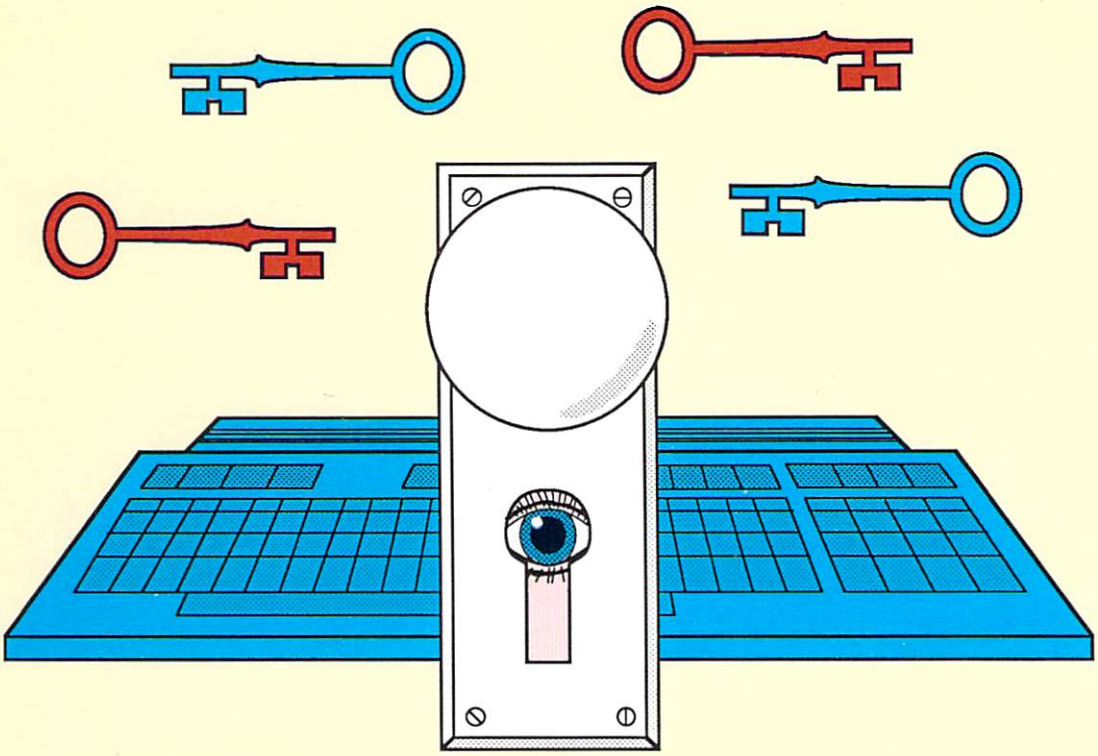
# COMMODORE® 128™

## Unlock the secrets of the C-128

# PEEKS & POKES

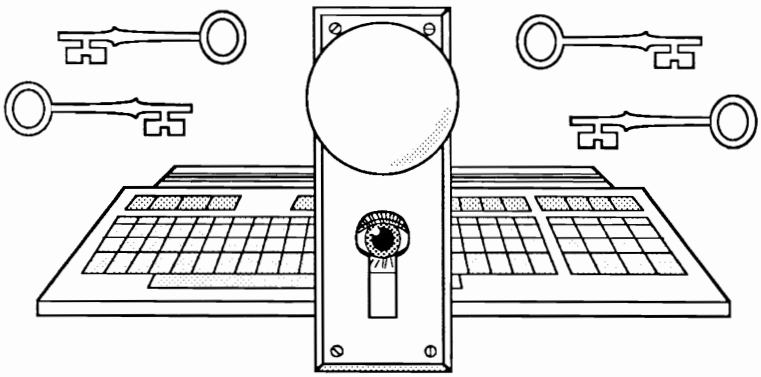


A Data Becker book published by

You Can Count On Abacus Software

# Commodore 128 Peeks & Pokes

Hans Joachim Liesert
Rudiger Linden



Data Becker Book

Published by

Abacus Software

First Printing, June 1986
Printed in U.S.A.

Data Becker GmbH
Merowingerstr.30
4000 Duesseldorf, West Germany

Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510



ISBN 0-916439-67-4

# Foreword

There's a new computer on the market: the Commodore 128. With its state-of-the-art hardware it should experience the same success as the C-64.

No manual can answer every question about a computer. We are going to supplement the Commodore 128 System Guide in an attempt to answer some questions that book may have raised. If you want to know how you can get more out of BASIC 7.0, then you have the right book in your hands.

The title **Peeks and Pokes** gives the impression that we are going to concentrate on these two commands. These instructions are the best way to gain access to the internal operations of your C-128. However, there are other commands we can use. These other commands can't be used in all manipulations, but we'll show you where they do work.

This book also gives you an introduction to machine language. As a matter of fact, you will be treated to two short courses—one for the 8502, and the other for the Z80.

This book is not a complete reference guide. However, you can get a good grasp of the basics of C-128 machine language programming without a lot of effort .

Have fun!

Hans Joachim Liesert, Rudiger Linden
Münster, December 1985.

# Table of Contents

# Chapter 1

# Operation of the C-128

# Operation of the C-128

The following sections will acquaint you with the C-128 and its operation, and explain some of its technical aspects. This first chapter is an introduction for readers who are unfamiliar with computers. If you know the fundamentals already, you may wish to skip to Chapter 2.

## 1.1 The Microprocessor

First we'll start off with some fundamental information on the brain of the computer—the *microprocessor*. The microprocessor is an integrated circuit that reads specific memory locations. In other words, it knows how to reach a fixed number of memory "cells". The microprocessor selects these cells through *address lines*. Each address line represents one *bit*, or binary digit, and can take on the values of 0 and 1.

Both the 8502 and the Z-80 microprocessors of the C-128 have 16 *address lines*. These address lines can access up to 2^16 = 65536 memory locations.

Theoretically, the C-128 microprocessor should not be able to access more than 65536 (abbreviated as 64K) memory cells. We know that the C-128 comes equipped with 128K of memory—how is it possible to access memory beyond the 64K limit? The answer is that the C-128 has a special electronic control that can switch between different 64K "banks" of memory.

Each memory location contains 8 bits, or one *byte*. To access these memory locations, the microprocessor has a *data bus* that consists of eight data lines. In this way the microprocessor can read from or write to a specific memory location.

The microprocessor can perform only simple operations. It can add, subtract, perform logical operations on the data, and store and retrieve data to and from memory. But it can do these things at a very high speed. Using combinations of simple operations at very high speed, the microprocessor can perform very complex operations.

## 1.2 The Operating System

Every time you read through a computer book, magazine or advertisement, you see terms like "Interpreter", "Operating system", or "Interrupt". This section explains some of these computer terms.

The *Interpreter* simply translates the BASIC instructions in a program into a language that the computer can understand. The C-128 can understand only its own special *machine language.*

The BASIC interpreter resides in ROM (Read-Only Memory) chips inside the C-128. The interpreter reads each line of a program and converts these program lines into a form that the 8502 microprocessor can understand.

If BASIC is running in direct mode, the interpreter gets the instructions from the keyboard buffer. This input buffer acts as a sort of delivery area. It constantly checks the operating system in ROM, as well as the cursor movement, peripherals, etc.

BASIC and the operating system are both machine language programs that begin to run when the computer is powered up, and continue to run until another machine language program is called. You can call such a machine language program yourself by issuing the BASIC SYS command.

The computer can only run a single program at a time. But the interpreter and the operating system are two programs that must both be operating at the same time. How does this work?

The simplest solution for running two programs simultaneously is to alternate between the two. Whenever BASIC is finished with a section of code, the computer switches over to the operating system. For example, when a peripheral device is accessed, BASIC is switched off and control is given to the operating system. The operating system then sends the information to the peripheral device. The control is switched back to BASIC.

There are problems with this, though. For example, the keyboard can only be accessed when the operating system is running. When a program is running, the <RUN/STOP> key should be accessible to stop the program in progress. To solve this problem, computer engineers developed the *interrupt.*

Every 1/60th of a second the processor stops the program that is running (whether it is a BASIC program or the operating system), then checks the keyboard and certain other operations. If the computer finds that the <RUN/STOP> key has been pressed, then the BASIC program is aborted. If another key has been pressed, it is stored in the keyboard buffer. You don't need to be concerned with this, however—the keyboard is checked much faster than you can type.

But from the microprocessor perspective, the time between two interrupts is quite long. The clock that runs a microprocessor usually runs at 1 to 2 million cycles per second. A typical machine language instruction only takes between 3 and 4 such cycles to execute. As you can see, a processor can perform thousands of instructions before it's stopped by an interrupt. After the keyboard interrupt, the processor continues to execute the program at the same location where the interrupt occurred.

Unfortunately, these interrupts can have a side-effect. After every interrupt, certain locations in memory are changed. BASIC must also keep track of these specific memory locations. There are many BASIC commands for graphics and sound that are controlled by interrupts. The location of each interrupt routine is put in a control register, as a BASIC command is given. A POKE in the wrong register can cause unexpected results!

There is a short test program at the end of this section. The FOR-NEXT loop listed below takes 46 seconds to run in the C-64 mode. By disabling the interrupts (we will discuss this in a later chapter), the program runs one whole second faster! The higher speeds cannot be measured with BASIC 7.0's internal clock, because the interrupt routines themselves are used in setting the clock.

Before you type in the program below, try typing the POKE command from line 10 in direct mode (without the line number). The cursor should disappear, and you will observe that the keyboard is no longer being read. When the interrupt is turned off, only <RUN/STOP>-<RESTORE> continues to function (Note: this program only works in the C-64 mode):

```
10 POKE 56334, PEEK (56334) and 254:
   REM INTERRUPT IS TURNED OFF
20 FOR I = 1 to 1000: PRINT I: NEXT I
30 POKE 56334, PEEK (56334) OR 1:
   REM INTERRUPT IS TURNED ON
```

## 1.3 The Interpreter

We mentioned earlier that the BASIC interpreter is responsible for translating BASIC commands into machine language. It is interesting to examine how the interpreter really works.

First we'll look at how the commands are interpreted. The C-128 doesn't store the commands in their full spelling--this would take up too much memory. The PRINT command alone would take 5 bytes (one byte for each letter).

Usually the command words are stored in an abbreviated form called *tokens*. This means that each command is stored in a code something like ASCII code. Numbers and characters not part of a command are saved directly in ASCII. Therefore, the command PRINT I requires only two bytes--the first for the token for the PRINT command and the second for the variable name.

The first part of the translation takes place as each program line is entered into memory. After the BASIC line is entered and the <RETURN> key is pressed, the interpreter converts all BASIC commands into their respective tokens.

The second part of the translation takes place after we enter the RUN command. The interpreter reads each token and then performs the needed work. For example, the PRINT command uses one subroutine to determine which variable should be displayed, and another subroutine to display that value on the screen. Other commands such as the arithmetic routines use different subroutines in ROM.

This book will help you get a closer look at these routines. For the more dedicated readers, consult *C-128 Internals* from Abacus. It contains a ROM listing of the operating system and many useful machine language programs.

## 1.4 PEEK and POKE

Imagine yourself in the following situation: You find a computer magazine with a super program to type in. You type in the entire 20K listing--and when you run it the word ERROR pops up, or worse yet, the computer simply crashes.

If you want to find the error you must first understand how the program works. After studying the program for a while, you're probably ready to say to yourself, "What in the world are all these stupid POKE commands?! No normal programmer would use them!" Well, let's find out just what they are.

### 1.4.1 POKE and PEEK

We'll start with the POKE command. The syntax of the POKE command is quite simple: POKE *address,byte*. *Address* can range from 0 to 65535, and *byte* can contain values between 0 and 255. This command puts the value of the byte into the memory location at the address indicated.

The POKE command can be used to do many things: filling up the monitor screen, setting a color, and many other things. With this command you can delve into the workings of the computer, as well as the operating system and the interpreter.

You can also look at these locations by using PEEK. The syntax for PEEK is: PRINT PEEK(*address*), where the *address* can range from 0 to 65535. It is important to note that PEEK is a *function*. Therefore the parameter must be placed within parentheses.

Together the two commands have the unique ability to modify individual memory locations. Before using them, you should be aware that randomly modifying memory within the C-128 can cause the computer to crash. Therefore know your memory map!

In the C-128 mode you have to use the BANK command to inform the PEEK and POKE commands which memory bank to use.

## 1.4.2 SYS and USR

Now we come to the commands which are of particular interest to the machine language programmer: SYS *address*, and PRINT USR(*x*). Both call programs in machine language.

The SYS command runs the machine language program at the address specified. When the machine language program is finished, control is returned to BASIC. The BANK command specifies the memory bank in which the machine language program is located.

The USR function operates similarly. However, it returns additional useful information. The first difference lies in the syntax. USR is a function like PEEK and SIN. Therefore its parameters must be contained within parentheses.

The location of the machine language routine for USR is specified in memory locations 785 and 786 (C-64 mode) or memory locations 86 and 87 (C-128 mode).Whenever the interpreter finds a USR function call it looks at these "vectors" to find where the program resides. It then transfers control to this location. When the machine language program is finished, it returns control back to BASIC.

The USR routine is useful for passing data back and forth between BASIC and machine languge. The value in the parentheses is returned to the interpreter in the *floating-point accumulator* (97 - 101 for the C-64, 99 - 104 for the C-128). The floating-point accumulator is an internal register in which all arithmetic operations are carried out. Using the information in this register, you can write your own machine programs that receive and operate on values. At the end of the USR routine, the number in the register is returned to BASIC. This allows you to send data to machine language programs in variables (for example: A=USR(*x*)).

As a result, machine language programmers can write much faster functions--such as a quick sorting routine. The capabilities of the USR function make it extremely useful.

### 1.4.3 WAIT

We've got just one more command to look at: the mysterious WAIT *address,X,Y* command.

This command tells the computer to halt the currently-running program for a period of time, then start running it again. When the computer sees a WAIT command it reads the byte of data located at *address*. It then takes that value and performs an XOR with the Y parameter. The XOR (shorthand for EXCLUSIVE-OR) is related to the boolean OR operator. The following table shows the results of an XOR operator:

```
XOR| 0 | 1
 ---------
 0 | 0 | 1
 ---------
 1 | 1 | 0
```

The result of a XOR is a 1 if either bit one or bit two is a 1, but not both. The XOR function is available in BASIC 7.0 only in the C-128 mode.

The result of this XOR operation is then ANDed with the X parameter. If the result is zero, the interpreter repeats the whole procedure. If the answer is anything other than zero the interpreter restarts the program at the point where it was halted.

There is another form of the WAIT command. In this form the Y parameter is not specified. Here the interpreter waits until the result of an AND with the byte of data located at *address* and the X parameter is zero.

### 1.4.4 Extended RAM commands

Unfortunately, there is an error in early printings of the Commodore 128 System Guide. The three commands affected are FETCH, STASH, and SWAP. These commands are used to transfer blocks of memory in different banks of memory. The third and fourth parameters must be exchanged to correct the error. The correct entries should look like this:

```
FETCH #bytes, intsa, expsa, expsb
```

```
FETCH #bytes, intsa, expsa, expsb
STASH #bytes, intsa, expsa, expsb
SWAP  #bytes, intsa, expsa, expsb
```

### 1.4.5 BANK

Here are a few comments about the BANK command. The table below lists the different parameters for each possible RAM/ROM combination. The interpreter uses memory location 981 as the reference byte for the bank command. You can use PRINT PEEK(981) to see which BANK is active.

| N | Configuration |
|---|---|
| 0 | BANK 0 |
| 1 | BANK 1 |
| 2 | BANK 2 |
| 3 | BANK 3 |
| 4 | Internal ROM functions, BANK 0 and I/O region. |
| 5 | Internal ROM functions, BANK 1 and I/O region. |
| 6 | Internal ROM functions, BANK 2 and I/O region. |
| 7 | Internal ROM functions, BANK 3 and I/O region. |
| 8 | External ROM functions, BANK 0 and I/O region. |
| 9 | External ROM functions, BANK 1 and I/O region. |
| 10 | External ROM functions, BANK 2 and I/O region. |
| 11 | External ROM functions, BANK 3 and I/O region. |
| 12 | Internal low ROM functions, high ROM, BANK 0 and I/O. |
| 13 | External low ROM functions, high ROM, BANK 0 and I/O. |
| 14 | BASIC-ROM, BANK 0 and Graphic generator. |
| 15 | BASIC-ROM, BANK 0 and I/O region. |

## 1.5 The Memory Management Unit

Regardless of what the title of this section implies, this won't get too technical, but to understand some programming tricks it helps to understand the internal construction of the computer.

### 1.5.1 The C-64 mode

You might have wondered why Commodore says the C-128 in C-64 mode has 64k of RAM even though BASIC has only 38k of memory available to use.

The 64 kilobytes are there--however, they cannot all be used. The 8502 microprocessor can address 64k of memory. Therefore it can only access 65536 different memory cells. If all this was RAM, there would be no room for the Zero page, ROM, the BASIC interpreter, or the operating system. Of the total 64 kilobytes, ROM takes up 20k, the Video RAM takes up 2k, and four more are taken up by the Zero Page. So what's left is the 38k we mentioned earlier.

Diagram 1.5.1 shows a block diagram of the memory of the C-64. You can see that RAM and ROM are located next to each other and have the same memory locations. If you wanted to switch between RAM and ROM you would have to change the contents of memory location 1. This, however, is not possible from BASIC. If we turned off the BASIC-ROM, the processor couldn't find the program, and the computer would crash.

This problem can be resolved somewhat, by using the POKE or LOAD commands to write to a memory location, and by using the PEEK or SAVE commands to examine a location. A POKE to a memory location in ROM will alter the same location in RAM, but a PEEK to a location in ROM will return the value from ROM.

The 4K in the upper third of RAM is at 49152 to 53247 is free memory. This area is for machine language programming, and can be accessed using the BASIC commands of PEEK and POKE.

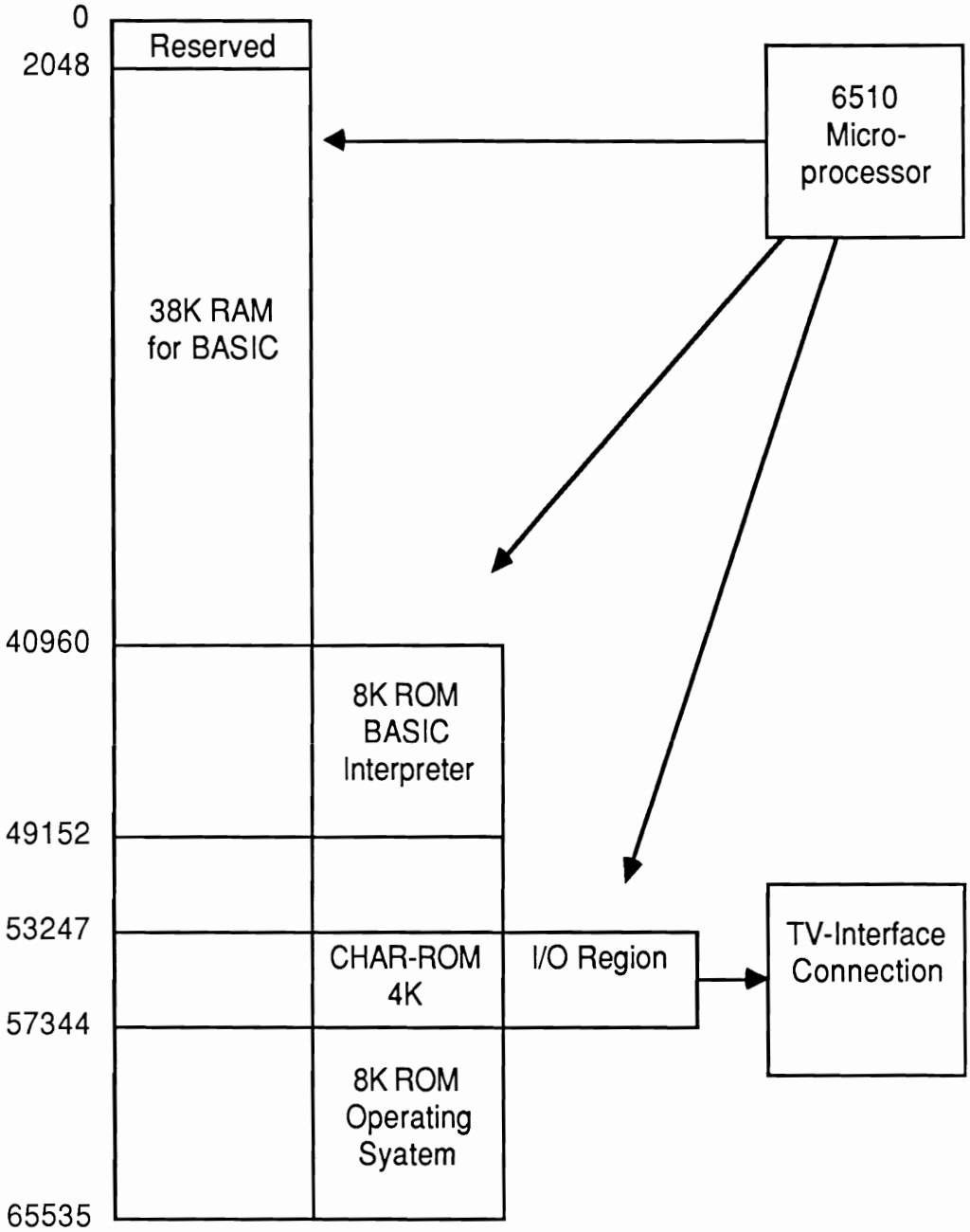

0
2048
Reserved
38K RAM
for BASIC
40960
8K ROM
BASIC
Interpreter
49152
53247
CHAR-ROM
4K
I/O Region
57344
8K ROM
Operating
Syatem
65535
6510
Micro-
processor
TV-Interface
Connection

DIAGRAM 1.5.1

Finally we come to the I/O (Input/ Output) region of memory. This is where the ports for the keyboard, the monitor, and the sound-generator are found. The computer writes data to this area of memory where it is converted to a video signal, a sound, or magnetic impulses which are saved on a floppy disk. This is also the area where the keyboard or any other input device writes its data and where the color RAM is located.

As the block diagram shows, the I/O memory is one of three sections of memory. If we change location 53280, we change the frame color. However, that byte doesn't lie in RAM--it stays in an I/O register. This is also true for the color RAM.

There are a total of four I/O regions in the C-64 mode. Two of these are the VIC (where the graphics and the monitor are managed), and the SID (the sound generator). There are also two CIA's (Complex Interface Adapters), for the keyboard and other interfaces--such as the USER-PORT.

## 1.5.2 The C-128 mode

Diagram 1.5.2 illustrates the layout of the C-128's memory. You can see that the memory is divided into banks of 64k. Remember, the microprocessors can only address 64k at a time. In order to access all of the memory available the processor uses the MMU (Memory Management Unit). The MMU switches between the appropriate banks as needed. The I/O region is also under the supervision of the MMU. While you're in the C-64 mode, the VDC (the 80-column video controller) and some control registers are still managed by the MMU. The MMU also controls which processor is active. The two microprocessors can be switched back and forth, just by changing a certain register in the MMU.

The I/O region of the C-64, however, cannot be changed. For more information on the MMU in both C-64 and C-128 modes see *C-128 Tricks & Tips* from Abacus.

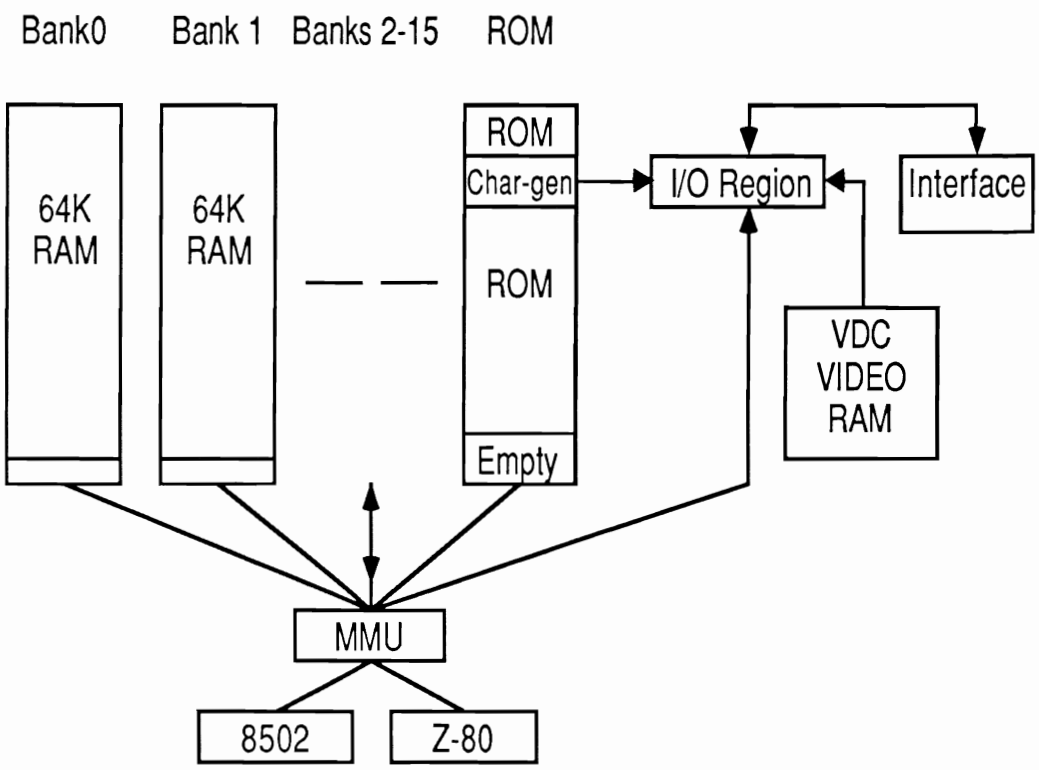



Diagram 1.5.2

## 1.5.3 The CP/M mode

When you are in CP/M mode, only 59K of memory is available for programming. This memory, plus some system routines, is stored in BANK 1. BANK 0 contains the BIOS (Binary Input Output System routines), it is located in addresses $0000 - $D000. The MMU can access all of these locations.

All of the CP/M routines are in BANK 0. These routines, together with the programs in user memory, can communicate with the upper end of the addresses, which is established as a common block. Moreover, a CP/M program can use the MMU to access RAM and ROM as necessary.

# Chapter 2

# The Zero Page

# The Zero page

## 2.1 The Zero page is greater than 0

If you look at the Appendix in the Commodore 128 System Guide, the section on the Zero page looks menacing, but it contains a wealth of programming tricks and possibilities if you know where to look.

The name of the Zero page is not entirely correct. The first 256 bytes of microprocessor memory is called the Zero page. In the C-64 mode this is the first kilobyte, or four pages of memory. In the C-128 mode it is the first four kilobytes, or 16 pages of memory. The operating system and the interpreter use registers in this area to keep track of the state of programs, numbers or codes. The first 256 bytes are useful for quickly getting data out of memory, since they can be accessed by a single byte. This is why we will usually store our data in this region.

Many registers in the Zero page must contain certain values so that the computer functions properly. There are other registers which can be used for any purpose.

In the C-128 mode the first four kilobytes are reserved for the internal data used by the BASIC interpreter and the operating system.

## 2.2 Pointers & stacks

Two structures which we will use from now on are the *pointer* and the *stack*.

A pointer, also known as a vector, points to specific locations in memory. They can point to either information or subroutines. For example, the cursor pointer always contains information about where the cursor is located on the screen, plus the code of the character at this location. Pointers to subroutines extend the use of the interpreter. Other vectors point to the character output routines. This means that it's possible to write machine language routines that change the `PRINT` command so that whatever shows on the screen is also sent to a printer.

Pointers have a specific format. They are always two bytes long. The first is called the *lowbyte*, and the second, the *highbyte*. To find the memory location the pointer is referring to, use the following formula:

```
ADDRESS = LOWBYTE + 256 * HIGHBYTE
```

It is standard for the lowbyte to appear before the highbyte in memory.

A one-byte pointer is an *offset* to a region in memory, between locations 0 and 255, and is added to a *base address*.

In C-128 mode the pointer must also contain information about the memory bank to be accessed. Only 64K of memory can be addressed at one time in any bank.

A *stack* is used to temporarily store data in memory and return it in reverse order when it is used. You can only get information from, or put information on, the top item of the stack. (See Diagram 2.2). Stacks are widely used for implementing subroutines. When a subroutine is called the return address is pushed onto the stack, and when a `RETURN` is reached the computer gets the return address from the top of the stack and continues execution from this address.

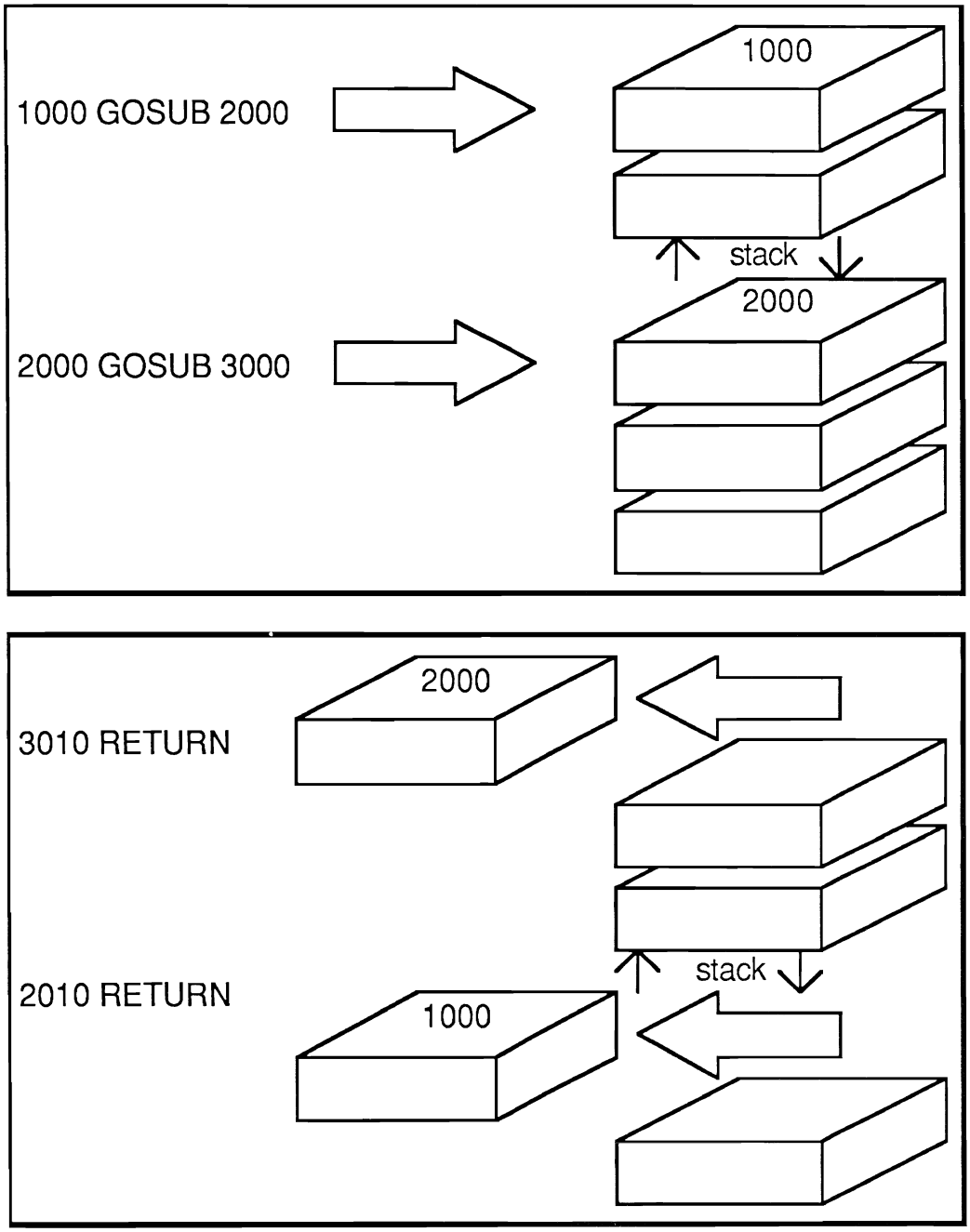

1000 GOSUB 2000
1000
stack
2000
2000 GOSUB 3000
2000
3010 RETURN
stack
2010 RETURN
1000

Diagram 2.2

Look at Diagram 2.2. When the interpreter receives a GOSUB command the line number of the GOSUB command is stored on top of the stack. Later, when the RETURN command is reached the interpreter pulls the line number back from the top of the stack.

The C-128 also has stacks for the interpreter, for variables, as well as other stacks that should never be changed by the user. These include:

- the Processor Stack (256-511) for machine language programs.
- a stack for BASIC subroutines
- a stack for FOR-NEXT loop

Now we will start with some applications. In the next sections you will find many interesting tricks that can help you program your C-128.

## SUMMARY OF POINTERS

```
ADDRESS = LOWBYTXE+256*HIGHBYTE

LOWBYTE = ADDRESS-INT(ADDRESS/256)*256
HIGHBYTE= INT(ADDRESS/256)
```

Pointers usually consist of two bytes that are always in the order of LOW/HIGH.

In the C-128 mode you will also have to instruct the MMU to access a particular BANK.

# Chapter 3

# Memory

# Memory

## 3.1 The memory map

In the Appendix you will find the memory map for the C-64 and the C-128. Next to the Zero page is the I/O region. There are several things you should notice in the Zero page listing in the Commodore 128 System Guide. For instance, the first 5 bytes of the supposedly usable region of 673 through 767 are reserved in the C-64 mode for the CIA. It also pays to be very careful when using Zero page for storing data. A misplaced `POKE` can confuse or crash the interpreter.

Try experimenting with the Zero page. Many tricks are discovered by accident, and as far as we know, there is no `POKE` combination that could destroy your computer.

## 3.2 How does bank switching work?

Bank switching is much more complicated than the BANK command leads you to believe. For one thing, ROM does not lie where we think it does.

### 3.2.1 The MMU configuration register

The Commodore 128 System Guide, states that the memory can be expanded up to 1 Megabyte. If you divide 1Mb into 64K sections, you get 16 different memory banks. In this case, the addressing capacity of the MMU has been all used up. There is no more room for things like ROM's and I/O regions.

Consequently, the C-128 was designed in much the same way as the C-64. ROM isn't accessed in its own memory bank. Rather, a command is reflected through every bank. It can be found at address D500 (hexadecimal) as well as under FF00 in every bank. Referencing either area is identical. It is necessary to point out that the D500 address is in the I/O region, and so it can be overwritten. In this case you can use the region at FF00 as an extended I/O region. This region is not overwritten.

The configuration register (CR) has many purposes. It controls every bit in the memory banks. The zero-bit takes care of the I/O region from D000 through DFFF, which is also used for switching banks. Only the zero-bit has to be set to access the zero bank. If it is set to 1, the processor accesses ROM or RAM in the region from D000-DFFF. The rules for setting the bits of the configuration register follow.

Bit one is used for addressing the region of memory from 4000-7FFF. If it is a zero, the MMU accesses the region containing the BASIC-ROM LOW (This is the lower region of BASIC). If bit 1 is set to 1 then you can access the RAM region of the same address.

The next section (8000 through BFFF) uses two bits together--namely bits 3 and 2. If the combination is 00, then BASIC-ROM HIGH is accessed. The combination 11 switches over to the RAM at the same address. The other combinations are for different areas of ROM. 01 is for the internal

ROM functions. A ROM function is for a particular ROM chip, which may be placed in an unused socket in the C-128 circuit board. This ROM chip could contain things like a word processing program. There are also external ROM functions. It is quite easy to access an external module—just access configuration register Bits 3 and 2 with a 10.

Bits 5 and 4 are used similarly, but they access the region of memory between C000-FFFF. This section is used to access things like the operating system ROM (use a combination similar to bits 3 and 2).

All that remain are bits 7 and 6. These are used only to access the different memory banks. In your C-128 only two banks exist, so the 7th bit is useless. Bit-6 determines which RAM bank is used.

This does not cover all the possibilities for the MMU. Here is a summary where all the banks are defined and much more:

## SUMMARY OF THE CONFIGURATION REGISTER

Addresses: D500 or FF00

Bit 0 turns on the I/O region whenever it is set to 0.

Bit 1 turns on the ROM for locations 4000-7FFF when it is set to 0.

Bits 2 and 3 are for locations 8000-BFFF.
Combination:
00 = ROM
01 = Internal ROM functions
10 = ROM modules
11 = RAM

Bits 5 and 4 work in the same manner except for locations C000-FFFFF.

Bit 6 determines which memory bank to use.

### 3.2.2 The pre-configuration mechanism

It's a slow and complicated process to switch banks every time you want to load the contents of the configuration register. There must be a way to switch quickly in the MMU. We must know which memory configurations we'll be using in our program to do this. Up to four such configurations can be stored in the pre-configuration PCR registers (memory locations D501-D504). These registers are always available. As soon as one of the loading registers LCR (memory locations FF01-FF04) has a word written to it, the MMU will independently switch it with the appropriate word in the PCR in the configuration register. This method is much quicker than the processor could do it.

### 3.2.3 The Mode Configuration Register

As its name suggests, the MCR (address D505) controls the modes of the computer. We will discuss only two bits of the MCR in this section.

The 0-bit controls which processor is active:

1: the 8502 is active
0: the Z-80 is active

The 6-bit controls the mode in which the computer will run:

1: the C-64 mode
0: the C-128 mode

There is a problem with this method, though. Every time we try to change the mode using a POKE, the computer locks up. The only way to change the mode of the computer without causing a crash is with a machine language program. Several operating system routines can be used for this purpose.

## 3.2.4 The RAM-configuration register

Now we come to an outstanding capability of the MMU. It establishes in RAM a common area for all of the memory banks. A specific area in memory for the high or low ends of the address range is not needed because they are all the same size. This is important, because the different regions of memory can exchange data with each other. These regions, and more, are controlled by the RAM configuration register (D506). The size of the common region of memory is held in the first two bits. Here is a table:

| BITS 1 0 | Size in K |
|---|---|
| 0 0 | 1 |
| 0 1 | 4 |
| 1 0 | 8 |
| 1 1 | 16 |

The second and third bytes determine where this common region is located. If the bits are 00, there is no common region of memory. If they are 01 this region is in the lower portion of memory starting at 0000. If they are 10, it is in the upper end of memory. The combination 11 has one half in low memory and the other in high memory.

The fourth and fifth bits are not used, just like bit 7. The sixth bit determines in which RAM bank the VIC chip looks for its data.

### SUMMARY OF THE RAM-CONFIGURATION REGISTER

BITS 1 and 0 determine the size of the common region.

BITS 2 and 3 determine if and/or where the region is located.

BIT 6 determines in which Bank the VIC chip finds its information.

### 3.2.5 Memory page pointer & version register

These registers have very specialized tasks. The memory page pointer allows you to switch the Zero page and the stack into the another region of memory. All you have to worry about are the small glitches in the machine language.

The version register helps commercial software applications determine in which version of the C-128 the program is running. In this way, an eventual change in the machine structure can be accounted for in future programs.

### 3.2.6 Memory configurations for the interpreter

BASIC 7.0 constantly uses the pre-configuration register mechanism. Four standard configurations are stored, and either bank 0 or bank 1 is switched. Just two of these combinations take up the entire ROM. There is also a common region of memory (1K) at the low end of memory.

If an interpreter command is encountered, there are two possibile consequences. In the direct mode, the command is passed through the keyboard buffer (this is where the computer stores everything that we type). The keyboard buffer lies in a common region of memory, reserved for every configuration. After this is finished, the ROM is turned on and the command is turned off.

If the interpreter command comes directly from program memory, a subroutine in the common region of memory copies the command from program memory into the low end of memory. Here the interpreter converts the command into the appropriate machine language equivalent.

Many commands require information from a different bank of memory. For example, variable memory uses a special subroutine to fetch the data from RAM.

The BASIC commands PEEK and POKE are designed so that they cannot alter the memory configuration. For this reason, you must use machine language routines if you want to access information from the character generator, for example.

### 3.2.7 Bank switching in the C-64 mode

In the C-64 mode everything is a lot easier. Byte 1 controls the partitioning of memory, and only bits 0 through 2 have any control. Normally all three bits are set to 1. If any of the bits are set to 0, then memory partitioning is altered.

If bit 0 is set to zero then the BASIC-ROM (40960 - 49151) is turned off. If bit 1 is a zero, then both BASIC and the operating system are turned off. If both bits 0 and 1 are set to zero, then the I/O region is also turned off. That leaves 62K for I/O use (video RAM and the Zero page are not overwritten). Bit 2 determines whether or not the character generator can be read (remember the triple assignment).

Unfortunately, this arrangement causes problems. If we turn off BASIC and the operating system, the machine locks up. Therefore, only a machine language program can change these bits.

There is something else which affects the character generator. The C-64 will lock up if bit 2 is set to zero. It won't be able to access the I/O region. The interrupt routines will continuously monitor the keyboard. At this point it would be helpful if we understood the interrupts (See Section 1.2).

## SUMMARY OF THE MEMORY LAYOUT FOR THE C-64

The memory layout is controlled by the first byte in memory (Byte 1).

| | |
|---|---|
| Bits 0-2 | are set to one in normal operation. By turning off specific bits you can change the memory configuration. |
| Bit 0 | turns off the BASIC-ROM. |
| Bit 1 | turns off both BASIC and the operating system at the same time. |
| Bits 0 & 1 | turns off the I/O region along with the BASIC-ROM and the operating system. |
| Bit 2 | is responsible for the output of the character generator. |

Turning off any of these interrupts from BASIC is not possible.

## 3.3 Memory protection

The architecture of the C-128 is designed to provide a large amount of memory for BASIC. As it is, the BASIC interpreter itself occupies a lot of memory. For example, to implement a second graphics page, you might want to protect this area from inadvertent destruction by the BASIC commands.

We can do this in several ways:

The beginning of a BASIC program resides at either location 7168 or 16384 of memory. BASIC storage is divided into two banks.

The Zero page contains a pointer to both the beginning and ending address of program storage. If we want to relocate the beginning of program storage to 20000, we set the Zero page pointers at memory locations 43/44 (for the C-64) or 45/46 (in the C-128) as follows:

```
POKE 46, (20000+1)/256
POKE 45, (20000+1)-PEEK(46)*256
```

Issue these commands in the above order, since the second POKE depends on the first. Don't forget to add 1 to the starting address 20000. The first byte of program storage must be set to a zero.

```
POKE 20000, 0
```

If you are using the C-128 mode, you're done!

If you are using the C-64 mode, you have an extra step. You must issue a NEW command to reset the additional pointers. A CLR is not enough!

So we have changed the pointers to the start of BASIC program storage with these POKEs. However, the program text itself is not relocated.

One command does relocate program text--the GRAPHIC command. This moves the current program text from 7168 to 16384.

If you change the start of BASIC program storage pointers manually, any program in memory is lost unless you first relocate it to the new program storage area. To preserve the program you must either retype it or load a copy of the program to the new program storage area (as described below).

In the C-128, you can relocate the variable storage area by one of two methods.

First, you can set the pointers at 47/48 to the start of the new variable storage area in `BANK 1`.

Second, you can set the pointers at 4626/4627 to the end of the program storage area in `BANK 0`.

In either case, you must also issue either a `CLR` or `NEW` command to reset the other pointers. These changes must be performed in direct mode before a program is entered. Alternatively you can use the following short program to set the pointers and load your program:

```
10 POKE 43,(2560+1)-INT(2561/256)*256
20 POKE 44,(2560+1)/256:POKE 2560, 0
30 LOAD "MAINPROGRAM",8
40 REM  <RUNSTOP> <RESTORE> NOW RESETS COMPUTER
```

Unfortunately, this doesn't work in the C-64 mode which still requires a `NEW` command. For this reason it cannot be loaded. Without these commands you cannot change the pointers. We can use the following steps instead:

1. Protect memory in the direct mode (like above).
2. Load a program
3. Start the program, and run it so all of the variables are used at least once.
4. View all of the pointers using the following commands:

```
PRINT PEEK(43), PEEK(44)
PRINT PEEK(45), PEEK(46)
PRINT PEEK(47), PEEK(48)
PRINT PEEK(49), PEEK(50)
```

5. Set the pointers using the following program:

```
1 POKE 43,PEEK(43):POKE 44,PEEK(44): POKE address,0
2 POKE 45,PEEK(45):POKE 46,PEEK(46): POKE 47,
  PEEK(47):POKE 48,PEEK(48): POKE 49,PEEK(49):
  POKE 50,PEEK(50)
3 CLR: LOAD "program"
```

Where *address* is the starting point of the BASIC program. This loads a runnable program.

In the following sections, you need only to learn the topic of the section. We will take care of protecting your memory for you.

## SUMMARY OF MEMORY PROTECTION

Setting the start of BASIC:

C-128:`POKE 45,`*LOW*`:POKE 46,`*HIGH*`:POKE` *address*`,0`
C-64:`POKE43,`*LOW*`:POKE 44,`*HIGH*`:POKE` *address*`,0:NEW`

BASIC Variable end setting

C-128: `POKE 57,`*LOW*`: POKE 58,`*HIGH*`: CLR`
C-64: `POKE 55,`*LOW*`: POKE 56,`*HIGH*`: CLR`

Additional possibilities for the C-128:

Variable storage beginning (47/48) on top,and program end (4626/4627) on the bottom.

## 3.4 Free Memory

The FRE() function returns an integer word. Integer variables in BASIC have a range from -32767 to +32767. The interpreter must use numbers greater than 32767, so it uses negative numbers. You can find the number of free bytes in C-64 mode by using the command PRINT 65538+FRE(0).

You may want to get rid of some data that you have sent to a machine language program, or you don't want to use an entire variable for a single byte, or even a single bit. Only the CLR command deletes all of the variables in a program.

Find some free memory in the Zero page to POKE your data into. A NEW or CLR command won't clear this page. Below is a list of free memory.

UNUSED AREAS IN C-128 MODE:

| | |
|---|---|
| 251 - 254 | |
| 996 - 1007 | |
| 1021 - 1023 | |
| 2816 - 3072 | Overwritten by cassette commands or a BOOT. |
| 3072 - 3583 | Overwritten by any RS-232 command. |
| 3584 - 4095 | use only if no sprites are used. |
| 4864 - 6143 | |
| 6144 - 7167 | Reserved for function keys |

UNUSED AREAS IN C-64 MODE:

| | |
|---|---|
| 251 - 254 | Could eventually change |
| 678 - 767 | |
| 780 - 783 | Use only if SYS is not used. |
| 820 - 827 | |
| 828 - 1019 | Overwritten by any cassette commands. |
| 1020 - 1023 | |
| 2024 - 2039 | |
| 49152 - 53247 | |

# Chapter 4

# External storage and peripherals

# External storage and peripherals

## 4.1 Retrieving data for graphics, monitor output, etc.

The C-128 can display graphics of all types. All of these graphics must be retrieved from memory at some point. This process is simplified on the C-128 by using the BSAVE command. The address of video RAM is discussed in a later section.

In C-64 mode, displaying graphics is not quite so simple. However, there are some helpful routines. The first program of this chapter is one that saves a certain region of memory onto a cassette.

Once again we can use the pointers in the Zero page to help. The pointers and registers for data management are located in the region from between 170-1995.

The most important pointers are those that point to the beginning and end of the memory range to be saved. The pointer to the beginning is found in locations 193 and 194. The pointer to the end is found in locations 174 and 175. The SAVE command can be called using the command SYS 62954. We must also enter a filename before we can save it. An easy way to do this is to put a REM line in the first program line containing the file name. The address of the program name is then set by the operating system in locations 187 and 188.

Now we'll discuss secondary addressing, device numbers, and the length of filenames. Take a look at this program:

```
10 REM filename
20 POKE 193,SL: POKE 194,SH:
   REM starting address (low/high)
30 POKE 174,EL: POKE 175,EH:
   REM ending address (low/high)
40 POKE 187, PEEK(43)+6: POKE 188, PEEK(44):
   REM points to filename
50 POKE 183,L: REM filename length
60 POKE 186,1: POKE 185,0: REM device/secondary
   address
70 SYS 62954: REM call the save routine in ROM
```

A note about line 40. The operating system keeps the vector to the filename in Zero page at 187/188. Since we're using the filename in the `REM` statement in line 10, this vector will point to the start of program storage + 6. If the result of `PEEK(43)+6` is greater than 255, then the computer will give an `ILLEGAL-QUANTITY-ERROR`. If this happens you should calculate the address using the formula `PEEK(43)+6+PEEK(44)*256`.

You can also write data to diskette. Use `PRINT#1, CHR$(x)`. This sends an ASCII character to the diskette.

To use this command, you must understand a little about how information is stored on a floppy. Every program on the diskette has an identifying entry in the directory, in addition to the program text. At the start of the text are two bytes that contain the starting address for the loading point of the program. Normally, these are 0 and 8 (0+256*8=2048=Start-of-BASIC). The text is saved byte by byte in tokenized form. Basically, the interpreter codes are similar to an ASCII code for the disk drive. If you need to read a pointer with the contents in locations in bytes 65 and 66, you can do this by using the following command:

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

where `A` and `B` are both string variables.

We can write a character to the diskette using the BASIC command:

```
PRINT #1, CHR$(X)
```

If we had used `PRINT #1,X,` then `X` would be saved as more than one byte (1 byte for each character). You should follow these steps when saving a program using this technique:

1. Make a directory entry by using the `OPEN` statement
2. Write the starting address to the diskette:
   Use the following formula:
   ```
   PRINT #1, CHR$(lowbyte);:
   PRINT #1, CHR$(highbyte);:
   ```
3. Write the program text to diskette

The text can even be a machine language program, which is saved byte by byte. Here is a program that saves the screen image located at $0400 to diskette:

```
10  OPEN 1,8,1, "0:IMAGE"
20  PRINT #1, CHR$(0);:
    PRINT #1, CHR$(4);:
    REM Starting point
30  FOR I= 1024 to 2023
40  PRINT #1, CHR$(PEEK(I));
50  NEXT I: CLOSE 1
```

You can reload the program or screen image using either of these commands:

```
        LOAD "IMAGE",8,1
  (or)  BLOAD "image"
```

The secondary address 1 (part of the `OPEN` statement) is very important. It tells DOS (Disk Operating System) that we want to save a program to diskette.

**Warning**: Do not omit the `CLOSE` statement at the end of the program. If you do, your program will not be saved correctly.

You can use this method for exchanging data between the C-64 and the C-128.

## 4.2 Merging programs by hand

Now we come to a common problem. Perhaps you have parts of programs that are saved, tested, and would work very well together, but you don't want to have to retype these program parts to use in new programs. At times like these it would be nice to have a MERGE command to combine two programs together. You can use the following technique to do this.

What we want to do is to append one program at the end of another. It would be nice if we could tell the interpreter where in memory we want to load the second section of code. We just have to worry about overwriting the old program.

If the program memory is protected, we can simply LOAD the program to free up the protected memory. The second program must have larger line numbers than the first. Otherwise, the interpreter will write over the old lines anyway.

Here are the steps required to merge two programs:

1. `PRINT PEEK(45),PEEK(46)` (C-64: 43/44)

   This prints the contents of the pointers to the start of program storage (normally, this would be between 1 and 28 for the 128, and 1 and 8 in C-64 mode). We should jot these down somewhere so that we can restore the configuration of the original program later.

2. `POKE 46,(PEEK(4624) + 256 * PEEK(4625) - 2)/ 256`
   ( C-64: 44/45/46 )
   `POKE 45,(PEEK(4624) + 256 * PEEK(4625) - 2)`
   `-PEEK(46)*256` ( C-64: 43/45/46/44 )

   The pointer to beginning of the variable storage is in locations 47 and 48 (45 and 46 in C-64). The end of the program is two bytes before this value. Remember that the program is in BANK 0 and the variables are in BANK 1. The pointer 4624/4625 points to the end of the program in C-128 mode.

   The two POKE commands adjust the program starting address to a point beyond the original program text to protect it from being overwritten.

3. NEW

   This initializes the pointers to the new variables and line numbers which will be used in the resulting program.

4. DLOAD or LOAD

   Now we just LOAD the new program into memory. You cannot use the command LOAD "*Name*",X,1 or BLOAD; this would put the address of the new program right on top of the old one, thereby overwriting it.

5. POKE 45,*original value A*: POKE 46,*original value B* (43/44)

   Here we return to the original memory configuration. Both programs are dependent on each other.

   There is a second method in the C-128 mode. In order to use this method the MERGE command must have an additional line in the target program, for example: 0 REM

   Now, use the command:

   BLOAD "*name*", ON B 0, P *address*

   to load the combined program after the old one. Replace *address* with the formula:

   ```
   PRINT PEEK(4624)+ 256 * PEEK(4625) - 2
   ```

   After the BLOAD command the pointers to the combined program are not yet correct. A LISTing will produce confusing gibberish. If we delete line 0, BASIC is forced to recalculate all the pointers to the program text. The program is now completely merged. **Warning**: Make sure that the programs do not contain the same line numbers.

## SUMMARY OF MERGING BY HAND

1. `PRINT PEEK(45), PEEK(46):`
   `REM PEEK(45) original value A`
   `PEEK(46) original value B`
2. `POKE 46,(PEEK(4624)+256*PEEK(4625) -`
   `2)/256`
   `POKE 45,(PEEK(4624)+256*PEEK(4625) - 2) -`
   `PEEK(46)*256`
3. `NEW`
4. `LOAD`
5. `POKE 45,`*`original value A`*
   `POKE 46,`*`original value B`*

Second method for the C-128:

1. Put a dummy line 0 in your program
2. `BLOAD "`*`name`*`", ON B 0,P` *address*
   where *address*= `PEEK(4624)+256*PEEK(4625)-2`
3. Remove dummy line 0 from program.

## 4.3 Program files

As mentioned in section 4.1, the first two bytes in a PRG-file contain the beginning address of the program. You can display these pointers with the following program:

```
10 OPEN 1,8,8,"name,p,r"
20 GET #1,A$: GET #1,B$
30 AD=ASC(A$+CHR$(0)) + 256*ASC(B$+CHR$(0))
40 PRINT "starting address"; AD
50 CLOSE 1
```

The directory is also a program file. You can display the directory by typing `OPEN 1,8,8,"$,p,r"` . Using `GET #1`, you can read the directory byte by byte.

# Chapter 5

# The 40-column screen

# The 40-column screen

We are going to discuss the normal setup and manipulation of the 40-column screen. You will discover that you don't have to use high resolution graphics to produce good looking pictures.

## 5.1 Bar graphs

Bar graphs can display quite a bit of information at a glance. For instance, you might create a graph to show sales for each month of the year. Unfortunately, the BOX command for the C-128 is mediocre at best. The program below makes bar graphs much easier to produce. You can use this as a subroutine in one of your programs. Just set the parameters for the subroutine in your program and type GOSUB 60500.

You can use graphic characters to make some impressive bar graphs. We can display 320 pixels horizontally, which is 40 characters (each character is 8 pixels wide).

For each line of the graph, we must calculate how many characters we need in reverse video and how many we need to fill in the rest of the line. The easiest way to do this is to set up an array. Here is the listing for the subroutine:

```
5 rem bar graph 5.1
10 dimxa$(7):fori=0to7:readxa$(i):next
15 printchr$(147)
20 data " ","{CMG}","{CMH}","{CMJ}","{CMK}","
{RVS} {OFF}","{RVS} {OFF}","{RVS} {OFF}"
30 y=120:x=1:v=1 :c0=3
32 gosub 60500
35 y=220:x=4:v=1 :c0=2
50 gosub60500: end
60500 ym=320-v*8:an$="":ify>ymthheny=ym
60510 xa=y/8:g=int(xa):xy=(xa-g)*8
60520 ifg>0thenfori=1tog:an$=an$+"{RVS} {OFF}
":next
```

```
60530 an$=an$+xa$(xy)
60540 c1=peek(235):c2=peek(236):c3=rclr(1)
60550 color 5,c0:char1,v,x,an$
60560 color 5,c3:char 1,c2,c1,"":return
```

A note about line 20. The commands in brackets represent the keystrokes of the characters. For example, {CMG} means to press the <C=> key and "G" at the same time.

The characters used in the program are loaded in the first few lines. You may wish to use the ASCII codes for the characters just to be on the safe side:

```
32, 165, 180, 181, 161, 182, 170, 167
```

You call this routine with GOSUB 60500. Store the length of the bars (1-320) in variable Y. Store the line number at which the bar is drawn on the screen in variable X (0-24). The code for the color of the bar is stored in C0. The location of where the bar should stop is stored in variable V. Line 60500 calculates the length of the bar and resets AN$. Line 60510 calculates the number of characters needed to fill in the length of the bar (G), and the number of characters needed to fill in the rest of the line (XA). Line 60520 stores all the characters in the AN$ string. Line 60530 fills in the rest of the AN$ string.

To preserve the normal PRINT command we save the current cursor position as well as the color (C1, C2, C3--line 60540). Then we set the cursor to its new location and new color (C0), and draw the bar. The last line puts the cursor back and returns the subroutine to the main program.

You can use this routine for many applications, just remember to stay in lower-case mode.

## 5.2 Types of operations in character mode

This section answers the question of where the upper- and lowercase characters come from. Let's say that the number 1 is stored in video RAM in location 1024. This means the letter A would appear at the current screen location. The form of the character is determined (for A as well as for every character) by the character ROM. The character ROM occupies memory from 53248 through 57343.

Each character image takes up 8 bytes in the character ROM. This creates the character matrix of 8x8 pixels. Each line of pixels is one byte. Each pixel is represented by one bit. If the bit is set to a 1, then the pixel is turned on to the color in the corresponding color RAM. If the bit is a 0, then the pixel in the matrix is the color of the background color. To display the matrix for a character, you can enter the screen code (not ASCII code) for the character and enter it into the following program:

```
 5 BANK 14
10 INPUT "Character code:";C
20 AD= 53248 + C * 8
30 FOR I= 0 TO 7
40 W=PEEK(AD+I)
50 FOR J=7 to 0 STEP -1
60 IF (W AND 2 ^ J) THEN
   PRINT "*";: ELSE PRINT ".";
70 NEXT J: PRINT: NEXT I
80 PRINT "PRESS RETURN": GETKEY A$: GOTO 5
```

This program allows you to display the bit pattern of the character in uppercase. If you want to see the character in lowercase change the base address (AD) in line 20 from 53248 to 55296, or add 256 to the screen code (example: C=1+256 = 257).

If the bit in variable W is a 1 the result of (W AND 2 ^ n) is a 1, and a "*" is printed. Otherwise the IF statement is false and a "." is printed.

As the title of the section indicates, the purpose of this section is to discuss all the modes of the VIC. However, the new modes are not directly supported by BASIC 7.0.

The EXTENDED-COLOR-MODE is similar to the normal text mode. The bits of the character pattern that are set to 1 are seen as a pixel in the foreground color. The color of the 0 bits can be different. They are set to the background color, 0-3 in locations 53281 - 53284.

The two high bits of the screen codes in the video RAM determine which of the 4 colors the 0 bit is set to. Consider these bits as individual numbers (number=Bit 7 *2 + Bit 6). This formula gives the number of the register (e.g. 1 0= 1*2+0=2).

These two bits are nothing more than pointers to a position in the character generator. Therefore, only six bits are needed and only the first 2^6 = 64 characters can be accessed.

The extended-color-mode can be turned on using:

```
POKE 53265, PEEK(53265) OR 64
```

and can be turned off using:

```
POKE 53265, PEEK(53265) AND 191.
```

The MULTI-COLOR-MODE, although somewhat more complicated, can result in some very nice output.

Recall that every screen character can only have as many as two different colors: the character color (from the color RAM) and the background color (from the VIC registers). Multi-color-mode, however, allows up to 4 colors for each character. This is possible because of a simplification of the pixel matrix.

The color RAM is responsible for the color byte. If bit 3 is not set (byte AND (2 ^ 3)=0), then everything remains as it was. Unfortunately, you can only use colors 0 through 7. You can use more colors, however, if bit 3 is set (the multi-color-flag).

If bit 3 is set to a 1, we are finally in the multi-colored mode. The normal 8x8 matrix is divided into two 4x8 matrices. Every two bits in the character generator are now tied to a single pixel. If both bits are set to 0 the color is set to the background color. If both bits are set to 1 the VIC gets the color from the color RAM. The other two possibilities (01 and 10) retrieve the color from the extended-color mode background color register. You can turn this mode on from BASIC using:

```
POKE 53270,PEEK(53270) OR 16
```

You can turn this mode off again with:

```
POKE 53270,PEEK(53270) AND 239
```

We run into a problem when we try to mix both graphics and text modes. These modes cannot be set using VIC, they can be set only with the operating system and then only by using some tricks. The VIC reports to the processor each time a graphic character is produced on the screen (this happens about 25 times every second). When this happens, the running program is interrupted long enough to switch from the graphics mode to the text mode. Since the processor is much faster than the VIC, any character can be placed on the screen. This has the effect of a mixed mode. In actuality, there are two modes which are being switched very rapidly.

This technique requires that the processor always has control of certain bits in the VIC. We can switch to the multicolor-text mode using the POKE command, which will at the latest 1/30th of a second change to the new mode. With the POKE command you can switch modes without a machine language program, but only in the C-64 mode. Such a machine program would be very short, and would require the numbers 120 and 96, using:

```
POKE 4864,120 : POKE 4865,96 : SYS 4864
```

This will turn off the interrupt which is responsible for graphics modes. There are some disadvantages to this, for the cursor disappears and the keyboard is no longer polled. This trick can be used for the multicolor-text mode if you just want to display some information (e.g.: Menus, pictures, etc.). To exit this mode use the following command:

```
BANK 15: SYS 27438
```

As long as the interrupt is off you can POKE around in the VIC without the operating system detecting any errors. Here is an example:

```
10 POKE 4864,120:POKE 4865,96:SYS 4864
20 POKE 53270, PEEK(53270) OR 16
25 FOR I = 1 TO 1000: NEXT
30 BANK 15:SYS 27438
```

As you can see the characters in this mode are very messy. Resourceful programmers copy the character generator into RAM and create their own characters.

Remember to get out of `BANK 15` when you are finished with the VIC register.

## SUMMARY OF THE OPERATION OF THE CHARACTER MODES

Turn on the Extended Color Mode:

```
POKE 53265,PEEK(53265) OR 64
```

Turn off the Extended Color Mode:

```
POKE 53265,PEEK(53265) AND 191
```

Turn on the Multi-color mode:

```
POKE 53270,PEEK(53270) OR 16
```

Turn off the Multi-color mode:

```
POKE 53270,PEEK(53270) AND 239
```

In C-128 mode you have to turn off the interrupts:

```
POKE 4864,120: POKE 4865,96: SYS 4864
```

To turn the interrupts back on:

```
BANK 15: SYS 27438
```

## 5.3 Character generator confusion

In C-128 mode, RAM is repeatedly moved through interrupt calls so that the memory area in ROM that controls the VIC's character generator can be accessed.

Bits 1 through 3 in memory location 53272 contain the address of the character generators. The contents are dependent upon other factors and are more difficult to change. Therefore, all of the following statements are for the normal configuration.

The following table shows which bit configuration points to each memory region:

| COMBINATION | REGION |
|---|---|
| 000 | 0 |
| 001 | 2048 |
| 010 | Upper Case |
| 011 | Lower Case |
| 100 | 8192 |
| 101 | 10240 |
| 110 | 12288 |
| 111 | 14336 |

The combination of 010 and 011 are a special cases. They address the character ROM (53248 through 55296).

The best way to access the three bits in location 53272 is to use the AND and OR functions. AND the byte with the binary number 1111 0001 (=241) to change the 3 bits in the byte. This will cancel all the bits which were set to 1 back to a 0. Then POKE the actual bit combination that you want into the byte using the OR function.

To have the character generator use the region at 2048, the bit combination 001 must be placed into address 53272. However, we can't change bit 0 of this byte--it is always set to 1. We instead create the decimal equivalent of the bit combination, in this case a 1. We have to shift the bit combination one bit to the left. To do this multiply the number by 2. The entire command looks like this:

```
POKE 53272,(PEEK(53272)AND 241) OR 2
```

You can place the character page in the graphics region, but you cannot use graphics at any time. You can protect this region by using:

```
GRAPHIC 1: GRAPHIC 0
```

One of the advantages of this is the fact that you can turn on the entire character page just by turning on the graphics.

To automatically load the character generator you have to use a loading program (See Section 3.3).

You can use the following code:

```
 5 REM Copying the Character page for the C-128
10 BANK 14
20 FOR I=0 TO 4095:POKE 8192+I,PEEK(53248+I):NEXT I
30 BANK 15
```

When you have copied the character ROM to RAM and have changed the pointer, the characters retain their old form only because they are generated from the patterns in RAM rather than ROM. They are now easy to change using POKE's. Characters are defined similar to sprites. Only the form of the matrix is different (8x8 instead of 21x24) in memory.

To examine the pattern you can use the program listed in Section 5.2. You will have to change the base address in line 20 from 53248 to 8192.

The new characters are now easy to POKE. There is no limit to their creative applications, especially in the multicolor mode.

You can turn off the normal generator and get at the new character page by using:

For upper case.

```
POKE 53272,(PEEK(53272)AND 241) OR 4
```

For lower case.

```
POKE 53272,(PEEK(53272)AND 241) OR 6
```

When we turn off the interrupts in the C-128 mode, we have to take care of some other chores that the interrupt normally does itself. After every interrupt, the memory location 2604 is copied into register 53272 where the operating system retrieves its information. We have to do this ourselves when we disable the interrupts. This is why it is best to turn off the interrupts only when we have to.

The C-128 mode offers yet another way to define characters. Memory location 217 determines if the VIC uses the character generator in ROM or from the RAM in BANK 0. Normally this location is set to 0. This indicates that the character generator is located in ROM. We can change this location using POKE 217,4. This would allow the VIC to get its characters from the region between 4096 through 8191 (without the need for interrupts). Normally, this would cause a lot of interrupts from the BASIC interpreter. There are only 256 bytes, from 4096 through 4351, that we are allowed to use. The function key assignments are normally saved in this region. If we ignore these definitions, we can define 256/8 = 32 different keys. Unfortunately, the first 10 bytes (4096-4105) of this memory area must be set to 0. This means we will only have room for thirty characters. The first will be composed entirely of zeros, and the second will have zeros for its first two bytes. You can change the remaining 30 bit patterns as you please, though, and POKE them into memory. Remember that you can PEEK the locations following 4351 but you cannot change them.

Of course, 32 characters are not very many but they are enough for most purposes.

This is the same for the region of 6144 to 8191. This is the color matrix for the high-resolution graphics. If you want to use this region you have to be in the graphics mode.

## SUMMARY OF THE CHARACTER GENERATORS

Bits 1-3 of the memory location give the location of the generator. You have to be aware that the BASIC memory region is protected (See section 3.3)

To turn a page on:

```
POKE 53272,(PEEK(53272)AND 241) OR X
```

To turn off a page:

```
POKE 53272,(PEEK(53272)AND241) OR 4 (6)
```

In the C-128 mode change the location from 53272 to 2604.

The second possibility for the C-128: VIC gets its information from RAM

```
ROM off: POKE 217,4
ROM on : POKE 217,0
```

## 5.4 Video RAM confusion

Similar to the character generator, video RAM is located in the memory from 53272 (2604: C-128).

The location of video RAM is set by register 24. The controller bits for register 24 are 4 through 7.

With these four bits you can alter the location of video RAM in one kilobyte increments. Normally only bit 4 is set. This selects the memory range from 1024 through 2023. Once again here is another table of bit combinations:

| The Value of Bits 4-7 in Register 24 | Location of Video RAM |
|---|---|
| 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 from the bit combination, the ROM is an exception. You must note that the VIC uses the character generator in ROM. These regions are in memory from 4096 through 8191, instead of at 53248 for the VIC. These can be changed and used with AND, OR, PEEK, and POKE. The high four bits are set to zero. This is most easily done using AND 15. The binary combination is set from decimal. If we wanted to use the video RAM from 15360 we should use 15. This number must be multiplied by 16 to move the bits to the proper location. It is then OR'ed with the number:

```
POKE 53272,(PEEK(53272)AND 15) OR X
```

Is there a connection between the number X and the binary combination above? Yes, there is! This determines how much memory in kilobytes the video RAM will take up. In the future you will not have to do the wearisome calculations to determine how many K this should take up by using this command:

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

This command can be a bit of a disappointment. You may see an awful mess of characters on the screen. We have forgotten the operating system region where the video RAM is located. The VIC is getting all of its information about what to display from a place where the operating system is non-existent. If you are in this situation you can press the <CLR> key and the operating system will clear video RAM and color RAM.

The computer uses the high-bits of memory location 2619 (C-128) or 648 (C-64) to store the video RAM starting address. This number multiplied by 60 is the actual address. For example, if we want to start at 15360: 15360/256 yields 60. If we use `POKE 2619,60` the C-64 is back in order. Luckily, there is a method that makes things easier. If we multiply the kilobyte number by 4 we get the high-byte. Unfortunately, the C-128 is somewhat stubborn, the `INPUT` command only works on the normal video page, unlike the C-64.

There are also a few things that you must pay attention to if you are also going to use sprites. The sprite pointers are located between 16376 through 16383. Once again you have to remember to protect the BASIC program storage when you change the video RAM.

It is possibile to define two different screen pages and then switch between them. The `PRINT` command can only access one of the pages, and the other has to be accessed using `PEEK`'s and `POKE`'s. Another problem is that the color RAM has to use the same color for both pages.

## SUMMARY OF THE VIDEO RAM

The video RAM is controlled by bits 4-7 in location 53272. The number of kilobytes that the monitor screen memory is to take up is set by K:

```
POKE 53272,(PEEK(53272)AND15)OR K*16
POKE 648,K*4
```

The C-128 uses location 2604 of the VIC switch and 2619 for the address high byte.

## 5.5 Different tricks for the 40-column screen

As with the normal character mode there are certain tricks for programming of normal text.

Let's begin with colors. The color register for the VIC is in a specific location. You can only change bits 4 through 7. The color codes vary from 0 to 15 and so only bits 0 through 3 are used.

Once in a while, you might have to access a byte at a specific memory location in the video RAM. There is a pointer to the beginning of each screen line at locations 224 and 225 (C-64: 209/210). If you add the contents of location 236 (C-64: 211) to this value, you have the address at which the cursor is located in video RAM.

There is also a pointer for the color RAM:

```
PRINT PEEK(226)+256*PEEK(227)
```

This points to the address of the beginning of the line for the color RAM. The C-64 mode is somewhat different, the pointer in bytes 243/244 directly gives the location of the cursor.

You can position the cursor to a specific screen location before a `PRINT` command. In BASIC 7.0 it is very easy—just use the `CHAR` command. The same ROM routine can be called for the C-64 mode using the following command:

```
POKE 214,line:POKE 211,column:SYS 58732
```

Have you ever wanted to turn the cursor on during input with the `GET` command? This really isn't very hard. Location 2599 (C-64: 204) informs the the interrupt routine if the cursor should be displayed or not (in this case a `PEEK(2599)` produces a zero). When we start a program the location 2599 is set to 1 by the interpreter and the cursor begins to blink.

We will look at what the operating system does in a short while. By using POKE 2599,0 the cursor is displayed. The interrupt routine doesn't think about it again. If we turn off the cursor with a POKE 2599,1 we have to make sure that the cursor is not in the middle of a blink, therefore leaving a reverse video character on the screen. This is very easy to do in the C-64 mode. All we do is wait using an IF statement until PEEK(207) becomes zero, then we can turn off the cursor.

Unfortunately, all we can do in the C-128 mode is print a blank character at the same location the blinking cursor was at to take the cursor off the screen.

To keep the input here is a tip for the next INPUT command:

```
INPUT "text(crsr right)(crsr right) Z (crsr left)
(crsr left)(crsr left)";A$
```

The Z is used as a cursor character upon issuing the INPUT command. As soon as a key is pressed the character disappears.

The next commands use the reverse video mode. Independent of whether or not the input string has a special control character in it you can use:

POKE 243,1 (C-64: POKE 199,1)

to print the string in reverse video. You can return to normal mode using:

POKE 243,0.

Do you want to print the control character of a string on the screen? Location 247 (C-64: 216) has the command. This contains the quantity of inserts. As we know, control characters are not printed in the insert mode, rather they are printed in reverse video. This mode can be turned on using POKE 247,X where X is the number of characters to be output.

Once more we are going to play around with the operating system. If you have used the Datasette then you know that during a cassette operation the monitor screen goes blank. Once again the VIC is responsible. Bit 4 of location 53265 decides if the screen should be displayed.

To turn the screen off use:

```
POKE 53265,PEEK(53265) AND 239
```

To turn the screen back on again use:

```
POKE 53265,PEEK(53265) OR 16
```

This should be used only for static (no movement) pictures since the VIC only periodically operates on the screen.

Perhaps you were working with the text window but somehow the screen was lost. Luckily, there is the WINDOW command. The command uses either 2 or 4 coordinates. If the window size does not change, you can get these coordinates using a PEEK. The location 228 gives the height and location 229 the bottom edge. The borders are found in bytes 230 and 231.

## SUMMARY OF 40-COLUMN SCREEN TRICKS

Actual position in the COLOR-RAM:
for the C-64:

```
PRINT PEEK(243)+256*PEEK(244)
```

for the C-128:

```
PRINT PEEK(226)+256*PEEK(227)+PEEK(236)
```

Actual position in video-ram:
of the C-64:

```
PRINT PEEK(209)+256*PEEK(210)+PEEK(211)
```

for the C-128:

```
PRINT PEEK(224)+256*PEEK(225)+PEEK(236)
```

Cursor column:

```
PRINT PEEK(236)    C-64: 211
```

Cursor line:

```
PRINT PEEK(235)    C-64: 214
```

Moving the cursor:

```
POKE 211,column:POKE 214,line:SYS 58732
```

Turning on the cursor:

for the C-64:

```
POKE 204,0
```

for the C-128:

```
POKE 2599,0
```

Turning off the cursor:

for the C-64:

```
POKE 204,1 (PEEK(207) blinking phase)
```

for the C-128:

```
POKE 2599,1
```

INPUT with a special cursor:

```
INPUT"text(2*crsr-right)Z(3*crsr-left)";A$
```

Turning on reverse video:

```
POKE 243,1 (C-64: 199)
```

Turning off the reverse video:

```
POKE 243,0 (C-64: 199)
```

Alternative character mode:

```
POKE 245, X (C-64: 216)
```

Turning off the screen:

```
POKE 53265,PEEK(53265) AND 239
```

Turning on the screen:

```
POKE 53265,PEEK(53265) OR 16
```

# Chapter 6

# High resolution graphics

# High resolution graphics

The C-128 mode makes graphics very easy—just turn on graphics, then set or clear the pixels. Although the graphic commands are flawless, there is still a little more that you might want from your graphics.

## 6.1 The graphics modes

The section below is similar to one in the Commodore 128 System Guide, but for the extra features we want, it helps to understand the hardware.

As with the normal character display, there are also different modes for high resolution graphics. There is no extended-color mode in high resolution graphics. In the normal mode we can access 320x200 pixels. These 64,000 pixels can be accessed with 8000 bytes (much like the character generator). This region of memory is called the *bit-map*. The bit-map is similar to a elevation map, but in place of hills and valleys we have bits that are set to 1 and/or 0. These bits are represented on the screen by pixels. The color of the pixel is determined by the video RAM (not by the color RAM). Every byte in the previous screen memory represents a specific region, which in the normal mode represents a character. The four high bits, which have been set to 1, determine the color (0-15) of the pixel. The low four bits, which are set to 0, are for the background color.

When the `GRAPHIC` command is given, the computer automatically saves the video RAM in another location so that the old, screen text is not destroyed.

In the multicolor mode the pixel matrix is made smaller. Instead of 320x200 pixels we now have only 160x200 pixels to use. Every pixel now requires 2 bits to be displayed, which means we still need 8000 bytes for the bit map, but each cell can be any of 4 colors. The colors no longer come from the old video RAM, but from both the background color register 0 and the color RAM. The two bit pixel combination derives the colors as follows:

| | |
|---|---|
| 00 | from register 53281 (background) |
| 01 | from the upper four bits of video RAM |
| 10 | from the lower four bits of video RAM |
| 11 | from color RAM |

## 6.2 The bit map

Now something about the location of the bit map in memory. Once again location 53272 has the information we are looking for. The state of bit 3 determines where the bit map is located. If bit 3 is 1, the bit map starts at location 8192, otherwise the bit map starts at location 0. We will usually avoid the zero page, for we could overwrite part of the operating system if we are not careful .

The set up of the bit map is similar to that of the character generator. The first 8 bytes give the 8 pixel lines for the first quadrant. This means when you turn on the graphics you can still display normal characters on the screen. All you have to do is copy the character generator into the bit map and change the bytes.

The multicolor mode is also quite similar. The only difference is that each pixel is twice as wide and requires two bits (as we stated in chapter 5).

The location of the bit map makes it necessary for us to protect the program storage area. In the C-128 mode, the computer takes care of this for us, however in the C-64 mode we have to do this ourselves.

## 6.3 Turning on graphics in the C-64 mode

To turn on the graphics we have to follow three steps. Then we must protect the bit map area in memory. We can do this with a loaded program (when it is finished loading) with the following commands:

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

This should be done in direct mode. Now the graphics can be turned on by the program. In order to do this bit 5 in location 53265 must be set to 1. This tells the VIC not to display characters but to display high resolution graphics. If you want more colors, you have to set the multicolor bit in location 53272 to a 1 (just like in the character mode). Then we set the location of the bit map by setting bit 3 of location 53272 to one. Finally we have to save the video RAM so that we don't destroy the screen when the graphics mode is activated.

When you finish this you will probably see a somewhat chaotic screen display. We still have to perform one last step. Using a FOR-NEXT loop we have to clear every byte of the bit map, because the video RAM information is still there. Here are the complete set of commands to do this:

```
POKE 43,65:POKE 44,63:POKE 16192,0:CLR:
  REM MEMORY PROTECTION
POKE 53265,59: REM TURN ON GRAPHIC MODE
(POKE 53270,216:REM MULTICOLOR MODE)
POKE 53272,40: REM BITMAP+VIDEO RAM
FOR I=8192 TO 16191: POKE I,0:NEXT:
  REM CLEAR MEMORY FOR BIT MAP
FORI=2048 TO 3047: POKE I, pixel-color * 16
  +background color: NEXT : REM SET COLOR
```

After these POKE's video RAM occupies the area of memory from 2048 to 3047. Starting in location 3072 we find another 5K for such things as sprites and machine language routines.

All graphics must come to an end, so to return everything back to normal use the following commands:

```
POKE 53265,155: REM TURN OFF GRAPHICS
(POKE 53270,8: REM TURN OFF MULTICOLOR)
POKE 53272,21: REM TURN ON CHARACTERS
```

If you have been trying these graphics tricks, you have probably struggled with the slow process of clearing the bit map. To speed things up a bit we have written this machine language routine listed below.

```
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,
       202,2088,246,160,232,145,
       251,136,208,251
4 DATA 141,0,11,96
```

You can start this program by entering `SYS 3600`. It will clear the bit map and then load the video RAM (2048-3047) with the foreground and background colors. Which color is loaded depends on location 2.

To set the colors you want use this statement:

`POKE 2,`*`color`* `*16 +` *`background color`*

This machine routine can be relocated to any area of memory. The beginning address is always the byte where the `FOR-NEXT` loop in line 0 will start. This will take only a fraction of the time than using BASIC.

Finally, a small tip if you want to use sprite graphics, colors and the clearing routine all in the main program on a diskette or cassette. This is actually quite easy. After loading the sprites, machine routines and so forth in a protected region of memory, the pointer in locations 43/44 is pointing to the normal beginning spot for BASIC. Naturally, you can save memory by changing the pointer to a higher address. For example, if the color RAM does not have to be saved. This program can (using only loading routines, with the pointer set) be loaded using `LOAD "`*`name`*`",8,1`. Now the graphics, sprites, etc. are all in memory. This is very helpful when programming games.

## SUMMARY OF GRAPHICS IN THE C-64 MODE

With the following POKE's you can turn on high-resolution graphics and multicolor graphics. The video RAM lies between 2048 and 3047, and the BASIC start must be moved to 16192.

```
POKE 53265,59: REM HIGH RESOLUTION ON
(POKE 53270,216: REM MULTICOLOR ON)
POKE 53272,40: REM VIDEO RAM SAVED
```

Now to return the video RAM from its saved location and remove the bit map.

```
POKE 53265,155: REM TURN OFF GRAPHICS
(POKE 53270,8: REM TURN OFF MULTICOLOR)
POKE 53272,21: REM TURN ON TEXT+
```

## 6.4 Setting pixels in C-64 mode

When you want to set a specific point on the graphics page, you can first of all draw your picture on graph paper and calculate the appropriate RAM location. You may correctly surmise that this method would take several days and nights of calculation and plotting (not to mention a few hundred tablets of aspirin). To preserve your sanity we have included here two routines that make the process faster and easier.

### 6.4.1 Setting pixels in the high resolution mode

The program listed below works on the same principle as the block graphic routine in section 5.1. This time we don't have to POKE special graphic characters so we don't need a table to determine where we are going to put the character.

```
61000 REM SET AND CLEAR PIXELS HIGH-RES
61010 Y=199-Y: IF Y<0 OR Y>199 RETURN
61020 IF X<0 OR X>319 THEN RETURN
61030 X1=INT(X/8): X2=INT(X/8):
      AD=8192+8*X1+320*X2+(Y AND 7)
61040 X3=2^(7-(X AND 7)):
      CA=2048+X1+40*X2
61050 POKE CA,(PEEK(CA) AND 15)OR 16*CO
61060 IF L=1 THEN POKE AD,PEEK(AD) AND
      (255 - X3) : RETURN
61070 POKE AD,PEEK(AD) OR X3: RETURN
```

The routine is invoked using GOSUB 61000. The coordinates X(0-319) and Y(0-199) give the point where the pixel should be set. The coordinate system's origin is in the lower left corner. The coordinates for X and Y cannot be set outside of their limits so lines 61010 and 61020 will ignore the command if either value is outside of its proper range.

Line 61030 calculates the address of the byte within the bit map that should be changed. The part INT(X/8) calculates the column in the color RAM. This word is multiplied by 8, so each cell in the color RAM is 8 bytes long and represents a location in the bit map. INT(Y/8) calculates the line in the

color RAM. In order to get to the actual address in the bit map, this word is multiplied by the value of the pixel in line 320. Y AND 7 automatically calculate the color quadrant.

The variable X3 tells the word which bits should be set. AD has the address of the pixel to be POKE'd, CA has the color. Line 61050 POKE's the color CO(0-15) in the high 4 bits of the color cell. Don't forget that changing the color of a pixel will change the color of the entire quadrant.

If L=1, then the routine clears the given pixel. In this case line 61060 is executed, otherwise the line sets the pixel to 1. A typical call for this routine is:

```
 5    POKE 53265,59: POKE 53272,40
10    X=100: Y=25: REM SET THE COORDINATES
15    CO=2: L=0: REM color=red, mode=set
20    GOSUB 61000: REM CALL ROUTINE
```

## 6.4.2 Pixels in the multicolor mode

In the multicolor mode you have to set 2 bits for each pixel, for every color combination. You can use the earlier method, but you will have to call the setting routine twice. For the first bit, simply double the X-coordinate. For the second bit, just add one to the doubled X-coordinate value. Each of the two bits are in the same byte. With a few simple changes you can do this and avoid calling the routine twice:

```
61000 REM MULTICOLOR PIXELS
61010 Y=199-y:IF Y<0 or Y>199 THEN RETURN 61020
      XX=2*X:IF X<0 OR X>318 THEN RETURN
61030 X1=INT(XX/8): X2=INT(Y/8):
      AD=8192+8*X1+320*X2+(Y AND 7)
61040 X3=2^(7-(XX AND 7)):
      X4=2^(7-((XX+1) AND 7))
61050 POKE AD,PEEK(AD) AND (255-(X3+X4))
61060 POKE AD,PEEK(AD) OR ((CO AND 2)/2*X4): RETURN
```

Once again this is called with GOSUB 61000, and the coordinates are the same as before [X (0-159), Y (0-199)]. The color and set/clear modes are no longer needed. The number for the color (0-3) is in CO. To clear a pixel

simply set CO to 0. The color from the bit combination is set using POKE to the needed register (the video RAM takes care of the clear routine). With just a little change the lines 61010 through 61030 are still pretty much the same as with the high-resolution mode.

Line 61040 combines the bit combination in variables X3 and X4. Then the two bits are cleared (line 61050). Finally, the bit combination of the affected pixel is set in line 61060. The CO AND 1 puts the combination in the lower byte, and (CO AND 2)/2 puts it in the high half. If a bit is zero the entire product is zero. The reason: the affected bit from memory OR'd with zero always produces zero. If the bit is a one and the bits are OR'd, the result is always a one.

Here is an example of a typical call to the subroutine:

```
5   POKE 53265,59: POKE 53272,40
10  X=100: Y=50: REM coordinates
15  CO=2: REM color for the high bits in the
    video RAM
20  GOSUB 61000: REM Call to the routine
```

## 6.5 Spirals

After this indepth explanation of the C-64 mode graphics manipulations, we'll return to the graphics commands of BASIC 7.0.

Very few figures have as many variations as the spiral. There is really no simple program for such a figure, although the principle is always the same. In order to understand the spiral we have to take a small step backwards and draw a circle "by hand." That is what the following program does:

```
 5 PI = π
 6 GRAPHIC 1,1
10 FOR I=0 to 359
20 DRAW 1,160+SIN(I/180*PI)*100,100 + COS(I/180*PI)
   * 100
30 NEXT
```

Look how easy it was to set 360 points. The coordinates are the SIN and COS multiplied by the radius 100. Experiment with different numbers. A spiral is basically a circle with a constantly growing radius (mathematically we have to apologize for this imprecise explanation). The following program does this for us:

```
 6 GRAPHIC 1,1
10 LOCATE 160,100
20 FOR I=0 TO 8000 STEP 10
30 X=COS(I/180*PI)*I/80+160:
   Y=SIN(I/180*PI)*I/80+100
40 DRAW TO X,Y: NEXT
```

You can produce an inifinite variety of spirals by changing the parameters of the above program. For example, the number of windings is controlled by the program loop length; the smoothness of the spiral is controlled by the loop's step width; the distance between the spiral windings is controlled by the radius, etc.

## 6.6 Pie charts

Piechart graphics can be displayed with the BASIC 7.0 quite easily. Below we have listed a small subroutine which you can tie into one of your programs:

```
    5  REM PIECHART 6.6
    6  GRAPHIC 1,1
   10  F=1: X=150: Y=100
   20  XR=75: YR=50
   30  AZ=10
   35  GOSUB 61000:END
61000  REM PIE CHARTS
61010  WB=0: FOR II=0 TO AZ-1
61020  WA=WB:WB=WB+W(II)*3.6
61030  CIRCLE F,X,Y,XR,YR,WA,WB
61040  DRAW TO X,Y: NEXT
61050  IF XR>YR THEN CIRCLE 1,X,Y+8,XR,YR,85,275:
       PAINT 1,X,Y+YR+5
61060  RETURN
```

Before you call this routine with a `GOSUB 61000`, you have to set certain variables. The value in `F` is the color, `X` and `Y` have the coordinate of the center of the circle. `XR` and `YR` are both radii. If `XR` is greater than `YR` we get a 3D effect. This is accomplished in line 61050.

This program has values which describe how the diagram looks. `AZ` contains the number of points which are going to be in the circle. Notice that all elements must add up to 100.

To realize how the graph is made you have to understand how the parameters to the `CIRCLE` command work. The `DRAW` command draws a line from the middle of the circle to the border. Line 61050 determines if the circle will have a 3D look to it. In this case we `DRAW` an ellipse under the circle, and fill in the area.

## 6.7 Painting program

You've probably heard of computer-aided design programs, like ***Cadpak***from Abacus, that let you easily create sophisticated graphic designs on the screen. These CAD packages offer you many design possibilities. For example, you can create your own greeting cards with a specialized motif, and then print them out. Here is a simplified version of such a CAD program:

```
 10  fori=1to67:reada:a$=a$+chr$(a):next
 20  graphic1,1:locate160,100:x=160:  y=100:f=1:
 30  sprsav a$,1:sprite 1,1,2,0,0,0,0
 40  movspr 1,173,140
 50  a$="":getkey a$
 60  ifa$="(crsr down)" then if y>0 then
       movspr1,+0,-1: y=y-1: goto50
 70  if a$="(crsr up)" then if y<199 then
     movspr1,+0,+1:y=y+1: goto50
 80  if a$="(crsr left)" then if x>0 then
     movspr1,-1,+0:x=x-1: goto50
 90  if a$="(crsr right)" then if x<319 then
     movspr1,+1,+0:x=x+1: goto50
100  ifa$=" "then drawf,x,y: xa=x: ya=y: goto50
110  if a$=chr$(13) then drawf,xa,yatox,y:  xa=x:
     ya=y: goto50
120  ifa$="p"then paintf,x,y: goto50
130  ifa$="f"then f=1-f: goto50
140  ifa$="s"then 1000
150  ifa$="l"then 1100
160  ifa$="t"then 1200
170  goto 50
1000 graphic0: print"save bit map"
1010 input"Qfilename";f$:bsave(f$),d0,u8,onb0,p8192
1020 graphic1:goto50
1100 graphic0:print"load bit map"
1110 input"filename";f$:bload(f$), d0,u8,onb0,p8192
1120 graphic1:goto50
1200 getkeya$:ifa$=chr$(13)then50
1210 charf,x/8,y/8,chr$(14)+a$: if x<31 then x=x+8:
     movspr1,+8,+0:goto1200:else50
9000 data0,16,0,0,16,0,0,16,0,0,16,0
9001 data0,16,0,0,16,0,0,16,0,0,16,0
```

```
9002 data0,0,0,0,0,0,255,131,254,0,0,0,0,0,0,16,0,
     0,16,0
9003 data 0,16,0,0,16,0,0,16,0,0,16,0,0,1,6,0,0,16,
     0,23,0,20,0
```

A sprite can be used as the graphic cursor. It is always located over a pixel. The cursor is moved using the cursor keys. The space bar sets an individual pixel. Pressing the <RETURN> key draws a line from the last pixel that was set to the pixel at the current cursor location. You change the color of a pixel (0 or 1) just by pressing the F-key. P starts a PAINT command at the actual cursor coordinates.

If you want to save your painting at some point, S will start the SAVE routine. You are prompted for a filename. The bit map will be saved to diskette. The LOAD command is started with the L key and operates in a manner similar to the SAVE routine.

There is a small treat with the T key. This key will put us in the text mode—every key stroke is displayed on the screen in the approximate position of the current pixel with the help of the CHAR command. You stay in this mode until you press <RETURN>.

## 6.8 Hardcopy

If you want to see your results from the painting program in black and white you need a hardcopy routine. Actually, this is best handled by a machine language program. Printing a picture requires that the entire graphic page (64000 pixels) be loaded, coded, and sent to the printer. This takes quite a while. We have two hardcopy programs in memory because the codes that must be sent to specific printers are quite different.

The first program is for the MPS-801 or compatible printer. The MPS-801 can print 7 vertical pixels at a time. Therefore, the bit map must be accessed with 28 lines of 7 pixels and another line with 4 pixels in the high bytes. In all, this gives 320 columns. The inner `FOR-NEXT LOOP` gets the 7 pixels and builds a variable `BY`. When this byte is finished it is sent to the printer (Line 60). After 320 columns, the printer head has to return to the leftmost position and the paper is fed one line. Then `PRINT#1` prints the control characters (Line 70).

The second program is for the EPSON or compatible printer. It is almost the same as the first program. The MPS-801 automatically sets its graphic mode. The Epson does not do this, so we have to send a special control character to do this before sending the rest of the data. The graphic mode is activated for only a single line, so this mode must be set before each new line is printed. Other than this, the only remaining difference with the previous routine is that the Epson can print 8 pixels at a time. Therefore, the inner `FOR-NEXT` loop is a little different.

With both programs you'll notice just how slow the BASIC interpreter really is. To reduce these effects we can set the `FAST` mode.You can also get rid of all the `REM` lines to save some execution time. Don't worry if nothing happens for a few seconds after you start the program.

```
1     REM ++++++++++++++++++++++++++++
2     REM MPS-801-GRAPHIC HARDCOPY 2.0
3     REM ++++++++++++++++++++++++++++
5     GRAPHIC1,1:FORI=100TO220STEP10:
      CIRCLE 1,I,100,100:NEXT:GETKEY A$
10    FAST:OPEN1,4,7:PRINT#1,CHR$(8);
20    FORZ=0TO27
30    FORX=0TO 319:BY=128
40    FORY=0TO6
50    LOCATE X,Y+Z*7:BY=BY+RDOT#(2)*2^Y
60    NEXTY:PRINT#1,CHR$(BY);
70    NEXTX:PRINT#1
80    NEXTZ
90    FORX=0TO319:BY=128
100   FORY=0TO3
110   CONTX,Y+196:BY=BY+RDOT#(2)*2^Y
120   NEXTY:PRINT#1,CHR$(BY);
130   NEXTX:PRINT#1
140   PRINT#1,CHR$(15);:CLOSE1
150   SLOW
160   GRAPHIC0


1    REM ++++++++++++++++++++++++++++++++
2    REM EPSON-GRAPHICS-HARDCOPY 2.0
3    REM ++++++++++++++++++++++++++++++++
5    GRAPHIC1,1:FORI=100TO220STEP10
     CIRCLE1,I,100,100:NEXT:GETKEY A$
10   FAST:OPEN1,4,1:PRINT#1,CHR$(27);
     CHR$(51);CHR$(23);
20   FORZ=0TO24:PRINT#1,CHR$(27);
     CHR$(75);CHR$(64);CHR$(1);
30   FORX=0TO319:BY=0
40   FORY=0TO7
50   LOCATEX,Y+Z*8:BY=BY+RDOT#(2)*2^(7-Y)
60   NEXTY:PRINT#1,CHR$(BY);
70   NEXTX:PRINT#1,CHR$(13);CHR$(10);
80   NEXTZ
90   CLOSE1
100  SLOW
110  GRAPHIC0
```

## 6.9 3-D graphics

Three-dimensional displays are undoubtedly the most interesting type of graphics. However, they also take the most time to produce. We have listed a small program to make things a little easier.

We have to change the coordinate system. In contrast to the normal two axes, we now have three axes: x, y, and z.

The z-axis is "slanted" to provide the needed perspective (see Diagram 6.9.1).

Since the DRAW command doesn't have parameters for three dimensions, we have to transform the three dimensions to two dimensions. This is called a *projection.* We will just work with simple parallel projection. In reality, parallel lines run into the distance without ever converging.

As you can see in the picture, the z-coordinate has to be moved either more or less to the right . There are two points on the z-axis with the coordinates (0,0,za) and (0,0,zb). As you see, the projection coordinates are determined through the addition of the factors za*cos(a) and zb*sin(a). The projection formula is:

```
x=0+za*cos(a)   y=0+za*sin(a)
```

or in the more general form:

```
x=xa+za*cos(a)   y=ya+za*sin(a)
```

Because this 2D coordinate system is not always the same as the screen coordinate system, we have to have a dimensioning factor (sx,sy)—(See program listing, line 100). The coordinate origin is determined by adding the needed factor to position the picture in the middle of the screen.

From the three original coordinates only one needs to be computed, so we can use two nested FOR-NEXT loops.

In the program below, all the values that must be altered often (for experimentation) are entered with INPUT commands. The program runs in graphic mode 1.

Type in the following program and run it using the following values:

```
w=45, sx=0.5, sy=0.5, xs=2, zs=20
```

```
1     REM ++++++++++++++++++++
2     REM 3D GRAPHICS PLOTTER
3     REM ++++++++++++++++++++
10    INPUT "Angle";w:
      z1=cos(w/180*PI):z2=sin(w/180*PI)
20    INPUT "X-dimension";sx
30    INPUT "Y-dimension";sy
40    INPUT "Step length X";xs
50    INPUT "Step length z";zs
60    GRAPHIC 1,1
70    FOR z=180 TO -180 STEP -zs
80    FOR x=-180 TO 180 STEP xs
90    y= sin(x/180*PI)*cos(z/180*PI)*50
100   bx=160+sx*(x+z*z1):
      by=100-sy*(y+z*z2)
110   DRAW 1,bx,by
120   NEXT x,z
```

Perhaps you noticed a problem with the program. A point that should normally be hidden by another shows through. To solve this problem, we must use what is called *hidden point removal.* Change line 110 to:

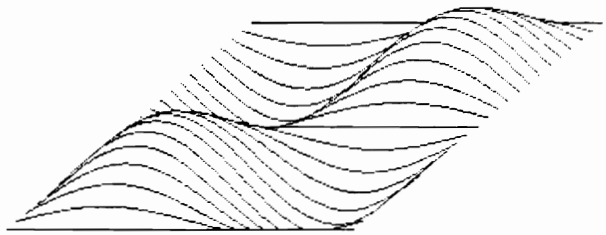
```
110 DRAW 1,bx,by: DRAW 1,bx,by+1 TO bx,200
```

We'll explain this with an example. We will draw a single surface. The surface is plotted obliquely from the bottom to the top in the space. The first point plotted is the one that is closest to the bottom—therefore no points can be below it. We then take the y-coordinate plus one and cover all points below. All points below this given point are not set. This method even works for the lines which contain curves. You will understand better if you run the program and see for yourself.

You might want to make a few changes to the program. Line 10 calculates the cosine and sine of the coordinates. Since these values are constants, they can be calculated once and saved in variables. This will save a lot of calculation time later on. The numbers in the FOR-NEXT loop (lines 70 and 80) can be changed to allow for smaller or larger steps.

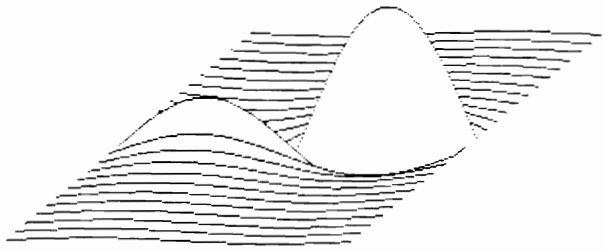
In line 90 you may change the value of the multiplier (50 in our program). This equation always produces values that are less than 1. The values only need to be set once. However, this value can be changed from function to function.

Experiment with different values to see what each value does to change the functions.



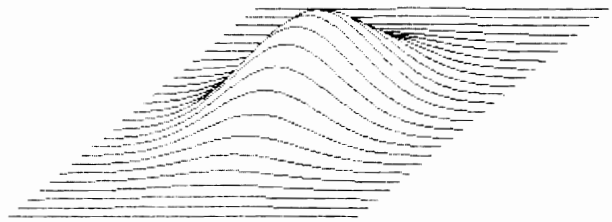
```
y = sin(x/180*PI) * (sin(z/180*PI) * 70
```

Diagram 6.9.1



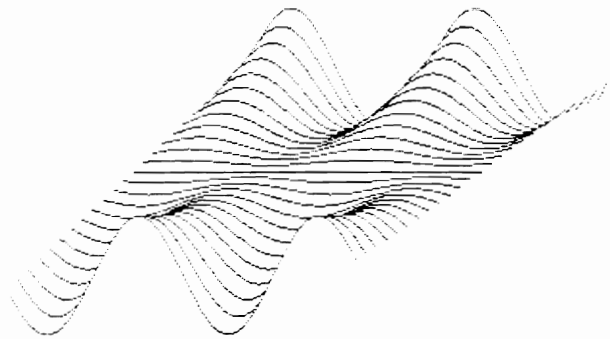
```
y = sin(x/180*PI) * 2 / (z/8+.5) * 40
```

Diagram 6.9.2



```
a  = x/180*PI   b = z/180*PI
y = exp(-(a*a+b*b)/2)/ sqr(2*PI)*300
```

Diagram 6.9.3



```
y = sin(x/30) * (exp(z/90) - 1 / exp(z/90)) * 10
```

Diagram 6.9.4

## 6.10 Raster line interrupts

Even though you may not have known what it was, you have undoubtedly used a *raster line interrupt* before. These are used primarily in graphic modes 2 and 4.

As you probably know, a TV or monitor screen is projected line by line. These lines are also called *rasters*. Using the VIC chips you can determine where in the raster a picture is to be placed. The operating system calculates at which raster line the division between graphics and text will take place. The VIC is programmed so that an interrupt is given at the right time. At the correct moment, the processor switches the program through the VIC from graphic mode to text mode. At the bottom border of the screen this little routine is repeated, except it is switched from text back to graphics.

You can thank the programmers at Commodore for this aggravation. In BASIC, the interrupt occurs after only 8 lines in the text mode, although the VIC issues more lines. Using this method, the highest pixels of a text line are mixed with the graphics. To change this you only have to alter location 2612. Here you can POKE the desired raster line into memory. This is explained in the following example:

```
10   GRAPHIC 2,1
20   FOR I=0 TO  100 STEP 10
30   CIRCLE 1,160,100,i
40   NEXT
50   PRINT"(CLR)(3xCRSR DOWN)GRAPHIC"
60   PRINT"(3xCRSR DOWN) and text"
70   PRINT"(3xCRSR DOWN)are mixed"
100  FOR I=48 TO 248: POKE 2612,I:NEXT
110  FOR I=248 to 48 STEP -1:POKE 2612,I
     :NEXT
120  GOTO 100
```

### SUMMARY OF RASTER INTERRUPTS

Memory location 2612 gives the current raster line. These bytes can be manipulated with POKE.

# Chapter 7

# Sprites

# Sprites

In this chapter we are going to assume that you have already read both chapters on sprites in the Commodore 128 System Guide, and that you understand how they operate. The following text has techniques for the C-64 mode sprites only, but these same techniques are made extremely easy with the sprite commands in BASIC 7.0.

## 7.1 Multi-color-sprites on the C-64

To turn on the multi-color mode, put the sprite number in the correct bits in VIC register 28 (53276). For example, to define sprite 6 in the multi-color mode use the following commands:

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

use these commands to set the byte back to zero:

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

Remember, in the multi-color mode 2 bits are needed to set a pixel. This leaves us with a 12x21 matrix. You already know how this mode operates. All you need to learn are the bit combinations needed to get certain colors.

If both bits are set to 0 the pixel is the same color as the background color. If both bits are set to 1, the VIC gets the color from the multi-color register in locations 37 and 38 (53285 and 53286). Which color is used is determined by bit 2. If this bit is set to 0 then register 37 is used, otherwise the VIC uses register 38. If the combination 1 0 is used the color comes from the normal sprite color register. In multi-color mode each sprite cannot be a different color as they can in other modes. In multi-color mode the same registers are used for all sprites and these range from 0 to 7. Multi-color sprites are just like normal sprites, only the conversion from bits to pixels is different.

**Warning**: Turn the sprites off when accessing the floppy drive (`POKE 53269,0`).

## SUMMARY OF MULTI-COLOR SPRITES

To turn the mode on: turn the correct bits on in register 28 (53276).

The colors come from register 37 for bits 01 or from register 38 for 11, the normal color register for sprites is used if the combination is 10.

## 7.2 Sprite Collisions

The VIC checks its registers every time a sprite comes "in contact" with another sprite or non-background color pixel. A collision of two or more sprites sets register 30 (53278). The numbers of the affected sprites are found through this register using:

```
PRINT PEEK(53278) AND 2^n
```

If sprite n was involved in the collision the result of this command will be zero. In BASIC 7.0 there is a command for this, BUMP. A collision is indicated if two pixels actually touch. The bits remain set and you can see them using the PEEK command. It may happen that a collision is announced even if the sprites are not actually touching. We recommend that after each look at the register, you set the bits back to 0, since the register does not reset itself.

The control of the background sprite collision follows in the same manner. The bit of the corresponding sprite in the register is set to one if the bit map or the character generator places a sprite over a pixel. In other words, every collision of a sprite with another character is registered. This is recorded in register 31 (53279). This register must also be reset to zero after a collision as it does not reset itself. Fortunately, in C-128 mode you can use the COLLISION command to check this register for you. To illustrate the programming of sprite collision control, a small game program is listed. It demonstrates a very simple auto race. To aim the car use the key Z (=left) and / (=right). If the car touches the borders or another car, a sprite collision is caught in either register 30 or 31, and the car crashes. You can omit the REM statements to make the program run a little faster. These are only for easy referral.

### SUMMARY OF COLLISIONS

Collisions of sprites with other sprites or background characters are indicated by setting the specific bits in one of two registers. These are registers 30 (53278) and 31 (53271) of the VIC. The bits remain set until the user accesses either register.

```
1 rem ++++++++++++++++++++++++++++++++++++
2 rem          c128 car race 7.2
3 rem ++++++++++++++++++++++++++++++++++++
10 scnclr0:color0,1:color 4,1:rem screen init
ialization
20 for i=3584 to 3646:read a:poke i,a:next:re
m auto sprite input
30 for i=3648 to 3710:read a:poke i,a:next:re
m crash sprite input
40 poke 2040,56:poke 2041,56:rem sprite point
er & counter
45 sprite 1,1,2,0,1,1,0:sprite 2,1,3,0,1,1,0:
rem activate sprites
50 for i=0 to 24:poke 1034+i*40,160:poke 5530
6+i*40,1
60 poke 1055+i*40,160:poke 55327+i*40,1:next
i:rem road output
70 movspr 1,168,170:x=168:rem put car at star
t
80 movspr 2,168,0:hx=168:hy=0:rem obstacle st
art
90 collision 1,200:collision 2,200:rem collis
ion control turned on
100 a= peek(212):rem poll keyboard
110 if a=12 then x=x-1:rem z key
120 if a=55 then x=x+1:rem / key
130 movspr 1,x,170:rem move the car
150 hy=hy+2:if hy>240 then hy=30:rem move fro
m bottom
160 hx=hx+ int(rnd(ti)*5)-2:if hx<120 then hx
=120:rem road turns left
170 if hx>216 then hx=216:rem right turn
180 movspr 2,hx,hy:goto100:rem obstacle move
200 poke 2040,57:sleep 3:run:rem crash
1000 rem auto sprite data
1100 data 0,0,0
1101 data 0,126,0
1102 data 0,126,0
1103 data 0,255,0
1104 data 12,255,48
1105 data 15,255,240
```

```
1106 data 12,255,48
1107 data 0,255,0
1108 data 0,255,0
1109 data 1,231,128
1110 data 1,195,128
1111 data 1,195,128
1112 data 1,195,128
1113 data 3,195,192
1114 data 3,195,192
1115 data 115,255,206
1116 data 115,255,206
1117 data 127,255,254
1118 data 115,255,206
1119 data 115,255,206
1120 data 0,0,0
2000 rem crash sprite data
2100 data 123,20,0
2101 data 0,24,0
2102 data 30,44,77
2103 data 21,126,3
2104 data 240,125,48
2105 data 15,205,240
2106 data 22,155,41
2107 data 1,205,156
2108 data 0,155,0
2109 data 1,201,108
2110 data 1,105,108
2111 data 1,095,028
2112 data 1,155,158
2113 data 23,115,192
2114 data 32,242,132
2115 data 35,239,216
2116 data 1,095,028
2117 data 32,242,132
2118 data 68,155,0
2119 data 27,125,48
2120 data 0,0,0
```

## 7.3 Priorities and movement regions

Sprites are usually superimposed over an existing screen picture. However, you can also place a character in front of a sprite on the screen. For example, you can have a sprite which represents an airplane pass behind a house. The VIC has a register to control the sprite "transparency".

Register 27 has the address 53275 (V+27). The bits in this register determine the priority of the sprite being drawn. Each sprite has a corresponding bit in the register. The sprite corresponding to the lowest bit will have the topmost priority. If you want to change the priority of a sprite use this command:

```
POKE 53275, priority number
```

Remember, any new sprite that you draw will automatically have a priority above the others on your screen.

The coordinates of sprites and graphics don't correspond directly. The movement region of the sprites is better defined as the output of the sprites to the monitor. You can put a sprite in the upper-left corner of the screen with the coordinates 24 and 50. These coordinates are the correction factor that you have to add to the graphic coordinates in order to correctly position a sprite. The middle of the screen is calculated with the values 160+24 and 100+50.

### SUMMARY OF PRIORITIES AND MOVEMENT REGIONS

Sprite priorities (before / behind) are controlled by the appropriate bit in the VIC register 27 (53275). By setting the bits the sprite then being printed is placed behind the current character.

The correction factor for sprites to graphics coordinates:

24 (X coordinate) and 50 (Y-coordinate)

## 7.4 Programming with sprites

Many game programs use animation to create movement on the screen. For example, a small sprite man has to run through a maze to get to (or get away from...) his sprite wife. With a second look you can see that there are only two or three positions for the arms and legs. The principle of animation is easy to understand. You have two blocks of sprites which are switched back and forth and moved along, giving the appearance of movement. In one block the man has his legs and arms in a somewhat closed position, and another with the legs and arms in a stretched out position, as if striding. If these two pictures are switched the figure appears to be running. You can see this for yourself with the SPRITE command. Unfortunately, the disadvantage to this method is that you can only have 8 different pictures. It is better to change the sprite pointer directly (2040-2047), then the VIC will be able to switch to another memory region easily to get a new picture. These pictures can be put into different section of the first 16K of the BANK 0, or else in the user region 1300-17FF. All you have to do is divide the region into blocks of 64 and put the value in the sprite pointer by the POKE command.

Notice that the sprite pointer for the graphic region is in addresses 8184 through 8191.

The natural assumption of all this is that there is enough room for the different pictures. To make sure there's room, move the BASIC start address to a higher address in memory.

The following program rolls a ball across the screen. It uses four different sprites.

```
1 rem ball 7.4
5 print chr$(147)
10 for i=0 to 62:read a:poke dec("0e00")+i,a:
next
20 for i=0 to 62:read a:poke dec("0e40")+i,a:
next
30 for i=0 to 62:read a:poke dec("0e80")+i,a:
next
40 for i=0 to 62:read a:poke dec("0ec0")+i,a:
next
50 sprite 1,1:movspr 1,160,100:movspr 1,90#5
60 poke 2040,56:for i=0 to 35:next
70 poke 2040,57:for i=0 to 35:next
80 poke 2040,58:for i=0 to 35:next
90 poke 2040,59:for i=0 to 35:next
100 goto 60
200 data 0,120,0,3,159,0,12,31
201 data 192,16,31,224,32,31,240,32
202 data 31,240,64,31,248,64,31,248
203 data 64,31,248,128,31,252,255,255
204 data 252,255,224,4,127,224,8,127
205 data 224,8,127,224,8,63,224,16
206 data 63,224,16,31,224,32,15,224
207 data 192,3,231,0,0,120,0
210 data 0,120,0,3,135,0,12,0
211 data 192,16,0,32,56,0,112,60
212 data 0,240,126,1,248,127,3,248
213 data 127,135,248,255,207,252,255,255
214 data 252,255,207,252,127,135,248,127
215 data 3,248,126,1,248,60,0,240
216 data 56,0,112,16,0,32,12,0
217 data 192,3,135,0,0,120,0
220 data 0,120,0,3,231,0,15,224
221 data 192,31,224,32,63,224,16,63
222 data 224,16,127,224,8,127,224,8
223 data 127,224,8,255,224,4,255,255
224 data 252,128,31,252,64,31,248,64
225 data 31,248,64,31,248,32,31,240
226 data 32,31,240,16,31,224,12,31
227 data 192,3,159,0,0,120,0
```

```
230 data 0,120,0,3,255,0,15,255
231 data 192,31,255,224,47,255,208,39
232 data 255,144,67,255,8,65,254,8
233 data 64,252,8,128,120,4,128,48
234 data 4,128,120,4,64,252,8,65
235 data 254,8,67,255,8,39,255,144
236 data 47,255,208,31,255,224,15,255
237 data 192,3,255,0,0,120,0
```

It is often useful to save sprite blocks on diskette or cassette. A program for saving them was given in section 4.1 for the C-128 mode for the BSAVE command.

One interesting misuse of high resolution sprites is to use them as small screen graphics inside of the character mode of the C-64. Let's say you want to display your favorite graph. However, you also want to add some comments on the graph. In the C-128 mode there is the graphic mode 2. One possibile way is to save the pixel matrices for the letters in a region of memory. This method, however, is very time-consuming. The other method uses sprites. We will need four, arranged in a square about the needed position on the screen. We must calculate for every pixel which bits to set inside of the sprite matrix. This is what the routine listed below will do.

```
1 rem c64 plotter 7.4
10 for i=704 to 767:poke i,0:next
20 for i=832 to 1023:poke i,0:next
30 poke 2040,11:poke2041,13:poke 2042,14:poke
 2043,15
40 v=53248:pokev,100:poke v+1,100:poke v+2,14
8:poke v+3,100
50 poke v+4,100:poke v+5,142:poke v+6,148:pok
e v+7,142
60 for i= 39 to 42:poke v+i,1:next i:poke v+2
1,15:poke v+23,15:poke v+29,15
90 print chr$(147)
100 input "x-coord ";x:input "y-coord ";y
110 gosub 62000:goto100
62000 y=41-y:if y<0 or y>41 then return
62010 if x<0 or x>47 then return
62020 bx=int (x/24):by=int(y/21): if bx=0 and
 by=0 then ba=704:goto 62040
62030 ba=768+bx*64+by*128
```

```
62040 bx=x-24*bx:by=y-21*by
62050 x1=int(bx/8):x2=7-(bx and 7):x3=by*3
62060 ad=ba+x3+x1
62070 if l=1 then poke ad,peek(ad) and (255-2
^x2):return
62080 poke ad,peek(ad) or (2^x2):return
```

This routine is called with GOSUB 62000. With high resolution graphics the coordinates are in X and Y, and the mode (set or clear) is in L. This routine uses the sprites 0-3 and the blocks 11,13,14,15. In the aforementioned version the sprites are enlarged in both directions. If you want the normal size to be displayed you have to correct the position (see lines 30 and 40).

Inside of the 4 sprites you can set 48x42 pixels. This routine will not handle multi-colored sprites—all the pixels must have the same color. The color is set in line 50.

We are now finished with this section on sprites. We hope that you will continue to experiment with sprites. There are many different possibilities for sprites in all sorts of applications.

# Chapter 8

# The 80 column screen

## The 80 column screen

One special feature of the C-128 allows you to display more than a single screen of text, graphics or data all at the same time.

This is possible since the computer has two different video chips installed:

1. The VIC, which is identical to the one in the C-64.

2. The VDC, or the Video Device Controller, a new part of the computer, whose purpose in the C-128 is to relay the RGB output.

We will be discussing the VDC in this section.

The purpose of the VDC is to take care of the RGB output. RGB stands for Red, Green, and Blue. These are the three colors that a normal color monitor is composed of. The colors are not composed from a single signal, as they are in a TV transmitter. There is a separate signal for each of the three colors.

At first glance, this may look quite complicated. The big advantage to RGB, though, is that you get the color you ask for. This is rather difficult to do with only one signal. What makes the VDC so likeable is not only the fact that it produces an RGB signal, but also that it uses its own 16K portion of RAM, not main memory. The information about the screen and character portion is located in this area.

It is possible to save the contents of the screen in the main memory. It is therefore possible to write a routine that uses the same information for different types of monitor screens.

The VDC, as with the VIC, controls register 37. This register is important throughout the rest of this chapter. A complete memory map is available for the C-128 in Appendix E.

## 8.1 The RAM region of the VDC

The RAM region of the VDC is divided into the three following areas:

1. Video RAM
   This is where the characters that appear on the monitor screen are saved. The addresses that encompass the VDC region are from $0000 through $3FFF. The region for the characters goes from $0000 through $07CF (We will see why this is important in a little bit).

2. Attribute RAM
   This is the region from address $0800 through $0FCF. This region directs the monitor. These are the 2000 bytes that are used to direct the output to the video RAM of the specified character. In order to accomplish this every byte in the video RAM has a corresponding byte in the attribute RAM. The output of the character depends on the bits set in the corresponding attribute byte.

   Bit 0: Lightness. If this bit is set the character is displayed in a color of a slightly lighter shade.

   Bit 1: Blue. The specified character is partly blue.

   Bit 2: Green. The specified character is partly green.

   Bit 3: Red. The specified character is partly red.

   Bit 4: Blinking. The specified character will be blinking when displayed.

   Bit 5: Underlined. A character of the video RAM will appear underlined on the screen.

   Bit 6: Reverse. This bit allows two possibilities; reverse, where the character appears reversed on the monitor. The second possibility is that the reverse character that also exists in the character ROM is displayed.

Bit 7: Alternative Character Set. This is different than the C-64 mode since the 80 character screen is different. You can change the character set with the <SHIFT> and the <COMMODORE> keys. The reason for this is that both character sets are present in the VDC and they can be switched simply by setting the 7 bit.

3. Character Generator
The character generator can be found in memory locations $2000 through $3FFF. This region takes up a full 8K. The reason it takes up so much space is:

a) The character generator contains both character sets.

b) The definition of each character requires 16 bytes. One word has the form described, the other is a null word.

The rest of the RAM region for the VDC is reserved for high-resolution 80 column graphics. You are probably asking yourself why you haven't come across this in any other C-128 references. We'll satisfy your curiosity in the next section. This section painted a rather crude picture of the VDC—we'll take care of the details later.

## 8.2 The VDC's registers

Now that we have praised the VDC, we have to look at the other side of the coin. The VDC's registers are quite complicated. This is because the VDC only has two registers that you can directly manipulate. These two registers are enough to cover the RAM region.

The two VDC registers have the addresses $D600 and $D601.

The only way to use the registers is as follows:

1. The number of the working register is in location $D600.

2. The value in the register mentioned in location $D600 is in location $D601.

For example:

```
POKE DEC("D600"),30:POKE DEC("D601"),10
```

Register 30 contains the value 10. If you want to see the value in a given register, use `PEEK(DEC"D601").`

It is somewhat more complicated to manipulate the bytes in the VDC RAM.

The first step is to specify the address that is to be changed. This happens with the help of registers 18 and 19.

Now the value in register 31 must be written. To complete the procedure we have to `POKE` register 30. This is to determine how often the character in registers 18 and 19 are to be written.

Here is an example:

1. `POKE`ing the address

```
POKE DEC("D600"),18:POKE DEC("D601"),12
POKE DEC("D600"),19:POKE DEC("D601"),4F
```

2. The value is written:

```
POKE DEC("D600"),31:POKE DEC("D601"),127
```

3. Finally:

```
POKE DEC("D600"),30:POKE DEC("D601"),1
```

As you can see this is quite complicated, but what's worse, it doesn't always work. The precise alteration of the contents of memory can only be accomplished with machine language.

The program listed below makes it easy to change every byte of the VDC:

```
10  fori=4864to4864+33
20  readx:pokei,x:next
40  data162,18,169,0,32,22,19,232,169,
      0,3,2,22,19,162,31,169,0,32,22,
      19,162,18,142,0,214,44,0,214,16,
      251,141,1,214,96
50  lo=dec("1309"):hi=dec("1303"):
    we=dec("1310")
60  fori=0to1999:rem Address
70  pokelo,i and 255
80  poke hi, i/256
90  poke we,127: rem character
100 sys dec("1300")
110 next
120 getkey a$
```

With the help of this program you can influence the contents of the VDC RAM. This makes it easy to POKE into the screen, the attribute, or into the character generator. For example: if you want to change the character definition in the character generator, you get the address of the character with the simple expression:

Address = 8192 + 16* (*character code*)

This address and the following seven locations contain the character.

# Chapter 9

# High resolution graphics with the 80-column screen

## High resolution graphics with the 80 column screen

As we mentioned in the previous chapter, the VDC can also be used for high resolution graphics. The graphics for the VDC are twice as sharp as those for the VIC, there are 640*200 pixels that are all monochrome. Unfortunately, these graphics are not supported by the new BASIC 7.0

```
1 rem pixel set 9.1
5 gosub 2000
10 a=dec("d600"):d=a+1:rem starting address
20 poke a,25:poked,128:rem turn on graphics
30 print chr$(147);
40 for i=0 to 30
50 print "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@";
60 next i
70 poke a,24: c=peek(d)
80 poke a,24: poke d,cor128
90 poke a,32: poke d,0
100 poke a,33:poke d,0
110 for i= 1 to 63
120 adr=i*256:hi=adr/256:lo=adr and 255
130 poke a,18:poke d,hi
140 poke a,19:poke d,lo
150 poke a,30:poke d,0
160 next i
200 :
210 :
220 rem coordinates are calculated (gosub 100
0)
230 :
240 :
250 gosub 1000:end
1000 lo=dec("1309"):hi=dec("1303"):we=dec("13
10")
1010 ad=int(x/8)+y*80
1020 x1=ad/256
1030 x2=ad and 255
1040 poke a,18:poke d,x1
1050 poke a,19:poke d,x2
1060 poke a,31:c=peek(d)
```

```
1070 poke lo,x2
1080 poke hi,x1
1090 poke we,c or 2^(7-(x and 7))
1100 sys dec("1300")
1110 return
2000 for i= 4864 to 4864+33
2010 readx: poke 1,x:next
2020 data 162,18,169,0,32,22,19,232,169,0,32,
22,19,162,31,169,0,32,22,19,162,18,142,0,214,
44,0,214,16,251,141,1,214,96
2030 lo=dec("1309"):hi=dec("1303"):we = dec("
1310")
2040 for i= 0 to 1999:rem address
2050 poke lo,i and 255
2060 poke hi,i/256
2070 poke po,127:rem character
2080 sys dec("1300")
2090 next
2112 return
```

```
5 rem block copy 9.1
10 a=dec("d600"):d=a+1:rem starting address
20 poke a,25:poked,128:rem turn on graphics
30 print chr$(147);
40 for i=0 to 03
50 print "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@";
60 next i
70 poke a,24: c=peek(d)
80 poke a,24: poke d,c or 128
90 poke a,32: poke d,0
100 poke a,33:poke d,0
110 for i= 1 to 63
120 adr=i*256:hi=adr/256:lo=adr and 255
130 poke a,18:poke d,hi
140 poke a,19:poke d,lo
150 poke a,30:poke d,0
160 next i
```

## 9.1 The bit map

You have probably already figured out the location in memory for these graphics. It is in the 16K of RAM that is under the direction of the VDC. This means it is possible to put both graphics and normal text on to the same screen.

The graphics regions of the video RAM, the attribute RAM, and the character generator all have access to the bit map. The bit map is responsible for the display on the monitor screen. Every bit that is set to 1 in this region turns on a pixel on the screen.

### 9.1.1. The order of the bytes in the bit map

The order of the bytes for these graphics is very user friendly. This makes the graphics very easy to use.

Here's how the bytes are ordered. First we'll look at the upper left corner of the screen with a "magnifying glass":

Addresses:

Line 0: 0000 0001 0002 0003 0004 0005 0006 ... through 0079
Line 1: 0080 0081 0082 0083 0084 0085 0086 ... through 0159
Line 2: 0160 0161 0162 0163 ... etc.

Here you can see that there is a byte-for-byte correspondence with the individual pixels until the line is full, and the next line begins.

From this order you can calculate the address that has the pixel you want to change, as follows:

```
Address = INT(X/8) + Y * 80
```

The pixel (the bit) is just as easy to calculate:

```
Value = 2^(7-(X AND 7))
```

## 9.1.2 Setting pixels

Before we start with the theory you will probably want to know how to enable graphics with the VDC.

The main register is register 25 of the VDC. You can enable the graphics by setting bit 7.

Use the following POKEs to enable the graphics:

```
POKE DEC("D600"),25:A=PEEK("D601")
```

and

```
POKE DEC("D600"),25:
POKE DEC("D601"), A OR 128
```

To set the text mode again POKE the original value of D601 back into D601.

When you first perform this operation there is a total mess on the screen. The pixel mix-up on the screen is simply the contents of the screen region of memory, for example, the attribute or character generator.

We have listed a program on page 109 that should help you make clear up this video mess. Here is a short explanation of the program.

First of all, we print "@"'s all over the screen. This is done with the help of a block copy routine. This fills the entire graphic region of the VDC and initializes the region. We could also do this with the 80-column POKE routine poking the "@" sign location by location, but the block copy is much quicker.

On page 110 we listed a program that allows you to set pixels so that you can use the graphics. This program uses the 80 character POKE routine from the last chapter. This routine must be installed at the start of the program.

When you want to finish your graphics, you have to reset bit 7 of the register back to a zero. This will return control back to the ASCII mode that you turned off earlier.

You have to protect the character generator region of memory before using graphics, or else you could destroy the characters in the RAM region of the VDC with your graphics.

# Chapter 10

# Sound generation

# Sound generation

What the VIC is to video display, the SID (Sound Interface Device) is to sound generation. There is a register for every possible parameter. Unfortunately, these are not described in the Commodore 128 System Guide. Since discussing every facet of the SID is beyond the scope of this book, we only cover at the basics of programming sound effects.

## 10.1. The operation of the SID

In this section we briefly explain what happens in the computer when we want to generate a sound.

If a specific start bit is set to 1, the SID produces a sound of a specific frequency or oscillation. This happens with a sort of "electronic potter's wheel", the Wave Form Generator. Through this, the sound is programmed with a specific wave form (triangular, square, sawtooth, round) and its characteristic sound patterns.

The SID forms what is called the *envelope curve*. This determines what volume the sound will have in the different phases. The envelope curve is composed of four different parameters. *Attack* means how fast the register reaches the highest volume of the envelope. After the maximum volume, the sound will stay at this level for a certain length of time. This is the *Decay* parameter. Then sound then will reduce to the value in the *Sustain* parameter. The volume remains at this level until the startbit is returned to a zero. The *Release* parameter describes how fast the tone should diminish. This parameter allows you to produce reverberations or echoes.

Other possibilities (that we won't discuss here) include: ring modulation, where the sound of each of two tones is dependent on the other; and filters, where you can filter out certain frequency ranges.

## 10.2 Programming sound in the C-64 mode

In this section we will discuss programming sounds and sound generation in the C-64 mode.

We know you want to see what a sound program looks like, but there are some things we have to take care of.

The first is volume. This is located in the low 4 bits of register 24 (54296). If you want to see the contents of this register or the others (0-24) using the PEEK command, you will be disappointed. These registers are made so that they can be written to but not viewed. A PEEK to these registers yields garbage.

The reverse situation exists in registers 25-28. They can only be read, not written to. Using a POKE will have no effect.

The 4 high bits in the volume register are normally set to 0, but we can change these values with a POKE command. Using `POKE 54296,0` the volume is reset back to its normal value, with `POKE 54296,15` sets the maximum volume. The volume can only be set to multiples of 3.

Next comes the frequency. This determines how high or low the tone of the sound gets. You can choose from 65536 different frequencies. For programming melodies, the appendix of notes in the System Guide is helpful. How you put the value of the frequency into the high and low bytes was discussed earlier, in the chapter about pointers. These numbers are in registers 0 and 1 (for voice 1), 7 and 8 (for voice 2), and 14 and 15 (for voice 3).

Now we will discuss the envelope curve. The attack, sustain, and decay of a tone is determined in register 5 (12 or 19 for the other voices). The attack is in the high bits, the decay in the lower bits. The values for the sustain and reverberation are in registers 6, 13, or 20; the sustain is in the upper bits. If this value is set to zero the voice is silent. The volume of the tone is in ratio with the maximum in register 24.

If you want to use square waves, the SID is equipped to play them. It can hold values between 0 and 4095 and store these values in the register pairs 2/3, 9/10 or 16/17. Only the low bits of the high byte are used. Any numbers higher than 15 in these registers don't make any difference.

So far so good. Now we are going to deviate from the Commodore 128 System Guide. The waveforms are set in register 4 (11 or 18, depending on the voice). Similar to the VIC memory locations, each bit has its own meaning. Bit 0 is the start-stop bit for the sound generator. When this bit is set to one, the sound with the correct voice and envelope wave is started. When it returns to zero, the envelope curve and sound will die out in a specified amount of time. Notice, that the SID cannot produce another tone until the first has completely died out. Therefore, when programming melodies, you have to use a very short decay value, so that you can produce tones in rapid succession.

Bits 1 and 2 in register 4 are for internal control. Bit 3 turns out to be quite useful to us. If two or more waveforms are present at the same time the SID shuts off all sound, and becomes quite confused. You can return the SID to a working state by setting bit 3 back to a 1 and by zeroing out the waveforms. To do this use `POKE 54276,8` to reinitialize the SID.

The waveform is determined by bits 4 through 7. To choose a waveform set the corresponding bit to 1. Bit 4 is for triangular, bit 5 for sawtooth, and bit 6 is for a square-wave. Bit 7 is for rounded waveforms. These waveforms can also be intermixed.

To start a sound, the waveform and the start bit must be set using a `POKE`. For that you will need the C-128 System Guide containing the codes for different sounds (17, 33, 65, 129). Turning the sound off is not as simple as setting register 4 (11 or 18 for the other voices) to zero. That would be equivalent to turning off your engine in the middle of the freeway. The SID cannot suddenly start a waveform, and so it cannot suddenly stop a waveform. This would result in snapping off the sound. If you just reset bit 0 the sound will die off nicely.

## SUMMARY OF TONE GENERATION

Volume is in register 24: Range 0-1

Attack: highbits in the halfbyte in Registers 5/12/19.

Decay: lowbits in the halfbyte in registers 5/12/19

Sustain: highbits in the halfbyte in registers 6/13/20.

Release: lowbits in the halfbyte in registers 6/13/20.

Waveforms: Register 4/11/18
Bit 4: Triangular
5: Sawtooth
6: Square
7: Rounded

Other: Bit 3: Initialization
1: Start-Stop-bit

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

## 10.3 BASIC 7.0 Music commands

### 10.3.1 PLAY and ENVELOPE

The C-128's PLAY command defines and plays musical notes and elements. It lets you select voice, octave, envelope, volume and notes. The syntax for the PLAY command is PLAY "*voice,octave,envelope,volume,notes*". The PLAY command has 10 preset defaults for musical instruments to make it easy to use. Here is an example:

```
PLAY V1O4T0U5X0CDEFGAB
```

So far everything sounds simple. You give a string of notes from a melody with the PLAY commands, and the C-128 plays the melody out.

However, there's a hitch. The accordian preset does not sound much like an accordian—and the drum preset sounds something like small waves at the seashore. We will give you a little help so you can alter the sounds so that they sound right (although this strictly from our personal taste). Luckily, the original waveforms detailed in the System Guide can be altered with the ENVELOPE command.

The sound that string instruments make decays quickly.

The original envelope curve for the piano is nearly perfect, but the sound can be made better. Just change the pulse width to 600 (ENVELOPE 0,,,,,,600).

A cymbal can also be imitated by setting that same value to 400. But the cymbal can generate sound for quite a while, so you should set the release time to 9 (ENVELOPE 6,,,,9,,400). This sounds especially good in melodies.

The guitar also requires corrections. The waveform for this instrument is best suited with the square wave. The length of the attack has to be changed, because there is no perfect fit. The best we can do is to use 3500. The release time also has to be changed (ENVELOPE 5,,,3,3,2,3500).

The flute sounds good, but the tone should decay much more quickly, so shorten it to 7 (`ENVELOPE 4,7`).

Changes have to be made for the accordian also. The sustain is too low, so the correction for this is `ENVELOPE 1,,,12`.

Now we come back to the drum sounds. Striking a drum creates a sound as quickly as it is hit, therefore, change the envelope with `ENVELOPE 3,,8,0`

The last critical point is the trumpet envelope curve. For the C-128 to sound anything like a trumpet, we have to change its envelope, especially the attack and decay times. Use the following command: `ENVELOPE 8,3,13`.

The C-128 is not limited to only the orchestral instruments. You can program vitually any sound you can imagine. To make the sound of a bell:

```
ENVELOPE 1,0,10,1,10,2,2048
```

And to make the sound of waves at the beach:

```
ENVELOPE 1,11,11,0,11,3
```

Or if we want to make a futuristic sound:

```
ENVELOPE 1,0,7,0,0,2,1800
```

Experiment and you will find that you can create any sound you desire.

## SUMMARY OF THE ENVELOPE CURVE

| | |
|---|---|
| PIANO: | `ENVELOPE 0,,,,,,600` |
| CYMBAL: | `ENVELOPE 6,,,,9,,400` |
| GUITAR: | `ENVELOPE 5,,,3,3,2,3500` |
| FLUTE: | `ENVELOPE 4,7` |
| ACCORDIAN: | `ENVELOPE 1,,,12` |
| DRUM: | `ENVELOPE 3,,8,0` |
| TRUMPET: | `ENVELOPE 8,3,13` |
| BELL: | `ENVELOPE 1,0,10,1,10,2,2048` |
| SEA WAVES: | `ENVELOPE 1,11,11,0,11,3` |
| FUTURE: | `ENVELOPE 1,0,7,0,0,2,1800` |

## 10.3.2. SOUND

The SOUND command is closely related to PLAY, and allows different acoustical possibilities. The sound of a motor is quite easy to duplicate. Just use a square wave with a very small frequency and an oscillating impulse. For example:

```
SOUND 1,1000,100,0,,0,2,200
SOUND 1,400,100,0,,0,2,3500
```

Of course, you can change the frequency and the length of the sound to emulate anything from a Diesel engine to a moped.

Footsteps are also no problem. The command:

```
SOUND 1,10000,1,0,,0,3
```

makes a very quick sound like that of footsteps.

A police siren can be simulated with the following:

```
SOUND 1,20000,1000,2,5000,1000,2,2048
```

You can change the length of the attack and the decay times with the sixth parameter. Moreover, the falsely named "max frequency" is parameter 5. To get a variable tone this parameter must be less than the starting frequency.

### SUMMARY OF SOUND

```
MOTOR:  SOUND 1,1000,100,0,,0,2,200
        SOUND 1,400,100,0,,0,2,3500
STEP:   SOUND 1,100000,1,0,,0,3
SIREN:  SOUND 1,20000,1000,2,5000,1000,2,2048
```

# Chapter 11

# The keyboard

# The keyboard

The most noticeable feature of the C-128 is its new keyboard. It is probably the best keyboard for a computer in its price range. Not only does it provide comfort when typing, but it also has all kinds of features for programming tricks.

## 11.1 Construction and operation of the keyboard

Normally, the keyboard is accessed with `INPUT` and `GET`. A glance at the Commodore 128 System Guide will show that the keyboard is also device number 0 and can be accessed with the normal `OPEN` command. Now the keyboard can be used just like a floppy disk or cassette. Unlike the normal `INPUT` command this method will not yield a question mark.

The interface to the keyboard is CIA 1. However, the VIC must also come in to play. The CIA 1 executes polling of the C-128 keyboard from the two parallel ports (closely related to the user-port). It reads 64 keys which are electrically divided up into 8 lines and 8 locations. The remaining 3 lines of the keyboard (the cursor keys and so forth) are connected to the VIC and are set in register 47. The two ports are programmed for output. This is where the output from the polling appears. When a key is pushed port B is switched to input. The interrupt routine gets the next key from the input buffer. This is then sent to the ASCII decoding table and this new value is replaced where the original value was.

When the interpreter is running in the direct mode, it gets its information from the keyboard, converts it to ASCII, and then processes the ASCII to produce a BASIC code (for example `RETURN` = 13). When a program is running, the keyboard buffer is unaffected until a `GET` or `INPUT` is found. With a `GET` command the interpreter simply gets one character from the keyboard buffer and stores it in the command variable. An `INPUT` command functions similarly, except that the characters come from the keyboard and the interpreter doesn't continue until the <RETURN> key is pressed.

The keyboard matrix has its own peculiarities. The <RESTORE>, <CAPS LOCK> and <40/80 DISPLAY> keys have no entry in the matrix. The <RESTORE> key works directly with the processor (similar to the <RESET> key) and uses its own special interrupt. This routine checks to see if the <RUN/STOP> key has been pressed. If this is the case, a sort of mini-reset occurs, and everything runs as before. The two other keys work on the MMU and change the way the system works.

Another peculiarity lies with the <SHIFT> keys. The computer can detect the difference between the left and right <SHIFT> keys, since both have their own locations in memory. The <SHIFT-LOCK> key, however, sends the same code as the left <SHIFT> key.

## 11.2 Pressing two keys at once

We will now discuss what we said in the last chapter in practical terms. For many programs we want to be able to poll different keys at the same time. An example would be: two spaceships that can be controlled independently by different keys. To do this we are going to look at the keyboard matrix.

The three memory locations the keyboard is tied to are 56320, 56321, and 53295. Normally all the bits in these registers are set to one. If a specific column is to be set, the effected bits in 56320 through 53295 must be set to 0. Similarly, the message is returned from the keyboard. If a key is pressed the appropriate bit is set to zero in location 5621.

We will finally learn what the interrupt routines do. With a POKE command we can look at a specific column and test the individual keys.

We can manually accomplish what the interrupt routine does:

```
POKE 4864,120: POKE 4865,96:SYS 4864
POKE 56320,column code:
POKE 53295,column code
IF (PEEK(56321)AND(28^bit#)=0)
    THEN PRINT "KEY PRESSED"
POKE 4864,88: SYS 4864
```

We can test a key with this program. If you want to observe more keys, all you need are more IF-THEN constructions and more POKE commands to check the appropriate columns. The column code can be determined with the formula:

```
CODE = 255 - 2^column number
```

The column number determines the position of the bit inside the memory locations 56320 through 53295 for the appropriate column. The IF-THEN construction in the above listing tests for a 0 in the appropriate location.

The line and column number can be looked up in the table matrix.

Another possibility for checking two keys is to use location 211. This is where the actual <SHIFT> pattern appears. The pattern in this register tells whether or not one of the five special keys have been pushed (<SHIFT>, <C=>, <CONTROL>, <ALT>, <CAPS LOCK>). If it is the <SHIFT>

key, bit 0 is set to 1. For the <C=> key it is bit 2, for <CONTROL> bit 3, <ALT> bit 4, and finally <CAPS LOCK> bit 5. The setting of one bit is completely independent of the others. It is possible for all five keys to be set at the same time. If we use this register, however, we must turn off the interrupt routine. Use the following command:

```
IF (PEEK(211)AND2^bit#) THEN PRINT "KEY PRESSED"
```

The computer can use BASIC to determine whether or not a key has been pressed using this routine.

## SUMMARY OF CHECKING FOR MORE THAN ONE KEY BEING PRESSED

There are two possibilities:

a. `PEEK(211)` gives the shift pattern.With this we can check 5 different keys independently of each other.

b. You can check different columns with `POKE 56320,X1` and `POKE 53295,X2`. To determine what key in the column was pressed check location 56321. The table we have been talking about is listed on the following page.

The keyboard matrix:

| Line | | Column | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 56320 | 53295 | 56321 | | | | | | | |
| (x1) | (x2) | 254 | 253 | 251 | 247 | 239 | 223 | 191 | 127 |
| 254 | 255 | del | cr | → | F7 | F1 | F3 | F3 | ↓ |
| 253 | 255 | 3 | W | A | 4 | Z | S | E | sh.-L |
| 251 | 255 | 5 | R | D | 6 | C | F | T | X |
| 247 | 255 | 7 | Y | G | 8 | B | H | U | V |
| 239 | 255 | 9 | I | J | 0 | M | K | O | N |
| 223 | 255 | + | P | L | - | . | : | @ | , |
| 191 | 255 | £ | * | ; | home | sh-R | = | ↑ | / |
| 127 | 255 | 1 | ← | ctrl | 2 | space | C= | Q | stop |

Additional Keys:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 255 | 254 | HELP | 8 | 5 | TAB | 2 | 4 | 7 | 1 |
| 255 | 253 | ESC | + | - | LF | enter | 6 | 9 | 3 |
| 255 | 251 | ALT | 0 | . | ↓ | ↑ | ← | → | no sc. |

## 11.3 Disabling keys

For many applications it is necessary to disable individual keys, or even the entire keyboard of the C-128.

In order to disable the entire keyboard it is advisable to turn off the interrupt routine. This means the cursor will disappear and the computer will lock up. You can restore the computer with <RUN/STOP> <RESTORE>.

The same goes for the `POKE 2592,0`. The keyboard is turned off, yet the cursor remains intact. The <RUN/STOP> key also operates. It is interesting to see how this actually works. This memory location is the length of the keyboard buffer. What we are doing is setting it to 0 (it's normally set to 10). The operating system seems to think that the keyboard buffer is full and forgets the last key pressed. If you do this when you are in BASIC, `GET` and `INPUT` will not work.

If you only want to disable the <RUN/STOP> key, you can use the command `POKE 808,112`. Now the BASIC program will only stop if you press the combination of keys <RUN/STOP> <RESTORE>. The `BREAK` function is returned with `POKE 808,110`.

You can inhibit the mini-reset ( <RUN/STOP> <RESTORE>) with `POKE 792,98`. The <RUN/STOP> key still works. If you combine the previous two `POKE`'s there is no way to stop a BASIC program (outside of turning off the C-128). You can re-enable the <RUN/STOP> <RESTORE> combination with `POKE 792,64`.

## SUMMARY OF DISABLING THE KEYBOARD

Turning off the entire keyboard:

1. Turn off interrupts
2. `POKE 2592,0` (Buffer length 0)

<RUN/STOP> off: `POKE 808,112`

<RUN/STOP> on: `POKE 808,110`

<RESTORE> off: `POKE 792,98`

<RESTORE> on: `POKE 792,64`

## 11.4. The key-repeat function

You use this repeat function whenever you want to move your cursor around the screen. However, the other keys are also tied into this function. When you press a key, after a short period of time the computer continues to print a key over and over again until the you let go of it. It is possible to repeat immediately after pressing the key, rather than having to wait for a moment. Conversely, you can turn off the repeat function completely (for individual keys or every single one). You use the `POKE` statement once again to accomplish this. For every key to repeat, location 2594 must have the value 128 stored in it. This points to an interrupt routine that repeats the key. If you change this value to 0 (`POKE 2594,0`), each key will only be printed once for each time pressed. This works for all keys except for the following:

- The cursor keys
- The <Insert> and <Delete> keys
- The space bar

It is possible to take away the repeat function from all of the keys. This is done with `POKE 2594,64`. After this no key will have the repeat function available to it.

### SUMMARY OF THE REPEAT FUNCTION

| | |
|---|---|
| `POKE 2594,128:` | Repeat all keys |
| `POKE 2594,64:` | Turn off repeat function |
| `POKE 2594,0:` | Repeat only for cursor,insert, and space keys |

## 11.5 Polling the keyboard one more time

As we mentioned earlier the interrupt routine puts the keys into the keyboard buffer as ASCII codes. During this time, the value is kept in a way station, memory location 212. This is where the keyboard code is stored, it is a pointer into the decoding table. The code is saved in this register. You can look at this register with a `PEEK(212)`.

A key viewed with a `PEEK(212)` will not be placed in the keyboard buffer. You can use `GET` or `INPUT` to bring the value into a variable. You can then test the value in any way you wish. Outside of this the keyboard codes can only be stopped by turning off the interrupt routines.

You can clear the keyboard buffer with the simple command:

```
POKE 208,0
```

It is then possible to have the computer wait until a key is depressed. To do this:

```
POKE 208,0: WAIT 208,1
```

As you can see, there are many possibilities for polling the keyboard. Don't be afraid to try something new.

### SUMMARY OF POLLING THE KEYBOARD

`PEEK(212)` gets the keycode directly from the pressed key .

`POKE 208,0` clears the keyboard buffer.

`POKE 208,0: WAIT 208,` waits for a keypress.

## Table of Key Codes

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

## Additional Codes

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ESC | = 72 | ↓ | = 84 | 3 | = 79 | 9 | = 78 |
| TAB | = 67 | ← | = 85 | 4 | = 69 | + | = 73 |
| HELP | = 64 | → | = 86 | 5 | = 66 | - | = 74 |
| LF | = 75 | 0 | = 81 | 6 | = 77 | . | = 82 |
| NO SC. | = 87 | 1 | = 71 | 7 | = 70 | ENTER | = 76 |
| ↑ | = 83 | 2 | = 68 | 8 | = 65 | | |

# Chapter 12

# The User port

# The User port

The User port turns the C-128 into a very versatile instrument. Unfortunately, the handbook doesn't say a single word about how to program it or use it. We are going to provide some groundwork so you can program this port.

All information in this chapter is true for both C-64 and C-128 operating systems. In the C-128 mode all you have to do is make sure that `BANK 15` is turned on.

## 12.1 All about the building blocks

Just as the keyboard and joystick use the CIA, so does the User port. The CIA is the building block. This chip's job is to receive and send information to the peripherals, as well as keep in touch with the processor.

This building block is actually made up of three parts. The first is the input for the parallel ports. There is also the timer (which you have already used and need to know nothing else about ), as well as a serial port.

The following sections cover the operation of these three elements.

### 12.1.1 The serial port

We are going to start with the most simple part of the User port. As you may already know, a computer works in *parallel mode.* This means it works with 8 bits at a time.

A *serial port* works more slowly than a parallel port, since it must work with one bit at a time. The operation of the serial port is fairly simple. It receives groups of eight bits from the processor and sends them out one at a time.

On the other hand, when the serial port receives information, it assembles the bits like pearls on a string. The port receives each bit individually, forms them into a byte and passes the byte onto the processor.

If the processor had to do all of this work data exchange over a serial port would be extremely slow, since every bit would be input one at a time.

Since data exchange in a serial manner is really only effective with a machine language routine, we won't be going into that here. There are, however, a complete set of routines in the ROM of the C-128 that can handle the RS232 serial port. You can work with the port using `OPEN 1,2`.

## 12.1.2 The timer

Whenever there is any internal function that requires timing, it is handled by the timer. You can load the timer's register with any value you wish. This value is then continuously decremented. When the register reaches zero the timer sends a signal to the processor. An example of this is the internal use of the interrupts. The timer is programmed so that the intervals are 1/60th of second apart. When the processor reacts to the alarm sent by the timer, the main program is stopped and the interrupt routine is started.

In this case it is time to clear up what happens when an interrupt routine is started. The 0 bit of the memory location 56334 is set when the timer sends the processor a signal. If the bit is set to zero, the timer is still running and no interrupt will happen.

In the C-128 mode interrupts are handled differently, `POKE`'s do not work with bit 0.

Outside of the above, you shouldn't goof around with the timer. In most cases the computer will crash up if the timer is changed.

## 12.1.3. The parallel port

All of the interfaces for the 6502 and 8502 have at least one thing in common: you can program the parallel ports. Usually, the operator works with two such ports, much like the CIA's.

Each of these ports has 8 data lines that you can program for input or output. The chip has two special registers for this purpose. The data control register determines what mode the individual lines are set to. A 1 means output, and a 0 is for input.

You must be careful when you use this register. If there is a 0 for the output mode, as it is with the start up of the computer, then any erroneous impulse to the peripheral devices could cause unknown problems. For example, data on a diskette could be erased.

The second register for the ports has different modes for different purposes. The first byte is for the input functions of the register, so the processor can use the input data from the port.

## 12.2. How do I use the User port?

The User port allows us to use a parallel port and different drivers for peripherals. Most of these drivers run using internal signals. So we are left with an 8-bit wide port and a "borrowed" controller. The controller is borrowed in the sense that the CIA 2 controls port A.

The CIA 2 has its base address at 56576. That is also the location of the addresses for the data register for port A (Reg 0) where bit 2 is the condition of the controller. All of the other controls are internal, therefore we can only use bit 2.

Another method is to use register 1 (56577). This is the data register for port B—the single User port we can use. In this case we can use all eight bits in the register. The data control register follows the number 2 (56578 for Port A [**Caution**—change only bit 2] and number 3 (56579 for Port B).

To turn on all 8 data lines for programming use:

```
POKE 56579,255
```

To turn off the data lines for input use only:

```
POKE 56579,0
```

When you are programming the controller you have to use some caution. To enable the input, type:

```
POKE 56578, PEEK( 56578) AND 251
```

To disable it again, type:

```
POKE 56578,PEEK(56578) OR 4
```

To write data out of this port we simply put the value in location 56577. The reverse allows us to read in values directly.

The port is opened to send or receive data by using:

```
POKE 56576,PEEK(56576) OR 4
```

and it is closed with:

```
POKE 56576,PEEK(56576) AND 251
```

If both commands are set directly from the program we can create a short pulse.

The command control covers Pin M of the user-port (See Appendix 2 or the C-128 System Guide). The eight data lines are found on pins C through L.

## SUMMARY OF PROGRAMMING THE USER-PORTS

Data control register for the 8 data lines : 56579.

Data control register for control: 56578 (only bit 2).

Data register for the port: 56577.

Data register for control: 56578 (only bit 2).

## 12.3 Examples of applications

The User port has many different applications. We won't give an example program for this section. However, we will give you a couple of suggestions to investigate. The simplest example uses lights or LED's, because they behave like transistors or relays: they're either on or off. You can make a light organ, where music is displayed by lights blinking on and off. With another program you can create more effects like this.

It is conceivable to couple two Commodore computers (it doesn't make any difference what types as long as they are both using user-ports), to transfer data back and forth. It is possible to transfer VIC-20 data to a C-128 at the same time the C-128 is displaying the input on its large high-resolution graphics screen.

Modems can be built with a serial interface, so that data can be sent over telephone lines. Another possibility is the connection of non-Commodore computers to the C-128. This is also true for teletypes, card punchers or readers, home robots, or calculators. The number of possibilities for the hobbyist are endless.

# Chapter 13

# BASIC and the operating system

# BASIC and the operating system

The operating system and BASIC offer us many commands that we have not detailed in any of the earlier sections. These functions (like LIST) are very helpful in accomplishing different tasks. We will discuss some of these possibilities in this chapter.

## 13.1 Making BASIC line numbers for a program

Let's say we want to write a program that will draw a graph of a function in high resolution graphics on the monitor screen. If there is a function the computer doesn't have, we must find a way of typing in the formula. One way is to allow the user to write a function in a specified program line using the function command DEFFN. For this the user must know how to program. It would be easier to be able to use an INPUT command to get the function. This means the input will be a string. Strings, of course, cannot be used directly to calculate values. The last possibility is to allow the computer to program itself. This turns out to be quite easy.

To understand this method we will take a quick look at how program lines are normally used. It all begins when a user types in a series of numbers and letters. These characters appear on the screen. If one of these characters is a <RETURN>, the BASIC-Interpreter puts the entire line (not just the typed-in characters) into an input buffer and changes the character sequence in a program line. Or if there is no line number at the beginning of the line, the BASIC interpreter directly executes the command. To use the interpreter then, it is possible to either type the lines or PRINT them. This is the basis for our method. Next, the text of the program line is output to the monitor screen. Now we only have to put a line number onto the beginning of the line. To do this we will use the keyboard buffer. We will put the ASCII-code in the keyboard buffer using a POKE. We follow this with an END. This causes two problems. By creating a new line in the C-64 mode, all the variables are reset (just like the normal program input). It is important to make a new line only at a point where no important data exists, like at the beginning of a program. If you have to save some variables, you can do this by POKEing the variables into free RAM.

Finally, the program should continue where it left off. This means we need a GOTO *xxx* line. This should follow the same pattern at the existing lines. Here is an example:

```
10 INPUT "TERM: Y=";A$: REM input function.
20 PRINT "(CLS)(3xCRSR DOWN)100 DEFFN F(X)=";A$:
   REM output line number
30 PRINT "GOTO 70(HOME)";: REM command to continue
   with the program.
40 POKE 842,13: POKE 843,13:REM 2xreturn
50 POKE 208,2: REM initialize keyboard input.
60 END
70 ...
```

The following lines must be changed for the C-64.

```
40 POKE 631,13: POKE 632,13:REM 2xreturn
50 POKE 198,2: REM keyboard initialization
```

If you type in this program and run it, you should begin to understand how this application works fairly quickly. This program can be run as often as you like. If you type something wrong, the interpreter will halt and print SYNTAX ERROR.

This application can be expanded. With this method you can even get rid of program lines that you will no longer use. It is possible to put entire subroutines into a program using only INPUT commands.

## 13.2 Protecting listings

It is possible to build code words for programs that contain personal data. The code word must not be LISTable. To protect these lines we use a POKE command.

To understand what we are going to do you should know a little about the format of the program line in memory. The first two bytes in the line are the pointer to the next line. The interpreter uses these bytes to jump from line to line. If both bytes are 0, there are no more lines in the program.

After the pointer come two bytes with the line number. These are similar to the pointer. Finally we find the commands in interpreter code. The end of a line is represented by a null. We can trick the interpreter by using this 0. If we POKE this 0 directly after the line number, when the program is LISTed the interpreter believes this line is empty and goes on to the next (the pointer is still ok). A GOTO is also unchanged, since the specific line in the text (the one being searched for) is still oriented with these line numbers. The routine that gets the next command in a program doesn't do this, it simply jumps over 4 bytes after reaching a zero. So in order to use the command you have to type the 5 needed characters (but not a command word). The first of these characters is written over the 0, the rest are used as place holders.

How do we know what bytes to overwrite? There is a trick for this too. We will build a STOP command before the protected line and let the program run up to this point. After the BREAK the pointer to the next command is in locations 61 and 62. If the STOP command is at the end of a line, the pointer points to the 0 byte at the end of the line. If you add 5 to this address, you will find the hidden bytes. To continue, use POKE AD,0. After this command the LISTing will still only give the line number. The text is no longer being pointed to. Now we have to get rid of the STOP command. Here is a summary of the steps:

1. Put a STOP before the line
2. Put 5 colons before the next line of code (placeholder,code)
3. AD=PEEK(4610)+256*PEEK(4611)+5
   (C-64: 61/62)
4. POKE AD,0
5. Get rid of STOP command

If you want to protect the entire listing, the easiest thing to do is change the vector to the LIST routine in the zero page. By doing this the computer will not find the LIST subroutine. This vector is in locations 774/775. Using POKE 775,139 (C-64: POKE 775,1) this vector is broken. To restore the LIST command use POKE 775,81 (C-64: POKE 775,167).

## SUMMARY OF PROTECTING LISTINGS

C-128: Turning on the protection POKE 775,139

Turning it off POKE 775,81

C-64: Turning on the protection POKE 775,1

Turning it off POKE 775,167.

## 13.3. RENUMBER

One of the most useful commands in BASIC 7.0 is RENUMBER. This command can also be simulated in the C-64 mode.

As you learned in the last section, every program line begins with two pointers. The first points to the beginning of the next line, the second really isn't a pointer, it's actually the current program line number. If we add 2 to the address of the pointer we get the address of the next line number. Using this technique we can search for all of the line numbers and by using POKE we can change these numbers. Here is a program to do this:

```
63900 BA=PEEK(43)+256*PEEK(44)
63910 INPUT "Start#";SA: INPUT "Step size";SW
63920 HI=Sa/256: LO=SA AND 255
63930 A=PEEK(BA+2)+256*PEEK(BA+3)
63940 IF A>= 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 "=" A: GOTO 63920
```

This routine must be placed at the end of an already existing program, and run with RUN 63900.

Line 63900 calculates the base address of the first line from the pointer to the start of BASIC. Line 63910 allows the user to enter the new starting line number for the program. If you want the program to start at line 10 and have all line numbers separated by 10, you should enter a 10 twice.

Line 63920 calculates the high and low bytes of the new line number, line 63930 has the old line number from memory. If this number is greater than or equal to 63900, then the renumbering program is ended, since this routine will not renumber itself.

The next lines POKE the high and low bytes of the new line number into memory.

Finally, the base address of the next line has to be calculated. The line number is increased by the step size, and the numbering protocol is output. This protocol points each new line number to the old equivalent. We use this method so that the GOTO, GOSUB and other such jumping commands inside of the program are not changed. If you own a printer I suggest that you print this protocol , so that you can have it in black and white in front of you. To do this put the following at the beginning of the routine:

```
OPEN 1,4: CMD 1
```

## 13.4 RENEW

The NEW command is one of the most frequently used BASIC commands. By typing its three characters (and pressing <RETURN>) the programmer, who has worked very hard on a program, can lose all of his work because he forgot to save it. Fortunately, we have a program that can help avoid the NEW-catastrophe.

The NEW command does not clear the program memory—it clears the two central pointers to the program. The first points to the starting point of the variables at the end of the program. After the NEW it will point to the program beginning for all new lines or variables. Since the pointer will be incremented after each new line, you cannot use any new commands after a NEW, if you still want to use RENEW. This includes typing a single character and getting a SYNTAX ERROR.

The second pointer finds the first line of the program. It is located in addresses 45 and 46, (2049 and 2050, 7169 and 7170, or 16385 and 16386: C-64). Now these contain nothing but zeros. This marks the end of the program. Therefore, there are two things for RENEW to do:

1. Search for the first line. It is marked with a 0. If this null is found, you have to add one to this for a pointer in the bytes 45 and 46 (2049 and 2050 or 7169 and 7170: C-64) and POKE it into memory.

2. Search for the end of the program The program end is found when there is a zero in the high byte of the pointer to the next line. If the end is found, add two to the address, this is the beginning of the variable region of the program.

Here is an application for both modes:

In the C-128 both operations can be operated from ROM routines. Type in the following instructions:

```
POKE (PEEK(45)+256*PEEK(46)),1:BANK 15
SYS 20303: SYS 20354
```

Type everything just as printed.

The POKE command resets the pointer to the first line. The next SYS command calls the routine to calculate the new line pointers. This subroutine starts all over and calculates the pointer from scratch. If the last line is found, the pointer to the last byte of the program is in an internal register. From this the next ROM routine calculates the pointer to the end of the text (4624/4625). That's all there is to it!

Before you try this trick, you have to pay close attention to everything. The POKE command has to always go directly to the starting byte of the program. If the pointer (bytes 45/46) is changed by something like a RESET or GRAPHIC or CLR, it must be set to its old value before the RENEW.

## 13.5 RESTORE

Perhaps you have faced the problem of getting data from the middle of a data line. This is no problem with BASIC 7.0, where we have the RESTORE line number command. In BASIC 2.0 there is also a useful command to do this.

We can remedy this problem with a pair of POKE commands . To do this we must understand that the interpreter saves the address of the last DATA element in the zero page much like it does with line numbers. The line number is saved as a byte pair (like pointers) in locations 63/64. The address of the byte to the last element of the DATA elements is found in 65/66.

If we want to simulate a RESTORE, we follow the next steps:

1. Read the data up to the needed element. If you want the 5th element you have to read the first 4.

2. `PRINT PEEK(63),PEEK(64)`

   The numbers that appear represent line numbers. Take note of these.

3. `PRINT PEEK(65),PEEK(66)`

   We have to take note of these numbers too. We build a pointer to the byte after the last DATA element.

4. `POKE 63,1. number:POKE 64,2.number`
   `POKE 65,3. number:POKE 66,4.number`

These commands are used in place of RESTORE in a program that needs its data pointers reset to the point before the read.

### SUMMARY OF THE RESTORE

The line number of the last DATA elements is in the memory locations 63 and 64. The address of the byte after the last element is in bytes 65 and 66. Both points can be changed with the POKE command.

## 13.6. Different tricks

After a break in the program or an error occurs, the computer still points to the last line completed. If you hastily removed something from the screen, there is usually no way to get it back. In this case the C-128 has the system variable EL and in the C-64 mode the locations 59 and 60 are helpful. They will contain the last line number used.

```
PRINT EL    (or)
PRINT PEEK(59)+256*PEEK(60)
```

If you want to prevent a program from being changed and then saved you can use the following sequence:

```
POKE 818,50 (C-128 mode)
```

This changes the vector so that it no longer works for saving programs.

Finally, here are some SYS locations with routines that could be helpful in programming:

SYS 65499 sets the TI$ to 000000. This is quicker than setting a new string.

A nice way to end programs is to use SYS 19910 (C-128) or in the C-64 mode SYS 42115 instead of END. This is a warm start of BASIC. This is the same as BASIC in the direct mode. READY will not appear and the cursor goes to the next line.

If you want to reset the computer at the end of a program use SYS 57344 (C-128) or SYS 64738 (C-64).

If you want to turn on the C-64 mode without the over used phrase "ARE YOU SURE?", look into the command SYS 57416.

If you feel that 1 MHz is much too slow in the C-64 mode, then the following POKE should make you happy:

```
POKE 53296,255
```

This turns on the 2MHz clock. If this is too fast use:

```
POKE 53296,0
```

## SUMMARY OF TRICKS FOR THE OPERATING SYSTEM

The last line number is in EL or memory locations 59 and 60.

SAVE protection: POKE 818,50

TI$ set to 0: SYS 65499

END without READY C-128: SYS 19910

END without READY C-64: SYS 42115

Reset C-64 mode: SYS 64738

Reset C-128 mode: SYS 57344

GO64 without control question: SYS 57416

2-MHz operation: POKE 53296,255

2-MHz operation off: POKE 53296,0

# Chapter 14

# Introduction to machine language

## Introduction to machine language

There are many publications that print programs for you to type in. Quite often these programs are written in machine language. This can look like gibberish to the beginning programmer. Machine language is not as easy to learn as BASIC, but it runs a lot faster and offers more possibilities to the programmer. For these reasons we are going to present the basics of machine language programming. After you are finished with this chapter you will know the basics of machine language programming and you will be able to understand how such a program works. You will also have the background to continue if you want. If you find that you don't like machine language it will not be a total loss. You can still use what you learn for other purposes, because languages like PASCAL or LOGO can also use this new information.

We are going to discuss both machine languages in the C-128 (8502 and the Z80).

## 14.1 What is machine language all about?

As you already know, machine language is the only way to program the processor directly, without the help of an interpreter or compiler. Because you are directly using the processor there are more possibilities for programs.

Machine languages have different commands that can be used to perform the same operations of BASIC or other languages. You can group the commands into three categories. The easiest to understand for the BASIC programmer are the jump commands. These are similar to the `GOTO` or `GOSUB` commands. Other commands are for data manipulation (for example: Addition). The last group are for the operations that move data from location to location within memory.

The first point we should make is that microprocessors do not have variables. They only know about the different memory locations and some internal registers. You are responsible for determining the difference between the data range and the program range of memory. As a general rule data can only be manipulated in the registers.

A machine language command is always comprised of an operation code (or *opcode*). This is the "number" of the operation. This opcode can be up to 3 bytes for the Z80 (it is always 1 byte for the 8502). In addition, the command can have up to 2 bytes for data. The Z80 can theoretically have a command up to 5 bytes long. Practically, though, the largest command is only 4 bytes—the 3-byte commands only have 1 byte for data. The 8502 commands can be up to 3 bytes long.

## 14.2 The clock

Inside every computer is a small quartz clock (4, 2, or 1 Megahertz = 4, 2, or 1 million ticks or cycles per second). This clock is needed to synchronize the different IC's. If the clock wasn't there, data in memory could be sent to the processor when the processor isn't ready to receive it. The microprocessor can also operate on more data in a shorter period of time with the help of the clock.

## 14.3 The structure of a microprocessor

Every microprocessor has internal registers where all of the operations take place. The most important register is the Accumulator. Most of the arithmetic and logical operations use the accumulator. The accumulator (short Acc or A) is an 8-bit register, it can therefore hold only 1 byte of data at a time. Most arithmetic operations use two operands (for example addition needs two numbers to add together). The first operand is in the accumulator, the second is in another register in the processor or stored in a memory location. After the addition the result is placed back in the accumulator. This is the same for all processors.

Another register has the name F for the Z80 and P for the 8502. It saves different flags and these vary from processor to processor. Some processors have a flag shows whether or not the contents of the accumulator is 0 or not.

The 8502 has two index registers (X and Y) each with 8-bits. These are extremely versatile registers.

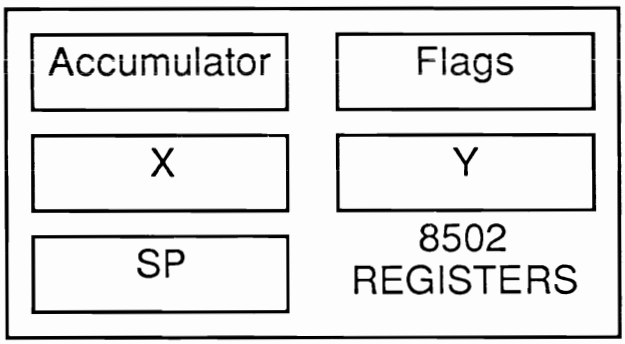



DIAGRAM 14.3

The Z80 has 6 more 8-bit registers with different qualities. Every two of these registers can be combined to form a 16-bit register. The pair HL is already a 16-bit register, it can be used as a 16-bit accumulator. This allows you to use larger numbers.

IX and IY are two index registers (each having 16 bits), that are pointers to the different locations in memory. Using these pointers it is easy to manipulate groups of data. We will explain how this works later.

The 16-bit register SP has a special purpose. This register always points to the top element of the stack (SP stands for stack pointer). Whenever a new element is pushed onto the stack or an old element is popped off, this register is appropriately changed to point to the top.

Finally, there are the I and R registers, which also have special purpoes.

If that was not enough, there are the registers A through L. These have a secondary register that can be switched with the jump register. You can work with individual registers whenever you want. The secondary register acts as an inbetween register.

These are all of the registers for the Z80 (see the appendix). In the next section we will go over how a command is actually run in the processor.

The number of registers only says a little about the performance of the processor. The 8502 can use the zero page much more efficiently than the Z80.

## 14.4 The operation of the microprocessor

Imagine that there is a machine language program in your computer's memory that is just waiting to be run. The processor must know where the program is in memory. There is a special 16-bit register for this purpose. This is the *program counter* (abbreviated PC). It has the address of the next command in memory to be executed. If this command is to be used, then the processor gets the contents of this address. After this byte has been brought into the processor, this register is incremented by one so that it points to the next command. At the same time the opcode is decoded. In other words, the microprocessor determines which command is to be executed. Some commands for the Z80 are composed of a few bytes (this means there could be as many as 256 commands). In this case, it is easy to decode since the bytes are all right next to each other (The PC always points to the actual location, since it is always incremented each time). Even so, the command could still have one or two bytes of data. These are also read in, but they are not decoded. They are stored in the specified registers, or are operated upon by the processor.

If we still have to change the data (by addition for example) then the processor performs the operation and saves the result (perhaps in the accumulator ). Then the processor goes on to the next command.

All of this does not happen instantaneously, for even electricity requires a certain amount of time to flow. Normally the Z80 takes a clock cycle to do things like get the opcode, decode it, perform the command, and save the result. More complex commands can take more cycles to operate. The 8502 has a different architecture and everything operates on two clock cycles.

### SUMMARY OF THE MICROPROCESSOR'S OPERATION

The PC always points to the location in memory of the next byte in the program. The data and the opcodes are read one after the other. The opcode is decoded and finally the command is performed.

## 14.5 The hexadecimal system

Whenever you program in machine language you express all your numbers in the hexadecimal system. In contrast to our everyday decimal number system, the hexadecimal system has 16 digits (rather than 10). These are 0-9 and A-F (for the values 10-15). This is the most frequently used number system, since the conversion from hexadecimal to binary is very easy. An advantage to the hexadecimal system is that each digit actually represents one half of a byte (four bits). The largest 2 digit hexadecimal number FF has the binary digit combination 1111 1111. This is the largest number representable by a single byte. It is easiest to work with a half-byte and convert it into a hex-digit. Here is a table for the decimal/binary/ hexadecimal number system equivalences:

| BIN | 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 |

If you had the byte 1010 1011, this would have the hexadecimal equivalent $AB (1010=$A and 1011=$B). This also works in the opposite direction.

To convert from hex to decimal values, change each hex-digit to its decimal equivalent. The rightmost digit is multiplied by the value 16^0=1, the second is 16^1=16, the third 16^2=256, etc. Finally, all of these values are added together, and the result is the decimal equivalent. Here is an example:

ABCD (The values 10,11,12,13,) =10*16^3+11*16^2+12*16^1+13*16^0

In the reverse direction (decimal to hex) you can divide the number by 16 and then convert the remainder to hex.Then repeat the process with the result. Continue to do this until the result is zero. Here is another example:

```
                5300/16 = 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 calculators that have this special function of switching bases (dec->hex, hex->dec, etc.). Good assemblers also have this function.

The C-128 has a similar function. The BASIC function `DEC` converts hexadecimal values into decimal values. Its syntax is `PRINT DEC(`*value*`)`. The C-128 also includes the `HEX$` function. Its syntax is `PRINT HEX$(VALUE)`.

## 14.6 Binary arithmetic

### 14.6.1 Addition

There's no reason to be scared off by the binary number system—it differs from the familiar decimal system only in the way it represents the values. Other, than this different value representation, they function in the same manner.

The sum of the zeros or a zero and a one (regardless of order) requires no carry to the next place. This is just normal addition. However, if we want to add 1+1, then we have the problem of a carry. In the decimal system this would have the result 2. However, there is no 2 in the binary system. We must then perform a carry to the next column:

```
  0     0     1     1
+ 0   + 1   + 0   + 1
---   ---   ---   ---
  0     1     1    10
```

It is just as easy to add together two byte values. Here you just add up each column, and make sure that you don't forget the carry:

```
01101101  =   109
00001001  =   + 9
    1 1       (carries)
----------    -------
01110110  =   118
```

In this case it is possible to add two ones together and have the carry enter a situation that already has two ones. This results in the situation 1+1+1=3. In binary this would 11 (this should already be clear). Examine the example below:

```
  10010011
+ 11011111
  1  11111  (carries)
  ----------
 101110010
```

This gives us a result that has 9 bits! The ninth bit is called the *carry bit* or the *overflow bit*. When this bit is set it indicates that the result of the two-byte addition is greater than the 8-bits can hold (0-255). If this is the case you must move up to 16-bit addition. There is no computer with only 8-bit addition, since most applications require a greater number range. The problem lies in the fact that an 8-bit microprocessor can only handle eight bits at a time. If we need 2-byte long addition, then the processor must add each word together one after the other. Now the only problem we have is the carry bit from the addition of the low order byte. This is handled simply by transferring this bit (the most-significant bit from the first addition) as a carry to the least-significant bit of the high order byte. Here is an example:

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

The righthand byte you should recognize from the previous example.

## 14.6.2 Subtraction

Whenever a computer has to subtract one number from another, what it actually does is multiply the second value by -1, and then adds the two values together. This is because a computer can easily perform functions like addition or negation (or AND, OR, NOT, or XOR), but subtraction does not fit into this category.

If you include negative numbers the range of numbers for a single byte is change from 0-255 or -127 to +127. The most significant bit (bit 7) changes the sign of the value. If it is set to 1 then the number is negative, whereas a 0 gives a positive value. Therefore, negating a value is done by simply setting bit 7. Here is an example:

```
  00000001
+ 10000001
-----------
10000010
```

In the decimal system this would be 1+(-1) = -2, which is obviously wrong. For this reason, we use another method. This method is called two's complement. A byte can be converted to two's complement quite easily with a multiplication by -1. By doing this we invert all the bits and then we simply add a one to the result.

```
Example:      01011011
Inverted:     10100100
            +        1
              ----------
              10100101
```

Using this scheme with 1-1 in the binary system we get:

```
              00000001
            + 11111111
              11111111  (carries)
              ----------
            1 00000000
```

As you can see there is still a carry. This is a result of performing subtraction. In this case we can ignore the final carry. If this were a 16-bit subtraction, then the previous two bytes are all zero. This gives us the result we would expect. It is important, if you are using 16-bit subtraction, to invert all 16 bits. Therefore, for a two-byte value of a -1 the binary two's complement version looks like: 11111111 11111111. If we overlook the carry bit now, we would get 11111111 00000000. This is wrong.

Luckily, programming a subtraction is not this complicated. The subtraction command in both processors has this two's complement procedure built right into it.

### 14.6.3 Multiplication

In case you don't believe it, machine languages only have two arithmetic commands, these are addition and subtraction. All other commands are a result of using just these two base commands, mostly as subroutines.

Since we won't be going into the intricacies of each machine language (we'll save that for another book), we will only go over the simplest algorithm for multiplication. This is not a widely used routine, since it is not very efficient, but it does work.

To calculate the product of x*n, is the same as adding x n-times. Obviously, this only works for whole numbers. For numbers with a decimal point, we have to use multiplication column by column, however, the same basic principle holds.

Here is an example to better show what we are trying to say:

$$4*3 = 4+4+4 = 12$$

## 14.6.4 Division

For division there is also a very simple algorithm. To divide x into n parts, you simply subtract n from x repeatedly. The number of subtractions it takes to get to the point where n is larger than x is the integer portion of the division. Here is an example:

```
    10/3 = ?
    10-3 = 7        count  = 1
    7-3  = 4               = 2
    4-3  = 1               = 3

=>  10/3 = 3  remainder  = 1
```

This algorithm works since the machine language allows for so many possibilities. A calculator works on the same principle, where each time you press a key you are also running a subroutine (naturally, with much more efficient algorithms).

From the four basic arithmetic operations you can perform higher functions like percentages, or the Sine function. You can actually perform all mathematical operations with the small functions AND-, OR-, XOR-, and the NOT- (In fact addition and subtraction are actually constructed from these functions).

## 14.7 How do comparisons work?

Comparisons are not uncommon in BASIC. However, how can you perform comparisons with a machine language? Here is an example of how we can do this:

A=B (=) A-B =0

As you can see the comparison of 2 numbers (here A and B) can be made. The computer knows the two are equivalent since the result on the right-hand-side is 0. The 0 is the number in the microprocessor's internal registers which can be determined if it is there or not. This happens for each bit-pair, then all of the bit locations are OR'd together:

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 are set to 0, then the result is a zero. If only one bit pair has a result of one, then the result will be one. Now, with this information, the processor can determine if the numbers are equal or not. If we use the accumulator we can determine equivalence by looking at the Z flag (Set if the accumulator's contents are zero).

For comparisons of greater and less than we will perform operations similar to those for equals. After the subtraction we see if the value in the accumulator is less than or greater than zero, this is checked in the sign bit.

A greater than B (=) A-B > 0 (bit-7=0)
A less than B (=) A-B < 0 (bit-7=1)

The sign bit is displayed in the S-flag in the Z80 (S means sign). This can be checked easily with special commands. The 8502 calls this the N-flag (N=negative).

Another Z80 flag is called the P/V flag. In this book we will simply call it the P flag. Th P flag flag has two functions. It can tell the parity (even or odd—you do not need to understand what this means). The P flag can also describe whether or not the sign bit was changed during an arithmetic operation.

The 8502 has a similar flag called V. This is also used for the detection of a change in the sign bit.

# Chapter 15

# 8502 machine language

# 8502 machine language

Now we are going to deal with al lot of technical information. Before we take our first steps in the fastest programming language in the world, you should take a look at the contents of the appendix. It describes all of the commands of the 8502. From these you should get a quick overview of what the language contains. Don't be alarmed—you don't need to understand everything.

## 15.1. The first 8502 program

We are going to use only addition and subtraction in this machine language program, since this is the easiest way to learn how the processor operates. We will begin with an addition program for two eight-bit numbers.

We used to POKE the two bytes that we wanted to add in to memory. There is no doubt that in BASIC this is a very nice method. However, there is no INPUT command in machine language.

The first principle of machine language programming is this: All data manipulations occur in the accumulator. For this reason the first value of the command must be loaded into the accumulator. We use the command LDA $*nnnn*. To clear the carry bit before any operation, we have to use the CLC command. Finally, we get to the addition command ADC $*mmmm*. This command takes the second number out of the memory location *mnmm* and adds it to the contents of the accumulator. The result is put back in the accumulator.

To save this result we use the STA $*xxxx*. This saves the memory location where it could be PEEK'd later. This completes the addition. All that remains is to return back to BASIC with RTS.

The entire command would look something like this:

```
LDA $nnnn
CLC
ADC $mmmm
STA $xxxx
RTS
```

These commands should be stored somewhere convenient in memory. Then we only need to remember where this place in memory starts. Luckily, the developers of the C-128 remembered to save a region in memory for machine language programs. It is in the range from 1300 through 17FF in BANK 0. It is easiest if the above program started at the first location (1300).

To load this program we simply use the ROM program MONITOR. This is called by its name from BASIC. Type in the following:

```
A 01300 LDA 17FD (CR)
```

This will assemble the first command, and translate the entered program into memory location 1300. The cursor skips to the next line and prints another A and the address 01303. The previous line now contains the command itself and the translation in its numeric equivalent.

All other programs are assembled in a similar manner. If you want to type in another program, all you need to do now is hit <RETURN> and the MONITOR exits the assemble mode.

When everything is typed in, the screen should look something like:

```
    PC      SR   AC   XR   YR   SP
;fb000      00   00   00   00   f8

a  01300  ad  fd  17  lda  $17fd
a  01303  18          clc
a  01304  6d  fe  17  adc  $17fe
a  01307  8d  ff  17  sta  $17ff
a  0130a  60          rts
```

If you made a mistake in the program you can change it as you would a BASIC command. Just move the cursor back to the mistake and type over it.

The program now is in memory . We only have to give the numbers to be added. The MONITOR also has another possibility. Type the following:

```
M  017fd, 017ff
```

With this command the C-128 displays eight bytes in a single line. This should not disturb you since the only part we are interested in is the first three hex values.

We will put the two numbers we want to add together (for example $38 and $05 (decimal 56 and 5) in the first two bytes. You can change the values once again just by moving the cursor and typing over the old values and pressing the <RETURN> key.

Using the command `X`, we can turn off the monitor and start up our machine program using the command `SYS 4864`.

If you made a mistake the cursor will disappear. In this case you will have to turn off the computer and then turn it back on again to start it all over. Otherwise you can use the addition program in two ways:

a. `PRINT PEEK(DEC("17FF"))`

b. You can go back into the monitor and retype `M 017FD,017FF`. The third byte will have the result.

You should run this program a few times with different operands, so that you better understand how this works.

## 15.2. The second step: 16-bit addition

As we have already pointed out you will have to use 16-bit addition for larger values. If you don't, you will have to come up with other algorithms because of the small numbers available with 8-bit addition. Below we have listed a program that will add any 16-bit number to a given constant. The number must be in location 17FF (high byte) and 17EE (low byte). A 16-bit value requires 2-bytes and two addition steps. Everything begins in the previous example with:

```
LDA $17FE    and
CLC
```

To add the constant we use ADC #*lowbyte*. This command doesn't get its value from an appointed location, but directly from the following byte (the constant).

After this command the lower half of the result is finished. This is saved using the command STA $17FC. The carry is saved in the carry bit and is not cleared after loading the second half with a LDA $17FF. Now we can continue with a normal addition with ADC #*highbyte*.

STA $17FD saves the second part of the result. Finally we return to the calling program with the RTS command. Here is the entire listing:

```
1300  LDA $17FE
1302  CLC
1303  ADC #E8
1305  STA $17FC
1307  LDA $17FF
1309  ADC #03
130B  STA $17FD
130D  RTS
```

In this case the constant is 03E8=1000 in the decimal number system.

## 15.3. Subtraction

We have already discussed how subtraction with the carry bit and the computer commands work. Here is a short program listing (similar to 8-bit addition).

```
1300 LDA $17FF
1302 SEC          (set the carry bit)
1303 SBC $17FE    (subtraction)
1305 STA $17FD
1307 RTS
```

## 15.4. Multiplication

You should remember from section 14.6. that multiplication is simply a repeated addition. We will write a machine language subroutine for multiplication using this method.

If we use this method, we can make use of the 8-bit addition we wrote earlier. We will start with this addition routine.

If we multiply two 8-bit values we can get a result up to 16-bits long. The addition routine must be set up to add an 8-bit value to a 16-bit result. When this is done the carry bit must be added to the highbyte. It all happens like this:

```
LDA $17FE   (low byte of the result)
CLC
ADC $17FC   (8-bit addition)
STA $17FE   (save the result)
LDA $17FF   (high byte of the result)
ADC #00     (add 0 and the carry bit)
STA $17FF   (save the result)
RTS         (end of subroutine)
```

After this addition the result is in 17FE/17FF, the 8-bit number is in 17FC. Now all we need is the length of the loop.The length given is in the multiplicator in memory location 17FD.

The simplest method to program a loop of variable length for each run of the program, is to use a special register and decrement it by 1. When the register reaches zero, then the loop is ended. The best register to use for this purpose is the X register. At the beginning of the loop the register is initialized with `LDX $17FD` (location $17FD contains the size of the multiplicator).

Now we get to calling the routine with `JSR $addition` (replace addition with the location to jump to). After the call to the addition routine the loop counter has to be decremented (that is subtract 1 from this value). This can be done with the one byte command `DEX`. One thing to be careful about--this command will also change the Z- and N- flags. They tell us if the value in the X register is negative or zero. Remember, though, these control bits are not only for the accumulator.

The opcode table in the appendix describes what flags are changed by each command. The branching commands use these flags to control the branching. We will need such information to operate our loop. We should continue branching to the top of the loop until X=0. The command for this operation is BNE $*address* (BNE=branch on not zero to address). If the Z-bit is set to one, then the result of the last operation was zero. If Z=0, then the result was not zero. The BNE command checks this bit. If it is zero, then the program jumps to the address following the command. Otherwise the next command in line is performed. The branch commands use a relative addressing mode—that is, the distance to the jump location is known. This distance is automatically calculated in the monitor. One drawback to this is that the distance can only be one byte in size. This means we can only have a jump of up to 129 bytes forward and 126 bytes backward. At first glance this doesn't seem like very much, but in actuality you will seldom need any more than this.

Before the loop begins, the two resultant bytes have to be cleared to zeros. See for yourself:

```
1300 LDA #00
1302 STA $17FF  (clear 17FF)
1304 STA $17FE  (clear 17FE)
1306 LDX $17FD  (load X)
1308 JSR $130E  (call subroutine)
130A DEX        (decrement X)
130B BNE $1308  (branch)
130D RTS        (end of main program)
130E LDA $17FE  (top of addition)
1310 CLC
1311 ADC $17FC
1313 STA $17FE
1315 LDA $17FF
1317 ADC #00
1319 STA $17FF
131B RTS        (end of program)
```

As you can see, the branch command is just as easy to use as a simple jump command.

# Chapter 16

# Programming the Z80

# Programming the Z80

So that you can learn the different machine languages independently, we will go over the same topics that we did for the 8502. Don't worry about learning one better than the other.

## 16.1 The first Z80 program

We will start this section by writing a very simple program. One of the easiest is an 8-bit addition program.

First of all, our program must somehow receive the two numbers to be added together. There is nothing like an `INPUT` command in machine language—the computer must get its values directly from memory. That is the first problem of our program. We can use the command "`LD A, (nnnn)`" much like we used the `PEEK` in BASIC. It takes the contents of *nnnn* and puts them in the accumulator.

Let's say we know where the second byte is located. However, we cannot bring it into the processor with a command like `LD B, (nnnn)`, since there is no such command in the Z80 machine language. What we can do is use the `HL` register pair as a pointer to our second value. What we do is load the address containing our value with `LD HL,` *nnnn*. Notice that the *nnnn* is no longer in parantheses. This indicates that the address is directly loaded into `HL` (not the contents).

The next command finally adds the two values together: `ADD A, (HL)`. Its meaning is to add the contents of the accumulator with the contents of the address stored in the HL register pair. The result is returned to the accumulator. If the result is out of the value range (0-255) then the Z80 sets the carry bit to 1.

If we don't need to use the result in the accumulator, we can use another command to store the result back into memory, where we can use a `PEEK` to look at it later. This is done with `LD (nnnn), A`. This command works much like the `LD A, (nnnn)` but in reverse.

Finally, we end the program with JP FFE0. This jump turns the 8502 back on, which is the processor which BASIC uses.

So far we have been programming with random locations. In our example you can use the addresses you want. All you have to do is make sure you are not using a region which the computer has to use for other purposes. For this reason we put the program in the free region of RAM from 1300 through 17FF. The data bytes are at the end of this region.

This is how our program looks:

```
Address     Command         Comments

1700     LD A,(17FF)   Load first value
1703     LD HL,17FE    2nd number in 17FE
1706     ADD A,(HL)    Add Acc and 17FE
1707     LD (17FD),A   Save result
170A     JP FFE0       End of program
```

If you look at the addresses you can see that different commands take up different amounts of memory. Commands using an address have at least 3 bytes. Others, like ADD A,(HL), use only a single byte.

Before you can POKE these bytes into memory, you have to know what they look like. In the appendix with the Z80 commands you can get a list of all of the opcodes. The opcode for LD A,(*nnnn*) is 3A and the two address bytes. Notice that the low byte always appears before the high byte. The complete command looks like:

```
3A FF 17
```

Here are the codes for the rest of the program:

```
21  FE 17   (LD HL,AB7E)
86          (ADD A,(HL))
32  FD 17   (LD (AB7D),A)
C3  E0 FF   (JP FFE0)
```

These values have to be POKE'd into memory. This can be done with the following BASIC program:

```
 0   BANK 0
 1   POKE  65518,195: POKE 65519,00: POKE 65520,23
 2   POKE 65500,88:POKE 65502,96
10   FOR I=DEC("1700") TO DEC("170C"): READ A$:
     POKE I,DEC(A$):NEXT
20   DATA "3A","FF","17","21","FE","17",
     "86","32","FD","17","C3","E0","FF"
30   INPUT "#1, #2";Z1,Z2
40   POKE DEC("17FF"),Z1:POKE DEC("17FE"),Z2
50   SYS DEC("2000"):PRINT PEEK(DEC("17FD")):GOTO30
```

Before we can start our routine we must tell the Z80 where our program is. The Z80 will look for the program's starting address in locations FFEE/FFEF. Line 1 sets the starting point at location 1700. Location FFED must have the value C3 which is the opcode for the jump instruction. Line 1 also takes care of this.

We use an 8502 routine to start up the Z80 processor. This routine is already in memory starting at location FFD0. However, we must set locations FFDC and FFDE with the values of 58 and 60 (hex) to return to BASIC when the machine language routine is finished. Line 2 POKEs these in for us.

Lines 10 and 20 load our Z80 routine into the memory starting at location 1700.

Line 30 simply INPUTs the two numbers to be added. Line 40 POKEs the two numbers into the locations in which our Z80 program will look for them.

Line 50 contains a SYS command that starts the 8502 routine which in turn will start the Z80 routine. When control is passed back to BASIC, line 50 PEEKs location 17FD and prints the answer that is stored there. For a more detailed explanation of this type of program see the book *128 Internals* from Abacus Software.

The subtraction program is almost exactly the same, except you have to change the ADD command to a SUB(HL). In the line 20 change the 7th value from 86 to 96.

## 16.2 How to program a loop

In BASIC if you want to perform a section of code more than once, you have two choices: a FOR-NEXT loop or a DO-LOOP. There is a third possibility bu it is not as nice and is not used as often. After each pass we add one to a loop variable, and use an IF-THEN to jump back to the beginning, if needed. No matter how difficult this is, this is the only way to write a loop in machine language. Instead of a variable we use a register. The loop count is put in this register and decremented each time through. The following program does nothing more than loop 255 times. We haven't given you the BASIC program to load this program into memory, since this is fairly easy to do. This machine program executes so quickly that the loop is finished in less than a second. Here is the program:

```
1700 LD B,FF      Load the loop size
1702 DEC B        Decrement counter
1703 JP NZ,1302   Jump to 1302 if B<>0
1706 RET          End of program
```

The first command puts the loop size into register B. This number is built into the program, this is built right next to the opcode. You are probably asking yourself why we chose register B for the loop. Just as there are special commands for the accumulator, there are similar commands for the B-register.

The DEC command (decrement = subtract by 1) subtracts one from the value in register B. When the result is 0, then the zero-flag is set to 1. Otherwise it remains set at 0. When the zero-flag is set to one the next command is performed, rather than jumping back up to 1302. JP NZ jumps to the address at the end of the command when the zero flag is not set. The next command is a RET, which ends the program.

## 16.3 More arithmetic routines

### 16.3.1 16-bit addition

As we said earlier we can only use numbers up to 255 with 8-bit addition. With 16 bits this number is increased to 65535. While the Z80 only has an 8-bit processor, it does have instructions for 16-bit arithmetic. It also has 16-bit registers.

For a 16-bit addition, one number must be in the register pair DE. This is done with the instructions LD DE, (*nnnn*) where the byte *nnnn* is in register D, and E has the value at address *nnnn*+1. This is also in the pointer format (low/high). A similar instruction exists for the HL register pair.

The addition instruction for 16-bits is ADD HL,DE. As you have probably already guessed, this instruction adds the two values in HL and DE and puts the result back into HL. After this step we can save the result in memory with LD (*nnnn*),HL.

The entire program looks like:

```
1700 LD DE,(17FE)
1704 LD HL,(17FC)
1707 ADD HL,DE
1708 LD (17FA),HL
170B JP FFE0
```

Compare this with the same program for the 8502.

Observe that each LD command requires two bytes. The loading program looks like this:

```
 0   BANK 0
 1   POKE 65518,195: POKE 65519,00: POKE 65520,23
 2   POKE 65500,88:POKE 65502,96
10   DATA "ED","5B","FE","17"
11   DATA "2A","FC","17"
12   DATA "19"
13   DATA "22","FA","17"
14   DATA "C3","E0","FF"
```

```
20    FOR I=DEC("1700") TO DEC("170D"):
      READ A$: POKE I,DEC(A$): NEXT
30    INPUT "#1, #2:";Z1,Z2
40    POKE DEC("17FE"),Z1 AND 255:
      POKE DEC("17FF"),INT(z1/256)
50    POKE DEC("17FC"),Z2 AND 255:
      POKE DEC("17FD"),INT8Z2/256)
60    SYS DEC("FFE0"):
      PRINT PEEK(DEC("17FA"))+256*PEEK(DEC("17FB"))
70    GOTO 50
```

This program works in the same way the 8-bit addition program worked. Only the machine language opcodes are different.

## 16.3.2 Multiplication

To multiply two numbers together we need to add a number over and over (just like in section 14.6.3.). In BASIC the algorithm would look like this:

```
10 INPUT "#1, #2: ";Z1,Z2: E=0
20 FOR I= 1 to Z1
30 E = E+Z2: NEXT
40 PRINT "result:";E
```

As you can see, this program needs a single loop and an addition. We have programmed both before in machine language.

The multiplication of two 8-bit numbers has a 16-bit result. This will work out fine with our 16-bit addition routine.

Only the addition command has to be inside of the loop. All of the commands require that the values be stored in a register, or saved in a location in memory. Using the first command, the number 1 is saved in the accumulator. Since the B register cannot get a value directly from a memory location, we store the contents of the accumulator into the B register, then put the second number into the accumulator and do the same thing with the E register. Now the loop is ready. The other 8-bit number is put in register pair HL.

The ADD HL,DE requires the D register to be set to 0 before the addition takes place. The HL register pair must also be set to 0 before the loop.

The loop begins with the ADD command. The HL register is only changed with the ADD command, so it will always contain the result from the previous addition.

The rest of the program is quite easy to figure out. DEC B and JP NZ,170D is the end of the loop, LD (17FC),HL saves the result in memory and JP FFE0 ends the program. Here is the listing:

```
1700  LD A,(17FF)   Number 1
1703  LD B,A        move it into B
1704  LD A,(17FE)   Number 2
1707  LD E,A        move it into E
1708  LD D,00       Clear D
170A  LD HL,0000    Clear HL
170D  ADD HL,DE     Add130E DEC B decrement B
170F  JP NZ,170D    Jump to 170D if B<>0
1712  LD (17FC),HL Save result
1715  JP FFE0       Return to BASIC
```

The loading program follows:

```
 0  BANK 0
 1  POKE  65518,195: POKE 65519,00: POKE 65520,23
 2  POKE 65500,88:POKE 65502,96
10  DATA "3A","FF","17"
11  DATA "47"
12  DATA "3A","FE","17"
13  DATA "5F"
14  DATA "16","00"
15  DATA "21","00","00"
16  DATA "19","05"
17  DATA "C2","0D","13"
18  DATA "22","FC","17"
19  DATA "C3","E0","FF"
20  FOR I=DEC("1700") TO DEC("1717"):
    READ A$: POKE I,DEC(A$): NEXT
```

```
30   INPUT "#1, #2: ";Z1,Z2
40   POKE DEC("17FF"),Z1:
     POKE DEC("17FE"),Z2
50   SYS DEC("FFD0"):
     PRINT PEEK(DEC("17FC"))+256* PEEK(DEC("17FD"))
60   GOTO 30
```

Once again only the opcodes have been changed.

## 16.4. How do we switch processors?

You have probably already asked yourself how the operating systems work. Perhaps you have also noticed that the jump address for the Z80 routine at the end of the subroutine to turn on the 8502 is POKE'd.

If you turn on one processor with the specified bits in the MMU, it is not very easy to turn off the other. Most likely, you are going to have to use a program to switch between the two. If we want to turn the processor back on later, we can POKE a jump command in our program.

For the 8502, we didn't have a JMP command, rather a CLI and an RTS. The CLI turns on the interrupts and the RTS command is for return back to BASIC.

The switch always happens between BANK 0 and BANK 1, but it can only switch operating system routines with POKE's.

## 16.5. Final look

Every processor like the 8502 and the Z80 has its strong points and weak points. The Z80 is better for operating on large blocks of data, whereas the 8502 is good for programming in BCD-arithmetic. BCD means Binary Coded Decimal, where every character of a decimal number is a half byte. You need more memory (one byte requires a number from 0 to 99), but it can be more accurate in calculations, because it doesn't switch to a dual system.

If you want to write a program in machine language you will need to decide which processor is better for what you want to do.

Of course, you can switch in the middle of a Z80 program over to an 8502 routine and use its operating system (even CP/M uses this principle).

# Appendix

# Appendix A:

## Machine commands for the 8502

ADC
The data and the carry bit are all added together into the accumulator. This command effects the N,V,Z and the C(arry) flags.

AND
The input data is ANDed with the contents of the accumulator and the result is returned to the accumulator. This command effects the N and Z flags.

ASL
The contents of the accumulator or a memory location are shifted to the left by one bit. The 7th bit is moved into the carry bit and bit 0 is filled with a zero. This command changes the N,Z, and C flags.

BCC *dd*
If the carry bit is zero the processor jumps to the address given by the address of the next command plus the offset dd. No flags are changed.

BCS *dd*
This is just like BCC, except the jump takes place when the carry bit is 1.

BEQ *dd*
Just like BCC, the jump takes place when Z=1.

BIT
The bits of a memory location are ANDed with the accumulator. The results are not saved, but if the result is not 0, then Z=1 otherwise Z=0. This is an easy way to test for equivalence. Outside of this both the most significant bits are stored in the flags N (bit 7) and V (bit 6).

BMI *dd*
Just like BCC, only jumps when N=1.

BNE *dd*
Just like BCC, only jumps when Z=0.

BPL *dd*
Just like BCC, only jumps when N=0.

BRK
Turns off the interrupts. This condition can be tested by checking if flag B is set to 1.

BVC *dd*
Just like BCC only jumps when V=0.

BVS *dd*
Just like BCC only jumps when V=1.

CLC
Clears the carry bit

CLD
Clears the D flag. The result of this is to turn off the BCD mode.

CLI
Clears the interrupt flag. The result is that interrupts are turned back on.

CLV
Clears the V flag

CMP
The input data is compared with the contents of the accumulator. If all the bits are the same Z=1. If the contents of the accumulator are smaller than the input then N=1. The carry bit is set to 1 if the accumulator is greater than or equal to the input. This is a great help when you want to branch on a given condition.

CPX
Just like the CMP command, however, the input is compared with the X register.

CPY
Just like the CMP command, however, the input is compared with the Y register.

DEC
The given location is decremented by 1. This changes the N and Z flags.

DEX
Just like DEC, except the X register is decremented.

DEY
Just like DEC, except the Y register is decremented.

EOR
This command takes the input data and XOR's it with the contents of the accumulator, the result is put back into the accumulator. The flags N and Z could be changed by this command.

INC
The input location is incremented by 1. The N and Z flags are changed.

INX
Just like INC, except the X register is changed.

INY
Just like INC, except the Y register is changed.

JMP
The processor jumps unconditionally to the given address.

JSR
The processor jumps to a subroutine at a given address.

LDA
The accumulator is loaded with the contents of the given address. Flags to check are N and Z.

LDX
Just like LDA, except the X register is loaded

LDY
Just like LDA, except the Y register is loaded.

LSR
The contents of the accumulator or a memory location are shifted to the right by one bit. Bit 0 is shifted into the carry bit, and Bit 7 is filled with a zero. The N,Z and C flags are changed.

NOP
Waits for two clock cycles. No operation occurs.

ORA
The input data is ORed with the contents of the accumulator, and the result is placed in the accumulator. The N and Z flags are changed.

PHA
The accumulator is pushed onto the stack.

PHP
The flags are pushed onto the stack.

PLA
The top of the stack is popped into the accumulator. Check the N and Z flags.

PLP
The flags are set by the contents of the top of the stack.

ROL
The contents of the accumulator or a memory location are rotated to the left by one bit. Bit 7 is copied into the carry bit, and also moves into bit-0. The flags N,Z, and C are changed.

ROR
The contents of the accumulator or a memory location are rotated to the right by one. Bit-0 is copied into the carry bit and is moved into bit-7. The flags N,Z, and C are changed.

RTI
Returns from an interrupt routine, and restores the program counter and the contents of the flags.

RTS
Returns from a subroutine, and returns the contents of the PC back to normal.

SBC
Just like ADC except this is subtraction.

SEC
Sets the carry bit.

SED
Clears the D-flag. The result of this is to turn on the BCD mode.

SEI
Clears the interrupt flag. The result is that interrupts are ignored.

STA
Saves the contents of the accumulator to a memory location.

STX
Saves the contents of the X register to a memory location.

STY
Saves the contents of the Y register to a memory location.

TAX
The contents of the accumulator are written to the X register. Check the N and Z flags.

TAY
The contents of the accumulator are written to the Y register. Check the N and Z flags.

TSX
The stack pointer is copied into the X register. The N and Z flags are changed.

TXA
The contents of the X register are written to the accumulator. The N and Z flags are changed.

TXS
The contents of the X register are saved in the stack pointer.

TYA
The contents of the Y register are written to the accumulator. The N and Z flags are changed.

# Appendix B:

## Opcode listing for the 8502

| Command | Opcode | Flags | Description |
|---|---|---|---|
| ADC $nnnn | 6Dnnnn | NV ZC | Add contents of nnnn to accumulator |
| ADC $nn | 65nnnn | NV ZC | Add location nn to accumulator |
| ADC #$nn | 69nn | NV ZC | Add byte nn to accumulator |
| ADC $nnnn,X | 7Dnnnn | NV ZC | Add location nnnn to X-register |
| ADC $nnnn,Y | 79nnnn | NV ZC | Add location nnnn to Y-register |
| ADC $nn,X | 75nn | NV ZC | Add location nn to X-register |
| ADC ($nn,X) | 61nn | NV ZC | Add byte from nn to X-register |
| ADC ($nn),Y | 71nn | NV ZC | Add byte from nn to Y-register |
| AND $nnnn | 2Dnnnn | N Z | 'AND' nnnn with accumulator |
| AND $nn | 25nn | N Z | 'AND' location nn with accumulator |
| AND #$nn | 29nn | N Z | 'AND' byte nn with accumulator |
| AND $nnnn,X | 3Dnnnn | N Z | 'AND' location nnnn with X-register |
| AND $nnnn,Y | 39nnnn | N Z | 'AND' location nnnn with Y-register |
| AND $nn,X | 35nn | N Z | 'AND' location nn with X-register |
| AND ($nn,X) | 21nn | N Z | 'AND' byte with contents of X-reg. |
| AND ($nn),Y | 31nn | N Z | 'AND' byte from nn with Y-register |
| ASL A | 0A | N ZC | Arithmetic shift left of accumulator |
| ASL $nnnn | 0Ennnn | N ZC | Arithmetic shift left of location nnnn |
| ASL $nn | 06nn | N ZC | Arithmetic shift left of location nn |
| ASL $nnnn,X | 1Ennnn | N ZC | Arithmetic shift left of nnnn+X-reg. |
| ASL $nn,X | 16nn | N ZC | Arithmetic shift left of nn+Y-register |
| BCC $dd | 90dd | | Jump to PC+dd if C=0 |
| BCS $dd | B0dd | | Jump to PC+dd if C=1 |
| BEQ $dd | F0dd | | Jump to PC+dd if Z=1 |
| BIT $nnnn | 20nnnn | NV Z | Test location nnnn |
| BIT $nn | 24nn | NV Z | Test location nn |
| BMI $dd | 30dd | | Jump to PC+dd if N=1 |
| BNE $dd | D0dd | | Jump to PC+dd if Z=0 |
| BPL $dd | 10dd | | Jump to PC+dd if N=0 |
| BRK | 00 | B I=1 | Software Interrupt |
| BVC $dd | 50dd | | Jump to PC+dd if V=0 |
| BVS $dd | 70dd | | Jump to PC+dd if V=1 |
| CLC | 18 | C=0 | Clear C |
| CLD | D8 | D=0 | Clear D |
| CLI | 58 | I=0 | Clear I |
| CLV | B8 | V=0 | Clear V |

| | | | | |
|---|---|---|---|---|
| CMP $nnnn | CDnnnn | N | ZC | Compare location nnnn with accum. |
| CMP $nn | C5nn | N | ZC | Compare location nn with accum. |
| CMP #$nn | C9nn | N | ZC | Compare byte nn with accumulator |
| CMP $nnnn,X | DDnnnn | N | ZC | Compare location nnnn+X-register with accumulator |
| CMP $nnnn,Y | D9nnnn | N | ZC | Compare location nnnn+Y-register with accumulator |
| CMP $nn,X | D5nn | N | ZC | Compare location nn+X-register with accumulator |
| CMP ($nn,X) | C1nn | N | ZC | Compare byte in nn+X-register with accumulator |
| CMP ($nn),Y | D1nn | N | ZC | Compare byte in nn+Y-register with accumulator |
| CPX $nnnn | ECnnnn | N | ZC | Compare location nnnn with X-reg. |
| CPX $nn | E$nn | N | ZC | Compare location nn with X-register |
| CPX #$nn | E0nn | N | ZC | Compare byte nn with X-register |
| CPY $nnnn | CCnnnn | N | ZC | Compare location nnnn with Y-reg. |
| CPY $nn | C4nn | N | ZC | Compare location nn with Y-register |
| CPY #$nn | C0nn | N | ZC | Compare byte nn with Y-register |
| DEC $nnnn | CEnnnn | N | Z | Decrement location nnnn |
| DEC $nn | C6nn | N | Z | Decrement location nn |
| DEC $nnnn,X | DEnnnn | N | Z | Decrement location nnnn+X-register |
| DEC $nn,X | D6nn | N | Z | Decrement location nn+X-register |
| DEX | CA | N | Z | Decrement X-register |
| DEY | 88 | N | Z | Decrement Y-register |
| EOR $nnnn | 4Dnnnn | N | Z | Exclusive-or location nnnn with accumulator |
| EOR $nn | 45nn | N | Z | Exclusive-or location nn with accumulator |
| EOR #$nn | 49nn | N | Z | Exclusive-or byte nn with accumulator |
| EOR $nnnn,X | 5Dnnnn | N | Z | Exclusive-or location nnnn+X-reg. with accumulator |
| EOR $nnnn,Y | 59nnnn | N | Z | Exclusive-or location nnnn+Y-reg. with accumulator |
| EOR $nn,X | 55nn | N | Z | Exclusive-or location nn+X-register with accumulator |
| EOR ($nn,X) | 41nn | N | Z | Exclusive-or byte from nn+X-register with accumulator |
| EOR ($nn),Y | 51nn | N | Z | Exclusive-or byte from nn+Y-register with accumulator |
| INC $nnnn | EEnnnn | N | Z | Increment location nnnn |
| INC $nn | E6nn | N | Z | Increment location nn |
| INC $nnnn,X | FEnnnn | N | Z | Increment location nnnn+X-register |

| | | | | | |
|---|---|---|---|---|---|
| INC $nn,X | F6nn | N | Z | Increment location nn+X-register |
| INX | EA | N | Z | Increment X-register |
| INY | C8 | N | Z | Increment Y-register |
| JMP $nnnn | 4Cnnnn | | | Jump to nnnn |
| JMP $(nnnn) | 6Cnnnn | | | Jump to contents of nnnn |
| JSR $nnnn | 20 | | | Jump to subroutine at nnnn |
| LDA $nnnn | ADnnnn | N | Z | Load location nnnn into accumulator |
| LDA $nn | A5nn | N | Z | Load location nn into accumulator |
| LDA #$nn | A9nn | N | Z | Load byte nn into accumulator |
| LDA $nnnn,X | BDnnnn | N | Z | Load location nnnn+X-register into accumulator |
| LDA $nnnn,Y | B9nnnn | N | Z | Load location nnnn+Y-register into accumulator |
| LDA $nn,X | B5nn | N | Z | Load location nn+X-reg. into accum. |
| LDA ($nn,X) | A1nn | N | Z | Load byte from nn+X-register into accumulator |
| LDA ($nn),Y | B1nn | N | Z | Load byte from nn+Y-register into accumulator |
| LDX $nnnn | AEnnnn | N | Z | Load location nnnn into X-register |
| LDX $nn | A6nn | N | Z | Load location nn into X-register |
| LDX #$nn | A2nn | N | Z | Load byte nn into X-register |
| LDX $nnnn,Y | BEnnnn | N | Z | Load location nnnn+Y-register into X-register |
| LDY $nnnn | ACnnnn | N | Z | Load location nnnn into Y-register |
| LDY $nn | A4nn | N | Z | Load location nn into Y-register |
| LDY #$nn | A0nn | N | Z | Load byte nn into Y-register |
| LDY $nnnn,X | BCnnnn | N | Z | Load location nnnn+X-register into Y-register |
| LDY $nn,X | B4nn | N | Z | Load byte from nn+X-register into Y-register |
| LSR A | 4A | N=0 | ZC | Logical shift right of accumulator |
| LSR $nnnn | 4Ennnn | N=0 | ZC | Logical shift right of location nnnn |
| LSR $nn | 46nn | N=0 | ZC | Logical shift right of location nn |
| LSR $nnnn,X | 5Ennnn | N=0 | ZC | Logical shift right of location nnnn+X-register |
| LSR $nn,X | 56nn | N=0 | ZC | Logical shift right of location nn+X-register |
| NOP | EA | | | Wait |
| ORA $nnnn | 0Dnnnn | N | Z | 'OR' location nnnn with accumulator |
| ORA $nn | 05nn | N | Z | 'OR' location nn with accumulator |
| ORA #$nn | 09nn | N | Z | 'OR' byte nn with accumulator |
| ORA $nnnn,X | 1Dnnnn | N | Z | 'OR' location nnnn+X-register with accumulator |

| | | | | | |
|---|---|---|---|---|---|
| ORA $*nnnn*,Y | 19nnnn | N | | Z | 'OR' location nnnn+Y-register with accumulator |
| ORA $*nn*,X | 15nn | N | | Z | 'OR' location nn+X-reg. with accum. |
| ORA ($*nn*,X) | 01nn | N | | Z | 'OR' byte from nn+X-register with accumulator |
| ORA ($*nn*),Y | 11nn | N | | Z | 'OR' byte from nn+Y-register with accumulator |
| PHA | 48 | | | | Push accumulator onto stack |
| PHP | 08 | | | | Push status word onto stack |
| PLA | 68 | N | | Z | Pop top off stack into accumulator |
| PLP | 28 | NV | BDI | ZC | Pop top off stack into status word |
| ROL A | 2A | N | | ZC | Rotate accumulator to left |
| ROL $*nnnn* | 2Ennnn | N | | ZC | Rotate location nnnn to left |
| ROL $*nn* | 26nn | N | | ZC | Rotate location nn to left |
| ROL $*nnnn*,X | 3Ennnn | N | | ZC | Rotate location nnnn+X-reg. to left |
| ROL $*nn*,X | 36nn | N | | ZC | Rotate location nn+X-reg. to left |
| ROR A | 6A | N | | ZC | Rotate accumulator to right |
| ROR $*nnnn* | 6Ennnn | N | | ZC | Rotate location nnnn to right |
| ROR $*nn* | 66nn | N | | ZC | Rotate location nn to right |
| ROR $*nnnn*,X | 7Ennnn | N | | ZC | Rotate location nnnn+X-reg. to right |
| ROR $*nn*,X | 76nn | N | | ZC | Rotate location nn+X-reg. to right |
| RTI | 40 | | | | Return from interrupt routine |
| RTS | 60 | | | | Return from subroutine |
| SBC $*nnnn* | EDnnnn | NV | | ZC | Subtract location nnnn from accum. |
| SBC $*nn* | E5nn | NV | | ZC | Subtract location nn from accumulator |
| SBC #$*nn* | E9nn | NV | | ZC | Subtract byte nn from accumulator |
| SBC $*nnnn*,X | FDnnnn | NV | | ZC | Subtract location nnnn+X-register from accumulator |
| SBC $*nnnn*,Y | F9nnnn | NV | | ZC | Subtract location nnnn+Y-register from accumulator |
| SBC $*nn*,X | F5nn | NV | | ZC | Subtract location nn+X-register from accumulator |
| SBC ($*nn*,X) | E1nn | NV | | ZC | Subtract byte from nn+X-register from accumulator |
| SBC ($*nn*),Y | F1nn | NV | | ZC | Subtract byte from nn+Y-register from accumulator |
| SEC | 38 | | | C=1 | Set C-flag |
| SED | F8 | | D=1 | | Set D-flag |
| SEI | 78 | | I=1 | | Set I-flag |
| STA $*nnnn* | 8Dnnnn | | | | Store accumulator in location nnnn |
| STA $*nn* | 85nn | | | | Store accumulator in location nn |
| STA $*nnnn*,X | 9Dnnnn | | | | Store accumulator in location nnnn+X-register |

| | | | | | |
|---|---|---|---|---|---|
| STA $*nnnn*,Y | 99nnnn | | | | Store accumulator in location nnnn+Y-register |
| STA $*nn*,X | 95nn | | | | Store accumulator in location nn+X-register |
| STA ($*nn*,X) | 81nn | | | | Store accumulator in location nn+X-register |
| STA ($*nn*),Y | 91nn | | | | Store accumulator in location nn+Y-register |
| STX $*nnnn* | 8Ennnn | | | | Store X-register in location nnnn |
| STX $*nn* | 86nn | | | | Store X-register in location nn |
| STX $*nn*,Y | 96nn | | | | Store X-register in location nn+Y-register |
| STY $*nnnn* | 8Cnnnn | | | | Store Y-register in location nnnn |
| STY $*nn* | 84nn | | | | Store Y-register in location nn |
| STY $*nn*,X | 94nn | | | | Store Y-register in location nn+X-register |
| TAX | AA | N | | Z | Copy accumulator into X-register |
| TAY | A8 | N | | Z | Copy accumulator into Y-register |
| TSX | BA | N | | Z | Copy status into X-register |
| TXA | 8A | N | | Z | Copy X-register into accumulator |
| TXS | 9A | N | V | B D I Z C | Copy X-register into status |
| TYA | 98 | N | | Z | Copy Y-register into accumulator |

# Appendix C:

## Machine language commands for the Z80

If you look at the opcode listing for the Z80 you will notice that there are many commands that only differ in the types of operands they use. These groups are described in this section. You don't have to learn all of the discrepancies, you can get a quick idea of what the groups are and all of the possibilities that they offer. Later, if you get into machine language programming you can use the opcode listing as a good reference.

ADC A,*x*
The operand x is added to the accumulator. The carry will also be set in the carry bit. The result is saved in the accumulator and the carry is saved in the carry bit. After this command all of the flags will reflect the contents of the accumulator.

ADC HL,*x*
This command functions in the same manner as the above only using 16-bits. The operand x is added to the 16-bit register pair HL. The result is saved in HL, and the flags are reflected as a result.

ADD A,*x*
The operand is added to the accumulator and the result is returned to the accumulator. The carry bit does not reflect the addition, however, all other flags will reflect the addition.

ADD HL,*x*
Just like the previous only 16-bit.

ADD IX,*x*
The operand is added to the index register, and the result is saved in the index register and the carry bit reflects the addition. The remaining flags, however, do not reflect the addition.

ADD IY,*x*
Just like the previous only with index register IY.

AND *x*
Logical AND of the operand with the accumulator, the result is saved in the accumulator. All flags reflect the operation and the carry bit is set to 0 (there is no carry with an AND).

BIT *n,x*
The nth bit of the operand x is tested. It is complemented and the result is placed in the Z-flag. If the bit is 1, then Z=0, and if the bit was 0 then Z=1. The flags S and P contain also incidentally contain the value.

CALL *nnnn*
Call the subroutine at the address nnnn (this is similar to a GOSUB).

CALL *condition,nnnn*
The subroutine in nnnn is called if a condition is met. The condition can be the test of any of the flags S,Z,P, and C. Depending upon the state of the condition the processor will branch to the subroutine or continue on to the next command.

CCF
Complement the carry-bit.

CP *x*
Compare the operand x with the accumulator. In other words, subtract and don't save the result, so that the accumulator will remain intact. If the operand is smaller than the accumulator, then S=1, if it is greater than or equal to the accumulator, then S=0. If both values are equal then Z=1, otherwise Z=0.

CPD
The HL register is a pointer to a location which is compared with the accumulator. If both values are equal then Z=1. If the accumulator is greater or the same then S=0, otherwise S=1. HL and BC are then decremented by 1. If the contents of BC=0, then the P-flag is set to 0, otherwise it remains set to 1. BC is used as a byte counter.

CPDR
Just like CPD, however it is repeated until a comparison reveals an equality (Z=1) or BC=0 (P-flag is set to 0).

CPI
Just like CPD, however, HL is incremented, not decremented.

CPIR
Just like CPDR, however, HL is incremented.

CPL
Invert all of the bits in the accumulator.

DAA
After an arithmetic operation (ADC, ADD, DEC, INC, NEG, SBC, SUB) the existing false BCD digit is placed in the accumulator, and the flags are corrected.

DEC *x*
The contents of the operand are decremented by 1 and re-saved in the operand. With the exception of the registers BC, DE, HL, IX, IY, and SP all the flags are set appropriately.

DI
After this command the Z80 ignores all maskable interrupts.

DJNZ *e*
The B register is decremented by 1. If B is not zero, then there is a relative jump to the address e.

EI
This turns the maskable interrupts back on.

EX *x,y*
The two operands are exchanged. X can also be used for (SP). In this case the second operand will be exchanged with the top of the stack.

EXX
The register BC, DE,and HL is exchanged with the second register.

HALT
The Z80 stops the program until an interrupt starts it up again.

IM *n*
Turn off the interrupt mode.

Mode 0: The next command is kept on the databus.
Mode 1: The Z80 jumps to location 0038 (hex) and starts off the interrupt routine.
Mode 2: Saves the lower half of the address whose upper half is saved in register I. Below this address is the pointer to the interrupt routine.

IN *x*, (C)
Register C has the port number where the byte is read into register x.

IN A, (*n*)
The accumulator is loaded from port n.

INC *x*
Just like DEC, however x is incremented.

IND
Register HL is the pointer to a location in memory where a byte from the port is saved. Register C has the number of the port. Outside of this HL and B are decremented. If B=0 then the Z-flag is set to 1. The flags (outside of the carry) reflect the command.

INDR
Just like IND except the command is repeated until B=0.

INI
Just like IND, except HL is incremented

INIR
Just like INDR, except HL is incremented.

JP *nnnn*
The program jumps to address nnnn (similar to GOTO).

JP *condition,nn*
Jumps when a condition is true(for more information see CALL).

JP *x*
Jumps to the address in the operand x (HL, IX, IY).

JR *n* and JR *condition,n*
Just like the JP commands, except the byte following the opcode is a relative address which is added to the program counter (PC).

LD *x,y*
This group belongs with most of the preceding commands, all of which follow one principle: The contents of the second operand will be transferred to the first. For example, when the contents of the accumulator are loaded into memory cell nnnn, or vice versa. The load command also applies to the 16-bit register: Should a 16-bit register be transferred in memory, the low byte will be placed in nnnn, and the high byte into nnnn+1. This format is always used in computer science (Note: Low then high!)

LDD
HL and DE are used as pointers into memory. The contents of the HL register address are also stored into the contents of the address stored in the DE register. After this HL, DE, and BC are decremented. If BC=0, then the P-flag is set to 0. In this case BC can be used as a counter.

LDDR
Same as LDD, except it is repeated until BC=0.

LDI
Just like LDD, except the registers HL and DE are incremented.

LDIR
Just like LDDR, except HL and DE are incremented.

NEG
The byte in the accumulator is multiplied by -1. The result is saved in the accumulator. P=1 if the contents of the accumulator was $80 before the command, and C=1 if the accumulator had a 0 prior.

NOP
This command doesn't do a thing.

OR *x*
This command OR's the contents of the accumulator with the operand. The result is found in the accumulator. After this the carry bit is set to 0, since there is no carry with a logical OR.

OTDR
HL is used as a pointer to a location whose contents are output to the port located in C. After this HL and B are decremented. This continues until B=0. Outside of the C and the Z flags (set to 1) all flags represent the operation.

OTIR
Just like OTDR except HL is incremented.

OUT (C),*x*
The operand x is output to a port which is located in C.

OUT (*n*),A
The accumulator is output to port n.

OUTD
HL is used as a pointer to a location whose contents are output to port C. Afterwards, HL and B are decremented. If B=0 then Z-flag is set to 1. All flags (outside of the carry flag) reflect the command.

OUTI
Just like OUTD, except HL is incremented.

POP *x*
The register pair in the operand is loaded with the contents of the top of the stack. AF stands for the accumulator and flags.

PUSH *x*
The register pair in the operand is pushed on to the top of the stack.

RES *n,x*
The nth bit of operand x is cleared.

RET and RET *condition*
This command is the return jump from a subroutine to the calling routine (similar to RETURN in BASIC). The condition can be true if a flag has a specific value.

RETI
Return jump from an interrupt routine (similar to RET).

RETN
Return jump from an NMI routine

RL *x* and RLA
The operand x is rotated to the left, where the carry bit is bit 0 and bit 7 is what the carry bit was. The RL command changes all flags.The RLA command (not RL A). This operates much like RL A, except that it only influences the carry bit.

RLC *x* and RLCA
The operand is rotated to the left, where bit-7 is moved into the carry bit. (see appendix). Except for RLCA all flags are changed.

RLD
This command is a BCD rotation, the left half of a memory location whose address is in HL, is put in the right side of the accumulator. The right half of the memory location is put in the left hand side, and the right side of the accumulator is put in the right half of the location. The flags S,Z, and P are influenced.

RR *x* and RRA
This command functions much like RL *x* and RLA, except the rotation is to the right.

RRC *x* and RRCA
Just like RLC *x* and RLCA, except the rotation is to the right.

RRD
Just like RLD, except for a right rotation. The lower accumulator nibble (a nibble is one half a byte) is switched into the upper nibble of a location in the HL register. The upper location nibble is put in the lower and the lower location nibble is put in the lower half of the accumulator.

RST *n*
Call to a subroutine at address n.

SBC A,*x*
The operand in x is subtracted from the accumulator, and the carry bit is changed. The result is left in the accumulator, and the new carry bit is saved. After the command all flags are representative of what happened.

SBC HL,*x*
This command functions like the last except for a 16-bit subtraction. The result is saved in the HL register pair.

SCF
Set the carry bit.

SET *n,x*
Set the nth bit of the operand x to 1.

SLA *x*
The operand is shifted to the left. Bit 0 is set to 0 and bit 7 moves into the carry bit.

SRA *x*
The operand is shifted to the right. Bit 7 remains the same and bit-0 is moved into the carry bit.

SRL *x*
The operand is shifted to the right, bit-7 is set to 0 and bit 0 is moved into the carry bit.

SUB *x*
The operand is subtracted from the accumulator.

XOR *x*
The operand is exclusive-or'd with the accumulator.

# Appendix D:

## Z80 opcodes

In the following table are all of the commands for the Z80 with their opcodes. The column for flags is a listing of all flags that could be changed by the command.

| Mnemonic | Opcode | Flags | Description |
|---|---|---|---|
| ADC A,A | 8F | S Z P C | Add carry and A into A |
| ADC A,B | 88 | S Z P C | ... into B |
| ADC A,C | 89 | S Z P C | ... into C |
| ADC A,D | 8A | S Z P C | ... into D |
| ADC A,E | 8B | S Z P C | ... into E |
| ADC A,H | 8C | S Z P C | ... into H |
| ADC A,L | 8D | S Z P C | ... into L |
| ADC A,n | CEnn | S Z P C | ... to the next Byte |
| ADC A,(HL) | 8E | S Z P C | ... to address in HL |
| ADC A,(IX+d) | DD8Edd | S Z P C | ... to address in IX+d |
| ADC A,(IY+d) | FD8Edd | S Z P C | ... to address in IY+d |
| ADC HL,BC | ED4A | S Z P C | Add Carry and HL into BC |
| ADC HL,DE | ED5A | S Z P C | ... into DE |
| ADC HL,HL | ED6A | S Z P C | ... into HL |
| ADC HL,SP | ED7A | S Z P C | ... into SP |
| ADD A,A | 87 | S Z P C | Add accum. into accum. |
| ADD A,B | 80 | S Z P C | ... into B |
| ADD A,C | 81 | S Z P C | ... into C |
| ADD A,D | 82 | S Z P C | ... into D |
| ADD A,E | 83 | S Z P C | ... into E |
| ADD A,H | 84 | S Z P C | ... into H |
| ADD A,L | 85 | S Z P C | ... into L |
| ADD A,n | C6nn | S Z P C | ... to the next Byte |
| ADD A,(HL) | 86 | S Z P C | ... to address in HL |
| ADD A,(IX+d) | DD86dd | S Z P C | ... to address in IX+d |

```
ADD A,(IY+d)  FD86dd   S Z P C     ... to address in
                                       IY+d
ADD HL,BC     09               C   Add HL into BC
ADD HL,DE     19               C     ... into DE
ADD HL,HL     29               C     ... into HL
ADD HL,SP     39               C     ... into SP
ADD IX,BC     DD09             C   Add IX into BC
ADD IX,DE     DD19             C     ... into DE
ADD IX,HL     DD29             C     ... into HL
ADD IX,SP     DD39             C     ... into SP
ADD IY,BC     FD09             C   Add IY into BC
ADD IY,DE     FD19             C     ... into DE
ADD IY,HL     FD29             C     ... into HL
ADD IY,SP     FD39             C     ... into SP
AND A         A7       S Z P C=0 'AND' accum.with
                                       accum.
AND B         A0       S Z P C=0   ... with B
AND C         A1       S Z P C=0   ... with C
AND D         A2       S Z P C=0   ... with D
AND E         A3       S Z P C=0   ... with E
AND H         A4       S Z P C=0   ... with H
AND L         A5       S Z P C=0   ... with L
AND n         E6nn     S Z P C=0   ... with the next
                                       Byte
AND (HL)      96       S Z P C=0   ... with address
                                       in HL
AND (IX+d)    DD96dd   S Z P C=0   ... with address
                                       in IX+d
AND (IY+d)    FD96dd   S Z P C=0   ... with address
                                       in IY+d
BIT 0,A       CB47       Z       Test Bit 0 of
                                       accum.
BIT 0,B       CB40       Z       Test Bit 0 of B
BIT 0,C       CB41       Z         ... of C
BIT 0,D       CB42       Z         ... of D
BIT 0,E       CB43       Z         ... of E
BIT 0,H       CB44       Z         ... of H
BIT 0,L       CB45       Z         ... of L
BIT 0,(HL)    CB46       Z         ... of the
                                       address in HL
BIT 0,(IX+d)  DDCBdd46   Z         ... of the
                                       address IX+d
BIT 0,(IY+d)  FDCBdd46   Z         ... of the
                                       address IY+d
BIT 1,A       CB4F       Z       Test Bit 1 of
                                       accum.
BIT 1,B       CB48       Z       Test Bit 1 of B
```

| | | | |
|---|---|---|---|
| BIT 1,C | CB49 | Z | ... of C |
| BIT 1,D | CB4A | Z | ... of D |
| BIT 1,E | CB4B | Z | ... of E |
| BIT 1,H | CB4C | Z | ... of H |
| BIT 1,L | CB4D | Z | ... of L |
| BIT 1,(HL) | CB4E | Z | ... of the address in HL |
| BIT 1,(IX+d) | DDCBdd4E | Z | ... of the address IX+d |
| BIT 1,(IY+d) | FDCBdd4E | Z | ... of the address IY+d |
| BIT 2,A | CB57 | Z | Test Bit 2 of accum. |
| BIT 2,B | CB50 | Z | Test Bit 2 of B |
| BIT 2,C | CB51 | Z | ... of C |
| BIT 2,D | CB52 | Z | ... of D |
| BIT 2,E | CB53 | Z | ... of E |
| BIT 2,H | CB54 | Z | ... of H |
| BIT 2,L | CB55 | Z | ... of L |
| BIT 2,(HL) | CB56 | Z | ... of the address in HL |
| BIT 2,(IX+d) | DDCBdd56 | Z | ... of the address IX+d |
| BIT 2,(IY+d) | FDCBdd56 | Z | ... of the address IY+d |
| BIT 3,A | CB5F | Z | Test Bit 3 of accum. |
| BIT 3,B | CB58 | Z | Test Bit 3 of B |
| BIT 3,C | CB59 | Z | ... of C |
| BIT 3,D | CB5A | Z | ... of D |
| BIT 3,E | CB5B | Z | ... of E |
| BIT 3,H | CB5C | Z | ... of H |
| BIT 3,L | CB5D | Z | ... of L |
| BIT 3,(HL) | CB5E | Z | ... of the address in HL |
| BIT 3,(IX+d) | DDCBdd5E | Z | ... of the address IX+d |
| BIT 3,(IY+d) | FDCBdd5E | Z | ... of the address IY+d |
| BIT 4,A | CB67 | Z | Test Bit 4 of accum. |
| BIT 4,B | CB60 | Z | Test Bit 4 of B |
| BIT 4,C | CB61 | Z | ... of C |
| BIT 4,D | CB62 | Z | ... of D |
| BIT 4,E | CB63 | Z | ... of E |
| BIT 4,H | CB64 | Z | ... of H |
| BIT 4,L | CB65 | Z | ... of L |

| | | | |
|---|---|---|---|
| BIT 4,(HL) | CB66 | Z | ... of the address in HL |
| BIT 4,(IX+d) | DDCBdd66 | Z | ... of the address IX+d |
| BIT 4,(IY+d) | FDCBdd66 | Z | ... of the address IY+d |
| BIT 5,A | CB6F | Z | Test Bit 5 of accum. |
| BIT 5,B | CB68 | Z | Test Bit 5 of B |
| BIT 5,C | CB69 | Z | ... of C |
| BIT 5,D | CB6A | Z | ... of D |
| BIT 5,E | CB6B | Z | ... of E |
| BIT 5,H | CB6C | Z | ... of H |
| BIT 5,L | CB6D | Z | ... of L |
| BIT 5,(HL) | CB6E | Z | ... of the address in HL |
| BIT 5,(IX+d) | DDCBdd6E | Z | ... of the address IX+d |
| BIT 5,(IY+d) | FDCBdd6E | Z | ... of the address IY+d |
| BIT 6,A | CB77 | Z | Test Bit 6 of accum. |
| BIT 6,B | CB70 | Z | Test Bit 6 of B |
| BIT 6,C | CB71 | Z | ... of C |
| BIT 6,D | CB72 | Z | ... of D |
| BIT 6,E | CB73 | Z | ... of E |
| BIT 6,H | CB74 | Z | ... of H |
| BIT 6,L | CB75 | Z | ... of L |
| BIT 6,(HL) | CB76 | Z | ... of the address in HL |
| BIT 6,(IX+d) | DDCBdd76 | Z | ... of the address IX+d |
| BIT 6,(IY+d) | FDCBdd76 | Z | ... of the address IY+d |
| BIT 7,A | CB7F | Z | Test Bit 7 of accum. |
| BIT 7,B | CB78 | Z | Test Bit 7 of B |
| BIT 7,C | CB79 | Z | ... of C |
| BIT 7,D | CB7A | Z | ... of D |
| BIT 7,E | CB7B | Z | ... of E |
| BIT 7,H | CB7C | Z | ... of H |
| BIT 7,L | CB7D | Z | ... of L |
| BIT 7,(HL) | CB7E | Z | ... of the address in HL |
| BIT 7,(IX+d) | DDCBdd7E | Z | ... of the address IX+d |

| | | | |
|---|---|---|---|
| BIT 7,(IY+d) | FDCBdd7E | Z | ... of the address IY+d |
| CALL nn | CDnnnn | | Call subroutine at nnnn |
| CALL C,nn | DCnnnn | | ... if Carry=1 |
| CALL M,nn | FCnnnn | | ... if S=1 (minus) |
| CALL NC,nn | D4nnnn | | ... if Carry=0 |
| CALL NZ,nn | C4nnnn | | ... if Z=0 |
| CALL P,nn | F4nnnn | | ... if S=0 (plus) |
| CALL PE,nn | ECnnnn | | ... if P=1 |
| CALL PO,nn | E4nnnn | | ... if P=0 |
| CALL Z,nn | CCnnnn | | ... if Z=1 |
| CCF | 3F | C | Complement Carry Flag |
| CP A | BF | S Z P C | Compare A with A |
| CP B | B8 | S Z P C | ... with B |
| CP C | B9 | S Z P C | ... with C |
| CP D | BA | S Z P C | ... with D |
| CP E | BB | S Z P C | ... with E |
| CP H | BC | S Z P C | ... with H |
| CP L | BD | S Z P C | ... with L |
| CP n | FEnn | S Z P C | ... with the next Byte |
| CP (HL) | BE | S Z P C | ... with address in HL |
| CP (IX+d) | DDBEdd | S Z P C | ... with address IX+d |
| CP (IY+d) | FDBEdd | S Z P C | ... with address IY+d |
| CPD | EDA9 | S Z P | Compare & decrement |
| CPDR | EDB9 | S Z P | Repeat compare & decrement |
| CPI | EDA1 | S Z P | Compare & increment |
| CPIR | EDB1 | S Z P | Repeat compare & increment |
| CPL | 2F | | Complement accum. |
| DAA | 27 | S Z P C | Adapt accum. for BCD |
| DEC A | 3D | S Z P | Decrement A |
| DEC B | 05 | S Z P | ... B |
| DEC BC | 0B | | ... BC |
| DEC C | 0D | S Z P | ... C |
| DEC D | 15 | S Z P | ... D |
| DEC DE | 1B | | ... DE |
| DEC E | 1D | S Z P | ... E |
| DEC H | 25 | S Z P | ... H |

| | | | |
|---|---|---|---|
| DEC HL | 2B | | ... HL |
| DEC IX | DD2B | | ... IX |
| DEC IY | FD2B | | ... IY |
| DEC L | 2D | S Z P | ... L |
| DEC SP | 3B | | ... SP |
| DEC (HL) | 35 | S Z P | ... address in HL |
| DEC (IX+d) | DD35dd | S Z P | ... address IX+d |
| DEC (IY+d) | FD35dd | S Z P | ... address IY+d |
| DI | F3 | | Save Interrupt |
| DJNZ e | 10ee | | Decrement & jump by ()0 |
| EI | FB | | Turn off interrupt |
| EX AF,AF" | 08 | | Exchange A/Flags with 2nd reg. |
| EX DE,HL | EB | | Exchange DE and HL |
| EX (SP),HL | E3 | | Exchange HL with stack |
| EX (SP),IX | DDE3 | | ... IX |
| EX (SP),IY | FDE3 | | ... IY |
| EXX | D9 | | ... BC,DE,HL with 2nd reg. |
| HALT | 76 | | Stop Z80 with Interrupt |
| IM 0 | ED46 | | Interrupt mode 0 |
| IM 1 | ED56 | | Interrupt mode 1 |
| IM 2 | ED5E | | Interrupt mode 2 |
| IN A,(C) | ED78 | S Z P | Read Byte from Port C in A |
| IN A,(n) | DBnn | | ... from Port n |
| IN B,(C) | ED40 | S Z P | ... in B |
| IN C,(C) | ED48 | S Z P | ... in C |
| IN D,(C) | ED50 | S Z P | ... in D |
| IN E,(C) | ED58 | S Z P | ... in E |
| IN H,(C) | ED60 | S Z P | ... in H |
| IN L,(C) | ED68 | S Z P | ... in L |
| INC A | 3C | S Z P | Increment accum. |
| INC B | 04 | S Z P | ... B |
| INC BC | 03 | | ... BC |
| INC C | 0C | S Z P | ... C |
| INC D | 14 | S Z P | ... D |
| INC DE | 13 | | ... DE |
| INC E | 1C | S Z P | ... E |
| INC H | 24 | S Z P | ... H |
| INC HL | 23 | | ... HL |
| INC IX | DD23 | | ... IX |
| INC IY | FD23 | | ... IY |
| INC L | 2C | S Z P | ... L |

| | | | |
|---|---|---|---|
| INC SP | 33 | | ... SP |
| INC (HL) | 34 | S Z P | ... address in HL |
| INC (IX+d) | DD34dd | S Z P | ... address IX+d |
| INC (IY+d) | FD34dd | S Z P | ... address IY+d |
| IND | EDAA | Z | Collect with decrement |
| INDR | EDBA | Z=1 | Repeat collect with decrement |
| INI | EDA2 | Z | Collect with increment |
| INIR | EDB2 | Z=1 | Repeat collect with increment |
| JP C,nn | DAnnnn | | Jump to nn, if C=1 |
| JP M,nn | FAnnnn | | ... to nn, if S=1 |
| JP NC,nn | D2nnnn | | ... to nn, if C=0 |
| JP nn | C3nnnn | | ... to nn |
| JP NZ,nn | C2nnnn | | ... to nn, if Z=0 |
| JP P,nn | F2nnnn | | ... to nn, if S=0 |
| JP PE,nn | EAnnnn | | ... to nn, if P=1 |
| JP PO,nn | E2nnnn | | ... to nn, if P=0 |
| JP Z,nn | CAnnnn | | ... to nn, if Z=1 |
| JP (HL) | E9 | | ... to address in HL |
| JP (IX) | DDE9 | | ... to address in IX |
| JP (IY) | FDE9 | | ... to address in IY |
| JR C,e | 38ee | | Relative jump, if C=1 |
| JR e | 18ee | | ... to PC+e |
| JR NC,e | 30ee | | ... if C=0 |
| JR NZ,e | 20ee | | ... if Z=0 |
| JR Z,e | 28ee | | ... if Z=1 |
| LD A,A | 7F | | Load accum. with accum. |
| LD A,B | 78 | | ... with B |
| LD A,C | 79 | | ... with C |
| LD A,D | 7A | | ... with D |
| LD A,E | 7B | | ... with E |
| LD A,H | 7C | | ... with H |
| LD A,I | ED57 | S Z P | ... with I |
| LD A,L | 7D | | ... with L |
| LD A,n | 3Enn | | ... with the next Byte |
| LD A,R | ED5F | S Z P | ... with R |
| LD A,(BC) | 0A | | ... out of address in BC |

| | | |
|---|---|---|
| LD A,(DE) | 1A | ... out of address in DE |
| LD A,(HL) | 7E | ... out of address in HL |
| LD A,(IX+d) | DD7Edd | ... out of address IX+d |
| LD A,(IY+d) | FD7Edd | ... out of address IY+d |
| LD A,(nn) | 3Annnn | ... out of |
| LD B,A | 47 | Load B with accum. |
| LD B,B | 40 | ... with B |
| LD B,C | 41 | ... with C |
| LD B,D | 42 | ... with D |
| LD B,E | 43 | ... with E |
| LD B,H | 44 | ... with H |
| LD B,L | 45 | ... with L |
| LD B,n | 06nn | ... with the next Byte |
| LD B,(HL) | 46 | ... out of address in HL |
| LD B,(IX+d) | DD46dd | ... out of address IX+d |
| LD B,(IY+d) | FD46dd | ... out of address IY+d |
| LD BC,nn | 01nnnn | Load BC with the next Bytes |
| LD BC,(nn) | ED4Bnnnn | ... with nnnn and nnnn+1 |
| LD C,A | 4F | Load C with accum. |
| LD C,B | 48 | ... with B |
| LD C,C | 49 | ... with C |
| LD C,D | 4A | ... with D |
| LD C,E | 4B | ... with E |
| LD C,H | 4C | ... with H |
| LD C,L | 4D | ... with L |
| LD C,n | 0Enn | ... with the next Byte |
| LD C,(HL) | 4E | ... out of address in HL |
| LD C,(IX+d) | DD4Edd | ... out of address IX+d |
| LD C,(IY+d) | FD4Edd | ... out of address IY+d |
| LD D,A | 57 | Load D with accum. |
| LD D,B | 50 | ... with B |
| LD D,C | 51 | ... with C |
| LD D,D | 52 | ... with D |

| | | |
|---|---|---|
| LD D,E | 53 | ... with E |
| LD D,H | 54 | ... with H |
| LD D,L | 55 | ... with L |
| LD D,n | 16nn | ... with the next Byte |
| LD D,(HL) | 56 | ... out of address in HL |
| LD D,(IX+d) | DD56dd | ... out of address IX+d |
| LD D,(IY+d) | FD56dd | ... out of address IY+d |
| LD DE,nn | 11nnnn | Load DE with the next Bytes |
| LD DE,(nn) | ED5Bnnnn | ... with nnnn and nnnn+1 |
| LD E,A | 5F | Load E with accum. |
| LD E,B | 58 | ... with B |
| LD E,C | 59 | ... with C |
| LD E,D | 5A | ... with D |
| LD E,E | 5B | ... with E |
| LD E,H | 5C | ... with H |
| LD E,L | 5D | ... with L |
| LD E,n | 1Enn | ... with the next Byte |
| LD E,(HL) | 5E | ... out of address in HL |
| LD E,(IX+d) | DD5Edd | ... out of address IX+d |
| LD E,(IY+d) | FD5Edd | ... out of address IY+d |
| LD H,A | 67 | Load H with accum. |
| LD H,B | 60 | ... with B |
| LD H,C | 61 | ... with C |
| LD H,D | 62 | ... with D |
| LD H,E | 63 | ... with E |
| LD H,H | 64 | ... with H |
| LD H,L | 65 | ... with L |
| LD H,n | 26nn | ... with the next Byte |
| LD H,(HL) | 66 | ... out of address in HL |
| LD H,(IX+d) | DD66dd | ... out of address IX+d |
| LD H,(IY+d) | FD66dd | ... out of address IY+d |
| LD HL,nn | 21nnnn | Load HL with the next Bytes |

| | | | |
|---|---|---|---|
| LD (HL),n | 36nn | | ... with the next Byte |
| LD (IX+d),A | DD77dd | | Load address IX+d with accum. |
| LD (IX+d),B | DD70dd | | ... with B |
| LD (IX+d),C | DD71dd | | ... with C |
| LD (IX+d),D | DD72dd | | ... with D |
| LD (IX+d),E | DD73dd | | ... with E |
| LD (IX+d),H | DD74dd | | ... with H |
| LD (IX+d),L | DD75dd | | ... with L |
| LD (IX+d),n | DD36ddnn | | ... with the next Byte |
| LD (IY+d),A | FD77dd | | Load address IY+d with accum. |
| LD (IY+d),B | FD70dd | | ... with B |
| LD (IY+d),C | FD71dd | | ... with C |
| LD (IY+d),D | FD72dd | | ... with D |
| LD (IY+d),E | FD73dd | | ... with E |
| LD (IY+d),H | FD74dd | | ... with H |
| LD (IY+d),L | FD75dd | | ... with L |
| LD (IY+d),n | FD36ddnn | | ... with the next Byte |
| LD (nn),A | 32nnnn | | Load location nnnn with accum. |
| LD (nn),BC | ED43nnnn | | ... with C and nnnn+1 with B |
| LD (nn),DE | ED53nnnn | | ... with E and nnnn+1 with D |
| LD (nn),HL | 22nnnn | | ... with L and nnnn+1 with H |
| LD (nn),IX | DD22nnnn | | ... and nnnn+1 with IX |
| LD (nn),IY | FD22nnnn | | ... and nnnn+1 with IY |
| LD (nn),SP | ED73nnnn | | ... and nnnn+1 with SP |
| LDD | EDA8 | P | Load and decrement |
| LDDR | EDB8 | P=0 | Repeat load and decrement |
| LDI | EDA0 | P | Load and increment |
| LDIR | EDB0 | P=0 | Repeat load and increment |
| NEG | ED44 | S Z P C | Negate accum. |
| NOP | 00 | | Wait (No operation) |
| OR A | B7 | S Z P C | 'OR' accum. with accum. |
| OR B | B0 | S Z P C | ... with B |

| | | |
|---|---|---|
| LD HL,(nn) | 2Annnn | ... out of nnnn and nnnn+1 |
| LD I,A | ED47 | Load I with accum. |
| LD IX,nn | DD21nnnn | Load IX with the next Bytes |
| LD IX,(nn) | DD2Annnn | ... out of nnnn and nnnn+1 |
| LD IY,nn | FD21nnnn | Load IY with the next Bytes |
| LD IY,(nn) | FD2Annnn | ... out of nnnn and nnnn+1 |
| LD L,A | 6F | Load L with accum. |
| LD L,B | 68 | ... with B |
| LD L,C | 69 | ... with C |
| LD L,D | 6A | ... with D |
| LD L,E | 6B | ... with E |
| LD L,H | 6C | ... with H |
| LD L,L | 6D | ... with L |
| LD L,n | 2Enn | ... with the next Byte |
| LD L,(HL) | 6E | ... out of address in HL |
| LD L,(IX+d) | DD6Edd | ... out of address IX+d |
| LD L,(IY+d) | FD6Edd | ... out of address IY+d |
| LD R,A | ED4F | Load R with accum. |
| LD SP,HL | F9 | Load SP with HL |
| LD SP,IX | DDF9 | ... with IX |
| LD SP,IY | FDF9 | ... with IY |
| LD SP,nn | 31nnnn | ... with the next Bytes |
| LD SP,(nn) | ED7Bnnnn | ... out of nnnn and nnnn+1 |
| LD (BC),A | 02 | Load address out of BC with accum |
| LD (DE),A | 12 | Load address out of DE with accum |
| LD (HL),A | 77 | Load address out of HL with accum |
| LD (HL),B | 70 | ... with B |
| LD (HL),C | 71 | ... with C |
| LD (HL),D | 72 | ... with D |
| LD (HL),E | 73 | ... with E |
| LD (HL),H | 74 | ... with H |
| LD (HL),L | 75 | ... with L |

| | | | |
|---|---|---|---|
| OR C | B1 | S Z P C | ... with C |
| OR D | B2 | S Z P C | ... with D |
| OR E | B3 | S Z P C | ... with E |
| OR H | B4 | S Z P C | ... with H |
| OR L | B5 | S Z P C | ... with L |
| OR n | F6nn | S Z P C | ... with the next Byte |
| OR (HL) | B6 | S Z P C | ... with address in HL |
| OR (IX+d) | DDB6dd | S Z P C | ... with address IX+d |
| OR (IY+d) | FDB6dd | S Z P C | ... with address IY+d |
| OTDR | EDBB | S Z P | Repeat output and decrement |
| OTIR | EDB3 | S Z P | Repeat output and increment |
| OUT (C),A | ED79 | | Output accum. to Port C |
| OUT (C),B | ED41 | | ... B to Port C |
| OUT (C),C | ED49 | | ... C to Port C |
| OUT (C),D | ED51 | | ... D to Port C |
| OUT (C),E | ED59 | | ... E to Port C |
| OUT (C),H | ED61 | | ... H to Port C |
| OUT (C),L | ED69 | | ... L to Port C |
| OUT (n),A | D3nn | | Output accum. to Port n |
| OUTD | EDAB | S Z P | Output and decrement |
| OUTI | EDA3 | S Z P | Output and increment |
| POP AF | F1 | | Get accum. & Flags from stack |
| POP BC | C1 | | Get BC from stack |
| POP DE | D1 | | Get DE from stack |
| POP HL | E1 | | Get HL from stack |
| POP IX | DDE1 | | Get IX from stack |
| POP IY | FDE1 | | Get IY from stack |
| PUSH AF | F5 | | Push accum. & Flags on stack |
| PUSH BC | C5 | | Push BC onto stack |
| PUSH DE | D5 | | Push DE onto stack |
| PUSH HL | E5 | | Push HL onto stack |
| PUSH IX | DDE5 | | Push IX onto stack |
| PUSH IY | FDE5 | | Push IY onto stack |
| RES 0,A | CB87 | | Clear Bit 0 of accum. |

| | | |
|---|---|---|
| RES 0,B | CB80 | Clear Bit 0 of B |
| RES 0,C | CB81 | ... of C |
| RES 0,D | CB82 | ... of D |
| RES 0,E | CB83 | ... of E |
| RES 0,H | CB84 | ... of H |
| RES 0,L | CB85 | ... of L |
| RES 0,(HL) | CB86 | ... of the address in HL |
| RES 0,(IX+d) | DDCBdd86 | ... of the address IX+d |
| RES 0,(IY+d) | FDCBdd86 | ... of the address IY+d |
| RES 1,A | CB8F | Clear Bit 1 of accum. |
| RES 1,B | CB88 | Clear Bit 1 of B |
| RES 1,C | CB89 | ... of C |
| RES 1,D | CB8A | ... of D |
| RES 1,E | CB8B | ... of E |
| RES 1,H | CB8C | ... of H |
| RES 1,L | CB8D | ... of L |
| RES 1,(HL) | CB8E | ... of the address in HL |
| RES 1,(IX+d) | DDCBdd8E | ... of the address IX+d |
| RES 1,(IY+d) | FDCBdd8E | ... of the address IY+d |
| RES 2,A | CB97 | Clear Bit 2 of accum. |
| RES 2,B | CB90 | Clear Bit 2 of B |
| RES 2,C | CB91 | ... of C |
| RES 2,D | CB92 | ... of D |
| RES 2,E | CB93 | ... of E |
| RES 2,H | CB94 | ... of H |
| RES 2,L | CB95 | ... of L |
| RES 2,(HL) | CB96 | ... of the address in HL |
| RES 2,(IX+d) | DDCBdd96 | ... of the address IX+d |
| RES 2,(IY+d) | FDCBdd96 | ... of the address IY+d |
| RES 3,A | CB9F | Clear Bit 3 of accum. |
| RES 3,B | CB98 | Clear Bit 3 of B |
| RES 3,C | CB99 | ... of C |
| RES 3,D | CB9A | ... of D |
| RES 3,E | CB9B | ... of E |
| RES 3,H | CB9C | ... of H |

| | | |
|---|---|---|
| RES 3,L | CB9D | ... of L |
| RES 3,(HL) | CB9E | ... of the address in HL |
| RES 3,(IX+d) | DDCBdd9E | ... of the address IX+d |
| RES 3,(IY+d) | FDCBdd9E | ... of the address IY+d |
| RES 4,A | CBA7 | Clear Bit 4 of accum. |
| RES 4,B | CBA0 | Clear Bit 4 of B |
| RES 4,C | CBA1 | ... of C |
| RES 4,D | CBA2 | ... of D |
| RES 4,E | CBA3 | ... of E |
| RES 4,H | CBA4 | ... of H |
| RES 4,L | CBA5 | ... of L |
| RES 4,(HL) | CBA6 | ... of the address in HL |
| RES 4,(IX+d) | DDCBddA6 | ... of the address IX+d |
| RES 4,(IY+d) | FDCBddA6 | ... of the address IY+d |
| RES 5,A | CBAF | Clear Bit 5 of accum. |
| RES 5,B | CBA8 | Clear Bit 5 of B |
| RES 5,C | CBA9 | ... of C |
| RES 5,D | CBAA | ... of D |
| RES 5,E | CBAB | ... of E |
| RES 5,H | CBAC | ... of H |
| RES 5,L | CBAD | ... of L |
| RES 5,(HL) | CBAE | ... of the address in HL |
| RES 5,(IX+d) | DDCBddAE | ... of the address IX+d |
| RES 5,(IY+d) | FDCBddAE | ... of the address IY+d |
| RES 6,A | CBB7 | Clear Bit 6 of accum. |
| RES 6,B | CBB0 | Clear Bit 6 of B |
| RES 6,C | CBB1 | ... of C |
| RES 6,D | CBB2 | ... of D |
| RES 6,E | CBB3 | ... of E |
| RES 6,H | CBB4 | ... of H |
| RES 6,L | CBB5 | ... of L |
| RES 6,(HL) | CBB6 | ... of the address in HL |
| RES 6,(IX+d) | DDCBddB6 | ... of the address IX+d |

| | | | |
|---|---|---|---|
| RES 6,(IY+d) | FDCBddB6 | | ... of the address IY+d |
| RES 7,A | CBBF | | Clear Bit 7 of accum. |
| RES 7,B | CBB8 | | Clear Bit 7 of B |
| RES 7,C | CBB9 | | ... of C |
| RES 7,D | CBBA | | ... of D |
| RES 7,E | CBBB | | ... of E |
| RES 7,H | CBBC | | ... of H |
| RES 7,L | CBBD | | ... of L |
| RES 7,(HL) | CBBE | | ... of the address in HL |
| RES 7,(IX+d) | DDCBddBE | | ... of the address IX+d |
| RES 7,(IY+d) | FDCBddBE | | ... of the address IY+d |
| RET | C9 | | Return from subroutine |
| RET C | D8 | | Return, if C=1 |
| RET M | F8 | | ... if S=1 |
| RET NC | D0 | | ... if C=0 |
| RET NZ | C0 | | ... if Z=0 |
| RET P | F0 | | ... if S=0 |
| RET PE | E8 | | ... if P=1 |
| RET PO | E0 | | ... if P=0 |
| RET Z | C8 | | ... if Z=1 |
| RETI | ED4D | | Return from Interrupt |
| RETN | ED45 | | ... from NMI-Routine |
| RL A | CB17 | S Z P C | Left rotation of Carry & A |
| RL B | CB10 | S Z P C | ... and B |
| RL C | CB11 | S Z P C | ... and C |
| RL D | CB12 | S Z P C | ... and D |
| RL E | CB13 | S Z P C | ... and E |
| RL H | CB14 | S Z P C | ... and H |
| RL L | CB15 | S Z P C | ... and L |
| RL (HL) | CB16 | S Z P C | ... and address in HL |
| RL (IX+d) | DDCBdd16 | S Z P C | ... and address IX+d |
| RL (IY+d) | FDCBdd16 | S Z P C | ... and address IY+d |
| RLA | 17 | C | ... and accum. |
| RLC A | CB07 | S Z P C | Left rotation of accum. |

| | | | |
|---|---|---|---|
| RLC B | CB00 | S Z P C | ... of B |
| RLC C | CB01 | S Z P C | ... of C |
| RLC D | CB02 | S Z P C | ... of D |
| RLC E | CB03 | S Z P C | ... of E |
| RLC H | CB04 | S Z P C | ... of H |
| RLC L | CB05 | S Z P C | ... of L |
| RLC (HL) | CB06 | S Z P C | ... of the address in HL |
| RLC (IX+d) | DDCBdd06 | S Z P C | ... of the address IX+d |
| RLC (IY+d) | FDCBdd06 | S Z P C | ... of the address IY+d |
| RLCA | 07 | C | ... of accum. |
| RLD | ED6F | S Z P | Decimal rotation left |
| RR A | CB1F | S Z P C | Right rotation of carry & accum |
| RR B | CB18 | S Z P C | ... and B |
| RR C | CB19 | S Z P C | ... and C |
| RR D | CB1A | S Z P C | ... and D |
| RR E | CB1B | S Z P C | ... and E |
| RR H | CB1C | S Z P C | ... and H |
| RR L | CB1D | S Z P C | ... and L |
| RR (HL) | CB1E | S Z P C | ... and address in HL |
| RR (IX+d) | DDCBdd1E | S Z P C | ... and address in IX+d |
| RR (IY+d) | FDCBdd1E | S Z P C | ... and address in IY+d |
| RRA | 1F | C | ... and accum. |
| RRC A | CB0F | S Z P C | Right rotation of accum. |
| RRC B | CB08 | S Z P C | ... of B |
| RRC C | CB09 | S Z P C | ... of C |
| RRC D | CB0A | S Z P C | ... of D |
| RRC E | CB0B | S Z P C | ... of E |
| RRC H | CB0C | S Z P C | ... of H |
| RRC L | CB0D | S Z P C | ... of L |
| RRC (HL) | CB0E | S Z P C | ... of the address in HL |
| RRC (IX+d) | DDCBdd0E | S Z P C | ... of the address IX+d |
| RRC (IY+d) | FDCBdd0E | S Z P C | ... of the address IY+d |
| RRCA | 0F | C | ... of accum. |
| RRD | ED67 | S Z P | Decimal rotation right |

| | | | |
|---|---|---|---|
| RST 00 | C7 | | Call routine in 00 |
| RST 08 | CF | | ... in 08 |
| RST 10 | D7 | | ... in 10 |
| RST 18 | DF | | ... in 18 |
| RST 20 | E7 | | ... in 20 |
| RST 28 | EF | | ... in 28 |
| RST 30 | F7 | | ... in 30 |
| RST 38 | FF | | ... in 38 |
| SBC A,A | 9F | S Z P C | Subtract Carry & A from A |
| SBC A,B | 98 | S Z P C | ... B from accum. |
| SBC A,C | 99 | S Z P C | ... C from accum. |
| SBC A,D | 9A | S Z P C | ... D from accum. |
| SBC A,E | 9B | S Z P C | ... E from accum. |
| SBC A,H | 9C | S Z P C | ... H from accum. |
| SBC A,L | 9D | S Z P C | ... L from accum. |
| SBC A,n | DEnn | S Z P C | ... the next Byte from A |
| SBC A,(HL) | 9E | S Z P C | ... address in HL from A |
| SBC A,(IX+d) | DD9Edd | S Z P C | ... address IX+d from accum |
| SBC A,(IY+d) | FD9Edd | S Z P C | ... address IY+d from accum. |
| SBC HL,BC | ED42 | S Z P C | Subtract C & BC from HL |
| SBC HL,DE | ED52 | S Z P C | ... DE from HL |
| SBC HL,HL | ED62 | S Z P C | ... HL from HL |
| SBC HL,SP | ED72 | S Z P C | ... SP from HL |
| SCF | 37 | C=1 | Set Carry-Bit to 1 |

# Appendix E:

## Memory map in the C-128 mode

| | |
|---|---|
| 0 | Data control register for 8502 |
| 1 | Data register for 8502 |
| 2 | Bank for the monitor |
| 3 | PC highbyte for monitor |
| 4 | PC lowbyte for monitor |
| 5 | Status byte for monitor |
| 6 | Accumulator for monitor |
| 7 | X-register for monitor |
| 8 | Z-register for monitor |
| 9 | Stack pointer for monitor |
| 10 | Leading character flag |
| 11 | Monitor column after last TAB |
| 12 | 0=LOAD, 1=VERIFY |
| 13 | Input buffer pointer |
| 14 | Flag for the DIM |
| 15 | Variable type 0=numeric, 255=string |
| 16 | Variable type 0=real, 1=integer |
| 17 | Flag for list, data, garbage collection |
| 18 | Pointer for FN, variable type for FOR |
| 19 | Input flag 0=INPUT, 64=GET, 152=READ |
| 20 | TAN / comparison flag |
| 21 | Status I/O device, flag for INPUT |
| 22/23 | Line number of 2-byte address |
| 24 | Pointer to string stack |
| 25/26 | Address for temporary string |
| 27-35 | Stack for short strings |
| 36-39 | Help pointer |
| 40-44 | Real result of multiplication |
| 45/46 | Pointer to BASIC program start |
| 47/48 | Pointer to BASIC variable start |
| 49/50 | Pointer to start of BASIC fields |
| 51/52 | Pointer to end of BASIC field+1 |
| 53/54 | Point to start of BASIC strings |
| 55/56 | Help pointer for strings |
| 57/58 | Pointer to end of BASIC strings |
| 59/60 | BASIC line number |
| 61/62 | BASIC program number |
| 63/64 | Pointer for USING search pointer |
| 65/66 | Status of DATA line |
| 67/68 | Pointer to next DATA |
| 69/70 | Vector fo INPUT routine |

| | |
|---|---|
| 71/72 | Variable name |
| 73/74 | Pointer to last variable |
| 75/76 | AND mask, pointer to LIST for FOR |
| 77/78 | Memory for BASIC program pointer |
| 79 | Mask for comparison |
| 80/81 | Pointer fo FN and garbage collection |
| 82-84 | Pointer for string comparison |
| 85 | Flag for HELP and LIST |
| 86/87 | USR vector |
| 88 | Oldov |
| 89 | Help pointer |
| 90/91 | Repeat carry pointer, DIM pointer |
| 92/93 | Carry pointer |
| 94 | Help pointer |
| 95/96 | Register for number exchange |
| 97 | Pointer of decimal point |
| 98 | Exponent |

# Appendix F:

## Memory map in the C-64 mode

| | |
|---|---|
| 0 | Processor port data control register |
| 1 | Processor port data register |
| 2 | Unused |
| 3/4 | Vector for floating point to fixed point conversion |
| 5/6 | Vector for fixed point to floating point conversion |
| 7 | Search pointer |
| 8 | Flag for special pointer |
| 9 | TAB column |
| 10 | 0=LOAD, 1=VERIFY last command |
| 11 | Input buffer pointer |
| 12 | DIM flag |
| 13 | Variable type: FF=string, 00=num. |
| 14 | Variable type: 80=integer, 00=floating point |
| 15 | Special pointer for LIST |
| 16 | Flag for FN's |
| 17 | Input: 00=INPUT, 40=GET, 98=READ |
| 18 | Pointer for ARCTAN/last compare |
| 19 | Topical file |
| 20/21 | Integer number, (ex:addr, FRE(0)) |
| 22 | Vector for string stack |
| 23/24 | Pointer to last string |
| 25-33 | String stack |
| 34-37 | Diverse pointer |
| 38-42 | Arithmetic register |
| 43/44 | Pointer to BASIC program start |
| 45/46 | Pointer to variable start |
| 47/48 | Pointer to field start |
| 49/50 | Pointer to end of field |
| 51/52 | Pointer to string start |
| 53/54 | String help pointer |
| 55/56 | Pointer to memory border |
| 57/58 | Immediate BASIC line number |
| 59/60 | Previous BASIC line number |
| 61/62 | Pointer to next command for CONT |
| 63/64 | Topical DATA line |
| 65/66 | Pointer to next data element |
| 67/68 | Pointer to last DATA/INPUT/GET |
| 69/70 | Actual variable (2 letters) |

| | |
|---|---|
| 71/72 | Pointer to actual variable |
| 73/74 | Pointer for present FOR-NEXT variable |
| 75/76 | Help register for BASIC program pointer |
| 77 | Help register for comparison |
| 78/79 | Pointer for FN's |
| 80-83 | Help register for strings |
| 84-86 | Jump vector for function |
| 87-91 | Arithmetic accumulator 3 |
| 92-96 | Arithmetic accumulator 4 |
| 97-101 | Arithmetic accumulator 1 |
| 102 | Pointer to accumulator 1 |
| 103 | Numbers for polynomial value |
| 104 | Rounded byte for accumulator 1 |
| 105-109 | Arithmetic accumulator 2 |
| 110 | Pointer to accumulator 2 |
| 111 | Compare register for accumulator 1&2 |
| 112 | Rounding byte |
| 113-114 | Pointer to polynomial value |
| 115-138 | CHRGET-routine gets next byte from BASIC program. |
| 122/123 | BASIC program pointer |
| 139-143 | Last RND value |
| 144 | Status (like variable ST) |
| 145 | Flags for keyboard column 1146 time constant for cassette operations |
| 147 | 0=LOAD, 1=VERIFY |
| 148 | Flag for IEC bus |
| 149 | Pointer to IEC bus |
| 150 | Flag for end of tape (cassette) |
| 151 | Exchange location for register |
| 152 | Digit of open file |
| 153 | Current input device (normal:0) |
| 154 | Current output device (CMD, normal:3) |
| 155 | Parity byte for cassettes |
| 156 | Flag for byte reception |
| 157 | Output mode (128=direct,0=prg.) |
| 158 | Proof sum in cassette operation |
| 159 | Error correction for cassette |
| 160-162 | Clock |
| 163 | Bit count for serial output |
| 164 | Counter for band operation |
| 165 | Counter for writing to band |
| 166 | Pointer in cassette buffer |
| 167-171 | Flags for band operation |
| 172/173 | Pointer to cassette buffer |
| 174/175 | Pointer to program end (LOAD/SAVE) |
| 176/177 | Time constant for cassette |

| | |
|---|---|
| 178/179 | Pointer to cassette buffer start |
| 180 | Bit counter (cassette) |
| 181 | Next bit for RS232 (sending) |
| 182 | Output byte |
| 183 | Character count in file name |
| 184 | Current logical file number |
| 185 | Current secondary address |
| 186 | Current device number(8:floppy) |
| 187/188 | Pointer to filename |
| 189 | Help pointer for serial output |
| 190 | Block count for band operation |
| 191 | Word buffer for serial output |
| 192 | Flag for cassette motor |
| 193/194 | Start addres for LOAD/SAVE |
| 195/196 | End address for LOAD/SAVE |
| 197 | Pressed key |
| 198 | Digit of pressed key in buffer |
| 199 | Flag for RVS |
| 200 | Line end for input (pointer) |
| 201/202 | Pointer for input cursor (line,col) |
| 203 | Pressed key |
| 204 | Flag for cursor (0=blinking) |
| 205 | Count for blink time |
| 206 | Character under cursor |
| 207 | Blinking flag |
| 208 | Flag for input of keyboard |
| 209/210 | Pointer to actual monitor line |
| 211 | Cursor column |
| 212 | Type of cursor (program/direct) |
| 213 | Length of screen line (40/80) |
| 214 | Cursor line |
| 215 | Last key |
| 216 | Digit of insert |
| 217-242 | High-byte of line start |
| 243/244 | Cursor position in color ram |
| 245/246 | Pointer to keyboard decode table |
| 247/248 | Pointer to input buffer for RS232 |
| 249/250 | Pointer to output buffer for RS232 |
| 251-254 | Free bytes for operating system |
| 255 | Start of the BASIC memory (*64) |
| 256-511 | Processor stack |
| 256-266 | Exchange register for format change |
| 256-318 | Correction for band error |
| 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 time error on serial bus |
| 646 | Current character color |
| 647 | Color under cursor position |
| 648 | High byte of TV-RAM base address |
| 649 | Maximum length of keyboard buffer |
| 650 | Flag for repeat (0=normal, 128=all, 127=out) |
| 651 | Counter for repeat function |
| 652 | Counter for repeat delay |
| 653 | Flag for SHIFT,Commodore&CTRL |
| 654 | Just like 653 |
| 655/656 | Pointer to keyboard decode table |
| 657 | Flag for character exchange |
| 658 | Flag for scrolling |
| 659 | Control register for RS232 |
| 660 | Command register for RS232 |
| 661/662 | Bit time |
| 663 | Status register for RS232 |
| 664 | Digit for output bit for RS232 |
| 665/666 | Baud rate for RS232 |
| 667 | Counter for reception byte RS232 |
| 668 | Counter for input of RS232 |
| 669 | Pointer to output byte for RS232 |
| 670 | Output pointer for RS232 |
| 671/672 | Exchange location for IRQ band operations |
| 673 | NMI flag CIA 2 |
| 674 | Timer A of the CIA 1675 interrupt flag of CIA 1 |
| 676 | Flag for timer A |
| 677 | Monitor line |
| 678-767 | Free RAM region |
| 704-766 | Sprite block 11 |
| 768/769 | Pointer for error report |
| 770/771 | Pointer to BASIC warm start |
| 772/773 | Pointer to exchange text/code |
| 774/775 | Pointer to exchange code/text |
| 776/777 | Pointer to command control |
| 778/779 | Pointer to output value |
| 780 | Accumulator for SYS |
| 781 | X-register for SYS |
| 782 | Y-register for SYS |
| 783 | P-register for SYS |
| 784-787 | USR-jump (address in 785/786) |

| | |
|---|---|
| 788/789 | Pointer to hardware interrupt |
| 790/791 | Pointer to BRK interrupt |
| 792/793 | Pointer to NMI |
| 794/795 | Pointer to OPEN |
| 796/797 | Pointer to CLOSE |
| 798/799 | Pointer to character input |
| 800/801 | Pointer to character output |
| 802/803 | Pointer to channel clear |
| 804/805 | Pointer to input |
| 806/807 | Pointer to output |
| 808/809 | Pointer to STOP key polling |
| 810/811 | Pointer to GET |
| 812/813 | Pointer to close all channels |
| 814/815 | Pointer to user IRQ |
| 816/817 | Pointer to LOAD |
| 818/819 | Pointer to SAVE |
| 820-827 | Free RAM region |
| 828-1019 | Cassette buffer |
| 832-894 | Sprite block 13 |
| 896-958 | Sprite block 14 |
| 960-1022 | Sprite block 15 |
| 1023 | Free |
| 1024-2023 | TV-RAM |
| 2024-2039 | Free |
| 2040-2047 | Pointer to sprites |
| 2048-40960 | BASIC memory |
| 8192-16192 | Bit map for hi-res graphics |
| 40960-49151 | BASIC interpreter ROM |
| 49152-53247 | 4K RAM for machine program |
| 53248-57343 | Character generator |
| 53248-53294 | Register for the VIC |
| 53295-54271 | 977 bytes empty |
| 54272-54300 | Register of the SID |
| 54301-55295 | 995 bytes empty |
| 55296-56295 | Color ram |
| 56296-56319 | 24 bytes empty |
| 56320-56335 | Register of the CIA 1 |
| 56320/56321 | Keyboard poll and joysticks |
| 56336-56575 | 240 bytes empty |
| 56576-56591 | Register of the CIA 2 |
| 56577&56579 | USER-PORT register |
| 56592-57343 | 752 bytes empty |
| 57334-65535 | Operating system ROM |

# Appendix G:

## VIC and VDC

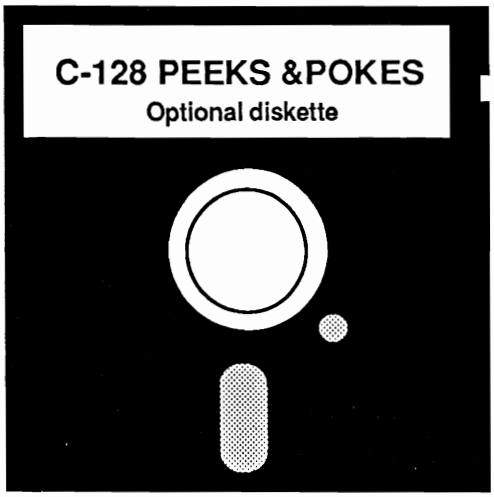
VIC: base address 53248

| REGISTER | CONTENTS |
|---|---|
| 0-15: | Sprite coordinates (X,Y) |
| 16: | MSB of the Y-coordinate |
| 17: | Control register |
| 18: | Raster line for interrupt |
| 19/20: | Light pen position (X,Y) |
| 21: | Sprite enable |
| 22: | Control register |
| 23: | Y-expansion |
| 24: | Video-RAM/char-ROM control register |
| 25/26: | Interrupt control |
| 27: | Sprite priorities |
| 28: | Sprite multicolor mode |
| 29: | X-expansion |
| 30: | Sprite/sprite collision |
| 31: | Sprite/data collision |
| 32: | Frame color |
| 33-36: | Background color 0-3 |
| 37/38: | Multi-color for sprites |
| 39-46: | Sprite colors |
| (47): | Keyboard control |
| (48): | 2-MHz control |

| | |
|---|---|
| VDC: | Base address 54784 |
| | |
| 54784: | Register selection |
| 54785: | Data transfer |
| | |
| 0: | Total count characters per line |
| 1: | Count of characters per line |
| 2/3: | Horizontal synchronization |
| 4: | Total count of characters |
| 5: | Vertical adjustment |
| 6: | Count lines |
| 7: | Vertical synchronization |
| 8: | Interlace mode |
| 9: | Raster line count |
| 10: | Cursor mode |
| 11: | Cursor size |
| 12/13: | Video ram start |
| 14/15: | Cursor position |
| 16/17: | Light pen (X,Y) |
| 18/19: | Update (HI,LO) |
| 20/21: | Attribute (HI,LO) |
| 22: | Brightness of characters |
| 23: | Length of a character |
| 24: | Vertical smooth scrolling |
| 25: | Horizontal smooth scrolling |
| 26: | Color |
| 27: | Address increment |
| 28: | Start character generator |
| 29: | Underline character |
| 30: | Word count |
| 31: | Data |
| 32/33: | Blockstart (HI,LO) |
| 34/35: | Display enable |
| 36: | RAMrefresh rate |

# Index

# Optional Diskette




For your convenience, the program listings contained in this book are available on an 1541 formatted floppy disk. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus Software
P.O. Box 7219
Grand Rapids, MI 49510

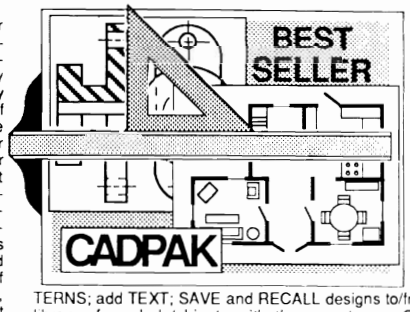
Or for *fast* service, call **(616) 241-5510**.

# SUPER SOFTWARE

## BASIC Compiler

Give your BASIC programs the speed and performance they deserve

The *complete* compiler and development package. Speed up your programs 5x to 35x. Many options: flexible memory management; choice of compiling to machine code, compact p-code or both. '128 version: 40 or 80 column monitor output and FAST-mode operation. '128 Compiler's extensive 80-page programmer's guide covers compiler directives and options, two levels of optimization, memory usage, I/O handling, 80 column hi-res graphics, faster, higher precision math functions, speed and space saving tips, more. A great package that no software library should be without. **128 Compiler $59.95**
**64 Compiler $39.95**



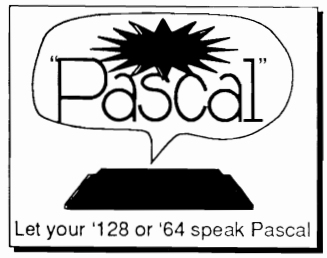

Remarkably easy-to-use interactive drawing package for accurate graphic designs. New dimensioning features to create exact scaled output to all major dot-matrix printers. Enhanced version allows you to input via keyboard or high quality lightpen. Two graphic screens for COPYing from one to the other. DRAW, LINE, BOX, CIRCLE, ARC, ELLIPSE available. FILL objects with preselected PATTERNS; add TEXT; SAVE and RECALL designs to/from disk. Define your own library of symbols/objects with the easy-to-use OBJECT MANAGEMENT SYSTEM–store up to 104 separate objects.
**C-128 $59.95**
**C-64 $39.95**

## Cuper Language Compiler

On your C128 or C64

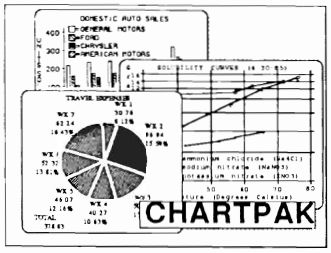
*The language of the 80's and beyond*

For school or software development. Learn C on your Commodore with our in-depth tutorial. Compile C programs into *fast* machine language. C-128 version has added features: Unix™-like operating system; 60K RAM disk for fast editing and compiling Linker combines up to 10 modules; Combine M/L and C using CALL; 51K available for object code; Fast loading (8 sec. 1571, 18 sec. 1541); Two standard I/O librarys plus two additional libraries—math functions (sin, cos, sqrt, etc.) & 20+ graphic commands (line, fill, dot, etc.).
**C-128** $59.95
**C-64** $59.95




Let your '128 or '64 speak Pascal

Not just a compiler, but a complete system for developing applications in Pascal with graphics and sound features. Extensive editor with search, replace, auto, renumber, etc. Standard J & W compiler that generates fast machine code. If you want to learn Pascal or to develop software using the best tools available–**SUPER Pascal** is your first choice.
**C-64 $59.95**




Easily create professional high quality charts and graphs without programming. You can immediately change the scaling, labeling, axis, bar filling, etc. to suit your needs. Accepts data from CalcResult and MultiPlan. C-128 version has 3X the resolution of the '64 version. Outputs to most printers.
**C-128 $39.95**
**C-64 $39.95**

## OTHER TITLES AVAILABLE:

### COBOL Compiler

Now you can learn COBOL, the most widely used commercial programing language, and learn COBOL on your 64. COBOL is easy to learn because it's easy to read. COBOL Compiler package comes complete with Editor, Compiler, Interpreter and Symbolic Debugger. **C-64 $39.95**

BEST SELLER

### Personal Portfolio Manager

Complete portfolio management system for the individual or professional investor. Easily manage your portfolios, obtain up-to-the-minute quotes and news, and perform selected analysis. Enter quotes manually or automatically through Warner Computer Systems. **C-64 $39.95**

### PowerPlan

One of the most powerful spreadsheets with integrated graphics. Includes menu or keyword selections, online help screens, field protection, windowing, trig functions and more. PowerGraph, the graphics package, is included to create integrated graphs and charts. **C-64 $39.95**

### Xper

XPER is the first "*expert system*" for the C-128 and C-64. While ordinary data base systems are good for reproducing facts, XPER can derive knowledge from a mountain of facts and help you make expert decisions. Large capacity. Complete with editing and reporting. **C-64 $59.95**

| | |
|---|---|
| Technical Analysis System for the C-64 | **$59.95** |
| Ada Compiler for the C-64 | **$39.95** |
| VideoBasic Language for the C-64 | **$39.95** |

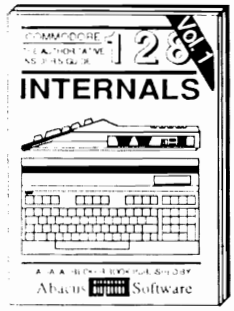


# Abacus Software

**P.O. Box 7219 Dept.M9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510**

Call **now** for the name of your nearest dealer. Or to order directly by credit card, MC, AMEX of VISA call **(616) 241-5510**. Other software and books are available–Call and ask for your **free** catalog. Add $4.00 for shipping per order. Foreign orders add $12.00 per item. Dealer inquires welcome–1400+ nationwide.

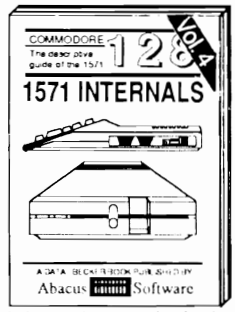
# Top shelf books from Abacus



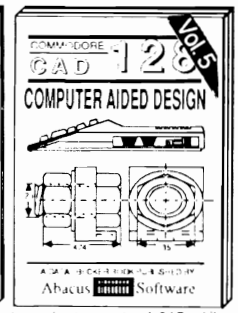

detailed guide presents the 128's operating system, explains graphic chips. Memory Management Unit, 80 column graphics and commented ROM listings. **500pp** $19.95


Get all the inside information on BASIC 7.0. This exhaustive handbook is complete with commented BASIC 7.0 ROM listings. Coming Summer '86. $19.95


Filled with info for everyone. Covers 80 column hi-res graphics, windowing, memory layout, Kernal routines, sprites, software protection, autostarting. **300pp** $19.95

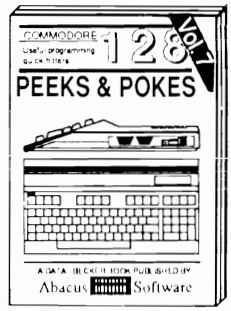



Insiders' guide for novice & advanced users. Covers sequential & relative files, & direct access commands. Describes DOS routines. Commented listings. $19.95

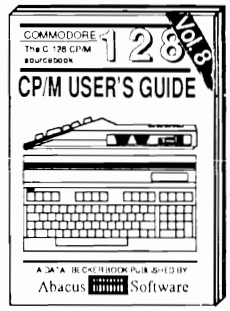



Learn fundamentals of CAD while developing your own system. Design objects on your screen to dump to a printer. Includes listings for '64 with Simon's Basic. **300pp** $19.95

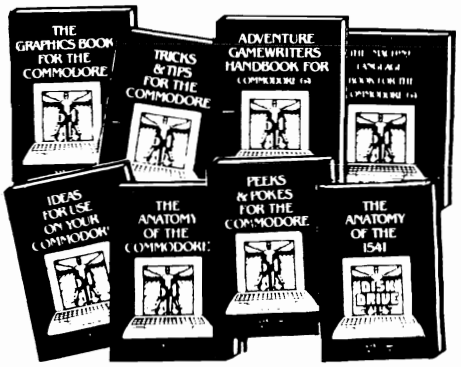

troduction to programing, problem analysis, thorough description of all BASIC commands with hundreds of examples, monitor commands, utilities, much more. $16.95




Presents dozens of programming quick-hitters. Easy and useful techniques on the operating system, stacks, zero-page, pointers, the BASIC interpreter and more. $16.95




Essential guide for everyone interested in CP/M on the 128. Simple explanation of the operating system, memory usage, CP/M utility programs, submit files & more. $19.95




**ANATOMY OF C-64** Insider's guide to the 64 internals. Graphics, sound, I/O, kernal, memory maps, more. Complete commented ROM listings. **300pp** $19.95

**ANATOMY OF 1541 & DRIVE** Best handbook on Happy-Speed explains all. Many examples and Utilities. Fully commented 1541 ROM listings. **500pp** $19.95

**MACHINE LANGUAGE C-64** Learn 6510 code write fast programs. Many samples and listings for complete assembler, monitor, & simulator. **200pp** $14.95

**GRAPHICS BOOK C-64** - best reference covers basic and advanced graphics. Sprites, animation, Hires, Multicolor, lightpen, 3D-graphics, IRQ, CAD, projections, curves, more. **350pp** $19.95

**TRICKS & TIPS FOR C-64** Collection of easy-to-use techniques: advanced graphics, improved data input, enhanced BASIC, CP/M, more. **275pp** $19.95

**1541 REPAIR & MAINTENANCE** Handbook describes the disk drive hardware. Includes schematics and techniques to keep 1541 running. **200pp** $19.95

**ADVANCED MACHINE LANGUAGE** Not covered elsewhere: - video controller, interrupts, timers, clocks, I/O, real time, extended BASIC, more. **210pp** $14.95

**PRINTER BOOK C-64/VIC-20** Understand Commodore, Epson-compatible printers and 1520 plotter. Packed: utilities; graphics dump; 3D-plot; commented MPS801 ROM listings, more. **330pp** $19.95

**SCIENCE/ENGINEERING ON C-64** In depth intro to computers in science. Topics: chemistry, physics, biology, astronomy, electronics, others. **350pp** $19.95

**CASSETTE BOOK C-64/VIC-20** Comprehensive guide; many sample programs. High speed operating system fast file loading and saving. **225pp** $14.95

**IDEAS FOR USE ON C-64** Themes: auto expenses, calculator, recipe file, stock lists, diet planner, window advertising, others. Includes listings. **200pp** $12.95

**COMPILER BOOK C-64/C-128** All you need to know about compilers: how they work; designing and writing your own; generating machine code. With working example compiler. **300pp** $19.95

**Adventure Gamewriter's Handbook** Step-by-step guide to designing and writing your own adventure games. With automated adventure game generator. **200pp** $14.95

**PEEKS & POKES FOR THE C-64** Includes in-depth explanations of PEEK, POKE, USR, and other BASIC commands. Learn the "inside" tricks to get the most out of your '64. **200pp** $14.95

**Optional Diskettes for books**
For your convenience, the programs contained in each of our books are available on diskette to save you time entering them from your keyboard. Specify name of book when ordering. **$14.95 each**

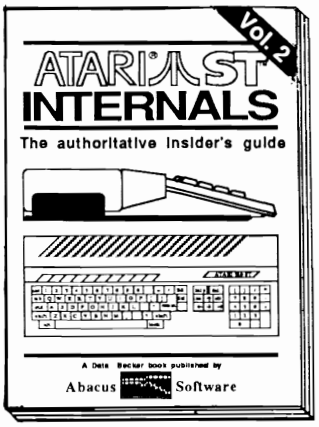
C-128 and C-64 are trademarks of Commodore Business Machines Inc

# Abacus Software

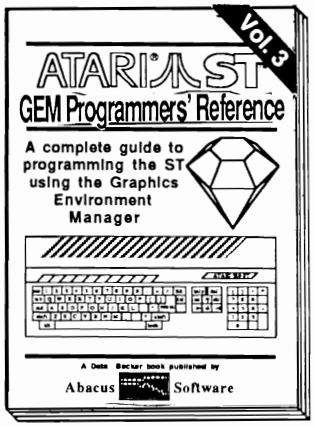
## P.O. Box 7219 Dept.M9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes available for all book titles - $14.95 each. Other books & software also available. Call for the name of your nearest dealer. Or order directly from ABACUS using your MC, Visa or Amex card. Add $4.00 per order for shipping. Foreign orders add $10.00 per book. Call now or write for your **free** catalog. Dealer inquires welcome--over 1400 dealers nationwide.

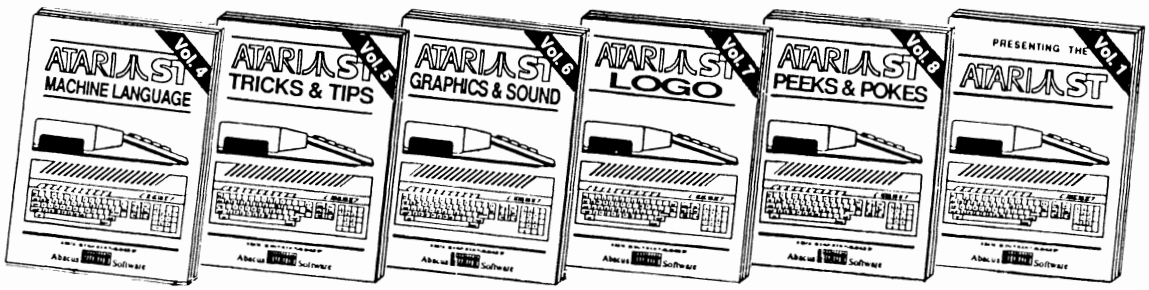
# ATARI® ST™ REFERENCE LIBRARY




Essential guide to learning the inside information on the ATARI ST. Written for the user who wants thorough and complete descriptions of the inner workings of the ST. Detailed descriptions of the sound and graphics chips, the internal hardware, the Centronics and RS-232 ports, GEM, important system addresses and plenty more. Also included is a complete documented BIOS assembly listing. This indispensible reference is a required addition to your ATARI ST library. 450 pages. $19.95




For the serious programmer in need of detailed information on the GEM operating system. Written especially for the Atari ST with an easy-to-understand format that even beginners will be able to follow. All GEM routines and examples are written in C and 68000 assembly language. Covers working with the mouse, icons, Virtual Device Interface (VDI), Application Environment Services (AES) and the Graphics Device Operating System. Required reading for the serious programmer intrested in understanding the ST. 450 pages. $19.95




**MACHINE LANGUAGE**
Program in the fastest language for your Atari ST. Learn the 68000 assembly language, its numbering system, use of registers, the structure & important details of the instruction set, and use of the internal system routines. 280pp $19.95

**TRICKS & TIPS**
Treasure trove of fascinating tips and tricks allows you to make full use of your ATARI ST. Fantastic graphics, refining programs in BASIC, assembler, and C. Includes program listings for RAM disk, printer spooler and more. $19.95

**GRAPHICS & SOUND**
A comprehensive handbook showing you how to create fascinating graphics and suprising music and sound from the ATARI ST. See and hear what sights and sounds that you're capable of producing from your ATARI ST. $19.95

**LOGO**
Take control of your ATARI ST by learning LOGO–the easy-to-use, yet powerful language. Topics covered include structured programming, graphic movement, file handling and more. An excellent book for kids as well as adults. $19.95

**PEEKS & POKES**
Enhance your programs with the examples found within this book. Explores using the different languages BASIC, C, LOGO and machine language, using various interfaces, memory usage, reading and saving from and to disk, more. $19.95

**PRESENTING THE ST**
Gives you an in-depth look at this sensational new computer. Discusses the architecture of the ST, working with GEM, the mouse, operating system, all the various interfaces, the 68000 chip and its instructions, LOGO. $16.95

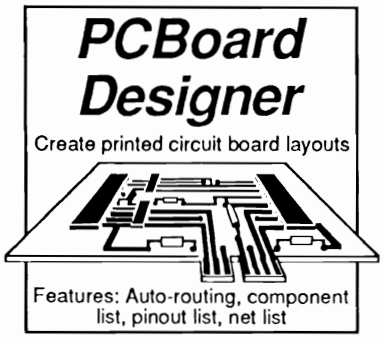
## Abacus Software

**P.O. Box 7219 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510**

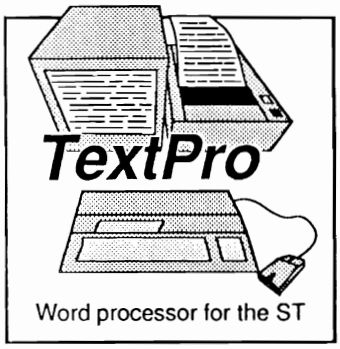
Optional diskettes are available for all book titles at **$14.95**
Call **now** for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add $4.00 per order for postage and handling. Foreign add $10.00 per book. **Other software and books coming soon**. Call or write for **free** catalog. Dealer inquiries welcome–over 1200 dealers nationwide.

# PROFESSIONAL PRODUCTIVITY

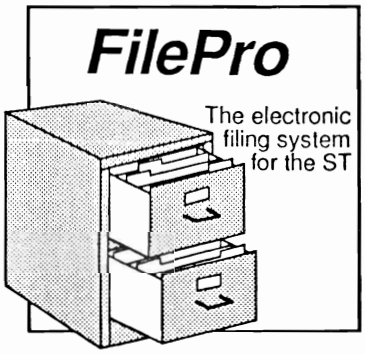
## *New ST software from a name you can count on...*



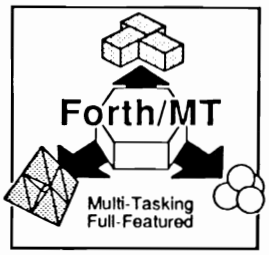

**PCBoard Designer**
Interactive, computer-aided design package that automates layout of printed circuit boards. **Auto-routing** with 45° or 90° traces; two-sided boards; pin-to-pin, pin-to-BUS or BUS-to-BUS. Rubberbanding of components during placement. Prints board layout, pinout, component list, net list. Output to Epson printer at 2:1. Pays for itself after first designed board. **$395.00**



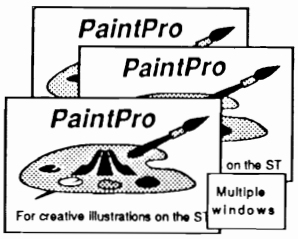

**ST TextPro**
Wordprocessor with professional features and easy-to-use! Full-screen editing with mouse or keyboard shortcuts. High speed input, scrolling and editing; sideways printing; multi-column output; flexible printer installation; automatic index and table of contents; up to 180 chars/line; 30 definable function keys; metafile output; much more. **$49.95**



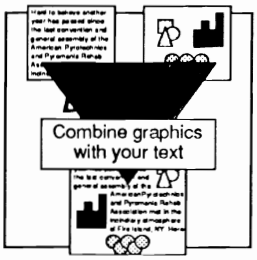

**ST FilePro**
A simple-to-use and versatile database manager. Features help screens; lightning-fast operation; tailorable display using multiple fonts; user-definable edit masks; capacity up to 64,000 records. Supports multiple files. RAM-disk support for 1040ST. Complete search, sort and file subsetting. Interfaces to TextPro. Easy printer control. **$49.95**




**ST Forth/MT**
Powerful, multi-tasking Forth for the ST. A complete, 32-bit implementation based on Forth-83 standard. Development aids: full screen editor, monitor, macro assembler. 1500+ word library. TOS/LINEA commands. Floating point and complex arithmetic. Available Sept. '86. **$49.95**




**ST PaintPro**
A GEM™ among ST drawing programs. Very friendly, but very powerful. A *must* for everyone's artistic or graphics needs. Use up to **three** windows. Free-form sketching; lines, circles, ellipses, boxes, text, fill, copy, move, zoom, spray, paint, erase, undo, help. **$49.95**




**ST Text Designer**
An ideal package for page layout on the ST. Accepts prepared text files from TextPro or other ASCII wordprocessors. Performs block operations— copy, move, columns. Merges bit-mapped graphics. Tools to add borders & separator lines, more. Available September '86. **$49.95**


**ST AssemPro**
Professional developer's package includes editor, two-pass interactive assembler with error locator, online help including instruction address mode and GEM parameter information, monitor-debugger, disassembler and 68020 simulator, more. Available Sept. '86. **$59.95**



# Abacus Software

**P.O. Box 7219 Dept.N9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510**

Call **now** for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add $4.00 per order for postage and handling. Foreign add $10.00 per item. **Other software and books coming soon.** Call or write for your **free** catalog. Dealer inquiries welcome–over 1400 dealers nationwide.

# COMMODORE® 128™

## Unlock the secrets of the C-128

# PEEKS & POKES

**Peeks & Pokes** clearly explains the procedures involved in exploring the C-128's operating system. Includes a large number of important POKEs and their uses as well as a first-class explanation of the C-128's internal design. Some of the topics discussed are:

- How a computer works
- How the interpreter works
- Bank switching
- Pointer and stack
- Mass storage and peripherals
- The basis of machine language
- Tone generation
- The user port
- 8502 & Z-80 machine language
- The operating system
- RAM expansion commands
- Zero page
- Memory map
- The 40/80 column screens
- 640 x 200 hires graphics
- The keyboard
- BASIC and operating system

About the authors:
Hans Joachim Liesert, student at RWTH, Aachen, Germany, has been involved with computers for many years and is the author of the book **Peeks & Pokes for the Commodore 64**. Rüdiger Linden is also a student at RWTH, studying industrial machine design, and is best known for an European book on computer games.

ISBN 0-916439-67-4



A Data Becker book published by

You Can Count On Abacus Software

P.O. Box 7219 Grand Rapids, MI 49510 · Telex 709-101 · Phone 616/241-5510