points in the text window. Assume that we have two values for the raster coordinates from registers 19/20 and want to draw a point at this exact spot. We must then con-

vert the raster coordinates to graphics coordinates. This can be done, based on the above discussion, with the following formulas:

```
x = (xr-40)*2
y = yr-40
```

whereby x and y are the graphics coordinates and xr, yr are the raster coordinates. Now we can set a point at the calculated spot.

This is the simplest use of the light pen and is also programmable from BASIC. Here too, however, your computer offers the opportunity to work with interrupts:

If the light pen sends an impulse, the 3rd bit of VIC register 25 will be set. If we have previously set the 3rd bit in register 26, an interrupt can be generated. Your program will be interrupted and your interrupt routine called. Programming examples and more information will be given in Chapter 4.

3.7.3 Sprite collisions

If you have read section 3.5.4.4, you already know that the collision of two sprites with each other or a sprite and the background can be determined by reading and analyzing the contents of the VIC registers 30 and 31. As you know, each sprite is here assigned to a bit and the bits are set corresponding to the sprites which collided. There is another possibility for registering sprite collisions from assembly language. As you may have guessed by now, this also uses interrupts. You can instruct the VIC (again by using

registers 25/26) to generate an interrupt when a collision occurs. We must distinguish between a collision between sprites and a collision between the background and a sprite. A bit is reserved in register 25 (IRR) for each of these cases. Bit 3 is set whenever a collision between two sprites occurs. In this case, the sprites involved are recorded in register 30. Correspondingly, bit 2 is set whenever a collision between a sprite and a background character occurs. An interrupt is not be generated until you set the corresponding bit in register 26. Do not forget to set the IRQ address and reset the IRR after each interrupt.

Chapter 4
Fundamental Graphics Programming

Now that we have heard enough theory about graphics capabilities, it is time to concern ourselves with putting these concepts to use. Otherwise, what good is it if we don't know how to use sprites or high-resolution graphics? What we lack is practical experience. For this reason, there is a section in this chapter which corresponds to each section in Chapter 3 and deals with programming and applications of the actual features.

Programs are written in BASIC whenever possible and the numerous machine language routines all have BASIC loaders in the event that you do not have an assembler. All programs are documented with REMs. The important lines are also described in the text. The BASIC routines for figures and applications are given only to clarify the matter at hand; it must be said in all honesty that the corresponding assembly language routines perform the desired function much faster. For this reason, a knowledge of machine language is very useful. If you have already mastered BASIC, this step is really not much farther. Try it!

4.1 Text and graphics on the low-res screen

The first and simplest way to put graphics on your screen is by using the graphics characters found on the front of the keys. These allow you to create wonderful and many-colored graphics on the screen if you use your imagination (always a necessary ingredient for creating graphics). The graphics characters can be used by various methods:

a) Control through the keyboard:

The characters which are shown on the left side of the key are normally displayed on the screen by simultaneously pressing the appropriate key together with the <SHIFT> key. Those which appear on the right are selected with the pressed <C=> key. In addition to these normal characters, you can also create reverse characters by first pressing the keys <CTRL> and <RVS ON> at the same time or by entering the command PRINT CHR$(18). To turn off the reverse mode, simply press <CTRL> <RVS OFF> or type PRINT CHR$(146).

Colors:

The 16 various character colors which can be displayed on the screen are selectable from the keyboard (a comprehensive table of the colors and their various assignments can be found in section 6.3). To display one of the first 8 colors, you press the <CTRL> key together with one of the keys <1-8> (an abbreviation of the color can be found on the front of each of these keys). If, on the other hand, you want to give the following characters one of the last 8 colors (colors 8-15), simply press the <C=> key together with one of the 8 color keys.

b) Control through the PRINT statement:

Each character has a certain ASCII code. The ASCII code of a character can be determined through the BASIC command ASC in the following manner (in this case for the letter "A"):

```
PRINT ASC("A")        or:
Z$ = "A" : PRINT ASC(Z$)
```

The computer prints 65. If you know the ASCII code of a character, you can see the character with the command CHR$ (again for the letter "A"):

```
PRINT CHR$(65)        or:
C = 65 : PRINT CHR$(C)
```

This displays an A on the screen. The advantage of this coding is the fact that characters are calculable, that is, a character can be determined by the result of an algorithm. At the same time, comparing characters is also much simpler.

Colors:

Even the various colors (in addition to all of the other control functions such as <CLR/HOME>, etc.) have ASCII codes. Unfortunately, only the codes for the first 8 colors are given in the CBM 64 User's Guide; here is a complete listing:

ASCII	Color	ASCII	Color
144	black	129	orange
5	white	149	brown
28	red	150	light red
159	cyan	151	grey 1
156	purple	152	grey 2
30	green	153	light green
31	blue	154	light blue
158	yellow	155	grey 3

c) Control through POKE:

All characters which appear on the screen are stored in special memory: the video RAM (see section 3.6). Here a distinction must be made between normal and reverse characters. Control codes, which do not appear on the screen, do not need to be saved in video RAM. In memory, a different code, called the screen code is used. If you want POKE directly into this memory, you must use these codes. For example,

```
POKE 1024,1
```

puts an A in the upper left corner of the screen. This A cannot be seen yet (assuming a character has not been there before). It lacks the color which is stored in the color RAM and which can be set with

```
POKE 55296,5
```

for instance (the 5 represents the color code which is to be stored in the color RAM). The character will be green (color 5). This relationship was explained in Section 3.6.1.

On the following pages we will present three programs to you which all yield the same result but do it in different ways. They are presented without comment on which is the best method for the purpose at hand. They should show you how each of the possibilities of character representation can be realized.

```
100 REM **********************
110 REM **                    **
120 REM **  LOW-RES/PRINT     **
130 REM **                    **
140 REM **********************
150 REM
160 PRINT CHR$(147) : PRINT : PRINT : REM CLEAR SCREEN/BLANK
 LINES
170 PRINT"                                ": REM M,N
180 PRINT" ◥◠                              ": REM M,U,I,U,K
190 PRINT"  ◡◥                             ": REM J,K,M
200 PRINT"                  ◜              ": REM N,M,U,K
210 PRINT"                  ▂             ": REM P
220 PRINT"              ▂▂▂▙▙▂▂▂           ": REM P,P,P,L,L,P,P
230 PRINT"                 ◿  ◹           ": REM N,M,M
240 PRINT"               ◿    ▮          ": REM N,M,L
250 PRINT"              ◿     ▮          ": REM N,M,L
260 PRINT"             ┌──────┐             ": REM O,Y,Y,Y,Y,P,N
270 PRINT"             ▮ ┌┐ ┌┐ ▮             ": REM H,A,S,A,S,N,N
280 PRINT"             ▮ └┘ └┘ ▮      ┌┐ ◠     ": REM H,Z,X,Z,X,N,N
,A,S,U,I
290 PRINT"             ▮  ┌─┐  ▮     ◿┴┴◝ ◹    ": REM H,O,P,N,N,U,E
,E,K,J,I
300 PRINT"             ▮  ┌ ▮  ▮◿  ─┼○──○┼    ": REM H,O,N,N,N,-,E
,W,C,C,W,E
310 PRINT"             ──┌────  ┌──────────────": REM Y,Y,O,Y,Y,O
,Y,Y,...
320 PRINT"               └──────┘              ": REM L,P,P,P,P,P,@
```

```
100 REM ************************
110 REM **                      **
120 REM **  LOW-RES GRAPHICS    **
130 REM **        /CHR$         **
140 REM ************************
150 REM
160 PRINT CHR$(147) : PRINT : PRINT : REM CLEAR SCREEN /
    BLANK LINES
170 FOR X=1 TO 290 : REM LOAD 290 ASCII CODES
180 READ CH : REM READ DATA
190 PRINT CHR$(CH); : REM WRITE CHARACTER
200 NEXT X
210 REM
220 REM ******************
230 REM **  DATA LINES   **
240 REM ******************
250 REM
260 DATA  32,  32, 205,  32, 206,  13
270 REM
280 DATA  32, 205, 213, 201,  32,  32
290 DATA  32,  32,  32,  32,  32,  32
300 DATA  32,  32,  32, 213, 203,  13
310 REM
320 DATA  32,  32, 202, 203, 205,  13
330 REM
340 DATA  32, 206,  32, 205,  32,  32
350 DATA  32,  32,  32,  32,  32,  32
360 DATA  32, 213, 203,  13
370 REM
380 DATA  32,  32,  32,  32,  32,  32
390 DATA  32,  32,  32,  32,  32,  32
400 DATA  32, 175,  13
410 REM
```

```
420 DATA  32,  32,  32,  32,  32,  32
430 DATA  32,  32,  32,  32, 175, 175
440 DATA 175, 204, 204, 175, 175,  13
450 REM
460 DATA  32,  32,  32,  32,  32,  32
470 DATA  32,  32,  32, 206, 205,  32
480 DATA  32,  32,  32,  32,  32, 205
490 DATA  13
500 REM
510 DATA  32,  32,  32,  32,  32,  32
520 DATA  32,  32, 206,  32,  32, 205
530 DATA  32,  32,  32,  32,  32, 182
540 DATA  13
550 REM
560 DATA  32,  32,  32,  32,  32,  32
570 DATA  32, 206,  32,  32,  32,  32
580 DATA 205,  32,  32,  32,  32, 182
590 DATA  13
600 REM
610 DATA  32,  32,  32,  32,  32,  32
620 DATA  32, 207, 183, 183, 183, 183
630 DATA 208,  32,  32,  32,  32, 206
640 DATA  13
650 REM
660 DATA  32,  32,  32,  32,  32,  32
670 DATA  32, 165, 176, 174, 176, 174
680 DATA 170,  32,  32,  32, 206,  13
690 REM
700 DATA  32,  32,  32,  32,  32,  32
710 DATA  32, 165, 173, 189, 173, 189
720 DATA 170,  32,  32, 206,  32,  32
730 DATA 176, 174, 213, 201,  13
740 REM
```

```
750 DATA  32,  32,  32,  32,  32,  32
760 DATA  32, 165,  32, 207, 208,  32
770 DATA 170,  32, 206,  32,  32, 213
780 DATA 177, 177, 203, 202, 201,  13
790 REM
800 DATA  32,  32,  32,  32,  32,  32
810 DATA  32, 165,  32, 207, 170,  32
820 DATA 170, 206,  32,  32,  45, 177
830 DATA 215, 195, 195, 215, 177,  13
840 REM
850 DATA  32,  32,  32,  32,  32,  32
860 DATA  32, 183, 183, 207, 183, 183
870 DATA 183,  32,  32,  32, 207, 183,
880 DATA 183, 183, 183, 183, 183, 183
890 DATA 183, 183,  13
900 REM
910 DATA  32,  32,  32,  32,  32,  32
920 DATA  32,  32,  32, 204, 175, 175
930 DATA 175, 175, 175, 175, 165,  13


100 REM ******************************
110 REM **                          **
120 REM **  LOW-RES GRAPHICS/POKE   **
130 REM **                          **
140 REM ******************************
150 REM
160 CO = 5 : REM COLOR = GREEN
170 PRINT CHR$(147) : REM CLEAR SCREEN
180 VR = 1024 + 2*40 : REM POKE START ADDRESS(VIDEO RAM)
190 CR = 55296 + 2*40 : REM POKE START ADDRESS(COLOR RAM)
200 FOR Y=0 TO 15 : REM 16 LINES
210 READ NC : REM GET NUMBER OF CHARACTERS IN LINE
```

```
220 VR = VR+40 : REM NEXT LINE (40 MEMORY LOCATIONS FARTHER)
230 CR = CR+40 : REM NEXT LINE (40 MEMORY LOCATIONS FARTHER)
240 FOR X=0 TO NC-1 : REM POKE CHARACTER NC
250 READ SC : REM READ SCREEN CODE
260 POKE VR+X, SC : REM AND WRITE IN VIDEO RAM
270 POKE CR+X, CO : REM POKE COLOR
280 NEXT X
290 NEXT Y
300 REM
310 REM **********************
320 REM **  SCREEN CODES  **
330 REM **********************
340 REM
350 DATA   5
360 DATA  32,  32,  77,  32,  78
370 REM
380 DATA  17
390 DATA  32,  77,  85,  73,  32,  32
400 DATA  32,  32,  32,  32,  32,  32
410 DATA  32,  32,  32,  85,  75
420 REM
430 DATA   5
440 DATA  32,  32,  74,  75,  77
450 REM
460 DATA  15
470 DATA  32,  78,  32,  77,  32,  32
480 DATA  32,  32,  32,  32,  32,  32
490 DATA  32,  85,  75
500 REM
510 DATA  14
520 DATA  32,  32,  32,  32,  32,  32
530 DATA  32,  32,  32,  32,  32,  32
540 DATA  32, 111
```

```
550 REM
560 DATA  17
570 DATA  32,  32,  32,  32,  32,  32
580 DATA  32,  32,  32,  32, 111, 111
590 DATA 111,  76,  76, 111, 111
600 REM
610 DATA  18
620 DATA  32,  32,  32,  32,  32,  32
630 DATA  32,  32,  32,  78,  77,  32
640 DATA  32,  32,  32,  32,  32,  77
650 REM
660 DATA  18
670 DATA  32,  32,  32,  32,  32,  32
680 DATA  32,  32,  78,  32,  32,  77
690 DATA  32,  32,  32,  32,  32, 118
700 REM
710 DATA  18
720 DATA  32,  32,  32,  32,  32,  32
730 DATA  32,  78,  32,  32,  32,  32
740 DATA  77,  32,  32,  32,  32, 118
750 REM
760 DATA  18
770 DATA  32,  32,  32,  32,  32,  32
780 DATA  32,  79, 119, 119, 119, 119
790 DATA  80,  32,  32,  32,  32,  78
800 REM
810 DATA  17
820 DATA  32,  32,  32,  32,  32,  32
830 DATA  32, 101, 112, 110, 112, 110
840 DATA 106,  32,  32,  32,  78
850 REM
860 DATA  22
870 DATA  32,  32,  32,  32,  32,  32
```

```
880 DATA  32, 101, 109, 125, 109, 125
890 DATA 106,  32,  32,  78,  32,  32
900 DATA 112, 110,  85,  73
910 REM
920 DATA  23
930 DATA  32,  32,  32,  32,  32,  32
940 DATA  32, 101,  32,  79,  80,  32
950 DATA 106,  32,  78,  32,  32,  85
960 DATA 113, 113,  75,  74,  73
970 REM
980 DATA  23
990 DATA  32,  32,  32,  32,  32,  32
1000 DATA  32, 101,  32,  79, 106,  32
1010 DATA 106,  78,  32,  32,  45, 113
1020 DATA  87,  67,  67,  87, 113
1030 REM
1040 DATA  26
1050 DATA  32,  32,  32,  32,  32,  32
1060 DATA  32, 119, 119,  79, 119, 119
1070 DATA 119,  32,  32,  32,  79, 119
1080 DATA 119, 119, 119, 119, 119, 119
1090 DATA 119, 119
1100 REM
1110 DATA  17
1120 DATA  32,  32,  32,  32,  32,  32
1130 DATA  32,  32,  32,  76, 111, 111
1140 DATA 111, 111, 111, 111, 101
```

The first of the three programs illustrates the creation of graphics by means of the PRINT statement, which is the shortest and in this case the most convenient. Here the picture is displayed line-by-line by PRINT statements after the screen is erased in line 160 with

```
PRINT CHR$(147)
```

Full use is made of the graphics characters. In the REM lines behind the PRINT statements are the keys on which the individual graphics characters are found (the spaces are naturally left out). You must press these keys together with <SHIFT> or <C=> (as described above) and the desired characters will appear on the screen.

In the second program the stored picture is created with PRINT CHR$ statements. They do not directly contain the individual characters, however, but create them using the

ASCII codes. The various ASCII codes are placed in DATA statements. As you probably already know, elements are separated by commas in DATA statements and can be read sequentially by the READ command. These values are then placed into memory (here CH). This occurs in our program in line 180. The variable CH contains the ASCII value of the next character. This assigned character is then PRINTed in line 190. The whole thing is enclosed in a FOR/NEXT loop which runs a total of 290 times in order to read all 290 data. At the end of each screen line (which are divided in the DATA lines by REM statements) is the number 13. This is the ASCII code for <RETURN> and tells the computer to place the next character at the start of the next line down. As you can see, quite a lot of data statements are necessary, making this method quite ineffective for our purpose. A slight improvement can be made by placing a marker at the beginning of each line together with the number of spaces in lines where many spaces (ASCII = 32) are used.

The third example demonstrates the use of the POKE command to write a character directly into the video RAM. Here too the screen codes are stored in DATA lines. Besides simply writing the character into memory, we must also place the color for each character in the corresponding color RAM. The heart of the program consists of two nested FOR/NEXT loops. The outer loop (lines 200-290) increments (after the inner loop finishes) the number of the line which is to be filled with graphics characters. At the start of the inner loop a value is read into variable CT (line 210) which gives the number of characters in the screen line. This serves to determine the number of times that the inner loop is executed. Within this loop the individual characters are read with READ (line 250) and POKEd into the running address in

the video RAM (line 260). The value 5 for green is also written into the corresponding place in the color RAM (line 270).

What is the most efficient way of creating a graphics picture? There are many different methods and you must decide for yourself which will be the simplest for any given application. In general, however, creating a really nice picture with many details is a time-consuming process, especially when you use self-defined characters. Simple random pictures can often create interesting screen effects, as is demonstrated by the following program:

```
100 REM **********************
110 REM **                    **
120 REM **  RANDOM PICTURE    **
130 REM **                    **
140 REM **********************
150 REM
160 PRINT CHR$(147) : REM ERASE SCREEN
170 PRINT CHR$(RND(1)*3 + 177); : REM ONE OF 3 CHARACTERS
    ASCII=177/178/179
180 GOTO 170
```

Another possibility for creating pictures is by using a particular routine for positioning the cursor at a desired screen position. With the standard BASIC this is only possible to do within a line. The small routine at line 1000 of the following program permits access to the entire screen:

```
100 REM ************************
110 REM **                    **
120 REM **  SINE CURVE IN TEXT  **
130 REM **                    **
140 REM ************************
150 REM
160 PRINT CHR$(147) : REM CLEAR SCREEN
170 FOR X=0 TO 39
180 Y=13*SIN(X/3)+12 : REM FUNCTION
190 GOSUB 1000 : PRINT "*"; : REM CALCULATE POSITION
200 NEXT X : END
210 REM
220 REM POSITION CALCULATION
1000 PRINT CHR$(19);:IF Y>0 THEN FOR L=1 TO Y:PRINT:NEXT L
1010 PRINT TAB(X);: RETURN
```

This program draws a sine curve with asterisks (*). The subroutine in line 1000 is given in X and Y the parameters for the column (X) and row (Y) of the desired cursor position. After a <HOME> (PRINT CHR$(19);), carriage returns are sent until the desired line is reached. Then a TAB command is used to reach the proper column. This small but extremely effective routine makes graphics programming in the text mode much easier.

Another aid in creating pictures is to draw the pictures on the screen using just the cursor keys and the keyboard (graphics characters). For best results, leave the color out first. Enter a line number, a PRINT (which can be abbreviated to a question mark) and a quotation mark in front of each line on the screen, either with insert or by overtyping. The last quotation mark is not absolutely necessary if the line ends with this PRINT expression. This method does have a few pitfalls:

Space must be made for the line number, question mark, and quotation mark. If this space is occupied by graphics, the INST key must be used to make room. Furthermore, if the line is 40 characters long (occupying the entire screen width) part of it will be moved down to the next line, thereby moving all of the lower lines down. Care must be taken in the last screen line. If you should happen to move the cursor down below this line, the entire screen will be shifted up and the top line will be lost. Another problem is that the last column may not be used in any line. In this case a new screen line will be added, which leads to other problems (although it can be avoided by placing a closing quotation mark and semicolon at the end of the offending line).

In addition, no inverse characters may also be placed directly within a PRINT statement. If a picture is to contain such, they must be replaced by the following sequence (after the line has been enclosed in quotation marks, or after the leading quotation mark):

<RVS ON> <...> <RVS OFF>

By <...> we mean all of the characters which are to appear on the screen in reverse (although they will appear normal in the program).

Third, color cannot be placed directly within a PRINT statement. Inserting color and control codes must be done by editing the PRINT statement. One final comment: The control characters are only properly placed within a PRINT expression (and not immediately executed) if you first place a quotation mark (") in front of them. This is quite inconvenient during later insertions and is not absolutely necessary. You create a blank space with the <INST> key and then

replace this with a control character. If you have the string "AB" and you want to insert a control character between A and B, this can be done simply with <INST> <control character>.

As you can see, this method of creating pictures is not all that easy although it is quite acceptable for the beginner or casual graphics designer. If you want to create professional graphics with high detail and quality, you should write a small screen editor, a program which allows you to move a (self-defined) cursor to create a picture with all of the colors and other possibilities. Admittedly, this is not all that simple, although it is a rewarding project. Those who are interested should be sure to read section 4.4.

4.2 Programming Bit-mapped Graphics

The complex organization of high-resolution and multicolor graphics makes them difficult to use. Even placing a single point on the screen requires a good many calculations. Just keeping in mind all of the factors necessary for initializing the graphics screen requires a good working knowledge of the entire topic of graphics.

Creating simple figures such as lines or circles is quite difficult and requires a lot of mathematical knowledge. For this reason we present the various routines that provide the capabilities presented in Chapter 3. It is not necessary to understand every step of a program in order to make use of it. Those who don't know what sin or cos mean, for example, should skip certain sections of this discussion (such as he section on circle creation). In spite of this he should read the corresponding section in Chapter 3 (section 3.4) about the fundamentals of graphics. Such information can be valuable for those who are interested in modifying these routines for their own purposes.

It should be made clear that a BASIC routine, no matter how clear or understandable, does not have the speed of a corresponding machine language program. This means that many effects can only be created in machine language. If you look at how long a BASIC routine takes to draw a circle, you will agree. To alleviate this problem, we will present a small assembly language graphics package (together with BASIC loader) at the end of this chapter. The individual functions of the graphics package can be controlled from BASIC in the same manner as many graphics extension packages. If you want to program only in BASIC, we also include some tips in section 6.1 for optimizing your BASIC programs.

All of the programs assume that the bit-mapped graphics memory is located $2000-$3FFF (8192-16383) and that the video RAM lies at $0400-$07FF (1024-2047). This makes it impossible to work with text and graphics at the same time, but it makes it easier for us. Be careful with long programs or those which require a large amount of memory so that they do not conflict with the graphics memory. If this should happen, simply place the following line at the start of your program:

```
POKE 45,0 : POKE 46,64
```

which sets the variables to $4000 (16384). It must be noted that any program change will destroy the graphics screen and that saving the program to tape or disk will also save the graphics memory. If you want to make changes to your program after RUNning it, your must first load it back in, make the changes and then save it before starting it again.

It is important for all of these things that you try them out directly on the computer. Only in this way will you become master of all the factors and relationships and learn how to make effective use of them. The computer is the practice!

4.2.1 Initializing graphics

Before we conjure up our figures on the screen, we must take care of a few things. We must initialize the graphics by enabling the high-resolution graphics mode. The high-resolution screen will immediately be displayed, but since we haven't put anything there yet, "garbage" representing the memory's previous contents will be displayed. In addition to clearing the graphics, we must also "clear" the color memory. Clearing the color memory involves setting the entire screen to a single background and point color. It will most likely not be a single color when the switch is first made to the grapics screen because the high-resolution color memory is the old text storage. We also need a way of disabling the graphics, turning the graphics screen off. Therefore we must have a total of four routines before we can start doing any graphics programming. The routines perform the following tasks:

- turn graphics on
- clear graphics screen
- erase color
- turn graphics off

These four algorithms are presented individually in the following discussion. You will see them used in later graphics programs in the form of subroutines. They should belong to your permanent repertoire.

4.2.1.1 Turning graphics on

The following things are necessary for enabling the graphics mode:

a) Storage:

First we must decide where in memory the video RAM and bit-mapped graphics should go. Register 24 (bits 3 and 4-7) of the video controller and register 0 (bits 0 and 1) of the CIA 2 are used to specify these areas. The remaining examples all have bit-mapped graphics storage located in the range $2000 to $3FFF (8192-16383) and video RAM at $0400-$07FF (1024-2047). If you wish to use other areas, you must make the appropriate changes.

b) Graphics mode:

We must decide whether we want a high-resolution graphics picture with only one color per 8x8 point field or a multi-color graphics picture which permits a higher color resolution but fewer points in the x-direction.

To keep matters straight-forward, we will to work with high-resolution graphics. Hi-res is turned on by setting bits 5 and 6 (bit 6 must be set in any event) of register 17 of the VIC and clearing bit 4 of register 22. This last bit is normally cleared and may not need to be cleared to zero. This is done with two simple POKEs which are found in lines 10070 and 10080 of the following routine.

Selecting an address range is also straight-forward. Since we are not going to move the graphic areas from the first 16K of memory (remember from section 3.3.2 that the VIC can only address 16K) we do not have to make any changes to the register in CIA 2. The base address of the

graphic area must be set to the upper 8K of this address range. So we set the third bit of register 24 to point to $2000 (8192). This is accomplished in line 10090 of following subroutine:

```
10000 REM ************************
10010 REM **                    **
10020 REM **  TURN ON GRAPHICS  **
10030 REM **                    **
10040 REM ************************
10050 REM
10060 V = 53248 : REM BASE ADDRESS - VIDEO CONTROLLER
10070 POKE V+17, PEEK(V+17) OR (8+3)*16 : REM GRAPHICS ON
10080 POKE V+22, PEEK(V+22) AND 256-16 : REM MULTI-COLOR
      OFF
10090 POKE V+24, PEEK(V+24) OR 8 : REM GRAPHICS TO $2000
      (8192)
```

The operators AND and OR are described in Chapter 2. The line numbers of the routine are deliberately high so that you can easily add these routines to the end of your own program. After running this routine, the text mode can be turned back on by pressing <RUN/STOP><RESTORE>

4.2.1.2 Clearing the graphics screen

Since other data is in the memory that we want to use as a bit-mapped graphic area, we get a wild display of lines or random points on the screen when we turn on graphics. To erase each graphics point, we must clear each bit in the graphic storage to 0. We can clear a total of 8 bits (one byte) by POKEing. All we need is a FOR/NEXT loop which runs

from the start of the graphics storage ($2000 = 8192) to the end at $3FFF (16383) in order to erase all of the points. Since a picture consists of 320x200 = 64000 points, we have to erase 64000/8=8000 bytes.

```
10200 REM ******************************
10210 REM **                            **
10220 REM **  CLEAR GRAPHICS SCREEN  **
10230 REM **                            **
10240 REM ******************************
10250 REM
10260 BG = 8192 : REM BASE ADDRESS OF GRAPHICS STORAGE
10270 FOR X=BG TO BG+8000 : REM 8000 BYTES
10280 POKE X,0 : REM ERASE
10290 NEXT X
```

As you see, this procedure takes quite some time to run. If you type in the graphics package at the end of this chapter, you'll see how quickly the same function can be done in machine language. The variable BG does not really belong to this routine, as with the variable V in the previous routine. They should be placed at the beginning of each program. If we want to run the individual routines as subroutines, we must terminate them with RETURN statements.

4.2.1.3 Clearing the color RAM

The color for high-resolution graphics is always placed in the video RAM (see section 3.4). The upper nybble (4 bits) of each byte determine the color of the set points in the bit-mapped graphic area. The lower nybble determines the color of the reset points--in other words the background

color. Since the video RAM contained text before we switched to graphic mode, small colored squares will appear on the screen wherever text was previously. To eliminate these patterns, we must set the color RAM to a single color. This is done by the following routine:

```
10400 REM  ********************
10410 REM  **                  **
10420 REM  **  CLEAR COLOR     **
10430 REM  **                  **
10440 REM  ********************
10450 REM
10460 BC = 1024 : REM BASE ADDRESS OF THE VIDEO RAM
10470 CO = 6*16 + 7 : REM POINT COLOR=BLUE, BACKGROUND =
      YELLOW
10480 FOR X=BC TO BC+1000 : REM 1000 BYTES
10490 POKE X, CO : REM WITH POINT AND BACKGROUND COLOR
10500 NEXT X
```

The same thing applies to the variable BC as for the variables in the previous routines. CO is a variable which can be passed by the calling program after being assigned the color values for the point and background colors (line 10470 will have to be removed). You may want to make some changes to the routines so that you can better understand the techniques. Make the changes with care, since you're working directly with the heart of the computer. If, for example, you set BC equal to 0, your computer will probably crash since you would be clearing the zero page page of the memory, which stores very important data needed by the '64.

4.2.1.4 Turning graphics off

So far you have had to return from the graphics mode with <RUN/STOP><RESTORE>. This also has the effect of stopping the current program, something which is not always desirable. To add this function to our set of routines, we must restore the contents of the appropriate registers to their original values.

- clear bits 5/6 of register 17
- clear bit 4 of register 22
- clear bit 3 of register 24

This is done by the following routine:

```
10600 REM ************************
10610 REM **                    **
10620 REM **  TURN GRAPHICS OFF  **
10630 REM **                    **
10640 REM ************************
10650 REM
10660 V = 53248 : REM BASE ADDRESS - VIDEO CONTROLLER
10670 POKE V+17, PEEK(V+17) AND 255-6*16 : REM GRAPHICS OFF
10680 POKE V+22, PEEK(V+22) AND 255-1*16 : REM MULTICOLOR
      OFF
10690 POKE V+24, PEEK(V+24) AND 255-8 : REM CHARACTER  SET
BACK TO $1000 (4096)
```

Now we have all of the important prerequisite routines for working with graphics. We can now move on to displaying something on our screen.

4.2.2 Simple figures in bit-mapped graphics

After having spent time with the preliminaries, we come to our first attempt at graphics programming. Beginning with drawing simple points on the screen we will move on to the basic geometric forms--lines and circles from which all other figures can be created.

4.2.2.1 Points

We first divide the graphic display into coordinates. The first value always indicates the x-coordinate, the number of points between the left screen window edge and the given point (0-319). The second value given is the y-coordinate, the number of points between the upper screen edge and the point (0-199). The origin (coordinates 0,0) lies in the upper left corner of the screen. The lower right corner has the coordinates 319,199.

So far, so good. But we can't give the computer a set of coordinates directly in order to set a point. If you have read section 3.4.2, then you understand the organization of the high-resolution bit-mapped graphic screen. We must perform some calculations to determine the byte and bit which correspond to a given point. Of most interest to us is the use of the following formula within programs.

Let us start with the calculation of the y-coordinate.

In order to calculate the number of the graphics line (a line consists of 8 rows) in which the point is found, we need only divide the y-coordinate by 8 (without remainder):

```
line number = INT(yc/8)
```

Since each screen line consists of 320 bytes (each row consists of 320/8 = 40 bytes), we must multiply this number by 320 in order to get the starting address of the line relative to the start address of the graphics storage:

```
line address = 320 * INT(yc/8)
```

The remainder of the division (yc AND 7) represents the number of the row in this line and must be added:

```
row address = 320 * INT(yc/8) + (yc AND 7)
```

Calculating the x-coordinate is somewhat more difficult because we must calculate not only the byte, but the bit as well:

First we calculate the address of the appropriate byte relative to the starting address of the row (see above). We compute:

```
byte address = 8 * INT(xc/8)
```

Now we calculate the position of the desired bit within the given byte by creating a mask. The appropriate bit is set in the mask; all others are 0:

```
mask = 2^(7-(xc AND 7))
```

These parts are gathered together into the following routine:

```
10700 REM ************************
10710 REM **                    **
10720 REM **  CALCULATE POINT   **
10730 REM **     (SETTING)      **
10740 REM ************************
10750 REM
10760 RA = 320 * INT(YC/8) + (YC AND 7)
10770 BA = 8 * INT(XC/8)
10780 MA = 2^(7-(XC AND 7))
10790 AD = SA + RA + BA
10800 POKE AD, PEEK(AD) OR MA
10810 REM
10900 REM ************************
10910 REM **                    **
10920 REM **  CALCULATE POINT   **
10930 REM **     (ERASING)      **
10940 REM ************************
10950 REM
10960 RA = 320 * INT(YC/8) + (YC AND 7)
10970 BA = 8 * INT(XC/8)
10980 MA = 255 - 2^(7-(XC AND 7))
10990 AD = SA + RA + BA
11000 POKE AD, PEEK(AD) AND MA
11010 REM
11020 REM INTERNAL PARAMETERS:
11030 REM ********************
11040 REM RA: ROW ADDRESS
11050 REM BA: BYTE ADDRESS
11060 REM MA: MASK
11070 REM AD: DESTINATION ADDRESS
11080 REM
11090 REM PRE-DEFINED PARAMETERS:
11100 REM ************************
```

```
11110 REM SA: GRAPHICS STORAGE START ADDRESS (8192)
11120 REM XC: X-COORDINATE
11130 REM YC: Y-COORDINATE
```

As you can see, we distinguish between setting and erasing points in HGR. These cases must be handled separately. The variable SA gives the starting address of the graphics storage and always contains 8192 for our purposes. These routines are intended to be used as subroutines, as are the previous routines and must therefore end with RETURN statements. The following program illustrates the use of these routines as subroutines:

```
100 REM    ******************
110 REM    **                **
120 REM    **  SINE CURVE    **
130 REM    **                **
140 REM    ******************
150 REM
160 V=53248 : REM START ADDRESS OF THE VIC
170 SA = 8192 : REM START ADDRESS OF THE GRAPHICS STORAGE
175 POKE V+32, 10 : REM BORDER COLOR
180 GOSUB 10000 : REM TURN GRAPHICS ON
190 GOSUB 10200 : REM CLEAR GRAPHICS SCREEN
200 CO = 7*16 + 2 : GOSUB 10400 : REM SET COLOR
210 YC = 100 : REM DRAW X-AXIS
220 FOR XC=0 TO 319
230 GOSUB 10700 : REM DRAW POINT
240 NEXT XC
250 XC = 160 : REM DRAW Y-AXIS
260 FOR YC=0 TO 199
```

```
270 GOSUB 10700 : REM DRAW POINT
280 NEXT YC
290 FOR XC=0 TO 319 : REM DRAW SINE CURVE
300 YC = 70 * SIN (XC/25.5) + 99
310 GOSUB 10700 : REM DRAW POINT
320 NEXT XC
330 POKE 198,0 : REM CLEAR KEYS
340 WAIT 198,255 : REM WAIT FOR KEY
350 GOSUB 10600 : REM GRAPHICS OFF
360 END
370 REM
10000 REM ************************
10010 REM **                    **
10020 REM **  TURN ON GRAPHICS  **
10030 REM **                    **
10040 REM ************************
10050 REM
10070 POKE V+17, PEEK(V+17) OR (8+3)*16 : REM GRAPHICS ON
10080 POKE V+22, PEEK(V+22) AND 255-16 : REM MULTI-COLOR OFF
10090 POKE V+24, PEEK(V+24) OR 8 : REM GRAPHICS AT $2000
      (8192)
10100 RETURN
10110 REM
10200 REM  ****************************
10210 REM  **                        **
10220 REM  **  CLEAR GRAPHICS SCREEN  **
10230 REM  **                        **
10240 REM  ****************************
10250 REM
10270 FOR X=SA TO SA+8000 : POKE X,0 : NEXT X
10300 RETURN
10310 REM
10400 REM   ******************
```

```
10410 REM   **                **
10420 REM   **  CLEAR COLOR   **
10430 REM   **                **
10440 REM   ********************
10450 REM
10460 BC = 1024 : REM BASE ADDRESS OF THE VIDEO RAM
10480 FOR X=BC TO BC+1000 : POKE X,CO : NEXT X
10510 RETURN
10520 REM
10600 REM ***************************
10610 REM **                       **
10620 REM **  TURN GRAPHICS OFF    **
10630 REM **                       **
10640 REM ***************************
10650 REM
10670 POKE V+17, PEEK(V+17) AND 255-6*16 : REM GRAPHICS OFF
10680 POKE V+22, PEEK(V+22) AND 255-16 : REM MULTI-COLOR OFF
10690 POKE V+24, PEEK(V+24) AND 255-8 : REM CHARACTER SET
      BACK TO $1000 (4096)
10695 RETURN
10700 REM ***********************
10710 REM **                   **
10720 REM **  COMPUTE PRINT    **
10730 REM **     SETTING       **
10740 REM ***********************
10750 REM
10760 RA = 320 * INT(YC/8) + (YC AND 7)
10770 BA = 8 * INT(XC/8)
10780 MA = 2^(7-(XC AND 7))
10790 AD = SA + RA + BA
10800 POKE AD, PEEK(AD) OR MA
10810 RETURN
```

A few changes have been made to the routines in order to make them faster. Except for removal of some REM statements and combining statements onto a single line, these routines are the same as those presented before. You might like to experiment by changing a few things, such as the numbers in line 300.

4.2.2.2 Lines

Drawing a line between two points on the screen appears to be somewhat difficult. You see this daily in various program demos but rarely give any thought to how it is done. The problem is this: How do I determine which points on the screen lie on this line? In order to solve this problem, we must have a short lesson in analytic geometry. What we're looking for is a formula with which we can calculate the points on our line.

All of us have at one time or another heard (generally from school) of the point-slope line equation:

$$y = mx + b$$

where:

- x and y are the coordinates of a point on the line;
- m is the slope of the line;
- b is the point at which the line crosses the y-axis (also called the y-intercept).

Altering this equation slightly we get:

$$b = y-mx$$

If we know two points on the line (our end points x_1, y_1 and x_2, y_2) we can set the following two equations equal to each other since their slopes are equal:

$$b = y_1 - mx_1 \quad --- \quad b = y_2 - mx_2$$

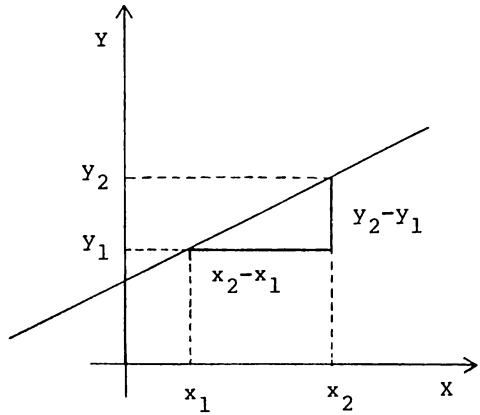
$$\rightarrow y_1 - mx_1 = y_2 - mx_2$$

$$\Leftrightarrow \qquad m = \frac{y_2 - y_1}{x_2 - x_1}$$

The last formula allows us to calculate the slope m of the previous equation. If b=0, the line passes through the origin at coordinates 0,0. If we shift this origin of the line to an end point, we must add the two corresponding coordinates (x_2 and y_2 in this case) to x and y. The following formula gives the equation of the line which already gives the shifted line and is set equal to m:

$$y = \frac{y_2 - y_1}{x_2 - x_1} * (x - x_2) + y_2$$

This formula is the basis for the program shown below and is calculated in pieces in the lines 10970, 11000, and 11020, whereby the x-coordinate XC always runs from X2 to X1 and the y-value is determined for each such x-value. A picture will help to clarify this formula:



The problem we encounter with this routine occurs when we try to draw a vertical line. In this case, x1=x2 and the denominator of the slope equals 0, leading to a DIVISION BY ZERO ERROR. We can get around this error by branching in line 10990 and there drawing a vertical line. As you will see, and what has been mentioned already, the speed of a BASIC program cannot compare to the speed of a machine language program. In spite of this fact, this routine, which can be used as a subroutine, will serve you well.

```
100 REM   **********
110 REM   **       **
120 REM   ** LINE **
130 REM   **       **
140 REM   **********
150 REM
160 V=53248: SA=8192
```

```
170 GOSUB 10000: REM GRAPHICS ON
180 CO=1*16+0:GOSUB 10400: REM SET COLOR
190 GOSUB 10200: REM ERASE GRAPHICS
270 X1=110: Y1=120: X2=130: Y2=140: REM END POINT
    COORDINATES
280 GOSUB 10900: REM LINE
290 WAIT 198, 255: REM WAIT FOR KEY
300 GOSUB 10600: REM GRAPHICS OFF
310 END
320 REM
10000 REM ***********************
10020 REM ** TURN GRAPHICS ON **
10040 REM ***********************
10050 REM
10070 POKE V+17, PEEK(V+17) OR (8+3)*16: REM GRAPHICS ON
10080 POKE V+22, PEEK(V+22) AND 255-16: REM MULTICOLOR OFF
10090 POKE V+24, PEEK(V+24) OR 8: REM GRAPHICS AT $2000
      (8192)
10100 RETURN
10110 REM
10200 REM ****************************
10220 REM ** CLEAR GRAPHICS SCREEN **
10240 REM ****************************
10250 REM
10270 FOR X=SA TO SA+8000: POKE X,0: NEXT X
10300 RETURN
10310 REM
10400 REM ******************
10420 REM ** CLEAR COLOR **
10440 REM ******************
10450 REM
10460 BA=1024: REM BASE ADDRESS OF THE VIDEO RAM
10480 FOR X=BA TO BA+1000: POKE X,CO: NEXT X
```

```
10510 RETURN
10520 REM
10600 REM ************************
10620 REM ** TURN OFF GRAPHICS **
10640 REM ************************
10650 REM
10670 POKE V+17, PEEK(V+17) AND 255-6*16: REM GRAPHICS OFF
10680 POKE V+22, PEEK(V+22) AND 255-16: REM MULTICOLOR OFF
10690 POKE V+24, PEEK(V+24) AND 255-8: REM CHARACTER SET
      BACK TO $1000 (4096)
10695 RETURN
10700 REM ********************
10720 REM ** COMPUTE POINT **
10730 REM **   (SETTING)    **
10740 REM ********************
10750 REM
10760 RA=320 * INT(YC/8) + (YC AND 7)
10770 BA=8 * INT(XC/8)
10780 MA=2^(7-(XC AND 7))
10790 AD=SA+RA+BA
10800 POKE AD, PEEK(AD) OR MA
10810 RETURN
10900 REM
10910 REM ***************
10930 REM ** DRAW LINE **
10950 REM ***************
10960 REM
10970 DY=Y2-Y1: DX=X2-X1: REM DIFFERENCES
10980 YC=Y2: XC=X2: REM Y-START
10990 IFDX=0THENFORYC=Y2TOY1STEPSGN(-DY):GOSUB11060:NEXTYC:
      GOTO 11050:REM VERT.
11000 DD=DY/DX: REM SLOPE
11010 FOR XC=X2 TO X1 STEP SGN(-DX)
```

```
11020 TC= INT(DD*(XC-X2)+Y2): REM COMPARE LINES
11030 IF TC<>YC THEN YC=YC+SGN(-DY): GOSUB 11060: GOTO 11030:
      REM DRAW VERTICAL
11040 GOSUB 11060: NEXT XC : REM NEXT X-COORD.
11050 RETURN
11060 GOSUB 10760: XC=XC+1: GOSUB 10760: XC=XC-1: RETURN:
      REM DRAW DOUBLE WIDTH
```

If you do not want to wait for the entire screen to be cleared, simply place a REM in front of line 190 in order to skip this procedure. The following variables are used to store various data:

Input values:
X1/Y1 and
X2/Y2 : end coordinates of line

Internal values:
DX/DY : differences of coordinate pairs
DD : the slope m
XC/YC : coordinate of the actual point
TC : temporary storage

Two points should be noted: The first concerns the function SGN, and the second, line 11060.

SGN has a very useful characteristic: If the number within the parentheses is positive, the result is 1, whereas if it is negative then the result is -1 (SGN returns 0 if the argument is 0). The function is used to determine the sign.

In line 11060 each point is doubled so that a double width point is plotted. This is necessary because individual points which have no neighbor in the x-direction appear very

dim or cannot be seen at all. Have fun with your line creations!

4.2.2.3 Circles and ellipses

Another important and often-used figure is the circle, or more general, the ellipse. Here too we will present a small demonstration routine with which you can draw ellipses and circles. First, however, we will present the mathematical foundations for those who are interested. In this book, we will examine two different methods of creating circles and ellipses. The first is somewhat simpler to understand and is presented now. The second, which results from the use of a polar-coordinate generator and allows the drawing of arcs, is found in the section "Pie charts" (5.1.3) in Chapter 5.

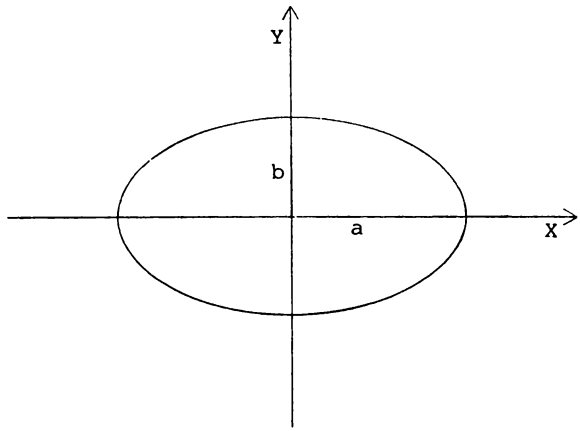
In our routine we proceed from the midpoint equation of the ellipse:

$$\frac{x^2}{a} + \frac{y^2}{b} = 1$$

where:

- x and y stand for the coordinates of the edge points of the ellipse;
- a is the radius in the x-direction;
- b is the radius in the y-direction.

The midpoint of the ellipse lies at the coordinate origin (0,0):



In order to use this formula in our program, we must first solve for y:

$$y = b * \left(1 - \frac{x^2}{a^2}\right)$$

With this complicated looking equation we can calculate the points of the ellipse. It must be noted, however, that only an arc of 90 degrees can be drawn at one time with this formula because an ellipse cannot be strictly represented by a function. In order to draw the other three arcs, we must reverse the signs of x and y. This is done for x in the FOR/NEXT loop in lines 11150-11190 where it assumes the values -1 and 1 in succession. Y is negated in line 11180. Because the above formula applies only to ellipses with center at the origin (0,0), we must add the appropriate

values to x and y as shown below.

If you want to draw a circle with this program, you must set both a and b to the same value (variables XR/YR) because a circle is a special case of a ellipse in which both radii are equal.

```
100 REM       ***************
110 REM       **           **
120 REM       **  ELLIPSE  **
130 REM       **           **
140 REM       ***************
150 REM
160 V=53248 : SA=8192
170 GOSUB 10000 : REM GRAPHICS ON
180 CO = 1*16 + 0 : GOSUB 10400 : REM SET COLOR
190 GOSUB 10200 : REM CLEAR GRAPHICS
270 XR=40:YR=20:XM=160:YM=100:REM X/Y-RADIUS===MIDPOINT
    COORDINATES
280 GOSUB 11100 : REM ELLIPSE
290 WAIT 198,255 : REM WAIT FOR KEY
300 GOSUB 10600 : REM GRAPHICS OFF
310 END
320 REM
10000 REM ************************
10020 REM **  TURN ON GRAPHICS  **
10040 REM ************************
10050 REM
10070 POKE V+17, PEEK(V+17) OR (8+3)*16 : REM GRAPHICS ON
10080 POKE V+22, PEEK(V+22) AND 255-16 : REM MULTI-COLOR OFF
10090 POKE V+24, PEEK(V+24) OR 8 : REM GRAPHICS TO $2000
      (8192)
10100 RETURN
10110 REM
```

```
10200 REM  ****************************
10220 REM  **  CLEAR GRAPHICS SCREEN  **
10240 REM  ****************************
10250 REM
10270 FOR X= SA TO SA+8000 : POKE X,0: NEXT X
10300 RETURN
10310 REM
10400 REM  ********************
10420 REM  **  CLEAR COLOR  **
10440 REM  ********************
10450 REM
10460 BC = 1024 : REM BASE ADDRESS OF THE VIDEO RAM
10480 FOR X=BC TO BC+1000 : POKE X,CO : NEXT X
10510 RETURN
10520 REM
10600 REM ************************
10620 REM **  TURN OFF GRAPHICS  **
10640 REM ************************
10650 REM
10670 POKE V+17, PEEK(V+17) AND 255-6*16 : REM GRAPHICS OFF
10680 POKE V+22, PEEK(V+22)AND 255-16 : REM MULTI-COLOR OFF
10690 POKE V+24, PEEK(V+24) AND 255-8 : REM CHARACTER SET
      BACK TO $1000 (4096)
10695 RETURN
10700 REM **********************
10720 REM **  COMPUTE POINT  **
10730 REM **     SETTING     **
10740 REM **********************
10750 REM
10760 RA = 320 * INT(YC/8) + (YC AND 7)
10770 BA = 8 * INT(XC/8)
10780 MA = 2^(7-(XC AND 7))
10790 AD = SA + RA + BA
```

```
10800 POKE AD, PEEK(AD) OR MA
10810 RETURN
11100 REM
11110 REM ********************
11120 REM **  DRAW ELLIPSE  **
11130 REM ********************
11140 REM
11150 FOR F2=-1 TO 1 STEP 2 : REM RIGHT/LEFT FLAG
11160 FOR X=0 TO F2*(XR) STEP F2
11170 TC = YR * SQR(1-X^2/XR^2):XC=X+XM : REM CALCULATE
      POINT
11180 YC = YM + TC:GOSUB 10760:YC=YM - TC:GOSUB 10760 : REM
      PRINTS ABOVE/BELOW
11190 NEXT X,F2:RETURN
```

The four parameters are:

XM/YM : coordinates of the midpoint
XR/YR : x/y radii (a and b)

4.3 Sprite programming

One of the most fascinating features of your computer is the ability to display up to 8 sprites on the screen at the same time. If you have read section 3.5, then you already had a short overview of sprite organization. Here you will learn how to acutally use sprites.

4.3.1 Creating sprites

The first step in programming sprites is designing the objects. This is quite a complicated undertaking since the layout of sprites in memory is relatively complicated.

As you know from section 3.5, a sprite has a resolution of 24x21 points (in multi-color 12x21). Each point is represented by a bit (two bits for MC) in memory. Eight bits form a byte. Each line of a sprite contains 24/8 = 3 bytes. These three bytes are adjacent to one another in memory. The next three bytes define the second line, and so forth.

To create a sprite you should draw it on a template such as that found in section 6.6 of this book (after first making copies of it) or create one using the sprite editor found on these next pages. Using a template, you can indicate each set point with an x (or a digit standing for the color in multi-color mode) and form a complete and clear picture of your sprite. Be sure to draw at least two points next to each other in the x-direction or else the point may be difficult to see. This does not apply to multi-color because a multi-color point is already double-width.

After the drawing is complete on the template, replace

each x with a 1 and each empty box with a 0. If you are developing a multi-color sprite, replace each pair of adjacent boxes with the color number. Using the conversion table in section 6.5, calculate the decimal value of each 8-bit group. By doing this, you will get a total of 63 numbers ranging from 0 to 255 which comprise the sprite definition.

Using BASIC, there are several ways to save and recall sprites. The first and simplest way is to use DATA statements. The DATA statements are placed one after another in order to save room, but the organization is clearer if you use the following form with three bytes across and 21 lines long:

```
1000 DATA 000,000,000
1010 DATA 000,000,000
1020 DATA 002,000,064
1030 DATA 001,000,128
1040 DATA 000,129,000
1050 DATA 000,066,000
1060 DATA 000,060,000
1070 DATA 000,126,000
1080 DATA 000,195,000
1090 DATA 001,141,128
1100 DATA 003,044,192
1110 DATA 031,255,248
1120 DATA 062,153,124
1130 DATA 125,066,190
1140 DATA 255,255,255
1150 DATA 001,255,128
1160 DATA 001,189,128
1170 DATA 003,060,192
1180 DATA 015,000,240
1190 DATA 015,000,240
1200 DATA 000,000,000
```

It is not immediately obvious that these DATA statements define a spaceship. Each DATA line contains the information for a single sprite line. In order to read this data into the appropriate memory locations (into the sprite block), we must add a small routine, which might look something like this:

```
100 AD = 13*64 : REM ADDRESS OF BLOCK 13
110 FOR X=0 TO 62
120  READ DT : REM READ 63 DATA ITEMS
130  POKE AD+X, DT : REM POKE INTO BLOCK 13
140 NEXT X
```

This routine reads the 63 data values and POKEs them into memory. This form of sprite storage requires quite a lot of space. Another less space-consuming method of storage involves writing the sprite data to disk or cassette in a sequential file. By doing this, you can save program memory at the expense of having to read the sprite pattern back from disk or cassette.

When first creating a sprite definition file, we still need the DATA statements. With the help of the following program, the individual values can be read from the DATA statements and placed on the disk as a sequential file:

```
10 OPEN 1,8,2,"SPRITE,S,W" : OPEN FILE FOR WRITING
20 FOR X=0 TO 62
30  READ DT : REM READ 63 DATA
40  PRINT#1, CHR$(DT) : REM WRITE TO DISKETTE
50 NEXT X             : REM (ASCII FORMAT)
60 CLOSE 1 : REM CLOSE FILE
```

The name of the file in this example is "SPRITE". The fol-

lowing program will read these data in again and POKE them directly into the appropriate memory locations:

```
10 AD = 13*64 : REM ADDRESS OF BLOCK 13
20 OPEN 1,8,2,"SPRITE,S,R" : REM OPEN SEQ. FILE FOR READING
30 FOR X=0 TO 62
40  INPUT#1, DT$ : REM READ DATA (ASCII FORMAT)
50  POKE AD+X, ASC(DT$+CHR$(0)) : REM AND POKE
60 NEXT X
70 CLOSE 1 : REM CLOSE FILE
```

It is also possible to save a sprite as a program file and read it back in the same manner.

As you can see, this whole thing is rather complicated. Following is a program which simplifies working with sprites. This sprite editor, written partly in BASIC and partly in machine language, makes it easy to create a high-resolution sprite that you may use in your own programs. The sprite editor creates program files which can be read in the same way as demonstrated in the last routine, if you first replace line 20 with the following line:

```
20 OPEN 1,8,2,"SPRITE,P,R" : REM OPEN PROGRAM FILE FOR
   READING
```

Although the program is quite lengthy, it is also very useful. After typing it in, but before you execute it, be sure to save it because it alters various BASIC pointers.

```
100 REM **********************
110 REM **                    **
120 REM **  SPRITE EDITOR     **
130 REM **                    **
140 REM **********************
150 REM
160 REM INITIALIZATION:
170 REM ***************
180 GOSUB2730:REM READ MACHINE LANGUAGE ROUTINES
190 POKE 53280,0:POKE 53281,0:REM BACKGROUND/BORDER COLOR
200 POKE650,255:REM REPEAT ALL CHARACTERS
210 POKE 45,0:POKE 46,80:RUN 220:REM BASIC END=$5000
220 REM
230 REM MACHINE LANGUAGE ROUTINES:
240 REM ***************************
250 IN%=18432:REM INIT. ROUTINE
260 PT%=18632:REM DRAW POINT
270 NE%=18567:REM COORDINATE SYSTEM
280 LD%=18503:REM LOAD CHARACTER SET
290 SV%=18531:REM SAVE CHARACTER SET
300 CA%=18758:REM CATALOG
310 CM%=18712:REM COMMAND IDENTIFICATION
320 IV%=18830:REM INVERT
330 SR%=18844:REM SHIFT RIGHT
340 SL%=18870:REM SHIFT LEFT
350 SU%=18895:REM SHIFT UP
360 SD%=18935:REM SHIFT DOWN
370 Q   =  704:REM SPRITE BLOCK ADDRESS
380 V   =53248:REM VIDEO CONTROLLER
390 REM
400 REM CONTROL CHARACTERS:
410 REM ********************
420 CO$=CHR$(147):REM CLEAR SCREEN
```

```
430 C1$=CHR$( 19):REM HOME
440 C2$=CHR$(183):REM HIGH LINE
450 C8$=CHR$( 99)+CHR$( 99)+CHR$( 99):C3$=CHR$(117)+C8$+
    CHR$(105)
460 C4$=CHR$(106)+C8$+CHR$(107):REM LOWER SPRITE WINDOW
    BORDER 1
470 C5$=CHR$(117)+C8$+C8$+CHR$(105):REM UPPER BORDER 2
480 C8$=CHR$(106)+C8$+C8$+CHR$(107):REM LOWER BORDER 2
490 C9$=CHR$( 98):REM MIDDLE LINE (VERT)
500 C6$=CHR$( 18):REM RVS ON
510 C7$=CHR$(146):REM RVS OFF
520 NA%=828:REM FILENAME LENGTH($C800)
530 DA%=186:REM DEVICE ADDRESS($BA)
540 KY%=821:REM KEY/COMMAND CODE
550 SS%=  1:REM SPRITE SIZE
560 YC%=822:REM Y-COORD
570 XC%=823:REM X-COORD
580 REM
590 REM DEFINE COLORS:
600 REM ***************
610 DATA 144,  5, 28,159,156, 30, 31,158
620 DATA 129,149,150,151,152,153,154,155
630 DIM C$(16):FOR Y=0 TO 15:READ X:C$(Y)=CHR$(X):NEXT Y
640 N=1:F(0)=0:F(1)=1:V$=" ":SYS IN%:REM COLORS/INIT
650 REM
660 REM ERASE ROUTINE (FIELD CONSTRUCTION)
670 REM ***********************************
680 SYS IN% : REM ERASE SPRITE
690 PRINT C0$
700 PRINT C1$;SPC(13);C$(7);"SPRITE EDITOR"
710 PRINT SPC(8);C$(1);"(C) 1984 BY AXEL PLENGE"
720 PRINT C$(4);:FOR X=1 TO 40:PRINT C2$;:NEXT X
730 PRINTC$(7)"  7"C$(6)"6543210"C$(7)"7"C$(6)"6543210"C$(7)
```

```
    "7"C$(6)"6543210";
740 SYS NE% : REM DRAW GRID
750 GOSUB 1820:PRINT:PRINT:REM CREATE STATUS FIELD
760 PRINT:PRINT:PRINTTAB(30);C3$
770 FOR X=1 TO 3:PRINTTAB(30);C9$;"   "C9$:NEXT X:PRINT
    TAB(30);C4$:REM TEST SPRITE1
780 PRINTTAB(27);"  ";C5$;"  ":FOR X=1 TO 5:PRINTTAB(27);
    "  ";C9$;"      ";C9$:NEXT X
790 PRINTTAB(27);"  ";C8$;"  ";:REM TEST SPRITE 2
800 POKE 53248+21,3:X=0:Y=0:REM SPRITES AT X,Y COORDINATES=0
810 REM
820 REM INPUT LOOP:
830 REM ***********
840 A=X+2:B=Y+4:GOSUB 2450:REM POSITIONING
850 POKE XC%,X:POKE YC%,Y:F=0:REM TRANSMIT COORDINATES
860 PRINT C$(7);C6$;" ";CHR$(157);:REM BLINK PHASE ON
870 FOR S=1 TO 50:GETA$:IF A$<>"" THEN 890
880 NEXTS:SYS PT%:FOR S=1 TO 50:GET A$:IF A$="" THEN NEXT S:
    GOTO 860:REM TURN OFF
890 REM
900 REM COMMAND RECOGNITION:
910 REM ********************
920 SYS PT%:C=ASC(A$):POKE KY%,C:SYS CM%:S=PEEK(KY%):REM
    PASS CMD
930 REM DIVISION:
940 ON S GOTO 1050,1050,1070,1070,1090,1090
950 ON S-6 GOTO 1110,1110,1910,1910,1910,1910
960 ON S-12 GOTO 1910,1910,1910,1910,650,1360
970 ON S-18 GOTO 1450,1490,1570,1130,2150
980 ON S-23 GOTO 1200,1970,1240,810
990 REM
1000 REM EXECUTE COMMANDS:
1010 REM *****************
```

```
1020 REM
1030 REM MOVE CURSOR:
1040 REM ************
1050 X=X+1:IF X=24 THEN X=0:GOTO 1090
1060 GOTO 810:REM RIGHT
1070 X=X-1:IF X<0 THEN X=23:GOTO 1110
1080 GOTO 810:REM LEFT
1090 Y=Y+1:IF Y=21 THEN Y=0
1100 GOTO 810:REM DOWN
1110 Y=Y-1:IF Y<0 THEN Y=20
1120 GOTO 810:REM UP
1130 REM
1140 REM EXIT:
1150 REM *****
1160 A=2:B=15:GOSUB 2450:REM POSITIONING
1170 PRINT C6$;C$(7);"EXIT?";C7$;C$(6):INPUT T$
1180 IF T$="Y" OR T$="YES" THEN SYS 64738:REM COLD START
1190 GOTO 690
1200 REM
1210 REM CATALOG:
1220 REM ********
1230 PRINT C0$:SYS CA%:GOSUB 2490:GOTO 690
1240 REM
1250 REM SHIFT:
1260 REM ******
1270 GOSUB 2530:GOSUB 2440:PRINT C$(1);"SHIFT":REM MESSAGE
     FIELD
1280 PRINT TAB(27)"TO:":PRINT TAB(27)"RIGHT(R),":PRINT
     TAB(27)"LEFT(L),"
1290 PRINT TAB(27)"UP  (U),":PRINT TAB(27)"DOWN(D):"
1300 GOSUB 2490
1310 IF T$="R" THEN SYS SR%:GOTO1350
1320 IF T$="L" THEN SYS SL%:GOTO1350
```

```
1330 IF T$="U" THEN SYS SU%:GOTO1350
1340 IF T$="D" THEN SYS SD%
1350 GOSUB 2530:GOTO 700
1360 REM
1370 REM SPRITE SIZE:
1380 REM ************
1390 ON SS%+1 GOSUB 1410,1420,1430,1440
1400 POKE V+23,A:POKE V+29,B:SS%=(SS%+1) AND 3:GOTO 810
1410 A=2:B=2:RETURN
1420 A=0:B=2:RETURN
1430 A=2:B=0:RETURN
1440 A=0:B=0:RETURN
1450 REM
1460 REM INVERT SPRITE:
1470 REM ***************
1480 SYS IV% : GOTO 700
1490 REM
1500 REM SAVE SPRITE:
1510 REM ************
1520 GOSUB 2530
1530 GOSUB 2440:PRINT C6$;C$(1);"SAVING":PRINT TAB(27);C6$;
     "SPRITE";C7
1540 GOSUB 1700:IF F=1 THEN F=0:GOTO 1490:REM READ
     INPUT/ERROR
1550 IF F=2 THEN F=0:GOTO 1630:REM ERROR
1560 SYS SP%:GOTO 1650:REM SAVE
1570 REM
1580 REM LOAD SPRITE:
1590 REM ************
1600 GOSUB 2530
1610 GOSUB 2440:PRINT C6$;C$(1);"LOADING":PRINT TAB(27);
     C$(1);"SPRITE";C7$
1620 GOSUB 1700:IF F=1 THEN F=0:GOTO 1570
```

```
1630 IF F=2 THEN F=0:GOTO 690
1640 SYS LD%
1650 REM READ ERROR (ONLY FOR DISK!):
1660 OPEN 1,8,15:INPUT#1,DS,DS$,DT,DB:CLOSE1
1670 IF DS<20 THEN 690:REM OK
1680 PRINT:T$=STR$(DS)+","+DS$+","+STR$(DT)+","+STR$(DB)
1690 GOSUB 2600:PRINT T$:FOR S=1 TO 2000:NEXT S:GOTO 690:REM
     FLASH
1700 REM
1710 REM INPUT NAME:
1720 REM ***********
1730 A$="":PRINT:PRINT TAB(27)"FILENAME"C$(6):PRINT TAB(27);
     :INPUT A$:T=LEN(A$)
1740 S=VAL(RIGHT$(A$,1))
1750 IF S<>0 AND LEFT$(RIGHT$(A$,2),1)=";" THEN T=T-2:
     POKE DA,S:REM DEVICE ADDR.
1760 IF T=0 THEN F=2:RETURN:REM NO NAME
1770 IF T>17 THEN 1800
1780 REM NAMES TO MACHINE LANGUAGE ROUTINES:
1790 POKE NA%,T:FOR S=1 TO T:POKE NA%+S,ASC(MID$(A$,S,1)):
     NEXT S:RETURN
1800 PRINT CHR$(145);:T$=C6$+"LENGTH!"+C7$:GOSUB 2600:REM
     ERROR MESSAGE
1810 PRINT C$(6):F=1:RETURN
1820 REM
1830 REM CREATE STATUS FIELD:
1840 REM ********************
1850 A=27:B=4:GOSUB 2450
1860 GOSUB 2530
1870 A=27:B=5:GOSUB 2450
1880 PRINT TAB(27);C$(7);"COLORS:"
1890 PRINT TAB(27);C$(2);:FOR S=1 TO 7:PRINT CHR$(163);:
     NEXT S:PRINT C$(6)
```

```
1900 FOR S=0 TO 1:PRINT TAB(27);"CLR.";S;":";F(S):NEXT S:
     RETURN
1910 REM
1920 REM PLOT:
1930 REM *****
1940 S=((S-12) AND 2)/2:REM DETERMINE PLOT COLOR
1950 T=X/8:AD=INT(T):T=2^(7-8*(T-AD)):AD=Y*3+AD+Q
1960 POKE AD,PEEK(AD) AND (255-T) OR S*T:GOTO 810
1970 REM
1980 REM SELECT COLOR:
1990 REM *************
2000 PRINTCHR$(147)
2010 A=0:B=4:GOSUB 2450:PRINT TAB(4);C$(1)"S"C$(2)"E"C$(3)
     "L"C$(4)"E"C$(5)"C";
2020 PRINT C$(6)"T "C$(7)"C"C$(4)"O"C$(6)"L"C$(2)"O"C$(7)"R"
2030 PRINT TAB(4);C$(1);CHR$(172);:FOR S=1 TO 32:PRINT
     CHR$(162);:NEXT S:PRINT CHR$(187)
2040 FOR S=1 TO 2:PRINT TAB(4);C6$;CHR$(161);
2050 FOR T=0 TO 15:PRINT C$(T);"  ";:NEXT T:PRINT C$(1);C7$;
     CHR$(161):NEXT S
2060 PRINT TAB(4);C6$;CHR$(161);
     " 0 1 2 3 4 5 6 7 8 9101112131415";C7$;CHR$(161)
2070 PRINT:PRINT C$(6);"BACKGROUND (F1) OR SPRITE CLR (F3):";
2080 GOSUB 2490:T=ASC(T$)-133:REM FUNCTION KEY
2090 IF T<0 OR T>1 THEN GOSUB 2590:GOTO 690:REM ERROR
2100 IF T>1 THEN T=T-4
2110 PRINT T:T$="":INPUT "    COLOR ";T$:S=ABS(INT(VAL(T$)))
2120 IF T$="" OR S>15 THEN GOSUB 2590:GOTO690:REM ERROR
2130 F(T)=S:POKE V+33,F(0):REM SET BACKGROUND COLOR
2140 POKE V+39,F(1):POKE V+40,F(1):GOTO 690:REM SET SPRITE
     COLOR
2150 REM
2160 REM COMMANDS:
```

```
2170 REM *********
2180 POKE V+21,0 : REM SPRITES OFF
2190 PRINT C0$;C6$;C$(2)"              ";C$(7);
2200 PRINT "COMMANDS";C$(2);"                    ";C7$;
2210 PRINT C$(4);:FOR S=1 TO 40:PRINT CHR$(184);:NEXT S:PRINT
2220 PRINT C$(1)" NO.  "C6$"COMMAND]"C7$"-"C$(5)" FUNCTION"
     C$(4)
2230 FOR S=1 TO 10:PRINT "-----";:NEXT S
2240 PRINT C$(1)" (1)  "C6$"(^...)"C7$"-"C$(5)" MOVE CURSOR"
2250 PRINT C$(1)"      "C6$"(2QWA )"C7$;C$(5)
2260 PRINT C$(1)" (2)  "C6$"(F1/F3)"C7$"-"C$(5)" PLOT/UNPLOT"
2270 PRINT C$(1)" (3)  "C6$"(F)    "C7$"-"C$(5)
     " SET BCKGND & SPRITE CLR"
2280 PRINT C$(1)" (4)  "C6$"(B)    "C7$"-"C$(5)
     " COMMAND MENU "
2290 PRINT C$(1)" (5)  "C6$"(G)    "C7$"-"C$(5)" SPRITE SIZE"
2300 PRINT C$(1)" (6)  "C6$"(I)    "C7$"-"C$(5)
     " INVERT SPRITE"
2310 PRINT C$(1)" (7)  "C6$"(V)    "C7$"-"C$(5)
     " SHIFT SPRITE"
2320 PRINT C$(1)" (8)  "C6$"(L)    "C7$"-"C$(5)
     " ERASE SPRITE"
2330 PRINT C$(1)" (9)  "C6$"(CTRLG)"C7$"-"C$(5)
     " GET-LOAD SPRITE"
2340 PRINT C$(1)"(10)  "C6$"(CTRLS)"C7$"-"C$(5)
     " SAVE-SAVE SPRITE"
2350 PRINT C$(1)"(11)  "C6$"(C)    "C7$"-"C$(5)
     " DIRECTORY/CATALOG"
2360 PRINT C$(1)"(12)  "C6$"(CTRLX)"C7$"-"C$(5)" EXIT"
2370 GOSUB 2490:POKE V+21,3:GOTO 690:REM WAIT + SPRITES ON
2380 REM
2390 REM SUBROUTINES:
2400 REM ************
```

```
2410 REM
2420 REM POSITIONING:
2430 REM ************
2440 A=27:B=5:REM MESSAGE FIELD
2450 PRINT Cl$;:FOR S=2 TO B:PRINT:NEXT S:PRINT TAB(A);:
     RETURN
2460 REM
2470 REM KEY INPUT:
2480 REM **********
2490 WAIT 198,255:GET T$:RETURN
2500 REM
2510 REM ERASE MESSAGE FIELD:
2520 REM ********************
2530 GOSUB 2440
2540 FORS=1TO6:PRINTTAB(27);:FORT=1TO4:PRINT"    ";:NEXT T:
     PRINT:NEXT S :REM ERASE MESSAGE FIELD
2550 RETURN
2560 REM
2570 REM FLASH ERROR:
2580 REM ************
2590 T$="ILLEGAL!"
2600 A=4:B=18:GOSUB2450:PRINTC$(1):FOR S=1 TO 9:
     PRINTTAB(4)T$:GOSUB 2630:PRINTCHR$(145);
2610 PRINT TAB(4)"            ";
2620 PRINT CHR$(145):GOSUB 2630:NEXT S:
     PRINT TAB(4)"            ":F=1:RETURN
2630 FOR T=1 TO 75:NEXT T:RETURN:REM DELAY LOOP
2640 REM
2650 REM ****************************
2660 REM **                        **
2670 REM **  MACHINE LANG. ROUTINES **
2680 REM **                        **
2690 REM ****************************
```

```
2700 REM
2710 REM DATA WILL BE ERASED AFTER START!!!
2720 REM
2730 FOR I = 1 TO 16 : READ X : NEXT I : REM SKIP FIRST DATA
     (COLORS)
2740 FOR I = 18432 TO 18969
2750 READ X : POKE I,X : S=S+X : NEXT
2760 DATA 162, 62,169,  0,157,192,  2,202, 16,250,169, 11
2770 DATA 141,248,  7,141,249,  7,169, 16,141,  0,208,169
2780 DATA 163,141,  1,208,169,  7,141,  2,208,169,201,141
2790 DATA   3,208,169,  3,141, 16,208,141, 21,208,169,  2
2800 DATA 141, 23,208,141, 29,208,169,  0,141, 27,208,141
2810 DATA  28,208,169,  1,141, 39,208,141, 40,208, 96,173
2820 DATA  60,  3,162, 61,160,  3, 32,249,253,169,  2,166
2830 DATA 186,160,  0, 32,  0,254,169,  0,162,192,160,  2
2840 DATA  76,213,255,173, 60,  3,162, 61,160,  3, 32,249
2850 DATA 253,169,  2,166,186,160,  0, 32,  0,254,169,192
2860 DATA 133,  2,169,  2,133,  3,169,  2,162,255,160,  2
2870 DATA  76,216,255,160,  0,169, 13, 32,210,255,152, 72
2880 DATA  56,233, 10,144, 12,168,233, 10,144,  4,168,169
2890 DATA  50, 44,169, 49, 44,169, 32, 32,210,255,152,  9
2900 DATA  48, 32,210,255,104,168,162,  0, 32,206, 72,169
2910 DATA  29, 32,210,255,232,224, 24,208,243,169,165, 32
2920 DATA 210,255,200,192, 21,208,194, 96,174, 55,  3,172
2930 DATA  54,  3,138, 72,152, 72,133,  2, 10,101,  2,133
2940 DATA   2,138,160,255, 56,233,  8,200,176,251,105,  8
2950 DATA 170,152, 24,101,  2,168,169,  0, 56,106,202, 16
2960 DATA 252, 57,192,  2,208,  3,160,  0, 44,160,  6,162
2970 DATA   6,185, 12, 73, 32,210,255,200,202,208,246,104
2980 DATA 168,104,170, 96,146, 31,111, 31,146,157, 18, 28
2990 DATA  32, 31,146,157,173, 53,  3,162,  0,160, 28,232
3000 DATA 221, 43, 73,240,  3,136,208,247,142, 53,  3, 96
3010 DATA  87, 29, 81,157, 65, 17, 50,145,133,137,134,138
```

```
3020 DATA 135,139,136,140, 76, 71, 73, 19,  7, 24, 66, 67
3030 DATA  70, 86,169, 36,133,  2,169,  1,162,  2,160,  0
3040 DATA  32,249,253,169,  2,166,186,160,  0, 32,  0,254
3050 DATA 169,  0,162,  0,160, 64,134, 95,132, 96, 32,213
3060 DATA 255,165, 95,164, 96, 32, 55,165,173,  0,  3, 72
3070 DATA 173,  1,  3, 72,169, 61,141,  0,  3,169,227,141
3080 DATA   1,  3, 32,195,166,104,141,  1,  3,104,141,  0
3090 DATA   3, 96,162, 62,189,192,  2, 73,255,157,192,  2
3100 DATA 202, 16,245, 96,162,  0,160,  3, 24,126,192,  2
3110 DATA 232,136,208,249,169,  0,106, 29,189,  2,157,189
3120 DATA   2,224, 63,208,233, 96,162, 63,160,  3, 24, 62
3130 DATA 191,  2,202,136,208,249,169,  0, 42, 29,194,  2
3140 DATA 157,194,  2,138,208,234, 96,162, 62,134,  2,160
3150 DATA  21,132,  3,166,  2,189,132,  2, 72,189,192,  2
3160 DATA 168,104,157,192,  2,152,202,202,202,198,  3,208
3170 DATA 239,166,  2,202,134,  2,224, 59,208,221, 96,162
3180 DATA   2,134,  2,160, 21,132,  3,166,  2,189,252,  2
3190 DATA  72,189,192,  2,168,104,157,192,  2,152,232,232
3200 DATA 232,198,  3,208,239,198,  2, 16,226, 96
3210 IF S <> 61707 THEN PRINT "ERROR IN DATA !!" : END
3220 PRINT "OK" : RETURN
```

```
7:      4800                    .OPT P1
10:                     ;
20:                     ;
30:                     ; MACHINE LANGUAGE ROUTINES
40:                     ; ***************************
50:                     ;
60:                     ;
70:                     ;
80:     4800                    *=   $4800
90:                     ;
100:                    ;
110:                    ;JUMP ADDRESSES AND REGISTERS
120:                    ;*****************************
130:                    ;
140:                    ;
150:    FFD2            CHROUT   =    $FFD2   ;OUTPUT CHARACTER
160:    FDF9            FNPAR    =    $FDF9   ;SET FILENAME
                                               PARAMETERS
170:    FE00            FPAR     =    $FE00   ;SET FILE
                                               PARAMETERS
180:    FFD8            SV       =    $FFD8   ;SAVE TO
                                               DISK/CASSETTE
190:    FFD5            LD       =    $FFD5   ;LOAD FROM
                                               DISK/CASSETTE
200:    D000            V        =    $D000   ;VIDEO CONTROLLER
                                               (53248)
210:    02C0            BLOCK    =    704     ;SPRITE BLOCK 11
220:    000B            BLOCKN   =    11      ;BLOCK NUMBER 11
230:    033C            LENGTH   =    $033C   ;FILENAME LENGTH
235:    033D            NAME     =    $033D
240:    0334            MODE     =    $0334   ;STORAGE MODE
250:    0335            KEY      =    $0335   ;COMMAND KEY
                                               PRESS
```

```
260:   0336           YCOORD   =    $0336   ;Y-COORDINATE
270:   0337           XCOORD   =    $0337   ;X-COORDINATE
280:                  ;
290:                  ;TEST SPRITE ERASE AND PARAMETERS
300:                  ;*********************************
310:                  ;
320:   4800 A2 3E     INIT     LDX  #62     ;63 BYTES
330:   4802 A9 00              LDA  #$00
340:   4804 9D C0 02 I1        STA  BLOCK,X  ;ERASE BLOCK 11
350:   4807 CA                 DEX
360:   4808 10 FA              BPL  I1
370:   480A A9 0B              LDA  #BLOCKN  ;BLOCK 11
380:   480C 8D F8 07           STA  $07F8    ;POINTER SPRITE
                                              0 TO 11
390:   480F 8D F9 07           STA  $07F9    ;POINTER SPRITE
                                              1 TO 11
400:   4812 A9 10              LDA  #16
410:   4814 8D 00 D0           STA  V+0      ;SPRITE 0
                                              X-COORD HIGH BYTE
420:   4817 A9 A3              LDA  #163
430:   4819 8D 01 D0           STA  V+1      ;Y-COORDINATE
440:   481C A9 07              LDA  #7
450:   481E 8D 02 D0           STA  V+2      ;SPRITE 1 X-COORD.
                                              HIGH BYTE
460:   4821 A9 C9              LDA  #201
470:   4823 8D 03 D0           STA  V+3      ;Y-COORDINATE
480:   4826 A9 03              LDA  #%00000011  ;X-C. HIGH
                                                BYTE=1
490:   4828 8D 10 D0           STA  V+16
500:   482B 8D 15 D0           STA  V+21     ;TURN SPRITES ON
510:   482E A9 02              LDA  #%00000010
520:   4830 8D 17 D0           STA  V+23     ;SPRITE 1
                                              Y-EXPANSION
```

```
530:   4833 8D 1D D0          STA  V+29     ;SPRITE 1
                                             X-EXPANSION
540:   4836 A9 00             LDA  #$00
550:   4838 8D 1B D0          STA  V+27     ;SPRITE-PRIORITY
560:   483B 8D 1C D0          STA  V+28     ;NORMAL COLOR
                                             SPRITES
570:   483E A9 01             LDA  #$01
580:   4840 8D 27 D0          STA  V+39     ;SPRITE 0  WHITE
590:   4843 8D 28 D0          STA  V+40     ;SPRITE 1  WHITE
600:   4846 60                RTS           ;RETURN
610:                ;
620:                ;LOAD SPRITE
630:                ;************
640:                ;
650:   4847 AD 3C 03 LOAD      LDA  LENGTH   ;NAME LENGTH
660:   484A A2 3D             LDX  #< NAME  ;FILENAME ADDRESS
                                             LOW
670:   484C A0 03             LDY  #> NAME  ;HIGH BYTE
680:   484E 20 F9 FD          JSR  FNPAR
690:   4851 A9 02             LDA  #$02     ;LOGICAL FILE
                                             NUMBER
700:   4853 A6 BA             LDX  $BA      ;DEVICE ADDRESS
710:   4855 A0 00             LDY  #$00     ;SECONDARY ADDRESS
720:   4857 20 00 FE          JSR  FPAR
730:   485A A9 00             LDA  #$00     ;LOAD/VERIFY FLAG
740:   485C A2 C0             LDX  #< BLOCK  ;START ADDRESS
                                               (LOW BYTE)
750:   485E A0 02             LDY  #> BLOCK  ;START ADDRESS
                                               (HIGH BYTE)
760:   4860 4C D5 FF          JMP  LD
770:                ;
780:                ;SAVE SPRITE
790:                ;************
```

```
800:                    ;
810:    4863 AD 3C 03 SAVE      LDA  LENGTH  ;FILENAME LENGTH
820:    4866 A2 3D              LDX  #< NAME  ;FILENAME ADDRESS
                                              (LOW)
830:    4868 A0 03              LDY  #> NAME  ;FILENAME ADDRESS
                                              (HIGH)
840:    486A 20 F9 FD           JSR  FNPAR
850:    486D A9 02              LDA  #$02     ;LOGICAL FILE
                                              NUMBER
860:    486F A6 BA              LDX  $BA      ;DEVICE NUMBER
870:    4871 A0 00              LDY  #$00     ;SECONDARY ADDRESS
880:    4873 20 00 FE           JSR  FPAR
950:                    ;
960:    4876 A9 C0              LDA  #< BLOCK
970:    4878 85 02              STA  $02
980:    487A A9 02              LDA  #> BLOCK
990:    487C 85 03              STA  $03      ;START ADDRESS
                                              (HIGH BYTE)
1000:   487E A9 02              LDA  #$02     ;ZERO-PAGE ADDRESS
                                              OF START
1010:   4880 A2 FF              LDX  #< BLOCK+$3F  ;END ADDRESS
                                                   (LOW BYTE)
1020:   4882 A0 02              LDY  #> BLOCK+$3F  ;END ADDRESS
                                                   (HIGH BYTE)
1030:   4884 4C D8 FF           JMP  SV       ;SAVE SPRITE
1450:                   ;
1460:                   ;
1470:                   ;ESTABLISH WORK FIELD
1480:                   ;**********************
1490:                   ;
1500:   4887 A0 00     GRID       LDY  #$00     ;LINE COUNTER=0
1510:   4889 A9 0D     NO         LDA  #$0D     ;CARRIAGE RETURN
1520:   488B 20 D2 FF              JSR  CHROUT
```

```
1530:  488E 98              TYA
1540:  488F 48              PHA
1550:  4890 38              SEC
1570:  4893 90 0C           BCC  N1       ;SPACE WHEN Y<10
1580:  4895 A8              TAY
1590:  4896 E9 0A           SBC  #10
1600:  4898 90 04           BCC  N2       ;Y<20
1610:  489A A8              TAY
1620:  489B A9 32           LDA  #"2"
1630:  489D 2C              .BYTE$2C      ;SKIP NEXT COMMAND
1640:  489E A9 31     N2    LDA  #"1"
1650:  48A0 2C              .BYTE$2C      ;SKIP NEXT COMMAND
1660:  48A1 A9 20     N1    LDA  #$20     ;" " SPACE
1670:  48A3 20 D2 FF        JSR  CHROUT   ;OUTPUT
1680:  48A6 98              TYA           ;REST
1690:  48A7 09 30           ORA  #$30     ;ESTABLISH ASCII
1700:  48A9 20 D2 FF        JSR  CHROUT   ;OUTPUT
1710:  48AC 68              PLA
1720:  48AD A8              TAY
1730:  48AE A2 00           LDX  #$00
1740:  48B0 20 CE 48 N3     JSR  POINT    ;DRAW A POINT
1750:  48B3 A9 1D           LDA  #$1D     ;CURSOR RIGHT
1760:  48B5 20 D2 FF        JSR  CHROUT
1770:  48B8 E8              INX           ;NEXT POINT
1780:  48B9 E0 18           CPX  #24      ;24 POINTS PER
                                           LINE
1790:  48BB D0 F3           BNE  N3
1800:  48BD A9 A5           LDA  #$A5     ;LINE (CHR$(165))
1810:  48BF 20 D2 FF        JSR  CHROUT
1820:  48C2 C8              INY           ;NEXT LINE
1830:  48C3 C0 15           CPY  #21      ;21 LINES
1840:  48C5 D0 C2           BNE  N0
1850:  48C7 60              RTS
```

```
1860:                 ;
1870:                 ;DRAW A COORDINATE
1880:                 ;******************
1890:                 ;
1900:  48C8 AE 37 03 POINT2   LDX  XCOORD  ;CONTROLLED BY
                                            BASIC
1910:  48CB AC 36 03          LDY  YCOORD  ;X/Y-COORDINATE
1920:  48CE 8A       POINT    TXA
1930:  48CF 48                PHA
1940:  48D0 98                TYA
1950:  48D1 48                PHA          ;SAVE COORDINATES
1960:  48D2 85 02             STA  $02
1970:  48D4 0A                ASL  A
1980:  48D5 65 02             ADC  $02     ;Y-COORDINATE*3
1990:  48D7 85 02             STA  $02     ;AND SAVE
2000:  48D9 8A                TXA
2010:  48DA A0 FF             LDY  #$FF    ;COUNTER=0
2020:  48DC 38                SEC
2030:  48DD E9 08    P3       SBC  #$08
2040:  48DF C8                INY
2050:  48E0 B0 FB             BCS  P3      ;X-COORDINATE/8
2060:  48E2 69 08             ADC  #$08    ;REMAINDER
2070:  48E4 AA                TAX
2080:  48E5 98                TYA
2090:  48E6 18                CLC
2100:  48E7 65 02             ADC  $02     ;Y*3+INT(X/8)
2110:  48E9 A8                TAY
2120:  48EA A9 00             LDA  #$00
2130:  48EC 38                SEC          ;SET A BIT
2140:  48ED 6A       P0       ROR  A       ;FIND CORRECT BIT
2150:  48EE CA                DEX          ;DECREMENT
                                            REMAINDER
2160:  48EF 10 FC             BPL  P0
```

```
2170:  48F1 39 C0 02          AND  BLOCK,Y  ;ERASE OTHER BITS
                                             OF SPRITE BYTE
2180:  48F4 D0 03             BNE  P1       ;BIT (=POINT)SET?
2190:  48F6 A0 00             LDY  #$00     ;NO
2200:  48F8 2C                .BYTE$2C      ;BIT COMMAND
2210:  48F9 A0 06    P1       LDY  #$06     ;BIT SET!
2220:  48FB A2 06             LDX  #$06
2230:  48FD B9 0C 49 P2       LDA  PNTTAB,Y ;CHARACTER FROM
                                             POINT TABLE
2240:  4900 20 D2 FF          JSR  CHROUT
2250:  4903 C8                INY
2260:  4904 CA                DEX
2270:  4905 D0 F6             BNE  P2       ;NEXT CHARACTER
2280:  4907 68                PLA
2290:  4908 A8                TAY
2300:  4909 68                PLA
2310:  490A AA                TAX           ;REPEAT
                                             COORDINATES
2320:  490B 60                RTS
2330:                 ;
2340:                 ;CHARACTERS FOR A COORDINATE
2350:                 ;*****************************
2360:                 ;
2370:  490C 92 1F 6F PNTTAB   .BYTE146,031,111,031,146,157
2380:  4912 12 1C 20          .BYTE018,028,032,031,146,157
2390:                 ;
2400:                 ;IDENTIFY KEY AS COMMAND
2410:                 ;************************
2420:                 ;
2430:  4918 AD 35 03 CMD1D    LDA  KEY      ;KEY CODE
2440:  491B A2 00             LDX  #$00     ;COMMAND CODE
2450:  491D A0 1C             LDY  #$1C     ;27-1 COMMANDS
                                             (COUNTER)
```

```
2460:  491F E8        B1      INX             ;INCREMENT COMMAND
                                               CODE
2470:  4920 DD 2B 49          CMP   CMDTAB-1,X  ;COMPARE KEY
                                                   WITH TABLE
2480:  4923 F0 03             BEQ   B2          ;FOUND
2490:  4925 88                DEY               ;NEXT COMMAND
2500:  4926 D0 F7             BNE   B1          ;NOT DONE
2510:  4928 8E 35 03 B2       STX   KEY         ;COMMAND CODE AS
                                                   OUTPUT
2520:  492B 60                RTS               ;BACK TO BASIC
2530:                 ;
2540:                 ;COMMAND KEY TABLE
2550:                 ;******************
2560:                 ;
2570:                 ;  W/CRSR-RT/Q/CRSR-LT/A/CRSR-DN
2580:  492C 57 1D 51 CMDTAB   .BYTE087,029,081,157,065,017
2590:                 ;   2/CRSR-UP/ F1/ F2/ F3/ F4
2600:  4932 32 91 85          .BYTE050,145, 133,137,134,138
2610:                 ;   F5/ F6/ F7/ F8/ L / G
2620:  4938 87 8B 88          .BYTE135,139,136,140,076,071
2630:                 ;   I/CTRL.S/CTRL.G/CRTL.X/B
2640:  493E 49 13 07          .BYTE073,019, 007, 024,066
2650:                 ;   C / F / V
2660:  4943 43 46 56          .BYTE067,070,086
2670:                 ;
2680:                 ;DISK CATALOG
2690:                 ;*************
2700:                 ;
2710:  4946 A9 24     CATALG  LDA   #$24      ;"$" AS FILENAME
2720:  4948 85 02             STA   $02
2730:  494A A9 01             LDA   #$01
2740:  494C A2 02             LDX   #$02
2750:  494E A0 00             LDY   #$00
```

```
2760:  4950 20 F9 FD        JSR  FNPAR
2770:  4953 A9 02           LDA  #$02
2780:  4955 A6 BA           LDX  $BA      ;DEVICE ADDRESS
2790:  4957 A0 00           LDY  #$00
2800:  4959 20 00 FE        JSR  FPAR
2810:  495C A9 00           LDA  #$00     ;LOAD/VERIFY FLAG
2820:  495E A2 00           LDX  #$00
2830:  4960 A0 40           LDY  #$40     ;START ADDRESS
2840:  4962 86 5F           STX  $5F
2850:  4964 84 60           STY  $60
2860:  4966 20 D5 FF        JSR  LD       ;LOAD CATALOG AS
                                           BASIC PROGRAM
2670:  4969 A5 5F           LDA  $5F      ;SIMULATE
2880:  496B A4 60           LDY  $60      ;BASIC PROGRAM
                                           START ADDRESS
2890:  496D 20 37 A5        JSR  $A537    ;ADD BASIC LINES
2900:  4970 AD 00 03        LDA  $0300
2910:  4973 48              PHA
2920:  4974 AD 01 03        LDA  $0301    ;ORIGINAL WARM
                                           START VECTOR
2930:  4977 48              PHA           ;SAVE
2940:  4978 A9 3D           LDA  #$3D
2950:  497A 8D 00 03        STA  $0300
2960:  497D A9 E3           LDA  #$E3
2970:  497F 8D 01 03        STA  $0301    ;AND SET TO RTS
2980:  4982 20 C3 A6        JSR  $A6C3    ;EXECUTE LIST
                                           COMMAND
2990:  4985 68              PLA
3000:  4986 8D 01 03        STA  $0301
3010:  4989 68              PLA
3020:  498A 8D 00 03        STA  $0300    ;GET OLD VECTOR
3030:  498D 60              RTS           ;BACK TO BASIC
3040:
```

```
3050:                 ;INVERT SPRITE
3060:                 ;**************
3070:                 ;
3080:  498E A2 3E     INVERS   LDX  #62      ;62+1 BYTES
3090:  4990 BD C0 02  IN1      LDA  BLOCK,X  ;GET BYTE
3100:  4993 49 FF              EOR  #$FF     ;INVERT
3110:  4995 9D C0 02           STA  BLOCK,X  ;WRITE BACK
3120:  4998 CA                 DEX
3130:  4999 10 F5              BPL  IN1      ;NEXT BYTE
3140:  499B 60                 RTS
3150:                 ;
3160:                 ;SHIFT SPRITE
3170:                 ;**************
3180:                 ;
3190:  499C A2 00     RIGHT    LDX  #$00     ;ABSOLUTE BYTE
                                              COUNTER
3200:  499E A0 03     RT1      LDY  #$03     ;BYTE COUNTER LINE
3210:  49A0 18                 CLC
3220:  49A1 7E C0 02  RT2      ROR  BLOCK,X  ;SHIFT
3230:  49A4 E8                 INX
3240:  49A5 88                 DEY
3250:  49A6 D0 F9              BNE  RT2
3260:  49A8 A9 00              LDA  #$00
3270:  49AA 6A                 ROR  A        ;LAST BIT IN ACC
3280:  49AB 1D BD 02           ORA  BLOCK-3,X  ;AND SET TO
                                                START
3290:  49AE 9D BD 02           STA  BLOCK-3,X
3300:  49B1 E0 3F              CPX  #63
3310:  49B3 D0 E9              BNE  RT1
3320:  49B5 60                 RTS           ;DONE
3330:                 ;
3340:                 ;
3350:  49B6 A2 3F     LE       LDX  #63      ;ABSOLUTE BYTE
```

```
                                          COUNTER
3360:  49B8 A0 03    LT1      LDY  #$03    ;BYTE COUNTER IN
                                          LINE
3370:  49BA 18                CLC
3380:  49BB 3E BF 02 LT2      ROL  BLOCK-1,X  ;SHIFT
3390:  49BE CA                DEX
3400:  49BF 88                DEY
3410:  49C0 D0 F9             BNE  LT2
3420:  49C2 A9 00             LDA  #$00
3430:  49C4 2A                ROL  A        ;LAST BYTE IN ACC
3440:  49C5 1D C2 02          ORA  BLOCK+2,X  ;AND SET TO END
3450:  49C8 9D C2 02          STA  BLOCK+2,X
3460:  49CB 8A                TXA
3470:  49CC D0 EA             BNE  LT1
3480:  49CE 60                RTS          ;DONE
3490:                ;
3500:                ;
3510:  49CF A2 3E    UP       LDX  #62     ;ABSOLUTE BYTE
                                          COUNTER
3520:  49D1 86 02             STX  $02
3530:  49D3 A0 15    UP1      LDY  #21     ;LINE COUNTER
3540:  49D5 84 03             STY  $03
3550:  49D7 A6 02             LDX  $02
3560:  49D9 BD 84 02          LDA  BLOCK-60,X  ;GET FIRST
                                               BYTE
3570:  49DC 48        UP2     PHA          ;IN TEMP STORAGE 1
3580:  49DD BD C0 02          LDA  BLOCK,X  ;GET BYTE
3590:  49E0 A8                TAY          ;IN TEMP STORAGE 2
3600:  49E1 68                PLA          ;GET TEMP
                                          STORAGE 1
3610:  49E2 9D C0 02          STA  BLOCK,X  ;PUT IN BYTE
3620:  49E5 98                TYA          ;GET TEMP
                                          STORAGE 2
```

```
3630:  49E6 CA              DEX
3640:  49E7 CA              DEX
3650:  49E8 CA              DEX          ;ABOVE BYTE
3660:  49E9 C6 03           DEC  $03     ;NEXT LINE
3670:  49EB D0 EF           BNE  UP2
3680:  49ED A6 02           LDX  $02
3690:  49EF CA              DEX
3700:  49F0 86 02           STX  $02     ;NEXT COLUMN
3710:  49F2 E0 3B           CPX  #59
3720:  49F4 D0 DD           BNE  UP1
3730:  49F6 60              RTS
3740:                ;
3750:                ;
3760:  49F7 A2 02    DOWN   LDX  #02     ;ABSOLUTE BYTE
                                          COUNTER
3770:  49F9 86 02           STX  $02
3780:  49FB A0 15    DN1    LDY  #21     ;LINE COUNTER
3790:  49FD 84 03           STY  $03
3800:  49FF A6 02           LDX  $02
3810:  4A01 BD FC 02        LDA  BLOCK+60,X  ;GET LAST BYTE
3820:  4A04 48       DN2    PHA          ;IN TEMP STORAGE 1
3830:  4A05 BD C0 02        LDA  BLOCK,X  ;GET BYTE
3840:  4A08 A8              TAY          ;IN TEMP STORAGE 2
3850:  4A09 68              PLA          ;GET TEMP
                                          STORAGE 1
3860:  4A0A 9D C0 02        STA  BLOCK,X  ;PUT IN BYTE
3870:  4A0D 98              TYA          ;GET TEMP
                                          STORAGE 2
3880:  4A0E E8              INX
3890:  4A0F E8              INX
3900:  4A10 E8              INX          ;ABOVE BYTE
3910:  4A11 C6 03           DEC  $03     ;NEXT LINE
3920:  4A13 D0 EF           BNE  DN2
```

```
3930:  4A15 C6 02            DEC  $02     ;NEXT COLUMN
3940:  4A17 10 E2            BPL  DN1
3950:  4A19 60               RTS
]4800-4A1A
NO ERRORS
```

By the way, if you do not have the time or the desire to enter this program by hand, a diskette containing all of the programs in this book is available as an option.

Assuming that you have entered it without error, the sprite editor looks like this:

```
                        TITLE
      ========================================
                                 color assign-
                                 ment field/
                                 secondary
                                 operations
             24x21 field
                                 Original
                                 field 1
                                 Original
                                 field 2
```

This layout gives you the necessary information about the features of your sprites.

24x21 field:

In this field you can create your sprite with the help of the various functions available (see below). The digits at the left edge of the screen give the y-coordinate, the row, and the digits at the top edge give the x-coordinate or the corresponding bit in the current byte of the sprite

storage. Each set bit (point) is shown as red square.

Original fields:

These fields show the actual sprite in the selected color and size in order to show you what the actual sprite looks like. Field 1 shows the current design of the object in normal size while field 2 shows it expanded in the x and/or y directions.

Color assignment field:

This field shows you the basic colors assigned to the sprite; they can be changed as desired (see below).

Secondary operations field:

This field shows you other information about the sprite.

-- Commands --

To perform the following commands press the appropriate key without <RETURN>.

- B -

Gives a short review of these commands. Pressing a key returns you to the spriteinput.

- <crsr> -

The field cursor can be moved by the 4 cursor keys (up/down, right/left).

- 2/Q/W/A -

For easier operation, these keys also act as cursor movement keys. The movement direction results from the corresponding

keyboard position.

- f1/f3 -
With these keys you can draw a point on the 24x21 field in the color assigned to the key (f1 is the background color, f3 is the sprite color).

- F - COLOR
F lets you assign colors to the keys f1/f3 (basic colors). The points drawn using these keys change color in the original field.
Note: Key f1/2 (= color 0) stands for the background color!

- G - SIZE
Pressing this key allows you to change the size of the sprite found in original field 2.

- I - INVERT
This command inverts a sprite (all set points become reset points and reset points become set points).

- V - SHIFT/MOVE
You can shift your drawing within the field as desired. So that no information is lost, the edges wrap around to the other side.
After pressing V, press R, L, U, or D for right, left, up, or down (you can see the results directly in the original fields).

- L - ERASE
Erases the entire sprite.

- C - DIRECTORY/CATALOG

Displays the contents of the disk directory on the screen. Pressing the <CTRL> key slows down the listing. Any other key returns you to sprite input.

- <CTRL>S / <CTRL>G - SAVE/GET

Saving (<CTRL>S) or loading (<CTRL>G) a sprite:

Enter the filename and the device number, separated by a semicolon. If the device number is absent, the last-used device is assumed. Now press <RETURN>.

If you pressed either of these key combinations by accident, simply press <RETURN> without typing anything else to return to the sprite input (this applies to all secondary operations such as V, F,...). When saving, a program file is created which can be read into memory directly, or treated as a file as mentioned above.

- <CTRL>X - EXIT

If you are done and want to exit the program, answer the question "EXIT??" after this command with Y or YES, erase the rest of the line and press <RETURN>. Any other input brings you back to sprite input.

4.3.2 Programming using the sprite features

Section 3.5 provides the theoretical knowledge about how a sprite is defined, turned on, and its properties changed. Now it is time to actually use this information. We'll illustrate these with a small sample program which uses almost all of the variations possible and even provides a look at the typical uses of sprites in animation. Animation is discussed in more detail later in the book.

Feel free to change this sample program as much as you like. You should be careful when changing the addresses of the POKE commands--the first parameter. Try to determine what effect the change will have before running it. If a computer crash and reloading does not bother you, then this warning does not apply. First the program itself:

```
1000 REM **********************
1010 REM **                  **
1020 REM **  SPRITE EXAMPLE  **
1030 REM **                  **
1040 REM **********************
1050 REM
1060 V = 53248 : REM BASE ADDRESS OF VIDEO CONTROLLER
1070 POKE V+32,0 : POKE V+33,0 : REM BORDER AND BACKGROUND
     = BLACK
1080 REM
1090 REM SPRITE DEFINITION IN BLOCK 13:
1100 REM ******************************
1110 A1 = 13*64 : REM ADDRESS OF BLOCK 13
1120 FOR X=0 TO 62
1130 READ DT : POKE A1+X,DT : REM READ DATA AND POKE INTO
     BLOCK 13
1140 NEXT X
1150 REM DATA FOR FIRST SPRITE:
1160 DATA 000,007,000
1170 DATA 056,013,128
1180 DATA 025,031,224
1190 DATA 126,031,204
1200 DATA 025,012,252
1210 DATA 056,007,000
1220 DATA 000,014,000
1230 DATA 000,014,000
```

```
1240 DATA 000,015,128
1250 DATA 000,014,192
1260 DATA 000,030,096
1270 DATA 000,062,048
1280 DATA 000,046,000
1290 DATA 000,079,000
1300 DATA 000,155,128
1310 DATA 001,025,192
1320 DATA 002,024,096
1330 DATA 003,024,096
1340 DATA 127,024,120
1350 DATA 188,024,000
1360 DATA 036,030,000
1370 REM
1380 REM SPRITE DEFINITION IN BLOCK 14:
1390 REM ******************************
1400 A2 = 14*64 : REM ADDRESS OF BLOCK 14
1410 FOR X=0 TO 62
1420 READ DT : POKE A2+X,DT : REM READ DATA AND POKE INTO
     BLOCK 14
1430 NEXT X
1440 REM DATA FOR SECOND SPRITE:
1450 DATA 000,003,010
1460 DATA 000,013,132
1470 DATA 058,031,224
1480 DATA 124,031,170
1490 DATA 058,015,250
1500 DATA 000,007,000
1510 DATA 000,014,000
1520 DATA 000,014,000
1530 DATA 000,014,000
1540 DATA 000,015,000
1550 DATA 000,015,128
```

```
1560 DATA 000,014,160
1570 DATA 000,030,000
1580 DATA 000,047,000
1590 DATA 000,077,128
1600 DATA 000,141,128
1610 DATA 001,057,128
1620 DATA 003,097,128
1630 DATA 127,113,128
1640 DATA 220,025,128
1650 DATA 036,001,224
1660 POKE 2046, A1/64 : REM FIRST BLOCK 13 FOR SPRITE 6
1670 POKE V+21, 2^6 : REM TURN SPRITE 6 ON
1680 POKE V+39+6, 1 : REM COLOR OF SPRITE 6 : WHITE
1690 POKE V+27,   0 : REM SPRITE IN FRONT OF BACKGROUND
1700 POKE V+28,   0 : REM NORMAL COLOR SPRITE
1710 POKE V+23, 2^6 : REM EXPAND SPRITE 6 IN Y-DIRECTION
1720 POKE V+29, 2^6 : REM EXPAND SPRITE 6 IN X-DIRECTION
1730 POKE V+2*6+1, 101: REM SET SPRITE 6 Y-COORDINATE
1735 POKE V+2*6  , 100: REM SPRITE X-COORDINATE FOR
     DEMONSTRATION
1740 POKE V+16,   0 : REM ERASE HIGH BYTE OF X-COORDINATE
1745 REM
1750 REM
1760 PRINT CHR$ (30) CHR$(147) : REM GREEN + ERASE SCREEN
1770 FOR X=1 TO 10
1780 PRINT : REM 10 BLANK LINES
1790 NEXT X
1800 PRINT CHR$(18); : REM RVS ON
1810 FOR X=1 TO 40
1820 PRINT " "; : REM 40 REVERSE GREEN CHARACTERS
1830 NEXT X
1840 REM
1850 REM
```

```
1860 REM ***********
1870 REM **        **
1880 REM **  RUN   **
1890 REM **        **
1900 REM ***********
1910 REM
1920 S = 10 : REM SPEED
1930 C = 1 : REM COLOR START
1940 FOR X=1 TO 400 STEP S
1950 C = C+.3: REM CHANGE COLOR
1960 IF C=16 THEN C=1
1970 REM ----------------
1980 POKE V+39+6, C : REM CHANGE COLOR
1990 POKE 2046, A1/64 : REM SPRITE 6 TO BLOCK 13
2000 CD=X:IFX>255THENCD=X-256:POKEV+16,2^6:GOTO2020:
     REM SET HIGH BIT OF X-COORD
2010 POKE V+16, 0 : REM ERASE HIGH BIT OF X-COORD
2020 POKE V+2*6, CD : REM MOVE SPRITE
2030 FOR Y=1 TO 40 : NEXT Y : REM DELAY LOOP
2040 REM ----------------
2050 POKE 2046, A2/64 : REM SPRITE 6 TO BLOCK 14
2060 CD=X+S/2:IFCD>255THEN CD=CD-256:POKE V+16,2^6:GOTO2080:
     REM HIGH BIT X-COORD
2070 POKE V+16, 0 : REM X-COORDINATE - ERASE HIGH BIT
2080 POKE V+2*6, CD : REM MOVE SPRITE
2090 FOR Y=1 TO 40 : NEXT Y : REM DELAY LOOP
2100 REM ----------------
2110 NEXT X
```

In the first line (line 1060), the variable V is again defined. This definition is executed before every sprite or graphics program. The border and background colors are changed to black.

Here is a discussion of the actual work:

a) Sprite definition:

You are already familiar with the routine at line 1100: The 63 data bytes which define a sprite are read from DATA statements and POKEd into memory. In the first case, the definition in block 13 is written into block 14 (at line 1400). The starting addresses of these two blocks are calculated by multiplying by 64 and placed in the variables A1 and A2.

So far we have defined two sprites, which, as you will see, look quite similar. In both cases it is an old man leading his dog and accompanied by a bird. The two sprites show two phases of our running man, the smoking pipe, and the flying bird (you can see the two separate sprites if you insert a STOP in the lines 2030 and 2090). Our intention here is not to see both sprites at the same time on the screen, but to make small changes to each sprite and show them in succession in order to develop movement through the changes.

b) Block vector:

We use the design for sprite 6 and first determine the block from which the sprite definition will come. At the start, block 13 will yield the definition. For this reason, we must POKE a 13 into the 6th of the last 8 bytes of the video RAM (the sprite definition vectors) which is done in line 1660.

c) Turning on:

We still have not yet done everything required to make the sprite visible. We must turn it on. This is done in line 1670 of our program. Here register 21 of the VIC is POKEd with the value $2^6 = 64$. This sets bit 6 of the register, instructing the VIC to turn on sprite 6.

d) Color:

In spite of all this, there is quite likely not a whole lot to see. This is because most of it lies outside the screen window. We will worry about this later. First we want to set the color of the sprites. Registers 39-46 of the VIC are used for setting sprite colors. The value 1, for white, is placed into the seventh of these 8 registers (which controls sprite 6; register number 39+6 = 45).

e) Priority:

In line 1690 we specify whether our sprite should appear in front of or behind the background characters. In other words: Will our sprite be covered by the green line to be drawn later, or will it cover the green line? We have chosen the latter case and cleared the appropriate bit in VIC register 27. This situation can, however, be changed with POKE V+27, 2^6. Try it once! See also section 3.5.4.3.

f) Multi-color:

We have planned and developed our sprite as a normal-color sprite. Therefore we must turn this mode on. This was decided very early, during the actual organization.

g) Expansion:

We must make a few more decisions before we are through: Do we want our object in the the original size or displayed expanded in either direction? In order to be better able to see the detail of the two sprites, we chose to expand the sprites in both the x and y directions and set the corresponding bits for sprite 6 in registers 23 and 29 of the VIC (see section 3.5.4.2). This is done in lines 1710/1720. The whole thing looks quite amusing in the normal size (achieved by clearing the values 2^6 = 0). Try also an expansion in only the x or y directions!

h) Positioning:

Finally we are nearing the end! Now we can make our sprite visible on the screen. To do so, we simply set the coordinates at which the object is to appear. Later we will change this position. As you learned in section 3.5.4.1, registers 0-16 are used to set the position. Register 2*6 = 12 is the low-byte of the x-coordinate for sprite 6, 2*6+1 = 13 is the y-coordinate, and bit 6 of register 16 is the high bit of the x-coordinate. These registers are set in lines 1730-1740.

Now our sprite is visible. We have done everything necessary in order to define a sprite. All of these steps are necessary in all sprite programs. You can see that despite the power of sprites, they are not very easy to work with. In time you will discover routines and programming methods which will ease this process or allow you to dispense with some steps.

Now back to our discussion.

After clearing the screen and drawing a green line, a loop starts which is executed a total of 400 times, ranging in values from 1 to 400 (line 1940). S determines the step size per loop execution.

A number of things happen in this loop:

First, the variable C is incremented by 0.3 (line 1950). C is used to change the color of the sprite within the loop (line 1980). Since only whole numbers (integers) are used for color, the color only changes when the integer portion of C changes. This happen approximately every third loop.

Next the main work of the program is performed: the movement and definition change. First the definition change:

As you know, memory location 2046 is used as the pointer to the block in which our sprite definition is found. Earlier we had placed the two objects in memory--one in block 13 and the other in block 14. In order to make it appear as if the sprites are animated, we must alternate between these two definitions. At this point (line 1990), the value 13 is POKEd into 2046 in order to make the vector point to the first sprite. Later on (line 2050) we switch to definition 14 and the second sprite is visible.

In order to make the motion of our object appear continual, we move it one or more points in the x-direction (the y-coordinate remains constant). The variable X is used for this. It is first saved in the temporary variable CD (for CoorDinate) at line 2000. Then it is checked to see if it is greater than 255. If this is the case, one byte no longer suffices to hold the x-coordinate and the high-bit in register 16 is set. Naturally, CD is decremented by 256 in order to avoid an ILLEGAL QUANTITY ERROR. The low-byte is also set and the sprite is moved (line 2020). In the third part of the loop (after a delay loop) this procedure is

executed one more time (lines 2060 ff.)

The above example should be helpful in explaining some of the techniques used in sprite programming. Other possibilities are not covered here: multi-color, collision handling, and working with several sprites simultaneously.

These topics will be taken up in the next section. In the meantime, you should know that in multi-color a total of 4 colors may be used per sprite, the x-resolution is cut in half, the size of the entire sprite remains the same, and the point size is increased by a factor of 2. In memory, a point is determined by two bits which denote the register from which the color will be taken. These registers are the multi-color registers 0 and 1 (VIC registers 37 & 38) and the registers which are different for each sprite, registers 39-46. If the two bits are zero, this part of the sprite is transparent (see section 3.5.3).

Two different types of collisions can be determined: sprite-sprite collisions (VIC register 30) and sprite-background collisions (VIC register 31). In both registers, a collision between the appropriate types sets the bit(s) corresponding to the sprite(s) involved. If you do not recognize these things, please read section 3.5.4.4. In our case, we will test to see if two sprites have collided.

Here too we have included an example program for illustrating what will be said below:

```
100 REM ************************
110 REM **                    **
120 REM **  SPRITE COLLISION  **
130 REM **    (MULTICOLOR)    **
140 REM ************************
150 REM
```

```
160 V=53248 : REM VIC BASE ADDRESS
170 A1 = 11*64 : REM BLOCK 11
180 FOR X=0 TO 62
190 READ DT : POKE A1+X, DT : REM READ AND POKE DEFINITION
200 NEXT X
210 POKE 2040, 11 : POKE 2042, 11 : REM BLOCK POINTERS OF
    SPRITES 0 AND 2 TO 11
220 POKE V+28, 2^0 OR 2^2 : REM SPRITE 0 AND 2 TO MULTI-COLOR
230 POKE V+27, 2^0 : REM ONLY SPRITE 2 HAS PRIORITY OVER
    BACKGROUND
240 POKE V+29, 2^0 OR 2^2 : POKE V+23, 2^0 OR 2^2 : REM BOTH
    LARGER
250 POKE V+21, 2^0 OR 2^2 : REM BOTH SPRITES ON
260 POKE V+37, 7 : REM SPRITE MULTICOLOR 0 = YELLOW
270 POKE V+38, 6 : REM SPRITE MULTICOLOR 1 = BLUE
280 POKE V+39, 5 : POKE V+41, 8 : REM INDIVIDUAL COLORS FOR
    SPRITES 0 AND 2
290 POKE V+16,0 : REM ERASE HIGH BIT OF X-COORD
300 POKE V+1,100 : POKE V+5,100 : REM SET Y-COORDINATES
310 X2 = 255 : REM START X-COORDINATES OF SPRITE 2
320 POKE V+0,0 : POKE V+4,X2 : REM SET X-COORDINATE
330 POKE V+30,0 : REM RESET COLLISION
340 FOR X0=1 TO 255 : REM X-COORDINATE SPRITE 0
350 X2 = X2-1 : REM DECREMENT X-COORDINATE OF SPRITE 2
360 POKE V+0, X0 : POKE V+4, X2 : REM X-COORDINATES FOR
    SPRITES 0/2 (LOW BYTES)
370 POKE V+1, 40*SIN(X0/30)+100 : REM MOVE IN SINE CURVE
380 POKE V+5, 40*COS(X0/30)+100 : REM MOVE IN COSINE CURVE
390 IF PEEK(V+30)=(2^0 OR 2^2) THEN GOTO 420
400 NEXT X0
410 REM
420 REM COLLISION ROUTINE
430 REM
```

```
440 FOR X=1 TO 255
450 POKE V+39,X : POKE V+41,X : REM CHANGE SPRITE COLOR
460 NEXT X
470 POKE V+0, 0 : REM SEPARATE SPRITES
480 POKE V+30,0 : REM RESET COLLISION
490 REM
500 REM SPRITE DATA
510 REM
520 DATA 008,000,128
530 DATA 010,002,128
540 DATA 002,138,000
550 DATA 064,136,005
560 DATA 080,168,021
570 DATA 084,032,093
580 DATA 117,033,125
590 DATA 125,101,253
600 DATA 127,103,253
610 DATA 123,103,173
620 DATA 123,103,173
630 DATA 123,103,173
640 DATA 123,103,173
650 DATA 123,103,173
660 DATA 123,103,173
670 DATA 123,103,173
680 DATA 123,103,173
690 DATA 127,103,253
700 DATA 095,101,253
710 DATA 021,097,117
720 DATA 005,032,084
```

First, the usual formalities for initializing the sprites are performed. In this program we will be working with only one sprite definition, which is placed in block 11

(lines 170-200). Two sprites, however, will appear on the screen at the same time. We have chosen sprites 0 and 2. Both sprites will have the same pattern. This is realized in line 210 in which both vectors for sprites 0 and 2 are directed to block 11. Both sprites therefore get their definitions from the same block. Despite this, all of the other parameters can be set and changed independently. This is done in line 230, for example, where the two different priorities over the background characters are set, and in 280 where the sprites are given different colors. As we said before, we want to work with multi-color sprites. The definition given in the DATA lines was therefore developed with this in mind. We must inform the VIC of this, as is done in line 220. Here we come across a previously-unencountered difficulty. Up until now, we have had to set only a single bit. Here, however, we want to set sprite 0 as well as sprite 2 as a multi-color sprite. This is done by combining the two individual values 2^0 (bit 0 set) and 2^2 (bit 2 set) with the OR operation. If you still encounter difficulties, review Chapter 2.

In lines 260-280 we see something new--the various colors for the multi-color sprites in the proper registers. Note that only color 3 (color channel 3) can be different for each sprite.

Our sprites are already turned on but you probably cannot see them on the screen yet. We'll now place them on the screen. This is done in lines 290-320. This procedure is not really necessary because all parameters (except the high-bit in register 16) are also set in the following movement loop.

Here we encounter another small difficulty. When the computer is first turned on, all sprite coordinates are set to zero. Since these coordinates are the same, the sprites

have collided by definition. And since we want to be notified when sprite collisions occurs, the hardware detects a collision immediately. To avoid this inadvertent collision, we must separate the sprites. After this--and this is very important--we reset the sprite-sprite collision register by writing the value 0 to it because its contents remain (actually only the one bit), until they are cleared, even if the collision is no longer is progress.

Now we can let it fly. Here, as in the previous example, the sprites are moved from within a loop. The loop counter (X0) is used as an increasing value for determining the x-coordinate of sprite 0. A value ranging from 255 down to 1 (X2) determines the x-coordinate of sprite 2 (line 360). In order to let sprite 0 "fly" along a sine curve and sprite 2 along a cosine curve, the y-coordinates are calculated by the appropriate formulas in lines 360 and 370. You might like to change a few of the parameters here (40 is the amplitude of the wave, 30 is the length of the curve and 100 shifts the entire curve in the y-direction).

Now we come to a very interesting part--the collision detection. In line 390, bits 0 and 2 of register 30, the sprite-sprite collision register, are checked to see if they are set. If this is the case, the program branches to the collision routine at line 440. Here a small color effect is created, the two colliding sprites are separated, and the collision register is cleared. We must separate the sprites or they will continue to cause a collision interrupt. Now we have all that is necessary to service collisions.

4.4 Character set programming

One topic which is not often discussed is modifying the character set. Doing so opens up a number of interesting possibilities. Many games use this capability to create graphics. Most of the effects similar to high-resolution graphics are available using character graphics at a speed increase of 8-9 times.

The Commodore 64 becomes even more flexible with this capability to create custom character sets. It surpasses most computers which must to have their character sets changed for use in different countries or applications. The '64 can display not only an American character set on your screen, but German, Swedish, Russian, or even Japanese. Text fonts can also be changed in style as is done in our word-processing package TEXTOMAT. As you can imagine, this capability can be very powerful. For this reason we present the techniques of altering the character set. This is not as simple as sprite programming, for instance, and cannot be done without machine language unless you have a BASIC extension program. It is recommended that you read and understand section 3.6 before proceeding with this section.

As you learned in section 3.6, each character definition consists of 8 bytes of 8 bits. This gives an 8x8 point matrix. 512 characters may be defined at the same time, although only 256 different characters may be displayed on the screen simultaneously. The entire set of 512 characters has a length of 4K and normally lies at $D000-$DFFF (53248-57343) in the character generator ROM. It is movable by changing register 24 of the VIC and bits 0/1 of CIA 2 (addresses: $D018 = 53272 and $DD00 = 56576).

Two cases must be distinguished when creating a character set. In the first case, only a few parts of the set will be changed (special characters). In the second case the entire set is changed. We will begin with the former case:

4.4.1 Modifying characters

If we want to replace a few characters in the character generator with some of our own, we must first copy the contents of the ROM to an unused section of RAM and then change the starting address in the VIC as explained in section 3.3.2. Then we can go in and change the individual characters.

What appears a very simple task requires some background knowledge and programming skill. As a matter of principle, the character generator cannot be read or copied from BASIC. Here's why. The location at which the character generator is stored ($D000-$DD00 or 53248-57343) also contains the input/output ports (I/O area) which must first be turned on by changing register 1 of the CPU (see section 3.5). Doing this switches out the I/O area which is used by the interrupt routine. The computer will crash if this is done from BASIC. This can be prevented in machine language by disabling the interrupts. For this reason we present a machine language program which will make a copy of the character set.

```
15:   C800                .OPT P1
20:                ;
30:                ;MOVE CHARACTER SET
40:                ;******************
50:                ;
60:   C800                *=   $C800   ;START ADDRESS
70:   D000          V     =    53248   ;BASE ADDRESS OF
                                        VIDEO CONTROLLER
80:   D000          SET   =    53248   ;BASE ADDRESS FOR
                                        CHARACTER SET
90:   3000          DEST  =    $3000   ;BASE ADDRESS FOR
                                        COPY
100:  C800 78       START SEI          ;DISABLE
                                        INTERRUPTS
110:  C801 A5 01          LDA  $01     ;CPY REG. 1
120:  C803 48             PHA          ;RETURN
130:  C804 29 FB          AND  #%11111011 ;ERASE BIT 2
140:  C806 85 01          STA  $01     ;MAKE CHARACTER
                                        GENERATOR READABLE
150:  C808 A9 D0          LDA  #>SET
160:  C80A 85 03          STA  $03     ;SOURCE ADDRESS
                                        HIGH BYTE
170:  C80C A9 30          LDA  #>DEST
180:  C80E 85 05          STA  $05     ;DESTINATION
                                        ADDRESS HIGH BYTE
190:  C810 A0 00          LDY  #$00
200:  C812 84 02          STY  $02
210:  C814 84 04          STY  $04     ;LOW BYTES = 00
220:  C816 A2 20          LDX  #$20    ;COUNTER FOR 4K
                                        BYTES
230:  C818 B1 02    COPY  LDA  ($02),Y ;LOW BYTE
240:  C81A 91 04          STA  ($04),Y ;AND COPY
250:  C81C C8             INY
```

```
260:   C81D D0 F9            BNE  COPY     ;256 TIMES
270:   C81F E6 03            INC  $03
280:   C821 E6 05            INC  $05      ;INCREMENT HIGH
                                            BYTES
290:   C823 CA               DEX
300:   C824 D0 F2            BNE  COPY     ;COPY 4K
310:   C826 68               PLA
320:   C827 85 01            STA  $01      ;CHARACTER GEN-
                                            ERATOR OFF /
                                            I/O ON
330:   C829 AD 18 D0         LDA  V+24     ;CHARACTER GEN-
                                            ERATOR ADDRESS
340:   C82C 29 F1            AND  #%11110001
350:   C82E 09 0C            ORA  #%00001100
360:   C830 8D 18 D0         STA  V+24     ;SET TO $3000
370:   C833 58               CLI           ;INTERUPT ENABLE
380:   C834 60               RTS           ;BACK TO BASIC
]C800-C835
NO ERRORS
```

And the BASIC loader for this program:

```
100 FOR I = 51200 TO 51252
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 120,165,  1, 72, 41,251,133,  1,169,208,133,  3
130 DATA 169, 48,133,  5,160,  0,132,  2,132,  4,162, 32
140 DATA 177,  2,145,  4,200,208,249,230,  3,230,  5,202
150 DATA 208,242,104,133,  1,173, 24,208, 41,241,  9, 12
160 DATA 141, 24,208, 88, 96
170 IF S <> 5884 THEN PRINT "ERROR IN DATA !!" : END
180 PRINT "OK"
```

If you want to change parts of the character set, you need only load this loader program, enter RUN and SYS 51200. The original character set is moved (you can, if you have a machine language program, also enter the machine language program directly, save it, and later load it with LOAD "name",8,1).

Let's not forget that the reason for all of this was to change various characters in the character generator. Now on to it.

From section 3.6 we are already acquainted with the 8x8 organization (4x8 with multi-color) of a character and its layout in 8 bytes. Before changing a character pattern, we must design the new character. One way to do this is by using a character development template. You can find one in section 6.7. Each row on this template corresponds to one byte in the character generator. Each set bit in the definition corresponds to a set point on the screen. As an example, we have created the following drawing:

Bit:	7	6	5	4	3	2	1	0
Byte 0:	.	.	.	*	*	*	.	.
Byte 1:	.	*	*	*	.	.	.	.
Byte 2:	*	*	.	.	*	*	*	*
Byte 3:	*	*	.	.	.	.	.	.
Byte 4:	*	*	.	.	*	*	*	*
Byte 5:	.	*	*	*	.	.	.	.
Byte 6:	.	.	.	*	*	*	.	.
Byte 7:	.	.	.	.	.	.	.	.

We get the following 8 bytes:

Byte 0: %0001 1100 = $1C = 028
Byte 1: %0111 0000 = $70 = 112

```
Byte 2: %1100 1111 = $CF = 207
Byte 3: %1100 0000 = $C0 = 192
Byte 4: %1100 1111 = $CF = 207
Byte 5: %0111 0000 = $70 = 112
Byte 6: %0001 1100 = $1C = 028
Byte 7: %0000 0000 = $00 = 000
```

Now the next question crops up: Which character will the new one replace, and to which of the 4 character sets (see section 3.6) should it belong? In our case we want to put this definition in place of the normal pound character which is obtainable in the upper case/graphics mode.

We must determine where the corresponding 8 bytes lie within the character generator. We use the formula given in Chapter 3:

```
address = base address + 8 * screen code
```

The base address of the character generator depends on the range to which we have moved it. Because we are in the addresses $3000-$3FFF (12288-16383) and we want to change the upper case/graphics mode, the definition of this character lies somewhere in the range $3000-$37FF (12288-14335). Now we need to know the screen code. We can find this from the screen code table in section 6.4. The pound sign has a screen code value of 28 ($1C). The reverse pound sign has a screen code value of 28+128 = 156. With this information we can complete our formula:

```
address = 12288 + 8*28 = 12512 = $30E0
```

Now we know that the definition of the pound sign is contained in the 8 bytes from $30E0 to $30E7 (12512-12519).

By storing the above 8 bytes of our special character in these positions, we immediately change the appearance of all pound signs on the screen. This is done by the following BASIC program (after the previous routine has copied and moved the character set):

```
10 REM CHANGE CHARACTER SET
20 FOR X= 0 TO 7
30  READ DT : REM READ 8 DATA
40  POKE 12288 + 8*28 + X, DT : REM AND POKE IN
50 NEXT X
60 DATA 28,112,207,192,207,112,28,0
```

Before RUNning this program, you should display a few pound signs on the screen so that you can see it in action. If you switch to the upper/lower case mode with <SHIFT><C=> you will see that the pound sign in this character set has not been changed. This operation can naturally be performed on any of the characters and thereby create a whole supply of special characters which can be used for creating graphics or for commercial purposes.

4.4.2 Modifying a character set

If we want to change an entire character set, we do not have to copy the old one. Creating a new character set can therefore be done in BASIC. Simply load the new character set into a specific memory area and inform the VIC that it should get its character definitions from that area.

First we must decide what sort of characters the new character set will contain. Producing the new characters and deciding what they will look like is the most tedious part

of character set programming and is what stops many programmers from doing it. Each character must be developed on paper, converted to numerical equivalents, and then placed in memory with a machine language monitor or--even more tedious--POKEd with BASIC. To automate this procedure, we now present a program which makes it much simpler to define a new character set. In addition to this character editor, the companion program disk for this book contains a version for working with multi-color characters.

```
10 REM ************************
20 REM **                      **
30 REM **  CHARACTER EDITOR    **
40 REM **                      **
50 REM ************************
60 REM
70 REM INITIALIZATION:
80 REM ****************
90 GOSUB 10000 : REM READ MACHINE LANGUAGE ROUTINES
100 POKE 53280,0:POKE 53281,0:REM BACKGROUND/BORDER COLOR
110 POKE 763,0:POKE 650,255:REM MODE=0: ALL CHARACTERS REPEAT
120 POKE 45,0:POKE 46,80:RUN 130:REM BASIC END=$5000
130 REM
140 REM MACHINE LANGUAGE ROUTINES:
150 REM ****************************
160 IN%=18432:REM INIT ROUTINE
170 PT%=18633:REM DRAW POINT
180 NE%=18586:REM COORDINATE SYSTEM
190 LD%=18487:REM LOAD CHARACTER SET
200 SV%=18515:REM SAVE CHARACTER SET
210 CA%=18765:REM CATALOG
220 CM%=18689:REM COMMAND IDENTIFICATION
230 Q  =13048:REM TEST CHARACTER ADDRESS
240 REM
250 REM CONTROL CHARACTERS:
260 REM ********************
270 C0$=CHR$(147):REM CLEAR SCREEN
280 C1$=CHR$( 19):REM HOME
290 C2$=CHR$(183):REM LINE
300 C3$=CHR$(117)+CHR$( 99)+CHR$(105):REM UPPER CHARACTER
    WINDOW BORDER
310 C4$=CHR$(106)+CHR$( 99)+CHR$(107):REM LOWER BORDER
320 C5$=CHR$( 98):REM LINE (VERT)
```

```
330 C6$=CHR$( 18):REM RVS ON
340 C7$=CHR$(146):REM RVS OFF
350 NA%=704:REM FILENAME LENGTH($02C0)
360 DA%=186:REM DEVICE ADDRESS($BA)
370 MO%=763:REM MODE
380 KY%=764:REM KEY/ COMMAND CODE
390 YC%=765:REM Y-COORD
395 XC%=766:REM X-COORD
400 REM
410 REM DEFINE COLORS:
420 REM ***************
430 DATA 144,  5, 28,159,156, 30, 31,158
440 DATA 129,149,150,151,152,153,154,155
450 DIM C$(16):FOR Y=0 TO 15:READ X:C$(Y)=CHR$(X):NEXT Y
460 N=1:F(0)=0:F(1)=1:V$=" ":SYS IN%:REM COLORS/INIT
500 REM
510 REM ERASE ROUTINE (FIELD CONSTRUCTION):
520 REM ************************************
530 FOR Y=Q TO Q+7:POKE Y,0:NEXT Y:REM ERASE TEST CHARACTER
540 PRINT C0$
550 PRINT C1$;SPC(12);C$(7);"CHARACTER EDITOR"
560 PRINT SPC(8);C$(1);"(C) 1984 BY AXEL PLENGE"
570 PRINT C$(4);:FOR X=1 TO 40:PRINT C2$;:NEXT X:PRINT C$(6):
    PRINT
580 PRINT "   76543210":SYS NE%
590 PRINT "   ";:FOR X=0 TO 7:PRINTC2$;:NEXT X
600 A=15:B=5:GOSUB 9000:REM POSITIONING
610 PRINT C3$:PRINT TAB(15);C5$;CHR$(127);C5$:
    POKE 55552,F(1):REM TEST CHARACTER W/ CLR
620 PRINT TAB(15);C4$:GOSUB 3000:X=0:Y=0:REM STATUS FIELD/
    X/Y-COORD=0
700 REM
710 REM INPUT LOOP:
```

```
720 REM ***********
730 A=X+3:B=Y+6:GOSUB 9000:REM POSITIONING
740 POKE XC%,X:POKE YC%,Y:F=0:REM TRANSMIT COORDINATES
750 PRINT C$(7);C6$;" ";CHR$(157);:REM BLINK PHASE ON
760 FOR S=1 TO 50:GETA$:IF A$<>"" THEN 800
770 NEXT S:SYS PT%:FOR S=1 TO 50:GET A$:IF A$="" THEN NEXT S:
    GOTO 750:REM TURN OFF
800 REM
810 REM COMMAND RECOGNITION:
820 REM ********************
830 SYS PT%:C=ASC(A$):POKE KY%,C:SYS CM%:S=PEEK(KY%):REM
    COMMAND TRANSMISSION/RETURN
840 REM DIVISION:
850 ON S GOTO 1600,2800,2900,1060,1060,1080,1080,1100,1100
860 ON S-9 GOTO 1110,1110,3100,3100,3100,3100
870 ON S-15 GOTO 3100,3100,3100,3100,500,1700
880 ON S-21 GOTO 1800,1900,2000,2100,1200,3400
890 ON S-27 GOTO 1300,3200,1400,700
1000 REM
1010 REM EXECUTE COMMANDS:
1020 REM ******************
1030 REM
1040 REM MOVE CURSOR:
1050 REM *************
1060 X=X+1:IF X=8 THEN X=0:GOTO 1100
1070 GOTO 700:REM RIGHT
1080 X=X-1:IF X<0 THEN X=7:GOTO 1110
1090 GOTO 700:REM LEFT
1100 Y=(Y+1)AND7:GOTO 700:REM DOWN
1110 Y=(Y-1)AND7:GOTO 700:REM UP
1200 REM
1210 REM EXIT:
1220 REM *****
```

```
1230 A=2:B=15:GOSUB 9000:REM POSITIONING
1240 PRINT C6$;C$(7);"EXIT?";C7$;C$(6):INPUT T$
1250 IF T$="Y" OR T$="YES" THEN SYS 64738:REM COLD START
1260 GOTO 540
1300 REM
1310 REM CATALOG:
1320 REM ********
1330 PRINT C0$:SYS CA%:GOSUB 9100:GOTO 540
1400 REM
1410 REM MOVE:
1420 REM *****
1430 GOSUB 8999:PRINT C$(1);"MOVE:":REM MESSAGE FIELD
1440 PRINT "TO: RIGHT(R), LEFT(L),":
     PRINT SPC(4) "UP   (U), DOWN(D):"
1450 GOSUB 9100:IF T$<>"R" THEN 1470
1460 FOR T=0 TO 7:R=PEEK(Q+T):POKE Q+T,(R/2) + (R AND 1)*128:
     NEXT T:REM RIGHT
1470 IF T$<>"L" THEN 1490
1480 FOR T=0 TO 7:R=PEEK(Q+T)*2:POKE Q+T,(R AND 255) + R/256:
     NEXT T:REM LEFT
1490 S=1:IF T$="U" THEN 1510
1500 S=-1:IF T$<>"D" THEN 1520
1510 FORT=0TO7:P(T)=PEEK((7ANDT+S)+Q):NEXTT:FORT=0TO7:
     POKEQ+T,P(T):NEXTT:REM UP/DOWN
1520 GOSUB 9200:GOTO 550
1600 REM
1620 REM MODE CHANGE:
1630 REM ************
1640 M=ABS(M-1):POKE MO%,M:GOSUB 3000:GOTO 700
1700 REM
1710 REM DEFINE CHARACTER:
1720 REM ****************
1730 GOSUB 8999:PRINT C6$;C$(5);"ENTER THE";C7$
```

```
1740 PRINT C$(7);"ASSIGNMENT OF THE CHARACTER:"
     C$(6);CHR$(127);C$(7)
1750 GOSUB 2500:IF F=1 THEN F=0:GOTO 540:REM
1760 FOR X=0 TO 7:POKE T+X,PEEK(Q+X):NEXT X:REM SAVE
1770 FOR X=1 TO 1500:NEXT X:GOSUB 9200:GOTO550:REM WAIT +
     ERASE
1800 REM
1810 REM GET CHARACTER:
1820 REM **************
1830 GOSUB 8999:PRINT C$(1);"ENTER THE CHARACTER":
     PRINT "TO WORK WITH:"
1840 GOSUB 2500:IF F=1 THEN F=0:GOTO 540:REM
1850 FOR Y=0 TO 7: POKE Q+Y,PEEK(T+Y):NEXT Y:GOSUB 9200:
     GOTO 550:REM LOAD
1900 REM
1910 REM INVERT CHARACTER:
1920 REM ******************
1930 FOR B=0 TO 7:POKE Q+B,255-PEEK(Q+B):NEXT B:GOTO 550
2000 REM
2010 REM SAVE CHARACTER SET:
2020 REM ********************
2030 GOSUB 8999:PRINT C6$;C$(1);"SAVING CHARACTER SET";C7$
2040 GOSUB 2300:IF F=1 THEN F=0:GOTO 2000:REM INPUT/ERROR
     CHECK
2050 IF F=2 THEN F=0:GOTO 2150:REM ERROR
2060 SYS SV%:GOTO 2200:REM SAVE
2100 REM
2110 REM LOAD CHARACTER SET:
2120 REM ********************
2130 GOSUB 8999:PRINT C6$;C$(1);"LOADING CHARACTER SET:";C7$:
2140 GOSUB 2300:IF F=1 THEN F=0:GOTO 2100
2150 IF F=2 THEN F=0:GOTO 540
2160 SYS LD%
```

```
2200 REM READ ERROR (ONLY FOR DISK!):
2210 OPEN 1,8,15:INPUT#1,DS,DS$,DT,DB:CLOSE1
2220 IF DS<20 THEN 500:REM OK
2230 PRINT:T$=STR$(DS)+","+DS$+","+STR$(DT)+","+STR$(DB)
2240 GOSUB 9310:PRINT T$:FOR S=1 TO 2000:NEXT S:GOTO 500:
     REM FLASH
2300 REM
2310 REM INPUT NAME:
2320 REM ***********
2330 A$="":PRINT:PRINT "FILENAME"+C$(6);:INPUT A$:T=LEN(A$)
2340 S=VAL(RIGHT$(A$,1))
2350 IF S<>0 AND LEFT$(RIGHT$(A$,2),1)=";" THEN T=T-2:
     POKE DA,S:REM DEVICE ADDR.
2360 IF T=0 THEN F=2:RETURN:REM NO NAME
2370 IF T>17 THEN 2390
2375 REM NAME TO MACHINE LANG. ROUTINES(704=$02C0):
2380 POKE NA%,T:FOR S=1 TO T:POKE NA%+S,ASC(MID$(A$,S,1)):
     NEXT S:RETURN
2390 PRINT CHR$(145);:T$=C6$+"LENGTH!"+C7$:GOSUB 9310:REM
     ERROR MESSAGE
2400 PRINT C$(6):F=1:RETURN
2500 REM
2510 REM CALCULATE CHARACTER ADDRESS:
2520 REM ****************************
2530 PRINT "CHARACTER SET(1-4): ";N:A$=""
2540 PRINT "KEY(F3) OR ASCII(F5)?":GOSUB 9100
2550 ON ABS(ASC(T$)-132) GOTO 2570,2570,2590,2590
2560 GOTO 9300
2570 INPUT "KEY";A$:A$=LEFT$(A$,1):IF A$="" THEN 9300:REM
     KEY INPUT
2580 T=ASC(A$):GOTO 2620
2590 INPUT "ASCII";A$:IF A$="" THEN 9300:REM ASCII INPUT
2600 T=VAL(A$)/255:T=INT((T-INT(T))*255):A$=CHR$(T)
```

```
2610 IF T<32 OR (T<160 AND T>127) THEN 9300
2620 IF T>191 THEN T=T-96:REM ASCII CONVERSION
2630 PRINT CHR$(145);"KEY:";A$;" ASCII:";T;:IF B>8 THEN V$=A$
2640 REM CONVERT (T IS THE NORMAL ASCII VALUE OF THE
     CHARACTER):
2650 IF T<64  THEN S=256:GOTO 2680
2660 IF T<128 THEN S=256*((32 AND T)/16)
2670 IF T>159 THEN S=256*(INT(T/32)-2)
2680 T=(N+47)*1024+S+(31 AND T)*8:RETURN
2800 REM
2810 REM CHANGE CHARACTER SET:
2820 REM **********************
2830 A=36:B=5:GOSUB 9000:PRINT C$(2);C6$;N;C7$;C$(6):REM
     INVERT NUMBER
2840 GOSUB 9100:S=VAL(T$):IF S=0 OR S>4 THEN S=N:REM S=NEW/
     N=OLD CHARACTER SET
2850 N=S:GOSUB 3000:GOTO 700
2900 REM
2910 REM GET CHARACTER (IN MODE 1):
2920 REM ***************************
2930 IF C<32 OR (C>127 AND C<160) THEN 700
2940 T=C:R=N:V$=A$:GOSUB 3000:GOTO 1850
3000 REM
3010 REM ESTABLISH STATUS FIELD:
3020 REM ************************
3030 A=20:B=5:GOSUB 9000:PRINT C$(7);"MODE:";M;"/ SET:";N:
     A$=V$:T=ASC(A$)
3040 PRINT TAB(20);CHR$(17);CHR$(17);C$(6);
3050 GOSUB 2620:PRINT CHR$(157);" ":PRINT
3060 PRINT TAB(20);C$(7);"COLOR ASSIGNMENT:"
3070 PRINT TAB(20);C$(2);:FOR S=1 TO 17:PRINTCHR$(163);:
     NEXT S:PRINT C$(6)
3080 FOR S=0 TO 1:PRINT TAB(20);"CHAR COLOR";S;":";F(S):
```

```
     NEXT S:RETURN
3100 REM
3110 REM PLOT:
3120 REM *****
3130 S=((S-12) AND 2)/2:REM SET PLOT COLOR
3140 T=2^(7-X):POKE Q+Y,PEEK(Q+Y) AND (255-T) OR S*T:GOTO 700
3200 REM
3210 REM SELECT COLOR:
3220 REM **************
3230 GOSUB 8999:PRINT TAB(4);C$(1)"S"C$(2)"E"C$(3)"L"C$(4)
     "E"C$(5)"C"C$(6)"T";" ";
3240 PRINT C$(7)"C"C$(4)"O"C$(6)"L"C$(2)"O"C$(7)"R"
3250 PRINT TAB(4);C$(1);CHR$(172);:FOR S=1 TO 32:PRINT
     CHR$(162);:NEXT S:PRINT CHR$(187)
3260 FOR S=1 TO 2:PRINT TAB(4);C6$;CHR$(161);
3270 FOR T=0 TO 15:PRINT C$(T);"  ";:NEXT T:PRINT C$(1);C7$;
     CHR$(161):NEXT S
3280 PRINT TAB(4);C6$;CHR$(161);
     " 0 1 2 3 4 5 6 7 8 9101112131415";C7$;CHR$(161)
3290 PRINT:PRINT C$(6);"BCKGND CLR (F1) CHAR CLR (F3)";
3300 GOSUB 9100:T=ASC(T$)-133:REM FUNCTION KEY
3310 IF T<0 OR T>1 THEN GOSUB 9300:GOTO 540:REM ERROR
3320 IF T>1 THEN T=T-4
3330 PRINT T:T$="":INPUT "    COLOR ";T$:S=ABS(INT(VAL(T$)))
3340 IF T$="" OR S>15 THEN GOSUB 9300:GOTO540:REM ERROR
3350 F(T)=S:POKE 53281+T,S:REM SET COLOR
3360 GOTO 540
3400 REM
3410 REM COMMAND SET:
3420 REM ************
3430 PRINT C0$;C6$;C$(2)"              ";C$(7);
3440 PRINT "COMMAND SET";C$(2);"               ";C7$;
3450 PRINT C$(4);:FOR S=1 TO 40:PRINT CHR$(184);:NEXT S:PRINT
```

```
3460 PRINT C$(1)" NO.  "C6$"COMMAND]"C7$"-"C$(5)" FUNCTION"
     C$(4)
3470 FOR S=1 TO 10:PRINT "----";:NEXT S
3480 PRINT C$(1)" (1)  "C6$"(^...)"C7$"-"C$(5)" MOVE CURSOR"
3490 PRINT C$(1)" (2)  "C6$"(<DEL>)"C7$"-"C$(5)" CHANGE MODE"
3500 PRINT C$(1)" (3)  "C6$"(CTRL^)"C7$"-"C$(5)
     " CHANGE CHARACTER SET"
3510 PRINT C$(1)" (4)  "C6$"(F1/F3)"C7$"-"C$(5)" UNPLOT/PLOT"
3520 PRINT C$(1)" (5)  "C6$"(F)    "C7$"-"C$(5)
     " CLRS 0-15 DEF.FOR F1/F3"
3530 PRINT C$(1)" (6)  "C6$"(B)    "C7$"-"C$(5)
     " COMMAND MENU"
3540 PRINT C$(1)" (7)  "C6$"(D)    "C7$"-"C$(5)
     " DEFINE CHARACTER"
3550 PRINT C$(1)" (8)  "C6$"(H)    "C7$"-"C$(5)
     " GET CHARACTER"
3560 PRINT C$(1)" (9)  "C6$"(I)    "C7$"-"C$(5)
     " INVERT CHARACTER"
3570 PRINT C$(1)"(10)  "C6$"(V)    "C7$"-"C$(5)
     " SHIFT CHARACTER"
3580 PRINT C$(1)"(11)  "C6$"(L)    "C7$"-"C$(5)
     " ERASE CHARACTER"
3590 PRINT C$(1)"(12)  "C6$"(CTRLG)"C7$"-"C$(5)
     " GET-LOAD CHARACTER"
3600 PRINT C$(1)"(13)  "C6$"(CTRLS)"C7$"-"C$(5)
     " SAVE-SAVE CHARACTER"
3610 PRINT C$(1)"(14)  "C6$"(C)    "C7$"-"C$(5)
     " DIRECTORY/CATALOG"
3620 PRINT C$(1)"(15)  "C6$"(CTRLX)"C7$"-"C$(5)" EXIT"
3630 GOSUB 9100:GOTO 540
8000 REM
8100 REM SUBROUTINES:
8200 REM ************
```

```
8300 REM
8400 REM POSITIONING:
8500 REM ************
8999 A=0:B=16:REM MESSAGE FIELD
9000 PRINT C1$;:FOR S=2 TO B:PRINT:NEXT S:PRINT TAB(A);:
     RETURN
9050 REM
9060 REM KEY INPUT:
9070 REM **********
9100 WAIT 198,255:GET T$:RETURN
9150 REM
9160 REM ERASE MESSAGE FIELD:
9170 REM ********************
9200 A=0:B=16:GOSUB 9000:FOR S=1 TO 5:GOSUB 9210:PRINT:
     NEXT S:RETURN
9210 FOR T=1 TO 9:PRINT "    ";:NEXT T:RETURN
9250 REM
9260 REM FLASH ERROR:
9270 REM ************
9300 T$="ILLEGAL!"
9310 PRINT C$(1):FOR S=1 TO 9:PRINT T$:GOSUB 9330:
     PRINT CHR$(145);
9320 GOSUB 9210:PRINT CHR$(145):GOSUB 9330:NEXT S:GOSUB 9200:
     F=1:RETURN
9330 FOR T=1 TO 75:NEXT T:RETURN:REM DELAY LOOP
9890 REM
9900 REM ******************************
9910 REM **                            **
9920 REM **  MACHINE LANG. ROUTINES    **
9930 REM **                            **
9940 REM ******************************
9950 REM
9960 REM DATA WILL BE ERASED AFTER START!!
```

```
9970 REM
10000 FOR I = 1 TO 16 : READ X : NEXT I : REM SKIP FIRST
      DATA (COLORS)
10005 FOR I = 18432 TO 18836
10010 READ X : POKE I,X : S=S+X : NEXT
10020 DATA 120,169, 51,133,  1,169, 48, 32, 26, 72,169,192
10030 DATA  32, 26, 72,169, 55,133,  1,169, 28,141, 24,208
10040 DATA  88, 96,133,  5,169,208,160,  0,132,  2,132,  4
10050 DATA 133,  3,162, 16,177,  2,145,  4,136,208,249,230
10060 DATA   5,230,  3,202,208,242, 96,173,192,  2,162,193
10070 DATA 160,  2, 32,249,253,169,  2,166,186,160,  0, 32
10080 DATA   0,254,169,  0,162,  0,160,192, 76,213,255,173
10090 DATA 192,  2,162,193,160,  2, 32,249,253,169,  2,166
10100 DATA 186,160,  0, 32,  0,254,169, 20,141, 24,208,169
10110 DATA  48,133,  5,169,192, 32, 30, 72,169,  0,133,  2
10120 DATA 169, 48,133,  3,169,  2,162,255,160, 63, 32,216
10130 DATA 255,120,169, 48,162, 51,134,  1, 32, 26, 72,169
10140 DATA  55,133,  1,169, 28,141, 24,208, 88, 96,160,  0
10150 DATA 169, 32, 32,210,255, 32,210,255,152,  9, 48, 32
10160 DATA 210,255,162,  0, 32,207, 72,169, 29, 32,210,255
10170 DATA 232,224,  8,208,243,169,165, 32,210,255,169, 13
10180 DATA  32,210,255,200,192,  8,208,212, 96,174,254,  2
10190 DATA 172,253,  2,152, 72,138, 72,169,  0, 56,106,202
10200 DATA  16,252, 57,248, 50,208,  3,160,  0, 44,160,  6
10210 DATA 162,  6,185,245, 72, 32,210,255,200,202,208,246
10220 DATA 104,170,104,168, 96,146, 31,111, 31,146,157, 18
10230 DATA  28, 32, 31,146,157,173,252,  2,162,  0,201, 20
10240 DATA 240, 35,201,148,240, 31,232,201, 30,240, 26,201
10250 DATA  94,208,  5,172,251,  2,240, 17,232,172,251,  2
10260 DATA 208, 11,160, 28,232,221, 47, 73,240,  3,136,208
10270 DATA 247,232,142,252,  2, 96, 87, 29, 81,157, 65, 17
10280 DATA  50,145,133,137,134,138,135,139,136,140, 76, 68
10290 DATA  72, 73, 19,  7, 24, 66, 67, 70, 86,169, 36,133
```

```
10300 DATA   2,169,  1,162,  2,160,  0, 32,249,253,169,  2
10310 DATA 166,186,160,  0, 32,  0,254,169,  0,162,  0,160
10320 DATA  64,134, 95,132, 96, 32,213,255,165, 95,164, 96
10330 DATA  32, 55,165,173,  0,  3, 72,173,  1,  3, 72,169
10340 DATA  61,141,  0,  3,169,227,141,  1,  3, 32,195,166
10350 DATA 104,141,  1,  3,104,141,  0,  3, 96
10360 IF S <> 46617 THEN PRINT "ERROR IN DATA!!" : END
10370 PRINT "OK" : RETURN
```

```
7:      4800                    .OPT P1
10:                     ;
20:                     ;MACHINE LANGUAGE ROUTINES
30:                     ;**************************
40:                     ;
50:                     ;
60:                     ;
70:     4800                    *=    $4800    ;START ADDRESS
80:                     ;
90:                     ;
100:                    ;JUMP ADDRESSES AND REGISTERS
110:                    ;*****************************
120:                    ;
130:    FFD2            CHROUT    =     $FFD2    ;CHARACTER OUTPUT
140:    FDF9            FNPAR     =     $FDF9    ;SET FILENAME
                                                   PARAMETERS
150:    FE00            FPAR      =     $FE00    ;SET FILE
                                                   PARAMETERS
160:    FFD8            SAVE      =     $FFD8    ;SAVE ON DISK/
                                                   CASSETTE
170:    FFD5            LOAD      =     $FFD5    ;LOAD FROM DISK/
                                                   CASSETTE
180:    32F8            TESTCH    =     $32F8    ;TEST CHARACTER
190:    02C0            LENGTH    =     $02C0    ;FILENAME LENGTH
200:    02FB            MODE      =     $02FB    ;MODE
210:    02FC            KEY       =     $02FC    ;COMMAND KEY
                                                   PRESS
220:    02FD            YCOORD    =     $02FD    ;Y-COORDINATE
                                                   FIELD
230:    02FE            XCOORD    =     $02FE    ;X-COORDINATE
                                                   FIELD
240:                    ;
250:                    ;COPY CHARACTER SET
```

```
260:                ;*******************
270:                ;
280:   4800 78     INIT     SEI          ;NO INTERRUPTS
290:   4801 A9 33           LDA  #$33
300:   4803 85 01           STA  $01     ;MAKE CHARACTER
                                          SET READABLE
310:   4805 A9 30           LDA  #$30    ;FROM $D000 TO
                                          $3000
320:   4807 20 1A 48        JSR  MOVE    ;COPY
330:   480A A9 C0           LDA  #$C0    ;COPY FROM $D000
                                          TO $C000
340:   480C 20 1A 48        JSR  MOVE
350:   480F A9 37           LDA  #$37
360:   4811 85 01           STA  $01     ;SELECT I/O
370:   4813 A9 1C           LDA  #$1C
380:   4815 8D 18 D0        STA  $D018   ;CHARACTER SET
                                          ADDRESS TO $3000
390:   4818 58              CLI          ;ENABLE INTERRUPTS
400:   4819 60              RTS
410:                ;
420:                ;COPY
430:                ;*****
440:                ;
450:   481A 85 05    MOVE     STA  $05     ;HIGH BYTE, DEST-
                                          INATION ADDRESS
460:   481C A9 D0           LDA  #$D0    ;HIGH BYTE,
                                          SOURCE ADDRESS
470:   481E A0 00    M0       LDY  #$00
480:   4820 84 02           STY  $02     ;LOW BYTE,
                                          SOURCE ADDRESS
490:   4822 84 04           STY  $04     ;LOW BYTE, DEST-
                                          INATION ADDRESS
500:   4824 85 03           STA  $03
```

```
510:   4826 A2 10            LDX  #$10
520:   4828 B1 02    M1      LDA  ($02),Y  ;LOAD
530:   482A 91 04            STA  ($04),Y  ;SAVE
540:   482C 88               DEY
550:   482D D0 F9            BNE  M1
560:   482F E6 05            INC  $05
570:   4831 E6 03            INC  $03
580:   4833 CA               DEX
590:   4834 D0 F2            BNE  M1       ;NEXT PAGE
600:   4836 60               RTS
610:                ;
620:                ;LOAD CHARACTER SET
630:                ;*******************
640:                ;
650:   4837 AD C0 02 LOADR     LDA  LENGTH   ;NAME LENGTH
660:   483A A2 C1            LDX  #$C1     ;FILENAME ADDRESS
                                            LOW
670:   483C A0 02            LDY  #$02     ;HIGH BYTE
680:   483E 20 F9 FD         JSR  FNPAR
690:   4841 A9 02            LDA  #$02     ;LOGICAL FILE
                                            NUMBER
700:   4843 A6 BA            LDX  $BA      ;DEVICE ADDRESS
710:   4845 A0 00            LDY  #$00     ;SECONDARY ADDRESS
720:   4847 20 00 FE         JSR  FPAR
730:   484A A9 00            LDA  #$00     ;LOAD/VERIFY FLAG
740:   484C A2 00            LDX  #$00     ;START ADDRESS
                                            (LOW BYTE)
750:   484E A0 C0            LDY  #$C0     ;START ADDRESS
                                            (HIGH BYTE)
760:   4850 4C D5 FF         JMP  LOAD
770:                ;
780:                ;SAVE CHARACTER SET
790:                ;*******************
```

```
800:                    ;
810:    4853 AD C0 02 SAVER     LDA  LENGTH  ;FILENAME LENGTH
820:    4856 A2 C1              LDX  #$C1    ;FILENAME ADDRESS
                                              (LOW)
830:    4858 A0 02              LDY  #$02    ;FILENAME ADDRESS
                                              (HIGH)
840:    485A 20 F9 FD           JSR  FNPAR
850:    485D A9 02              LDA  #$02    ;LOGICAL
                                              FILENUMBER
860:    485F A6 BA              LDX  $BA     ;DEVICE ADDRESS
870:    4861 A0 00              LDY  #$00    ;SECONDARY ADDRESS
880:    4863 20 00 FE           JSR  FPAR
890:    4866 A9 14              LDA  #$14
900:    4868 8D 18 D0           STA  $D018   ;ORIGINAL CHAR.
                                              SET SITUATION
910:    486B A9 30              LDA  #$30    ;DESTINATION AD-
                                              DRESS (HIGH BYTE)
920:    486D 85 05              STA  $05
930:    486F A9 C0              LDA  #$C0    ;SOURCE ADDRESS
                                              (HIGH BYTE)
940:    4871 20 1E 48           JSR  M0      ;FROM $C000 TO
                                              $3000
950:    4874 A9 00              LDA  #$00
960:    4876 85 02              STA  $02     ;START ADDRESS
                                              (LOW BYTE)
970:    4878 A9 30              LDA  #$30
980:    487A 85 03              STA  $03     ;START ADDRESS
                                              (HIGH BYTE)
990:    487C A9 02              LDA  #02     ;ZERO PAGE ADDRESS
                                              OF START ADDRESS
1000:   487E A2 FF              LDX  #$FF    ;END ADDRESS (LSB)
1010:   4880 A0 3F              LDY  #$3F    ;END ADDRESS (MSB)
1020:   4882 20 D8 FF           JSR  SAVE    ;START TS AT $3000
```

```
1030:  4885 78                SEI          ;BLOCK INTERRUPTS
1040:  4886 A9 30             LDA  #$30    ;DESTINATION ADDR.
                                            (MSB)
1050:  4888 A2 33             LDX  #$33
1060:  488A 86 01             STX  $01     ;MAKE CHARACTER
                                            SET READABLE
1070:  488C 20 1A 48          JSR  MOVE    ;FROM $D000 TO
                                            $3000
1080:  488F A9 37             LDA  #$37
1090:  4891 85 01             STA  $01     ;SELECT I/O
1100:  4893 A9 1C             LDA  #$1C
1110:  4895 8D 18 D0          STA  $D018   ;CHARACTER SET
                                            ADDRESS TO $3000
1120:  4898 58                CLI          ;ENABLE INTERRUPTS
1130:  4899 60                RTS          ;BACK TO BASIC
1140:                ;
1150:                ;CREATE WORK FIELD
1160:                ;******************
1170:                ;
1180:  489A A0 00    GRID     LDY  #$00    ;LINE COUNTER=0
1190:  489C A9 20    N0       LDA  #$20    ;" "
1200:  489E 20 D2 FF          JSR  CHROUT
1210:  48A1 20 D2 FF          JSR  CHROUT  ;2 SPACES
1220:  48A4 98                TYA
1230:  48A5 09 30             ORA  #$30    ;CONVERT LINE
                                            COUNTER TO DIGITS
1240:  48A7 20 D2 FF          JSR  CHROUT  ;OUTPUT
1250:  48AA A2 00             LDX  #$00
1260:  48AC 20 CF 48 N1       JSR  POINT   ;DRAW A POINT
1270:  48AF A9 1D             LDA  #$1D    ;CURSOR RIGHT
1280:  48B1 20 D2 FF          JSR  CHROUT
1290:  48B4 E8                INX          ;NEXT POINT
1300:  48B5 E0 08             CPX  #$08    ;EIGHT POINTS PER
```

```
                                                LINE
1310:  48B7 D0 F3            BNE  N1
1320:  48B9 A9 A5            LDA  #$A5      ;LINE (CHR$(165))
1330:  48BB 20 D2 FF         JSR  CHROUT
1340:  48BE A9 0D            LDA  #$0D      ;CARRIAGE RETURN
1350:  48C0 20 D2 FF         JSR  CHROUT
1360:  48C3 C8               INY            ;NEXT LINE
1370:  48C4 C0 08            CPY  #$08      ;8 LINES
1380:  48C6 D0 D4            BNE  N0
1390:  48C8 60               RTS
1400:                ;
1410:                ;DRAW A COORDINATE
1420:                ;******************
1430:                ;
1440:  48C9 AE FE 02 POINT2  LDX  XCOORD    ;CONTROLLED BY
                                             BASIC
1450:  48CC AC FD 02         LDY  YCOORD    ;X/Y-COORDINATE
1460:  48CF 98       POINT   TYA
1470:  48D0 48               PHA
1480:  48D1 8A               TXA
1490:  48D2 48               PHA            ;SAVE COORDINATES
1500:  48D3 A9 00            LDA  #$00
1510:  48D5 38               SEC            ;SET A BIT
1520:  48D6 6A       P0      ROR  A         ;SEARCH FOR RIGHT
                                             BIT
1530:  48D7 CA               DEX
1540:  48D8 10 FC            BPL  P0
1550:  48DA 39 F8 32         AND  TESTCH,Y ;ERASE OTHER BITS
                                             OF TEST CHARACTER
1560:  48DD D0 03            BNE  P1        ;BIT (=POINT) SET?
1570:  48DF A0 00            LDY  #$00      ;NO
1580:  48E1 2C               .BYTE$2C       ;BIT COMMAND
1590:  48E2 A0 06    P1      LDY  #$06      ;BIT SET!
```

```
1600:  48E4 A2 06              LDX  #$06
1610:  48E6 B9 F5 48 P2        LDA  PNTTAB,Y  ;CHARACTER FROM
                                               POINT TABLE
1620:  48E9 20 D2 FF           JSR  CHROUT
1630:  48EC C8                 INY
1640:  48ED CA                 DEX
1650:  48EE D0 F6              BNE  P2        ;NEXT CHARACTER
1660:  48F0 68                 PLA
1670:  48F1 AA                 TAX
1680:  48F2 68                 PLA
1690:  48F3 A8                 TAY            ;GET COORDINATES
                                               BACK
1700:  48F4 60                 RTS
1710:                ;
1720:                ;CHARACTERS FOR A COORDINATE
1730:                ;*****************************
1740:                ;
1750:  48F5 92 1F 6F PNTTAB    .BYTE146,031,111,031,146,157
1760:  48FB 12 1C 20           .BYTE018,028,032,031,146,157
1770:                ;
1780:                ;IDENTIFY KEY AS A COMMAND
1790:                ;**************************
1800:                ;
1810:  4901 AD FC 02 CMDID     LDA  KEY       ;KEY CODE
1820:  4904 A2 00              LDX  #$00      ;COMMAND CODE
1830:  4906 C9 14              CMP  #$14      ;DEL (=MODE
                                               CHANGE)?
1840:  4908 F0 23              BEQ  B2        ;YES
1850:  490A C9 94              CMP  #$94      ;INSERT (AS DEL)?
1860:  490C F0 1F              BEQ  B2        ;YES
1870:  490E E8                 INX            ;COMMAND CODE+1
1880:  490F C9 1E              CMP  #$1E      ;CTRL "UP ARROW"?
1890:  4911 F0 1A              BEQ  B2        ;YES
```

```
1900:  4913 C9 5E          CMP  #$5E     ;UP ARROW?
1910:  4915 D0 05          BNE  B0       ;NO
1920:  4917 AC FB 02       LDY  MODE     ;ONLY IN MODE 0
1930:  491A F0 11          BEQ  B2
1940:  491C E8        B0   INX           ;COMMAND CODE+1
1950:  491D AC FB 02       LDY  MODE     ;MODE 1?
1960:  4920 D0 0B          BNE  B2       ;YES, THEN NO
                                          MORE COMMANDS
1970:  4922 A0 1C          LDY  #$1C     ;27-1 COMMANDS
                                          (COUNTER)
1980:  4924 E8        B1   INX           ;INCREMENT COMMAND
                                          CODE
1990:  4925 DD 2F 49       CMP  CMDTAB-3,X  ;COMPARE KEY
                                             WITH TABLE
2000:  4928 F0 03          BEQ  B2       ;FOUND
2010:  492A 88             DEY           ;NEXT COMMAND
2020:  492B D0 F7          BNE  B1       ;NOT YET DONE
2030:  492D E8        B2   INX
2040:  492E 8E FC 02       STX  KEY      ;COMMAND CODE AS
                                          RESPONSE
2050:  4931 60             RTS           ;BACK TO BASIC
2060:                ;
2070:                ;COMMAND KEY TABLE
2080:                ;******************
2090:                ;
2100:                ;   W/CRSR-RT/Q/CRSR-LT/A/CRSR-DN
2110:  4932 57 1D 51 CMDTAB   .BYTE087,029,081,157,065,017
2120:                ;   2/CRSR-UP/ F1/ F2/ F3/ F4
2130:  4938 32 91 85          .BYTE050,145 ,133,137,134,138
2140:                ;   F5/ F6/ F7/ F8/ L / D
2150:  493E 87 8B 88          .BYTE135,139,136,140,076,068
2160:                ;   H/ I/<HOME>/CTRL.G/CTRL.X/ B
2170:  4944 48 49 13          .BYTE072,073,019,007 ,024 ,066
```

```
2180:                 ;   C / F / V
2190:  494A 43 46 56          .BYTE067,070,086
2200:                 ;
2210:                 ;DISK CATALOG
2220:                 ;*************
2230:                 ;
2240:  494D A9 24     CATALG  LDA  #$24     ;"$" AS FILENAME
2250:  494F 85 02             STA  $02
2260:  4951 A9 01             LDA  #$01
2270:  4953 A2 02             LDX  #$02
2280:  4955 A0 00             LDY  #$00     ;SEE ABOVE
2290:  4957 20 F9 FD          JSR  FNPAR
2300:  495A A9 02             LDA  #$02
2310:  495C A6 BA             LDX  $BA      ;DEVICE ADDRESS
2320:  495E A0 00             LDY  #$00
2330:  4960 20 00 FE          JSR  FPAR
2340:  4963 A9 00             LDA  #$00     ;LOAD/VERIFY FLAG
2350:  4965 A2 00             LDX  #$00
2360:  4967 A0 40             LDY  #$40     ;START ADDRESS
2370:  4969 86 5F             STX  $5F
2380:  496B 84 60             STY  $60
2390:  496D 20 D5 FF          JSR  LOAD     ;LOAD CATALOG AS
                                             BASIC PROGRAM
2400:  4970 A5 5F             LDA  $5F      ;SIMULATE
2410:  4972 A4 60             LDY  $60      ;BASIC PROGRAM
                                             START ADDRESS
2420:  4974 20 37 A5          JSR  $A537    ;ADD BASIC LINES
2430:  4977 AD 00 03          LDA  $0300
2440:  497A 48                PHA
2450:  497B AD 01 03          LDA  $0301    ;ORIGINAL WARM
                                             START VECTOR
2460:  497E 48                PHA           ;SAVE
2470:  497F A9 3D             LDA  #$3D
```

```
2480:  4981 8D 00 03          STA  $0300
2490:  4984 A9 E3             LDA  #$E3
2500:  4986 8D 01 03          STA  $0301    ;AND SET TO RTS
2510:  4989 20 C3 A6          JSR  $A6C3    ;EXECUTE LIST
                                             COMMAND
2520:  498C 68                PLA
2530:  498D 8D 01 03          STA  $0301
2540:  4990 68                PLA
2550:  4991 8D 00 03          STA  $0300    ;GET OLD VECTOR
                                             BACK
2560:  4994 60                RTS           ;BACK TO BASIC
]4800-4995
NO ERRORS
```

The character editor works in much the same way as the sprite editor in section 4.3. We have secondary operations, color division, editor (8x8) and original fields. The program has two modes:

- 0: Input mode:
  Mode 0 allows you to use all of the commands to create your character.

- 1: Call-up mode:
  Mode 1 allows you to call up a previously defined character from the character set (By default, the character editor contains the entire original character set.)

Character sets, each having 128 characters, are numbered from 1-4 (normal upper case/graphics, reverse upper case/graphics, normal upper/lower case, reverse upper/lower

case).

All of the commands from the sprite editor commands (except for Z) also work in the character editor. the following commands are also available in addition to these:

- <DEL> -
Change mode (mode 0-1)

- <CTRL>^ -
Changes the number of the character set with which you are working (1-4) (^ also works in mode 0).

- D/H - DEFINE/GET
With D you can assign your character to an ASCII character (key) and place it into the character set.
With H you can get an existing character from the character buffer and display it on the screen (similar to mode 1).
Both of these commands work with the current character set. You must decide whether you want to enter the character with an ASCII value (f5/f6) or by pressing a key (f1/f3). <RETURN> must follow the entry of a character by ASCII value.

- <CTRL>S / <CTRL>G -
Save or get an entire character set. See sprite editor for more information.

Once you have created your own personal character set with the character editor and saved it onto diskette, you can load it back in after exiting the program simply by entering LOAD "name",8,1. It will load into $3000 (12288) (in order to load it somewhere else, you will need a monitor

or machine language loader program). Next, you must enter the following command to move the original character set vector so that the VIC will take its definitions from the new character set:

```
POKE 53248+24, PEEK(53248) AND 241 OR 12
```

Here, the three bits 1-3 of VIC register 24, which determine address bits 11-13 of the character generator, are first cleared (241 = %1111 0001) and then the top two of the three bits are set (12 = %0000 1100). The new character set immediately appears on the screen. It's that simple.

4.5 Input/Output of graphics and characters sets

It can often take a long time to create a picture on the screen. If we must go through this process each time we wish to display this picture, we would quickly lose interest in it. There are, however, several ways to shorten this process.

We can save the picture to disk or cassette, and then recall it at any time. We can also shorten the time it takes to see the picture again by making hardcopies, printing it on a graphics printer (dot matrix), providing we have one.

We will show you the necessary techniques for doing these things. Because almost every printer has its own way of outputting graphics, it is not possible to have a routine for each. Therefore our discussion will be as general as possible so that you can develop a routine for your own printer. If necessary, you may also be able to find appropriate articles in computer magazines. If this is too complicated for you, you might also look for a graphics expansion package which supports your printer.

4.5.1 Saving/loading

One problem with input and output of graphics screens or character sets to or from diskette or cassette is that the computer must be told the starting and ending address of the memory range to saved. This information is not available in BASIC. The computer can load and save BASIC programs without this information, but must be explicitly told the address ranges of machine language programs or other data.be

The following routine performs this task. It is written so that it can be easily added to the other graphics routines:

```
10 REM ****************************
20 REM **                          **
30 REM **  SAVE GRAPHICS SCREEN    **
40 REM **                          **
50 REM ****************************
60 REM
70 FI$="GRAPHICS" : DA = 8 : REM FILENAME AND DEVICE ADDRESS
80 S = 8192 : E = 16192 : REM START AND END ADDRESS (HERE
   GRAPHICS AT $2000)
90 GOSUB 11200 : END : REM SAVE
11200 REM
11210 REM   ********************
11220 REM   **  SAVE SCREEN   **
11230 REM   ********************
11240 REM
11250 SYS 57812 FI$,DA : REM PASS DISK PARAMETERS (TO $E1D4)
11260 X = E/256
11270 POKE 175, INT(X) : REM HIGH BYTE OF END ADDRESS ($AF)
11280 POKE 174, (X-INT(X))*256 : REM LOW BYTE OF END ADDRESS
      ($AE)
11290 X = S/256
11300 POKE 194, INT(X) : REM HIGH BYTE OF START ADDRESS ($C2)
11310 POKE 193, (X-INT(X))*256 : REM LOW BYTE OF START
      ADDRESS ($C1)
11320 SYS 62954 : REM SAVE ROUTINE ($F5EA)
11330 RETURN
```

Lines 70 and 80 pass the necessary information to the actual save routine to create a file. Line 11250 calls a

part of the normal BASIC SAVE command which accepts the filename and device address (for disk: DA=8; for cassette: DA=1). This command appears unconventional, but the form is nevertheless correct. After this routine is called with SYS 57812, two parameters are required as with a SAVE command. Therefore, no SYNTAX ERROR results.

The following lines pass the starting and ending addresses of the memory range to be saved to the appropriate memory locations for the SAVE routine called in line 11320.

We can use this routine for a variety of purposes. Character sets, sprites, graphics, text, and color memory can all be saved to diskette or cassette. The deciding factor is simply the choice of the parameters. The above example saves the bit-mapped graphics memory which lies at $2000-$3F3F (8192-16192). For character sets, which are stored in $3000-$3FFF (12288-16383), we must change line 80 to the following:

```
80 S = 12288 : E = 16384
```

As you see, we must add 1 to the ending address. If you want to save the text page found from $0400 to $07E7 (1024-2023) on disk, you would write:

```
80 S = 1024 : E = 2024
```

You would enter the same thing to save the color portion of your graphics screen (if it lies in the same location) to tape or disk. If you have a sprite definition in block 11 and wish to save it, you can do that too. Because block 11 lies at $02C0-$02FD (704-766), line 80 must read:

```
80 S = 704 : E = 767
```

This program can also be used to store machine language programs without the need for a monitor.

If you want to load any of these various things in again, all you have to do is enter LOAD"name",8,1 (for disk owners) or LOAD"name",1,1 for datasette.

4.5.2 Hardcopy

The Achilles heel of all commercial programs is often printer operation. Because there are many different types of printers and each one uses its own control codes, ASCII coding, or interface methods (especially for graphics), few programs can service more than three or four types of printers. If you are looking for a graphics extension package, pay attention to the kinds of printers that it supports. Large programs are difficult to maintain without a printed listing. If you work with graphics, you must be sure to pay attention to the graphics capabilities of the printer (also the sharpness and clarity of the graphics image).

As we said before, it is impossible to present routines for all types and brands of printers. Therefore we will present a routine for the Commodore 1525/MPS 801 which is one of the most complicated because of its 7-pin head. Routines for other printers can be developed from this model. It is assumed that the graphics page lies at $2000-$3F3F (8192-16191).

```
11800 REM **************************
11810 REM **   GRAPHIC HARDCOPY   **
11820 REM **  VIC 1525 / MPS 801  **
11830 REM **                      **
11840 REM **************************
11850 REM
11860 SA = 8192
11870 OPEN1,4 : REM OPEN PRINTER CHANNEL
11880 Y=35:MA=255:NL=28:CS=6:REM Y-COORD. / MASK /
      # OF LINES / COLUMN SIZE
11890 FOR FLAG=1 TO 2
11900 FOR LN=1 TO NL : REM NL LINES
11910 PRINT#1,CHR$(8)CHR$(27)CHR$(16)CHR$(0)CHR$(80);:
      REM CENTERED + GRAPHICS ON
11920 XC=0
11930 FOR CL=1 TO 40 : YC=Y : REM COLUMN COUNTER
11940 FOR X=0 TO SG:GOSUB12150:LW(X)=PEEK(AD):YC=YC+1:NEXT X:
      REM LOAD 8*7 POINTS
11950 FOR X=0 TO 7 : REM 8 BYTES TO PRINTER
11960 MS=2^(7-X):B2=0
11970 FOR L=0 TO CS:B1=-2*((LW(L) AND MS)>0):
      B2=B2 OR (B1^L+(L+B1=0)):NEXT L
11980 B2=(B2 AND MA) OR 128:PRINT#1,CHR$(B2)
11990 NEXT X : REM NEXT PRINTER BYTE
12000 XC=XC+8
12010 NEXT CL : REM NEXT COLUMN
12020 PRINT#1 : REM RETURN
12030 Y=Y+7 : REM Y-COORDINATE
12040 NEXT LN : REM NEXT LINE
12050 NL=1 : REM LAST LINE ONLY
12060 MA = 16 : REM MASK (REMOVE TOP 4 BITS)
12070 CS=4 : REM COLUMN SIZE
12080 NEXT FLAG : REM ONE MORE TIME FOR LAST LINE
```

```
12090 CLOSE 1:END
12100 REM
12110 REM ************************
12120 REM **  CALCULATE POINT  **
12130 REM ************************
12140 REM
12150 AD = SA + 320 * INT(YC/8) + (YC AND 7) + 8 * INT(XC/8):
      RETURN
```

This routine can also be added to your collection of subroutines. You need only call it with a GOSUB. You can also create your graphics at SA=8192 and then load and execute this program. Because the graphics will not be disturbed by so doing, a faithful reproduction of the graphics screen is produced on paper by the printer.

Such a routine takes a long time to run in BASIC. This program can be streamlined as explained in section 6.1, although it is presented here in this form for clarity.

It is important for understanding this routine to note that a column of 7 bits is always sent to the printer (each set bit results in a printed point on paper), whereby the high bit must always be set (required by this printer). We encounter a small difficulty at the bottom of the screen: Four rows are left over since 200 (the number of points in the y-direction) is not evenly divisible by 7. These are sent in another pass with a pair of changed parameters. In the main part of the program (lines 11940-11990) the first column of 7 graphics bytes are read (line 11940) and then sent column by column to the printer (lines 11950-11990). The operation L+B1=0, when executed in line 11970, returns -1 when L+B1=0 and 0 if this is not the case. For example, 4=6 returns 0 but 4=4 returns -1. The trick can be helpful in many applications in which conditions determine portions

of expressions, without the need for an IF...THEN or ON...GOTO. This hardcopy routine is reproduced in the graphics package in assembly language.

4.6 IRQ handling

You are already acquainted with the various interrupt capabilities for graphics applications as explained in section 3.7. There these capabilities were presented and all of the theoretical groundwork was laid. Now it is time to start the most difficult of all sections: programming interrupts. The fact that interrupts can only be serviced by machine language makes the whole topic difficult to understand for beginners. Even experienced assembly language programmers must learn completely new programming techniques. But have no fear. You can try out the examples given here without knowing anything about how the interrupt programming is actually done.

We will illustrate the interrupt techniques for both raster line IRQs and lightpen IRQs. With this knowledge you can go on to develop your own programs for sprite collisions, etc. Before you read this section, however, be sure to read section 3.7 first.

Let's recall some essentials which are necessary for servicing general IRQs of the VIC. Interrupts are controlled interruptions of normal program execution. There are 4 different sources which can cause the video controller to generate an interrupt: raster line, light pen, sprite-sprite collision, or sprite-background collision. Each is selected by setting the appropriate bits in the Interrupt Mask Register (IMR), VIC register 26. If an interrupt is generated, the bit corresponding to the source is set in register 25 of the VIC, the Interrupt Request Register (IRR) and bit 7 of this memory location is set to indicate that one of the 4 conditions was met. This register must be cleared after each interrupt by rewriting the contents back into it. If the

interrupts are enabled, an IRQ calls an interrupt service routine. The address of this routine is stored at memory locations $0314/$0315 (788/789). An IRQ can be inhibited by setting the interrupt flag in the CPU (SEI instruction).

4.6.1 Raster line IRQ

Again, we should recall those things which are necessary for a raster-line interrupt:

Bit 0 of the IRR and IMR control this method of interrupt. From register 18 and bit 7 of register 17 you can determine the current screen line on which the VIC is working. The coordinate system used by the VIC is different from the normal system (see section 3.7). If these registers are written to, they determine the raster line at which an interrupt is to occur.

With this information it is now possible to create our own interrupt routine and have direct access to these events. Here is the assembly listing of one such raster-interrupt program:

```
70:    C800                .OPT P1
100:   C800                *=   $C800
110:   00FB        COLOR1  =    $FB
120:   00FC        COLOR2  =    $FC
130:   00FD        UP      =    $FD
140:   00FE        DOWN    =    $FE
150:   0314        IRQVECT =    $0314
160:   EA31        IRQOLD  =    $EA31
170:   D012        RASTER  =    $D012
180:   D019        IRR     =    $D019
190:   D01A        IMR     =    $D01A
```

```
200:   D020          BORDER   =    $D020
210:   D021          BGROUND  =    $D021
220:                 ;
230:                 ;INITIALIZE
240:                 ;**********
250:                 ;
260:   C800 78       INIT     SEI          ;DISABLE INTER-
                                             RUPTS
270:   C801 A9 1F             LDA  #< IRQNEW
280:   C803 8D 14 03          STA  IRQVECT
290:   C806 A9 C8             LDA  #> IRQNEW
300:   C808 8D 15 03          STA  IRQVECT+1 ;CHANGE IRQ
                                               VECTOR
310:   C80B A5 FD             LDA  UP
320:   C80D 8D 12 D0          STA  RASTER  ;SET 1ST RASTER
                                             LINE
330:   C810 AD 11 D0          LDA  RASTER-1
340:   C813 29 7F             AND  #$7F
350:   C815 8D 11 D0          STA  RASTER-1 ;CLEAR HIGH BIT
360:   C818 A9 81             LDA  #%10000001 ;MASK
370:   C81A 8D 1A D0          STA  IMR     ;SELECT RASTER
                                             LINE IRQ
380:   C81D 58                CLI          ;ENABLE INTERRUPT
390:   C81E 60                RTS
400:                 ;
410:                 ;INTERRUPT ROUTINE
420:                 ;******************
430:                 ;
440:   C81F AD 19 D0 IRQNEW   LDA  IRR     ;INTERRUPT
                                             REGISTER
450:   C822 8D 19 D0          STA  IRR     ;CLEAR
460:   C825 30 07             BMI  IRQRAS  ;RASTER LINE IRQ
                                             PRINT
```

```
470:                    ;NORMAL IRQ
480:   C827 AD 0D DC           LDA  $DC0D    ;CLEAR CIA 1 IRR
490:   C82A 58                 CLI           ;ENABLE INTERRUPTS
500:   C82B 4C 31 EA           JMP  IRQOLD   ;TO OLD ROUTINE
510:   C82E AD 12 D0 IRQRAS    LDA  RASTER   ;RASTER POSITION
520:   C831 C5 FE              CMP  DOWN     ;LOWER VALUE
530:   C833 B0 10              BCS  SECOND   ;YES = > JUMP
540:   C835 A5 FB              LDA  COLOR1
550:   C837 8D 20 D0           STA  BORDER   ;SET BORDER
560:   C83A 8D 21 D0           STA  BGROUND  ;AND BACKGROUND
                                              COLOR
570:   C83D A5 FE              LDA  DOWN     ;LOWER VALUE
580:   C83F 8D 12 D0           STA  RASTER   ;IN RASTER
590:   C842 4C BC FE           JMP  $FEBC    ;POSITION
600:   C845 A5 FC     SECOND   LDA  COLOR2
610:   C847 8D 20 D0           STA  BORDER   ;SET BORDER AND
620:   C84A 8D 21 D0           STA  BGROUND  ;BACKGROUND COLOR
630:   C84D A5 FD              LDA  UP       ;UPPER VALUE
640:   C84F 8D 12 D0           STA  RASTER   ;IN RASTER
650:   C852 4C BC FE           JMP  $FEBC    ;POSITION
]C800-C855
NO ERRORS
```

For those who do not have an assembler or monitor, we include what are called "BASIC loader programs" for all of the assembly language programs in this book. These loader programs POKE the necessary machine codes into memory. A check is provided to ensure that the DATA has been entered correctly. Here is the BASIC loader program for the raster line IRQ, which also includes a small application:

```
1000 FOR I=51200 TO 51284
1010 READ X : POKE I,X : S=S+X : NEXT
```

```
1020 DATA 120,169, 31,141, 20,  3,169,200,141, 21,  3,165
1030 DATA 253,141, 18,208,173, 17,208, 41,127,141, 17,208
1040 DATA 169,129,141, 26,208, 88, 96,173, 25,208,141, 25
1050 DATA 208, 48,  7,173, 13,220, 88, 76, 49,234,173, 18
1060 DATA 208,197,254,176, 16,165,251,141, 32,208,141, 33
1070 DATA 208,165,254,141, 18,208, 76,188,254,165,252,141
1080 DATA  32,208,141, 33,208,165,253,141, 18,208, 76,188
1090 DATA 254
1100 IF S <> 11288 THEN PRINT "ERROR IN DATA !!" : END
1110 PRINT "OK"
1120 F1 = 7 : F2 = 6 : REM COLORS   AND 2 (BANDS IN COLOR 1)
1130 UP = 60 : DN = 150 : REM UPPER UND LOWER EDGES OF THE
     BAND
1140 POKE 251,F1:POKE 252,F2:POKE 253,UP:POKE 254,DN : REM
     PASS
1150 SYS 51200 : REM INITIALIZE ROUTINE
1160 REM
1170 FOR X=1 TO 5000 : NEXT X : REM DELAY LOOP
1180 REM
1190 REM MOVE:
1200 REM *****
1210 FOR X=40 TO 240 : POKE 253,X : POKE 254, X+10 : NEXT X
1220 GET A$ : IF A$="" THEN 1190 : REM KEY?
```

First a few things about the machine language program.

The program creates a yellow band across a blue background on the screen. This is possible by continually switching the screen and border colors every time the video controller reaches certain raster lines.

Before our own interrupt routine is called by the processor, we must take care of a few things. This is done in the initialization routine. First, the IRQ vector at $0314/$0315 (788/789), which normally points to the original

interrupt routine in ROM at $EA31 (59953), is directed to our new interrupt service routine (here at $C81F (51231). For this purpose the high and low bytes of the new address are written to this vector (lines 270-300). So that an interrupt does not occur while we are doing this (interrupts normally occur every 1/60th of a second in order to read the keyboard, etc), we must first disable the interrupts by setting the interrupt flag in the microprocessor with the SEI instruction (line 260).

Next, the raster line at which the first raster interrupt should occur is determined. This is the upper edge of the yellow stripe and is stored in zero-page location $FD (253), in which a BASIC program can write the desired value. The value is now written into the raster line register (VIC register 18) in lines 310/320. The first interrupt will then occur when the video controller is building this line. Since we do not use the high bit here, it is cleared in lines 330-350.

The next important step is enabling the interrupts by setting bit 0 of the IMR, the interrupt mask. This causes an interrupt to be generated when the corresponding event occurs.

At the end of our initialization, the I flag is again cleared so that the interrupts can be serviced (CLI instruction).

The problem with interrupt handling is that there are now two sources from which an interrupt can be generated:

- timer IRQ
- raster IRQ

Those who write their programs in machine language and can replace or do without such functions as the keyboard polling, cursor blinking, or TI$-clock handling are naturally free to do away with the timer interrupt. For our own purposes, we must determine which event generated the interrupt because the same routine is also called by the CIA 1 system interrupt timer. Therefore we first load the IRR and write this same value back into it in order to clear the interrupt (otherwise another interrupt would be generated immediately after our routine ended, ad infinitum). If a raster IRQ was generated, bits 0 and 7 of the IRR will be set (see section 3.7). Bit 7 is checked in line 460 and a branch is made to the second part (line 510) if it is set. If the IRQ was caused by the timer, we want to jump to the normal (original) interrupt routine which begins at $EA31 (59953).

Here however we encounter a small problem. For each interrupt, the 6510 processor automatically saves the return address and the processor status register on the stack and--most important for us--sets the interrupt flag so that no interrupts may occur during our interrupt service routine. Let's assume that we branch to $EA31 as normal and the execute the necessary things. A raster line interrupt may be generated during the course of this routine. But because the I flag is set (disabling interrupts), this interrupt is not serviced. This problem results in a brief but annoying flashing on the screen because the background and border are not always switched at an early enough time. Therefore we must enable interrupts during the execution of the normal interrupt routine and allow it to be interrupted in turn by other IRQs. This sounds very complicated and somewhat "dangerous," but it is quite valid and even logical, if you think about it.

We can simply clear the I flag before the jumping to the normal interrupt routine and everything will be in order. But we have forgotten something again. Before the flag is cleared we must first--as always--remove the source of the interrupt (in this case the timer). This is again done by clearing the IRR. The IRR is in the register set of the CIA 1 and has the address $DC0D (56333). Clearing it is done somewhat differently. All that is necessary is to read the register, as is done in line 480. Now we have done everything required for problemless operation and can continue with our own interrupt routine (at line 510).

We will be making two switches between background colors, once from color 1 to color 2 and then back again from color 2 to color 1. We must first check to see which phase we are in. This is done by reading the current raster line (line 510) in order to compare it with the lower edge of our yellow band (line 520). If the current line is greater than or equal to this value, we must switch to the second value, jumping to line 600. We can also theoretically check which color the background is, which also has its advantages. (Note: Because it takes a while until the program reaches this exact spot after the IRQ, the raster beam will have moved several lines down. Therefore we do not have the exact line at which the interrupt occurred. Remember that before our routine is called, some instructions are executed in ROM which also take time. This is the reason that the place at which the color change occurs does not entirely agree with the desired position.)

The rest is relatively simple: We change the border and background colors and set the next raster interrupt to the upper or lower border value (whichever it isn't, currently). At the end we jump to the original interrupt routine because

some registers must be restored and an RTI (ReTurn from Interrupt) executed.

The four necessary pieces of information which define the band are stored in zero-page locations:

Address (hex-dec)	Contents
$FB - 251	color 1
$FC - 252	color 2
$FD - 253	upper border
$FE - 254	lower border

By changing the contents of these locations you can set the color of the band, the background color, and the position of the upper and lower borders. This is best done with a small BASIC program such as the one given above. It loads the machine language program, sets the four parameters and executes the initialization routine. An example of this routine's use is given at line 1200. Perhaps you can write a program which allows you to highlight individual lines on the screen, under cursor control. Or, if you are well-acquainted with machine language, you could switch between text and graphics displays instead of background colors. Have you ever seen 16 sprites or two character sets on the screen at the same time? There are innumerable possibilities.

4.6.2 Light pen

Please read this section, even if you do not own a light pen.

You already know what a light pen is, how it works and how it is integrated into the operation of the computer from section 3.7.2. The most important concepts are repeated here:

The light pen is an instrument with which the computer can identify the position on the screen to which it is pointed. The x and y coordinates of this point can be determined from VIC registers 19 and 20. The coordinates correspond to the raster lines (see Section 3.7). The light pen can also generate an interrupt. Bit 3 of the IMR and IRR pertain to the light pen. Its connection to control port 1 is the same as the connection of the joystick fire button on this port.

How is a light pen used and how is it programmed?

The light pen can be read in two different ways. The simplest is through a small BASIC program which simply reads the two registers which contain the screen coordinates. An example would be to draw a dot at the point at which the light pen is pointed. The following demonstration program performs this function. It must be used in conjunction with the graphics routines in section 4.2.

```
100 REM ****************
110 REM **              **
120 REM **  LIGHTPEN    **
130 REM **              **
140 REM ****************
150 REM
160 V = 53248 : REM VIC BASE ADDRESS
170 SA = 8192 : REM GRAPHIC]START ADDRESS
180 GOSUB 10000:GOSUB 10200:FA=7*16+2:GOSUB 10400 : REM
    INITIALIZE GRAPHICS
190 XP = PEEK(V+19) : REM LIGHTPEN X-COORDINATE
```

```
200 YP = PEEK(V+20) : REM LIGHTPEN Y-COORDINATE
210 XC = 2 * (XP - 40) : REM CALCULATE X-COORD.
220 YC = YP - 40 : REM CALCULATE Y-COORD.
230 IF XC>319 OR XC<0 OR YC>199 OR YC<0 THEN 190 : REM TEST
    RANGE
240 GOSUB 10700 : REM DRAW POINT
250 GOTO 190
10000 REM ************************
10010 REM **                      **
10020 REM **  TURN ON GRAPHICS    **
10030 REM **                      **
10040 REM ************************
10050 REM
10060 V = 53248 : REM BASE ADDRESS - VIDEO CONTROLLER
10070 POKE V+17, PEEK(V+17) OR (8+3)*16 : REM GRAPHICS ON
10080 POKE V+22, PEEK(V+22) AND 255-16 : REM MULTICOLOR ON
10090 POKE V+24, PEEK(V+24) OR 8 : REM GRAPHICS TO $2000
      (8192)
10095 RETURN
10100 REM
10110 REM
10200 REM  ****************************
10210 REM  **                        **
10220 REM  **  CLEAR GRAPHIC SCREEN  **
10230 REM  **                        **
10240 REM  ****************************
10250 REM
10260 BG = 8192 : REM BASE ADDRESS OF GRAPHICS MEMORY
10270 FOR X=BG TO BG+8000 : REM 8000 BYTES
10280 POKE X,0 : REM ERASE
10290 NEXT X : RETURN
10300 REM
10310 REM
```

```
10400 REM   ********************
10410 REM   **                **
10420 REM   **  CLEAR COLOR   **
10430 REM   **                **
10440 REM   ********************
10450 REM
10460 BV = 1024 : REM BASE ADDRESS OF VIDEO RAM
10470 CO = 6*16+7 : REM POINT COLOR=BLUE/BACKGROUND=YELLOW
10480 FOR X=BV TO BV+1000 : REM 1000 BYTES
10490 POKE X, CO : REM WITH POINT AND BACKGROUND GOLOR
10500 NEXT X : RETURN
10510 REM
10520 REM
10600 REM ************************
10610 REM **                    **
10620 REM **  TURN OFF GRAPHICS  **
10630 REM **                    **
10640 REM ************************
10650 REM
10660 V = 53248 : REM BASE ADDRESS - VIDEO CONTROLLER
10670 POKE V+17, PEEK(V+17) AND 255-6*16 : REM GRAPHICS OFF
10680 POKE V+22, PEEK(V+22) AND 255-16 : REM MULTICOLOR OFF
10690 POKE V+24, PEEK(V+24) AND 255-8 : REM CHARACTER SET
      BACK TO $1000 (4096)
10695 RETURN
10700 REM ***********************
10710 REM **                   **
10720 REM **  CALCULATE POINT  **
10730 REM **     (SETTING)     **
10740 REM ***********************
10750 REM
10760 RA = 320 * INT(YC/8) + (YC AND 7)
10770 BA = 8 * INT(XC/8)
```

```
10780 MA = 2(7-(XC AND 7))
10790 AD = SA + RA + BA
10800 POKE AD, PEEK(AD) OR MA : RETURN
10810 REM
10900 REM ************************
10910 REM **                    **
10920 REM **  POINT CALCULTAION  **
10930 REM **     (CLEARING)     **
10940 REM ************************
10950 REM
10960 RA = 320 * INT(YC/8) + (Y AND 7)
10970 BA = 8 * INT(XC/8)
10980 MA = 255 - 2(7-(XC AND 7))
10990 AD = SA + RA + BA
11000 POKE AD, PEEK(AD) AND LA : RETURN
11010 REM
11020 REM INTERNAL PARAMETERS:
11030 REM ********************
11040 REM RA: ROW ADDRESS
11050 REM BA: BYTE ADDRESS
11060 REM MA: MASK
11070 REM AD: DESTINTAION ADDRESS
11080 REM
11090 REM EXTERNAL PARAMETERS:
11100 REM ********************
11110 REM SA: START OF GRAPHICS MEMORY (SUCH AS 8192)
11120 REM XC: X-COORDINATE
11130 REM YC: Y-COORDINATE
```

In this program, the screen coordinates of the point are read in lines 190 and 200 after the graphics are initialized and the graphics screen is cleared. These screen coordinates are converted to our normal graphics coordinates

by the formula given in section 3.7.2 (lines 210/220). The point must then be checked to ensure that it does not lie outside of the screen window, indicated by a negative value or one that is too large. Now the point-set routine can be called to draw a point at that place on the screen. You can expand upon this program as desired with many functions, building it up to entire drawing program. The light pen could be used to determine end points of lines or the mid-point and radius of a circle. You can also use the light pen to select menu items, and so on.

There is one problem, however. A point is drawn at the spot indicated whether or not the light pen is actually pointed at the screen. This is not so bad here, but it could cause errors in other programs.

Another option is to read the light pen by servicing the interrupt. Every time the light pen sends an impulse (only when the light pen is actually pointing at the screen) an interrupt can be generated. The interrupt routine could then draw a point on the screen.

Still another possibility is to use the light pen an alarm: the light pen can monitor a light. When someone enters and breaks the beam between the light and the light pen, the light pen sends an impulse to the computer which can then sound an alarm (perhaps using the internal synthesizer connected to a stereo system for greater volume).

Instead of an alarm, we will have to be satisfied with changing the color of the screen border:

```
100:   C800                    *=    $C800
110:   00FB           COLOR    =     $FB
120:   0314           IRQVECT  =     $0314
130:   EA31           IRQOLD   =     $EA31
140:   D019           IRR      =     $D019
150:   D01A           IMR      =     $D01A
160:   D020           BORDER   =     $D020
170:                  ;
180:                  ;INITIALIZE
190:                  ;**********
200:                  ;
210:   C800 78        INIT     SEI           ;DISABLE
                                              INTERRUPTS
220:   C801 A9 16              LDA   #< IRQNEW
230:   C803 8D 14 03           STA   IRQVECT
240:   C806 A9 C8              LDA   #> IRQNEW
250:   C808 8D 15 03           STA   IRQVECT+1  ;REPLACE IRQ
                                                 VECTOR
260:   C80B A9 00              LDA   #$00
270:   C80D 85 FB              STA   COLOR    ;COLOR BLACK
280:   C80F A9 88              LDA   #%10001000  ;MASK
290:   C811 8D 1A D0           STA   IMR      ;SELECT LIGHTPEN
                                               IRQ
300:   C814 58                 CLI
310:   C815 60                 RTS
320:                  ;
330:                  ;INTERRUPT ROUTINE
340:                  ;******************
350:                  ;
360:   C816 AD 19 D0 IRQNEW    LDA   IRR      ;INTERRUPT
                                               REGISTER
370:   C819 8D 19 D0           STA   IRR      ;ERASE
380:   C81C 30 07              BMI   IRQRAS   ;RASTER LINE-IRQ?
```

```
390:                    ;NORMAL IRQ
400:    C81E AD 0D DC           LDA  $DC0D    ;ERASE CIA 1-IRR
410:    C821 58                 CLI           ;ENABLE INTER-
                                               RUPTS
420:    C822 4C 31 EA           JMP  IRQOLD   ;TO OLD ROUTINE
430:    C825 A5 FB     IRQRAS   LDA  COLOR
440:    C827 8D 20 D0           STA  BORDER   ;BORDER COLOR
450:    C82A E6 FB              INC  COLOR    ;INCREMENT COLOR
460:    C82C 4C BC FE           JMP  $FEBC    ;REPLACE
]C800-C82F
NO ERRORS
```

And here is the loader program:

```
100 FOR I = 51200 TO 51246
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 120,169, 22,141, 20,  3,169,200,141, 21,  3,169
130 DATA   0,133,251,169,136,141, 26,208, 88, 96,173, 25
140 DATA 208,141, 25,208, 48,  7,173, 13,220, 88, 76, 49
150 DATA 234,165,251,141, 32,208,230,251, 76,188,254
160 IF S <> 5910 THEN PRINT "ERROR IN DATA   !!" : END
170 PRINT "OK"
```

It is started as usual with

```
SYS 51200
```

All of the line numbers in the following discussion refer to the assembly language source listing:

The initialization routine which begins in line 220 should already be familiar to you. First the IRQ vector is changed to point to our own routine and the corresponding interrupt is turned on by setting bit 4 of the IMR (Inter-

rupt Mask Register). Our interrupt routine at line 360 contains nothing special. The interrupt is again checked and the corresponding branches are made as required. The procedure is easy to understand, but it is important to note that for the first time the interrupt routine changes a normal memory location. This brings about our desired effect but it can be dangerous if one does not know exactly what is being done with this location at all times, because an interrupt can be generated at any time and within any routine. It has the advantage that the results of an interrupt routine can be read by a BASIC program, as is done, for instance, with the TI$ clock.

Remember that this screen flashing can also be created by pressing the fire button of a joystick inserted in control port 1.

You can see that interrupts and interrupt control permit a number of very useful and desirable possibilities, and, although they can be difficult to work with, the effort will often pay off.

4.7 A small graphics package

We have heard and read quite a bit about graphics and graphics programming. After trying the sample programs that we so carefully typed in, we may have been surprised at how slowly they ran - even after optimizing them using the tips from section 6.1. We became very impatient as we waited for them to appear. These BASIC programs demonstrate the techniques but keep us waiting to long.

We will now try to rectify this. Here is a relatively comprehensive graphics package written in machine language that will make your graphics faster and subsequently more interesting.

If you browse through the next pages, you'll see that it is quite a lot of work to type in and debug the whole assembly language listing. The advantages are truly tremendous, however. Anyone who is really serious about working with graphics needs the appropriate tools.

You have these choices:

1) You can continue to use the simple but slow BASIC routines;
2) you can take the time and type in the following listing;
3) you can purchase a commercial graphics extension package which does the work for you;
4) you can purchase the optional program diskette for this book which contains all of the programs and listings in this book.

The option of ignoring graphics entirely is not recommended.

Back to our topic: We present the assembly language

listing of the graphics package with many comments for those who are interested in the individual routines. You will notice that some things are organized differently (in the interest of speed) than they were in the BASIC versions.

After this source listing is the BASIC loader for those who do not have an assembler.

```
10:    C800                    *=    $C800
40:                    ;
50:                    ; **********************
60:                    ; **                    **
70:                    ; ** HIGH RESOLUTION   **
80:                    ; ** GRAPHICS PACKAGE  **
90:                    ; **                    **
100:                   ; **********************
110:                   ;
120:                   ;
130:                   ;
140:                   ; ROM JUMP ADDRESSES
150:                   ; ********************
160:                   ;
170:   B79E            GETBYT    =     $B79E    ;GET BYTE VALUE
180:   AEFD            CHKCOM    =     $AEFD    ;CHECK FOR COMMA
190:   B7F1            CHKGET    =     $B7F1    ;CHKCOM+GETBYT
200:   B7EB            GETCOR    =     $B7EB    ;GET COORDINATES
210:   B248            QERR      =     $B248    ;ILLEGAL QUANTITY
220:   D000            V         =     $D000    ;VIDEO CONTROLLER
230:                   ;
240:                   ;ZERO PAGE REGISTERS
250:                   ;*********************
260:                   ;
270:   0063            OFFX      =     $63
280:   00AB            MSC       =     $AB      ;MASK
290:   0069            DIF0      =     $69
300:   006A            DIF1      =     $6A
310:   006B            DIF2      =     $6B
320:   006C            DIF3      =     $6C
330:   006D            DIF4      =     $6D
340:   006E            DIF5      =     $6E
350:   006F            TEMP      =     $6F
```

```
360:   0070          CTR       =    $70
370:   00AC          A         =    $AC
380:   00AD          B         =    $AC+1
390:   0014          XC        =    $14
400:   0097          FLG       =    $97
410:   00FD          USE       =    $FD      ;MISC
420:                 ;
430:                 ;CONSTANTS
440:                 ;*********
450:                 ;
460:   2000          GRAPH     =    $2000    ;START OF GRAPHICS
470:   0400          VIDEO     =    $0400    ;START OF VIDEO
                                               RAM
480:   0021          GSTART    =    $21      ;GRSTART+1-MSB
490:   003E          GEND      =    $3E      ;GREND-1-MSB
500:                 ;
510:                 ;
520:                 ;CONTROL ADDRESSES
530:                 ;******************
540:                 ;
550:   C800 4C 24 C8          JMP  INIT     ;GRAPHICS ON
560:   C803 4C 41 C8          JMP  GOFF     ;GRAPHICS OFF
570:   C806 4C 12 C9          JMP  GCLEAR   ;CLEAR SCREEN
580:   C809 4C 31 C9          JMP  SCOLOR   ;SET COLOR(CLEAR)
590:   C80C 4C 2A C9          JMP  PCOLOR   ;PLOT COLOR
600:   C80F 4C 58 C8          JMP  PLOT     ;SET POINT
610:   C812 4C 55 C8          JMP  UNPLOT   ;ERASE POINT
620:   C815 4C 6B C8          JMP  SLINE    ;DRAW LINE
630:   C818 4C 68 C8          JMP  CLLINE   ;ERASE LINE
640:   C81B 4C 43 CA          JMP  GLOAD    ;LOAD GRAPHICS
650:   C81E 4C 52 CA          JMP  GSAVE    ;SAVE GRAPHICS
660:   C821 4C 69 CA          JMP  HARDC    ;HARDCOPY
                                               (CBM 1525, ETC.)
```

```
670:                    ;
680:                    ;
690:                    ;TURN GRAPHICS ON
700:                    ;*****************
710:                    ;
720:    C824 EA         INIT     NOP          ;NO BLOCKADE
730:    C825 AD 11 D0            LDA  V+17
740:    C828 8D 1E CB            STA  STORE1
750:    C82B AD 18 D0            LDA  V+24
760:    C82E 8D 1F CB            STA  STORE2
770:    C831 A9 3B               LDA  #%00111011
780:    C833 8D 11 D0            STA  V+17    ;SAVE ALL CONTENTS
790:    C836 A9 18               LDA  #%00011000
800:    C838 8D 18 D0            STA  V+24    ;TO $2000
810:    C83B A9 60               LDA  #$60    ;$60 FOR RTS AS
                                               BLOCKADE
820:    C83D 8D 24 C8            STA  INIT    ;INIT WILL BE
                                               BLOCKED
830:    C840 60                  RTS
840:                    ;
850:                    ;
860:                    ;TURN GRAPHICS OFF
870:                    ;*****************
880:                    ;
890:    C841 AD 1E CB GOFF       LDA  STORE1
900:    C844 8D 11 D0            STA  V+17
910:    C847 AD 1F CB            LDA  STORE2
920:    C84A 8D 18 D0            STA  V+24
930:    C84D A9 EA               LDA  #$EA    ;CODE FOR NOP
940:    C84F 8D 24 C8            STA  INIT    ;FOR BLOCKADE
950:    C852 4C 44 E5            JMP  $E544   ;CLEAR SCREEN
960:                    ;
970:                    ;
```

```
980:                    ;ERASE POINT
990:                    ;************
1000:                   ;
1010:  C855 A2 00       UNPLOT    LDX  #$00     ;ERASE FLAG
1020:  C857 2C                    .BYTE$2C
1030:                   ;
1040:                   ;
1050:                   ;SET POINT
1060:                   ;**********
1070:                   ;
1080:  C858 A2 80       PLOT      LDX  #$80     ;SET FLAG
1090:  C85A 86 97       PL1       STX  FLG
1100:  C85C 20 FD AE              JSR  CHKCOM
1110:  C85F 20 79 C8              JSR  TESCOR   ;GET COORDINATES
1120:  C862 20 94 C8              JSR  HPOSN    ;CALCULATE ADDRESS
1130:  C865 4C E2 C8              JMP  PLT      ;SET/ERASE POINT
1140:                   ;
1150:                   ;
1160:                   ;DRAW LINE
1170:                   ;**********
1180:                   ;
1190:  C868 A2 00       CLLINE    LDX  #$00     ;ERASE-FLAG
1200:  C86A 2C                    .BYTE$2C
1210:                   ;
1220:                   ;
1230:                   ;DRAW LINE
1240:                   ;**********
1250:                   ;
1260:  C86B A2 80       SLINE     LDX  #$80     ;SET-FLAG
1270:  C86D 20 5A C8              JSR  PL1      ;SET FIRST POINT
1280:  C870 20 FD AE              JSR  CHKCOM
1290:  C873 20 79 C8              JSR  TESCOR   ;GET SECOND COORD.
1300:  C876 4C BB C9              JMP  HLINE    ;DRAW LINE
```

```
1310:                 ;
1320:                 ;
1330:                 ;TEST COORDINATES
1340:                 ;*****************
1350:                 ;
1360:  C879 20 EB B7 TESCOR   JSR  GETCOR  ;GET
1370:  C87C 8A               TXA
1380:  C87D A8               TAY
1390:  C87E A6 15            LDX  XC+1
1400:  C880 C0 C8            CPY  #200    ;Y-COORD. >=200?
1410:  C882 B0 0D            BCS  ILLFF   ;YES!
1420:  C884 A5 14            LDA  XC
1430:  C886 E0 01            CPX  #>320   ;X-COORD. >=320?
1440:  C888 90 06            BCC  T1      ;YES
1450:  C88A D0 05            BNE  ILLFF   ;A XC-LOW
1460:  C88C C9 40            CMP  #<320   ;X XC-HIGH
1470:  C88E B0 01            BCS  ILLFF   ;Y YC
1480:  C890 60         T1     RTS
1490:  C891 4C 48 B2 ILLFF    JMP  QERR    ;ILLEGAL QUANTITY
1500:                 ;
1510:                 ;
1520:                 ;CALCULATE ADDRESS
1530:                 ;******************
1540:                 ;
1550:  C894 8C 1C CB HPOSN    STY  YC      ;Y-C
1560:  C897 8D 1A CB          STA  XCL     ;X-CL
1570:  C89A 8E 1B CB          STX  XCH     ;X-CH(TEMPORARY
                                            STORAGE)
1580:  C89D 85 14             STA  XC
1590:  C89F 86 15             STX  XC+1
1600:  C8A1 98                TYA
1610:  C8A2 4A                LSR  A
1620:  C8A3 4A                LSR  A
```

```
1630:  C8A4 4A          LSR  A        ;INT(Y/8)
1640:  C8A5 AA          TAX
1650:  C8A6 BD 2F CB    LDA  MULH,X   ;320*INT(Y/8)
                                       (LOW BYTE)
1660:  C8A9 85 AD       STA  B
1670:  C8AB 8A          TXA
1680:  C8AC 29 03       AND  #3       ;ISOLATE BITS 0
                                       AND 1
1690:  C8AE AA          TAX
1700:  C8AF BD 49 CB    LDA  MULL,X   ;320*INT(Y/8)
                                       (LOW BYTE)
1710:  C8B2 85 AC       STA  A
1720:  C8B4 98          TYA           ;Y-COORD.
1730:  C8B5 29 07       AND  #7       ;(Y AND 7)
1740:  C8B7 18          CLC
1750:  C8B8 65 AC       ADC  A        ;OFF Y=320*INT
                                       (Y/8)+(Y AND 7)
1760:  C8BA 85 AC       STA  A        ;***************
1770:  C8BC A5 14       LDA  XC
1780:  C8BE 29 F8       AND  #$F8     ;OFFX=8*INT(X/8)
1790:  C8C0 85 63       STA  OFFX     ;***************
1800:  C8C2 A9 20       LDA  #>GRAPH  ;GRAPHICS PAGE
1810:  C8C4 05 AD       ORA  B
1820:  C8C6 85 AD       STA  B        ;+SA
1830:  C8C8 18          CLC
1840:  C8C9 A5 AC       LDA  A
1850:  C8CB 65 63       ADC  OFFX     ;AD=OFFY+OFFX+SA
1860:  C8CD 85 AC       STA  A        ;***************
1870:  C8CF A5 AD       LDA  B
1880:  C8D1 65 15       ADC  XC+1
1890:  C8D3 85 AD       STA  B
1900:  C8D5 A5 14       LDA  XC
1910:  C8D7 29 07       AND  #7
```

```
1920:  C8D9 49 07          EOR  #7       ;7-(X AND 7)
1930:  C8DB AA             TAX
1940:  C8DC BD 4D CB       LDA  MSCTAB,X  ;MASK TABLE
1950:  C8DF 85 AB          STA  MSC      ;2(7-(X AND 7))
1960:  C8E1 60             RTS
1970:                ;
1980:                ;
1990:                ;DOT POINT
2000:                ;**********
2010:                ;
2020:  C8E2 A0 00    PLT   LDY  #0
2030:  C8E4 08             PHP           ;SAVE CARRY FLAG
2040:  C8E5 A5 AB          LDA  MSC      ;MASK
2050:  C8E7 24 97          BIT  FLG      ;PLOT/UNPLOT FLAG
2060:  C8E9 30 05          BMI  PL2      ;PLOT
2070:  C8EB 49 FF          EOR  #$FF     ;UNPLOT
2080:  C8ED 31 AC          AND  (A),Y
2090:  C8EF 2C             .BYTE$2C      ;SKIP NEXT COMMAND
2100:  C8F0 11 AC    PL2   ORA  (A),Y    ;PLOT
2110:  C8F2 91 AC          STA  (A),Y
2120:  C8F4 A5 AC          LDA  A        ;SET COLOR
2130:  C8F6 85 FD          STA  USE
2140:  C8F8 A5 AD          LDA  B
2150:  C8FA 4A             LSR  A
2160:  C8FB 66 FD          ROR  USE
2170:  C8FD 4A             LSR  A
2180:  C8FE 66 FD          ROR  USE
2190:  C900 4A             LSR  A
2200:  C901 66 FD          ROR  USE      ;ADDRESS/8
2210:  C903 29 03          AND  #$03
2220:  C905 09 04          ORA  #$04     ;VIDEO RAM AT
                                          $0400
2230:  C907 85 FE          STA  USE+1
```

```
2240:  C909 AD 1D CB          LDA  COLOR    ;COLOR IN
2250:  C90C 91 FD             STA  (USE),Y  ;VIDEO RAM
2260:  C90E 28                PLP           ;CARRY FLAG
2270:  C90F A4 6F             LDY  TEMP
2280:  C911 60                RTS
2290:                ;
2300:                ;
2310:                ;CLEAR GRAPHICS SCREEN
2320:                ;**********************
2330:                ;
2340:  C912 A9 20    GCLEAR   LDA  #>GRAPH
2350:  C914 85 FE             STA  USE+1
2360:  C916 A0 00             LDY  #<GRAPH  ;Y=0
2370:  C918 84 FD             STY  USE      ;GRAPHIC START
                                             ADDRESS
2380:  C91A A2 20             LDX  #$20     ;LENGTH
2390:  C91C 98                TYA           ;Y=0
2400:  C91D 91 FD    GC1      STA  (USE),Y  ;ERASE
2410:  C91F C8                INY
2420:  C920 D0 FB             BNE  GC1
2430:  C922 E6 FE             INC  USE+1
2440:  C924 CA                DEX
2450:  C925 D0 F6             BNE  GC1
2460:  C927 4C 37 C9          JMP  SCOL1    ;ERASE VIDEO RAM
2470:                ;
2480:                ;
2490:                ;DEFINE POINT COLOR
2500:                ;*******************
2510:                ;
2520:  C92A 20 F1 B7 PCOLOR   JSR  CHKGET   ;COMMA + COLOR
2530:  C92D 8E 1D CB          STX  COLOR
2540:  C930 60                RTS
2550:                ;
```

```
2560:                 ;
2570:                 ;SET COLOR
2580:                 ;**********
2590:                 ;
2600:  C931 20 F1 B7 SCOLOR   JSR  CHKGET  ;COMMA + COLOR
2610:  C934 8E 1D CB          STX  COLOR
2620:  C937 A2 03    SCOL1    LDX  #3
2630:  C939 A9 04             LDA  #>VIDEO
2640:  C93B 85 FE             STA  USE+1
2650:  C93D A0 00             LDY  #<VIDEO
2660:  C93F 84 FD             STY  USE     ;ADDRESS OF VIDEO
                                            RAM
2670:  C941 84 97             STY  FLG     ;Y=0
2680:  C943 AD 1D CB          LDA  COLOR   ;COLOR
2690:  C946 91 FD    GM9      STA  (USE),Y ;SET
2700:  C948 C8                INY
2710:  C949 C4 97             CPY  FLG
2720:  C94B D0 F9             BNE  GM9
2730:  C94D E6 FE             INC  USE+1
2740:  C94F CA                DEX
2750:  C950 F0 03             BEQ  GM9.
2760:  C952 10 F2             BPL  GM9
2770:  C954 60                RTS
2780:  C955 A2 E8    GM9.     LDX  #$E8
2790:  C957 86 97             STX  FLG     ;PROTECT SPRITE
                                            VECTORS
2800:  C959 D0 EB             BNE  GM9     ;ABSOLUTE
2810:                 ;
2820:                 ;
2830:                 ;VECTOR ROUTINES
2840:                 ;****************
2850:                 ;
2860:  C95B A5 AC    DOWN     LDA  A       ;POINT ADDRESS LOW
```

```
2870:  C95D 29 07          AND  #7
2880:  C95F C9 07          CMP  #7
2890:  C961 F0 05          BEQ  DN1      ;LOWER BORDER
2900:  C963 38             SEC
2910:  C964 A9 00          LDA  #0
2920:  C966 B0 04          BCS  DN2      ;ADD 1 (C=1!)
2930:  C968 A9 38    DN1   LDA  #$38     ;ADD 320-7=313
2940:  C96A E6 AD          INC  B        ;C=1!
2950:  C96C 65 AC    DN2   ADC  A
2960:  C96E 85 AC          STA  A
2970:  C970 A9 00          LDA  #0
2980:  C972 65 AD          ADC  B
2990:  C974 85 AD          STA  B
3000:  C976 60             RTS
3010:                ;
3020:  C977 30 E2    UD    BMI  DOWN
3030:                ;
3040:  C979 A5 AC    UP    LDA  A        ;ADDRESS
3050:  C97B 29 07          AND  #7
3060:  C97D F0 05          BEQ  UP1      ;UPPER BORDER
3070:  C97F 18             CLC
3080:  C980 A9 FF          LDA  #$FF
3090:  C982 90 04          BCC  UP2      ;SUBTRACT 1
3100:  C984 A9 C7    UP1   LDA  #$C7     ;SUBTRACT
                                          320+7=327
3110:  C986 C6 AD          DEC  B        ;C=1!
3120:  C988 65 AC    UP2   ADC  A
3130:  C98A 85 AC          STA  A
3140:  C98C A5 AD          LDA  B
3150:  C98E E9 00          SBC  #0
3160:  C990 85 AD          STA  B
3170:  C992 60             RTS
3180:                ;
```

```
3190:                 ;
3200:  C993 46 AB     RIGHT     LSR  MSC      ;SHIFT MASK
3210:  C995 90 0E               BCC  RT2      ;INNER BYTE
3220:  C997 66 AB               ROR  MSC      ;OUTER BYTE
3230:  C999 A5 AC               LDA  A
3240:  C99B C8                  INY
3250:  C99C 18                  CLC
3260:  C99D 69 08               ADC  #8
3270:  C99F 85 AC               STA  A
3280:  C9A1 90 02               BCC  RT2
3290:  C9A3 E6 AD               INC  B        ;ADD 8
3300:  C9A5 60        RT2       RTS
3310:                 ;
3320:  C9A6 10 EB     RL        BPL  RIGHT
3330:                 ;
3340:  C9A8 06 AB     LEFT      ASL  MSC
3350:  C9AA 90 0E               BCC  LT1
3360:  C9AC 26 AB               ROL  MSC
3370:  C9AE A5 AC               LDA  A
3380:  C9B0 88                  DEY
3390:  C9B1 38                  SEC
3400:  C9B2 E9 08     LT3       SBC  #8
3410:  C9B4 85 AC               STA  A
3420:  C9B6 B0 02               BCS  LT1
3430:  C9B8 C6 AD               DEC  B        ;SUBTRACT 8
3440:  C9BA 60        LT1       RTS
3450:                 ;
3460:                 ;
3470:                 ;DRAW LINE
3480:                 ;**********
3490:                 ;
3500:  C9BB 48        HLINE     PHA           ;A    X2 LOW BYTE
3510:  C9BC AD 1B CB            LDA  XCH      ;X    X2 HIGH BYTE
```

```
3520:  C9BF 4A              LSR  A       ;Y    Y2
3530:  C9C0 AD 1A CB        LDA  XCL     ;XCL  X1 LOW BYTE
3540:  C9C3 6A              ROR  A       ;XCH  X1 LOW BYTE
3550:  C9C4 4A              LSR  A       ;YC   Y1
3560:  C9C5 4A              LSR  A
3570:  C9C6 85 6F           STA  TEMP    ;X1 / 8 TEMP
                                          STORAGE
3580:  C9C8 68              PLA
3590:  C9C9 48              PHA
3600:  C9CA 38              SEC
3610:  C9CB ED 1A CB        SBC  XCL
3620:  C9CE 48              PHA
3630:  C9CF 8A              TXA
3640:  C9D0 ED 1B CB        SBC  XCH
3650:  C9D3 85 6C           STA  DIF3    ;X2-X1
3660:  C9D5 B0 0A           BCS  L3      ;NEGATIVE?
3670:  C9D7 68              PLA          ;YES
3680:  C9D8 49 FF           EOR  #$FF
3690:  C9DA 69 01           ADC  #1
3700:  C9DC 48              PHA
3710:  C9DD A9 00           LDA  #0
3720:  C9DF E5 6C           SBC  DIF3    ;CHANGE IN SIGN
3730:  C9E1 85 6A     L3    STA  DIF1
3740:  C9E3 85 6E           STA  DIF5    ;(X2-X1) HIGH
3750:  C9E5 68              PLA
3760:  C9E6 85 69           STA  DIF0
3770:  C9E8 85 6D           STA  DIF4    ;(X2-X1) LOW
3780:  C9EA 68              PLA
3790:  C9EB 8D 1A CB        STA  XCL
3800:  C9EE 8E 1B CB        STX  XCH
3810:  C9F1 98              TYA
3820:  C9F2 18              CLC
3830:  C9F3 ED 1C CB        SBC  YC      ;Y2-Y1
```

```
3840:  C9F6 90 04          BCC  L4      ;NEGATIVE?
3850:  C9F8 49 FF          EOR  #$FF
3860:  C9FA 69 FE          ADC  #$FE    ;SIGN CHANGE
3870:  C9FC 85 6B    L4    STA  DIF2    ;(Y2-Y1)
3880:  C9FE 8C 1C CB        STY  YC
3890:  CA01 66 6C           ROR  DIF3    ;(X2-X1)/2
3900:  CA03 38              SEC
3910:  CA04 E5 69           SBC  DIF0    ;(Y2-Y1)-(X2-X1)
3920:  CA06 AA              TAX          ;LOW BYTE TO X REG
3930:  CA07 A9 FF           LDA  #$FF
3940:  CA09 E5 6A           SBC  DIF1
3950:  CA0B 85 70           STA  CTR     ;HIGH BYTE TO CTR
3960:  CA0D A4 6F           LDY  TEMP
3970:  CA0F B0 05           BCS  L5      ;ABSOLUTE
3980:  CA11 0A        L1    ASL  A
3990:  CA12 20 A6 C9        JSR  RL      ;RIGHT/LEFT
4000:  CA15 38              SEC
4010:  CA16 A5 6D     L5    LDA  DIF4
4020:  CA18 65 6B           ADC  DIF2
4030:  CA1A 85 6D           STA  DIF4
4040:  CA1C A5 6E           LDA  DIF5
4050:  CA1E E9 00           SBC  #0      ;(X2-X1)-(Y2-Y1)
                                          TO (2-X1)
4060:  CA20 85 6E     L2    STA  DIF5
4070:  CA22 84 6F           STY  TEMP
4080:  CA24 20 E2 C8        JSR  PLT     ;DRAW POINT
4090:  CA27 E8              INX
4100:  CA28 D0 05           BNE  L6
4110:  CA2A E6 70           INC  CTR
4120:  CA2C D0 01           BNE  L6      ;DECREMENT COUNTER
4130:  CA2E 60              RTS
4140:  CA2F A5 6C     L6    LDA  DIF3
4150:  CA31 B0 DE           BCS  L1
```

```
4160:  CA33 20 77 C9          JSR  UD       ;UP/DOWN
4170:  CA36 18                CLC
4180:  CA37 A5 6D             LDA  DIF4
4190:  CA39 65 69             ADC  DIF0
4200:  CA3B 85 6D             STA  DIF4
4210:  CA3D A5 6E             LDA  DIF5
4220:  CA3F 65 6A             ADC  DIF1
4230:  CA41 50 DD             BVC  L2       ;ABSOLUTE
4240:                 ;
4250:                 ;
4260:                 ;LOAD GRAPHICS
4270:                 ;***************
4280:                 ;
4290:  CA43 20 FD AE GLOAD    JSR  CHKCOM   ;COMMA?
4300:  CA46 20 D4 E1          JSR  $E1D4    ;GET PARAMETER
4310:  CA49 A0 20             LDY  #>GRAPH
4320:  CA4B A2 00             LDX  #<GRAPH ;START ADDRESS
4330:  CA4D A9 00             LDA  #$00     ;LOAD FLAG
4340:  CA4F 4C D5 FF          JMP  $FFD5    ;LOAD
4350:                 ;
4360:                 ;
4370:                 ;SAVE GRAPHICS
4380:                 ;***************
4390:                 ;
4400:  CA52 20 FD AE GSAVE    JSR  CHKCOM   ;COMMA?
4410:  CA55 20 D4 E1          JSR  $E1D4
4420:                 ;END ADDRESS
4430:  CA58 A2 3F             LDX  #>GRAPH+8000
4440:  CA5A A0 40             LDY  #<GRAPH+8000
4450:  CA5C A9 00             LDA  #<GRAPH
4460:  CA5E 85 FD             STA  $FD
4470:  CA60 A9 20             LDA  #>GRAPH
4480:  CA62 85 FE             STA  $FE      ;START ADDRESS
```

```
4490:  CA64 A9 FD              LDA  #$FD     ;POINTER
4500:  CA66 4C D8 FF           JMP  $FFD8    ;SAVE
4510:                ;
4520:                ;
4530:                ;HARDCOPY
4540:                ;********
4550:                ;
4560:  CA69 20 F1 B7 HARDC     JSR  CHKGET   ;GET LOGICAL FILE#
4570:  CA6C 86 67              STX  $67
4580:  CA6E 20 0F F3           JSR  $F30F    ;FIND LOGICAL
                                              FILE#
4590:  CA71 20 1F F3           JSR  $F31F    ;SET FILE
                                              PARAMETERS
4600:  CA74 A6 67              LDX  $67
4610:  CA76 20 C9 FF           JSR  $FFC9    ;OPEN CHANNEL
4620:  CA79 A9 FF              LDA  #$FF
4630:  CA7B 85 61              STA  $61      ;MASK
4640:  CA7D A9 07              LDA  #7
4650:  CA7F 85 FD              STA  USE      ;SIZE
4660:  CA81 A9 1C              LDA  #28
4670:  CA83 85 97              STA  FLG      ;LINE COUNTER
4680:  CA85 A9 00              LDA  #0
4690:  CA87 8D 20 CB           STA  TMP      ;YC MARKER
4700:  CA8A A9 28     HA1      LDA  #40
4710:  CA8C 8D 21 CB           STA  FLG2     ;COUNTER
4720:  CA8F A2 04              LDX  #4
4730:  CA91 BD 2A CB HA1.      LDA  HATAB,X  ;CENTERED
4740:  CA94 20 D2 FF           JSR  $FFD2    ;OUTPUT
4750:  CA97 CA                 DEX
4760:  CA98 10 F7              BPL  HA1.
4770:  CA9A A9 00              LDA  #0
4780:  CA9C 85 63              STA  $63
4790:  CA9E 85 64              STA  $64      ;XC=0
```

```
4800:  CAA0 AD 20 CB HA2      LDA  TMP
4810:  CAA3 85 65             STA  $65      ;YC
4820:  CAA5 A9 00             LDA  #0
4830:  CAA7 85 FE             STA  USE+1    ;BUFFER POINTER
4840:  CAA9 A5 63    HA3      LDA  $63      ;XC-L
4850:  CAAB A6 64             LDX  $64      ;XC-H
4860:  CAAD A4 65             LDY  $65      ;YC
4870:  CAAF 20 94 C8          JSR  HPOSN    ;CALCULATE
                                             POSITION
4880:  CAB2 A0 00             LDY  #0
4890:  CAB4 B1 AC             LDA  (A),Y    ;GET BYTE
4900:  CAB6 A6 FE             LDX  USE+1    ;BUFFER POINTER
4910:  CAB8 9D 22 CB          STA  BUFF,X   ;IN BUFFER
4920:  CABB E6 65             INC  $65      ;YC
4930:  CABD E8                INX
4940:  CABE 86 FE             STX  USE+1
4950:  CAC0 E4 FD             CPX  USE
4960:  CAC2 D0 E5             BNE  HA3
4970:  CAC4 A9 00             LDA  #0
4980:  CAC6 A0 07             LDY  #7
4990:  CAC8 A6 FD    HA4      LDX  USE
5000:  CACA 1E 22 CB HA5      ASL  BUFF,X   ;GET A BIT
5010:  CACD 2A                ROL  A        ;AND SAVE IN ACC
5020:  CACE CA                DEX           ;INCREMENT BUFFER
                                             POINTER
5030:  CACF 10 F9             BPL  HA5
5040:  CAD1 25 61             AND  $61      ;FOR LAST LINE=$0F
5050:  CAD3 09 80             ORA  #$80     ;HIGH BYTE
5060:  CAD5 20 D2 FF          JSR  $FFD2    ;SEND
5070:  CAD8 88                DEY           ;SEND 8 BYTES
5080:  CAD9 10 ED             BPL  HA4
5090:  CADB A5 63             LDA  $63      ;XC-L
5100:  CADD 18                CLC
```

```
5110:  CADE 69 08            ADC  #8
5120:  CAE0 85 63            STA  $63      ;XC+8
5130:  CAE2 90 02            BCC  HA6
5140:  CAE4 E6 64            INC  $64      ;XC-H
5150:  CAE6 CE 21 CB HA6     DEC  FLG2
5160:  CAE9 D0 B5            BNE  HA2
5170:  CAEB A9 0D            LDA  #$0D     ;CARRIAGE RETURN
5180:  CAED 20 D2 FF         JSR  $FFD2    ;SEND
5190:  CAF0 AD 20 CB         LDA  TMP
5200:  CAF3 18               CLC
5210:  CAF4 69 07            ADC  #7
5220:  CAF6 8D 20 CB         STA  TMP      ;YC+7
5230:  CAF9 C6 97            DEC  FLG
5240:  CAFB F0 03            BEQ  HA8.
5250:  CAFD 4C 8A CA HA8     JMP  HA1
5260:  CB00 A9 04    HA8.    LDA  #4
5270:  CB02 C5 FD            CMP  USE
5280:  CB04 F0 0C            BEQ  HA7
5290:  CB06 85 FD            STA  USE      ;LAST LINE
5300:  CB08 A9 01            LDA  #1
5310:  CB0A 85 97            STA  FLG      ;FLAG
5320:  CB0C A9 0F            LDA  #$0F
5330:  CB0E 85 61            STA  $61      ;MASK
5340:  CB10 D0 EB            BNE  HA8
5350:  CB12 A9 0F    HA7     LDA  #15      ;NORMAL MODE
5360:  CB14 20 D2 FF         JSR  $FFD2
5370:  CB17 4C CC FF         JMP  $FFCC    ;CLOSE CHANNEL
5380:                ;
5390:                ;
5400:                ;INTERNAL STORAGE
5410:                ;****************
5420:                ;
5430:  CB1A 30       XCL     .BYTE48       ;XC-LOW-TEMP
```

```
                                          STORAGE
5440:  CB1B 30        XCH       .BYTE48   ;XC-HIGH-TEMP
                                          STORAGE
5450:  CB1C 30        YC        .BYTE48   ;YC-TEMP STORAGE
5460:  CB1D 20        COLOR     .BYTE32   ;COLOR
5470:  CB1E 3A        STORE1    .BYTE58   ;V+17 REG. TEMP
                                          STORAGE
5480:  CB1F 20        STORE2    .BYTE32   ;V+24 REG. TEMP
                                          STORAGE
5490:  CB20 82        TMP       .BYTE130  ;TEMP STORAGE
5500:  CB21 20        FLG2      .BYTE32   ;TEMP STORAGE
5510:  CB22 58 20 3A BUFF       .BYTE88, 32, 58, 32, 143, 32,
                                     87, 65  ;BUFFER FOR HARD COPY
5520:                 ;
5530:                 ;
5540:                 ;TABLES
5550:                 ;*******
5560:                 ;
5570:                 ;PRINTER CHARACTERS
5580:                 ;(BACKWARDS)
5590:                 ;
5600:                 ;CENTERED/GRAPHICS ON
5610:                 ;
5620:  CB2A 50 00 10 HATAB      .BYTE80, 0, 16, 27, 8
5630:                 ;
5640:                 ;
5650:                 ;MULTIPLICATION TABLE
5660:                 ;(N*320 FOR N=0 TO N=24)
5670:                 ;
5680:                 ;HIGH BYTES
5690:                 ;
5700:  CB2F 00 01 02 MULH       .BYTE 0, 1, 2, 3, 5, 6, 7, 8,
                                    10, 11, 12, 13, 15, 16
```

```
5710:  CB3D 11 12 14          .BYTE17, 18, 20, 21, 22, 23,
                                   25, 26, 27, 28, 30, 31
5720:                 ;
5730:                 ;LOW BYTES
5740:                 ;
5750:  CB49 00 40 80 MULL     .BYTE$00, $40, $80, $C0
5760:                 ;
5770:                 ;
5780:                 ;MASK TABLE
5790:                 ;
5800:  CB4D 01 02 04 MSCTAB   .BYTE%00000001, %00000010,
                                   %00000100, %00001000
5810:  CB51 10 20 40          .BYTE%00010000, %00100000,
                                   %01000000, %10000000
5820:  CB55                   .SYM
]C800-CB55
NO ERRORS
```

```
100 FOR I = 51200 TO 52054
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA  76, 36,200, 76, 65,200, 76, 18,201, 76, 49,201
130 DATA  76, 42,201, 76, 88,200, 76, 85,200, 76,107,200
140 DATA  76,104,200, 76, 67,202, 76, 82,202, 76,105,202
150 DATA 234,173, 17,208,141, 30,203,173, 24,208,141, 31
160 DATA 203,169, 59,141, 17,208,169, 24,141, 24,208,169
170 DATA  96,141, 36,200, 96,173, 30,203,141, 17,208,173
180 DATA  31,203,141, 24,208,169,234,141, 36,200, 76, 68
190 DATA 229,162,  0, 44,162,128,134,151, 32,253,174, 32
200 DATA 121,200, 32,148,200, 76,226,200,162,  0, 44,162
210 DATA 128, 32, 90,200, 32,253,174, 32,121,200, 76,187
220 DATA 201, 32,235,183,138,168,166, 21,192,200,176, 13
230 DATA 165, 20,224,  1,144,  6,208,  5,201, 64,176,  1
240 DATA  96, 76, 72,178,140, 28,203,141, 26,203,142, 27
250 DATA 203,133, 20,134, 21,152, 74, 74, 74,170,189, 47
260 DATA 203,133,173,138, 41,  3,170,189, 73,203,133,172
270 DATA 152, 41,  7, 24,101,172,133,172,165, 20, 41,248
280 DATA 133, 99,169, 32,  5,173,133,173, 24,165,172,101
290 DATA  99,133,172,165,173,101, 21,133,173,165, 20, 41
300 DATA   7, 73,  7,170,189, 77,203,133,171, 96,160,  0
310 DATA   8,165,171, 36,151, 48,  5, 73,255, 49,172, 44
320 DATA  17,172,145,172,165,172,133,253,165,173, 74,102
330 DATA 253, 74,102,253, 74,102,253, 41,  3,  9,  4,133
340 DATA 254,173, 29,203,145,253, 40,164,111, 96,169, 32
350 DATA 133,254,160,  0,132,253,162, 32,152,145,253,200
360 DATA 208,251,230,254,202,208,246, 76, 55,201, 32,241
370 DATA 183,142, 29,203, 96, 32,241,183,142, 29,203,162
380 DATA   3,169,  4,133,254,160,  0,132,253,132,151,173
390 DATA  29,203,145,253,200,196,151,208,249,230,254,202
400 DATA 240,  3, 16,242, 96,162,232,134,151,208,235,165
410 DATA 172, 41,  7,201,  7,240,  5, 56,169,  0,176,  4
420 DATA 169, 56,230,173,101,172,133,172,169,  0,101,173
```

```
430 DATA 133,173, 96, 48,226,165,172, 41,  7,240,  5, 24
440 DATA 169,255,144,  4,169,199,198,173,101,172,133,172
450 DATA 165,173,233,  0,133,173, 96, 70,171,144, 14,102
460 DATA 171,165,172,200, 24,105,  8,133,172,144,  2,230
470 DATA 173, 96, 16,235,  6,171,144, 14, 38,171,165,172
480 DATA 136, 56,233,  8,133,172,176,  2,198,173, 96, 72
490 DATA 173, 27,203, 74,173, 26,203,106, 74, 74,133,111
500 DATA 104, 72, 56,237, 26,203, 72,138,237, 27,203,133
510 DATA 108,176, 10,104, 73,255,105,  1, 72,169,  0,229
520 DATA 108,133,106,133,110,104,133,105,133,109,104,141
530 DATA  26,203,142, 27,203,152, 24,237, 28,203,144,  4
540 DATA  73,255,105,254,133,107,140, 28,203,102,108, 56
550 DATA 229,105,170,169,255,229,106,133,112,164,111,176
560 DATA   5, 10, 32,166,201, 56,165,109,101,107,133,109
570 DATA 165,110,233,  0,133,110,132,111, 32,226,200,232
580 DATA 208,  5,230,112,208,  1, 96,165,108,176,222, 32
590 DATA 119,201, 24,165,109,101,105,133,109,165,110,101
600 DATA 106, 80,221, 32,253,174, 32,212,225,160, 32,162
610 DATA   0,169,  0, 76,213,255, 32,253,174, 32,212,225
620 DATA 162, 63,160, 64,169,  0,133,253,169, 32,133,254
630 DATA 169,253, 76,216,255, 32,241,183,134,103, 32, 15
640 DATA 243, 32, 31,243,166,103, 32,201,255,169,255,133
650 DATA  97,169,  7,133,253,169, 28,133,151,169,  0,141
660 DATA  32,203,169, 40,141, 33,203,162,  4,189, 42,203
670 DATA  32,210,255,202, 16,247,169,  0,133, 99,133,100
680 DATA 173, 32,203,133,101,169,  0,133,254,165, 99,166
690 DATA 100,164,101, 32,148,200,160,  0,177,172,166,254
700 DATA 157, 34,203,230,101,232,134,254,228,253,208,229
710 DATA 169,  0,160,  7,166,253, 30, 34,203, 42,202, 16
720 DATA 249, 37, 97,  9,128, 32,210,255,136, 16,237,165
730 DATA  99, 24,105,  8,133, 99,144,  2,230,100,206, 33
740 DATA 203,208,181,169, 13, 32,210,255,173, 32,203, 24
750 DATA 105,  7,141, 32,203,198,151,240,  3, 76,138,202
```

```
760 DATA 169,  4,197,253,240, 12,133,253,169,  1,133,151
770 DATA 169, 15,133, 97,208,235,169, 15, 32,210,255, 76
780 DATA 204,255, 48, 48, 48, 32, 58, 32,130, 32, 88, 32
790 DATA  58, 32,143, 32, 87, 65, 80,  0, 16, 27,  8,  0
800 DATA   1,  2,  3,  5,  6,  7,  8, 10, 11, 12, 13, 15
810 DATA  16, 17, 18, 20, 21, 22, 23, 25, 26, 27, 28, 30
820 DATA  31,  0, 64,128,192,  1,  2,  4,  8, 16, 32, 64
830 DATA 128, 76, 79
840 IF S <> 104883 THEN PRINT "ERROR IN DATA !!" : END
850 PRINT "OK"
```

How do we use this graphics package?

The 12 new commands are used with SYS commands. Some SYS commands require parameters. This seem unusual at first because the SYS command does not normally allow additional parameters. The best way to use these routines is illustrated in the sample programs: Define 12 variables at the beginning of the program which contain the addresses of the individual routines. During your program, call the commands using using the variables and the parameters. This table lists the jump addresses and the syntax of the various commands:

```
SYS 51200              - turn graphics on
SYS 51203              - turn graphics off
SYS 51206              - clear graphics screen
SYS 51209,PC*16+BC     - set colors
SYS 51212,PC*16+BC     - change plot color
SYS 51215,X,Y          - set point
SYS 51218,X,Y          - erase point
SYS 51221,X1,Y1,X2,Y2 - draw line
```

```
SYS 51224,X1,Y1,X2,Y2 - erase line
SYS 51227,"name",DA   - load graphics screen
SYS 51230,"name",DA   - save graphics screen
SYS 51233,LF          - hardcopy S. GP-100VC
```

Explanations:

```
PC      - color of a graphics point (0-15)
BC      - background color from 0 to 15
X       - x-coordinate of a point (0-319)
Y       - y-coordinate of a point (0-199)
X1/2    - x-coordinate of start/end point
Y1/2    - y-coordinate of start/end point
"name"  - filename (may also be string variable)
DA      - device address (1 or 8)
LF      - logical filenumber for OPEN LF,DA
```

Before we begin our demonstration program, we want to review a few things.

The first three commands are straight forward. Notice that the screen is not cleared when the graphic mode is turned on, but is when it is turned off.

When setting the color, the color is the same for all of the graphic points and for the entire background. If you choose the color green for the background (color code: 5) and purple (color code: 4) for the point color, you would set PC=5 and BC=4. The corresponding color byte of the video RAM is not set until the point is plotted. The color can first be set by the appropriate command and then changed with the second command so that all figures drawn after this time are drawn in the new color.

Two commands are available to plot and erase points. When these commands are performed, the actual color is set

in the color RAM.

If you want to draw or erase lines between two points (X1,Y1) and (X2,Y2), use the commands listed next. What was said before about the color setting also applies here.

The two commands for saving and restoring graphics screens are used exactly as the BASIC commands for saving and loading BASIC programs--by giving the filename and the device address (DA=1 for cassette, DA=8 for diskette).

Before you start a hardcopy on the VIC 1525/MPS 801, you must OPEN the printer channel with OPEN 1,4. SYS HC,1 will then start the graphics dump (the 1 in the command indicates logical filenumber 1 which we chose in our OPEN command) and a CLOSE 1 must follow the SYS command (see demonstration program).

The demonstration program should help to clear up any remaining uncertainties:

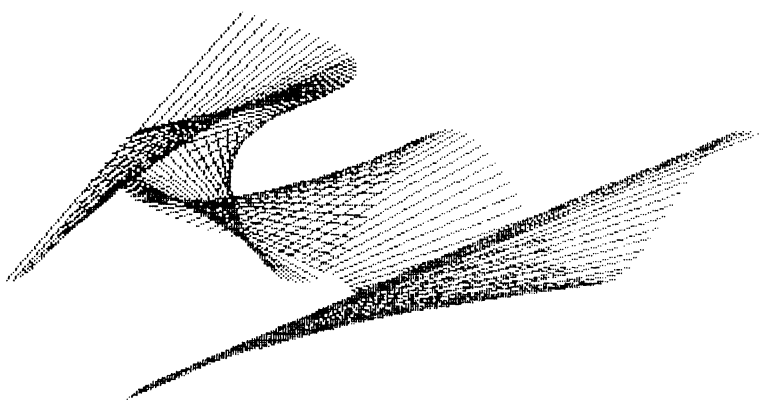
```
10 REM ************************
20 REM **                      **
30 REM **  GRAPHICS PACKAGE    **
40 REM **                      **
50 REM **     - D E M O -      **
60 REM **                      **
70 REM ************************
80 REM
90 REM MACHINE LANGUAGE ROUTINES:
100 IN=51200 : OF=51203 :REM INIT      /GRAPHICS OFF
110 GC=51206 : SC=51209 :REM GCLEAR    /SET COLOR
120 PC=51212 : PL=51215 :REM PCOLOR    /PLOT
130 UP=51218 : SL=51221 :REM UNPLOT    /SET LINE
140 CL=51224 : GL=51227 :REM CLR LINE /GLOAD
150 GS=51230 : HC=51233 :REM GSAVE     /HARDCOPY
200 REM
```

```
210 REM EXAMPLES:
220 REM *********
230 REM
300 SYS IN : REM GRAPHICS ON
310 SYS GC : REM CLEAR GRAPHICS SCREEN
320 SYS SC,1*16+2 : REM SET COLOR
330 SYS PC,7*16+2 : REM SET PLOT COLOR
340 REM
350 REM FIGURE 1:
360 REM *********
370 REM
380 FOR X=1 TO 319 STEP 4
390 SYS SL,X,50,70,X/1.6 : REM LINES
400 NEXT X
410 REM
420 FOR X=1 TO 5000 : NEXT X : REM DELAY LOOP
430 GOSUB 2000 : SYS GC : REM CLEAR GRAPHICS SCREEN
440 REM
450 REM FIGURE 2:
460 REM *********
470 REM
480 FOR X=1 TO 319 STEP 3
490 SYS SL,X,40,50*SIN(X/30)+100,X/1.6
500 NEXT X
510 REM
520 FOR X=1  TO 5000 : NEXT X : REM DELAY LOOP
530 GOSUB 2000 : SYS GC : REM CLEAR GRAPHICS SCREEN
540 REM
550 REM FIGURE 3:
560 REM *********
570 REM
580 FOR X=1 TO 319 STEP 2
590 SYS SL,X,40*COS(X/20)+100,50*SIN(X/30)+100,X/1.6
```

```
600 NEXT X
610 REM
620 FOR X=1 TO 5000 : NEXT X : REM DELAY LOOP
630 GOSUB 2000 : SYS GC : REM ERASE GRAPHICS SCREEN
640 REM
1000 WAIT198,255 : REM WAIT FOR KEY
1010 SYS OF : END
2000 SYS OF : REM GRAPHICS OFF
2010 PRINT "SELECT ONE:":PRINT
2020 PRINT "(1) - LOAD GRAPHICS"
2030 PRINT "(2) - SAVE GRAPHICS"
2040 PRINT "(3) - HARDCOPY"
2050 PRINT "(4) - CONTINUE" : PRINT
2060 POKE 198,0 : REM CLEAR KEYS
2070 WAIT 198,255 : REM WAIT FOR KEY
2080 GET A$
2090 ON VAL(A$) GOTO 2200,2300,2400,2500
2100 GOTO 2000
2200 REM
2210 REM LOAD GRAPHICS:
2220 REM **************
2230 REM
2240 INPUT "FILENAME,DA";FI$,DA
2250 SYS GL,FI$,DA : REM LOAD
2260 SYS IN : REM GRAPHICS ON
2270 SYS SC,16*3+9
2280 FOR X=1 TO 5000 : NEXT X : REM WAIT
2290 GOTO 2000
2300 REM
2310 REM SAVE GRAPHICS:
2320 REM **************
2330 REM
2340 INPUT "FILENAME,DA";FI$,DA
```

```
2350 SYS GS,FI$,DA : REM SAVE
2360 GOTO 2000
2400 REM
2410 REM HARDCOPY:
2420 REM *********
2430 REM
2440 PRINT "READY PRINTER AND PRESS A KEY"
2450 POKE 198,0 : WAIT 198,255 : GET A$
2460 PRINT : PRINT "PLEASE WAIT!"
2470 OPEN 1,4 : REM OPEN PRINTER CHANNEL
2480 SYS HC,1 : REM HARDCOPY
2490 CLOSE 1 : REM CLOSE CHANNEL
2500 REM
2510 REM CONTINUE:
2520 REM *********
2530 REM
2540 SYS IN : SYS SC,16*2+7 : RETURN
```

We have given you a few applications examples in this demo program. You can now use these routines in your own programs. Have fun!



Chapter 5 : Applications

Now that we have that large chapter on the fundamentals of graphics programming behind us, it is time to move on to the diverse applications for our graphics expertise

What shall we do with our knowledge? How does one use lines and circles to represents facts and relationships?

In this chapter, numerous examples will be used to show how all of these techniques can be used to season and enrich your work. You can learn many new techniques to use in your programs.

Three major sections cover various uses for high-resolution graphics and using sprites for moving text, which you may find in commercial applications. The last section shows you the possibilities which are available for such things as games. Several useful routines are also presented.

5.1 Graphics applications

Let us get right to the point. All of the following example programs use the commands of the small graphics package presented in Chapter 4 to take advantage of their speed. If you have not made the effort and typed our utility program in, you can also use the BASIC subroutines which were presented earlier in the chapter. You need only place the parameters in the appropriate variables and call the given subroutine. Don't forget to set V and SA for the base addresses of the video controller and graphics storage with V=53248 and SA=8192.

If you have a graphics extension package you can replace the individual SYS commands with the corresponding commands from that package. The programs are deliberately written so that most changes are easy to make.

High-resolution graphics work well for commercial applications, but are also good for your personal applications (school, job, free time).

5.1.1 Graphing functions

One popular use of high-resolution graphics is for graphing various functions. Sine or cosine curves are most often used in sample programs (see example program in section 4.2.1.1). There are two major reasons for this: First, the pictures look quite impressive and second, the problem of exceeding the graphable range is easy to overcome. In order to clarify this point we must discuss a little more of the mathematics.

A function is a one-to-one correspondence of one set of values with another. Each element of the first set is paired

with exactly one element of the second set (although the reverse need not apply). The elements of the first set are called x in algebraic functions, the second set y. X and y are linked by a mathematical formula or equation (in the case of the lines in section 4.2.2.2, through the line equation):

y = f(x) read as: y equals f of x

If you wish to determine a function pair, you replace x with the desired value and calculate y. The association between x and y can be represented graphically in a coordinate system. The horizontal axis of the system represents all possible x values (abscissa) and the vertical axis represents all possible y values (ordinate) of the function. If we know the x,y pair we can draw a perpendicular from the x-value to the x axis and a corresponding line from the y-value to the y-axis. Our point lies at the point of intersection between these two lines. To display a picture which is as accurate as possible, we must calculate many such coordinate pairs and plot them within our coordinate system. Using a computer, this is done by varying the x-value and computing the corresponding y-values. You can easily vary the x-value by using a FOR/NEXT loop. The coordinate pair is used to plot the point on the screen. The same principle was used when drawing a line or circle in Chapter 4. The following program should render this clear:

```
70 REM MACHINE LANGUAGE ROUTINES:
100 IN=51200 : OF=51203 :REM INIT    /GRAPHICS OFF
110 GC=51206 : SC=51209 :REM GCLEAR  /SET COLOR
120 PC=51212 : PL=51215 :REM PCOLOR  /PLOT
130 UP=51218 : SL=51221 :REM UNPLOT  /SET LINE
```

```
140 CL=51224 : GL=51227 :REM CLR LINE/GLOAD
150 GS=51230 : HC=51233 :REM GSAVE   /HARDCOPY
160 REM
170 REM ********************
180 REM
200 SYS IN : SYS GC : SYS SC,16*1+6 : REM INITIALIZE
GRAPHICS
210 FOR X=0 TO 319
220 Y = X^2 : REM CALCULATE FUNCTION VALUE
230 IF Y>199 THEN WAIT 198,255 : SYS OF : END : REM RANGE
OVERFLOW
240 SYS PL,X,Y : REM SET POINT
250 NEXT X
```

As you see in line 230, the y-value is checked to see if still lies within the legal range of the screen, in order avoid an ILLEGAL QUANTITY ERROR.

This parabola function does not produce the expected results. Only half of the parabola appears on the screen. The function does not fill the screen because far too few points were calculated in order to achieve a good curve. This is because of the following:

The coordinate system of the '64 is limited by predetermined dimensions. We cannot, for example, use negative numbers or numbers greater then 319 for x or 199 for y. Despite this, there are functions which have interesting characteristics in these ranges. Simple angle-functions such as:

```
y = sin(x)    or    y = cos(x)
```

return only values between -1 and 1 for y and have an oscillation period of 2*pi, or approximately 6, which is not

directly needed for our application.
There are methods which can help us with this dilemma:

- scaling
- offsets
- distortion (independent scaling)

a) Scaling:

Scaling is adjusting the magnitude of the function values to fit within the screen coordinates. This is accomplished simply by multiplying the x and y values by the same factor. This can be done in two ways:

1) The normal value for x is used in the unaltered function and the x and y values are then multiplied by the scaling factor. Whether the graph will be enlarged or reduced depends on this factor (here called f):

Factor	Result
0<f<1	reduced
f>1	enlarged

If you choose an f<0, the function will appear reversed from left to right and from top to bottom. A BASIC routine for this would be something like:

```
110 F=30 : REM ENLARGEMENT FACTOR
120 FOR X=0 TO 10
130 Y = SIN(X) : REM CALCULATE VALUE
140 Y = F*Y : X = F*X : REM SCALE
150 IF Y>199 OR X>319 THEN WAIT 198,255 : SYS OF : END
160 SYS PL,X,Y : REM DRAW POINT
170 NEXT X
```

This program requires the graphics package and appropriate variable definitions in order to run. In addition, it still yields negative values for y.

2) You can build the scaling directly into the function. You directly create a scaled value for x (through the FOR/NEXT loop) and then reset this value within the function by dividing by f. The y-scaling then follows directly from the formula. This method could look like this:

```
110 F=30 : REM ENLARGEMENT FACTOR
120 FOR X=0*F TO 10*F
130 Y = F * SIN(X/F) : REM SCALE VALUE
140 IF Y>199 THEN WAIT 198,255 : SYS OF : END
150 SYS PL,X,Y : REM DRAW POINT
160 NEXT X
```

This form has three advantages in particular: First, it is shorter and faster; second, illegal values of x can be checked from the start because we select the range of the loop directly; and third, we calculate 301 values instead of the above 11, which naturally increases the accuracy of the curve's appearance. In spite of all this, we still have the same problem: y always becomes negative.

b) Offsetting:

Offsetting is shifting an entire graph within the coordinate system. We can move the sine curve out of the negative range and into the positive so that y assumes only positive values. Or we can shift the previous para-

bola to the right to that the left branch becomes visible.

This is done by adding certain values to the two coordinates. These values need not be the same, as with scaling, in order not to affect the proportions of the graph. If we add a value b to the y-coordinate, we shift the graph up if b is positive or down if it is negative (and thereby execute an subtraction). If this is done to the x-coordinate (a is added) a shift right (a>0) or left (a<0) will result. Here we have the same two possibilities as were given for scaling:

1) The values of the two coordinates are calculated and the two addends are added. The lines 130-140 in the program under a)1) would be replaced (without scaling):

```
130 Y = SIN(X)
140 Y = Y+B : X = X+A
```

2) Here too we pack the whole thing into one formula and get (lines 120-130 of the program under a)2)):

```
120 FOR X=1 TO 10
130 Y = SIN(X-A) + B
```

You see that A is subtracted from x instead of added.
Our parabola program could be rewritten by replacing line 220 with (for example):

```
220 Y = (X-13)^2 + 5
```

Now we get an entirely different result. Here a=13 and b=5. If you choose larger values for a, for example,

it will not be drawn because y is then too large. We will explain later how to solve this problem.

c) Distortion:

Distortion is a general form of scaling. Here x and y need not be multiplied by the same factor (we now call the factors simply f1 and f2). This changes the proportions of the curve, that is, it either becomes shorter (for 0<f1/2<1) or longer (f1/2>1) in the x direction (for f1) or y direction (for f2). Now most of the curves can be drawn. Replace line 220 of the parabola function program with:

```
220 Y = 0.01 * ((X-139)/1)^2 + 5
```

Here scaling and distortion are used at the same time, leading to this interesting result. The individual values which can be changed are: f1=1, f2=0.01, a=139, b=5 (instead of multiplying by 0.01, we could also divide by 100 for the sake of simplicity).

There are a few things which should be cleared up. Our parabola always stands on its head and changing the various parameters always leads to ILLEGAL QUANTITY ERRORS.

This phenomenon occurs because our coordinate system itself is standing on its head. Our zero point lies not in the lower corner but in the upper left corner. The y values should grow smaller as we go down, not larger. This peculiarity can be solved by using a simple trick. We simply rotate the curve by reversing the sign of all of the y values. Because we will then often get negative values for y, we shift the curve a certain distance down (actually up), by adding 195, for example.

```
220 Y = - 0.01 * ((X-139)/1) + 139
```

Now to the error messages which have in the meantime become even more frequent. The cause lies simply in our incomplete range checking. For one, we never test to see if y is negative. Second, we end our program whenever it encounters an overflow. If you change the previous parabola program at line 210 as follows, you will be free to make any changes you like to a, b, f1, or f2 for every function:

```
210 F1 = 1 : F2 = 0.1 : A = 160 : B = 100
220 FOR X=0 TO 319
230 Y = - F2 * ((X-A)/F1)^2 + B
240 IF Y>199 OR Y<0 THEN NEXT X : GOTO 270
250 SYS PL,X,Y : REM SET POINT
260 NEXT X
270 WAIT 198,255 : SYS OF : END
```

The computer can run through a lot of values before it draws anything. In this program the origin of the shifted coordinate system is placed at 160,100, the middle of the screen. We could also draw the axes to make this clearer.

Another alternative is to define the function as a BASIC DEF FN. With this command a function is defined which can then be called from anywhere in the program without having to write it over again. The following changes must be made:

```
215 DEF FN F(X) = - F2 * ((X-A)/F1)^2 + B
230 Y = FN F(X)
```

The value for the various variables is always placed in the formula. You can use the function anywhere without having to access it within the loop or to search for it.

The following program goes somewhat farther. It draws a desired function for you. You can enter your function using the INPUT statement and the program develops the function assignment by POKEing the appropriate bytes into a free BASIC area. You do not have to understand this relatively complicated routine. The function is entered without scaling or offset parameters. These will be added later in the main loop.

Be absolutely sure to type in the first part of the program up to the DEF FN line (line 350) exactly as shown, with the same number of spaces given. If you do not do so, the program will in all likelihood crash. Before you RUN it, you should first save the program!

```
100 REM ************************
110 REM **                      **
120 REM **  FUNCTION PLOTTER    **
130 REM **                      **
140 REM ************************
150 REM
160 REM
170 REM DEF FN - ROUTINE:
180 REM ******************
190 REM
200 RESTORE:DIM A$(25):PRINT CHR$(147)
210 PRINT "ENTER THE FUNCTION TO GRAPH:":PRINT:PRINT:PRINT
220 GOSUB 500 : REM INPUT ROUTINE
230 AD=2684-19:REM POKE START
```

```
240 FOR X=1 TO 25:READ A$(X):NEXT X:REM READ DATA
250 FORX=1TOLEN(A$):AD=AD+1
260 Q=X:FORY=1TO25:B$=A$(Y):FORZ=1TOLEN(B$):C$=MID$(A$,X,1)
270 IFC$=" "THENAD=AD-1:NEXTX:GOTO310
280 IFC$=MID$(B$,Z,1)THENX=X+1:NEXTZ:POKEAD,Y+169:X=X-1:
    NEXTX:GOTO310
290 X=Q:NEXTY
300 POKEAD,ASC(C$):NEXTX
310 POKEAD+1,58:POKEAD+2,143
340 REM PLACE HOLDERS:
350 DEFFNF(X)=000000000000000000000000000000000000000000000000
    0000000000000000000
400 REM 25 FUNCTIONS:
410 DATA +,-,*,/,^,AND,OR,<,=,>,SGN,INT,ABS,USR,FRE,POS,SQR,
    RND,LOG,EXP
420 DATA COS,SIN,TAN,ATN,PEEK
430 GOTO 600 : REM CONTINUE
500 INPUT "F(X)=";A$ : RETURN : REM INPUT ROUTINE (FOR
    CONSTRUCTION)
540 REM
550 REM      *****************
560 REM **** GRAPHICS ROUTINES ****
570 REM      *****************
580 REM
600 IN=51200 : OF=51203 :REM INIT     /GRAPHICS OFF
610 GC=51206 : SC=51209 :REM GCLEAR   /SET COLOR
620 PC=51212 : PL=51215 :REM PCOLOR   /PLOT
630 UP=51218 : SL=51221 :REM UNPLOT   /SET LINE
640 CL=51224 : GL=51227 :REM CLR LINE /GLOAD
650 GS=51230 : HC=51233 :REM GSAVE    /HARDCOPY
655 POKE 2,0 : REM GRAPHICS FLAG
660 REM
670 REM       ************
```

```
680 REM  **** MAIN PROGRAM ****
690 REM       ************
700 REM
710 PRINT : PRINT "ENTER THE FOLLOWING PARAMETERS:" :PRINT
720 INPUT "F1 (XSCALE)  ";F1
730 INPUT "F2 (YSCALE)  ";F2
740 INPUT "A  (X OFFSET)";A
750 INPUT "B  (Y OFFSET)";B
1000 SYS IN : SYS SC,16*1+5 : REM INITIALIZE
1005 IF PEEK(2)=0 THEN SYS GC : REM CLEAR GRAPHICS SCREEN
1007 POKE 2,0 : REM ERASE-FLAG
1010 REM
1020 REM DRAW AXES:
1030 REM **********
1040 IF A>=0 AND A<320 THEN SYS SL,A,0,A,199 : REM X-AXIS
1050 IF B>=0 AND B<200 THEN SYS SL,0,B,319,B : REM Y-AXIS
1060 REM
1070 REM DRAW ROUTINE:
1080 REM *************
1210 FL% = 1 : REM OUTSIDE-FLAG
1220 FOR X=1 TO 319
1230 Y=-F2*FNF((X-A)/F1)+B
1240 IF Y<0 OR Y>199 THEN FL%=1:NEXT X:GOTO1280
1250 IF FL%=0 THEN SYS SL,X1,Y1,X,Y
1260 FL%=0
1270 X1=X:Y1=Y:NEXT X : REM SAVE LAST COORD.
1280 POKE 198,0 : WAIT 198,255 : GET A$ : SYS OF : REM
     GRAPHICS OFF
1290 REM ***** MENU: *****
1300 PRINT "SELECT ONE:" : PRINT : PRINT
1310 PRINT "(1) OTHER PARAMETERS"
1320 PRINT "(2) ANOTHER FUNCTION"
1330 PRINT "(3) DON'T CLEAR GRAPHICS"
```

```
1340 PRINT "(4) SAVE GRAPHICS"
1350 PRINT "(5) LOAD GRAPHICS"
1360 PRINT "(6) HARDCOPY"
1390 PRINT "(7) EXIT"
1450 WAIT 198,1 : GET A$
1460 ON VAL(A$) GOTO 700,1480,1490,1510,1600,1650,1500
1470 GOTO 1450 : REM INPUT ERROR
1480 RUN : REM ANOTHER FUNCTION
1490 POKE 2,1 : PRINT "OK" : PRINT : GOTO 1450 : REM SET FLAG
1500 END : REM EXIT
1510 REM
1520 REM SAVE GRAPHICS:
1530 REM
1540 INPUT "FILENAME,DA";FI$,DA
1550 SYS GS,FI$,DA : REM SAVE
1560 PRINT "OK" : GOT 1450
1570 REM
1580 REM LOAD GRAPHICS:
1590 REM
1600 INPUT "FILENAME,DA";FI$,DA
1610 SYS GL,FI$,DA : REM LOAD
1620 SYS IN : SYS SC,16*1,5 : GOTO 1280
1630 REM
1640 REM HARDCOPY:
1645 REM
1650 PRINT "READY PRINTER AND PRESS A KEY"
1660 OPEN 1,4 : SYS HC,1 : CLOSE 1 : REM HARDCOPY
1670 PRINT "OK" : GOTO 1450
```

Note that this program was also written for the graphics package in section 4.7 and must be rewritten as described earlier in the chapter if it to be used with other graphics routines.

As mentioned before, you are first required to enter a function. The INPUT used here is in a subroutine in line 500 and can be replaced by a different input loop, such as GET. The displacement to line 500 is necessary because no changes may be made under any circumstances to lines up to and including line 350. The function is then contained in A$ and is passed to line 350 by the translation routine. This forms a DEF FN line by POKEing the necessary bytes directly into the BASIC storage and so changes line 350. You can take a look at this line after you have entered the first function. The address at which this line begins, or at which the program begins to write the function definition, is in line 230 and can be changed by those who understand the routine and want to make such changes.

Your function can contain any of the 25 BASIC functions found in lines 410-420. The syntax is the same as in BASIC.

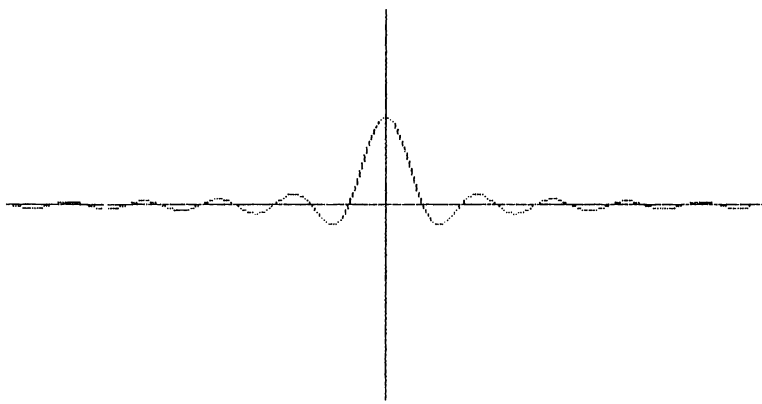
After the obligatory jump address definitions, you are required to enter the two scaling factors f1/f2 (independent factors--distortion or "straight" scaling is possible) and the offsets a and b (lines 710-750). Here you can use what you just learned. F1 and f2 are greater than 1 for most functions (the range 10-70 is recommended for angle functions, for example). As you know, f1 determines the size (stretching) of the graph from left to right. F2, on the other hand, stretches the curve from top to bottom. The recommended initial values for a and b are a=160 and b=100, which place the origin of the coordinate system in the center of the screen. If, for example, you wish to display only the range in which x is positive, then you choose a=0, and so on.

After switching to graphics mode and clearing the screen (depending on the erase flag) the axes of the coordinate system are drawn at line 1040 according to the select-

ed offsets. If you want, you can also draw tic markers along the axes.

The plotting of the function begins in line 1250. Here is the routine which we developed above. Only one small thing is changed. For clarity, the the individual points are connected to form lines. This gives a better-looking picture.

Once the curve is drawn, you can return to the main menu by pressing any key. Here you can choose from various options. You can select a completely new function, change only the parameters, retain the old screen while the new function is drawn (good for making comparisons), save/load the graphics screen, create a hardcopy, or simply end the program.



Be sure that the syntax of your function is correct or a SYNTAX ERROR will occur. You should also be sure that undefined values do not creep in. These include: too high a

number in general, the value zero as a denominator or in a logarithmic function, and negative values as angles or arguments to log functions.

Negative values can be avoided through the use of the absolute function, for example:

```
SQR(ABS(X))
```

The value zero in the denominator can be avoided by replacing X with the term (X-(X=0)):

```
instead of 1/X    : 1/(X-(X=0))       or
instead of LOG(X): LOG(ABS(X-(X=0)))
```

If X is equal to zero, the expression X=0 in Commodore BASIC evaluates to -1, otherwise it remains 0. Greater or less than signs (>,<) can be used similarly.

If an error does occur, you can start the program with RUN 350, whereby the function is retained.

5.1.2 3-dimensional graphics

Few computer advertisements and displays from other areas as well do without at least some kind of three-dimensional graphics pictures in order to tout their products. Such drawings provide enormous aesthetics, causing the viewer to attribute great computation power to the product. More and more graphics designers are seeking to create complex 3-dimensional pictures with the help of the computer and its capabilities.

Even personal users find it great fun to become involved with 3D. But most readers lack the "know how" for draw-

ing in 3D. Therefore some background is presented here which relies on information presented in section 5.1.

5.1.2.1 Parallel projection

There are two types of projections:

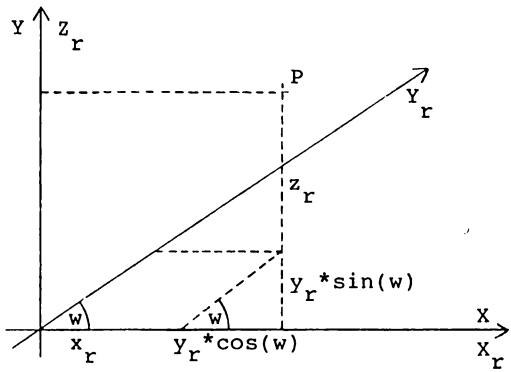
- parallel projections
- central projections

In the following, we will concern ourselves with the first type of projection. Later we will provide the necessary knowledge for the second.

First we must expand our 2-dimensional coordinate system. We will call the axes x, y, and z. Each point of a 3-D object (a house, for example) has three coordinates (x,y,z). This coordinate tuple can be calculated in the plane coordinates x and y for representation on the screen (because we are familiar only with these coordinates).

Before we do this, we must first describe the appearance of our 3-D coordinate system.

The x and z axes lie in the drawing plane and stand perpendicular to each other. The y axis passes through the drawing plane and forms the angle w with the x axis:



As can be seen from the drawing, the following formulas can be derived which perform the conversion between space and plane coordinates.

```
xplane = xspace + yspace*cos(w)
yplane = zspace + yspace*sin(w)
```

The angle w determines the perspective, the angle at which one looks at the object. If we want to shift the picture in order to place the object in the middle of the screen or move the object back farther into the distance, we can add three offsets a, b, and c to the x,z, and y coordinates (see section 5.1). We then get:

```
xplane = a+x + (y+c)*cos(w)
yplane = b+z + (y+c)*sin(w)
```

If we want to shift a figure in the plane only, we

choose c=0. It will often occur that parts of the picture cannot be seen because either they extend beyond the screen borders or because they are too small and the screen resolution does not permit them to be seen. For this reason we will again introduce the scaling factors which allow us to easily enlarge the object or change the length, height or depth of the image. In order to change all three factors, we must multiply the space coordinates by the three factors f1, f2, and f3 for the x, y, and z coordinates:

x_{plane} = f1*(a+x) + f3*(y+c)*cos(x)
y_{plane} = f2*(b+z) + f3*(y+c)*sin(x)

If you only want to enlarge the figure without distorting it, you would choose the same value for all three factors. What was said in section 5.1 also applies to these three factors:

f1/2/3 > 1 => enlargement
0 < f1/2/3 < 1 => reduction

In order to get the figure on the screen as exactly as possible and since the origin of the space coordinates does not always agree with the origin of the screen (upper left corner), we need to introduce two offsets v1 and v2. V1 shifts the plane coordinate system in the +x direction and v2 in the -y direction. To place the coordinate system origin at the center of the screen, for instance, you would select v1=160 and v2=100. Here too our y plane coordinates are upside down and we must change the sign. This results in the following two formulas:

```
xplane =  f1*(a+x) + f3*(y+c)*cos(w) + v1
yplane = -f2*(b+z) - f3*(y+c)*sin(w) + v2
```

The following program represents a house in three dimensions. The required corner points are placed with their three coordinates in DATA lines (lines 1110-1140). The 3-D coordinates are converted to plane coordinates and the individual points are connected together. The DATA lines following these contain the information concerning which points should be connected. The points are numbered starting at 1, and the numbers of the second points to be connected are placed sequentially in the DATA lines. The order in which the lines are drawn is irrelevant. Preceding the point and line data are the number of points (line 1100) and the number of lines (2100) which are to be drawn.

Once you become familiar with the form of the data, you can define and represent your own 3-D figures. For example, you could measure your computer and enter the data into the program, producing a three-dimensional picture of it.

Why go though all the trouble of entering the coordinates in three dimensions? By doing so we can enlarge, reduce, distort, rotate in perspective, or shift the figures as desired. You do this by changing the individual parameters a, b, c, f1, f2, f3, v1, v2 and w. It takes a while before you can see the results of the various changes. Remember that the scaling factors f1, f2, and f3 determine the units of the axes--the object will appear to move back and forth when these are changed. If this disturbs you, you can make a few simple changes to the above formula:

```
xplane =  f1*a+x + f3*y*cos(w)+c + v1
yplane = -f2*b-z - f3*y*sin(w)-c + v2
```

Now f1, f2, and f3 have no effect on the spacing, although this formula is somewhat objectionable mathematically. The scaling is applied only to the object, not to the coordinate system. One interesting thing is contained in lines 600-620. The space-x and space-z axes have just been drawn. In order to draw the y-axis fading off into the distance, we simply calculate the zero (or 199) point on the y axis (taking the angle w into consideration) and draw a line to it from the origin.

Important for all drawing procedures is the prior checking of the value for range overflow. A very good professional drawing program would still draw the visible part of a line even if one or two points of it (thereby including an end point) were outside the screen. Our simple program does not draw this line at all.

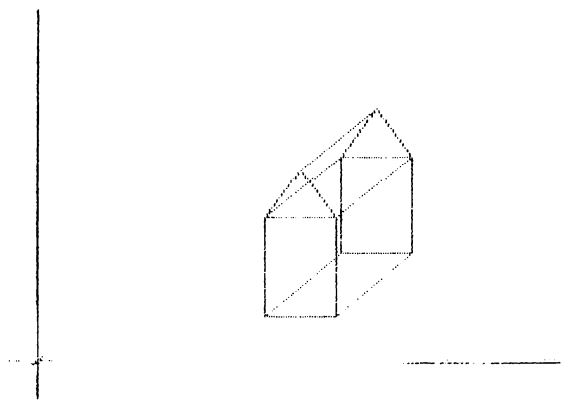
Likewise, the lines which are supposed to be hidden are drawn. This problem of hidden line removal is an important theme of 3-D graphics. Later we will present an extremely simple method for 3-dimensional functions.

Perhaps you would like to make other changes to this program. You can, for example, save the data now stored in DATA lines on disk in some form or other and then load it in again later. (DATA lines are naturally unsuited for this purpose. You could place all the data in arrays and save it onto disk as a sequential file.) Or you can convert the program into a completely menu-driven program; a small example of this can be found in section 5.1.

```
100 REM     ********************
110 REM     **                  **
120 REM     **  3-D DESIGNER    **
130 REM     **                  **
140 REM     ********************
150 REM
200 REM
210 REM       ******************
220 REM **** GRAPHICS ROUTINES ****
230 REM       ******************
240 REM
250 IN=51200 : OF=51203 :REM INIT  /GRAPHIC OFF
260 GC=51206 : SC=51209 :REM GCLEAR/SET COLOR
270 PC=51212 : PL=51215 :REM PCOLOR/PLOT
280 UP=51218 : SL=51221 :REM UNPLOT/SET LINE
290 CL=51224 : GL=51227 :REM CLINE /GLOAD
300 GS=51230 : HC=51233 :REM GSAVE /HARDCOPY
400 REM
410 REM          **********
420 REM    ****  PARAMETERS ****
430 REM          **********
440 REM
450 F1 = 5 : F2 = 5 : F3 = 3 : REM SCALES
460 AR = 15 : BR =  0 : CR = 10 : REM OFFSETS OF SPACE
    COORDS.
470 W = /4: REM VIEWING ANGLE (IN RADIANS)
480 SI = SIN(W) : CO = COS(W) : REM CONSTANTS (NOT CHANGED)
490 V1 = 100 : V2 = 180 : REM OFFSETS OF PLANE COORDS.
500 REM
510 REM          ************
520 REM    ****  CALCULATIONS ****
530 REM          ************
540 REM
```

```
550 SYS IN : SYS GC : SYS SC,16*7+8 : REM INITIALIZE GRAPHICS
560 FG=0:IF V1<0 OR V1>319 THEN FG=1:GOTO 580 : REM FLAG
570 SYS SL,V1,0,V1,199 : REM SPACE Z-AXIS
580 IF V2<0 OR V2>200 THEN FG=1:GOTO 600
590 SYS SL,0,V2,319,V2 : REM SPACE X-AXIS
600 Z2 = V1 - (199-V2)/SI*CO : Z1 = V1 + V2/SI*CO
610 IF FG=0 AND Z1>=0 AND Z1<320 THEN SYS SL,V1,V2,Z1,0:
    REM SPACE Y-AXIS UP
620 IF FG=0 AND Z2>=0 AND Z2<320 THEN SYS SL,V1,V2,Z2,199:
    REM SPACE Y-AXIS DOWN
630 READ NP : DIM X%(NP),Y%(NP) : REM NUMBER OF POINTS
640 FOR PN=1 TO NP : REM POINT NUMBER
650 READ XR,ZR,YR
660 X%(PN) =  F1*(XR+AR) + F3*(YR+CR)*CO+V1
670 Y%(PN) = -F2*(ZR+BR) - F3*(YR+CR)*SI+V2
680 NEXT PN : REM NEXT POINT
700 REM
710 REM         *****
720 REM     *** LINES ***
730 REM         *****
740 REM
750 READ NL : REM NUMBER OF LINES
760 FOR PN=1 TO NL
770 READ P1,P2 : REM READ POINT NUMBERS
775 IF X%(P1)<0 OR Y%(P1)<0 OR X%(P2)<0 OR Y%(P2)<0 THEN
    GOTO 790:REM OUTSIDE
777 IFX%(P1)>319ORY%(P1)>199ORX%(P2)>319ORY%(P2)>199THEN
    GOTO 790:REM OUTSIDE
780 SYS SL,X%(P1),Y%(P1),X%(P2),Y%(P2) : REM CONNECTION
790 NEXT PN : REM NEXT LINE
900 POKE 198,0 : WAIT 198,255 : GET A$
910 SYS OF : LIST : REM GRAPHICS OFF
1000 REM
```

```
1010 REM        ***********
1020 REM ****  COORDINATES  ****
1030 REM        ***********
1100 DATA 10 : REM NUMBER OF POINTS
1110 DATA  0, 0, 0,  6, 0, 0,  6,10, 0
1120 DATA  0,10, 0,  3,15, 0,  3,15,15
1130 DATA  6,10,15,  6, 0,15,  0, 0,15
1140 DATA  0,10,15
2000 REM
2010 REM        ***********
2020 REM ****  CONNECTIONS  ****
2030 REM        ***********
2100 DATA 17 : REM NUMBER OF LINES
2110 DATA  1, 2,  2, 3,  3, 4,  4, 1
2120 DATA  4, 5,  5, 3,  5, 6,  6, 7
2130 DATA  7, 3,  7, 8,  8, 2,  8, 9
2140 DATA  9, 1,  9,10, 10, 4, 10, 6
2150 DATA 10, 7
```



If you have typed this program in and tried it out, you will see how interesting it is to work with it. You might like to save a few pictures on disk or make a hardcopy on a printer.

If you want to rotate an object in space (not just change the perspective), you must first calculate the rotated coordinates x',y',z' from the unrotated space coordinates x,y,z before you can convert them to plane coordinates. This is done with the help of the following formulas:

```
Rotation about the x-axis:
x' =  x
y' =  y*cos(u) + z*sin(u)
z' = -y*sin(u) + z*cos(u)

Rotation about the y-axis:
x' =  x*cos(t) + z*sin(t)
y' =  y
z' = -x*sin(t) + z*cos(t)

Rotation about the z-axis:
x' =  x*cos(s) + y*sin(s)
y' = -x*sin(s) + y*cos(s)
z' =  z
```

whereby u,s,t : rotation angle about the given axis

If you want to rotate a figure about two axes, you must execute these rotations sequentially: First calculate the rotated coordinates for the rotation in the first axis, then place the resulting variables in the equations for the

rotation about the second axis. This also applies to a rotation about all three axes.

Creating three-dimensional pictures has applications in many areas. Because the computer can represent and change objects or situations true-to-scale in this manner, these techniques are well suited for construction or development of certain objects such as buildings, mechanisms, machines, or machine parts. Computer-aided design (CAD) is an important part of industrial and technical design. Cars and airplanes are designed, technical drawings are produced, or motors and machinery are placed on the screen and then onto paper. 3-D graphics are excellently suited for simulation of even complex procedures.

5.1.2.2 Central projection

When we look at objects, we can determine their distance from us because the farther away they are, the smaller they appear. The classic example is a long set of train tracks which grows steadily narrower and meets the horizon off in the distance. All lines appear to move toward a virtual point, the vanishing point. Our task is to develop a mathematical procedure to bring the parallel lines to a point. Because the derivation of the corresponding formulas would take too long, we will start the presentation with scaling (f1/f2/f3) and offsets (a/b/c). All of the axes are perpendicular to each other and the z-axis fades off beyond the screen plane (before it was the y-axis!):

$$x_{plane} = (f1*x+a)/q$$

$$y_{plane} = (f2*y+b)/q$$

with: $q = 1-(f3*z+c)/vpt$

```
x,y,z     : space coordinates
f1,f2,f3  : scaling in x,y, and z directions
a,b,c     : offsets in the x,y and z directions
vpt       : distance (z-coordinates) of the vanishing point
```

You must first calculate q so that you can later calculate the two desired plane coordinates. If you want to rotate your object in all three directions, this becomes considerably more complicated:

```
          x = (f1*(A*x + D*y + G*z)+a) / q
          y = (f2*(B+x + E*y + H*z)+b) / q
with:   q = 1-(f3*(C*x + F*y + I*z)+c) / q
```

This looks quite complicated. We have just begun, however. These are the definitions of A-I:

```
A =  cos(s)*cos(t)
B =  sin(s)*cos(t)
C = -sin(t)
D = -sin(s)*cos(u) + cos(s)*sin(t)*sin(u)
E =  cos(s)*cos(u) + sin(s)*sin(t)*sin(u)
F =  cos(t)*sin(u)
G =  sin(s)*sin(u) + cos(s)*sin(t)*cos(u)
H = -cos(s)*sin(u) + sin(s)*sin(t)*cos(u)
I =  cos(t)*cos(u)
```

```
s: angle of rotation about the z-axis
t: angle of rotation about the y-axis
u: angle of rotation about the x-axis
```

Such a mammoth task can naturally only be accomplished in machine language in any reasonable length of time. Even there we have time problems. The best option is to create a table of sine and cosine values (in a certain angle spacing, such as 1 degree) and then simply look up the appropriate value in the table instead of calculating it while the program is running.

5.1.2.3 3-D functions

One very attractive application of the 3-D technique is the representation of three-dimensional functions. Complicated connections which must otherwise by inferred from numerous unclear tables become readily visible through graphing. You are already acquainted with the use of two-dimensional diagrams to illustrate the course of sales throughout the year, for example. If you want to represent the dependence of sales on price of a particular product at the same time, you must draw such a diagram for each year. This leads to a collection of graphs superimposed on one another. Such a complex relationship can be simplified by using a single 3-D diagram. You can get a good, fast overview of the subject and can set the optimal price for your product in the next year (month) from such a diagram

Here we will present the technique of creating mathematical functions. In this case the individual values necessary to create the picture are calculated. You can of course gather these from various table, which generalizes the application range of the procedures described.

When graphing a 3-D function we use something called a value matrix. As an example we will use the function

$z = x^2 + y^2$

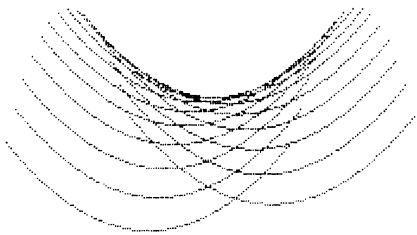
Recall from section 5.1.1 that the function value f(x) was obtained in a FOR/NEXT loop in which the value x (the so-called index variable) was continually incremented and the value y calculated from it. This is how we acquired our value pairs for graphing our function.

For three-dimensional functions, on the other hand, we require two index variables, x and y, because the function is now written f(x,y), with which we must calculate the third coordinate. In order to do justice to this situation we must use two FOR/NEXT loops, one nested within the other. In the first, the x value is incremented, and the second, the y value. The following program clarifies this state of affairs:

```
100 REM **********************
110 REM **                    **
120 REM **  3-D: Z=Y^2-X^2    **
130 REM **                    **
140 REM **********************
150 REM
230 REM **** GRAPHICS ROUTINES ****
240 REM      *****************
250 IN=51200 : OF=51203 :REM INIT  /GRAPHIC OFF
260 GC=51206 : SC=51209 :REM GCLEAR/SET COLOR
270 PC=51212 : PL=51215 :REM PCOLOR/PLOT
280 UP=51218 : SL=51221 :REM UNPLOT/SET LINE
290 CL=51224 : GL=51227 :REM CLINE /GLOAD
300 GS=51230 : HC=51233 :REM GSAVE /HARDCOPY
```

```
320 REM
330 REM **** PARAMETERS ****
340 REM      **********
400 W = 3.1415/8
410 A = 0 : B = 0 : C = 0
420 F1 = 20 : F2 = 5 : F3 = 8
430 V1 = 160 : V2 = 100
440 CO = COS(W) : SI = SIN(W)
520 REM
530 REM **** DRAW ****
540 REM      ****
600 SYS IN : SYS GC : SYS SC,16*1+4 : REM INITIALIZE GRAPHICS
610 FOR YR=3 TO -4 STEP -0.5
620 FOR XR=3 TO -3 STEP -0.05
630 ZR=YR*YR-XR*XR : REM FUNCTION
640 X=F1*(A+XR) + F3*(YR+C)*CO + V1
650 Y=F2*(B+ZR) + F3*(YR+C)*SI + V2
660 SYS PL,X,Y: REM DRAW POINT
670 NEXT XR,YR
1000 POKE 198,0:WAIT 198,255:SYS OF : REM GRAPHICS OFF
```

The FOR/NEXT loops extend from line 610 to 670. The two index values are used to calculate the z value in line 630, in order to convert these space coordinates into plane coordinates in the usual manner. The only problem here, as always, is the selection of parameters and the x and y value ranges. Because 3-D graphics has so many parameters, it is a good idea to begin with simple values for the parameters (such as: a,b,c=0, f1,2,3=1, w=3.1415/4, v1=160, v2=100).



Cross-hatching:

We can add a very nice effect to our drawing with a simple technique. So far we have gotten only single curves to communicate the three-dimensional aspect step by step. But you often sees quasi-curved grids which make the graphics look quite a bit more flexible (griding or cross-hatching). This is accomplished simply through an apparent rotation of the curve. The rotation is apparent because it is not really a rotation at all, but an alteration of the step size by which the x or y values are arrived at, which appears behind the STEP command of the two FOR/NEXT loops.

In the previous example, y was calculated with a step size of -0.5 and x with -0.05. What appears is a "stripe pattern" not an symmetrical plane which we would expect if both step sizes were the same.

If we exchange the step sizes, the second stripe pattern runs exactly perpendicular to the first. This is used in the next program:

```
100 REM ************************
110 REM **                    **
120 REM **  3-D: Z=Y^2-X^2    **
130 REM **       GRID         **
140 REM ************************
150 REM
230 REM **** GRAPHIC ROUTINES ****
240 REM      ****************
250 IN=51200 : OF=51203 :REM INIT  /GRAPHICS OFF
260 GC=51206 : SC=51209 :REM GCLEAR/SET COLOR
270 PC=51212 : PL=51215 :REM PCOLOR/PLOT
280 UP=51218 : SL=51221 :REM UNPLOT/SET LINE
290 CL=51224 : GL=51227 :REM CLINE /GLOAD
300 GS=51230 : HC=51233 :REM GSAVE /HARDCOPY
320 REM
330 REM **** PARAMETERS ****
340 REM      **********
400 W = 3.1415/8
410 A = 0 : B = 0 : C = 0
420 F1 = 20 : F2 = 5 : F3 = 8
430 V1 = 160 : V2 = 100
440 CO = COS(W) : SI = SIN(W)
520 REM
530 REM **** DRAW ****
540 REM      ****
600 SYS IN : SYS GC : SYS SC,16*1+4 : REM INITIALIZE GRAPHICS
610 SY=-0.5 : SX=-0.05 : REM STEP WIDTHS
620 FOR  CT=1 TO 2 : REM COUNTER
630 FOR YR=3 TO -4 STEP SY
640 FOR XR=3 TO -3 STEP SX
650 ZR=YR*YR-XR*XR : REM FUNCTION
660 X=F1*(A+XR) + F3*(YR+C)*CO + V1
```

```
670 Y=F2*(B+ZR) + F3*(YR+C)*SI + V2
680 SYS PL,X,Y: REM DRAW POINT
690 NEXT XR,YR
700 IF  CT=1 THEN GOSUB 1100 : SYS GC : SY=-0.05 : SX=-0.5
710 NEXT  CT
720 FOR T=8192 TO 8192+8000:POKET,PEEK(T)ORPEEK(T+8192):NEXTT
    :REM ORING
1000 POKE 198,0:WAIT 198,255:SYS OF:END: REM GRAPHICS OFF
1100 FOR T=8192 TO 8192+8000:POKET+8192,PEEK(T):NEXTT :
     REM TRANSMIT
1110 RETURN
```

You see that the graph is drawn twice in two different ways. After the first time, the entire graphics screen (at $2000 = 8192) is copied to or temporarily stored at $4000 (8192+8192 = 16384). Next we change the STEP and draw it again. Finally, the two graphics screens are combined with each other with OR which causes one to overlay the other. Have patience, the process of copying and overlaying takes quite a while!

In our special example the temporary storage and ORing were not necessary; we could have drawn the second picture directly on top of the first. This is procedure is necessary for our next step, however.

Hidden lines:

In this section we will present a simple procedure for erasing lines which should be hidden by planes lying in front of them using mathematical and other functions.

In daily life we cannot, as a general rule, see through solid objects. As you see when drawing our function, however, we can see through the object. Our lines show through although they should be hidden behind planes lying in front

of them.

For functions, their are two methods, among others, which can be used to suppress these lines or erase them after they have been drawn. The first is extremely simple, but effective:

When drawing our graphs, we note that we always draw from back to front with respect to the y-axis. If we calculate a point as usual and plot it on the screen, then we simply erase everything under the point in a line down to the end of the graphics window. This method hides everything behind each newly drawn plane, assuming that it lies in space beneath the drawn point. This produces a top view of the object. If you simply add a single line to the previous program you will see this effect:

```
685 SYS CL,X,Y+1,X,199 : REM ERASE UNDERLYING POINTS
```

The only problem with this are the edges of the graph. Here one should be able to see the "underside" of the graph. This is not possible, however, with our algorithm, as you will see if you RUN the program, because this "underside" is erased. In order to correct this defect one could erase not down to the edge of the screen but only between the current point and the point on the next line. To do this we must know the corresponding point on the next line. Drawing the graph using this method takes somewhat longer, however. Perhaps you would like to try to rewrite the previous program. There are some nice functions which you could try:

$$z = x^2+y^2$$

$$z = 1/(1+x^2+y^2)$$

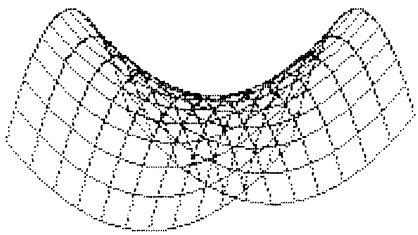
$$z = SQR(1-x^2/4-y^2/9)$$

$$z = \sin(x)/x+\sin(y)/y$$

```
z = sin(1/x)/x+sin(1/y)/y
```

You can also combine these functions. This produces some interesting results such as the following (a combination of equations 4 and 5):

```
z = sin(x)/x+sin(1/y)/y
```



5.1.2.4 Moving pictures in 3-D

In the last few paragraphs you have become acquainted with the steps necessary to create 3-dimensional pictures. If these objects are complex, plotting them takes quite a long time. How can we use this process to create moving sequences? There are two possibilities here. The first is relatively simple: You can draw a picture in a graphics storage area which is not visible at the moment while the

contents of another graphics storage area are displayed on the screen. When the picture is completed, you simply switch to the new graphics page (see section 3.3.2). The now-invisible first page can be used to create the next picture, and so on.

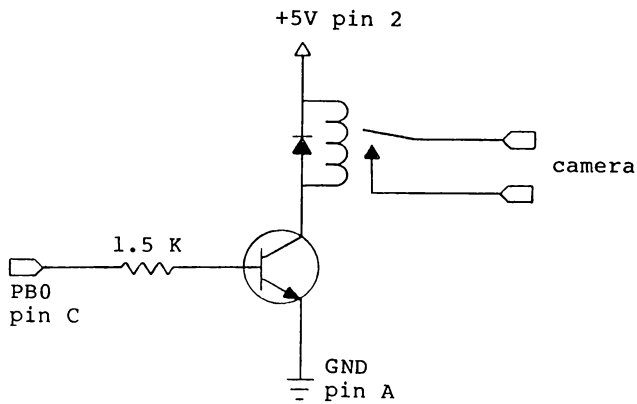
This method does not allow for very quickly moving pictures, but in spite of this, it is usable for certain effects. Rotations or step-wise enlargements of objects work quite well.

The second method is a bit more expensive. You need a movie camera with single frame control and remote release. Theoretically, it also possible to use a camera without such features. In such a situation you must press the trigger for a very brief length of time in order to take as few pictures as possible. With a bit of work you can get quite good results. Once these preparations are taken care of, you can produce your own computer graphics films. What can otherwise be seen only in the movie theater or in science fiction on television now comes to your home.

You need only point the camera at the screen (preferably on a tripod), connect the remote release and proceed: You program the computer with a set of pictures. The computation or creation time is irrelevant. Each time a picture is finished, press the remote release once in order to photograph it. This must be be done in small steps so that the motion does not proceed too fast. If in doubt, photograph a scene several times.

A more elegant method is to control the camera by the computer. This is possible if the camera has an electrical remote release. If you have only a cable release, camera stores sell adapters to remotely control the camera release. You can then let your computer run by itself.

In order to do this, you must build a small interface. The computer sends a signal via the user port each time a picture is supposed to be taken. The camera is then triggered by means of the circuit arrangement. You can build this interface yourself. You will need a user-port socket and the appropriate connector for the camera release in addition to the parts shown in the schematic.



This interface was tested and produces excellent results. The camera can be controlled with the following commands:

```
10 C2=56576 : REM BASE ADDRESS CIA 2 ($DD00)
20 POKE C2+2, PEEK(C2+2) OR 1 : REM PIN TO OUTPUT
30 POKE C2,0 : REM CAMERA OFF
40 POKE C2,1 : REM CAMERA ON
```

5.1.3 Graphing statistics

One popular application of graphics, especially for commercial use, is the representation of complicated tables in easily understandable diagrams or charts. Various methods of representation are used. The most important are:

- line chart
- bar chart
- pie chart

a) Line chart:

If you have many observations that are dependent on a single factor (such as the sales of a firm in dependence on the month, for various months), we would generally display the information in this form of a line chart. The same techniques are used for this type of representation as were used for graphing two-dimensional functions (section 5.1.1). The main difference between the two is the source of the individual data points. Using functions, they were calculated. Here they are taken from a two-dimensional table. You should read section 5.1.1 before proceeding. The rules for offsetting, scaling, and distortion of the graph are the same as in graphing the functions. The units on the two coordinate axes play a special role here and should be noted.

b) Bar charts:

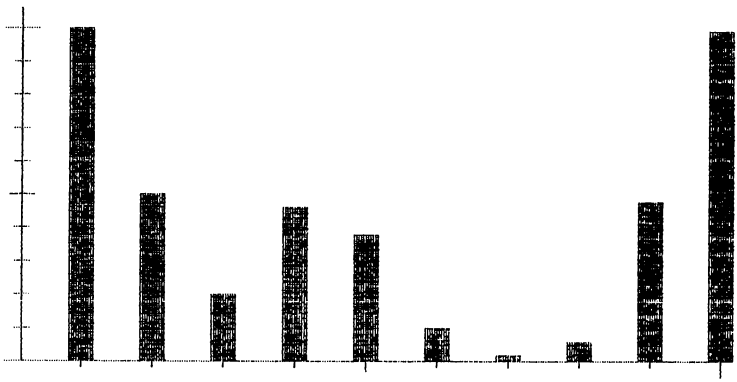
Bar charts are somewhat more complicated. Bar charts are used to graph relatively small data sets which are to be made easier to see. A value pair serves not only to determine the location of a point on the screen, but also the height and position of the bar which extends from

that point to the x-axis. The following program demonstrates such a chart. The numbers given in the DATA statement at the end of the program determine the height of each of the ten bars and may stand for anything. You could also ask for such data with INPUT and save these on diskette (or cassette) for later use.

```
100 REM ******************
110 REM **              **
120 REM **  BAR CHART   **
130 REM **              **
140 REM ******************
150 REM
230 REM **** GRAPHICS ROUTINES ****
240 REM      ******************
250 IN=51200 : OF=51203 :REM INIT  /GRAPHICS OFF
260 GC=51206 : SC=51209 :REM GCLEAR/SET COLOR
270 PC=51212 : PL=51215 :REM PCOLOR/PLOT
280 UP=51218 : SL=51221 :REM UNPLOT/SET LINE
290 CL=51224 : GL=51227 :REM CLINE /GLOAD
300 GS=51230 : HC=51233 :REM GSAVE /HARDCOPY
320 REM
330 REM **** AXES ****
340 REM      ****
350 SYS IN : SYS GC : SYS SC,16*7+10 : REM INITIALIZE
    GRAPHICS
360 READ CT : REM NUMBER OF VALUE PAIRS
370 READ HI : REM HIGHEST VALUE
380 XH = 300 : YH = 180 : REM NUMBER OF POINTS IN X/Y
    DIRECTIONS (MAX.)
390 XE=INT(XH/CT*1) : YE=INT(YH/HI*10) : REM CALCULATION
    OF (ONES) TENS
400 SYS SL,10,10,10,190 : REM Y AXIS
```

```
410 SYS SL,10,190,310,190 : REM X AXIS
420 T=-1:FOR Y=190 TO 10 STEP -YE
430 T=T+1:IF T/5-INT(T/5)=0 THEN SYS SL,5,Y,15,Y : REM
    LARGE LINE
435 IF T/10-INT(T/10)=0 THEN SYS SL,3,Y,17,Y : REM LARGE
    LINE
440 SYS SL,7,Y,13,Y : REM MARK
450 NEXT Y
460 T=0:BE=20+XE/2 : REM BEGIN FIRST LINE
470 FOR X=BE TO 310 STEP XE
480 T=T+1:IF T/5-INT(T/5)=0 THEN SYS SL,X,195,X,190 : REM
    LARGE LINE
485 IF T/10-INT(T/10)=0 THEN SYS SL,X,198,X,190 : REM
    LARGE LINE
490 SYS SL,X,193,X,190 : REM MARK
500 NEXT X
600 REM
610 REM **** CHART ****
620 REM      *****
630 BR=XE-20 : REM CALCULATE BAR WIDTH
640 PO=BE-BR/2 : REM START POSITION OF FIRST BAR
650 FOR T=1 TO CT
660 READ DA : REM READ DATA
670 Y=190-DA*YE/10 : REM BAR HEIGHT
680 FOR X=PO TO PO+BR : REM BAR WIDTH
690 SYS SL,X,Y,X,190 : REM DRAW BAR
700 NEXT X
710 PO=PO+XE : REM NEW BAR POSITION
720 NEXT T
800 POKE198,0:WAIT 198,255:SYS OF:END
900 REM
910 REM **** DATA ****
920 REM      ****
```

```
1000 DATA 10 : REM NUMBER OF VALUES
1010 DATA 100 : REM HIGHEST VALUE
1100 DATA 100,50,20,46,38,10,2,6,48,99
```



The program looks quite complicated because many formulas are used. These "formulas" are easy to understand, however, because they only determine the formatting of the axes and the bars.

We have written this program such that it can be used with a variable number and and general magnitude of data. Therefore, two additional values are required: The number of data and the largest value which will be read in lines 360-370. We have reserved a 300x180-point field for the chart (line 380).

The first problem is drawing the axes because we want them to bear the unit markers. The vertical (y-axis) displays each 10th unit with a small line. Every 50th unit is denoted by a mark twice as long as the first, and

each 100th unit with a mark three times as long. On the horizontal (x-axis), the same division is used except every first, fifth, and tenth units are marked.

First we calculate the number of points which will fall on the one (for x-axis) or ten (for y-axis) units place so that we do not go too far right or too far up, but still use the entire screen areas (line 390).

Next we draw the plane axes (lines 400-410). Then the axes are marked according to the scheme given above. This procedure should be relatively easy to understand. A few comments: In lines 430, 435, 490, and 495, the units are checked to see if the 5th or 10th marker is being drawn and, if so, this marker is correspondingly lengthened. The expression behind the IF does nothing more than remove the integer portion of the number (also called fraction--the "opposite" of INT). The starting point of the x-scaling is calculated in line 460. This formula is also partially responsible for the width of the bars.

Now the actual bars are drawn. The bar width is set so that sufficient space is left between two bars (line 630). You should be able to understand the formula for the starting position of the bars and their height.

You can change the data at line 1000 as desired but you should be careful not to use any negative numbers and not to use too many. With extremely high values you should change the scale units of the axes.

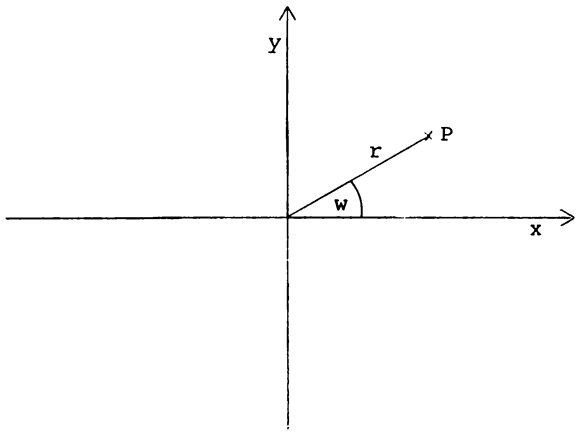
c) Pie charts:

This method of graphing statistics is suited for clearly representing divisions of sets into sub-parts and the relationship of the parts to the whole. Here a pie (circle) is drawn which represents the sum of all of the elements (100%) and the various pieces of the pie are

then drawn. The size of the piece gives the relationship between this piece and the whole as well as to other pieces.

We often see such charts in an atlas or encyclopedia to demonstrate the relationship between goods imported and goods exported, or for any application in which we deal with percentages or portions of a finite and well-defined whole.

How can this be done within a program? It will have something to do with the rather complex relationships in a circle which must be divided into certain angle sections. We are already acquainted with the creation of circles (ellipses) from section 4.2.2.3. The circle formula given there is not suited for drawing sections defined by angle, however. Therefore we will here introduce you to the creation of circles using polar coordinates (as mentioned in section 4.2.2.3). Polar coordinates are an alternative method for representing functions. Something called a polar coordinate system is used, which uses an angle, w, and a radius, r, to determine the position of a point instead of x and y axes. The radius gives the distance of the point (in a straight line) from the origin. This looks something like this:



A circle can then be constructed simply by changing the angle w and keeping r a constant. In general for any desired ellipse with center at the origin (0,0) we can convert from the polar coordinates (w,r) into Cartesian coordinates (x,y) in the following manner:

```
x = a*cos(w)
y = b*sin(w)
```

whereby the parameters are:

a: radius of the ellipse in the x direction
b: radius of the ellipse in the y direction

With the help of these two formulas it is possible to determine a given point on the ellipse by specifying the angle of the polar coordinates of the point.

One thing must still be clarified. There are various ways of specifying an angle:

- specification in degrees (0-360)
- specification in radians (0-2*pi)

The most familiar to us is the first method. Our computer uses the second, however, in which 360 degrees correspond to the value 2*pi (pi=3.1415...). This value in turn corresponds to the length of an arc with radius r about the specified angle. The following formulas allow you to convert between radians and degrees:

```
deg = 180*2*pi/rad     or
rad = 180*2*pi/deg
```

Now we have the necessary tools for understanding the following program:

```
100 REM *****************
110 REM **               **
120 REM **  PIE CHART    **
130 REM **               **
140 REM *****************
150 REM
230 REM **** GRAPHICS ROUTINES ***
240 REM      *****************
250 IN=51200 : OF=51203 :REM INIT  /GRAPHICS OFF
260 GC=51206 : SC=51209 :REM GCLEAR/SET COLOR
270 PC=51212 : PL=51215 :REM PCOLOR/PLOT
280 UP=51218 : SL=51221 :REM UNPLOT/SET LINE
290 CL=51224 : GL=51227 :REM CLINE /GLOAD
300 GS=51230 : HC=51233 :REM GSAVE /HARDCOPY
```

```
320 REM
330 REM **** ELLIPSE ****
340 REM      *******
350 PI=3.1415
360 SYS IN : SYS GC : SYS SC,16*5+13 : REM INITIALIZE
    GRAPHICS
370 A = 100 : B = 60 : V1=160 : V2=80 : REM PARAMETERS
380 AN=0:GOSUB 930:X1=X:Y1=Y: REM SET X1 AND Y1
390 SP=7*PI/180 : REM STEP
400 BE=0:EN=2*PI : REM START AND END ANGLE
410 GOSUB 800 : REM DRAW ELLIPSE
420 BE=0:EN=1.03*PI : REM APPROX. 180 DEGREES
430 V2=100 : REM SET DEEPER
440 GOSUB 800 : REM DRAW ELLIPSE
500 REM
510 REM **** PIE PIECES ***
520 REM      **********
530 READ C  : DIM P(C)  : REM NUMBER OF PARTS
540 FOR S=1 TO C
550 READ P(S) : REM READ DATA
560 SU=SU+P(S) : REM FORM SUM
570 NEXT S
580 AN = 0 : REM START ANGLE
590 FOR S=1 TO C
600 PR=P(S)/SU : REM CALCULATE PERCENT
610 AS=2*PI*PR : REM ANGLE OF SECTION
620 AN=AN+AS : REM REAL ANGLE
630 V2=80:GOSUB 930:REM CALCULATE COORDINATES
640 SYS SL,V1,V2,X,Y : REM DIVIDING LINE
650 IFAN>PI THEN 690 : REM ONLY ON VISIBLE SIDE
660 X1=X:Y1=Y : REM SAVE
670 V2=100:GOSUB 930:REM CALCULATE LOWER COORD.
680 SYS SL,X1,Y1,X,Y : REM VERTICAL TO LOWER ARC
```

```
690 NEXT S
799 WAIT 198,255:SYS OF:END
800 REM
810 REM **** ELLIPSE ARCS ****
820 REM      ************
830 FORAN=BE TO EN+SP STEP SP : REM DETERMINE ANGLE
840 GOSUB 930 : REM DETERMINE COORDINATES
850 SYS SL,X1,Y1,X,Y : REM LINE
860 X1=X:Y1=Y
870 NEXTAN : RETURN
900 REM
910 REM **** CALCULATE POINT ****
920 REM      ***************
930 X = A*COS(AN) + V1
940 Y = B*SIN(AN) + V2 : RETURN
1000 REM
1010 REM **** DATA ****
1020 REM      ****
1100 DATA 6 : REM NUMBER
1110 DATA 20,10,15,40,30,8
```

You can use the lines 800-940 in your own programs as an alternate method of forming circles. The parameters are:

BE: start angle
EN: end angle
SP: step unit
V1: x-coordinate of center
V2: y-coordinate of center
A : x radius
B : y radius

The variable SP determines the angle spacing in which the individual points of the ellipse will be calculated, the accuracy. These points are joined by a line in line 850. If you choose a large SP such as 30 or 45, you will get a special effect: a polygon (such as an octagon) will drawn, which can be used for other applications.

Line 380 must be included in and executed by any programs in which you use only the ellipse routine. Removing it can also create certain special effects (see below).

In the previous program we did not want to draw a simple circle and then divide it into the appropriate sections. Rather, the whole format should have a 3-D look to it. A round cylinder with a certain thickness is drawn and then divided into the various "pie" pieces and appears to be viewed at a diagonal above it. We first draw an ellipse as the top surface (line 410). Then we draw the same figure, only a few points lower, and as only half an ellipse in order to represent the lower edge of the cylinder (line 440). The construction of the cylinder is slightly "fudged" because we must represent the sides as straight lines (which looks proper), so we draw the ellipse just over 180 degrees (r*pi), which can hardly be noticed due to the resolution.

Now we come to the actual pie pieces:

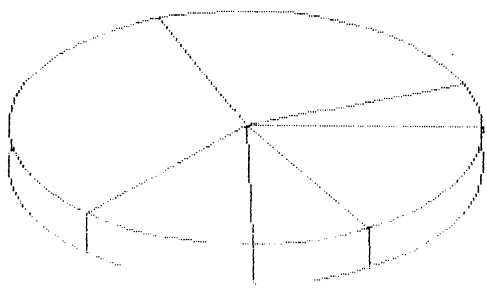
First we should say something about the data structure used. At the start of the actual data is the total number of items, as usual. Then follow the corresponding data, which may be of any size. These determine the relative size of each of the pieces.

In the first part of the routine beginning at line 500, N number of data are read into a single-dimensional array and (this is the real reason for the loop) the sum

of the parts is formed (lines 540-570).

The divisions are dealt with in the next FOR/NEXT loop: Line 600 calculates the percentage of the whole which each piece represents. Line 610 determines the resultant size of the circle (ellipse) section in radians which is then used for controlling the incrementation of the actual angle W (line 620). Since the angle of the section in question is known, the position of the corresponding point on the ellipse can be calculated. A line is then drawn from this point to the origin of the ellipse (line 640).

Now we must decide if this section is found on the back or front half of the cylinder. If it is on the back half, we simply proceed to the next value. If it is front, we must draw a line down the front of the cylinder. This is done by simply calculating the position on the lower edge and drawing a line from the point on the ellipse to this point (line 650-680). This suffices to complete the three-dimensional effect.



Naturally, this program is extensible. You could, for example, draw the pieces in different colors or with cross-hatching, and so on.

5.2 Moving messages

One enchantingly simple and striking method of placing moving letters on the screen is the construction of letter sprites. The sprites possess a fantastic resolution for creating type and can be easily enlarged, moved, etc.

The only problem with this undertaking is the limited memory space which is at our disposal in BASIC. The four blocks which can be used freely for sprites from BASIC are not enough for this purpose. We can move the entire VIC address range up to the 16, 32, or 48K range (see section 3.3.2) to get more room. We do encounter a small difficulty here, however. This procedure moves not only the sprite blocks, but all of the screen storage, including video RAM and graphics storage which considerably complicates matters.

We want to make use of a second possibility: We pack all of the sprite definitions which we use in our program into the area from $2000-$3FFF (8192-16383) where we normally place our graphics storage. Of course, we can no longer display graphics as long as we are using the sprites, but one must be prepared to make compromises. We use blocks 8192/64=128 to 255 and have space for a total of 127 sprite definitions, which should suffice quite well. We need only take care that we do not write very large programs or programs which require a great deal of storage space.

Let us move on to the letter definitions. Normally it is a waste of space to put sprites in DATA statements within

our BASIC program. Alternative methods were presented in section 4.3. Here we will make an exception, principally because we are working with only a few letter definitions.

You can create the individual sprites with the sprite editor in section 4.3 and later read them directly back into memory. You can create an entire alphabet in this manner and call them up as needed.

We want to stick to our original task: programming moving text. Here is a small BASIC program:

```
100 REM ********************
110 REM **                  **
120 REM **  MOVING TEXT     **
130 REM **                  **
140 REM ********************
150 REM
160 V = 53248 : REM BASE ADDRESS OF VIDEO CONTROLLER
170 POKE V+32,0 : POKE V+33,0 : REM BORDER AND BACKGROUND =
    BLACK
175 PRINT CHR$(147)
180 REM
190 REM **** STORE ****
200 REM      *****
210 FOR X=1 TO 9 : REM 9 LETTERS
220 READ A$ : REM NAME OF THE LETTER
230 BK=ASC(A$)-32+128 : REM SPRITES ORDERED BY ASCII IN
    MEMORY (BLOCK NUMBER)
240 AD=BK*64 : REM ADDRESSES AT 8192
250 FOR Y=AD TO AD+62
260 READ DA : POKE Y,DA : REM READ AND WRITE DATA
270 NEXT Y,X : REM READ 63 DATA/8 LETTERS
300 REM
310 REM **** INITIALIZATION ****
```

```
320 REM      **************
330 FOR X=0 TO 7
340 POKE V+39+X,X+1 : REM SET COLORS OF SPRITES 0-7
350 NEXT X
360 POKE V+23,255 : REM ALL SPRITES LARGE (Y)
370 POKE V+29,255 : REM ALL  SPRITES LARGE (X)
380 POKE V+27,0 : REM PRIORITY
390 POKE V+28,0 : REM NORMAL COLORS
400 SP=0 : REM START SPRITE NUMBER
410 CN=8 : REM NUMBER OF LETTERS ON THE SCREEN (MAX. 8)
420 SG=330/CN : REM SPACING OF SPRITES
430 SD = 20 : REM SPEED
500 REM
510 REM **** MOVING TEXT ****
520 REM      ***********
530 TE$="DATA-BECKER---" : REM TEXT
540 FOR LP=1 TO 10 : REM 10 LOOPS
550 FOR LT=1 TO LEN(TE$)
560 LT$=MID$(TE$,LT,1) : REM RUNNING LETTER
570 BK =ASC(LT$)-32 + 128 : REM BLOCK NUMBER OF LETTER
580 POKE 2040+SP,BK : REM SET SPRITE TO CORRESPONDING BLOCK
590 POKE V+SP*2,95 : REM SPRITE X-COORD. LOW BYTE
600 POKE V+SP*2+1,100 : REM SPRITE Y-COORD.
610 POKE V+16,PEEK(V+16) OR 2^SP : REM SPRITE X-COORD. HIGH
    BIT
620 POKE V+21,PEEK(V+21) OR 2^SP : REM SPRITE ON
630 SP=SP+1 : REM NEXT SPRITE NUMBER
635 IF SP=CN THEN SP=0
640 FOR K=1 TO SG STEP SD : REM SD STEPS *** MOVE ***
650 FOR S=0 TO CN-1 : REM MOVE CN SPRITES
660 AD=V+S*2:XL=PEEK(AD)-SD:XH=PEEK(V+16)AND2^S:
    REM X-COORDINATE OF THE SPRITES
670 IF XL<0 THEN XL=256+XL:POKE V+16,PEEK(V+16) AND 255-2^S:
```

```
    IFXH=0THENXL=0
680 POKE AD,XL
685 GOSUB 800
690 NEXT S,K
700 NEXT LT : REM NEXT LETTER
710 NEXT LP : REM NEXT LOOP
800 POKE V+21,PEEK(V+21) AND 255-2^SP:RETURN : REM SPRITE OFF
1000 REM
1010 REM **** SPRITE DATA ****
1020 REM      ***********
1090 REM
1100 DATA A : REM LETTER A
1110 DATA 000,000,000, 000,000,000, 003,255,192
1120 DATA 007,000,224, 006,000,096, 006,000,096
1130 DATA 006,000,096, 006,000,096, 006,000,096
1140 DATA 006,000,096, 007,255,224, 006,000,096
1150 DATA 006,000,096, 006,000,096, 006,000,096
1160 DATA 006,000,096, 006,000,096, 006,000,096
1170 DATA 006,000,096, 000,000,000, 000,000,000
1190 REM
1200 DATA B : REM LETTER B
1210 DATA 000,000,000, 000,000,000, 003,255,000
1220 DATA 003,001,128, 003,000,192, 003,000,192
1230 DATA 003,000,192, 003,000,192, 003,001,128
1240 DATA 003,255,000, 003,001,128, 003,000,192
1250 DATA 003,000,096, 003,000,096, 003,000,096
1260 DATA 003,000,096, 003,000,192, 003,001,128
1270 DATA 003,255,000, 000,000,000, 000,000,000
1290 REM
1300 DATA C : REM LETTER C
1310 DATA 000,000,000, 000,000,000, 001,255,192
1320 DATA 003,129,192, 003,000,000, 003,000,000
1330 DATA 003,000,000, 003,000,000, 003,000,000
```

```
1340 DATA 003,000,000, 003,000,000, 003,000,000
1350 DATA 003,000,000, 003,000,000, 003,000,000
1360 DATA 003,000,000, 003,000,000, 003,129,192
1370 DATA 001,255,192, 000,000,000, 000,000,000
1390 REM
1400 DATA D : REM LETTER D
1410 DATA 000,000,000, 000,000,000, 015,255,000
1420 DATA 012,001,128, 012,000,192, 012,000,192
1430 DATA 012,000,192, 012,000,192, 012,000,192
1440 DATA 012,000,192, 012,000,192, 012,000,192
1450 DATA 012,000,192, 012,000,192, 012,000,192
1460 DATA 012,000,192, 012,000,192, 012,001,128
1470 DATA 015,255,000, 000,000,000, 000,000,000
1500 DATA E : REM LETTER E
1510 DATA 000,000,000, 000,000,000, 003,255,192
1520 DATA 003,001,192, 003,000,000, 003,000,000
1530 DATA 003,000,000, 003,000,000, 003,006,000
1540 DATA 003,255,000, 003,006,000, 003,000,000
1550 DATA 003,000,000, 003,000,000, 003,000,000
1560 DATA 003,000,000, 003,000,000, 003,001,192
1570 DATA 003,255,192, 000,000,000, 000,000,000
1590 REM
1600 DATA K : REM LETTER K
1610 DATA 000,000,000, 000,000,000, 003,001,128
1620 DATA 003,003,000, 003,006,000, 003,012,000
1630 DATA 003,024,000, 003,048,000, 003,096,000
1640 DATA 003,192,000, 003,192,000, 003,096,000
1650 DATA 003,048,000, 003,024,000, 003,012,000
1660 DATA 003,006,000, 003,003,000, 003,001,000
1670 DATA 003,000,192, 000,000,000, 000,000,000
1690 REM
1700 DATA R : REM LETTER  R
1710 DATA 000,000,000, 000,000,000, 001,255,000
```

```
1720 DATA 003,001,128, 003,000,192, 003,000,192
1730 DATA 003,000,192, 003,000,192, 003,001,128
1740 DATA 003,255,000, 003,192,000, 003,096,000
1750 DATA 003,048,000, 003,024,000, 003,012,000
1760 DATA 003,006,000, 003,003,000, 003,001,000
1770 DATA 003,000,192, 000,000,000, 000,000,000
1790 REM
1800 DATA T : REM LETTER T
1810 DATA 000,000,000, 000,000,000, 015,255,192
1820 DATA 012,048,192, 000,048,000, 000,048,000
1830 DATA 000,048,000, 000,048,000, 000,048,000
1840 DATA 000,048,000, 000,048,000, 000,048,000
1850 DATA 000,048,000, 000,048,000, 000,048,000
1860 DATA 000,048,000, 000,048,000, 000,048,000
1870 DATA 000,048,000, 000,000,000, 000,000,000
1890 REM
1900 DATA "-" : REM DASH
1910 DATA 000,000,000, 000,000,000, 000,000,000
1920 DATA 000,000,000, 000,000,000, 000,000,000
1930 DATA 000,000,000, 000,000,000, 000,000,000
1940 DATA 000,000,000, 003,255,192, 000,000,000
1950 DATA 000,000,000, 000,000,000, 000,000,000
1960 DATA 000,000,000, 000,000,000, 000,000,000
1970 DATA 000,000,000, 000,000,000, 000,000,000
```

This program can only communicate the fundamentals of moving text. It is up to you to write the corresponding applications programs. The whole thing naturally goes quite quickly in machine language. If you look through the program, the course of execution should be made quite understandable by the numerous REM lines.

5.3 The secrets of the games

There are already many good games available for the Commodore 64. Games show off the capabilities and qualities of a computer very well since they often push the computer to its limits. These limits are set fairly high on the '64 and even the best game programmers find it difficult to make full use of all of its capabilities. But what good is it to have a computer which is capable of doing so many things if you don't know how to make it do them?

In this book you have already learned much which will help you to make optimum use of graphics, sprites, and screen output in general. Naturally, not all of what the Commodore 64 has to offer as explained in Chapter 3 (Hardware Foundations), can be brought to application.

Most games are written in machine language because BASIC is simply too slow. There are some utilities written in assembly language which can be used as an extension of the BASIC command set for improving the speed of your programs (such as the graphics package). Here we will present some techniques and extensions especially for games (they are also quite useful for other applications, of course). This will enable you to write fast games even in BASIC.

5.3.1 Animation

Animation is the creation of moving pictures on the screen. Naturally, video games "live" on animation. Without movement on the screen, it just doesn't get anywhere. We therefore want to occupy ourselves with this important theme first.

On the Commodore 64 one distinguishes between 5 types of animation:

- internal sprite motion
- sprite movement
- internal character motion
- character movement
- graphical animation

By "internal" we mean motion within the object itself, without changing its position on the screen.

We have already presented and explained the first two types in detail in the examples of sprite programming in section 4.3.2. You should read those paragraphs in any case because sprite programming is one of the more important fundamentals of video games, if not the most important.

We also will not discuss the last type here because it is derived from the various sections on graphics and for reasons of speed is only rarely found in games.

The other two points are extremely popular, however. They are generally used in connection with altering the character set.

a) Internal character motion:

The principle of internal character motion is the same as that for sprite motion. A figure composed of one or more characters is changed through the constant altering of the condition of certain parts of the figure so that it appears to move.

Assume that we want to control a small man such that it constantly moves its arms and legs apart. We compose this man of several parts so that it does not become too small. In order to program the desired motion, we prepare

two or more men which represent other phases of the motion. These various figures are then displayed at the same place on the screen in sequential order at a given rate of change, and we have the desired effect.

In order to bring the diverse characters to a specific position on the screen, we use the routine presented in section 4.1 for cursor control within a program. The animation would be particularly effective with an altered character set or a few changed characters. Section 4.4 illustrates how this can be done simply. Multicolor characters are especially useful here. You can see already that all of a programmer's knowledge and abilities come to a head when working with games.

The following example will illustrate the use of the original character set for this theme:

```
100 REM ********************
110 REM **                  **
120 REM **  ANIMATION-1     **
130 REM **                  **
140 REM ********************
150 REM
160 A$(0)=" "       +CHR$(119)
170 A$(1)=CHR$( 99)+CHR$(123)+CHR$( 99)
180 A$(2)=CHR$(167)+CHR$(183)+CHR$(165)
190 A$(3)=" "       +CHR$(113)
200 A$(4)=CHR$(173)+CHR$(123)+CHR$(189)
210 A$(5)=CHR$(183)+CHR$(183)+CHR$(183)
300 PRINT CHR$(147) : REM CLEAR SCREEN
310 X=18 : REM CURSOR POSITION
320 FOR I =0 TO 5 STEP 3
330 Y=12 : GOSUB 1000 : REM POSITIONING
340 PRINT A$(I):GOSUB 1010 : REM HEAD
```

```
350 PRINT A$(I+1):GOSUB 1010 : REM BODY/ARMS
360 PRINT A$(I+2) : REM LEGS
370 FOR S=1 TO 100 : NEXT S : REM DELAY LOOP
380 NEXT I
390 GOTO 320
400 REM
410 REM **** POSITIONING ****
420 REM      ***********
1000 PRINT CHR$(19);:IF Y>0 THEN FOR T=1 TO  Y:PRINT:
     NEXT T
1010 PRINT TAB(X);: RETURN
```

At the beginning of this program (lines 160-210) the parts for the two phases of the motion of the man are defined. We construct it out of a total of 7 characters which are organized into 3 consecutive rows. Each of the three rows is placed in a separate array (A$(...)--numbers 0-2 for the first phase and numbers 3-5 for the second). Then the starting coordinates of the man are determined and positioned (line 330). The rest is relatively easy to understand: The three rows are drawn (1st phase) whereby only a portion of the positioning routine is called. In the second execution of the FOR/NEXT loop, the rows of the 2nd phase are drawn. The speed of the motion can be controlled by changing the length of the delay loop in line 370. You should try to represent this man using custom characters (see section 4.4). Then it is possible to execute more than two successive motion phases, which naturally enhances the effect.

b) Character movement:

As with sprites, we can also change the position of our man on the screen. This is done in a manner similar

to that given in section 4.3. We draw our man at a specific position on the screen. After a delay, we erase it again and redraw it shifted a bit to the left, right, top, or bottom, etc. A continual movement results from this steady shifting. The only problem here is the resolution of the movement. Normally, we can only move our object in increments of one character width (8 pixels). We can correct this defect by programming intermediate phases where the object is between two character locations. This must be done by changing the character set.

The following program combines the internal motion technique with the character movement to create a very nice looking picture. If we control this man with a joystick, it works quite well.

```
100 REM ********************
110 REM **                  **
120 REM **  ANIMATION-2     **
130 REM **                  **
140 REM ********************
150 REM
160 A$(0)=" "+" "       +CHR$(119)+" "       +" "
170 A$(1)=" "+CHR$( 99)+CHR$(123)+CHR$( 99)+" "
180 A$(2)=" "+CHR$(167)+CHR$(183)+CHR$(165)+" "
190 A$(3)=" "+" "       +CHR$(113)+" "       +" "
200 A$(4)=" "+CHR$(173)+CHR$(123)+CHR$(189)+" "
210 A$(5)=" "+CHR$(183)+CHR$(183)+CHR$(183)+" "
300 PRINT CHR$(147) : REM CLEAR SCREEN
305 SP=1:B=0:E=33
310 FOR X=B TO E STEP SP : REM COLUMN POSITION
320 FOR  I=0 TO 5 STEP 3
330 Y=12 : GOSUB 1000 : REM POSITIONING
340 PRINT A$(I):GOSUB 1010 : REM HEAD
```

```
350 PRINT A$(I+1):GOSUB 1010 : REM BODY/ARMS
360 PRINT A$(I+2) : REM LEGS
370 FOR S=1 TO 60 : NEXT S : REM DELAY LOOP
380 X=X+SP
390 NEXT  I
400 X=X-SP
410 NEXT X
420 SP=-SP :  TM=B :  B=E :  E=TM :  GOTO 310 :  REM SWAP
(CHANGE MOVE DIRECTION)
500 REM
510 REM **** POSITIONING ****
520 REM      ***********
1000 PRINT CHR$(19);:IF Y>0 THEN FOR T=1 TO Y:PRINT:
     NEXT T
1010 PRINT TAB(X);: RETURN
```

We have placed an additional FOR/NEXT loop around the motion routine. In addition, we must expand the row definitions in lines 160-210 with spaces before and after so that we do not have to erase the entire character when it is moved, provided we move the object in one character increments. The variables SP, B, and E contain the necessary parameters for right and left movement.

With this we have given you some tips which, in addition to sprites, allows you to add some simple animation to your games.

5.3.2 Scrolling

Few have not seen or played with the video arcade game "Defender" or similar action games in which you control a space ship that flies through space with unearthly speed. Or

you are driving a race car, concentrating on the race track, pursued by rivals who seek to run you off the road.

If you look a bit more closely, you will see that the space ship or the car does not really move, but rather the background moves past it. The entire screen (or part of it) is moved a certain direction while the object (a sprite) usually has limited movement. Such movement or scrolling of the screen is a very time-consuming process and can therefore by executed only in machine language. The text mode is usually used instead of graphics (for which 8K of memory must be moved), which is little limitation for games because we can change characters as desired and so get a quasi-high-resolution picture. In the text mode we only have to move 1K bytes which reduces the work considerably. The following assembly-language routine can be called in the same manner as the graphics package commands:

```
10:    CC00                    *=    $CC00
20:                    ;
30:                    ;***************
40:                    ;**            **
50:                    ;** SCROLLING **
60:                    ;**            **
70:                    ;***************
80:                    ;
110:   B7F1            CHKGET   =     $B7F1
120:   07F6            UP       =     2038
130:   07F7            DOWN     =     2039
140:   00FD            FLAG     =     $FD
150:   00FE            NUMBER   =     $FE
160:   0061            ADDRESS  =     $61
200:   CC00 20 F1 B7   START    JSR   CHKGET   ;GET COMMA AND
                                                BYTE
```

```
210:   CC03 8A              TXA          ;RIGHT/LEFT FLAG
220:   CC04 4A              LSR  A
230:   CC05 08              PHP
240:   CC06 20 F1 B7        JSR  CHKGET
250:   CC09 E0 19           CPX  #25     ;TOO LINE
300:   CC0B 90 02           BCC  S1
310:   CC0D A2 18           LDX  #24
320:   CC0F 8E F6 07 S1     STX  UP
330:   CC12 20 F1 B7        JSR  CHKGET  ;BOTTOM LINE
340:   CC15 E0 19           CPX  #25
350:   CC17 90 02           BCC  S2
360:   CC19 A2 18           LDX  #24
370:   CC1B 8E F7 07 S2     STX  DOWN
380:   CC1E 8A              TXA
390:   CC1F AE F6 07        LDX  UP
400:   CC22 AC F7 07        LDY  DOWN
410:   CC25 38              SEC
420:   CC26 ED F6 07        SBC  UP      ;UP-DOWN
430:   CC29 B0 08           BCS  S3
440:   CC2B 49 FF           EOR  #$FF    ;SWAP UP<DOWN=>
450:   CC2D AE F7 07        LDX  DOWN
460:   CC30 AC F6 07        LDY  UP
470:   CC33 85 FE    S3     STA  NUMBER  ;COUNTER
480:   CC35 28              PLP
490:   CC36 08              PHP          ;RT/LT FLAG
500:   CC37 90 03           BCC  S4
510:   CC39 C8              INY          ;RIGHT LINE
                                          FARTHER
520:   CC3A 98              TYA          ;AND START DOWN
530:   CC3B AA              TAX
540:   CC3C BD CB CC S4     LDA  MULH,X  ;HIGH BYTE ADDRESS
550:   CC3F 85 62           STA  ADDRESS+1
560:   CC41 BD E5 CC        LDA  MULL,X  ;LOW BYTE
```

```
570:   CC44 85 61            STA  ADDRESS
580:   CC46 28               PLP
590:   CC47 08               PHP          ;RT/LT FLAG
600:   CC48 90 08            BCC  MOVE
610:   CC4A E9 01            SBC  #1      ;-1 FOR RIGHT
620:   CC4C 85 61            STA  ADDRESS
630:   CC4E B0 02            BCS  MOVE
640:   CC50 C6 62            DEC  ADDRESS+1
650:                ;
660:                ;MOVE
670:                ;****
680:                ;
690:   CC52 A5 62    MOVE    LDA  ADDRESS+1
700:   CC54 29 03            AND  #3
710:   CC56 09 04            ORA  #4      ;BASE ADDRESS =
                                            $0400
720:   CC58 28               PLP
730:   CC59 08               PHP          ;FLAG
740:   CC5A 20 86 CC         JSR  MOVE1   ;MOVE VIDEO RAM
750:   CC5D 28               PLP
760:   CC5E 08               PHP          ;FLAG
770:   CC5F A5 61            LDA  ADDRESS
780:   CC61 90 0A            BCC  M1
790:   CC63 69 27            ADC  #39     ;C=1! /RIGHT
800:   CC65 85 61            STA  ADDRESS
810:   CC67 90 0C            BCC  M2
820:   CC69 E6 62            INC  ADDRESS+1
830:   CC6B B0 08            BCS  M2      ;ABSOLUTE
840:   CC6D E9 27    M1      SBC  #39     ;C=0! /LEFT
850:   CC6F 85 61            STA  ADDRESS
860:   CC71 B0 02            BCS  M2
870:   CC73 C6 62            DEC  ADDRESS+1
880:   CC75 A5 62    M2      LDA  ADDRESS+1
```

```
890:   CC77 29 03          AND  #3
900:   CC79 09 D8          ORA  #$D8     ;COLOR RAM AT
                                          $D800
910:   CC7B 28             PLP
920:   CC7C 08             PHP
930:   CC7D 20 86 CC       JSR  MOVE1    ;MOVE COLOR RAM
940:   CC80 C6 FE          DEC  NUMBER
950:   CC82 10 CE          BPL  MOVE
960:   CC84 28             PLP
970:   CC85 60             RTS
980:                ;
990:                ;DIVIDER
1000:               ;******
1010:               ;
1020:  CC86 85 62   MOVE1  STA  ADDRESS+1
1030:  CC88 90 03          BCC  LEFT
1040:  CC8A 4C AB CC       JMP  RIGHT
1050:               ;
1060:               ;MOVE LEFT
1070:               ;*********
1080:               ;
1090:  CC8D A0 00   LEFT   LDY  #0
1100:  CC8F B1 61          LDA  (ADDRESS),Y
1110:  CC91 AA             TAX          ;SAVE FIRST BYTE
1120:  CC92 A0 27          LDY  #39
1130:  CC94 B1 61   L2     LDA  (ADDRESS),Y
1140:  CC96 48             PHA          ;MARKER 1
1150:  CC97 8A             TXA          ;GET MARKER 2
1160:  CC98 91 61          STA  (ADDRESS),Y
1170:  CC9A 68             PLA          ;GET MARKER 1
1180:  CC9B AA             TAX          ;GET MARKER 2
1190:  CC9C 88             DEY
1200:  CC9D 10 F5          BPL  L2
```

```
1210:  CC9F 18              CLC
1230:  CCA0 A5 61           LDA  ADDRESS
1240:  CCA2 69 28           ADC  #40      ;NEXT LINE
1250:  CCA4 85 61           STA  ADDRESS
1260:  CCA6 90 02           BCC  L3
1270:  CCA8 E6 62           INC  ADDRESS+1
1280:  CCAA 60       L3     RTS
1290:                ;
1300:                ;MOVE RIGHT
1310:                ;**********
1320:                ;
1330:  CCAB 38       RIGHT  SEC
1340:  CCAC A5 61           LDA  ADDRESS
1350:  CCAE E9 28           SBC  #40
1360:  CCB0 85 61           STA  ADDRESS  ;SUBTRACT 40
1370:  CCB2 B0 02           BCS  R1
1380:  CCB4 C6 62           DEC  ADDRESS+1
1390:  CCB6 A0 28    R1     LDY  #40
1400:  CCB8 B1 61           LDA  (ADDRESS),Y  ;GET LEFT
                                               BYTE
1410:  CCBA AA              TAX
1420:  CCBB A0 01           LDY  #1
1430:  CCBD B1 61    R3     LDA  (ADDRESS),Y
1440:  CCBF 48              PHA          ;MARKER 1
1450:  CCC0 8A              TXA          ;GET MARKER 2
1463:  CCC1 91 61           STA  (ADDRESS),Y
1466:  CCC3 68              PLA          ;MARKER 1
1469:  CCC4 AA              TAX          ;IN MARKER 2
1470:  CCC5 C8              INY
1475:  CCC6 C0 29           CPY  #41
1480:  CCC8 D0 F3           BNE  R3
1490:  CCCA 60              RTS
1500:  CCCB 04 04 04 MULH   .BYTE4,4,4,4,4,4,4,5,5,5,5,5,5
```

```
1510:  CCD8 06 06 06          .BYTE6,6,6,6,6,6,6,7,7,7,7,7,7
1520:  CCE5 00 28 50 MULL     .BYTE$00,$28,$50,$78,$A0,$C8,
                                   $F0
1530:  CCEC 18 40 68          .BYTE$18,$40,$68,$90,$B8,$E0
1540:  CCF2 08 30 58          .BYTE$08,$30,$58,$80,$A8,$D0,
                                   $F8
1550:  CCF9 20 48 70          .BYTE$20,$48,$70,$98,$C0,$E8
]CC00-CCFF
NO ERRORS
```

Here is the BASIC loader:

```
100 FOR I = 52224 TO 52480
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA  32,241,183,138, 74,  8, 32,241,183,224, 25,144
130 DATA   2,162, 24,142,246,  7, 32,241,183,224, 25,144
140 DATA   2,162, 24,142,247,  7,138,174,246,  7,172,247
150 DATA   7, 56,237,246,  7,176,  8, 73,255,174,247,  7
160 DATA 172,246,  7,133,254, 40,  8,144,  3,200,152,170
170 DATA 189,203,204,133, 98,189,229,204,133, 97, 40,  8
180 DATA 144,  8,233,  1,133, 97,176,  2,198, 98,165, 98
190 DATA  41,  3,  9,  4, 40,  8, 32,134,204, 40,  8,165
200 DATA  97,144, 10,105, 39,133, 97,144, 12,230, 98,176
210 DATA   8,233, 39,133, 97,176,  2,198, 98,165, 98, 41
220 DATA   3,  9,216, 40,  8, 32,134,204,198,254, 16,206
230 DATA  40, 96,133, 98,144,  3, 76,171,204,160,  0,177
240 DATA  97,170,160, 39,177, 97, 72,138,145, 97,104,170
250 DATA 136, 16,245, 24,165, 97,105, 40,133, 97,144,  2
260 DATA 230, 98, 96, 56,165, 97,233, 40,133, 97,176,  2
270 DATA 198, 98,160, 40,177, 97,170,160,  1,177, 97, 72
280 DATA 138,145, 97,104,170,200,192, 41,208,243, 96,  4
290 DATA   4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5
300 DATA   6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7
```

```
310 DATA   7,  0, 40, 80,120,160,200,240, 24, 64,104,144
320 DATA 184,224,  8, 48, 88,128,168,208,248, 32, 72,112
330 DATA 152,192,232,  0,  0
340 IF S <> 27098 THEN PRINT "ERROR IN DATA !!" : END
350 PRINT "OK"
```

This program is compatible with the graphics package from Chapter 4, meaning that both machine language programs can be stored in memory and used at the same time. The call is generally done with SYS, as mentioned before, and has similar syntax:

```
SYS 52224,d,s,e
```

d: direction of scrolling (0=left/1=right)
s: starting line and
e: ending line between which the screen will be scrolled

This small BASIC program demonstrates the use of this extension:

```
100 REM ******************
110 REM **              **
120 REM **  SCROLLING   **
130 REM **              **
140 REM ******************
150 REM
200 SR = 52224 : REM SCROLL ADDRESS
210 PRINT CHR$(147) : REM CLEAR SCREEN
220 PRINT:PRINT"    THE SCROLL COMMAND ALLOWS YOU TO"
230 PRINT"SCROLL ONE OR MORE DESIRED SCREEN LINES"
```

```
240 PRINT"           AT THE SAME TIME."
250 PRINT "           SEE HOW IT WORKS"
260 PRINT "              WATCH IT GO   "
270 FOR X=1 TO 5000 : NEXT X
280 FOR X=1 TO 40
290 FOR Y=1 TO 50 : NEXT Y
300 SYS SR,1,6,6
310 NEXT X
320 PRINT:PRINT " THE WHOLE THING NATURALLY MOVES FASTER"
325 PRINT
330 FOR X=1 TO 4000 : NEXT X
340 FOR X=1 TO 200 : SYS SR,1,8,8 : NEXT X
350 PRINT"      AND IN A DIFFERENT DIRECTION"
360 FOR X=1 TO 4000 : NEXT X
370 FOR X=1 TO 400 : SYS SR,0,10,10 : NEXT X
380 PRINT : PRINT"    A WHOLE SCREEN OF TEXT WILL SCROLL"
390 FOR X=1 TO 4000 : NEXT X
400 FOR X=1 TO 200 : SYS SR,0,0,24 : NEXT X
410 PRINT:PRINT
420 PRINT
430 FOR X=1 TO 4000 : NEXT X
440 PRINT:PRINT"         ---- ABACUS SOFTWARE ----"
450 FOR X=1 TO 400:SYS SR,0,17,17:FOR Y=X TO 150: NEXT Y,X
```

As you can see, this command is quite a lot of fun and makes many interesting effects possible.

We have presented the most important fundamentals for programming games. Now it is up to you to implement these.

Chapter 6 : Reference Information

6.1 Program optimization

In the previous chapters we have presented many BASIC programs. But their slowness is very annoying, ruining many otherwise fascinating effects. This section contains some tips on making your BASIC programs run faster.

There are two basic methods of speed improvement:
1) Optimization of the BASIC program itself
2) Replacing slow BASIC routines with assembly language

These suggestions pertain to the first point:

- Avoid REM lines (at least in often-executed loops).
- Avoid too many lines. Often-executed loops, etc., should by placed in as few lines as possible (use the command abbreviations in order to pack as many commands as possible into a line).
- Avoid spaces between or within commands which are not necessary for syntax or other reasons.
- Limit the time spent in computation within time-critical loops by making calculations or portions of calculations outside the loop whenever possible.
- Avoid computations involving numbers directly (constants) within loops; it is better to assign these numbers to variables before the loop and then reference the variables. Otherwise these numbers will have to be converted to binary each time through the loop.
- Avoid subroutine calls (GOSUB) or GOTOs in time-critical loops.

- Construct loops with FOR/NEXT whenever possible--avoid IF.
- Define the most-used variables first (particularly loops indices). An X=0, for example, at the start of the program will suffice if the actual definition of the variable will not be known until later in the program.
- Place program sections belonging together near to each other so that the computer does not have to search far for the destinations of GOTO and GOSUB.
- Place DATA lines together and use as few lines as possible.

Almost all of these suggestions make the program harder to read and understand, and should therefore be used only after it is "up and running."

The second point, the execution of machine language routines, is naturally somewhat difficult, but represents the most sensible and effective measure for speeding up non-arithmetic processes. The machine language program is in DATA lines and must be read by the BASIC program with READ and POKEd into the appropriate place in memory so that it can later be called with SYS (see character editor, sprite editor, and the many examples which rely on assembly language programs).

The most impressive use of this technique is its application to various graphics routines (see graphics package). Here the most time-consuming work is taken away from BASIC and given to an assembly language routine. Two more such routines are erasing the graphics storage and setting the color:

Clear graphics screen:

```
100 FOR I = 51200 TO 51221
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 169, 32,133,254,160,  0,132,253,162, 32,152,145
130 DATA 253,200,208,251,230,254,202,208,246, 96
140 IF S <> 3772 THEN PRINT "ERROR IN DATA !!" : END
150 PRINT "OK"
```

Set color:

```
100 FOR I = 51222 TO 51261
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA  32,241,183,134,151,162,  3,169,  4,133,254,160
130 DATA   0,132,253,132,  2,165,151,145,253,200,196,  2
140 DATA 208,249,230,254,202,240,  3, 16,242, 96,162,232
150 DATA 134,  2,208,235
160 IF S<>5970 THEN PRINT "ERROR IN DATA !!" : END
170 PRINT "OK"
```

These two routines can be built into your program if you have not typed in the graphics package (both functions are found in the graphics package). The syntax of the two commands is:

```
SYS 51200 : REM CLEAR GRAPHICS SCREEN
SYS 51222, 16*PC+BC : REM SET COLOR
```

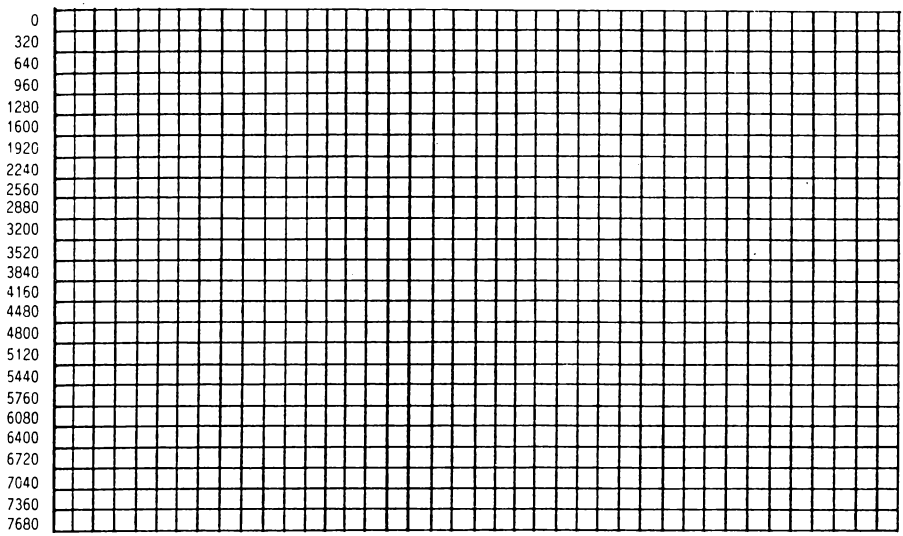
PC: point color
BC: background color

6.2 Organization of graphics storage

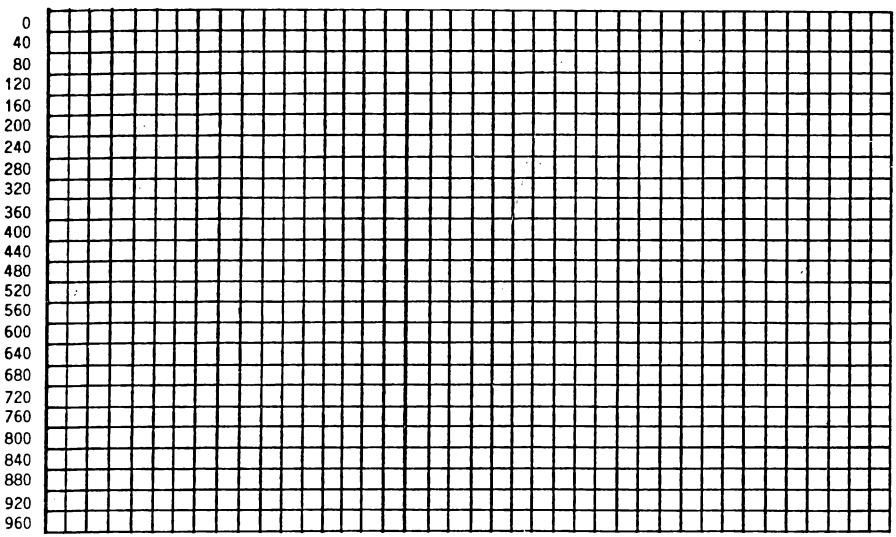
In the following, the construction of the two most important graphics-storage areas is represented: video RAM and bit-mapped graphics memory. The base address must always be added to the relative addresses at the start of each line. This depends on the memory situation. After power-up, the video RAM starts at 1024 ($0400). The color RAM has the same construction as the video RAM and has the base address 55296 = $D800.

In the sketch of the bit-mapped graphics memory, each box must be further divided into 8x8 smaller boxes in order to symbolize the individual pixels. More about the construction can be found in Chapter 3.

6.2.1 Graphics storage



6.2.2 Video RAM



6.3 Color table

The Commodore 64 can display a total of 16 different colors. Each of these colors is assigned a so-called color code which is POKEd into the appropriate register in order to assign that color to something. All 16 colors may be displayed on the screen at once in the text mode. Correspondingly, there is also an ASCII code which makes it possible to set the text color from within a program. The typical graphics characters appear within PRINT statements. The following table contains all of the control possibilities for easy reference.

Key	ASCII Output			Codes		Color
		dec	hex	dec	hex	
<CTRL>	1	144	$90	000	$00	black
<CTRL>	2	005	$05	001	$01	white
<CTRL>	3	028	$1C	002	$02	red
<CTRL>	4	159	$9F	003	$03	cyan
<CTRL>	5	156	$9C	004	$04	purple
<CTRL>	6	030	$1E	005	$05	green
<CTRL>	7	031	$1F	006	$06	blue
<CTRL>	8	158	$9E	007	$07	yellow
<C=>	1	129	$81	008	$08	orange
<C=>	2	149	$95	009	$09	brown
<C=>	3	150	$96	010	$0A	light red
<C=>	4	151	$97	011	$0B	dark grey
<C=>	5	152	$98	012	$0C	medium grey
<C=>	6	153	$99	013	$0D	light green
<C=>	7	154	$9A	014	$0E	light blue
<C=>	8	155	$9B	015	$0F	light grey

6.4 Screen codes

Each character on the screen is assigned a specific ASCII code as well as a screen code. The latter is the code by which the character is stored in the video RAM. If you want to POKE a character directly into this text storage, you would use these values. To obtain the code for a reverse character, simply add 128 to its normal counterpart.

Codes	ASCII	Character	
		Set 1	Set 2
0	64	@	@
1	65	A	a
2	66	B	b
3	67	C	c
4	68	D	d
5	69	E	e
6	70	F	f
7	71	G	g
8	72	H	h
9	73	I	i
10	74	J	j
11	75	K	k
12	76	L	l
13	77	M	m
14	78	N	n
15	79	O	o
16	80	P	p
17	81	Q	q
18	82	R	r
19	83	S	s
20	84	T	t
21	85	U	u
22	86	V	v
23	87	W	w
24	88	X	x
25	89	Y	y
26	90	Z	z
27	91	[	[
28	92	£	£
29	93	]	]
30	94	↑	↑
31	95	←	←

Codes	ASCII	Character	
		Set 1	Set 2
32	32		
33	33	!	!
34	34	"	"
35	35	#	#
36	36	$	$
37	37	%	%
38	38	&	&
39	39	'	'
40	40	(	(
41	41	)	)
42	42	*	*
43	43	+	+
44	44	,	,
45	45	-	-
46	46	.	.
47	47	/	/
48	48	0	0
49	49	1	1
50	50	2	2
51	51	3	3
52	52	4	4
53	53	5	5
54	54	6	6
55	55	7	7
56	56	8	8
57	57	9	9
58	58	:	:
59	59	;	;
60	60	<	<
61	61	=	=
62	62		>
63	63		?

Codes	ASCII	Character	
		Set 1	Set 2
64	96	─	─
65	97	♠	A
66	98	│	B
67	99	─	C
68	100	─	D
69	101	─	E
70	102	─	F
71	103	│	G
72	104	│	H
73	105	╮	I
74	106	╯	J
75	107	╭	K
76	108	╰	L
77	109	╱	M
78	110	╲	N
79	111	╭	O
80	112	╮	P
81	113	●	Q
82	114	─	R
83	115	♥	S
84	116	│	T
85	117	╮	U
86	118	╳	V
87	119	○	W
88	120	♣	X
89	121	│	Y
90	122	♦	Z
91	123	+	+
92	124	▒	▒
93	125	│	│
94	126	π	▚
95	127	◥	▨

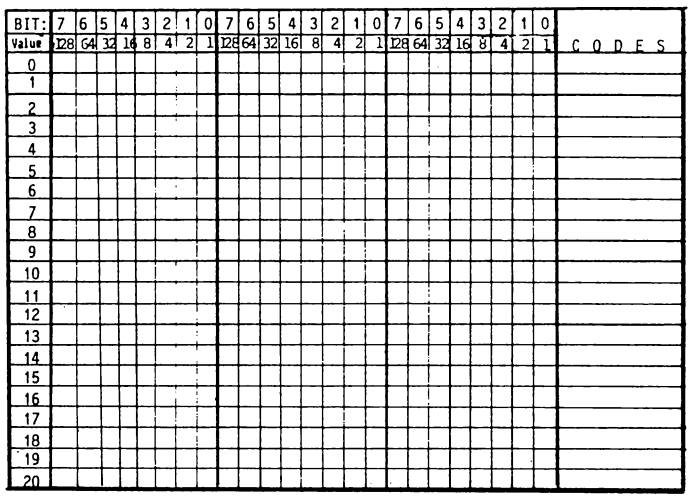
Codes	ASCII	Character	
		Set 1	Set 2
96	160		
97	161	▌	▌
98	162	▬	▬
99	163	▔	▔
100	164	▁	▁
101	165	▏	▏
102	166	▒	▒
103	167	▕	▕
104	168	▔	▔
105	169	◤	▨
106	170	▌	▌
107	171	├	├
108	172	▗	▗
109	173	└	└
110	174	┐	┐
111	175	▂	▂
112	176	┌	┌
113	177	┴	┴
114	178	┬	┬
115	179	┤	┤
116	180	▎	▎
117	181	▊	▊
118	182	▋	▋
119	183	▁	▁
120	184	▃	▃
121	185	▂	▂
122	186	▙	▚
123	187	▖	▖
124	188	▝	▝
125	189	┘	┘
126	190	▘	▘
127	191	▛	▛

6.5 Dec/hex/binary conversion

		%	%	%	%	%	%	%	%	%	%	%	%	%	%	%	%
		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
%0000 0000	0	$00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
%0001 0000	16	$10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
%0010 0000	32	$20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
%0011 0000	48	$30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
%0100 0000	64	$40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
%0101 0000	80	$50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
%0110 0000	96	$60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
%0111 0000	112	$70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
%1000 0000	128	$80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
%1001 0000	144	$90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
%1010 0000	160	$A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
%1011 0000	176	$B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
%1100 0000	192	$C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
%1101 0000	208	$D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
%1110 0000	224	$E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
%1111 0000	240	$F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

6.6 Sprite development sheet

This form make it easier to design sprites. You simply fill in the boxes where points are to be set and then convert each set of eight boxes into a numeric value, treating the boxes as binary digits (a filled box is a 1, else 0). This results in three such values per line, which are the DATA values for the sprite. Multi-color sprites are created by coloring the boxes in pairs in which each pair represents one of 3 colors or is transparent (01, 10, 11, or 00).



6.7 Character development sheet

As with the sprite development sheet, the following table can be used for developing single and multi-color characters.

BIT	7	6	5	4	3	2	1	0	
Value:	128	64	32	16	8	4	2	1	C O D E S
7									
6									
5									
4									
3									
2									
1									
0									

6.8 VIC register overview

Register		Address		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
dec	hex	dec	hex	128	64	32	16	8	4	2	1
00	$00	53248	$D000	Sprite 0 --- x - coordinate (bits 0-7)(0-255)							
01	$01	53249	$D001	Sprite 0 --- y - coordinate (0-255)							
02	$02	53250	$D002	Sprite 1 --- x - coordinate (bits 0-7)(0-255)							
03	$03	53251	$D003	Sprite 1 --- y - coordinate (0-255)							
04	$04	53252	$D004	Sprite 2 --- x - coordinate (bits 0-7)(0-255)							
05	$05	53253	$D005	Sprite 2 --- y - coordinate (0-255)							
06	$06	53254	$D006	Sprite 3 --- x - coordinate (bits 0-7)(0-255)							
07	$07	53255	$D007	Sprite 3 --- y - coordinate (0-255)							
08	$08	53256	$D008	Sprite 4 --- x - coordinate (bits 0-7)(0-255)							
09	$09	53257	$D009	Sprite 4 --- y - coordinate (0-255)							
10	$0A	53258	$D00A	Sprite 5 --- x - coordinate (bits 0-7)(0-255)							
11	$0B	53259	$D00B	Sprite 5 --- y - coordinate (0-255)							
12	$0C	53260	$D00C	Sprite 6 --- x - coordinate (bits 0-7)(0-255)							
13	$0D	53261	$D00D	Sprite 6 --- y - coordinate (0-255)							

Register		Address		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
dec	hex	dec	hex	128	64	32	16	8	4	2	1
14	$0E	53262	$D00E	Sprite 7 --- x - coordinate (bits 0-7)(0-255)							
15	$0F	53263	$D00F	Sprite 7 --- y - coordinate (0-255)							
16	$10	53264	$D010	Sprite 0-7 --- x - coordinates (bit 8)(*256)							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
17	$11	53265	$D011	Raster bit 8	extend color	hi-res graphic	picture off	25/24 lines	y scrolling		
18	$12	53266	$D012	raster line (actual/set)(bits0-7)(0-255)							
19	$13	53267	$D013	lightpen --- x - coordinate (0-255)							
20	$14	53268	$D014	lightpen --- y - coordinate (0-255)							
21	$15	53269	$D015	Sprite on/off							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
22	$16	53270	$D016	not used			multi color	40/38 column	x scrolling		
23	$17	53271	$D017	Sprite --- y - enlargement							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
24	$18	53272	$D018	screen storage addr (*1024)				char. gen. addr (*2048)			not
				bit 13	bit 12	bit 11	bit 10	bit 13	bit 12	bit 11	used
25	$19	53273	$D019	IRQ flag	not used			light pen	sp/sp collis.	sp/bgd collis.	raster line
26	$1A	53274	$D01A	not used				light pen	sp/sp collis.	sp/bgd collis.	raster line
27	$1B	53275	$D01B	Sprite - Background Priority							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
28	$1C	53276	$D01C	Multi-color sprites							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0

Register		Address		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
dec	hex	dec	hex	128	64	32	16	8	4	2	1
29	$1D	53277	$D01D	Sprite --- x - enlargement							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
30	$1E	53278	$D01E	Sprite - Sprite collision							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
31	$1F	53279	$D01F	Sprite - Background collision							
				Sp. 7	Sp. 6	Sp. 5	Sp. 4	Sp. 3	Sp. 2	Sp. 1	Sp. 0
32	$20	53280	$D020	Border color (0-15)							
33	$21	53281	$D021	Background color no. 0 (0-15)							
34	$22	53282	$D022	Background color no. 1 (0-15)							
35	$23	53283	$D023	Background color no. 2 (0-15)							
36	$24	53284	$D024	Background color no. 3 (0-15)							
37	$25	53285	$D025	Common sprite color no. 0 in multi-color (0-15)							
38	$26	53286	$D026	Common sprite color no. 1 in multi-color (0-15)							
39	$27	53287	$D027	Color for sprite no. 0 (0-15)							
40	$28	53288	$D028	Color for sprite no. 1 (0-15)							
41	$29	53289	$D029	Color for sprite no. 2 (0-15)							
42	$2A	53290	$D02A	Color for sprite no. 3 (0-15)							
43	$2B	53291	$D02B	Color for sprite no. 4 (0-15)							

Register	Address	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
dec hex	dec hex	128	64	32	16	8	4	2	1
44 $2C	53292 $D02C	Color for sprite no. 5 (0-15)							
45 $2D	53293 $D02D	Color for sprite no. 6 (0-15)							
46 $2E	53294 $D02E	Color for sprite no. 7 (0-15)							

6.9 Bibliography

Commodore 64

Larsen, Sally Greenwood. Sprite Graphics for the Commodore 64. Micro Text Publications. ISBN: 0-13-838144-5

Compute!'s First Book of Commodore 64. Compute! Books Publications. ISBN: 0-942386-20-5

Krute, Stan. Commodore 64 Graphics & Sound Programming. Tab Books, Inc. ISBN: 0-8306-0140-6

Angerhausen, Michael, Rolf Brueckmann, Lothar Englisch, Klaus Gerits. The Anatomy of the Commodore 64. Grand Rapids: Abacus Software, Inc., 1984.

General

Faux/Pratt. Computational Geometry for Design and Manufacture. Ellis Horwood Limited. ISBN: 0-85312-114-1

Hearn/Baker. Computer Graphics for the IBM Personal Computer. Prentice-Hall, Inc. ISBN: 0-13-164335-5

Encarnacao. Computer Aided Design-Modelling, Systems Engineering, CAD Systems. Springer-Verlag. ISBN: 0-387-10242-6

Brodlie. Mathematical Methods in Computer Graphics and Design. Academic Press. ISBN: 0-12-134880-6

Barnhill/Boehm. Surfaces in Computer-Aided Geometric Design. North-Holland. ISBN: 0-444-86550-0

Myers. Microcomputer Graphics. Addison-Wesley Publishing Co. ISBN: 0-201-05092-7


1st
FIRST PUBLISHING LTD



This was brought to you

from the archives of

http://retro-commodore.eu