Peeks & Pokes COMMODORE 64

A Data Becker Book from First Publishing Limited



1st
FIRST PUBLISHING LTD

TABLE OF CONTENTS

		Page
1.	THE WAY A MICROPROCESSOR WORKS	1
1.1.	A SPIDER WITH 16 LEGS: THE MICROPROCESSOR	1
1.2.	WHAT IS AN OPERATING SYSTEM?	3
1.3.	HOW DOES THE INTERPRETER WORK?	6
1.4.	PEEK, POKE AND OTHER COMMON WORDS	7
1.4.1.	PEEK & POKE	7
1.4.2.	SYS & USR	8
1.4.3.	A SHORT EXCURSION INTO BINARY ARITHMETIC	10
1.5.	COMPUTER DESIGN	15
1.6.	A RESET SWITCH	19
2.	ZERO PAGE	20
2.1.	WHAT IS ZERO PAGE	20
2.2.	POINTER & STACKS	21
3. 3.	MEMORY	24
3.1.	THE MEMORY MAP	24
3.2.	THE MAGICAL BYTE 1	25
3.3.	MEMORY PROTECTION	30
3.4.	FREE MEMORY	33
4.	MASS STORAGE AND PERIPHERALS	35
4.1.	STORAGE OF GRAPHICS, SCREEN CONTENTS, ETC	35
4.2.	HAND MERGING	39
4.3.	DIRECTORIES	42
4.4.	THE PREIPHERALS	45
4.5.	THE STATUS VARIABLE (ST)	47
5.	THE SCREEN	49
5.1.	BLOCK GRAPHICS	49
5.2.	BAR CHARTS	53
5.3.	TYPES OF CHARACTER MODES	55
5.4.	TRANSFERRING THE CHARACTER GENERATOR	60
5.5.	RELOCATING VIDEO RAM	64
5.6.	ASSORTED SCREEN TRICKS	67
6.	HIGH-RESOLUTION GRAPHICS	71
6.1.	THE GRAPHIC MODES	71
6.2.	THE BIT-MAP	72
6.3.	ACTIVATING GRAPHICS	74

6.4.	POINT SETTING	78
6.4.1.	POINT SETTING IN HIGH-RES MODE	78
6.4.2.	POINTS IN MULTICOLOUR MODE	80
6.5.	DRAWING LINES	82
6.6.	DRAWING CIRCLES	84
7.	SPRITES	86
7.1.	MULTICOLOUR SPRITES	86
7.2.	COLLISIONS	88
7.3.	PRIORITIES AND RANGE OF MOVEMENT	90
8.	TONE PRODUCTION	95
8.1.	SID'S WORKINGS	95
8.2.	PROGRAMMING SID	97
9.	THE KEYBOARD	101
9.1.	KEYBOARD DESIGN AND OPERATION	101
9.2.	READING TWO KEYS SIMULTANEOUSLY	103
9.3.	KEYBOARD CUTOFF	106
9.4.	THE REPEAT FUNCTION	108
9.5.	ANOTHER METHOD OF READING THE KEYBOARD	110
10.	JOYSTICKS, LIGHTPEN AND OTHERS	113
10.2.	PADDLES	116
10.3.	THE LIGHT PEN	117
10.4.	OTHER ACCESSORIES	119
11.	THE USER PORT	120
11.1.	INTERFACE CHIPS IN GENERAL	120
11.1.1.	THE SERIAL PORT	120
11.1.2.	THE TIMER	121
11.1.3.	THE PARALLEL PORT	122
11.2.	HOW DO I USE THE USER PORT?	123
11.3.	SAMPLE APPLICATIONS	125
12.	BASIC & THE OPERATING SYSTEM	126
12.1.	PRODUCING BASIC PROGRAM LINES	126
12.2.	LIST PROTECTION	129
12.3.	RENUMBER	131
12.4.	RENEW	133
12.5.	RESTORE	136
12.7.	BASIC EXTENSIONS	140

13.	MACHINE LANGUAGE	142
13.1.	WHAT IS MACHINE LANGUAGE, ANYWAY?	142
13.2.	TIME	144
13.3.	THE HEXADECIMAL SYSTEM	145
13.4.	BINARY ADDITION	147
13.6.	HIGHER ARITHMETIC	151
13.7.	COMPARISONS	152
13.8.	MONITOR COMMANDS	153
13.8.1.	DATA MANIPULATION COMMANDS	153
13.8.2.	JUMP COMMANDS	157
13.8.3.	DATA MOVEMENT	159
13.9.	THE SIMULATOR	161
13.10.	THE FIRST PROGRAM	163
13.11.	THE SECOND STEP: 16-BIT ADDITION	166
13.13.	MULTIPLICATION	168
13.14.	OTHER POSSIBILITIES	171
13.15.	HOW DO SYS-EXTENSIONS WORK?	173
14.	APPENDIX : PROGRAM LISTINGS	174
15.	EXPLANATIONS OF SPECIAL SYMBOLS	183
16.	MEMORY MAP	184
17.	INDEX	192

All rights reserved. No part of this book may be reproduced in any form (by printing, photocopying or any other method), stored in electronic media, or abridged or expanded, without written permission from FIRST Software/Publishing.

IMPORTANT!

The commands, procedures and programs presented in this book are copyright. They are here only for the use of amateurs and teachers, and should not be used commercially.

All commands, technical instructions and programs contained in this book have been carefully worked on by the authors, i.e., written and run under careful supervision. However, mistakes in printing are not totally impossible; beyond that, FIRST Publishing can neither guarantee nor be held legally responsible for the reader's not adhering to the text which follows, or for faulty instructions received. The authors will always appreciate receiving notice of subsequent mistakes.

FOREWORD

You know the problem: You have read through the Commodore 64 User's Manual. Now you want to know more, instead of resigning yourself to the lack of information maybe you're wondering how to use sprite collisions and movement; how to produce high-resolution graphics; or how to scan for two keys simultaneously. Chapter 16 contains a MEMORY MAP. But that presents two questions: 1.) What's a Memory Map and 2.) What can I do with it? If these are your problems, you have the right book in hand.

Let's take a trip together through the 64's memory and operating system—what's that? You don't know what an operating system is? No problem! We'll talk about that, too.

The book is divided into three parts. In Part I, we'll lay the foundation for the tricks to follow. You'll also find explanations of PEEK, POKE and other BASIC commands. Once you understand the section dealing with your computer's programming and functions, a multitude of tricks follow, all of them in BASIC. Please note that no knowledge of machine language is necessary until later, when you feel more comfortable about programming.

Along with tricks, you will find a brief summary at the end of each section, so that you won't have to keep looking up every single definition as you go along.

I guarantee you that all of the tricks and programs work, if you type them in exactly as written.

Regarding machine language: In Part III you'll find a monitor simulation program for the Commodore 64, and a beginner's guide to machine language, to help prepare you for later study.

VIC-20 owners should be reminded that in almost all cases, and in reference to the zero page, the VIC runs the same as the 64 (with some modifications). It may help to take a look at other books from FIRST Publishing.

All that remains is for me to wish you the best in exploring new possibilities of your 64.

Hans Joachim Liesert
Munster, W. Germany May 1984

1. THE WAY A MICROPROCESSOR WORKS.

In the following section you shall learn about the Commodore 64 and how it functions. Incidentally, those who are already knowledgeable in computer technology can skip these pages. The 'hackers' among you will have to excuse my oversimplifications, but I wanted to make things as easy to understand as possible.

1.1. A SPIDER WITH 16 LEGS: THE MICROPROCESSOR.

First some fundamentals. Every microprocessor can address only a certain amount of memory, i.e., it can only address a certain number of memory cells. This number depends on the number of address lines the processor has. Every address line represents a bit (A single switch in the processor is called a bit, from bi nary digi t . A group of 8 bits is called a BYTE) and a bit can have one of two conditions: 0 and 1.

If you've ever seen an integrated circuit you'll know that the chip appears to have legs. The 6510 microprocessor (the "brain" of your '64) has 16 legs. With these legs (total number of address lines), it can address 2 ↑ 16, or 65535 memory cells (64K).

Each of these cells contain 8 bits, or one byte. Once the 6510 has addressed a memory cell with its 16 "legs" (address lines), it then uses 8 other "legs" (data lines) to either read or store 8 bits of data from or to that memory cell.

Microprocessors also have their own number system, known as the hexadecimal system. It has sixteen symbols (0-9 and A-F for 10 to 15) and offers the advantage that a hex (short for hexadecimal) number can always be expressed in 4 bits: F = 15 = 1111 (hexadecimal = decimal = binary).

In the hexadecimal number system one byte requires 2 hex digits. An address requires 4 hex digits.

1.2. WHAT IS AN OPERATING SYSTEM?

When you read through most computer literature, you often run into such words as "operating system", "interpreter" and "interrupt".

The function of the interpreter is as simple as its name:
It takes a program written in BASIC, and simply translates it into a language that the computer can understand.

The '64 and all other microcomputers can, in fact, understand only their own special machine language. The BASIC interpreter (a program in ROM) has the task of reading the program lines in memory and reworking them so the 6510 processor can understand them.

When a BASIC command is entered in direct mode, it is not held in program memory, but is sent to the BASIC input buffer.

This input buffer acts as a sort of "keystroke delivery area", i.e., it checks the operating system (yet another program in ROM), keyboard input, cursor movement and peripherals.

Both the BASIC interpreter and the operating system are machine language programs. Both become active when the computer is turned on, and remain on until another machine language program is called. If a certain machine language instruction occurs, the machine language routine is terminated and the computer is returned to BASIC.

As you know from BASIC programming, a computer in general can only handle one program at a time. Although the BASIC interpreter and the operating system are two separate programs, they perform their assigned tasks simultaneously. They have to. How can they do this?

The simplest way for two programs to run simultaneously is to use a "mutual call". Whenever BASIC finds that its work is finished, it switches the operating system in and out.

This is done so that peripheral devices can be handled, for example. BASIC handles only the passing of the information; the operating system itself signals the appropriate device for action. This has its limits, e.g. keyboard input is accepted only when the operating system is running. But there are times while a program is running that the RUN/STOP should be checked. This is done by an INTERRUPT. Every 1/60 of a second, the processor temporarily interrupts the current machine language program (whether it be the interpreter or the operating system) to check for keyboard input, update the clock and other items. If the computer "senses" that the RUN/STOP key is depressed, it stops the BASIC program. If another key is pressed, that keys value is stored in the keyboard buffer.

Using this technique more than 15 characters per second can be sensed. The microprocessor runs at a speed of approximately 980000 beats per second, with an average of 3 to 4 beats needed to execute a single machine language instruction. The processor can therefore execute thousands of instructions betweens interrupts. After the interrupt checks for keyboard input, the process resumes the machine language program as if the interrupt never happened.

We'll conclude this section with a brief test. A FOR-NEXT loop, such as the one listed below, requires about 46 seconds to execute. If we turn off the interrupt (we'll show you how this works in a later chapter), the program runs one second faster! However, since the interrupt routine controls the internal clock, and you've just turned it off, you cannot use the variable TI to time the program.

Before you key in the short program below, you should first test out the POKE command (line 10) in direct mode. If the cursor disappears, and the keyboard is not being read, you have turned off the interrupt—you can fix it just by using RUN/STOP-RESTORE. Have fun!

```
10 POKE 56334, PEEK(56334) AND 254:REM INTERRUPT OFF
20 FOR I=1TO1000:PRINT I:NEXT I
30 POKE 56334, PEEK(56334)OR21:REM INTERRUPT ON
```

1.3. HOW DOES THE INTERPRETER WORK?

As already mentioned, the BASIC interpreter translates the BASIC commands. Though the user only sees the result (in the form of a running program), it is interesting to see how it works.

Let's begin with the PRINT command. The '64 doesn't store lines of a BASIC program exactly as they are entered at the keyboard. A PRINT command alone would require 5 bytes of memory. Instead all BASIC comands (also called keywords) are converted to unique one-character tokens. Numbers and characters that are not keywords are stored as entered. So for example the statement: PRINT I requires only two bytes - one for the token (PRINT) and one for the variable name(I).

The name for this function of the interpreter is called "tokenizing". As unbeleivable as it may seem, all of this and much more takes place between the time you press the (RETURN)-key and the reappearance of the cursor on the screen.

The second major function of the interpreter is called into play after the RUN command is entered. Because the tokens are unique, the interpreter can easily determine which of its routines are to service the corresponding keyword.

For those of you who have a real interest in the machine's "innards", you might be interested in The Anatomy of the Commodore '64. It includes, among other things a complete ROM listing of the interpreter and operating system, and contains many useful machine language programs.

1.4. PEEK, POKE AND OTHER COMMON WORDS.

Put yourself in the following situation in a computer magazine, you find a super progam that you would like to use. You have typed in the 20K listing and RUN it. Now you're ready to try it out so you RUN it. However it quickly ends with ?ERROR.

You don't want to compare EVERY character in the magazine with what's in memory. Understanding how the program works doesn't help to eliminate the mistake. Then you suddenly see that this stupid POKE command was missing! Now is the time to finally find out how POKE works.

1.4.1. PEEK & POKE.

Let's take the POKE command first.

It's syntax is well known:

POKE address, value

The purpose of this statement is to put value into location address. The address may be between 0 and 65535; value may be between 0 and 255

By POKEing you can write directly to the screen; change the border or character colour; and create many other interesting effects.

Now let's quickly look at the PEEK command.

It's syntax too, is well-known:

PRINT PEEK (address)

The purpose of this statement is to examine the contents of a memory cell located at the address. Again address may be between 0 and 65535.

As the operating system is functioning, it records vital information in memory. So it uses part of memory as a "scratch pad". If we want to examine this information we can do so using PEEK.

It is important to remember that PEEK is a function, and should be used within parentheses (A = PEEK...), or another expession altogether will result.

1.4.2. SYS & USR.

These two commands are of particular interest to the machine language programmer: SYS address and USR(x) . Both call machine language routines.

The value of address is a decimal number and it represents the beginning location of the machine language program.

The last instuction of this machine language program should be RTS which will then return to BASIC.

The USR function works in a similar way, but has an important and useful extension. The first difference lies in the syntax. USR is a function like PEEK and SIN, and must therefore stand as an expression.

Before using the USR function, you must place the starting address of the machine language routine in memory locations 785 and 786. When the interpreter executes a USR function, it looks into these two locations for the start of the program, then it executes that program. When finished, it returns to BASIC.

The most important aspect of the USR function is its ability to transfer data to and from the machine language program. The value in parentheses is put into the floating point accumulator (97-101) for use by the interpreter. The floating point accumulator is an internal math register in which all arithmetical operations of BASIC are performed.

From there, an independent machine language program fetches and works with numbers. At the end of the USR-routine, the value contained in the floating point accumulator is returned to BASIC. Armed with this knowledge, you can send to and receive variables from a machine language routine (using A = USR(x)).

This arrangement makes it possible for the machine language programmer to define particularly fast functions (e.g., utilities, or sort routines). So you see, the USR function is in itself a "Super Command".

1.4.3. A SHORT EXCURSION INTO BINARY ARITHMETIC.

So you can add the commands described above to your growing BASIC knowledge. We'd like to explain the use of AND, OR and NOT. You may have probably not used IF-THEN statements in this form:

```
IF A = 0 AND B = 0 THEN 100
```

These are logical or BOOLEAN expressions. You should understand how the computer handles the logical comparisons. Take a moment to enter the following statements:

```
PRINT(1 = 2)
PRINT(1 = 1)
```

In both, you are asking the computer to print the logical result of the comparison in parenthesis. In the first you are really asking "does 1 = 2?"? Of course not! So the result is FALSE. The '64 represents a FALSE as a zero so a zero is printed. In the second you are really asking "does 1 = 1?". Of course! So the result is TRUE. The '64 represents TRUE as − 1 so this is printed.

How does this relate to the above IF-THEN statement? An IF-THEN statement is not preformed if the result is FALSE. Knowing this you can also write the following statement:

```
50 IF 3 * A THEN 110
60 REM will execute only if A<>0
110 REM
```

Here line number 60 will be preformed next only if the value of A is not zero. If A=0 (and then 3*A is also zero which makes the comparison FALSE) then line number 110 is preformed immediately after line 50.

Now let's look at binary arithmetic a little closer. AND, OR and NOT are known as BOOLEAN OPERATORS. They are logical rules for combining values. The result of these logical combinations are TRUE (−1) or FALSE (0).

Two bits can be combined using these BOOLEAN OPERATORS. The table below shows the results of these combinations.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

You can translate these operators verbally:

a) AND- the result is 1, when both value 1 AND value 2 are 1

b) OR- the result is 1 when either value 1 OR value 2 are 1
Next is the operator NOT. This function simply inverts the a bit.

NOT	0	1
	1	0

Let's move on. Now we're presented with a small problem. BASIC programmer's seldom work with individual bits. Instead they use decimal numbers. To calculate the value of expression (45 AND 123), we do so in the following manner:
1. Convert decimal numbers to the binary number system. This is easier than you might think; you simply divide the decimal number by two (noting the remainder of each division as a bit) until the final result is 0.

Example:
23/2 = 11, remainder 1 ----1
11/2 = 5, remainder 1 ---1
5/2 = 2, remainder 1 --1
2/2 = 1, remainder 0 -0
1/2 = 0, remainder 1 1

decimal 23 = 10111 binary

To do the opposite, see the Commodore '64 Programmer's Reference Guide (the chapter on Machine Language pg. 217).

The numbers 45 and 123 look like this in binary:

45 = 00101101
123 = 01111011

2. Combine binary numbers

In our own example, 45 AND 123 gives us:

```
   00101101
AND 01111011
------------
   00101001
```

3. Convert the result back to decimal

00101001=41

Of course, you can do this easily by keying in PRINT 45 AND 123. But it is helpful to be able to view the bit comparison.

What purpose does this serve? In addition to the logical comparisons, these expressions are often used to set (turn on) or reset (turn off) individual bits. By using an AND operand with the value 254, the bit rightmost bit is always reset (turned off). By using an OR operation with the value 1, the rightmost bit is always set (turned on). Try this with any number!

Now for another mysterious command:

WAIT ADDRESS, X, Y.

The WAIT command pauses until a specified bit pattern appears in a specific memory cell. When the interpreter comes to a WAIT command, it first checks the contents of the given memory cell. This number is combined with the number Y using the EXCLUSIVE OR operand (another BOOLEAN OPERAND).

As the name suggests, XOR (short for EXCLUSIVE OR) is related to the OR operator. This table shows the way it works.

XOR	0	1
0	0	1
1	1	0

XOR- the result is 1 when either value 1 OR value 2 are 1, but NOT both.

The result of the XOR operation is then ADDed to the value X. If the result is 0, then the procedure is repeated again, so the '64 continues to WAIT. If the Y argument is not specified, then the interpreter waits until the contents of the given memory cell AND X are not equal to 0.

```
WAIT address, X, Y is equivalent to
100 IF (PEEK(address) X OR Y) AND X THEN 120
110 GOTO 100
120
```

1.5. COMPUTER DESIGN.

Don't worry, this won't be too technical. This will help you to understand a few very useful tricks, and know some of the '64's inner workings.

Have you wondered how Commodore can advertise the Commodore '64 as having "64K RAM", while BASIC only has 38 K to use.

Well, there actually are 64 kilobytes on hand, but they can't be directly used. The 6510 microprocessor can address 64K - it can address 65536 different memory locations. Utilizing the computer's entire RAM memory is possible; unfortunately, a computer also needs ROMs and memory locations for internal functions. One of these areas is called zero page; the others are the BASIC interpreter, ROM (Read Only Memory), the operating system and the character generator. We'll discuss these in detail later.

The ROM uses 20K. 44K of the 64K is now available. Take another 2K for video RAM and zero page, and 4K of free RAM (Random Access Memory) at 49152. Thus 38K are remaining for BASIC.

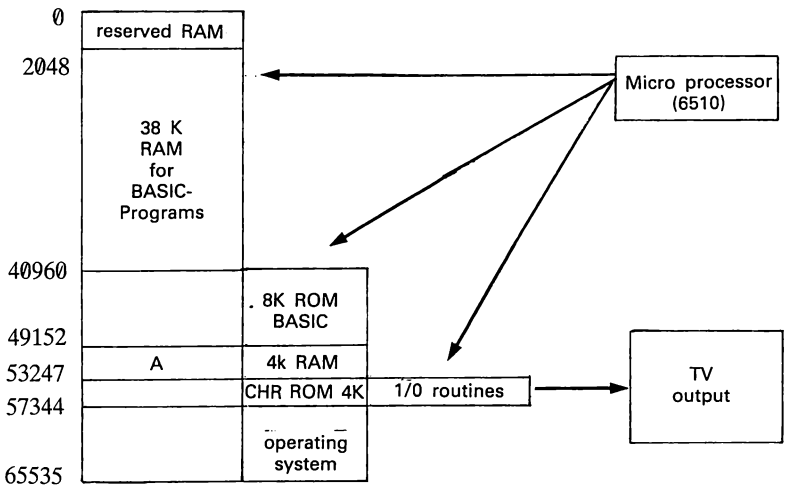


Figure 1 shows a simplified block memory map of the computer. As you can see, ROM and RAM lie next to each other within the same memory range. Memory cell 1 acts as a switch that determines which "bank" of memory is accessed.
If the switch is one way then RAM is accessed. If the switch is another way, ROM is accessed. If you switch off the ROMs, then you can't access BASIC. Therefore the processor no longer finds BASIC, but rather finds empty memory. What usually follows is a locking-up (CRASH) of the computer, and a scream from the user, who has just lost his program.

However, the '64 gives us a little help. It is possible to write into the RAM located in the same range as BASIC (you can do this from BASIC) using POKE or LOAD. However you cannot read these using PEEK or SAVE. Understand that if a POKE, addressing a memory cell in ROM, affects the underlying RAM, then a PEEK with the same address actually reads the ROM in that location.

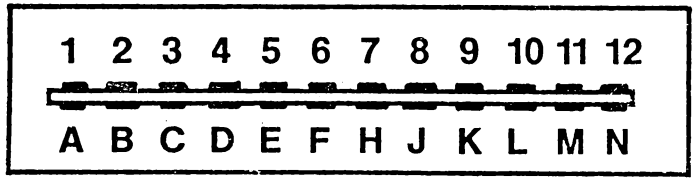
Look for a moment at the 4K RAM in figure 1 labeled A. This area can be used for machine language, and can also be used for data storage in BASIC with PEEK and POKE.

Finally, look at the I/O section. I stands for INPUT, O for OUTPUT. Here are the chips which control the interfaces, the keyboard, the screen, and sound generation. The processor delivers its data here, and the respective chip controls a TV signal, a sound or floppy disk access. Data from the keyboard or peripherals is also processed here.
Moreover, colour RAM is located in this area.

As the block map shows, the memory is "stacked" threefold, with the I/O chips using some of the RAM. Thus, when we call on location 53280 (which alters the border colour POKE 53280, X), that byte is not written into RAM, rather it goes to an I/O chip. This also applies to colour RAM.

The '64 contains 4 I/O chips in all. Two of these you already know of, namely the VIC-II (for graphics and screen), and the SID (for music synthesis). There are also two CIAs (complex interface adapters), which serve the keyboard and the external interfaces such as the user port.

Fig. 2. USER-PORT on the back of the CBM-64



1.6. A RESET SWITCH.

When you experiment with PEEK and POKE, the computer can "lock up" or CRASH. In many cases, just hitting RUN/STOP-
RESTORE keys will fix the problem. Sometimes shutting off the computer seems the only solution. To avoid this, a do-it-yourself RESET SWITCH is recommended. You will need a USER PORT edge connector (e.g., CARDCON 251-12-50-170 by TRW) and a pushbutton switch. The connector costs about £3.50. The user port is very versatile, and this connector can be used for many other projects.

The switch is connected to pins 1 and 3 of the user port (see Figure 2). When the switch is closed, the processor goes through a RESET, i.e., a warm-start. You can also reset in BASIC by typing in SYS 64738. A machine language program will still be there after a reset (if the power has not been switched off), and can be restarted with the proper SYS XXXXX command (if you know the starting address). If you have a program that undoes the NEW command, even BASIC programs can be recovered.

A warning to disk drive owners: When the computer is reset, the disk drive will go through initialization. Before pressing the reset switch, make sure to remove the disk from the drive, to avoid any problems.

2.1. WHAT IS ZERO PAGE.

The Appendix of the Programmer's Reference Guide contains a section about zero page. Understanding zero page, opens up a storehouse of new tricks and programming possibilities—provided you have been introduced to it properly.

The term "zero page" refers to the first 256 bytes of memory. This is the first page of memory. Each page of memory is 256 bytes long. The operating system and the interpreter use zero page as a "scratch pad" to store numbers, conditions and codes.

To guarantee orderly operation of the computer, many locations in zero page contain specific values. Changing these values arbitrarily can affect a particular operation of the '64 and if not done carefully may result in a "lock-up" or crash. Zero page contains 256 memory locations (0-255 decimal, 0-FF hexadecimal) that contain pointers and stacks that the computer uses to keep track of its own operation. They are located in the first 256 memory locations so that the computer can read the contents of these locations very quickly.

2.2. POINTER & STACKS.

A pointer contains an address to a specific place in memory. It is sometimes called a vector. A pointer may point to either information or machine language routines.

An example of a pointer of information is the cursor pointer, which contains the address in screen memory of the current cursor location.

An example of a pointer to a machine language routine is the PRINT vector. We can alter this vector so that, when a PRINT statement is encountered by BASIC, processing is handled by our own special routine which prints on the screen and the printer simultaneously. Thus we can extend the functions of the BASIC interpreter by altering these pointers.

Vectors are stored in a specific format. They occupy two bytes in memory, known as:

1) the LOWBYTE and (least significant part of the address)
2) the HIGHBYTE (most significant part of the address)

The 6510, the heart of the '64, requires that an address be specified with its lowbyte first, and the highbyte following.

To change a pointer to it's decimal address, use the following formula:

DECIMAL ADDRESS = LOWBYTE + (256 * HIGHBYTE)

Since each page of memory in the computer is 256 bytes long, the value of the highbyte is considered the page number. The lowbyte then gives the actual location on the page.

Pointers may also be one-byte in length. These are used to keep track of the size of the keyboard buffer; the current cursor position or the current size of cassette buffer remaining.

A stack is a temporary storage area in memory. Here data is stored by "pushing" it onto the stack. The next data item is also stored by "pushing" it onto the stack, but the other items on the stack are pushed down to make room for the new item.

Conversely, when data is retrieved, it is "popped" off of the stack. As the top item is retrieved, the other data items push themselves towards the top of the stack. Therefore data is always retrieved in a first-in/last-out order.

BASIC subroutines (and machine language subroutines, too) use the stack. When a subroutine is called, the current program location is stored on the stack. Then the work of the subroutine is carried out. When the subroutine has completed its work, and the corresponding RETURN in BASIC (or RTS in machine language) is encountered, the computer retrieves the current program location from the stack, and processing resumes at that location.

SUMMARY: POINTERS.

ADDRESS = LOW BYTE + 256 * HIGH BYTE
LOW BYTE = ADDRESS − HIGH BYTE * 256
HIGH BYTE = INT(ADDRESS/256)

Under normal circumstances, pointers stand in two bytes, always arranged in the sequence LOW/HIGH.

MEMORY

3.1. THE MEMORY MAP.

Chapter 16 contains a complete memory map of the '64. Following the zero page listings, you will find the I/O (Input/Output) section listed. Pay close attention to the deviations from the zero page listing in the Programmer's Reference Guide. If you wish to use zero page memory for data storage, a word of caution is in order: One false POKE may "lock up" the computer!

Don't let this stop you from experimenting. Many tricks have been discovered accidentally, and new techniques come to light if you only look in the right places for them. As far as I know, you can't hurt your computer by POKEing it.
Have fun experimenting!

3.2. THE MAGICAL BYTE 1.

This byte is magical because—as already mentioned—it guides the disposition of memory usage. This is controlled up by bits 0-2 of byte 1. Under normal circumstances, all three bits (0, 1, 2) equal 1. Should one of these bits become 0, the layout of memory is changed.

The BASIC ROMs (40960-49151) can be switched off by changing bit 0 from a 1 to a 0, while BASIC and the operating system can be switched off simultaneously by changing bit 1 from a 1 to a 0. Should both bit 0 and bit 1 read 0, the I/O section is switched off as well; you then have 62K of RAM at your disposal (not including the 2K for zero page and video RAM). Finally, bit 2 controls the character generator (remember: three layers of memory).

This system has a catch to it; If we switch off BASIC and the operating system, the computer locks up. We can still use this 62K but only with machine language.

The character generator location is also controlled by byte 1. You cannot put a program in the character generator locations as the computer needs to know what the characters displayed on the video display must look like and this information is contained in the character generator. To switch the character generator ROM in, bit 2 should be 0, however, if bit 2 reads 0, the '64 will lock up because it can no longer go to the I/O section of memory. Nothing else will operate in the computer except the interrupt routine, which searches for keyboard input. In order to access the 62k we must also eliminate the interrupt with a well-known POKE (see Chapter 1.2.).

Here is a tiny machine language routine ,that will not allow BASIC to function, but will allow you to PEEK & POKE into a full 62K. I have listed the corresponding BASIC program so that the operation of the machine language program will be easily understood:
Machine language loader

```
10 DATA 120, 165, 1, 41, 252, 133, 1, 160, 0, 177, 251, 133, 2, 165,
1, 9, 3, 133, 1, 88, 96
20 DATA 120, 165, 1, 41, 252, 133, 1, 160, 0, 165, 2, 145, 251, 165,
1, 9, 3, 133, 1, 88, 96
30 FOR I = 680 TO 721:READ A:POKE I, A:NEXT I
```

BASIC version of preceding machine language program

```
1 REM ATTENTION! DO NOT RUN UNDER ANY CIRCUMSTANCES
2 REM BASIC VERSION OF MACHINE LANGUAGE PROGRAM 10 POKE
56334, PEEK(56334)AND 254:REM INTERRUPT OFF
20 POKE 1, PEEK(1) AND 252:REM ROM SHUT OFF
30 POKE 2, PEEK(PEEK(251)+256*PEEK(252))
40 POKE 1, PEEK(1)OR 3:REM ROM BACK ON
50 POKE 56334, PEEK(56334)OR1:REM INTERRUPT ON
```

Let's have a closer look at the program: Lines 10 and 50 turn the interrupt off and on. In line 20 bits 0 and 1 in byte 1 have been cleared switching out the BASIC ROM and the I/O, now we have 62 K of usable RAM. In line 30 the desired byte has been read and stored in memory at byte 2, so it will be accessible with a normal BASIC PEEK. The address of the location to be PEEKed is laid out in the pointer at memory locations 251 and 252.

Let's say you want to PEEK address 56000 (usually found under colour RAM). The high byte of the pointer is calculated as follows:

```
HIGH BYTE = INT(56000/256)
```

To preserve the low byte, we simply combine the higher-integer byte with the decimal number 56000:

LOW BYTE = 56000 — HIGH BYTE * 256
To set the pointers, we use the following commands:
POKE 251, LOW BYTE:POKE 252, HIGH BYTE
After typing in SYS 680, you can view the desired byte by means of PRINT PEEK (2).

When we want to write to the RAM under the I/O section, we cannot do so with normal POKE commands. Instead we must switch off ROM here as well.

30a POKE (PEEK(251)+256*PEEK(252)), PEEK(2)
The program to POKE underlying ROM is almost identical to the PEEK program, only line 30 is replaced in the BASIC comparison by line 30a. The principle of its operation is almost the same, only the integer to be POKEd must be placed in byte 2 before executing the machine language routine.
The pointer in locations 251 and 252 still contains the address.

POKE 251, LOW BYTE:POKE 252, HIGH BYTE
Start this program with POKE2, (Integer), then SYS 701 to execute the machine language routine.

The loader program includes both machine language routines.
Line 10 includes the complete PEEK program, line 20 the POKE program. Both lines differ by only 4 bytes.

The routines can be and moved anywhere in memory, i.e. they can be POKEd in wherever you want them in memory (see line 30 of the loader program for the current starting address).
This property is known as relocatability. The length of each routine amounts to 21 bytes.

Now we are in the situation of having 62 K of usable RAM, of which 38 K is for BASIC and variables. The remaining 24 K can be reached by the machine language routine described above.

Because the above routines are not easy to use, another version can be found below. To PEEK an address using the following machine language program:

the syntax is PRINT USR(ADDRESS).

To POKE under the I/O section, use the following combination:

SYS 715, ADDRESS, BYTE
You're probably wondering how such an unusual syntax can be allowed. By using ROM routines to the fullest, you can program in your own commands of the sort described here.

This program is also relocatable; however, you must tell the computer what the USR vector location is, this is done in line 70. Remember locations 785 and 786 point to the location in memory of the machine language routine used by the USR function. Calculate the address as follows:

HIGH BYTE = INT(Address/256) INT(678/256) = 2
LOW BYTE = Address HIGH BYTE * 256 678 — 512 = 166

Machine lanaguage loader for USR function to read and write hidden RAM under ROM.

```
10 DATA 165, 20, 72, 165, 21, 72, 32, 247, 183, 120, 165, 141, 252, 133
20 DATA 1, 160, 0, 177, 20, 168, 165, 1, 9, 3, 133, 1, 88, 104, 133, 21
30 DATA 104, 133, 20, 76, 162, 179, 32, 253, 174, 32, 138, 173, 32, 247
40 DATA 183, 32, 253, 174, 32, 158, 183, 165, 1, 41, 252, 133, 1, 138
50 DATA 160, 0, 145, 20, 165, 1, 9, 3, 133, 1, 96
60 FOR I=678 TO 747: READ A: POKE I, A: NEXT I
70 POKE 785, 166:POKE 786, 2
```

SUMMARY: MEMORY LAYOUT.

Memory location 1 guides memory in general. Bit 0-2 are normally 1. By clearing these bits, the layout of memory is changed. Bit 0 switches off BASIC ROM, BIT 1 handles BASIC and the operating system simultaneously, both bits together also affects the I/O register. Bit 2 guides the character generator.

3.3. MEMORY PROTECTION.

Now that we have released 62 K of memory, let's turn in the opposite direction: We shall reduce the memory available to BASIC to protect certain data from the interpreter. Now would be a good time to ask, what good will that do? Let's suppose that you wanted to write a program that uses 8 different sprites.

You can put four of them in blocks 11, 13, 14 and 15.
However, block 15 is limited to video RAM and BASIC—neither are good places to

write information to because BASIC may overwrite you data. We must shift the beginning of BASIC to create a protected place in memory. To do that, we change the zero page pointers to indicate a different start and end of memory respectively.

To make BASIC begin at memory location 2560, we'll have to change the pointers in locations 43 and 44 by this manner:

```
POKE 43, (2560+1) AND 255
POKE 44, (2560+1)/256
```

The addition of 1 is necessary to direct the pointer to the start of the first BASIC line. The first byte in BASIC must be 0, therefore we POKE the new start of BASIC with a 0:

```
POKE 2560, 0
```

All that remains is to add the CLR statement to set the variable pointers (among other things) to the new situation.
Before CLR, these still pointed to 2048 (the old start of BASIC).

To move the end of BASIC down, we use a similar procedure. We use, of course, locations 55 and 56, and we POKE in a 0. Unfortunately, this method has a great disadvantage to it. When a pointer is changed, the range of the program text in memory is not automatically shifted. Thus these commands must be put in before such commands as LOAD. The simplest way around this is a brief loader program which sets the pointers, then loads the main program. Here is a sample program to type in:

```
10 POKE 43, (2560+1) AND 255
20 POKE 44, (2560+1)/256: POKE 2560, 0: CLR
30 LOAD "MAIN PROGRAM"
```

When this program is run, the '64 truly does the impossible.
The first two lines raise the pointers of the start of BASIC. BASIC will no longer allow this program to be listed (and as far as BASIC is concerned, it no longer exists). In spite of its invisibility, the remaining line is executed.
Only GOTO or similar commands cannot be executed, since in this case, the pointer position (the pointer to the next BASIC line number) can only look after the last line number concerned.

When the LOAD command is preformed from within a BASIC program the newly LOADed program is executed beginning at the first program line. Thus, the newly LOADed program starts automatically.

If the position of the pointers to the start and end of BASIC is changed by hand first, the sprites (or whatever) will not need the troublesome lines mentioned above. Should the program already be typed in and the memory adjustment is not performed, you should save the program to disk or cassette, and return it to memory using the example loader program.

SUMMARY: MEMORY PROTECTION
To raise the start of BASIC:

POKE 43, LOW: POKE 44, HIGH: POKE ADDRESS, 0: CLR

To lower the end of BASIC:

POKE 55, LOW: POKE 56, HIGH: CLR

3.4. FREE MEMORY.

Early versions of the C-64 had a problem with the FRE(0) function. If free memory is less than 32768 bytes, then when PRINT FRE(0) is used, a positive number of free bytes is displayed. If, however, the amount of free memory is greater than 32768 (e.g., after power-up), then PRINT FRE(0) will yield a negative number, and no other information about that greater memory. Why is that?

The FRE function supplies an integer value. However, BASIC's integer values are limited from −32767 to +32767. The interpreter avoids larger numbers (like 38000) by giving a negative integer. To learn the actual amount of free memory, we use PRINT 65538 + FRE(0), when FRE(0) is less than 0.

Now on to another topic. Folks often like to use DATA statements to store a machine language program; or else, they don't like to squander even a byte of memory by making a variable absolute. Likewise, it is possible that, in a program in which all variables have been cleared out (by CLR), one or more absolute variables can be maintained. It is recommended that a free spot in zero page be found into which the DATA can be POKEd. This DATA will not be touched by a NEW or a CLR. Below you will find a list of the free spaces in zero page and below BASIC, suitable for just that purpose (to be used as necessary).

SUMMARY: FREE BYTES in ZERO PAGE & below BASIC
Memory Location in Decimal

2
251-254
678-767
780-783 can be used to protect
820-827 important variables and data
828-1019 and MACHINE LANGUAGE PROGRAMS
1020-1023
2024-2039

PRINT PEEK (152): Number of open files
PRINT PEEK (153): actual input device
PRINT PEEK (154): actual output device
PRINT PEEK (184): actual file
PRINT PEEK (185): actual secondary address
PRINT PEEK (186): actual device
PRINT PEEK (147): last read command
SYS 62255: Close all files
PRINT PEEK (150): cassette motor flag

4.5. THE STATUS VARIABLE (ST).

The status variable (ST) checks the status of Input/Output operations; i.e. it shows LOAD and VERIFY errors on tape, and checks on the use of the serial port. ST is a reserved variable in Commodore BASIC. Location 144 in zero page contains the value of the I/O STATUS Word: ST.

Different bits are displayed within the status register, depending on the type of error. If no errors are encountered, ST=0.

1) The serial bus displays the message DEVICE NOT PRESENT if the value of ST = − 128.

2) Should a write error be encountered, ST=1

3) If a read error, then ST=2.

4) If the end of file be reached, the value of ST is 64 (for both cassette and disk).

5) If the end-of-tape marker is encountered, ST = − 128

6) If a checksum error is encountered, ST = 32
These errors can appear even if things are running properly, and no apparent errors are disclosed. If the error cannot be "ironed out" (through the block controlling LOAD and VERIFY), bit 4 will remain set (=16). If an error in block length appears, ST is set to 4 (too short) or 8 (too long).

If several errors are encountered simultaneously, the bits involved will be added together, then displayed. For example if a checksum error occurs, and the end of tape is reached, ST= −96= −128 + 32. This makes finding disk and tape errors much easier.

SUMMARY:STATUS VARIABLE (ST)

ST=0 No errors
ST=1 Write error
ST=2 Read error
ST=32 Checksum error
ST=−128 Device not present or end of tape
ST=64 End of file

4.1. STORAGE OF GRAPHICS, SCREEN CONTENTS, ETC.

The SAVE routine of the BASIC interpreter is not exactly the easiest to work with. If we want to store anything other than a BASIC program, such as graphic pages, machine language or similar materials, the '64 will refuse to save it unless we specify the starting and ending addresses to the ROM routines. Under these circumstances, we use a trick (primarily with the Datasette) to produce "artificial" files (files and records on tape or diskette that are not BASIC program files).

Time and again we have called upon the services of the zero page. In the range of locations 170-195, we find the pointers and registers for the Input/Output routines.

The pointers for storing the beginning and ending address locations of the I/O routines are very important. We can find the starting address pointer in locations 193 and 194, while the ending address lies in registers 174 and 175. The SAVE command

can be called with SYS 62954; but we must name the file before sending it to the disk or datasette. A good way to do this is to write the name in a REM statement at the beginning of program memory. The location of the filename is found by the operating system via the pointer in bytes 187-188.

Finally, we must set the secondary address, device number, and length of filename. This can be acomplished with the following program:

```
10 REM FILENAME
11 REM SL=LOW BYTE OF START ADDRESS
12 REM SH=HIGH BYTE OF START ADDRESS
13 REM EL=LOW BYTE OF ENDING ADDRESS
14 REM EH=HIGH BYTE OF ENDING ADDRESS
15 REM L=LENGTH OF FILE NAME IN LINE 10 (FIRST LINE OF PROGRAM)
16 D=1: REM D= 1 OR 8 (DEVICE NUMBER)
20 POKE 193, SL:POKE 194, SH:REM START ADDRESS (L/H)
30 POKE 174, EL:POKE 175, EH:REM END ADDRESS (L/H)
40 POKE 187, PEEK(43)+6:POKE 188, PEEK(44)
   :REM FILENAME POINTER
50 POKE 183, L:REM LENGTH OF FILENAME
60 POKE 186, D:POKE 185, 0:REM DRIVE/SECONDARY ADDRESS
70 SYS62954:REM CALL THE SAVE ROUTINE IN ROM
```

Such a routine SAVEs program files, graphic pages and other memory areas, and will allow you to LOAD the files back in with the same starting address as when it was SAVED, using LOAD "FILENAME", 1, 1.

Some explanation about line 40 is needed. These locations (187 & 188) are where the operating system points to the filename in memory, which in our example is in the first program line after the REM, so you add 6 to the start-of-BASIC pointer (43 & 44) to get it into proper position. If PRINT PEEK (43)+6 give a value larger than 255, you will get an ?ILLEGAL QUANTITY ERROR. In this case, use the formula PRINT PEEK (43) + 6 + PEEK (44) * 256 to calculate the new pointer bytes.

Life is considerably easier for disk drive owners. Thanks to the Disk Operating System (DOS) they create files on the disk with a simple OPEN command and then write to disk using PRINT#1, PEEK(Address). To use this procedure intelligently, you must first know the format for storing program files on diskette. Every program consists of a directory entry and program text; the first two bytes at the beginning of that text are stating the starting LOAD address. Under normal circumstances, these are 0 and 8 (0+256*8=2048=start of BASIC). The program text is a set of interpreter codes stored in a series of bytes. Basically every byte from the disk drive is treated as an ASCII code.

When we send a character to the disk drive using PRINT #1, CHR$(x), the number (x) will be stored in the proper position on disk.
Knowing this, here is an easy method of producing program files:

1. Produce the directory entry
   This is done for us by an OPEN command.

2. POKE the starting address
   This is accomplished in the following manner:
   PRINT #1, CHR$ (LOW BYTE): PRINT #1, CHR$ (HIGH BYTE)

3. Text storage
   Text is stored as a series of bytes:
   PRINT#1, CHR$(X)
   Here is a program, which copies the screen contents to disk:

```
10 OPEN 1, 8, 1, "0:SCREEN"
20 PRINT #1, CHR$ (0): PRINT #1, CHR$ (4)
   :REM STARTING ADDRESS POINTER
30 FOR I = 1024 TO 2023
40 PRINT #1, CHR$(PEEK(I))
50 NEXT I:CLOSE 1
```

The program file can be loaded back in with LOAD "SCREEN", 8, 1. The "0:" in line 10 specifies Drive 0, the default on single drives (Dual drives systems contain drives 0 and 1). It is very important that the secondary address 1 be used with the OPEN command when we want to SAVE a file, secondary address 0 and 1 are reserved for LOADing and SAVEing.

Don't forget to CLOSE the file at the end of the routine; otherwise the file will not be saved correctly and be lost!

4.2. HAND MERGING

We now come to a frequent problem. Often part of a program has been split off for testing and debugging, and must later be used with the other program. In most cases, there is no other alternative but to key in the entire program all over again. But by using a MERGE program, you can easily connect the two parts without much hassle. With a few simple commands, you can do this "by hand" in DIRECT MODE.

The problem is, when the newer program is LOADed in, it overwrites the old program in memory. It would be preferable to have the second part LOAD properly behind the orginal program. This would be fine, logic says, if the old program can be protected from the interpreter (this, too, we can do)!

Once the memory area of the first program is protected, we can easily LOAD the new program section, and then "release" the protected memory. It is important that the second program section has higher line numbers than the first. Otherwise, the lines will still merge, but the BASIC interpreter will not execute them properly.

Here, then, is the pro edure:

1. A = PEEK(43):b = PEEK(44):PRINT A, B
This gives us the pointer for the beginning of BASIC (normally 1 and 8). It is imperative that we write these two numbers down as A and B; we'll need them later to re-establish the old configuration.

2. PRINT PEEK(45), PEEK(46)
In locations 45 and 46, you will find the pointer for the start of available memory. Two bytes before this location, (the start of variables), the BASIC program ends.

From these numbers you can calculate the exact size of program text. We can alter the pointers to protect the program in memory. With two POKE commands, we can move the beginning of program text to the end of program text, and protect it from being overwritten when we LOAD the next program.

3. NEW
To correct the remaining pointers to the new situation, and to clear any undesirable remaining variables, we must initialize the remaining memory with NEW. However, the original program is untouched because it is protected.

4. LOAD
Now we LOAD the program to be merged into memory. We can't use the syntax LOAD "NAME", X, 1, or else the original program will be overwritten, regardless of where the pointers are located.
The possibility exists at this moment, of loading and listing the directory without destroying the protected program in memory. If this is done then before the next step, we should again type in NEW, which will clear the directory from memory (we don't want THAT merged).

5. Return memory to orginal configuration.

POKE 43, (# that was A):POKE 44, (# that was B)

POKEing memory locations 43 & 44 with their original values returns us to the original memory configuration. Both programs are now merged with one another, and can now be SAVEd and RUN as one.

SUMMARY: HAND MERGING.

1. PRINT PEEK(43), PEEK(44):REM note the numbers!

2. POKE 43, (PEEK (45) + 256 * PEEK (46) − 2) AND 255
   POKE 44, (PEEK (45) + 256 * PEEK (46) − 2)/255.

3. NEW.

4. LOAD "program to be merged", 8.

5. POKE 43, 1st previously-noted number: POKE 44, 2nd previously-noted number.

4.3. DIRECTORIES.

This section covers the Commodore 1541 disk drive directories.

The first trick we already know from the last section; that is, how to load a directory without losing a program in memory. It can also be useful to load the directory from a program and put aside the listing (e.g., for database programs, etc.). You will find a program of this sort in the 1541 manual, and on the TEST-DEMO disk, under the name "DIR". You can easily use it for your own purposes. It is also interesting to consider the structure of the directory.
Good information about this can be found in the 1541 manual.
Besides this, you should try addressing the directory sometime by using OPEN1, 8, 5, "$" (just to be safe, try it with a scratch disk first), and reading it with GET#1, a$.

You can determine a lot of information about a disk by using the directory.

The easiest form is LOAD"$$", 8: This will load in the disk name and the number of blocks free, nothing more.

If you only want to look at certain entries (e.g., all files beginning with ABC), you can use LOAD"$:ABC*", 8.

A similar procedure is used for file types: Here the command is LOAD"$:*:=TYPE", 8, whereby "TYPE" represents the first character of the file type (PRG, SEQ, REL, USR). The directory will load, but only the programs with the specified file type will appear.

The SCRATCH command functions in much the same way (OPEN 15, 8 15, "S:*=S": CLOSE 15 will scratch (delete) all sequential files).

To conclude this chapter, here is a little trick to save some money. Normally, the disk drive works only with "single-sided" disks, i.e., only one side can be used.

Most single-sided disks, however, can be used on both sides, if you just make a second write-protect notch. A standard hole-punch works best. Just line up the mark using a regular notched disk, put the hole-punch in place, and punch out the new notch. The additional notch can be smaller than the original, and can be a half-moon-shaped notch, as Figure 3 shows.

To be absolutely sure that the new side works properly, you should run it through the CHECK-DISK program on the TEST-DEMO disk. It will write data to all tracks of a formatted disk, and test for read errors, or other obvious problems.

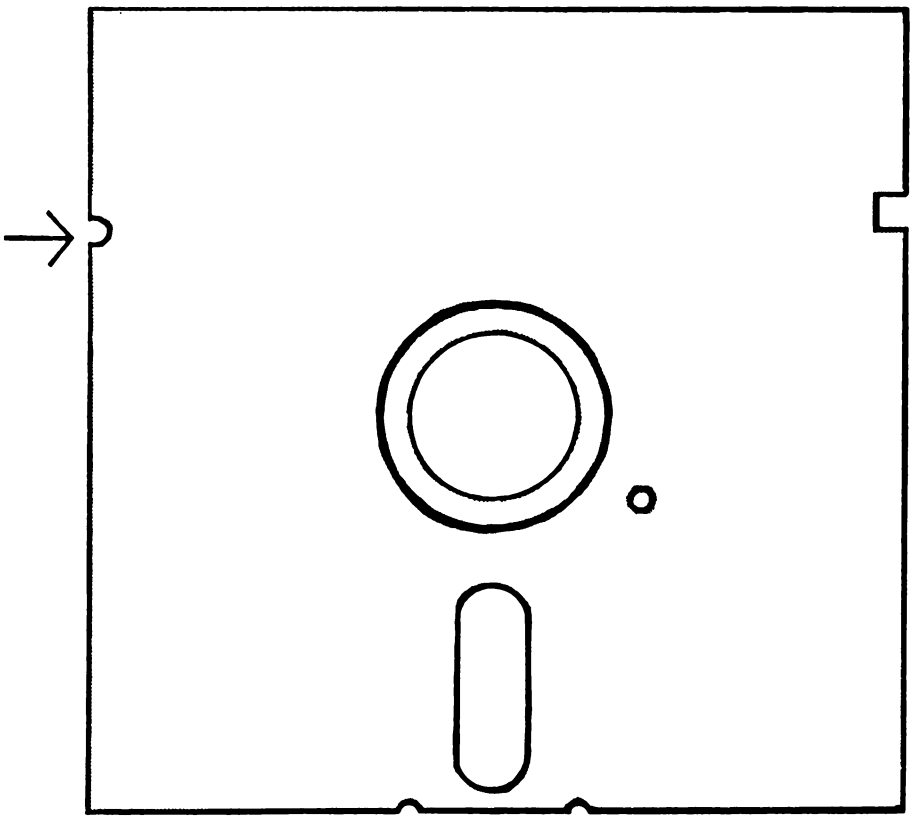
Should any blocks be damaged, the screen will tell you so.
Once the program terminates (it takes a while, sad to say), the directory will be full, and will say "0 BLOCKS FREE".
You can change this by using OPEN 1, 8, 15, "V":CLOSE 1; then you will have 664 blocks available.

SUMMARY:DIRECTORIES

LOAD"$$", 8 loads only the header and number of blocks free.

LOAD"$:ABC*", 8 loads the directory, but only with files beginning with ABC.

LOAD"$:*=TYPE", 8 loads the directory, but only with files of the type predetermined (P, S, R, U). LOAD"$*=S", 8 will load only the sequential files into the directory.



A second write-protect notch

4.4. THE PERIPHERALS.

Now that the "big" tricks are past, here are a couple of PEEKs and POKEs that can help you with input and output of data.

It can be handy to know how many files are open at a given time. A maximum of 10 files can be open at a time. If an eleventh is opened, the computer responds with ?TOO MANY FILES OPEN ERROR. This can be avoided if you PEEK (152) beforehand, to check the number of open files.

It is common knowledge that the transfering of data from the screen to the peripherals can be controlled with CMD. The physical address of devices available can be found in location 154 (1=Datasette, 4=printer, 8=disk drive). By the same token, overlaying CMD with POKE 154, 3 cancels all output devices, but a file remains undisturbed by this. In addition, output can be directed by POKE 154, X ,where X is the device address.

Memory location 153 functions in a similar way. This is where you find the address of the actual input device.
Should the computer be controlling a remote interface (such as a modem), a 2 would be in this location. The normal reading is 0 (keyboard), but if the computer receives data from a peripheral, the corresponding device number is stored in this location.

Memory location 184 contains the number of the actual file used. The secondary file address is in register 185. Here you can alter a printer to print in a certain mode, etc.

The third of this set is location 186; PEEK here to find the number of the device last used. You can use this address from within a program to detect the use of either tape or disk. After LOADing, the program sets which device was used (i.e., from where the program itself was LOADed) into 186, and can act accordingly.

Register 147 could also be of interest. This is where the last read command (whether for tape or disk) is stored; it will mark either a 0 (LOAD) or a 1 (VERIFY). If you don't care about losing files, you can always use SYS 62255 to close all open files abruptly.

This last trick applies to the Datasette owner. Memory location 150 contains the cassette motor flag. When the motor is on, this byte contains 0; otherwise, it will have a value other than 0.

SUMMARY:PERIPHERAL TRICKS

PRINT PEAK (152): NUMBER OF OPEN FILES
PRINT PEEK (153): CURRENT INPUT DEVICE
PRINT PEEK (154): CURRENT OUTPUT DEVICE
PRINT PEEK (184): CURRENT LOGICAL FILE NUMBER
PRINT PEEK (185): CURRENT SECONDARY ADDRESS
PRINT PEEK (186): CURRENT DEVICE NUMBER
PRINT PEEK (147): FLAG (LOAD/VERIFY)
SYS 62255: CLOSE ALL FILES
PRINT PEEK (150): FLAG (CASSETTE MOTOR)

5. THE SCREEN.

Here is an example of normal screen design and its control, because you don't necessarily need high-resolution graphics to program a good picture.

5.1. BLOCK GRAPHICS.

Have you ever looked closely at the '64's graphic characters?
You'll find that they take up one quarter of the total number of screen characters on the keyboard. As it stands now, half of the keyboard is occupied (letters and graphics); if we include reversed characters, then we have covered the remaining 50 percent.

The screen has 25 lines and 40 columns, but the "quarter- square" graphics (block graphic characters, on the left front of the keys) can use 50 × 80 points on screen. The characters accessed by using the Commodore key, can be used while in either upper-case or lower-case mode.

We can use these graphics characters to build charts; by using a subroutine we only need to specify a single number.
Big problem, though: Once a point is already on screen if we POKE a second quarter-square point into the same screen location, the old point is cleared out to make way for the new one. Somehow, the old point must be combined with the new one.

It is possible to copy the entire screen contents into an array, and then place individual points respectively using this array. A subroutine would then transfer the contents of the arrays into the proper screen location. It's simpler to write every possible combination of screen characters and set up your own points in a POKE table.

This table would be created using BASIC. Then, every time the block graphic routine is called, the '64 can look in the POKE table for the desired character. Below you will find a program that can generate block- or quarter-point graphics.

Initialization occurs in the first part, which creates the necessary table. This is fairly extensive, as is the initial clear routine.

The dot-setting and -clearing routine is at line 60000. The set routine is called with GOSUB 60000, the clear with GOSUB 60001.

If the variable L equals 0 (lines 60000-60001), the first section of the table will be used (set points); if L=1 then the clear routine will be used.

The coordinates of the points addressed will be placed in variables X (0-49) and Y (0-79). The colour code to be used is placed in variable CO. Here's the listing:

```
1 PRINT"[CLEAR]":CO=14
10 DIM PD%(1, 1, 1, 15), P2%(15)
20 FOR B=0TO1:FOR X=0TO1:FOR Y=0TO1:FOR Z=0TO15
30 READ PD%(B, X, Y, Z):NEXT Z, Y, X, B
40 FOR I=0 TO 15:READ P2%(I):NEXT I
50 DATA 126, 126, 226, 97, 127, 97, 251, 226, 252, 127, 236, 160, 252, 251, 236, 160
51 DATA 123, 97, 255, 123, 98, 97, 254, 236, 98, 252, 255, 254, 252, 160, 236, 160
52 DATA 124, 226, 124, 255, 225, 236, 225, 226, 254, 251, 255, 254, 160, 251, 236, 160
53 DATA 108, 127, 225, 98, 108, 252, 225, 251, 98, 127, 254, 254, 252, 251, 160, 160
54 DATA 32, 32, 124, 123, 108, 123, 225, 124, 98, 108, 255, 254, 98, 225, 255, 254
55 DATA 32, 126, 124, 32, 108, 126, 225, 226, 108, 127, 124, 225, 127, 251, 226, 251
56 DATA 32, 126, 32, 123, 108, 97, 108, 126, 98, 127, 123, 98, 252, 127, 97, 252
57 DATA 32, 126, 124, 123, 32, 97, 124, 226, 123, 126, 255, 255, 97, 226, 236, 236
58 DATA 32, 126, 124, 123, 108, 97, 225, 226, 98, 127, 255, 254, 252, 251, 236, 160
60000 L=0:GO TO 60010
60001 L=1
60010 Y=49-Y:S=INT(X/2):Z=INT(Y/2):PO=S+40*Z
60020 X1=X-2*S:X2=Y-2*Z:X3=PEEK(1024+PO)
60030 F=0:FOR I=0TO15:IFX3=P2%(I)THEN X3=I:F=1
60040 NEXT I
60050 IF F=0THEN X3=0:IF L=1THEN RETURN
60060 POKE1024+PO, PD%(L, X1, X2, X3):POKE55296+PO, CO:RETURN
```

The character rasters calculate the position in screen memory (PO) and type of character in lines 60010-60020 (X1 for left-right, X2 for up-down). Lines 60030-60040 look for characters already available in screen memory, and place them in the proper table column (X3). If the character is not a graphic character, line 60050 registers that fact.

The set mode simply superimposes the new character over the old, leaving the clear procedure to the discretion of the user.

The crowning glory of this subroutine is line 60060, in which the characters are taken out of the table and POKEd.

The origin of the coordinate system used by the subroutine can be found in the lower left corner (lower left corner =0, 0 right top=49, 79). Using this subtoutine it is very simple to position the block graphic characters anywhere on the screen.

5.2. BAR CHARTS.

It is often helpful to graphically exhibit data values using bar graphics. For example, you might want to display a business's monthly sales using bar charts. Unfortunately, hardly any standard home computer possesses a command to produce bar charts. Below is a subroutine for the producing bar charts. You will find that it works in much the same way as the block graphic subroutine.

Here, too, the graphic characters serve us well. In a horizontal direction, we can allow for 320 different lengths of bars, that is, figuring 40 columns, each capable of a width of 8 dots (40*8=320). For each width (0-8), there is a graphic character present in the '64.

We must calculate the total number of reverse spaces in the length of the bar (8 dots each, or 1 column width), and select the appropriate graphic character. Again, we use a small array. Here's the listing:

```
10 DIMXA$(7):FOR I=0TO7:READ XA$(I):NEXT
20 DATA " ", "[CMDR-G]", "[CMDR-H]", "[CMDR-J]", "[CMDR-K]",
 "[RVS- ON][CMDR-L][RVS- OFF]", "[RVS-ON][CMDR-N][RVS-OFF]",
 "[RVS- ON][CMDR-M][RVS OFF]"
60500 YM=320-V*8:AN$="":IFY<YMTHEN Y=YM
60510 XA=Y/8:G=INT(XA):XA=(XA-G)*8
60520 IFG>0THEN FOR I=1TOG:AN$=AN$+"[RVS ON][SPACE][RVS
      OFF]":NEXT I
60530 AN$=AN$+XA$(XA)
60540 C1=PEEK(214):C2=PEEK(211):C3=PEEK(646)
60550 POKE646, CO:POKE214, X:POKE211, V:SYS58732:PRINT AN$
60560 POKE646, C3:POKE214, C1:POKE211, C2:SYS58732:RETURN
```

The meaning of the first few lines should be clear; that is where the necessary characters are read into the array from DATA statements. Just to be safe, here are the ASCII codes of the characters in line 20 (without RVS ON/RVS OFF):

" "=32, "CMDR-G"=165, "CMDR- H"=180, "CMDR -J"=181,
"CMDR-K"=161, "CMDR-L"=182, "CMDR-N"=170, "CMDR-M"=167

The subroutine is called with GOSUB 60500. Bar length (1-320) is supplied by the variable Y; the column is given in X (0-39). To facilitate the use of graphics, the line from which the bar will start must be specified in V(0-24).
Should the length of the bar exceed line length, the bar will be automatically scaled down by line 60500. Line 60510 computes the total number of reverse spaces in the bar

(G), and the total remaining points (XA). Line 60520 joins the total characters into one string (AN$), while line 60530 connects the character representing the remainder.

The normal PRINT statement won't work here, so the old cursor position is stored in the variable (C1, C2,); see line 60540). The computer uses locations 214 and 211 in zero page as pointers to the row and column for the cursor position. Then the program SYS's to the machine language routine at location 58732 that sets the cursor to the locations given in locations 211 and 214. The next line changes the cursor to the desired colour (=CO) by changing the data in location 646, and then sets up the bar. The last line sets things back to the original cursor position and colour thus ending the subroutine.

Since only standard block graphic characters are used, this routine can be used in almost any application.

5.3. TYPES OF CHARACTER MODES.

In this section, we clear up the question: Where do the little and big As, Bs and Cs etc., come from? The number 1 when stored in location 1024 (video RAM) is the number which causes the letter A to appear on the screen. We don't know how that character is formed at the moment but by examining the memory map we can find the character ROM. The pattern for A (and its siblings from B to the last graphic character) is stored in character ROM, which lies in locations 53248-57343.

All these screen characters require a total of 4K, or 8 bytes per character; each character is enclosed in an 8 × 8 matrix. Respectively, eight pixels consist of 1 byte, while a single point represents a bit. If the bit equals 1, it appears on the screen as a point in the colour stored in the corresponding colour register; if the bit is 0, this point will appear in the matrix of the background colour (named in location 53280).

The screen code number (1 for A) in video RAM has a separate task: It is multiplied (within the VIC II — Video Interface Chip) by 8, and addresses the character generator, which points to the desired pattern. Try this once: Take any character; look up the screen code (not the ASCII code) in the '64 Handbook or the Programmer's Reference Guide, and start up the following mini-program:

```
1 DIMM(7)
5 PRINT"[CLEAR]"
10 INPUT"SCREEN CODE NUMBER (A=1)";C
20 AD=53248+C*8
30 POKE 56334, PEEK(56334)AND254
```

```
40 POKE1, PEEK(1)AND251
50 FOR I=0TO7:M(I)=PEEK(AD+I):NEXT
60 POKE1, PEEK(1)OR4
70 POKE56334, PEEK(56334)OR1
80 FORI=0TO7:FORJ=7TO0STEP-1
90 IF(M(I)AND2 ↑ J)THENPRINT"★";:GO TO110
100 PRINT".";
110 NEXT J:PRINT:NEXT I
120 PRINT"HIT A KEY":POKE198, 0:WAIT198, 1:GET A$:GO TO5
```

This program gives you a "closeup view" of the bit patterns of the upper-case characters. If you want to see lower-case instead, the address in line 20 must be changed from 53248 to 55296. Another possibility: Add 256 to the screen code—for example, C = 1 + 256 = 257.

Don't worry if you don't understand these little program tricks—they will be explained as we go on.

The program is divided into two parts: The first part (lines 5-70) takes the bytes requested from the character generator, and reads them into the array M(I). Next, the interrupt is switched off (line 30) and the character generator ROM is turned on (line 40). The FOR-NEXT loops and PEEKs follow, along with the renewal of the I/O section and the interrupt. Incidentally, AD contains the starting address of the character chosen.

In the second part, the eight bits of the pattern are taken apart. When the ninth bit of M(I) is 1, the expression (M(I)AND2 ↑ n) a 1; this takes the IF command through its paces, branches out on the THEN command, and gives us an asterisk on screen. In the opposite condition, a period appears (line 100). If no comparison is found for the IF-
THEN, or if the task of "expanding" the character is completed, the program ends in a WAIT; if this is unequal to 0, the interpreter waits for a keypress (location 198).

Now, back to the subject at hand! As you gathered from the title, regular character mode is not the only character mode.

The EXTENDED COLOUR MODE is quite similar to normal mode.
The bits of the character patterns equal to 1 are stored in colour RAM as the character colour. The colours of the 0-bits can be different; they adjust to the background colours 0-3 (53281-53284).

Both the highermost bits of the screencodes in video RAM govern which of the above registers are used by the 0-bits.
If you think of both these numbers as one individual number (NUMBER = BIT 7 * 2 + BIT 6), then the number of the register involved is easily figured (EXAMPLE: 10 = 1 * 2 + 0 = 2).

These two bits, however, no longer allow the character generator mode to be set—that remains in 6 bits. In other words, only the first 64 characters of the set (2 ↑ 6 = 64) can be controlled in extended colour mode.

To turn extended colour mode on use POKE 53265, PEEK (53265) OR 64, and off with POKE 53265, PEEK (53265) AND 191.

Multicolour mode is fairly complicated, but with proper use, it can yield some fantastic results.

As you recall, every screen character can have a maximum of two colours; character colour (from colour RAM), and background colour (from the VIC-II registers). On the other hand, multicolour mode allows up to four colours per character. To do this will mean a necessary simplification of the character matrix, though.

This time, colour RAM is responsible for the colour byte; if bit 3 isn't constant (BYTE(AND2 ↑ 3)=0), then it remains the old value. Unfortunately, only colours 0-7 are usable, as bit 3 (which is used for the remaining colours) is not reserved for the multicolour flag.

However, if bit 3 is equal to 1, multicolour capability is on (finally!). The normal 8 × 8 matrix is converted into a 4 × 8 matrix; every two bits of the character generator are now condensed into one dot. If both bits are 0, this dot maintains the background colour. If both bits are 1, the VIC-II makes them the character colour (remember—only colours 0-7). With the other two combinations (01 or 10), the colours are taken from background colour registers 1 & 2, like extended colour mode. You can turn this mode on from BASIC using POKE 53270, PEEK (53270) OR 16, and shut it off with POKE 53270, PEEK (53270) AND 239.

As you can see, the characters in this mode look pretty chaotic and broken up. Shrewd programmers, though, can copy the character generator into RAM, and design their own "rational" multicolour characters.

SUMMARY: Types of Character Modes
Extended background colour mode—on:

POKE 53265, PEEK (53265) OR 64

Extended background colour mode—off:

POKE 53265, PEEK (53265) AND 191

Multicolour mode—on: POKE 53270, PEEK (53270) OR 16

Multicolour mode—off: POKE 53270, PEEK (53270) AND 239

5.4. TRANSFERRING THE CHARACTER GENERATOR.

As you can easily guess, the location of the VIC-II's character generator can be changed (the VIC-II is one of the '64's great features; it supervises the work of the microprocessor, generates the video signal, controls sprites, high-resolution graphics, and operates the internal timer for the system). Location 53272 is of special interest to us here.

Bits 1-3 determine the location of the character generator.
This depends, of course, on other factors which are harder to influence. Therefore, the following instructions pertain to the removal of the normal character set configuration.

The table below shows which bit combination produces the character set address.

000 : 0
001 : 2048
010 : upper case
011 : lower case
100 : 8192
101 : 10240
110 : 12288
111 : 14336

Of particular interest are the combinations 010 and 011; they address the ROM (53248-55296) charater set.

The three bits in location 53272 are best controlled by AND and OR. To change the parameters, you should first use an AND combination with the binary number 11110001 (=241): Bits 1-3 are cancelled by that. Then the combination desired can be put in using OR. Example: To shift the character generator to 2048 (start of BASIC!), the combination 001 must be put in location 53272. The entire command for moving the character generator to 2048 looks like this:

POKE 53272, (PEEK(53272)AND 241) OR 2
With that, the generator is indeed shifted, but we can't do anything with it, because now the screen only displays a mess of dots. We must fix this; with the little program below, the character generator can be plucked out of ROM and copied into RAM. Before doing that we should move the start of BASIC to 6144, to free the first 4 K of BASIC for the character generator. This is done with:

POKE 43, 1:POKE 44, 24:POKE 6144, 0: CLR
A loader (see Chapter 3.3.) must be used in a program to automatically relocate the character generator.

The program below will move the character set to 2048 and copy the characters from character rom into our new character set:

```
10 POKE 56334, PEEK (56334) AND 254:REM INTERRUPT OFF
20 POKE 1, PEEK (1) AND 251: REM CHARACTER ROM ON
30 FOR I = 0 TO 4095: POKE 2048 + I, PEEK (53248+I):NEXT I
40 POKE 1, PEEK (1) OR 4: REM ROM OFF
50 POKE 56334, PEEK (56334) OR 1: REM INTERRUPT ON
```

When you switch over the character generator, the characters all have their old forms, but they are now coming from RAM and can be altered!
Now they can be easily changed with the POKE command.
Screen characters can even be defined like sprites, the only differences being matrix size (8 × 8 as opposed to 21 × 24) and position in memory.

For viewing patterns, you can again use the program from Chapter 5.2., except now the switching off of the interrupts and the alteration of location 1 is unnecessary since the charater set now lies in RAM. Naturally, the starting address in line 30 (now 2048) must also be changed.

It's easy now to POKE in the new screen characters—try it! There are no limits to the programmer's creativity, especially in multicolour mode. By the way, you can switch off the new character set with POKE 53272, (PEEK (53272) AND 241)OR 4 (for upper case; for lower case, use OR 6).

SUMMARY: Transferring the Character Generator
The location of the generator is given by bits 1-3 of the memory location 53272. Before relocating the character set, protection of the memory range selected is necessary (see Chapter 3.3.). After switching off the interrupt and switching over the ROM, the character set can be read and POKEd into RAM.

To turn a particular character set on:
POKE 53272, (PEEK (53272) AND 241) OR X
To turn a particular set off:

POKE 53272, (PEEK (53272) AND 241) OR 4(or OR 6)

5.5. RELOCATING VIDEO RAM.

Like the character generator, the video RAM is also shifted by memory location 53272, only this time, the task is accomplished by bits 4-7.

With these four bits, the VIDEO RAM can be shifted in one-kilobyte increments. Under normal circumstances only bit 4 is constant, selected by the contents is locations 1024-2023.

Here again is a table of bit combinations and the resulting VIDEO RAM locations:

0000 : 0
0001 : 1024
0010 : 2048
0011 : 3072
0100 : ROM
0101 : ROM
0110 : ROM
0111 : ROM
1000 : 8192
1001 : 9216
1010 : 10240
1011 : 11264
1100 : 12288
1101 : 13312
1110 : 14336
1111 : 15360

As you can see, the combinations form an exception to the rule (characterized by the ROM). This is necessary for the VIC-II to reach the character generator in ROM. These ranges are reflected in memory from 4096 to 8191: The ROM for the VIC-II is here, and not at 53248! Again, switching can be done with AND, OR, PEEK and POKE. First, the four highest-value bits must be cleared: AND 15 works best. Then, we must change the binary combination to a decimal number; if we wanted to transfer Video RAM to 15360, then this number would be 15. This number must be

multiplied by 16 (because of byte displacement) to give the value for X.
We use the end result (X) in an OR combination. The complete command is this:

POKE 53272, (PEEK(53272)AND 15)OR X
That number which we'd calculated from binary for multiplication by 16—does it look familiar? RIGHT—it gives where the video RAM will be put in Kilobytes. You won't need those troublesome binary numbers any more—just supply the desired number in K, and use the following command:

POKE 53272, (PEEK (53272)AND 15) OR (K * 16)
You may be badly disappointed when you try this command, though; the screen will be an utter mess of characters, because we haven't told the operating system where the new VIDEO RAM is. The VIC-II now has screen data in a location where the operating system doesn't look for it. Press CLR when in this situation, so that the operating system clears the old screen memory and colour RAM. Prepare for a new problem, though.

Memory location 648 tells the 64 the high byte of the video RAM starting address. We get this by dividing the starting address by 256. For our purposes—15360/256=60. Poke648, 60 will put the world back in order, for us and the 64. Luckily, there is a shortcut for this as well: If the kilobyte number is multiplied by 4, you will get the desired high byte number for location 648 (15=4 = 60).

When you move Video RAM the sprite pointers no longer lie in locations 2040-2047, but behind the new Video RAM. In our own example, they would now lie in the range from 16376-16383.

Remember: Whenever video RAM is shifted, BASIC must be protected and the sprite pointers are moved.

The possibility of page-flipping (defining two separate screen pages, and flipping them back and forth on demand) can be done by the above method. Indeed, the PRINT statement will only work with the page on screen, while the other page must be PEEKed and POKEd. ANOTHER PROBLEM:
Color RAM cannot be shifted, so both pages must use the same colour memory.

SUMMARY: Relocating Video RAM
VIDEO RAM can be directed by bits 4-7 in location 53272.
The number of kilobytes (pages) that the relocation requires is inserted in K:

POKE 53272, (PEEK (53272) AND 15) OR K * 16
POKE 648, K * 4

5.6. ASSORTED SCREEN TRICKS.

Even in normal character mode, there are a few tricks that can make programming, etc., much easier.

You can find out the character colour switched on at the moment by PEEKing location 646; and with POKE 646, colourcode, you can do the same thing as the key combinations CTRL-colour or CMDR-colour. This method has an advantage over direct keyboard colour changes: For example, it's useful when you want to change the colour "accidentally", using RND.

Memory location 647 contains the background colour under the cursor. It doesn't allow changes through POKE 647, x, though.

Regarding alterations: All the memory locations dealing with colour can be altered by bits 4-7. This should not bother us; colour codes only reach from 0 to 15, and hence, only bits 0-3 are reserved. That's why if location 55296 in colour RAM is changed to a 0, it suddenly creates a 32 or other value.

In cells 243 and 244, you will find the pointer for the current position in colour RAM. It is always activated by the operating system, whenever a character is printed. If a 'control' key is pressed (e.g., HOME), the pointer remains in the old position, as such a command is unimportant where colour RAM is concerned. Example: PRINT "[HOME]" leaves the pointer untouched, but PRINT"[HOME]ABC" will move it to the new cursor position.

There is a similar pointer for video RAM. In fact, it is divided in two. Registers 209-210 form the pointer for the current screen line address. All you need do is add the current column (0-39) from register 211 to figure out the address of the byte lying "underneath" the cursor.

The number of the current line (0-24) stands in location 214. By using locations 211 and 214, we can position the cursor on the screen very easily. POKE the column into 211, the line into 214. Of course, that isn't enough; the operating system doesn't "know" that the cursor has been moved. Here's a ROM routine that will take care of the whole business for us. You can call the routine with SYS 58732. Here's the list of commands:

POKE 211, COLUMN:POKE 214, LINE: SYS 58732
This trick has already been employed in Chapter 5.2.

Haven't you wished that the cursor would stay on during an application using GET? Location 204 tells the operating system (or, to be more exact, the interrupt routine) whether the cursor should appear (in this case, PEEK(204)=0), or whether it will be off. If we begin a program that tells the interpreter that location 204 contains a 1, the cursor continues to blink even with a GET statement.

POKE 204, 0 will turn the cursor on "for the duration"; the interrupt doesn't give this a second thought. However, if we merely shut it off with POKE 204, 1, the cursor in the form of a reverse character may stay on the screen.
Another POKE will help: Location 207 if the flag for the last cursor blink, a value of 1 is cursor on, and a 0 value is cursor off. Using POKE 207, 0:POKE 204, 1, we can shut the cursor off as well as the operating system ever could. Here's a tip for your next INPUT statement:

INPUT"TEXT[CRSR-RT][CRSR-RT] Z [CRSR-LEFT]CRSR-LEFT][CRSR-LEFT]";A$

This statement will give you Z as the default for A$, with the cursor over that character. To change the default, just type any other character, which will overwrite Z.

The next POKE applies to reverse mode. Regardless of whether the given string contains the RVS "control character" or not, you can switch on reverse mode with POKE 199, 1. Poke 199, 0 turns it off.

POKE 199, 1:PRINT"This is in reverse characters"
Would you like "control characters" (colours, RVS, etc.) printed in program strings on screen? Well, location 216 stands at the ready. It gives the numbers for such inserts.

As you know, control keys do not execute in insert mode, they just appear as reverse characters. This mode is switched on with POKE 216, X (X must be greater than 0).

POKE 216, 1: PRINT"[HOME][HOME][HOME]"
As a fitting conclusion, we'll play with the operating system one more time. If you have worked with the Datasette before, then you know that the screen goes blank during cassette operation. The VIC-II chip is responsible for this. Bit 4 of location 53265 tells the screen to display itself: You can shut the screen off with:

POKE 53265, PEEK(53265)AND 239,

Turn it on again with:

POKE 53265, PEEK (53265) OR 16.

SUMMARY: Assorted Screen Tricks
Changing character colour: POKE 646, colour code
Current colour code: PRINT PEEK (646)
Current position in colour RAM:PRINT PEEK (243)+256*PEEK(244)
Current position in video RAM: PRINT PEEK (209) + 256 * PEEK (210) + PEEK (211)
Cursor column:PRINT PEEK (211)
Cursor line: PRINT PEEK (214)
Cursor placement:POKE 211, column:POKE 214, line: SYS 58372
Cursor on: POKE 204, 0
Cursor off: POKE 207, 0:POKE 204, 1
INPUT with special cursor: INPUT"TEXT[CRSR RT][CRSR RT]Z[CRSR LEFT]CRSR LEFT][CRSR LEFT]";A$
Reverse on: POKE 199, 1
Reverse off: POKE 199, 0
Insert-mode on: POKE 216, x
Screen off:POKE 53265, PEEK(53265)AND 239
Screen on:POKE 53265, PEEK (53265) OR 16.

6. HIGH-RESOLUTION GRAPHICS.

6.1. THE GRAPHIC MODES.

High-resolution graphics include two different modes, just like normal graphics. Extended colour mode is included in this group (what good can it be for high resolution?). In "normal" mode, we have 320 × 200 points to work with; these 64000 points are stored in 8000 bytes. This 8 K of screen memory is known as the BIT-MAP . Like an actual map, it supplies bits in memory indicating whether there are hills (points) on the terrain (screen) or not. Video RAM supplies the point colour (NOTE: not colour RAM). Every byte in the former screen memory is equipped for a certain range, in which it communicates with characters when in normal mode.
The colours (0-15) of the points are given by the 4 highest-valued bits (represented by 1-bits), and the 4 lowest bits control the background point-colour (0-bits).

In multicolour mode, the dot matrix is more limited; instead of 320 × 200 points, we only have 160 × 200 to work with, Here again, we have 2 bits representing a point, and 8000 bytes for the bit-map, but 4 colours are used for each screen cell. The colours do not just originate in old video RAM, but also from colour RAM and background colour. Register 53281 (for background colour) is used for all 00 combinations. The video RAM reads 01 as the highest 4 bits, and 10 as the lowest. If both bits are 1, the VIC-II gets the colour code from the byte in colour RAM.

6.2. THE BIT-MAP.

On to the location of the bit-map in memory. As with many other things, location 53272 controls the location of the BIT-MAP. Dependent upon the condition of bit 3, the bit-map can start at 8192 (when bit 3=1), or at location 0. The latter is used precious little (when the bit-map is formed in zero page), as you can't use that section since the operating system uses that area.

The Bit-Map is designed like the character generator; the first 8 bytes represent the 8 points of the first square (or character block), and so on. Therefore, you'll find it will allow you to use normal graphic characters.
Multicolour mode is similar, only here, it requires 2 respective bits to produce a double-width point (but you remember that from Chapter 5).

Because the BIT-MAP will be located at 8192 you must move the start of BASIC to protect this memory. As only 8000 bytes are necessary (and not 8192 bytes, which is exactly 8 K), we can protect memory from 16192 on (8000 + 8192). You should therefore be able to properly calculate the pointers (see Chapter 3.3.).

Since normal BASIC memory begins at 2048, we have 6K of memory that is not being used. This free memory can be used for sprites, reserving colours and other screen pages. As you now know, video RAM will change colour memory. Not only that, it will overwrite old text. To avoid this problem, every time high-resolution graphics are turned on, another screen page will be cut off (see Chapter 5.5.).

A few disadvantages: The sprite pointer must now be POKEd in twice, as the sprites can be used in both modes: Once for character mode (2040-2047), and again for the shifted range in high-resolution. Unfortunately, this doesn't work very well in multicolour mode, because colour RAM is also used, and can change the colour addressed in the Video RAM.

6.3. ACTIVATING GRAPHICS.

We must follow three steps to turn on hi-res graphics.

1) Protect the memory range; this is done by a loader program (if all is readied beforehand), which includes the following commands:

```
POKE 43, 65:POKE 44, 63:POKE 16192, 0:CLR
```

These commands for program relocation can also be entered by hand.

2) The graphics can then be switched on within the program. To do that, bit 5 in location 53265 must be set at 1. This done, the VIC-II knows that high-resolution graphics, not characters, will be used. Should multicolour graphics be wanted, the multicolour bit in location 53270 must have a 1 put in (exactly like in character mode). This isn't enough; the location of the bit-map is indicated by bit 3 in location 53272, which must also read 1.

3) Once this is completed, we are rewarded with—a mess on the screen. Finally the bit-map and video RAM must be cleared, a FOR-NEXT loop will work here. Here is the complete sequence of commands:

```
POKE 43, 65:POKE 44, 63:POKE 16192, 0:CLR: REM PROTECT MEMORY
POKE 53265, 59:REM SWITCH ON GRAPHIC MODE
(POKE 53270, 216: REM SWITCH ON MULTICOLOUR MODE)
POKE 53272, 40:REM BIT MAP LOCATION + VIDEO RAM SHIFT TO 2048
FOR I=8192 TO 16191: POKE I, 0:NEXT: REM CANCEL BIT-MAP
FOR I=2048 TO 3047: POKE I, POINT COLOUR * 16 + BACKGROUND
COLOUR: REM SET COLOURS
```

After these POKEs, we find the video RAM has moved to 2048-3047. There is now an additional 5 K at our disposal starting at 3072, for such diverse material as sprites, m/l routines, etc. To end any graphic routine, use the following POKEs:

```
POKE 53265, 155:REM GRAPHIC MODE OFF
(POKE 53270, 8:REM MULTICOLOUR MODE OFF)
POKE 53272, 21:REM UPPER-CASE CHARACTER SET ON
```

You've probably noticed the slow clearing of the bit-map by the BASIC FOR/NEXT loop. Here's a short m/l program that speeds up that task:
Machine language loader to clear BIT-MAP

```
0 FOR I=3600 TO 3659: READ A: POKE I, A: NEXT
1 DATA 169, 32, 133, 252, 169, 0, 133, 251, 162, 31, 160, 0, 145, 251, 136, 208, 251,
    230, 252
2 DATA 202, 208, 246, 160, 64, 145, 251, 136, 16, 251, 169, 8, 133, 252, 165, 2, 162, 3,
    160
3 DATA 0, 145, 251, 136, 208, 251, 230, 252, 202, 208, 246, 160, 232, 145, 251, 136,
    208, 251
4 DATA 141, 0, 11, 96
```

SYS 3600 will start this program. It will clear the bit-map first, then load video RAM (2048-3047) with the point and background colours. Location 2 determines what these colours are. POKE 2, POINT COLOUR * 16 + BACKGROUND COLOUR will keep the program informed of this.

The m/l routine is totally relocatable, i.e., it will work equally well whether placed in the cassette buffer or anywhere else. The starting address is always the byte with which the FOR-NEXT loop in line 0 begins. Try the routine once for speed; it will execute in fractions of a second.

Finally, a little tip: If you store your sprites, graphic pages, colours and hi-res clear programs together with the main program on disk or cassette, life will be considerably simpler for you. Once sprites, machine language and graphics are loaded into the reserved memory (I assume you've made the proper preparations first), set the pointers in 43 and 44 back to the normal start-of-BASIC, and then SAVE as usual. Naturally, if you want to save memory, you can leave the pointer aimed at the higher address when (for instance) colour RAM is not being stored; then LOAD the program (using a loader to set the pointers first) back in with LOAD"NAME", 8, 1, and the graphics, sprites, etc., will be in memory. This is particularly useful for game programs.

SUMMARY: Activating Graphics
The following POKEs will activate high-resolution and multicolour high-res graphics. This time video RAM is placed between 2048-3047, moving the start-of-BASIC to 16192

```
POKE 53265, 59:REM HIGH-RES ON

(POKE 53270, 216: REM MULTICOLOUR ON AS WELL)

POKE 53272, 40:REM BIT MAP & VIDEO RAM MOVED
```

Afterwards, video RAM and the bit-map should be cleared.

Deactivation works like this:

```
POKE 53265, 155:REM GRAPHICS OFF

(POKE 53270, 8:REM MULTICOLOUR OFF)

POKE 53272, 21:REM UPPER-CASE ON
```

6.4. POINT SETTING.

To turn on a specific point on a graphic "page", you can first draw the picture out on millimeter-size graph paper, then transform the "little boxes" into byte contents. If you come up with a vision (built of blood, sweat, tears and fits of madness), please share your work with me. Below are two routines which can make such programming considerably easier.

6.4.1. POINT SETTING IN HIGH-RES MODE.

The subroutine listed below works on the same principle as the block graphic routine in Chapter 5.1. This time, though, no special graphic characters must be POKEd in, nor is a table needed to find coordinates:

```
61000 REM SETTING AND CLEARING POINTS IN HIGH-RES
61010 Y=199-Y:IFY<0ORY>199 THEN RETURN
61020 IFX<0ORX>319 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8):AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2&(7-(XAND7)):CA=2048+X1+40*X2
61050 POKECA, (PEEK(CA)AND15)OR16*CO
61060 IFL=1THENPOKEAD, PEEK(AD)AND(255-X3): RETURN
61070 POKEAD, PEEK(AD)ORX3:RETURN
```

The subroutine is called with GOSUB 61000. Coordinates X(0-319) and Y(0-199) give the locations of the points. Should X and Y not be within allowable range, the routine will end at 61010 or 61020, respectively. This means it's possible to draw lines that appear to go beyond the screen border.

Line 61030 computes the address to be changed within the bit-map. INT (X/8) is put in to represent the columns for colour RAM. This value is multiplied by 8, so that one cell is represented in colour RAM by 8 bytes of the bit-map. INT (Y/8) does the same for the lines in colour RAM; to get the correct address for the bit-map, this value (number of lines) multiplied by 8 will give us the number of points per line possible (320). Also, Y AND 7 represents the line within a "colour square".

The variable X3 gives the value (which must combine with the bits already set) which controls the setting or clearing of the point desired. Finally, AD contains the POKE address for the point, while CA contains the colour. Line 61050 POKEs the colour from CO (0-15) into the four highest-value bits of the proper colour memory cell. Please note that with the point's colour change, the entire square involved changes colour as well.

If variable L equals 1, this means that the routine will clear the point given. In this case, line 61060 will return to the main program, otherwise the "setting" section in the last line is called. A typical call routine could look like this:

```
X=100:Y=25:REM COORDINATES SET
CO=2:L=0:REM COLOUR=RED, MODE=SET
GOSUB 61000:REM CALL SUBROUTINE
```

6.4.2. POINTS IN MULTICOLOUR MODE.

Two bits per point must be used in multicolour mode in order to produce the different colour combinations. This method is used because the point routine must be called TWICE for each multicolour point. The first bit would simply be doubled for the X-coordinate, and the doubled coordinate would be increased by 1 for the second bit. Since these two bits lie in the same byte in every case, the twofold calculation of POKE addresses can be done without. Here is the program for point setting in multi-colour mode:

```
61000 REM MULTICOLOUR POINT SETTING
61010 Y=199-Y:IFY<0ORY>199 THEN RETURN
61020 X=2*X:IFX<0ORX>318 THEN RETURN
61030 X1=INT(X/8):X2=INT(Y/8):AD=8192+8*X1+320*X2+(YAND7)
61040 X3=2↑(7-(XAND7)):X4=2↑(7-((X+1)AND7))
60150 POKEAD, PEEK(AD)AND(255-(X3+X4))
60160 POKEAD, PEEK(AD)OR(COAND1)*X3+(COAND2)/2*X4):RETURN
```

Call with GOSUB 61000: The coordinates are newly placed in X(0-159) and Y(0-199). The colour and set/clear modes no longer need be given, since the colour number (0-3) will control this. The colour number is given for the bit combination in CO. If a point is erased, CO simply becomes 0. The colours through which the bit combinations should be addressed must be loaded beforehand by POKEing in the specified register (since the video RAM takes this over from the clearing routine). Aside from lines 61010-61030, normal high-res mode is not that different from multicolour mode.

Line 61040 calculates the masking combination for both bits (X3 and X4), then the two next bits are unset (line 61050).
Finally, the bit combination is blended with the corresponding byte in line 61060. CO AND 1 give the low byte of the combination, and (CO AND 2)/2 the high byte. If the bit is 0, then the complete product will equal 0.
Result: The bit concerned will remain in its old position if 0 or an OR combination; otherwise if combined with 1, the bit will be 1 in any case. Here is an example for a typical subroutine call:

```
X=100:Y=50:REM COORDINATES
CO=2:REM COLOUR OF HIGHERMOST BITS HELD IN VIDEO RAM
GOSUB 61000: REM CALL THE SUBROUTINE
```

6.5. DRAWING LINES.

The following subroutine is equally suited for either graphic mode. It uses the point-setting routine from Chapter 6.4. as a subprogram, from which it limits the range for individual points per line:

```
61100 REM DRAW LINES
61110 IFABS(XE-XA)<ABS(YE-YA)THEN61160
61120 SP=(YE-YA)/ABS(XE-XA+1E-20):YK=YA
61130 FORXX=XATOXESTEPSGN(XE-XA)
61140 YK=YK+SP:Y=INT(YK+.5):X=XX:GOSUB61000
61150 NEXTXX:RETURN
61160 SP=(XE-XA)/ABS(YE-YA+1E-20):XK=XA
61170 FORXX=YATOYESTEPSGN(YE-YA)
61180 XK=XK+SP:X=INT(XK+.5):Y=XX:GOSUB61000
61190 NEXTXX:RETURN
```

Call with GOSUB 61100. The starting coordinates are given in XA and YA, the end coordinates in XE and YE. When using point setting routines, the colour (CO) and mode (clear/set in L) for high-res (or, for multicolour, just the colour number) must be given as well.

The algorithm is actually quite simple. First it must be established whether the distance between X coordinates is less than the distance between Y coordinates (line 61110). If the coordinates were flipped around, the program would still work fine. If all looks good so far, on to the next step: Let's say the X distance is greater than the Y distance. That is, the more points per line in the Y-direction, the less our chances of getting a nice, even, sloping line (instead, we get a rough zig-zag line). This can be fixed nicely with a FOR-NEXT loop (line 61130) which "flip-flops" between XA and XE and calculates the proper coordinates. If the X-distance is less than the Y-distance, more points would have to be put in the Y-direction. To reverse the above procedure, we'd just start at the Y-direction and calculate the X-value.

This calculation is also very simple: The step size for the loop (SP) is calculated. It gives the distance of two successive points in the Y-direction. With every pass of the loop, the variable YK is raised by the step size. This value is rounded off (INT(YK+.5)) and then given as the Y-value in the point setting routine (line 61140).

If the coordinates of the set points don't lie in the allowable range, the routine will intercept these.
Unfortunately, this subroutine is not very fast, but for simple applications (such as plotting functions) it's more than adequate. If you want speedy routines, you should

replace this subroutine with a suitable graphics utility program, such as GRAPHICS-64 from First Software or the software from the graphics book which is available from First Publishing. Both include commands for work with high-res graphics, sprites, etc.

6.6. DRAWING CIRCLES.

The circle is one of the most frequently-used graphic symbols. It's not merely designed from a set of lines—so a special subroutine is written below. It's very slow, but in my opinion, it's better to draw a circle slowly than not to draw it at all. Like the line-drawing routine in the last chapter, this one will work in both graphic modes:

```
61200 REM CIRCLE DRAWING
61210 FORXX=0TOR*0.7
61220 YY=INT(SQR(1-(XX/R)↑2)*R)
61230 X=XA+XX:Y=YA+YY:GOSUB61000
61240 X=XA+XX:Y=YA-YY:GOSUB61000
61250 X=XA-XX:Y=YA-YY:GOSUB61000
61260 X=XA-XX:Y=YA+YY:GOSUB61000
61270 X=XA+YY:Y=YA+XX:GOSUB61000
61280 X=XA+YY:Y=YA-XX:GOSUB61000
61290 X=XA-YY:Y=YA-XX:GOSUB61000
61300 X=XA-YY:Y=YA+XX:GOSUB61000
61310 NEXTXX:RETURN
```

The call command is GOSUB 61200. The variables used are: X and Y for the coordinates of the circle's centre point; R for the radius (measured by the number of points given); and CO and L for "normal" high-resolution, or just CO as the colour number for multicolour mode.

Once you see the circle appearing on the screen, you will probably follow the basis of the routine which is the circle equation X↑2+Y↑2=1 (or in converted form, Y=SQR(1-X↑2)).

In the loop, the radius is calculated from the centrepoint of the circle, it checks the X-value with the corresponding Y-value every 45 degrees. If you lengthen the loop to 90-degree points, then you'll run into the same problem as with the line algorithm ("ragged" lines). The steeper the fall of the circle, the more often points must be set in the Y-direction, and the higher the X-value.

The formula above refers to the standard circle (with a radius of 1), the coordinates in the calculation of line 61200 must first be divided by R, and again multiplied by the radius. That's all: Have fun with circles!!

7. SPRITES.

Sprites are probably the best-known feature of the 64. None of its other graphics capabilities is quite so versatile.
The following sections give you some of the possibilities that sprites offer.

7.1. MULTICOLOUR SPRITES.

Besides normal high-resolution graphics, the VIC-II also lets you program multicolour sprites. Multicolour mode is switched on by setting certain bits for each sprite number in VIC register 28 (location 53276). For example, to define sprite 6 in multicolour mode, use the following:

```
POKE 53276, PEEK (53276) OR (2 ↑ 6)
```

You can set the bit back to 0 using

```
POKE 53276, PEEK (53276) AND (255-2 ↑ 6).
```

Now you can switch the sprite in and out of multicolour operation. In multicolour mode 2 bits represent one point on the matrix, and that leaves us only a 12 × 21 matrix. Now that this mode is running, the only thing missing is for you to supply the information as to which bit combinations and colours will be used. If both bits are 0, the point involved will be transparent, i.e., that pair of dots will appear as the background colour. If the lowest bit of the two is 1, the colour will come from VIC-II registers 37 and 38 (53285 and 53286). Bit 2 decides which of the two is used; if it reads 0, register 37 is used, otherwise it will be 38. If the combination is 10, the colour information will be displayed from the normal sprite colour register. These colours will work with any other sprite, not just with multi colour sprites. They are only in the ranges 0-7, in the same registers for all sprites. Multicolour sprites are defined exactly as normal ones, except the order of points is changed. Screen coordinates also remain constant. The big advantage is that you can mix different sprite and graphic modes. So, high-res and multicolour sprites can lead a peaceful coexistence on the same screen.

A few limitations: During disk operation, you should respect the fact that the sprites should be shut off (POKE 53269, 0), as the VIC-II also regulates the timer for the I/O operations. The more sprites on the screen, the longer the disk access takes . . . this can be disturbing.

SUMMARY: Multicolour Sprites
Multicolour mode on: Bit specified in reg. 28 (53276) set at 1.

Colors in reg. 37 (01) and 38 (11), also the case for normal sprite colour registers (if 10).

Different modes of sprites and graphics can be mixed.

7.2. COLLISIONS.

The VIC-II senses every contact a sprite makes, either with another sprite, or with a point onscreen. Following a collision between 2 or more sprites, this contact appears in register 30 (53278). The number of the sprites involved will show in the corresponding bits of this register. So, you can establish whether sprite N was involved or not by using PRINT PEEK (53278) AND 2 ↑ N. If a collision has occured the value will be 1 otherwise the value is 0. The collision is registered when two points actually touch (2 sprites in the same range, for instance), but not points and blank space. The bits remain set at "collision" status until you clear them with POKE 53278, 0. So the bits will register a collision, even though the sprites withdrew from one another long ago. Therefore, I suggest that you clear all the bits in this register every time you read them. To check quickly to see whether the numbers have been reset after a collision, the PEEK command is very useful.

The control of sprite/background collisions operates in the same manner. The specified bits will be changed to 1 when one of the sprites involved reads one in the bit map or the character generator. In other words, every contact between a sprite and a character or graphic point will be noted in register 31 (53279).

There is a short game listed in Chapter 14 which illustrates the programming logic behind sprite collision control, handled in the form of a simple Road Race. Using the Z (=left) and / (=right) keys, the object of the game is to avoid the border. Every time your car hits the wall or another car, the collision register is so marked, causing a CRASH. Obviously, you can omit the REMs; they only slow the program down anyway.

SUMMARY: Collisions
Contact between sprites, or a sprite and background characters, is stored and shown in registers 30 (53278) and 31 (53279) of the VIC-II chip. The bits will remain set after collision unless otherwise cleared.

The program "ROAD RACE" can be found in Chapter 14.

7.3. PRIORITIES AND RANGE OF MOVEMENT.

Did you know that there are different priorities between screen characters, graphics and sprites? In most cases, the sprites will stand in front of the actual screen contents, but sometimes it's preferable to have the character stay in front of the sprite instead (say, when an airplane should fly behind a house). Once again, the VIC-II has a register for this.

That register is in address 53275 (VIC-II + 27). If a bit here is set at 1, the corresponding sprite will travel behind the screen character. When all these bits are 0 (normal setting), the sprite has higher priority than the character and will travel in front of the screen character.

Sprites have display priorities, the sprite with the lower number will always be "illustrated" in front of the sprite with a higher number;but if the foremost sprite has a lower priority than the screen characters used, it will still appear in front of the other sprites, but it will appear under text, even if it takes precedence over the other sprites. This allows for some simple optical illusions.

Have you ever tried producing graphics that could be connected to sprites? If so, then you've ascertained that the coordinates of sprites and graphics don't match. The field of movement of sprites is much more defined, to where they can virtually walk off the screen. You can move such a "graphic-within-a-sprite", by using the coordinates 24 and 50 as an offset (starting coordinates from the upper left hand corner). These two values represent the correction factor to which you must add to the graphic character to position it properly on the sprite. The middle of the screen can be reached with the values 160+24 and 100+50 (all based on the upper left corner of the sprite).

SUMMARY: Priorities and Range of Movement
Sprite priority (in front of/behind characters) is regulated by the corresponding bit in VIC-register 27 (53275). With proper setting of this bit, the sprite can pass behind characters.

Correction factor for sprites as opposed to graphic coordinates: 24(X) and 50 (Y).

7.4. IDEAS FOR SPRITE PROGRAMMING.

Many game programs use animation, e.g., maybe a lifelike little sprite man runs after his sprite lady. On first glance, it looks as if the arms and legs are really moving, but if you look closer, you'll see that in fact there are only two or three different positions for the legs. This should make the principle behind animation clear. In this

case, two sprites are used as two separate blocks which are switched back and forth during movement. In one block, the sprite is defined with arms and legs closed, while the other block has them "out". Both pictures are alternated by the sprite data pointers (2040-2047), giving the illusion of a moving figure. In reality, it is only a set of constantly changing "copies" of the sprite; it's that simple. A natural prerequisite is that we have enough memory available for the different pictures—there again, you should shift the start-of-BASIC into a higher range of memory. Even if you use high-resolution graphics you should have enough memory in any case.

High-res sprites can be used as tiny graphic screens in character mode to produce interesting effects. Suppose you'd like to produce graphs for any function in high-res, while simultaneously putting in a few comments. One possibility would be to reproduce the characters artificially, reading the dot patterns from the character generator, and setting up the appropriate points for the necessary characters. That's a roundabout, or even backwards, way of doing it, since the character set is already redefined to produce the graphs. There are sprites available, though. Take four of them, place them in the position you want them on screen, and build all the characters need using the sprite matrices. This is done for us by the routine below. I have skipped any detailed explanation this time, since the program design and high- resolution dot-setting have already been discussed:

```
10 FORI=704TO767:POKEI, 0:NEXT
20 FORI=832TO1023:POKEI, 0:NEXT
30 POKE2040, 11:POKE2041, 13:POKE2042, 14:POKE2043, 15
40 V=53248:POKEV, 100:POKEV+1, 100:POKEV+2, 148:POKEV+3, 100
50 POKEV+4, 100:POKEV+5, 142:POKEV+6, 148:POKEV+7, 142
60 FORI=39TO42:POKEV+I, 1:NEXTI:POKEV+21, 15:POKEV+23, 15:
     POKEV+29, 15
100 INPUT"#HOME–X-COORDINATE";X:INPUT"Y-COORDINATE";Y
110 GOSUB 62000:GOTO100
62000 Y=41-Y:IFY⟨0ORY⟩41 THEN RETURN
62010 IFX⟨0ORX⟩47 THEN RETURN
62020 BX=INT(X/24):BY=INT(Y/21):IFBX=0ANDBY=0THENBA=704:
     GOTO62040
62030 BA=768+BX*64+BY*128
62040 BX=X-24*BX:BY=Y-21*BY
62050 X1=INT(BX/8):X2=7-(BXAND7):X3=BY*3
62060 AD=BA+X3+X1
62070 IFL=1THENPOKEAD, PEEK(AD)AND(255-2&X2):RETURN
62080 POKEAD, PEEK(AD)OR(2&X2):RETURN
```

Call the routine with GOSUB 62000. Mode instructions (set/clear) are given in L so that the high-res graphics can be given in X and Y coordinates. It will use sprites 0-3

and blocks 11, 13, 14 and 15. In the version given, the sprites are enlarged in both directions; if you prefer to have them normal-sized, you'll have to correct the positions (lines 30 and 40) and alter line 60.

Between the four sprites, 48 × 42 points can be set. Since this doesn't apply to multicolour sprites, all points will have to be the same colour (this is determined in line 50).

If you like, the routine can be transferred from sprite graphics to line characters; it's quite easy. All other functions (like priority, collisions, etc.) can usually also be set up for these four sprites.

With that, the chapter on sprite programming draws to a close. Don't hold back from experimenting with the VIC registers, though. There are many possibilities for sprites, just waiting to be discovered!

8. TONE PRODUCTION.

The SID (Sound Interface Device) is to sound what the VIC-II is to graphics. The possibilities of SID are extensive, below is a solid foundation for learning sound programming.

8.1. SID'S WORKINGS.

The thesis for this section is to show you what happens in the computer when a tone is produced.

If a certain start-bit is set at 1, SID first looks for the frequency of the sound, then produces the appropriate oscillation. This is put through a sort of "electronic food processor" called the waveform generator; it takes the tone and gives it a tone shape using a programmed waveform (triangle, sawtooth, pulse, or white noise).

Then, SID designs an envelope (which gives the volume of the tone at various stages of production) based upon what is programmed into the ADSR. The four capabilities of the ADSR are:

1) The attack, the speed at which the tone first swells;

2) decay, the speed at which the tone decreases;

3) sustain, or how long the tone hangs on; and

4) release, when the tone is switched off by resetting the start-bit to 0.

An echo, among other sounds, can be reproduced using these materials. In addition, the pulse wave can be altered by changing the pulse width, which influences the tone shape.

Other possibilities not mentioned here are ring modulation (producing a tone independent of the tone shape available) and filtering (filtering out different frequency ranges to change tone shape)

8.2. PROGRAMMING SID.

When doing a sound program one thing should always be at the beginning; the volume, which is POKEd into the 4 lowest-value bits of register 24 (54296). Any attempt to read this or any other sound register (0-24) using PEEK will meet with failure: Basically, a special construction lets these bytes be written to, but not read. PEEKing will only give you senseless results. It's the exact reverse with registers 25-28: Here you can only read them, and POKEing is useless.

Back to the music. Since the 4 highest-value bits in the volume register are usually set at 0, we can easily POKE in the volume we want. POKE 54296, 15 turns the volume to "full blast", and we can turn it off again with POKE 54296, 0. The volume affects all three voices simultaneously.

Next comes the frequency, or the note's pitch. You have your choice of 65536 different frequencies: You'll get whatever you put in. The note table in the Appendix of the Programmer's Reference Guide is useful for this. You remember the chapter on pointers; the frequency numbers are split up into high-byte, low-byte pairs. These are POKEd into registers 0 and 1 (voice 1), 7 and 8 (voice 2), or 14 and 15 (voice 3).

Now you should set the ADSR. The attack and decay duration are controlled by register 5 (or 12, or 19, dependent on what voice you're using). The attack is the high bits, while the decay is in the low bits. A similar system holds for sustain and release, found in registers 6, 13, or 20, with release set by the highest-value bits. The volume of the tone is in proportion to the volume set in register 24. If you want to use the pulse wave, SID must know the pulse width; it can handle values between 0 and 4095, and can be dealt with through the register pairs 2/3, 9/10 or 16/17. Of the higher bytes of these pairs, only the first (lowest- value) bits are used. Numbers higher than 15 make no difference in this register.

To set the waveform, we should look at register 4(/11/18).
Like some VIC-II locations, every bit here has a special meaning all of its own. Bit 0 represents the start-stop bit for tone production; when it is set to 1, the corresponding voice is switched on and the ADSR started. When set back to 0, the tone will end, the duration dependent now upon the ADSR. Please note when programming that SID will not produce new tones very quickly if a long sustain time is set. Should you want to program a tune with fast note values, it is recommended that you choose a small number for sustain time.

Bit 3 of register 4 (54276) is also useful to us. If two or more waveforms are switched on simultaneously, the SID can block them, and not produce any tone at all. By setting bit 3 and clearing the waveforms, we can overcome this problem. Reinitialize SID with POKE 54276, 8.

In order to switch on a tone, the waveform and start-bit must be POKEd simultaneously; see the 64 Handbook for the sound codes (17, 33, 65, 129). Switching the tone off is not merely a matter of changing register 4 (/11/18) to 0; that's similar to turning your car off while speeding down the motorway! If you set it to 0, the SID suddenly finds no waveform, no ADSR, and no graceful way to end the note, the result being the typical shut-off ⟨CLICK⟩. If only bit 0 is cleared, then you circumvent the click, and the sustain eventually fades. You can do this by POKEing the waveform codes with -1 (i.e., 16, 32, 64 or 128). Look here, we get our bits back two times over! To clarify the preceding material here is a program that lets you play tunes on the keyboard:

```
10 PRINTCHRS(147)
20 PRINT " W E T  Y U"
30 PRINT "A S D F G H J K"
100 S=54272
110 POKE S+24, 15:REM VOLUME
120 POKE S+5, 136:REM ATTACK/DECAY
130 POKE S+6, 248:REM SUSTAIN/RELEASE
140 POKE S+4, 8:REM SID INITIALISATION
150 FORI=0TO40:NEXTI:POKES+4, 16: REM STARTBIT=0
160 GETA$:IFA$=""THEN160
170 IF A$="A" THEN POKE S, 207: POKE S+1, 34
180 IF A$="S" THEN POKE S, 18: POKE S+1, 39
190 IF A$="D" THEN POKE S, 219: POKE S+1, 43
200 IF A$="F" THEN POKE S, 118: POKE S+1, 46
210 IF A$="G" THEN POKE S, 39: POKE S+1, 52
220 IF A$="H" THEN POKE S, 138: POKE S+1, 58
230 IF A$="J" THEN POKE S, 181: POKE S+1, 65
```

```
240 IF A$="K" THEN POKE S, 157: POKE S+1, 69
250 IF A$="W" THEN POKE S, 225: POKE S+1, 36
260 IF A$="E" THEN POKE S, 101: POKE S+1, 41
270 IF A$="T" THEN POKE S, 58: POKE S+1, 49
280 IF A$="Y" THEN POKE S, 65: POKE S+1, 55
290 IF A$="U" THEN POKE S, 5: POKE S+1, 62
300 POKES+4, 17:GOTO150
```

Lines 10-30 define the keyboard layout, followed by a preparatory section (100-140). There is a short FOR-NEXT loop in line 150, producing a short sustain of constant duration, while in this same line, the startbit is set to 0.
If you'd like to change the waveform, alter the values in this line and in line 300. Lines 160-290 are self- explanatory; when a specific key is pressed, a certain frequency is sounded.

SUMMARY: Tone Production

Sound Interface Device: 54272-55295
register 24 = 54272 + 24

Volume: register 24, Range: 0-15

Attack: Highermost halfbyte in registers 5/12/19

Decay: Lowermost halfbyte in registers 5/12/19

Release: Highermost halfbyte in registers 6/13/20

Sustain: Lowermost halfbyte in registers 6/13/20

Waveform: Registers 4/11/18 Bit 4: triangle
Bit 5: sawtooth
Bit 6: pulse wave
Bit 7: noise
Bit 3: initialization
Bit 1: start-stop-bit

Frequency: Register pairs 0/1, 7/8 or 14/15

9. THE KEYBOARD.

It's clear that the keyboard is the 64's most striking external feature. Hardly a computer in this price range has such a quality keyboard. Not only is it comfortable to use, but many programming tricks are possible, as you'll see in this chapter.

9.1. KEYBOARD DESIGN AND OPERATION.

The keyboard is normally addressed in BASIC programs using GET and INPUT. One more is using OPEN with the keyboard's device number (which is 0). Unlike the normal INPUT command, this method does not display a question mark prompt. Part of the I/O range is connected to it, the CIA 1 interface, which reads the two parallel ports (which work closely with the user port). The 64 keys, which are electrically divided into 8 columns and 8 lines, are shared with these ports. One of these ports is programmed for output; this is where the keyboard columns are read. If a key is pressed, both ports register input. The interrupt routine, well-suited for keyboard input, has nothing else to do than choose the columns and rows pressed. A decoder table in ROM then calculates the ASCII code of the key and stores it in the keyboard buffer.

When the interpreter runs in direct mode, the interrupt takes the ASCII code in the buffer and deals with it accordingly (e.g., ⟨RETURN⟩=ASCII 13). If just a BASIC program is running, the keyboard buffer remains unaltered until the appearance of GET, INPUT or the program end. With a GET statement, the interpreter takes the first character from the buffer and stores it to program given variables. An INPUT statement works in much the same way, only it appears on the screen, and is acted upon only after ⟨RETURN⟩ is hit.

The rows and columns forming the keyboard matrix have two peculiarities: The RESTORE key is not in the matrix—it operates directly through the processor (like the RESET switch in Chapter 1.6.), and clears out a special interrupt there. This routine tests whether the RUN/STOP key has been pressed at the same time. If the RUN/STOP-RESTORE combinations is detected then a sort of mini-reset occurs, hopefully putting things back to normal.

The second oddity lies in the SHIFT key. The computer can distinguish between left and right shift keys, which lie in different columns. But the SHIFT-LOCK key is read as a form of the left SHIFT key, i.e., the system can't distinguish between the two.

The entire principle is easily transferred to the VIC-20; only the electrical arrangement of the keys is different.

9.2. READING TWO KEYS SIMULTANEOUSLY.

It may be desirable in many programs to read several keys simultaneously, such as when steering two spaceships independently.

Look at the keyboard matrix (Figure 4). The two memory locations involving the keyboard are 56320 and 56321. Normally all of the bits in these registers are 1. If a certain column is to be read, the corrseponding bit in 56320 will be changed to a 0. When reading a row if a key is pressed, the corresponding bit in 56321 is set to 0.

We can select a specific column using a POKE, and test bits read from the keyboard, using one PEEK. If the two keys are in different columns, they can simply be read one after another.

Since the interrupt routine can interfere with our handiwork (since it is involved with choosing columns), we'll switch it off. Besides that, we must turn off the RUN/STOP key using POKE 788, 52, otherwise every reading of the lowest keyboard line will call for a BREAK. Here are all the commands:

```
POKE 56334, PEEK (56334) AND 254
POKE 788, 52
POKE 56320, COLUMN CODE
IF (PEEK (56321) AND (2 ↑ BIT NUMBER)=0) THEN PRINT "KEY PRESS"
POKE 56334, PEEK (56334) OR 1
POKE 788, 49
```

This allows the keyboard to be read directly from BASIC. Should you wish to read several keys, further IF-THEN constructions, and more POKEs to read the different columns must be inserted. The column code is calculated with this formula:

```
CODE=255-2 ↑ COLUMN NUMBER
```

The column number is represented by the position of bits in location 56320, from which the column is chosen. The IF-THEN construction in the above program commands has the task of testing whether the bit desired has been set to 0.

You can get the line and column numbers from Figure 4.

Memory cell 653 offers another aspect of reading two keys at the same time: This is where the current SHIFT-pattern is shown, i.e., the bits of this register tell whether the SHIFT, COMMODORE or CTRL keys are pressed. Bit 0 is set to 1 by SHIFT, bit 2 by C=, and bit 3 by CTRL. This bit setting is done independently so that all three keys can be pressed simultaneously if necessary. The interrupt routine is again responsible for this. If we wanted to use 653, we woudn't have to switch off the interrupt. The 64 can be told from BASIC to look for a specific keypress using the following command:

```
IF (PEEK (653) AND 2 ↑ BITNUMBER) THEN PRINT "KEY PRESSED"
```

SUMMARY: Simultaneous Keyboard Reading

There are two possible methods:

a. PEEK (653) checks the SHIFT pattern; the SHIFT, C= and CTRL keys can be independently read.

b. After a column is chosen with POKE 56320, X, it can be determined which key in that column has been pressed. The arrangement is listed in Figure 4.

									56321	Bit
	1	£	+	9	7	5	3	INST DEL	254	0
	←	*	P	I	Y	R	W	RET.	253	1
	CNTRL	;	L	J	G	D	A	CRSR ⇨	251	2
	2	CLR HOME	-	Ø	8	6	4	F7	247	3
	SPACE	SHIFT RIGHT	.	M	B	C	Z	F1	239	4
	C=	=	:	K	H	F	S	F3	223	5
	Q	↑	@	O	U	T	E	F5	191	6
	RUN STOP	/	,	N	V	X	SHIFT LEFT	CRSR ↓	127	7
56320	127	191	223	239	247	251	253	254		
Bit	7	6	5	4	3	2	1	0		

SHIFT LOCK = SHIFT LEFT, RESTORE NOT TESTED

FIG. 4. Keyboard matrix.

9.3. KEYBOARD CUTOFF.

Wouldn't it be nice to turn off a few keys (such as RUN/STOP), or even the entire keyboard. There are many possibilities on the 64.

To turn the keyboard off completely, the interrupt routine should be turned off: The cursor disappears, and the computer appears to be locked-up, but RUN/STOP-RESTORE will cancel this state.

The same goes for POKE 649, 0: This also turns the keyboard off, but the cursor is still there and the RUN/STOP key remains operational. The actual function of location 649 gives the amount of space available in the keyboard buffer.
If the length is set to 0 (normally 10), this means that the operating system thinks that the buffer is already full, and therefore "forgets" any keys pressed; although BASIC will hold all keypresses in the buffer (whether in direct mode, or in a GET or INPUT statement), no input will work.

POKE 655, 71 gives the same results, only it alters all the characters in the keyboard decoder table. RESULT: The interrupt routine can't form any ASCII codes—and the keyboard buffer remains empty. POKE 655, 72 will put things right again.

If you want to turn off the RUN/STOP key, use POKE 788, 52.
Then only RUN/STOP-RESTORE will stop a program. Poke 788, 49 returns the BREAK function.

The mini-reset (RUN/STOP-RESTORE) can be prevented with POKE 792, 193. Using a combination of the last couple of POKES, BASIC programs can be made "unstoppable" (aside from the OFF/ON method). In addition, if you want to protect the program from a LIST, POKE 808, 234 should be given; this will render BREAK inoperative, while a LIST will only show gibberish.

SUMMARY: Keyboard Cutoffs
Whole keyboard off:
1. Switch off interrupt
2. POKE 649, 0 (keyboard buffer length to 0)
3. POKE 655, 71 (changes pointer to decoder table)

RUN/STOP off: POKE 788, 52

RUN/STOP on: POKE 788, 49

RESTORE off: POKE 792, 193

RESTORE on: POKE 792, 71

BREAK off + List Protection: POKE 808, 234

9.4. THE REPEAT FUNCTION.

You've used this before, last time you wanted to send the cursor elsewhere on the

screen. In a few cases, though, you've probably wanted the repeat function to work on keys other than the cursor controls. Wish no longer!
Memory location 650 controls the entire repeat function. Normally there is a 0 in this register, that shows the interrupt routine that only the cursor keys and shift key should repeat. If bit 6 is set (by POKE 650, 64), the repeat function is turned off completely, while POKE 650, 128 does the exact opposite: Now you have all the normal keys repeating, like A, S, D, etc. Along with these useful details, there's something else you should know about the repeat function: Before a keypress will automatically repeat, there is a brief time lag (about 0.5 second); this is to prevent any accidental repeating caused by the user holding a key down a bit too long (an accidental repeat could disturb the user's work). This time lag is produced in location 652. From there, the interrupt counts down to 0 from the number avaiable (normally 16). The repeat function will only start when 0 is reached. The contents of memory cell 651 are counted down in a similar manner; when 0 is reached, this register is loaded with a new starting value (4), and a new "keypress" is transferred into the buffer. Therefore, POKE 651, 255 can stretch out the repeat time lag to around 4 seconds. That's the whole secret to keyboard repeating.

SUMMARY: Repeat Function

POKE 650, 128: All keys repeat
POKE 650, 64: Repeat off
POKE 650, 0: Normal condition
POKE 651, 255: 4 second delay to repeat

9.5. ANOTHER METHOD OF READING THE KEYBOARD.

As you know from the last section, the interrupt routine puts characters into the keyboard buffer as ASCII-codes; on the way there, though, there is a "way-station" - location 203. Here is where the so-called keyboard codes (which serve as pointers within the decoder table) are stored. The code appears in this register for as long as the key is held. By means of PEEK (203), for example, you can program in "time-dependent" input, the result of which depends on how long the key is held down. You will find a summary of keyboard codes in Table 1, which unfortunately haven't much in common with ASCII-codes.

A keypress discovered by PEEK (203) is not cleared from the keyboard buffer; that's handy for checking data input.

The keyboard buffer can also be cleared. Memory cell 198 gives the number of ASCII codes already stored. POKE 198, 0 will clear it, so that every newly arived

character overwrites the old. After clearing the buffer, WAIT 198, 1 will halt things until the next keypress. As soon as a new character arrives, it will register in the buffer location (198). The WAIT command simply has the task of returning control to the program on recognizing the arrival of a new character. Using this technique a character from a GET statement could be held in the buffer; it'll save you typing in a lot of IF-THENs.

The keyboard buffer lies in locations 631-640. Here is where the characters are put as ASCII-codes, so that BASIC can call them. You can simulate keypresses by POKEing in characters. Remember to increase the pointer in location 198 so that BASIC will "notice" the POKEd characters.

As you have probably figured out for yourself, reading the keyboard is pretty versatile. Take these ideas, and use them to your advantage!

SUMMARY: Reading the Keyboard

PEEK (203) gives the keyboard code of the key pressed.

POKE 198, 0 clears the keyboard buffer.

POKE 198, 0: WAIT 198, 1 clears the buffer and waits for a keypress

The keyboard buffer is contained in locations 631-640.

Keypresses can be simulated by POKEing characters into the keyboard buffer.

A	10	O	38	2		59	@		46	F5	6
B	28	P	41	3		8	*		49	F7	3
C	20	Q	62	4		11	↑		54	STOP	63
D	18	R	17	5		16	:		45	SPC	60
E	14	S	13	6		19	;		50		
F	21	T	22	7		24	=		53		
G	26	U	30	8		27	RET		1		
H	29	V	31	9		32	,		47		
I	33	W	9	L	ARR	57	.		44		
J	34	X	23	+		40	/		55		
K	37	Y	25	-		43	CRSR	DN	7		
L	42	Z	12	£		48	CRSR	RT	2		
M	36	O	35	CLR		51	F1		4		
N	39	1	56	DEL		0	F3		5		

TABLE 1: Keyboard Codes stored in location 203

10. JOYSTICKS, LIGHTPEN AND OTHERS.

Everyone knows what they are, but few realize how they work. These additional devices for games and graphic programs are quite common. The joystick is the most diverse; some people go so far as to say that a C-64 without a joystick is not fully equipped. Here are descriptions of each accessory, in which both operation and reading techniques are discussed.

10.1 THE JOYSTICK Many are puzzled by it, but they know it works. The joystick is really a sort of keyset on the 64, which reads from one keyboard column. Both joystick ports are connected to CIA 1. Port 1 responds to location 56321, the positions corresponding to the keys in column 7. Memory cell 56320 must contain the value 127 to read port 1. This is always the case if the interrupt routine for reading the keyboard has ended; if the keyboard was previously read by locations 56320/1, then joystick use must be preceded by POKE 56320, 127. Different bits in 56321 are cleared when the joy stick is moved. The table below shows which bits are affected:

BIT	7	6	5	4	3	2	1	0
Direction	—	—	—	FIRE	RIGHT	LEFT	DOWN	UP
KEY	—	—	—	SPC	2	CTRL	↑	1

The keyboard layout is also given so that the joystick can be "simulated" by the keyboard.

If the RUN/STOP key is not turned off, then you can use location 145 for a another purpose. This is where the operating system produces a copy of location 56321, so joystick port 1 can also be read with PEEK (145). It's a bit more complicated with joysitck port 2. It's in memory cell 56320, but this location is really designed for column selection (i.e., output), and joystick reading is dependent upon input. Therefore, this port must be switched to the CIA; you can do this with POKE 56322, 224. This command does the following:

1) Memory location 56320 now reads the joystick movement in 56321;
2) the keyboard is turned off, which can be rectified by either RUN/STOP-RESTORE or POKE 56322, 255.

Joystick operation is simple. Basically, it consists of five or fewer separate switches. One is used for the fire button, while the remainder are underneath the stick handle, set in four different directions. The position of the stick turns on the corresponding key—and the 64 notes that in the proper memory locations.

There are differences between the many joysticks on the market. The simplest examples (e.g., the Commodore joystick) work with simple foil contacts while various relatives use microswitches which actually "click" into position.

When purchasing a joystick, make sure that it has the smoothest movement possible; otherwise, fatigue can set in quickly during a game. All ATARI-compatible joysticks will fit the Commodore machines.

SUMMARY: Joysticks

Read joystick port 1: PEEK (56321)
Location 56320 must contain 127 to do so

Read joystick port 2: POKE 56322, 224: REM PORT SWITCHED
PEEK (56320)

Port 1 can also be read using PEEK (145)

10.2. PADDLES.

Paddles are generally known as regulating knobs. Their task is to convey the positions of the knobs to the computer using a numeric value; they give but one direction, as opposed to the joystick. Each paddle is made from a potentiometer; the higher you turn the knob, the more current goes through the computer (simply put), and vice versa. The 64 converts the incoming current from an analog measurement to a digital number using its built in AD (analog-digital) converter. This number is then read into a special register; the AD converter and this register are part of SID. Two paddles can be attached to each joystick port (..only one plug is needed per set of two), so there are two converters and two registers. Their addresses are 54297 and 54298. Each paddles has a fire button; this can be read like "left" and "right" on the joystick, in registers 56321 (port 1) and 56320 (port 2, assuming you haven't forgotten to switch it over to input mode).

We can attach a total of 4 paddles to our 64, but only 2 controls are at our disposal so far. It is be possible to switch back and forth between the two ports. By setting the bits in location 56320, measurement of port 2 occurs. This will happed only if the interrupt is not turned off.

SUMMARY: Paddles
Paddle values are read in registers 54297 and 54298.

Fire buttons correspond with joystick "right" and "left".

Switch the AD converter to port 2 by setting bit 7 in location 56320.

10.3. THE LIGHT PEN.

Now we come to a wonder of technology—or at least, it looks like one. How can such simple looking device as a light pen set points on the screen? Getting a grip on its operation is really not hard.

The actual dot setting is done by a program which works much like the graphic routines in Chapter 6. All the pen has to do is supply the coordinates for the dots; these can be found in two registers.

But how does the VIC-II know exactly where the light pen is pointing on the screen? To answer this question, think about the design of a TV picture. As you already know, it consists of individual dots. An electron gun draws line after line on the screen. When a dot should be lit up, the gun will light that dot; if the dot remains dark, so will the beam. This screen movement takes place so quickly that our eyes see a constant picture. The "renewal" of an individual picture takes but fractions of a second.

If we hold the light pen on the screen while the electron gun is on, impulses are sent to the 64 thru the light pen.
In order for the 64 to determine where a video signal should be produced, it can constantly define the coordinates onscreen. The X and Y values lie in VIC-II registers 19 (53267) and 20 (53268). PEEKing can accomplish this in BASIC.

The values received lie in a range from 0-255. You can calculate and set the corresponding dot in your graphic routine.

SUMMARY: Light pen
Light pen coordinates are given in VIC-II locations 19 (53267) and 20 (53268). Using a corresponding graphic routine you can set dots onscreen using the light pen.

10.4. OTHER ACCESSORIES You've probably seen graphic tablets in computer magazines; they operate somewhat like drawing on a sheet of paper, only the picture appears in high-res on the screen.

There are different principles of operation for such tablets, but they all have one thing in common; you draw using a pen, your finger, or whatever's available. Most work on the principle of transmitting more or less current to the paddle input. Then the computer takes over, so you must have the necessary software.

There is also a joystick made for the paddle input, which I like to call a proportional joystick. It doesn't merely supply general movement, but rather determines exact positioning dependent upon coordinates; this is made possible by a X-Y potentiometer. Actually, it is handled as two potentiometers in one housing, using one controller (one pot for X, the other for Y). When you move the stick, the values which the pots furnish change, dependent upon direction. In this manner, you can control any point on the screen.

The great advantage to these two devices lies in that they supply ready-made coordinates: With them, you can spare yourself a lot of the back-and-forth movement necessary with traditional joysticks.

11. THE USER PORT.

The user port makes the C-64 a very versatile instrument.
Here are some hints on the basics of programming the user port.

11.1. INTERFACE CHIPS IN GENERAL.

Like the keyboard and the joysticks, the user port is supported by a CIA, only this time, CIA 2 does the work.
The CIAs (Complex Interface Adaptors) are interface chips, or I/O chips. These are chips whose task is to send and receive data from peripherals, and guarantee communication with the processor.

Generally, such a chip has three elements: The first is a unit for parallel ports, which you'll remember from the section on reading the keyboard; second, a time unit (which you have used, perhaps without even knowing it); lastly, a serial port.

The following three sections will explain the operation of these elements to you.

11.1.1. THE SERIAL PORT.

Let's begin with the simplest part: As you know, the computer works with parallel bytes, i.e., 8 bits are moved and manipulated simultaneously. A serial interface, however, moves the 8 bits of a byte one after another over a wire. It is somewhat slower than parallel transfer, but has the advantage that 1 data line can be used instead eight seperate lines; for example, data can be transferred over telephone lines.

The processor supplies the parallel bits to the interface chip, the chip sends them out one after another and in proper order. Bits received are arranged like pearls on a string until the byte is completed: The byte is then given to the processor.

The 64 ROM already contains the complete software needed to operate a serial RS-232 port (from the user port). The RS-232 interface can be opened by the command OPEN 1, 2 +(parameters).

11.1.2. THE TIMER.

Whenever the computer needs to time something the timer is set into action. The register can be loaded with any value up to 255. This value is continually decreased; when it reaches 0, the timer sends a signal to the processor. The interrupt represents this sort of internal controled timing.
The timer could be programmed so that within 1/60 of a second, an alarm would ring, after which the timer would restart. The processor could react to this alarm by calling up the main program.

This is how the switching of the interrupt works (Chapter 1). Bit 0 of location 56334 determines whether the timer is counting, or whether it has stopped. If the bit is 0, then the timer remains static. RESULT: No more interrupt.

Aside from this trick, you shouldn't fool with the timer:
In most cases, experiments end with a lock-up.

11.1.3. THE PARALLEL PORT.

All interface chips for the 6510 (or 6502) have one thing in common; the programming of the parallel port. Most have the same chip for two such ports, such as the CIAs (Complex Interface Adaptors).

All these ports work with 8 data lines, either programmed for output or input; this is handled in the chip by two special registers. The data direction register (DDR) shows the direction each individual line is switched to. A 1 means output, 0 means input.

The second register for the port has different tasks for each mode. It acts as a "catch byte" for the input lines, i.e., where the processor can call for and receive data.

The output lines allow the processor to send data to the peripherals. Communicating with the addressed device which contains the data is known as "handshaking". The processor has delivered its byte to the I/O chip by means of a special handshake line, and can receive data bits in this register from its addressed "partner". The sender waits for the next byte until the receiver announces receipt of the data through the handshake line. The handshake procedure runs on either one or two lines.

Data transmission does not absolutely have to be run by the handshake system, but without this system data may be lost.

11.2. HOW DO I USE THE USER PORT?

The user port is essentially a parallel port with different "accessory lines". Most of these lines supply internal signals, so we are limited to an 8-bit-wide port and a "lent-out" direction line. Lent-out, because it's coming from port A of CIA2, which represents a data line.

CIA2 has a starting address of 56576. That is also the address of the data registers for port A (reg. 0), where bit 2 again gives the condition of the direction line. All the other lines in this port are used internally, so we should only manipulate bit 2! The other registers begin with register 1 (56577): That is the data register for port B, which represents the user port proper. Here the 8 free data lines are accessible.

The DDR (Data Direction Register) follows at register number 2 (56578 for port A: Remember, change bit 2 ONLY) and 3 (56579 for port B). This has already been used elsewhere.
POKE 56579, 255 programs all 8 lines for output; POKE 56579, 0 sets them for input.

Programming the direction line must be approached with some caution. POKE 56578, PEEK(56578) AND 251 switches input off, while POKE 56578, PEEK (56578) OR 4 does just the opposite.
To send data out the port, we just write to memory location 56577, and do the inverse to read incoming data. We can switch the direction line with POKE 56576, PEEK (56576) OR 4, and turn it off with POKE 56576, PEEK (56576) AND 251. If we put both commands one after another in a program, then a shorter impulse can be produced.

The direction line is represented by pin M of the user port (see Figure 2, or the 64 User's Guide), and the 8 data lines can be found on pins C-L.

SUMMARY: Programming the User Port

DDR for eight data lines:	56579
DDR for direction line:	56578 (bit 2 only)
Data register for port:	56577
Data register for direction line:	56576 (bit 2 only)

11.3. SAMPLE APPLICATIONS.

The user port is quite versatile; here are some suggestions for your own experimentation.

One of the simplest examples is switching lamps or LEDs, connected to the user port by a series of driver transistors or relays. This could be taken a step further and perhaps become a light organ, controlling the lights according to the volume of the music (measured by a microphone rigged to the AD converter). A strobe light or other effects could be accomplished with other programs.

Coupling two Commodore computers (any type, since they all possess the same user port) for data exchange is also conceivable. This way, for example, a VIC-20 could take measurements of the high-res graphics a 64 is putting on a large screen.

Electronically adept readers can also build their own serial interface to receive and transmit data by telephone [TRANSLATOR'S NOTE: The 1600 and 1650 modems are unavailable outside of North America..]. It's also possible to connect a non-Commodore printer to the C-64. Equally suitable are punch card readers and punchers, home robots and pocket calculators. The do-it-yourself possibilities are limitless!

12. BASIC & THE OPERATING SYSTEM The Commodore 64 operating system and BASIC offer us many useful functions. It is sometimes desirable to in uence these functions to our own ends (e.g., LIST to screen and printer). Here, then, is a discussion on the possibilities of such manipulation.

12.1. PRODUCING BASIC PROGRAM LINES.

Say you want to write an all-purpose program that will draw a high-res graph on the screen. If the program doesn't give the function for the graph, it must be possible to type the function in. A simpler version of the program comes to us, which inserts the function using DEFFN; yet the user needs programming knowledge to change it—it would have been more convenient to use INPUT to enter the function. Yet if we input the function as a string, it will be stored in memory as a string and not as a function. The last possibility (and the simplest) is to have the program change itself, which the operating system allows us to do.

In order to understand the method behind this, we should understand the normal origin of a program line. It all begins when you type in a carefully planned (hopefully) sequence of characters and numbers. These symbols appear onscreen at the same time that they are typed in. If one of the characters is a ⟨RETURN⟩, then the BASIC interpreter takes the entire screen line (not just the characters typed in), puts it into the BASIC input buffer, and converts this set of characters into a program line, or (if no line number exists) treats it as a direct command. The interpreter dosen't know whether characters are PRINTed to the screen or typed in from the keyboard. From that concept comes our method: First, the text intended for the new program line will be put onscreen, then we cause a ⟨RETURN⟩ over this line and it is entered as a new program line. To do this, an artificial keypress is produced, by POKEing it into the keyboard buffer as an ASCII code. Next an END-of-program follows, so that the keypresses in the keyboard buffer are executed after termination of the program. This causes two problems: During the production of a new line, the variables will be cleared (which is also done by normal program loading). It follows that the new program line should be placed where no important data is stored (like at the beginning of the program). To preserve any variables
POKE these into free RAM, where the operating system won't touch them.

The second problem will occur after the line has been created: After the line has been created ,you'll have to put in an artificial GOTO xxx, so that the program will go to the proper program line. Here is a sample listing:

```
10 INPUT"FUNCTION";A&:REM INPUT FUNCTION TERM
20 PRINTCHR$(147)+CHR$(17)+CHR$(17)+CHR$(17)+"100
   DEFFNF(X)=";A$:REM NEW PROGRAM LINE TO BE ENTERED
30 PRINT"GOTO70";CHRS$(19):EM COMMAND TO RESTART PROGRAM
40 POKE631, 13:POKE632, 13:REM TWO ⟨RETURN⟩S IN KEYBOARD
     BUFFER
50 POKE198, 2:REM DUMP KEYBOARD BUFFER
60 END
70 REM ACTUAL START OF PROGRAM
```

When you have keyed in and started this program, you'll quickly see its logic from what appears onscreen. The line produced is indistinguishable from a normal program line. This program can be run as often as you like. If bad input occurs, the interpreter will quit with a ?SYNTAX ERROR after running through the new created line.

By the way, this application can be greatly extended.
Program lines which you don't really need can be deleted using this procedure; also, several ines can be produced at the same time. This is all made possible by the INPUT command and the use of the keyboard buffer in the subroutine.

12.2. LIST PROTECTION.

When doing your own programming you may wish to make a program line invisible when the program is listed. This can be achieved with a POKE.

To understand how this works, it is necessary to know about program line format. The first couple bytes form a number—the pointer to the next line (or line-link). That way, the interpreter can "jump" from line to line. If these two bytes read 0, there are no more program lines in memory, and the program ends.

The line number is in the two bytes which follow the line pointer. This, too is designed like a pointer. Next, the actual BASIC commands follows in interpreter code. The end-of-line is represented by a 0. It is with this 0 that we can play a little trick on the interpreter; by POKEing in a 0 after the line number, this causes the LIST routine to prematurely jump to the next line (the pointer at the beginning of the line remains undisturbed). A GOTO would not be affected, since the routine still looks for the line, and the pointers are still aimed in the proper direction.
The routine which looks for the next BASIC command in the program will jump to the 0 in these four bytes, causing "errors" during a program run. To avoid hindering command execution, any five characters (please, no commands! must be written into the line: The first of these characters will be overwritten by the 0, while the remaining four serve as place holders.

How do we know, though, which bytes we must overwrite?
There's a little trick for that, too. We put a STOP-command in front of the protected line, and let the program go up to there; after the BREAK cause by the STOP, the pointer to the next BASIC command is in locations 61 and 62. If the STOP is put at the end of the line, the pointer shows the end-of-line and a 0. Add 5 to this address to get the byte needed, then hide the line with POKE AD, 0. After this command, only the line number will be listed—the text won't show. All that remains is to delete the now-unnecessary STOP command. Here's the procedure:

1. Put STOP at the end of the line preceding the line to be protected.

2. Put 5 place-holders (any characters) at the beginning of the line.

3. AD= PEEK (61) + 256 * PEEK (62) + 5.

4. POKE AD, 0.

5. Delete STOP command.

To protect the entire program from a LIST, alter the LIST- routine vector. This vector is in locations 774/775. POKEing 775, 1 will alter the pointer so that every list command works like a RUN/STOP-RESTORE. This protection can be removed with POKE 775, 167.

12.3. RENUMBER.

The owner of a BASIC extension such as VICTREE or SIMON'S BASIC has the RENUMBER command. This command lets you renumber a program in memory—this has advantages for such things as merging. This command can also be simulated without a BASIC extension.

As you remember from the last section, every program line in memory begins with 2 pointers;
1.) to the beginning of the next line in memory,
2.) gives the line number in pointer format (not a pointer per se).

If we add 2 to the first address we get the address of the next line number. In this way, we can fit new line numbers to any program using a POKE. Here's the program:

```
63900 BA= PEEK (43) + 256 * PEEK (44)
63910 INPUT "STARTING NUMBER";SA: INPUT "INTERVAL";SW
63920 HI = SA/256 :LO = SA AND 255
63930 A=PEEK (BA+2) + 256 * PEEK (BA+3)
63940 IFA <= 63900 THEN PRINT"OK!!":END
63950 POKE BA+2, LO :POKE BA+3, HI
63960 BA = PEEK(BA) + 256 * PEEK (BA+1) : SA=SA+SW
63970 PRINT SA-SW SA"="A "="A: GOTO63920
```

This routine will renumber a program; start it with RUN 63900. We use high line numbers to be sure that the routine stays above the end of the program being changed, because the routine will not renumber itself.

Line 63900 computes the starting address of the first line using the start-of-BASIC pointer. Line 63910 asks the user to input the starting line number and interval between program lines. If, say, you want the program to begin with line 10, and renumber in steps of 10, then you give a 10 for each prompt.

Line 63920 calculates the new line-number's high- and lowbyte, while line 63930 holds the old line number in memory. If this is larger than or equal to 63900, the renumber routine breaks off, since the routine cannot renumber itself. The next line POKEs in the high- and lowbyte of the new number.

Finally, the starting address of the next line must be calculated, the line number is raised by the step entered in the INPUT, and a renumber protocol is printed. This protocol shows the old equivalent of every new line number. This makes the modification of GOSUB, GOTO and other commands much easier: Our routine doesn't change these
within the program.

I recommend that printer owners make a prinout of the newly-renumbered program; to change those jump commands, it's best to have the program for review in hardcopy. To get a printout, just insert OPEN1, 4:CMD1 at the beginning of the routine.

12.4. RENEW.

The NEW command is the one most likely to cause insanity in computer owners: Every computer has one. By accidentally keying in those three characters (+⟨⟨RETURN⟩⟩), many programmers have unwittingly lost the fruits of their labors, because the computer has "forgotten" the program which it previously had in memory. To prepare for such twists of fate, I have written a program that can actually undo the NEW command.

It's no secret that NEW doesn't clear memory completely, but simply resets the two supporting program pointers. The first of these two is the pointer to the beginning of variables. After NEW, it points to the start of program, so all variables and the old program (which is actually still in memory) can be overwritten.

The second pointer is found in the first program line, at addresses 2049 and 2050. Normally it points to the next line; after NEW, however, it contains two zeroes (to mark the end-of-program). Therefore, RENEW has two things to accomplish:

1) Find the end of the first line, which should be marked with a 0. Once this zero is found, then ADDRESS + 1 must be POKEd into the pointer in bytes 2049-2050;

2) Find the end-of-program, which is signified by a 0 in the highbyte of the pointer for the next line. When the end of the program is found, the address given by the start of variables is raised by 2; the pointer to the end of BASIC can now be set.

There is still a problem. When we key in the program, the old program is destroyed by the new one. Therefore, BASIC must be shifted to a location not used by the interpreter.

The 4 K RAM from 49162-53247 is a good bet. In order to move the entire BASIC range, we use four commands:

POKE44, 192:POKE56, 208:POKE49152, 0:NEW
Now we have created a second independent memory range, and RENEW can be put in.

Here's the program:

```
10 AD=2052
20 AD=AD+1:IFPEEK (AD) ⟨⟩ 0 THEN 20
30 AD=AD+1
40 POKE 2049, AD AND 255:POKE 2050, AD / 256
50 IF PEEK (AD+1) ⟨⟩ 0 THEN AD=PEEK (AD) +
256 * PEEK (AD+1):GOTO50
60 PRINT "POKE 45, " (AD+2) AND 255
":POKE 46, " INT ((AD+2)/256)":POKE 44, 8:POKE 56, 160:CLR"
70 PRINT"[CRSR UP][CRSR UP][CRSR UP][CRSR UP][CRSR UP]"
```

Now for a few explanations. Line 20 looks for the end of the first line; if this is found, the address is raised by 1 (line 30) and the pointer is restored to the second line (line 40).

Line 50 "jumps" from pointer to pointer until it finds the end of program (0). The newly-found address will not be directly POKEd in because the routine itself will lock up.
Instead, the necessary commands are PRINTed to the screen (line 60), and the cursor travels over them (line 70). After the program ends, the user need only hit ⟨RETURN⟩ to complete the variable pointer.

Now the orginal program will be restored in memory. If you have transferred the start-of-BASIC (e.g., for high-res graphics), you'll have to alter the starting address in lines 10 and 20, and the POKE command in line 60. After these POKEs, BASIC is again put back (POKE 44, 8). You can now use the old program as usual.

RENEW should only be used once. A restoration is only possible if you have altered the variable pointer. If the variable pointer has not been shifted by a NEW or POKE, the computer will lock-up.

Regarding lock-ups: If you recovered from a crash using a reset switch, you can restore a BASIC program again by setting up RENEW. So long as the current is still on, all the data is still there.

12.5. RESTORE

The RESTORE command in BASIC sets the pointer back to the first DATA statement. Wouldn't it be nice if we could set it to a particular DATA statement.

A couple of POKE commands can accomplish this. To do so, you should know that the interpreter stores both the line number and the address of the last DATA statement in zeropage. The line number is stored as a pair of bytes (similar to pointers) in locations 63/64; we can find the bytes of the last data statement in 65/66.

If we want to simulate a RESTORE, then we do the following:

1. Let the DATA statements before the DATA statement desired be read (e.g., in direct mode).
Should the RESTORE be desired on the fifth statement, then the first four DATA statements should be read;

2. PRINT PEEK (63), PEEK (64)
The numbers which appear represent the line number of the last DATA statement read. Write 'em down!

3. PRINT PEEK (65), PEEK (66)
Write these down too! They form the pointer to the byte after the last DATA statement;

4. POKE 63, 1st number: POKE 64, 2nd number: POKE 65, 3rd number: POKE 66, 4th number

Put these commands in in the section of your program where you wish to RESTORE to the desired DATA statement. This way, the pointer will be set to the desired DATA statement.
BASIC gets the impression that it has read the proceeding DATA-lines.

SUMMARY: RESTORE

The line number of the last DATA statement is stored in memory cells 63 and 64. The byte address of the last element found is the pointer in bytes 65 and 66. Both pointers can be altered by POKE.

12.6. DIFFERENT TRICKS.

After a program BREAK or an error, the computer mentions the line at which the program left off, but if you've cleared the screen a bit too hastily, you will have missed the line number. Memory cells 59 and 60 offer help; this is where the last line number is stored (in pointer format). Use this command to view it.

```
PRINT PEEK (59) + 256 * PEEK (60).
```

You can prevent a SAVE with this sequence:

```
POKE 801, 0:POKE 802, 0:POKE 818, 165
```

This turns the vectors necessary so that SAVE is impossible. Disadvantage: The computer hangs up by pressing RUN/STOP-RESTORE.

Finally, a few SYS commands to use in your own programs:

SYS 65499 sets TI to 000000; this is faster than assigning a new string.

Aesthetically-minded Commodore owners can end a program with SYS 42115 instead of END; this will bring up a warm start in BASIC as if BASIC were switched in directly. That way, the cursor immediately goes to the next line, and no READY is given. Also, a CONT is ineffectual after this SYS.

If you want a program to end with a power-up screen, SYS 58253 will suffice but you will lose the program in memory. An invisible ?SYNTAX ERROR can be obtained with SYS 44808.

SUMMARY: Operating System Tricks
Last line number is stored in memory cells 59 and 60
SAVE-protect: POKE 801, 0: POKE 802, 0: POKE 818, 165
Setting TI to 0: SYS65499
END without READY: SYS 42115
Power-up screen: SYS 58253
SYNTAX ERROR: SYS 44808

12.7. BASIC EXTENSIONS.

Almost any Commodore owner knows about BASIC extensions, but few of them know what such programs do. The first example of these useful aids appeared in the days of the dear departed PET. At first, there were the so-called Toolkit Commands,

which eased program editing. Among them were AUTO, which automatically numbered program lines, saving the programmer lots of typing; FIND, which looked for specific expressions in the program text; TRACE, which let you control and test a program one step at a time; DUMP, which dropped all variables and their contents; and you recall RENUMBER, RENEW and MERGE from previous chapters.

More advanced versions make file handling very easy.
MASTER 64 is a BASIC extension that includes both "toolkit"
and advanced file handleing commands. Since BASIC doesn't easily support the fantastic sound and graphic capabilities of the 64, many BASIC extensions are offered with special commands for drawing and tone production. SYNTHY-64 is one such program for music from DATA BECKER and GRAPHICS 64 is for graphic programs from FIRST PUBLISHING.

A few program extensions place structural commands at your disposal, with which you can write programs without GOTO.
In this case the individual sections of a program are written in modules (similar to subroutines). Instead of GOSUB to call a plotting routine, we would use CALL PLOT X, Y. This techniques promotes better programming.

12.8. OTHER PROGRAMMING LANGUAGES.

The Commodore 64 has the capacity for loading and using programming languages other than BASIC. The best-known could very well be PASCAL. The big characteristic of this language is structured programming, i.e., GOTO is taboo (a few newer versions of PASCAL have this command, but it is usually nonexistent). This prevents programmers from aimless coding before a solid concept is figured out. PASCAL is a complied language, i.e., before running, the program text is first converted into a language understood by the computer (machine language or pseudocode, which is almost as fast).

At the opposite end of the spectrum, FORTH represents an interpreter language. Here, too, structure is of great value; you can't do roundabout programming trying to define commands (neither here nor in m/l). FORTH gives you only a few fundamental commands, and the ability to define your own. The result is a high-speed language.

LOGO is equally versatile: This language is so easy to learn that even first-graders can pick it up. Its main selling points: Turtle graphics (you control an intelligent turtle that can produce some impressive graphics), and modularity. LOGO lends itself equally well to mathematical and geometrical problems.

13. MACHINE LANGUAGE.

In the long run, if you are serious about 64 programming, you can't do without knowing machine language. Many beginners, however, find it particularly difficult to think in machine language. This chapter should help: With the Simulator at the end of this book, you can experiment with m/l. Then you can decide whether you want to get into machine language programming, or whether you'd just rather go back to BASIC (not a bad way to go, either). Since the Simulator is written in BASIC, naturally it doesn't possess the extraordinary speed of real m/l; this you've seen from the graphic clear routine in Chapter 6.

13.1. WHAT IS MACHINE LANGUAGE, ANYWAY?

As you know, machine language represents the most direct way to communicate with the processor, without using a compiler or an interpreter. This is why machine language moves with such high speed.

Machine language embraces different fundamental operations from which all the complex commands of BASIC (or whatever language) are assembled. Commands can be divided roughly into three groups. The easiest for BASIC programmers to understand are the jump commands, which can jump around the program in memory, much like GOTO and GOSUB. The second set of commands manipulate data (e.g., addition, combinations, etc.). The last group handles the operations that move the data from one place in memory to another.

It is fundamental to know that this type of microprocessor doesn't know variables; it only knows normal memory cells and internal registers. Generally, data manipulation is executed in the internal registers only.

A machine language command always requires one byte (called the operations code) and up to two bytes for operands, etc. A memory cell can therefore be a command, an address or datum.

13.2. TIME.

The entire computer is controlled by a tiny invisible quartz, which keeps time (0.98 MHz = 980, 000 beats or cycles per second).

The processor can execute fundamental operations through the time cycle, which means these operations can occur during the execution of a single m/l command. The shortest m/l commands require two cycles—for "fetch and decode a command from memory" and then "execute the command". More complicated operations require more cycles.

13.3. THE HEXADECIMAL SYSTEM.

To learn machine language, you should understand the hexadecimal number system. This system consists of 16 numbers (0-9 and A-F). It's used so frequently because the conversion from binary to hex is a very simple one; you take the binary number in halfbytes, and convert them into a hex number. The table below shows the decimal and binary equivalents:

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

The byte 10101011 would be the hex number AB (1010=A 1011=B). In order to convert hex numbers to decimal, you must first change all the digits into decimal equivalents. These numbers will be multiplied in their places of value by powers of 16, and the products finally added.

An example:

ABCD (hex)

```
     A          B          C          D
= 10 * 16 ↑ 3 + 11 * 16 ↑ 2 + 12 * 16 ↑ 1 + 13 * 16 ↑ 0
= 10 * 4096   + 11 * 256    + 12 * 16     + 13 * 1
= 43981
```

To do the reverse, you continually divide the decimal number by 16, and note the remainders as hex numbers.

Example:

```
53000 / 16 = 3312 remainder 8 ---- 8
 3312 / 16 =  207 remainder 0 --- 0
  207 / 16 =   12 remainder 15 -- F
   12 / 16 =    0 remainder 12—C
53000 (dec.)  =  CF08 (hex)
```

There are many pocket calculators with special functions for base conversion. Good assemblers and monitors also offer such functions.

13.4. BINARY ADDITION.

It should be said from the beginning: Binary addition is distinguished from decimal only in the numeric system; aside from that, it works the same as decimal addition.

The sum of two zeros, or 0 and 1 (equal in whatever number system) needs no explanation, since they're added normally. If we want to calculate 1+1, however, then we have a problem. The result in decimal would be 2; not so in binary. Therefore, a transferrence to the next place should occur (like the carrying over of 10 in decimal):

```
Binary     Decimal
  1          1
+ 1        + 1
------     -----
 10          2
```

Entire bytes work just as easily:

```
  01101110   = 110
+ 00001001   =   9
------------ -----
  01110111   = 119
```

If three ones must be added then a transfer is coming (1+1+1=3), then the result is 11 (is this clear?).

```
  10010011
+ 11011111
------------
 101110010
```

We have a nine-bit result. That ninth bit is called the carry bit, or transfer bit. It shows that the addition of two eight-bit numbers has overstepped the allowable range for one byte (0-255); it must then be added to a second byte, which puts us into 16-bit addition. Computers do not
come equipped to only handle 8-bit numbers, since numbers have an infinite range. Fact is, an 8-bit microprocessor such as the 6510 can only work with 8 bits at a time; when a number stands in, say, two bytes, the addition will carry over through both numbers. Both parts of the number can be added independently until that carryover occurs, all you need to do is bring in the carry bit to work with greater numbers. The carry bit has the task of moving from the last place of the first byte to the first place of the second byte, and storing the bit there. An example: (Note the transfer bits!)

```
  1111111      11111   (transfer bits)
  00110101     10010011
+ 10011011     11011111
------------  ----------
  11010001     01110010
```

13.5. BINARY SUBTRACTION.

If a computer wants to subtract one number from another, it first produes the negative equivalent of this number, and adds the two. It does so because addition and negation are fundamentals of electronics (like AND, OR, XOR, NOT), but subtraction is not.

In order to represent a neagtive number, a byte must have its normal range (0-255) shifted to −128 to + 127. The highest value bit (bit 7) serves as an indicator. If it's set at 1, we have a negative number; if it's 0, then the byte is positive. However, a number can't be made negative just by setting bit 7. An example makes the difficulty clear:

```
  00000001        1
+ 10000001    + (−1)
----------   ------
  10000010    (−2)?
```

If translated into decimal, this would mean that 1−1=−2. Therefore, we'll take another route: In some digital machines, such as the 6510 processor, the principle of twos complement is used for subtraction. A byte can easily form the twos complement by multiplying by −1 and then adding 1. Thus, all the bits should be inverted, and 1 added to the byte.

Example:

```
                01011011
inverted:       10100100
              +        1
              ----------
                10100101
```

If we then compute 1-1 in binary, we'll get the correct answer.

```
                00000001
              + 11111111
              ----------
                100000000
```

As you can see, a carry apparently results, but otherwise the subtraction is correct. In subtraction if the carry bit is 1, no carryover results; with 0 we exceed the range. From this foundation, the carry bit must be set to 1 before every subtraction.

Aside from setting the carry bit, the 6510 executes all these tasks automatically during a subtraction command.

13.6. HIGHER ARITHMETIC.

6510 machine language only has two arithmetic commands—for addition and subtraction. All the other types of arithmetic must be assembled from these fundamentals. For instance, to multiply X * N, X is simply added N times. This naturally works for whole numbers only; for fractions, a special algorithm is necessary to figure the number out place-by-place, rather than compute the entire number at once. The principle is basically the same as above.

To divide X by N, simply take N from X continually. The number of subtractions possible until N is greater than X is the result. Here is an example:

```
10 / 3 = ?
10 - 3 = 7    number register = 1
 7 - 3 = 4    number register = 2
 4 - 3 = 1    number register = 3
10 / 3 = 3    remainder 1
```

These methods are permissible thanks to the vast speed of machine language. By the way, a pocket calculator works on the same principle; every time you press a calculator key, a tiny m/l program runs (naturally using the same algorithms).

Still higher forms of math are composed of these four basic functions (e.g., exponentiation, sines, etc.). In short, all mathematical operations can be expressed through tiny AND, OR, XOR and NOT operations.

13.7. COMPARISONS.

Comparisons in BASIC are nothing unusual. Yet how can you produce them in machine language? Let's have a look at an example:

A = B (=) A − B = 0
As you can see, a comparison between two numbers (A & B) can easily be formed. For the computer, this form has the advantage of having a zero at the right side of the equation. The microprocessor can only determine the location of the zero; it must determine whether 0 is in the internal math register (known as the ACCUMULATOR) or not.
Thus, all the bits combine with one another through an OR operand—something like this:

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

If all 8 bits of the accumulator were 0, then the result of this chain of combinations would be 0; in all other cases (i.e., if at least one bit is 1), then the result would be 1. Therefore, the microprocessor tells whether the math register (where the result of the last question almost always stands) is equal or unequal to 0—voila, the first two comparisons are created. For a comparison such as A = B or A is unequal to B, we just subtract both numbers one after another, then establish whether the contents of the accumulator is 0.

For "greater than" and "less than", we take a similar route: After the subtraction, wee see whether the number in the accumulator is less than or greater than 0, distinguishable in the bit indicated:

A 〉 B (=) A − B 〉 0 (fulfilled when Bit 7 = 0) positive
A 〈 B (=) A − B 〈 0 (fulfilled when Bit 7 = 1) negative

13.8. MONITOR COMMANDS.

Now that we've laid the groundwork for machine language, we should put forth the individual commands used in programming with a machine language monitor. We'll keep the concept of putting the commnds in three subheadings. Have no fear if you don't understand the commands right at the beginning.
They will be explained in the following examples.

13.8.1. DATA MANIPULATION COMMANDS.

ADC: add with carry
This command adds a following operand and the carry bit to the number in the accumulator. If a transfer occurs, then this is stored in the carry bit.

There are two possibilities for this operand: Either the bits of the address given should be added; or the number should go directly into memory after the command.

Possible forms:
ADC $HH (add contents of location HH)
ADC #HH (add number HH to the accumulator)
SBC: subtract with carry

Like ADC, but used for subtraction.

Possible forms:
SBC $HH (subtract contents of HH)
SBC #HH (subtract HH)

AND: and with accumulator

Leads AND- combinations of accumulator contents through with operands. Carry bit will not be observed. Operand same as ADC.

Possible forms:
AND $HH
AND #HH

ORA: or with accumulator

Like AND, but for OR- combinations.

Possible forms:
ORA $HH
ORA #HH

EOR: exclusive or with accumulator

Like AND, but for Exclusive-OR- combinations.

Possible forms:
EOR $HH
EOR #HH
DEC: decrement
Diminishes byte at the given address by 1. If the result is 0, the zero bit will be set, as will the negative bit for negative numbers. The carry bit will not be altered, however.

Only DEC $HH is possible

DEX: decrement X

Like DEC, but it diminishes the X-register

Only DEX is possible

INC: increment

Like DEC, only it increases by 1

Only INC $HH is possible

INX: increment X

Like INC, but for the X-register

Only INX is possible

CLC: clear with carry

Clears the carry bit

Only CLC is possible

SEC: set carry

Sets carry bit to 1

Only SEC is possible

ASL: arithmetic shift left
Shifts all the bits in the accumulator one place to the left. Bit 7 will be shoved into the carry bit, while Bit 0 will be loaded with a 1.

Only ASL possible

LSR: logical shift right
Shifts all bits in the accumulator one place to the right. Bit 0 will be shifted to the carry bit, Bit 7 will be 0.

Only LSR is possible

13.8.2. JUMP COMMANDS.

JMP: jump

The program will go to the specified address.

Only JMP $HH possible

JSR: jump to subroutine

Call to a subroutine at the given address.

Only JSR $HH possible

RTS: return from subroutine

Returns from the subroutine, or goes to the monitor.

Only RTS possible

BCC: branch on carry clear

Branches to the given address when carry bit is 0.

Only BCC $HH possible

BCS: branch on carry set

Branches to the address given when the carry bit is 1.

Only BCS $HH possible

BEQ: branch on equal to zero
Branches when zero ag is 1. The zero ag indicates whether the result of the last operation was 0.

Only BEQ $HH possible

BNE: branch on not equal to zero

Branches when zero ag is 1.

Only BNE $HH possible

BMI: branch on minus
Branches if the negative ag is 1. The negative ag indicates whether the result of the last operation was less than 0.

Only BMI $HH possible

BPL: branch on plus

Branches when the negative ag is 0.

Only BPL $HH possible

13.8.3. DATA MOVEMENT.

LDA: load accumulator
Loads accumulator with the argument which follows. If the argument is an address, the accumulator will be the same value as the addressed byte. In the case of direct values, the accumulator will be loaded with the byte which follows.

Possible forms:
LDA $HH (load contents of address HH)
LDA #HH (load HH into the accumulator)

LDX: load X

Like LDA, only for the X-register.

Possible forms:
LDX $HH (load contents of address HH)
LDX #HH (load HH in the X-register)

STA: store accumulator

Store the contents of the accumulator in the given address.

Only STA $HH possible

STX: store X

Like STA, only for the X-register.

Only STA SHH possible
TAX: transfer accumulator into X

Load X with accumulator's contents.

Only TAX possible

TXA: transfer × into accumulator

Load accumulator with the contents of X.

Only TXA possible

13.9. THE SIMULATOR.

You will find a listing for a machine language simulator in Section 14. This will introduce you to m/l programming. Since it's written in BASIC, it is very very slow. However, those who have a compiler can compile it, which will speed it up somewhat.

When it is run, the simulator will be in Assembler Mode.
Now it can be given m/l commands. The input lines must be made in the upper third of the screen following these instructions:

ADDRESS COMMAND OPERAND
Single byte commands slip to the operand. A space should be left between individual instructions. If an error occurs, three question marks will appear at the right border of the input line. Pressing any key will dispose of the question marks, and the next command can be given.

Important register contents and bits are continually shown in the upper third of the screen. The last four bytes of the address range are shown in hexadecimal, binary and ASCII. Beneath that, you'll see the Program Counter (PC), Accumulator, X-register and different bits (Carry, Negative, Zero). The last display conveys the TRACE status (of/off).
All numbers are given in hexadecimal!
To enter command mode the left-arrow must be given as input. Then follow with the command characters, as described below.

The first of these is known as the C-command; it will clear the lower part of the screen. It is recommended that no keys such as CLR are used during command input, as the screen mask might be destroyed.

The next command is called T. This toggles the TRACE mode on and off.

G will start a "machine language" program at the given address.

D will disassemble (i.e., LIST) the program between two given addresses. Z will assign the given address byte a certain value.

13.10. THE FIRST PROGRAM.

Our first program in machine language will be adding and subtracting, since that's one of the best ways to learn the operation of a microprocessor. Let's begin with a program which adds two numbers.

We put both bytes to be added in advance in locations FF & FE. This can be done in direct mode by using the Z command; not a very comfortable method, but it will do for now. Therefore, you type the left-arrow, a Z, a space, the address (FF or FE), another space, the number desired (in hex) and RETURN. If three question marks appear at the right edge, the input was wrong. This can also be the case if the cursor travels to another line during input. If you've done everything properly, however, the value should be displayed in the register section. This is how you store the numbers to be added.

As you've probably already noticed, the beginning of the input line is always given as a hex address. It takes the address at which the assembler command will be stored if possible. Before we can do this, we still have to consider the format of these commands.

The first rule of assembler programming is: Almost all data manipulation takes place in the accumulator. From there, the accumulator will be loaded by the first command in the program, with the first number. LDA FF will serve our purposes. If the carry bit is still set, we must clear it with CLC.

Next comes the addition command proper, ADC FE. This command takes the second number in memory cell FE and adds it to the contents of the accumulator. The result of this addition is placed back into the accumulator.

To display the result, the command STA FD is conveyed to one of the four memory cells visible on the upper third of the screen. The addition is now done. All that's missing is a return to the assembler using RTS. If the carry bit is set during the addition, it will likewise register onscreen.

The entire program looks like this:

```
LDA $FF
CLC
ADC $FE
STA $FD
RTS
```

These commands should be stored in memory starting at 00. Put this address at the beginning of the input line, then move the cursor so that a space will be between the address and the command. Then type in the command and operand exactly as listed above. The command will be stored after you hit ⟨RETURN⟩. If a wrong address is typed in at the beginning of the line, just type in the proper address, and continue as usual.

The entire program is entered in this manner. You can disassemble (list) it anytime: To do so, enter a left arrow, D (for disassemble), beginning and ending addresses. If there is an error in the listing, just type in the command again.

The program can now be started with G 00 (don't forget the left-arrow). After the program runs, the register display and the cursor will reappear.

Should you want the display to reappear after each command, then the TRACE mode should be switched on. Each command will then run individually, and be displayed in the lower right-hand corner; and the Simulator will wait for a keypress. Thus, you can follow a program run closely.

13.11. THE SECOND STEP: 16-BIT ADDITION.

We bring about the handling of greater numbers using special algorithms. Below is a program that will add any 16-bit number to a constant. The numbers will again be assigned to memory cells FF (highbyte) and FE (lowbyte). When the number has 16 bits, two bytes and two addition procedures are used. First though, we begin as usual with

LDA $FE and
CLC.

In order to add a constant, we use ADC #lowbyte. This command adds the second number to the byte which follows the address, rather than holding it in one particular address.

After this command, the lowest-value half of the result is already complete. It is displayed by STA $FC. An eventual transfer is now stored in the carry bit, and will not be cleared by the loading of the second half (by LDA $FF). Now normal addition can be carried out with ADC #highbyte.

STA $FD displays the second half of the result; RTS ends the program. Here's the listing: 03E8 (=1000 in decimal) was chosen as the constant.

```
 0 LDA $FE
02 CLC
03 ADC #E8
05 STA $FC
07 LDA $FF
09 ADC #03
0B STA $FD
0D RTS
```

13.12. SUBTRACTION.

Since subtraction up to the carry bit has been covered in the section on Addition, all you will need is a listing of the program.

```
00 LDA $FF
02 SEC          (carry placed)
03 SBC $FE      (subtract)
05 STA $FD
07 RTS
```

13.13. MULTIPLICATION.

You'll recall from Chapter 13.6. that multiplication is represented as multiple addition. To put this knowledge to good use, we'll now write a machine language subroutine to perform multiplication.

Using the previous method, the addition program will be changed into a constant subroutine (which really isn't necessary, but we need the programming practice) which will be called by a loop. So, back to addition.

When we multiply two 8-bit numbers, the result will be 16-bit; therefore, tha addition routine must add a single byte to a 2-byte number. This can be simplified if you think of an 8-bit number with 0 as the highbyte, and perform a normal 16-bit addition. Then, the transfer of the highbyte is guaranteed in the result. It looks like this:

```
LDA $FE         (lowbyte of the result)
CLC
ADC $FC         (add 8-bit number)
STA $FE         (back into the stack)
LDA $FF         (highbyte of the result)
ADC #00         (add 0 and carry bit)
STA $FF
(into the stack)

RTS             (return)
```

The 8-bit number previously stood in FC; after the addition, the result goes in FE/FF. Now all we need is the loop. The length of the loop will be given through the multiplier which was stored previously in FD by the Z command.

The simplest method of programming a loop with variable length is to use a special register that decreases by 1 step during each run-through. When the register reaches 0, the loop ends. The x-register is best-suited for this; to begin the loop, use LDX $FD (FD contains the multiplier).

Next, the subroutine follows, called by JSR $Addition (Addition should be set up with a JMP command). The loop number will decrement after each pass of the subroutine.
This is handled by the single-byte command DEX. The peculiarity of DEX is that it will also alter the Z and N bits, then it will show whether the contents of the x-register are null or negative. The register bits do not simply cover the accumulator.

Table 2 shows which command can affect which bits. When a branch condition occurs, the control bits on hand execute the so-called branch commands. We find such a condition in our own loop, which should break when x=0. If X is unequal to 0, then a return should follow; that is the purpose of BNE $Address (branch if not equal to zero). If the zero bit is 1, then this means that the result of the last operation was 0, it is made unequal to 0 when Z=0. If the zero bit is 0, then it points to the next address, and goes on with the next command. You can follow the program ow with the TRACE mode and the Program Counter (PC). They tell the processor where the next command stands. After a branch command, you can read this result in PC as well.

The branch commands on a genuine 6502-6510 assembler are simplified somewhat, as jump addresses normally aren't used; rather, just the distance to the next command is given (e.g., 3 forward, or 20 back).

Before the loop begins, we must remember to clear both "answer bytes". Have a look at the listing:

```
00 LDA #00
02 STA $FF (clear FF)
04 STA $FE (clear FE)
06 LDX $FD (load X)
08 JSR $0E (call subroutine)
0A DEX       (decrement X)
0B BNE $08 (branch if not equal to zero)
0D RTS       (end of main program)
0E LDA $FE (addition subroutine)
10 CLC
11 ADC $FE
13 STA $FE
15 LDA $FF
17 ADC #00
19 STA $FF
1B RTS (end of subroutine)
```

COMMAND	*	C	N	Z	COMMAND	*	C	N	Z
ADC	*	X	X	X	LDA	*		X	X
AND	*		X	X	LDX	*		X	X
ASL	*	X	X	X	LSR	*	X	X	X
CLC	*	X			ORA	*		X	X
DEC	*		X	X	SBC	*	X	X	X
DEX	*		X	X	SEC	*	X		
EOR	*		X	X	TAX	*		X	X
INC	*		X	X	TXA	*		X	X
INX	*		X	X					

TABLE 2. Guide bits influenced

13.14. OTHER POSSIBILITIES.

There you have the basic principles of assembly language. It is redundant to say that "real" machine language offers many more possibilities. For example, you can load parts of memory with different types of addresses, such as one- dimensional arrays, special commands to make individual interrupt routines possible, and much more.

Therefore, this section shall cover a few techniques which you will always encounter.

We'll start with the shift command. It shifts the accumulator bits one place to the left or right. It can be used easily to double or halve the accumulator contents; a left shift will double the byte (if you did so with a decimal number, it would be multiplied by 10), while the byte is halved by doing the opposite. Please note the shift scheme in Figures 5 and 6.

FIG. 5 ASL BIT 0 LOADED WITH 0

C 7 6 5 4 3 2 1 0 Accumulator bits

FIG. 6 LSR BIT 7 LOADED WITH 0

7 6 5 4 3 2 1 0 C Accumulator bits

You already know the next command from BASIC. Along with AND, OR and EOR, data can also be combined with the accumulator by using ADC. This allows for the noted bit- mapping techniques covered earlier, but in machine language.

The INX command is a close relative of DEX. It works basically the same, only the X-register is increased by 1, rather than decreased. This is called incrementing. Now you should also be able to explain DEC and INC: They work directly on memory cells rather than the X-register.

All of these commands are alike in that a transfer of the place is not registered in the carry bit: If a byte is to be incremented past FF, then the next INC value will be 00. Needless to say, this means these commnads are hardly suited for arithmetic.

TAX and TXA still remain. These will set the accumulator and X-register so that the two match values. TAX loads the X-register with accumulator contents, while TXA does the opposite.

Don't hold back from experimenting with machine language&
The Simulator has an advantage over "real" m/l in that it will never lock-up. Every program done on the Simulator can be stopped with the RUN/STOP key.

13.15. HOW DO SYS-EXTENSIONS WORK?

It should at least be explained in passing how to program such extended commands as SYS ADDRESS, DATUM.

After a SYS-command, the internal BASIC program pointer (similar to the Program Counter) points to the byte after the address. A normal SYS-command would return the interpreter to us from the m/l routine, after a syntax check.

However, we can change the interpreter (by means of JSR Subroutine) to read data after the next comma, colon or end-of-line into the oating accumulator (which is a set of reserved cells on zeropage, rather than a processor register). The BASIC program pointer will then point to the byte after the data. If the data were incorrect, the ROM- routine would quit with an error. As long as normal numeric values are used, things will run correctly. Also, different subroutines from interpreter-ROM are at our disposal. If data such as integers or addresses are to be used, then a further ROM routine will be needed to convert the numbers in the floating accumulator into the desired format. These can be held in specified places in memory by our machine language programs, and even be used for other functions:
That's the whole secret.

Command extensions such as SIMON'S BASIC or GRAPHICS 64 have further perfected this system. Here the routine (which is kept up-to-date by the commands) extends to a special section where the extension is decoded, and branches into the specified subroutine.

14. APPENDIX : PROGRAM LISTINGS.

M/L SIMULATOR

```
1000 REM================================
1001 REM MACHINE LANGUAGE SIMULATOR 1.0.

1002 REM COPYRIGHT 1984 BY
1003 REM HANS JOACHIM LIESERT
1004 REM PRODUCT OF DATA BECKER &
1005 REM ABACUS SOFTWARE
1006 REM================================
1010 REM================================
1011 REM INIT
1012 REM================================.
1020 PRINTCHR$(142);CHR$(8):POKE788, 52
1030 PRINT"[CLEAR][CMDR-A][38 SHIFT-*][CMDR-S]";
1040 PRINT"[SHIFT-MINUS] REG    HEX    BIN
     ASC [SHIFT-MINUS]";
1050 PRINT"[SHIFT-MINUS]FF/FE=
     [SHIFT-MINUS]";
1060 PRINT"[SHIFT-MINUS]FD/FC=
     [SHIFT-MINUS]";
1070 PRINT"[CMDR-Q][38 SHIFT-*][CMDR-W]";
1080 PRINT"[SHIFT-MINUS]PC=    AC=    XR=C=N=
     qaZ=    T:    [SHIFT-MINUS]";
1090 PRINT"[CMDR-Z][38 SHIFT-*][CMDR-X]";
1100 PRINT"[40 SHIFT-*]"
1110 PRINT"[15 SHIFT-*][CMDR-R][9 SHIFT-*][CMDR-R]
     [14 SHIFT-*]";
1120 PRINT ADR MNE[6 SPACES][SHIFT-MINUS] CODES
     [SHIFT-MINUS] MESSAGES"
1130 FORI=1TO13:PRINT"[15 SPACES][SHIFT-MINUS][9
     SPACES][SHIFT-MINUS]":NEXT
1140 PRINT"[15 SPACES][SHIFT-MINUS][9 SPACES][SHIFT-MINUS]";
1150 DIMB(255), C(255), H$(255), EB$(34), EB(34),
     P1$(10), P2$(10), P3$(10)
1160 FORI=0TO255:READH$(I):B(I)=255:C(I)=255:NEXTI
1170 FORI=0TO34:READEB$(I), EB(I):NEXTI
1180 FORI=0TO10:P1$(I)="[14 SPACES]":P2$(I)=
     "[5 SPACES]":NEXT
1997 REM================================
1998 REM ASSEMBLER
1999 REM================================
2000 GOSUB 4000
```

```
2010 POKE214, 8:POKE211, 0:SYS58732
2015 IFAD=256THENAD=0
2020 PRINT"[2 CRSR DN]";H$(AD);"[3 CRSR LFT]";
2030 INPUT"[2 CRSR DN][3 CRSR LFT]";IN$
2040 IFLEFT$(IN$, 1)="[LEFT-ARROW]"THEN 2200
2050 PO=1:GOSUB4400:IFEFTHEN4500
2060 AD=O:A$=MID$(IN$, 4, 3)+" ":FL=×-1
2070 FORI=0TO8:IFA$=EB$(I)THENFL=I
2080 NEXTI:IFFL<>-1THENB(AD)=EB(FL):C(AD)=FL:AD=AD+1:
     GOTO2000
2090 A$=MID$(IN$, 4, 5)
2100 FORI=9TO34:IFA$=EB$(I)THENFL=I
2110 NEXTI:IFFL=-1ORAD=255THEN4500
2120 B(AD)=EB(FL):C(AD)=FL
2130 PO=9:GOSUB4400:IFEFTHEN4500
2140 AD=AD+1:B(AD)=O:C(AD)=O
2150 AD=AD+1:GOTO2000
2197 REM==================================
2198 REM DIRECT COMMANDS
2199 REM==================================
2200 A$=MID$(IN$, 2, 1)
2210 IFA$="T"THENT=1-T:GOTO2000
2220 IFA$="Z"THEN2300
2230 IFA$="D"THEN2400
2240 IFA$="G"THEN2500
2250 IFA$="C"THEN4600
2260 GOTO4500
2300 PO=4:GOSUB4400:IFEFTHEN4500
2310 AD=O:PO=7:GOSUB4400:IFEFTHEN4500
2320 B(AD)=O:FL=-1
2330 FORI=0TO34:IFO=EB(I)THENFL=I
2340 NEXTI:IFFL<>-1THENC(AD)=FL:GOTO2000
2350 C(AD)=O:GOTO2000
2397 REM==================================
2398 REM DISASSEMBLER
2399 REM==================================
2400 PO=4:GOSUB4400:IFEFTHEN4500
2410 AD=O:PO=7:GOSUB4400:IFEFTHEN4500
2420 IFAD>OTHEN4500
2430 H1$=H$(AD):H3$=" ":H4$=H$(B(AD))
2440 IFC(AD)>34THENH2$="??? ":GOTO2470
2450 H2$=EB$(C(AD))
2460 IFC(AD)>8THENAD=AD+1:H3$=H$$(B(AD))
2470 AD=AD+1:GOSUB4300
2480 IFAD<=OTHEN2430
2490 GOTO2000
```

```
2497 REM=====================================
2498 REM PROGRAM RUN
2499 REM=====================================
2500 PO=4:GOSUB4400:IFEFTHEN4500
2510 PC=O
2520 CO=C(PC):AD=PC:PC=PC+1:IFCO〉34THEN3300
2530 IFCO〈9THENONCO+1GOTO2700, 2720, 2730, 2770, 2790, 2810, 2830, 2840,
       2850
2540 CO=CO-8: IFCO〈9THENONCOGOTO2860, 2880, 2900,
     2910, 2920, 2940, 2960, 2980
2550 CO=CO-8: IFCO〈9THENONCOGOTO3000, 3020, 3040,
     3080, 3100, 3120, 3140, 3150
2560 CO=CO-8: IFCO〈9THENONCOGOTO3160, 3170, 3180,
     3190, 3200, 3210, 3220, 3240
2570 ONCO-8GOTO3260, 3270
2590 N=0:IFA〉127THENN=1
2600 Z=0:IFA=0THENZ=1
2610 IFPEEK(203)=63THEN3400
2615 IFT=0THEN2520
2620 GOSUB4000:FORI=0TO9:P3$(I)=P3$(I+1):NEXTI
2630 P3$(10)=H$(AD)+" "+EB$(C(AD))
2640 IFC(AD)〉8THENP3$(10)=P3$(10)+H$(C(AD+1)):GOTO2660
2650 P3$(10)+P3$(10)+" "
2660 POKE214, 12:POKE211, 0:SYS58732
2670 FORI=0TO10:PRINTSPC(27);P3$(I):NEXTI
2680 POKE198, 0:WAIT198, 1:GET IN$:GOTO2520
2696 REM=====================================
2697 REM MAJOR COMMANDS
2698 REM=====================================
2699 REM ASL
2700 A=2*A:C=0:IFA〉255THENA=A-256:C=1
2710 GOTO2590
2719 REM CLC
2720 C=0:GOTO2610
2729 REM DEX
2730 X=X-1:IFX〈0THENX=X+256
2740 Z=0:IFX=0THENZ=1
2750 N=0:IFX〉127THENN=1
2760 GOTO2610
2769 REM INX
2770 X=X+1:IFX〉255THENX=X-256
2780 GOTO 2740
2789 REM LSR
2790 A=A/2:C=0:IFINT(A)〈〉0THENA=INT(A):C=1
2800 GOTO2590
2809 REM RTS
```

```
2810 IFS=0THEN2000
2820 PC=S(S):S=S-1:GOTO2610
2829 REM SEC
2830 C=1:GOTO2610
2839 REM TAX
2840 X=A:GOTO2740
2849 REM TXA
2850 A=X:GOTO2590
2859 REM ADC #
2860 A=A+B(PC)+C:C=0:IFA>255THENA=A-256:C=1
2870 PC=PC+1:GOTO2590
2879 REM ADC $
2880 A=A+B(C(PC))+C:C=0:IFA>255THENA=A-256:C=1
2890 PC=PC+1:GOTO2590
2899 REM AND #
2900 A=A AND B(PC):PC=PC+1:GOTO2590
2909 REM AND $
2910 A=A AND B(C(PC)):PC=PC+1:GOTO2590
2919 REM BCC $
2920 IFC=
0THENPC=B(PC):GOTO2610
2930 PC=PC+1:GOTO2610
2939 REM BCS $
2940 IFC=1THENPC=B(PC):GOTO2610
2950 PC=PC+1:GOTO2610
2959 REM BEQ $
2960 IFZ=1THENPC=B(PC):GOTO2610
2970 PC=PC+1:GOTO2610
2979 REM BMI $
2980 IFN=1THENPC=B(PC):GOTO2610
2990 PC=PC+1:GOTO2610
2999 REM BNE $
3000 IFZ=0THENPC=B(PC):GOTO2610
3010 PC=PC+1:GOTO2610
3019 REM BPL $
3020 IFN=0THENPC=B(PC):GOTO2610
3030 PC=PC+1:GOTO2610
3039 REM DEC $
3040 H=B(C(PC)):H=H-1:IFH<0THENH=H+256
3050 Z=0:IFH=0THENZ=1
3060 N=0:IFH>127THENN=1
3070 B(C(PC))=H:C(C(PC))=H:PC=PC+1:GOTO2590
3079 REM EOR #
3080 H=A OR B (PC):A=A AND B(PC)
3090 H=NOT(A):A=H AND A:PC=PC+1:GOTO2610
3100 H=A OR B(C(PC)):A=AAND B(PC)
```

```
3110 GOTO3090
3119 REM INC $
3120 H=B(C(PC)):H=H+1:IFH〉255THENH=H-256
3130 GOTO3050
3139 REM JMP $
3140 PC=B(PC):GOTO2610
3149 REM JSR $
3150 S=S+1:S(S)=PC+1:PC=B(PC):GOTO2610
3159 REM LDA #
3160 A=B(PC):PC=PC+1:GOTO2590
3169 REM LDA $
3170 A=B(C(PC)):PC=PC+1:GOTO2590
3179 REM LDX #
3180 X=B(PC):PC=PC+1:GOTO2740
3189 REM LDX $
3190 X=B(C(PC)):PC=PC+1:GOTO2740
3199 REM ORA #
3200 A=A OR B(PC):PC=PC+1:GOTO2590
3209 REM ORA $
3210 A=A OR B (C(PC)):PC=PC+1:GOTO2590
3219 REM SBC #
3220 A=A-B(PC)-1+C:C=1:IFA〈0THENA=A+256:C=0
3230 PC=PC+1:GOTO2590
3239 REM SBC $
3240 A=A-B(C(PC))-1+C:C=1:IFA〈0THENA=A+256:C=0
3250 PC=PC+1:GOTO2590
3259 REM STA $
3260 B(C(PC))=A:C(C(PC))=A:PC=PC+1:GOTO2610
3269 REM STX $
3270 B(C(PC))=X:C(C(PC))=X:PC=PC+1:GOTO2610
3296 REM=====================================
3297 REM RUNTIME ERROR
3298 REM=====================================
3300 POKE214, 8:POKE211, 0:SYS58732:
     PRINT"BAD CODE ERROR IN ";H$(PC-1)
3310 POKE198, 0:WAIT198, 1:GETIN$:GOTO2000
3396 REM=====================================
3397 REM BREAK
3398 REM=====================================
3400 POKE214, 8:POKE211, 2:SYS58732:
     PRINT"BREAK IN ";H$(PC)" ";
3410 POKE198, 0:WAIT198, 1:GETIN$:GOTO2000
3997 REM=====================================
3998 REM REGISTER INDICATOR
3999 REM=====================================
4000 H$=H$(B(255))+" "+H$(B(254))
```

```
4010 POKE214, 2:POKE211, 9:SYS58732:PRINTH$;" ";
4020 H$="":FORJ=255TO254STEP-1:H=B(J)
4030 FORI=7TO0STEP-1:IF(2&IANDH)THENH$=H$+"1":GOTO4050
4040 H$=H$+"0"
4050 NEXTI:H$=H$+" ":NEXTJ
4060 PRINTH$;" ";
4070 H1$=CHR$(B(255)):
     IFB(255)<32ORB(255)>127ANDB(255)<160THENH1$=" "
4080 H2$=CHR$(B(254)):
     IFB(254)<32ORB(254)>127ANDB(254)<160THENH2$=" "
4090 PRINTH1$;" ";H2$
4100 H$=H$(B(253))+" "+H$(B(252))
4110 POKE214, 3:POKE211, 9:SYS58732:PRINTH$;" ";
4120 H$="":FORJ=253TO252STEP-1:H=B(J)
4130 FORI=7TO0STEP-1:IF(2&IANDH)THENH$=H$+"1":GOTO4150
4140 H$=H$+"0"
4150 NEXTI:H$=H$+" ":NEXTJ
4160 PRINTH$;" ";
4170 H1$=CHR$(B(253)):
     IFB(253)<32ORB(253)>127ANDB(253)<160THENH1$=" "
4180 H2$=CHR$(B(252)):
     IFB(252)<32ORB(252)>127ANDB(252)<160THENH2$=" "
4190 PRINTH1$;" ";H2$
4200 POKE214, 5:POKE211, 4:SYS58732:PRINTH$(PC);
4210 PRINT"[4 CRSR RT]";H$(A);"[4 CRSR RT]";H$(X);
     "[3 CRSR RT]";
4220 PRINTMID$(STR$(C), 2, 1);"[3 CRSR RT]";
     MID$(STR$(N), 2, 1);"[2 CRSR RT]";
4230 PRINTMID$(STR$(Z), 2, 1);"[5 CRSR RT]";
4240 IFTTHENPRINT"ON ";:RETURN
4250 PRINT"OFF";:RETURN
4297 REM====================================
4298 REM COMMAND INPUT
4299 REM====================================
4300 FORP=0TO09:P1$(P)=P1$(P+1):P2$(P)=P2$(P+1):NEXTP
4310 P1$(10)=H1$+" "+H2$+H3$+" "
4320 P2$(10)=H4$+" "+H3$
4330 POKE214, 12:POKE211, 0:SYS58732
4340 FORP=0TO10:PRINT"[CRSR RT]";P1$(P);"[2 CRSR RT]"
     ;P2$(P):NEXTP:RETURN
4397 REM====================================
4398 REM GET AN ARGUMENT
4399 REM====================================
4400 I1=ASC(MID$(IN$, PO, 1)):I2=ASC(MID$(IN$, PO+1, 1)):EF=0
4410 IFNOT(I1>47ANDI1<58ORI1>64ANDI1<71)THENEF=1:RETURN
4420 IFNOT(I2>47ANDI2<58ORI2>64ANDI2<71)THENEF=1:RETURN
```

```
4430 I1=I1-48:IFI1⟩9THENI1=I1-7
4440 I2=I2-48:IFI2⟩9THENI2=I2-7
4450 O=I1=16+I2:RETURN
4497 REM=====================================
4498 REM ERROR
4499 REM=====================================
4500 POKE241, 8:POKE211, 36:SYS58732
4510 PRINT"???";:POKE198, 0:WAIT198, 1:GETIN$
4520 PRINT"[3 CRSR LFT] ";:GOTO2000
4597 REM=====================================
4598 REM CLEAR SCREEN
4599 REM=====================================
4600 POKE214, 12:POKE211, 0:SYS58732
4610 FORI=0TO10:P1$(I)="[14 SPACES]":P2$(I)="[6 SPACES]":
     P3$(I)="[10 SPACES]"
4620 PRINT"[CRSR RT]";P1$(I);"[2 CRSR RT]";
     P2$(I);"[4 CRSR RT]";P3$(I):NEXTI:GOTO2000
7997 REM=====================================
7998 REM HEXADECIMAL TABLE
7999 REM=====================================
8000 DATA "00", "01", "02", "03", "04", "05", "06", "07", "08", "09"
8005 DATA "0A", "0B", "0C", "0D", "0E", "0F"
8010 DATA "10", "11", "12", "13", "14", "15", "16", "17", "18", "19"
8015 DATA "1A", "1B", "1C", "1D", "1E", "1F"
8020 DATA "20", "21", "22", "23", "24", "25", "26", "27", "28", "29"
8025 DATA "2A", "2B", "2C", "2D", "2E", "2F"
8030 DATA "30", "31", "32", "33", "34", "35", "36", "37", "38", "39"
8035 DATA "3A", "3B", "3C", "3D", "3E", "3F"
8040 DATA "40", "41", "42", "43", "44", "45", "46", "47", "48", "49"
8045 DATA "4A", "4B", "4C", "4D", "4E", "4F"
8050 DATA "50", "51", "52", "53", "54", "55", "56", "57", "58", "59"
8055 DATA "5A", "5B", "5C", "5D", "5E", "5F"
8060 DATA "60", "61", "62", "63", "64", "65", "66", "67", "68", "69"
8065 DATA "6A", "6B", "6C", "6D", "6E", "6F"
8070 DATA "70", "71", "72", "73", "74", "75", "76", "77", "78", "79"
8075 DATA "7A", "7B", "7C", "7D", "7E", "7F"
8080 DATA "80", "81", "82", "83", "84", "85", "86", "87", "88", "89"
8085 DATA "8A", "8B", "8C", "8D", "8E", "8F"
8090 DATA "90", "91", "92", "93", "94", "95", "96", "97", "98", "99"
8095 DATA "9A", "9B", "9C", "9D", "9E", "9F"
8100 DATA "A0", "A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8", "A9"
8105 DATA "AA", "AB", "AC", "AD", "AE", "AF"
8110 DATA "B0", "B1", "B2", "B3", "B4", "B5", "B6", "B7", "B8", "B9"
8115 DATA "BA", "BB", "BC", "BD", "BE", "BF"
8120 DATA "C0", "C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9"
8125 DATA "CA", "CB", "CC", "CD", "CE", "CF"
```

```
8130 DATA "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7", "D8", "D9"
8135 DATA "DA", "DB", "DC", "DD", "DE", "DF"
8140 DATA "E0", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8", "E9"
8145 DATA "EA", "EB", "EC", "ED", "EE", "EF"
8150 DATA "F0", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9"
8055 DATA "FA", "FB", "FC", "FD", "FE", "FF"
8197 REM=====================================
8198 REM COMMAND TABLE
8199 REM==============================
8200 DATA "ASL ", 10, "CLC ", 24, "DEX ", 202, "INX ",
     232, "LSR ", 74, "RTS ", 96
8201 DATA "SEC ", 56, "TAX ", 170, "TXA", 138
8203 DATA"ADC #", 105, "ADC $", 101, "AND #", 41, "AND $", 37
8205 DATA"BCC $", 144, "BCS $", 176, "BEQ $", 240
8215 DATA"BMI $", 48, "BNE $", 208, "BPL $", 16
8220 DATA"DEC $", 198, "EOR #", 73, "EOR $", 69
8225 DATA"INC $", 230, "JMP $", 76, "JSR $", 32
8230 DATA"LDA #", 169, "LDA $", 165, "LDX #", 162, LDX $", 166
8235 DATA"ORA #", 9, "ORA $", 5
8240 DATA"SBC #", 233, "SBC $", 229, "STA $", 133
8245 DATA"STX $", 134
```

ROAD RACE

```
1 REM==================================
2 REM======== ROAD RACE ==============
3 REM==================================
10 PRINT"[CLEAR]":POKE53280, 0:POKE53281, 0:V=53248:     REM SCREEN
        PREPARATION
20 FORI=832TO894:READA:POKEI, A:NEXT:
   REM READ AUTO SPRITE DATA
30 FORI=896TO958:READA:POKEI, A:NEXT:
   REM READ CRASH SPRITE DATA
40 POKE2040, 13:POKE2041, 13:POKEV+39, 1:POKEV+40, 2:
   REM SPRITE POINTERS & COLORS
50 FORI=0TO24:POKE1036+U=40, 160:POKE55308+I=40, 1
60 POKE1051+I=40, 160:POKE55323+I=40, 1:NEXTI:REM ROAD $ COLOR
70 POKEV, 168:POKEV+1, 170:POKEV+21, 3:X=168:
   REMSTART PO.CAR & SPRITES
80 POKEV+2, 168:POKEV+3, 0:HX 168:HY=0:
   REM START POS. BACKGROUND
90 POKEV+30, 0:POKEV+31, 0:REM CLEAR COLLISION CONTROL
100 A=PEEK(203):REM READ KEYBOARD
```

```
110 IFA=12THENX=X-1:REM Z PRESSED
120 IFA=55THENX=X+1:REM / PRESSED
130 POKEV, X:REM MOVE CAR
140 IFPEEK(V+30)〈〉0ORPEEK(V+31)〈〉0THENPOKE2040, 14:
    FORI=0TO500:NEXT:RUN:REM CRASH
150 HY=HY+2:IFHY〉240THENHY=30: REM MOVEMENT BELOW
160 HX=HX+INT(RND(TI)=5)-2:IFHX〈120THENHX=120:
REM COORDINATES & LEFT BORDER
170 IFHX〉216THENHX=216:REM RIGHT BORDER
180 POKEV+2, HX:POKEV+3, HY:GOTO100:REM BACKGROUND MOVE-
          MENT
1000 REM AUTO DATA
1100 DATA0, 0, 0
1101 DATA0, 126, 0
1102 DATA0, 126, 0
1103 DATA0, 255, 0
1104 DATA12, 255, 48
1105 DATA15, 255, 240
1106 DATA12, 255, 48
1107 DATA0, 255, 0
1108 DATA0, 255, 0
1109 DATA1, 231, 128
1110 DATA1, 195, 128
1111 DATA1, 195, 128
1112 DATA1, 195, 128
1113 DATA3, 195, 192
1114 DATA3, 195, 192
1115 DATA115, 255, 206
1116 DATA115, 255, 206
1117 DATA127, 255, 254
1118 DATA115, 255, 206
1119 DATA115, 255, 206
1120 DATA0, 0, 0
2000 REM SPRITE-DATA CRASH
2100 DATA123, 20, 0
2101 DATA0, 24, 0
2102 DATA30, 44, 77
2103 DATA21, 126, 3
2104 DATA240, 125, 48
2105 DATA15, 205, 240
2106 DATA22, 155, 41
2107 DATA1, 205, 156
2108 DATA0, 155, 0
2109 DATA1, 201, 108
2110 DATA1, 105, 108
2111 DATA1, 95, 28
```

```
2112 DATA1, 155, 192
2113 DATA23, 115, 192
2114 DATA32, 242, 132
2115 DATA35, 239, 216
2116 DATA1, 95, 28
2117 DATA32, 242, 132
2118 DATA68, 155, 0
2119 DATA27, 125, 48
2120 DATA0, 0, 0
```

15. EXPLANATIONS OF SPECIAL SYMBOLS.

From time to time, you'll find a little mark in the text.
This mark, which looks like this ("&"), can be found on your Commodore keyboard to the left of the RESTORE key (the up- arrow key); it is used in exponentiation.

In the few listings that require Commodre characters, these characters have been replaced with bracketed descriptions that are much easier to read than standard Commodore printouts. A few examples:
[9 SPACES] means hit the spacebar nine times.

[SHIFT−minus]—hold shift and press the minus sign.

[CRSR−DN] equals cursor down.

[CMDR−A] means hold the Commodore key (C) and press A.

[CLEAR] is the CLEAR/HOME key [SHIFT−*] tells you to hold shift and strike the asterisk key.

16. MEMORY MAP.

Location	*Contents*
0	on-chip Data Direction Register
1	on-chip Input/Output Register
2	unused
3/4	vector for conversion of floating—integer
5/6	Jump Vector: Convert Integer—Floating
7	search character

Location	*Contents*
8	quote-mode flag
9	column from last TAB
10	flag for last load command; 0= load, 1=verify
11	input buffer pointer—dimensions
12	DIM-flag
13	variable types; $FF=string, $00=number
14	$80=integer, $00= floating
5	flag for quote mode, LIST, garbage collection
16	FNx flag
17	input: 00=INPUT, 40=GET, 98=READ
18	TAN flag—last comparison result: 1=greater, 2=equal, 4=less
19	INPUT prompt flag
20/21	integers, e.g. addresses, FRE(0)
22	vector for string stack
23/24	pointer for last string
25-33	string stack
34-37	utility pointers
38-42	arithmetic register
43/44	pointer to start-of-BASIC
45/46	pointer to start of variables
47/48	pointer to start of BASIC arrays
49/50	pointer to end of BASIC arrays
51/52	pointer to the start of string storage
53/54	string pointer
55/56	pointer to end of BASIC memory
57/58	present BASIC line number
59/60	previous BASIC line number
61/62	pointer to next command for CONT
63/64	current DATA line
65/66	pointer to next DATA statement
67/68	pointer for last INPUT/DATA/GET routine
69/70	current variable name (2 characters)
71/72	pointer to current variable data
73/74	pointer to current FOR/NEXT variable
75/76	register for BASIC program lines
77	register for comparisons
78/79	pointer for FNx
80-83	register for strings
84-86	jump vector for functions
87-91	floating accumulator # 3

Location	*Contents*
92-96	floating accumulator # 4
97-101	floating accumulator # 1
102	sign; floating accumulator #1
103	counter for polynomial values
104	overflow for accumulator #1
105-109	floating acclumulator #2
110	sign for floating accumulator #2
111	comparison register between accumulators #1 and #2
112	overflow byte
113-114	pointer for polynomial values
115-138	CHRGET routine —gets next byte of BASIC Text
139-143	last RND value
144	status (like variable ST)
145	flags for keyboard column 1
146	time constant for cassette operating system
147	0=LOAD, 1=VERIFY
148	flag for serial bus
149	character for serial bus
150	flag for end-of-tape (end of cassette)
151	temporary storage of data
152	number of files open
153	current input device (normal: 0)
154	current output device (CMD, normal: 3)
155	parity byte for cassette operating system
156	flag for tape byte received
157	output mode (128= direct, 0= program)
158	checksum for cassette operating system
159	error correction for cassette operating system
160-162	clock
163	bit counter for serial output
164	counter for tape operation
165	tape-write counter
166	pointer in cassette buffer
167-171	flags for tape operation
172/173	pointer for cassette buffer
174/175	pointer to end-of-program (LOAD/SAVE)
176/177	time constant for cassette
178/179	pointer to start of cassette buffer
180	bit counter (cassette)

Location	*Contents*
181	next bit to be sent by RS-232
182	byte sent out by RS-232
183	length of current filename
184	current logical file number
185	current secondary address
186	current device number (e.g., 8 = disk drive)
187/188	pointer to file name
189	register for serial output
190	block counter for cassette operations
191	word buffer for serial output
192	cassette motor flag
193/194	starting address for LOAD/SAVE
195/196	ending address for LOAD/SAVE
197	key(s) pressed (normal=64)
198	number of keypresses in buffer
199	flag for RVS
200	input line end (pointer)
210/202	pointer for input cursor (line, column)
203	see location 197
204	flag for cursor (0=blinking)
205	number for blink time
206	character under cursor
207	cursor blink flag
208	flag for input from keyboard
209/210	pointer to current screen line
211	cursor column
212	cursor status (program mode/direct mode)
213	length of screen line (40/80)
214	cursor line number
215	last key pressed
216	number of INSERTS
217-242	high byte of beginning of line
243/244	cursor position in colour RAM
245/246	pointer for keyboard decoder table
247/248	input buffer pointer for RS-232
249/250	output buffer pointer for RS-232
251-254	free memory for operating system
255	beginning of BASIC memory (* 64)
256-511	processor stack
256-266	temporary storage for format conversion
256-318	correction factor for tape errors

Location	*Contents*
512-600	BASIC input buffer
601-610	logical file numbers
611-620	device numbers
621-630	secondary addresses
631-640	keyboard buffer
641/642	pointer to BASIC RAM—start
643/644	pointer to BASIC RAM—end
645	flag for timeout on serial bus
646	current character colour
647	background colour under cursor
648	high byte of top of screen memory
649	maximum length of keyboard buffer
650	flag for repeat (0=normal, 128=all, 127=off)
651	repeat speed
652	repeat delay
653	flag for SHIFT, Commodore and CTRL keys
654	see 653
655/656	pointer to keyboard decoder table
657	flag for changing character sets
658	scrolling flag
659	control register for RS-232
660	command register for RS-232
661/662	bits per second
663	RS-232 status register
664	number bits remaining for RS-232 transmission
665/666	baud rate for RS-232
667	pointer to byte received from RS-232
668	RS-232 input pointer
669	pointer to byte transmitted from RS-232
670	RS-232 output pointer
671/672	IRQ storage during tape use
673	CIA 2 NMI flag
674	timer A—CIA 1
675	CIA 1 interrupt flag
676	flag for timer A
677	current screen line
678-767	free RAM
704-766	sprite block 11
768/769	error information pointer
770/771	BASIC warm start pointer
772/773	BASIC tokenizer pointer

Location	*Contents*
774/775	BASIC LIST pointer
776/777	error execution pointer
778/779	BASIC token reading
780	accumulator storage
781	X-register storage
782	Y-register storage
783	P-register storage
784-787	USR jump (address in 785/786)
788/789	hardware-interrupt pointer
790/791	BRK-interrupt pointer
792/793	NMI pointer
794/795	OPEN pointer
796/797	CLOSE pointer
798/799	character input pointer
800/801	character output pointer

Location	*Contents*
802/803	KERNAL clear routine pointer (CLRCHN)
804/805	input pointer
806/807	output pointer
808/809	STOP key pointer
810/811	GET pointer
812/813	KERNAL close routine pointer (CLALL)
814/815	pointer for user-defined IRQ
816/817	LOAD pointer
818/819	SAVE pointer
820-827	free RAM
828-1019	cassette buffer
832-894	sprite block 13
896-958	sprite block 14
960-1022	sprite block 15
1023	free
1024-2023	VIDEO RAM
2024-2039	free
2040-2047	sprite pointer
2048-40960	BASIC memory
8192-16192	bit-map for high-resolution graphics
40960-49151	BASIC interpreter ROM
49152-53247	4 K RAM for m/l programs
53248-57343	character generator
53248-53294	VIC-II register

Location	*Contents*
53295-54271	977 bytes empty
54272-54300	SID registers
54301-55295	995 bytes empty
55296-56295	colour RAM
56296-56319	24 bytes free
56320-56335	CIA 1 registers
56320/56321	joysticks and keyboard reading
56336-56575	240 bytes empty
56576-56591	CIA 2 registers
56577 & 56579	user port registers
56592-57343	752 bytes empty
57334-65535	operating system ROM

17. INDEX.

A
Accumulator ..... 13.7.
ADC ..... 13.8.1.
Addition program ..... 13.10.
Addition program (16-bit) ..... 13.11.
AD converter ..... 10.2.
AND ..... 1.4.3., 13.8.1.
Animation ..... 7.4.
Attack ..... 8.1., 8.2.
Assembler mode ..... 13.9.

BASIC
—extensions ..... 12.7.
—input buffer ..... 1.2., 12.1.
—line production ..... 12.1.
Bar graphics ..... 5.2.
BCC ..... 13.8.2.
BCS ..... 13.8.2.
BEQ ..... 13.8.2.
Binary
—addition ..... 13.4.
—arithmetic ..... 1.4.3.
—subtraction ..... 13.5.
Bit-map ..... 6.2..
Block graphics ..... 5.1.
BMI ..... 13.8.2.
BNE ..... 13.8.2.
Boolean operators ..... 1.4.3.

C
Carry bit ..... 13.4., 13.5.
Cassette motor ag ..... 4.4.
Character generator ..... 3.2., 5.3.
—relocation ..... 5.4.
Circle drawing ..... 6.6.
CLC ..... 13.8.1.
Collisions ..... 7.2.
Color RAM ..... 5.3.
Color RAM pointer ..... 5.6.
Comparisons ..... 1.4.3., 13.7.

Cursor
—on/off ..... 5.6.
—placement ..... 5.6.
—line ..... 5.6.
—column ..... 5.6.

D
Data bus ..... 1.1.
Data
manipulation ..... 13.8.1.
Data
movement ..... 13.8.3.
Data pointer ..... 12.5.
DEC ..... 13.8.1.
Decay ..... 8.1., 8.2.
Device, current ..... 4.4.
DEX ..... 13.8.1.
Directories ..... 4.3.
Direct commands ..... 13.9.
Division ..... 13.6.

E
End without READY ..... 12.6.
EOR ..... 13.8.1.
Exclusive
OR ..... 1.4.3.
Extended colour mode ..... 5.3.

F
Files
—closing ..... 4.4.
—current ..... 4.4.
—open ..... 4.4.
Floating
accumulator ..... 1.4.2.
FORTH ..... 12.8.
FRE-function ..... 3.4.
uency ..... 8.1., 8.2.

G
Graphics, turning on ..... 6.3.
Graphic page storage ..... 4.1.
Graphics tablet ..... 10.4.

H
Hexadecimal system ..... 13.3.
High byte ..... 2.2.
High-resolution graphics ..... 6.1.

I
INC ..... 13.8.1.
INPUT ..... 5.6.
Input devices ..... 4.4.
Interface chips ..... 11.1.
Interpreter ..... 1.2., 1.3.
Interrupt ..... 1.2.
INX ..... 13.8.1.
I/O register ..... 1.5., 3.2.

J
JMP ..... 13.8.2.
Joystick ..... 10.1.
JSR ..... 13.8.2.
Jump commands ..... 13.8.2.

K
Keyboard
—buffer ..... 9.1., 9.5.
—code ..... 9.5.
—matrix ..... 9.1.

L
LDA ..... 13.8.3.
LDX ..... 13.8.3.
Light pen ..... 10.3.
Line drawing3j. 6.5.
Line format ..... 12.2.
Line number, last ..... 12.6.
List protection ..... 12.2.
LOGO ..... 12.8.
Loops ..... 13.13.
Low byte ..... 2.2.
LSR ..... 13.8.1.

M
Machine language ..... 13.1.
Memory, free ..... 3.4.

Memory protection ..... 3.3.
Memory
—division ..... 3.2.
—map 3.1., 14
—considerations ..... 1.5., 3.2.
Merging by hand ..... 4.2.
Multicolour
—graphics ..... 6.1.
—mode ..... 5.3.
—sprites ..... 7.1.
Multiplication ..... 13.6.
Multiplication program ..... 13.13.

N
NOT ..... 1.4.3.

O
OR ..... 1.4.3.
ORA ..... 13.8.1.
Output devices ..... 4.4.

P
Paddle input ..... 10.2.
PASCAL ..... 12.8.
Parallel port ..... 11.1.3.
PEEK ..... 1.4.1.
Pointer ..... 2.2.
POKE ..... 1.4.1.
Priorities ..... 7.3.
Proportional joystick ..... 10.4.
Point setting ..... 6.4.1., 6.4.2.

R
Release ..... 8.1., 8.2.
Relocatability ..... 3.2.
RENEW ..... 12.4.
Renumber ..... 12.3.
Repeat function ..... 9.4.
Reset switch ..... 1.6.
Restore ..... 12.5.
RTS ..... 13.8.2.

S
SAVE-protect ..... 12.6.
SBC ..... 13.8.1.
Screen on ..... 12.6.
SEC ..... 13.8.1.
Serial port ..... 11.1.1.
Shift commands ..... 13.14.
SHIFT-pattern ..... 9.2.
SID
—Operations ..... 8.1.
—Programming ..... 8.2.
Simulator ..... 13.9.
Single-byte commands ..... 13.9.
Sprite
—graphics ..... 7.4.
—pointers ..... 5.5.
—storage ..... 7.4.
STA ..... 13.8.3.
Stack ..... 2.2.
Status variable (ST) ..... 4.5.
Structuring ..... 12.7., 12.8.
STX ..... 13.8.3.
Subroutines ..... 13.13.
Subtraction program ..... 13.12.
Sustain ..... 8.1.
SYS ..... 1.4.2.
SYS-extensions ..... 13.15.

T
TAX ..... 13.8.3.
Time ..... 13.2.
Timer ..... 11.1.2.
TI ..... 12.6.
Tokens ..... 1.3.
Trace ..... 13.10.
TXA ..... 13.8.3.

U
User port ..... 11.2.
USR ..... 1.4.2.

V
VIC-II ..... 5.3.
Video RAM
—relocation ..... 5.5.
—pointers ..... 5.6.
Volume ..... 8.1., 8.2.

W
WAIT ..... 1.4.3.
Waveforms ..... 8.1., 8.2.
Write-protect notch ..... 4.3.

Z
Zeropage ..... 2.1.


1st
FIRST PUBLISHING LTD



This was brought to you

from the archives of

http://retro-commodore.eu