COMMAND EXTENSIONS

After you switch on the C-128 it displays the following message:

```
COMMODORE BASIC V7.0 122365 BYTES FREE
(C)1985 COMMODORE ELECTRONICS, LTD.
(C)1977 MICROSOFT CORP.
ALL RIGHTS RESERVED
```

Microsoft developed the popular version of BASIC called *MBASIC* that's widely used on CP/M computers. *MBASIC* has a large command set; perhaps this is why the BASIC 7.0 of the C-128 has so many commands. BASIC V7.0 is upward compatible to the BASIC V2.0, V4.0 and V3.5. The V2.0 version is the grandfather of the other BASICs; the VIC-20 and the C-64 contain BASIC V2.0. BASIC V4.0 is used in the large Commodore computers, such as the 8000 series. Commands for the disk drives were added to the commands of BASIC V2.0 (for example, DIRECTORY).

The next BASIC version, BASIC V3.5, is for the C-16, C-116 (available in Europe) and the Plus/4 computers. BASIC V3.5 contained many new graphic commands. Programming aids and a machine language monitor are also included in V3.5

The C-128 features BASIC V7.0. Even more commands have been added, such as sprite control, music programing (limited in version V3.5) and disk commands. And there is another convenient option: the command GO 64 .

This command turns the C-128 into a C-64 .

Even with all the new commands, you'll find that more are required. How about PRINT AT?, or POP?, or GOTO X? and RESTORE X?. We easily could continue this list. As a frequent programmer you'll rely on several helpful routines, such as APPEND. What then? You'd usually call them with a SYS and some values behind it. But how many addresses can you remember, and how understandable is a program when it's full SYS commands?

Luckily it's easy to insert new commands into the operating systems of Commodore computers. We use the routine called CHRGET (CHRGET means CHaRacter + GET) to accomplish this small miracle.

8.1. WHAT IS THE CHRGET ROUTINE ?

When you turn on the C-128 the CHRGET routine is copied from the ROM to the RAM. We will explain that later. Right now we'll deal with the function of the CHRGET routine.

With the CHRGET routine, the computer takes a character from the BASIC text, meaning from the program memory or from the BASIC input buffer. The BASIC interpreter always jumps to the CHRGET routine to get the next character. Lets have a look at the CHRGET routine for the C-64:

```
0073    INC   $7A
0075    BNE   $0079
0077    INC   $7B
```



```
0079    LDA   $HHLL
007C    CMP   #$3A
007E    BCS   $008A
0080    CMP   #$20
0082    BEQ   $0073
0084    SEC
0085    SBC   #$30
0087    SEC
0088    SBC   #$D0
008A    RTS
```

The first three instructions increment the value of the text pointer. The least significant byte of the pointer is incremented. But if there is a carry (meaning greater than 255), then the most significant byte of the pointer is also incremented. When you look at the address of the pointer (called $HHLL in the above routine) you will notice it is located within the CHRGET routine. Now the ROM - RAM copy procedure makes sense: the routine changes itself!

The actual value of $HHLL is the pointer to a location within the BASIC program text. The pointer is called TXTPTR and is actually a vector at $007A. The CHRGET routine loads the accumulator (LDA instruction at address $0079) with the next character of BASIC text pointed to by TXTPTR.

This character is checked in several ways. First the character will be compared with the ASCII value for the semi-colon. The routine returns when the character is larger or equal to this value. If the character is the semi-colon, the zero-flag becomes set , in addition to the carry-flag. If the character number is smaller, the tests continue. The first test checks to see if the character is a space character. If so, the next character will be examined. In effect, space characters are ignored (PRINTX has the same

effect as PRINT X). The two next commands check to see if the character is a number between 0 - 9.

The CHRGET routine of the C-128 has been changed:

```
0380  INC  $3D
0382  BNE  $0386
0384  INC  $3E
0386  STA  $FF01
0389  LDY  #$00
038B  LDA  ($3D), Y
038D  STA  $FF03
0390  CMP  #$3A
0392  BCS  $039E
0394  CMP  #$20
0396  BEQ  $0380
0398  SEC
0399  SBC  #$30
039B  SEC
039C  SBC  #$D0
039E  RTS
```

The first three instructions increment a pointer again. This pointer is not within the CHRGET routine anymore, but in zero page. The next instruction assures that the computer takes this information from the correct bank. This instruction is similar to the BASIC command BANK 0. RAM Bank 0 is always activated. That guarantees that the program can use the full 64K of the RAM in Bank 0 (minus the first few pages that are used by the system).

The next two commands instructions retrieve a character from the BASIC text. Because the Y register is used as an offset, it must be set to zero first.

The instruction at address $038D switches the banks back again. This machine language instruction is equivalent to the BASIC command BANK 14, which switches on BASIC interpreter, character generator, and RAM Bank 0.

The following instructions are just like the C-64's, meaning the check procedure for semi-colon, space character, etc.

A last word on the CHRGET routine: once in a while the operating system jumps to the CHRGET routine at address $0386 ($0079 in the C-64). Then it's called CHRGOT. You will also get a character, but the pointer is not incremented. The last character will be read.

8.2. CHANGING THE CHRGET ROUTINE

Here we limit our discussion to the C-128, because there is enough literature available on how to change the routine on the C-64. To insert new commands you have to change the CHRGET routine--but where do you do that? Theoretically you could do it anywhere, but only two solutions are practical:

a) Each time you get a character, a certain function will be executed.

b) A new command must be inserted.

In a) you want to change the CHRGET routine at the start. Here is an example. Enter the monitor and type in the following assembler listing:

```
0380  JSR  $0B00
0383  NOP
0384  NOP
0385  NOP
--------------
0B00  INC  $3D
0B02  BNE  $0B06
0B04  INC  $3E
0B06  LDA  #$00
0B08  STA  $FF00
0B0B  LDA  $D020
0B0E  E0R  #$FF
0B10  STA  $D020
0B13  RTS
```

Leave the monitor with the "X" command, type in a character and press the <RETURN> key. The border color will change (the border color of the forty column screen, because we are manipulating the VIC chip). Let's have a closer look at the listing.

It has two parts. Part one is located in the CHRGET routine and alters the routine itself. When you enter a character now, the computer will call a subroutine at $0B00. This subroutine constitutes part two. Here, it continues the first part of the CHRGET routine that was replaced by the first part. Then the I/O is switched on to gain access to the VIC chip (Bit 0 is responsible for this; see Chapter 9, "Banking"). The access to the VIC chip occurs with the next three instructions. Here we change the border color. The bank that was on before the called routine switches on again. Finally, you return to the CHRGET routine with a RTS.

You have to remember one thing: the functions performed by the CHRGET routine must be kept intact. If any section is missing, the operating system will not function properly anymore. For example, the system will crash if you leave out the instructions in the second part of the assembler listing that raise the indicator in the CHRGET - routine.

Very seldom are commands one character in length. Usually you would like an action performed only if several characters are entered. This character sequence becomes the new command.

The CHRGET routine can accomplish this task. Enter the following assembler listing (you should reset the computer before entering the listing):

```
0380    INC    $3D
0382    BNE    $0386
0384    INC    $3E
0386    STA    $FF01
0389    LDY    #$00
038B    LDA    ($3D), Y
038D    JMP    $0B00
0390    STA    $FF03
0393    NOP
- - - - - - - - - - - - - - - - -
0B00    CMP    #$FF
0B02    BNE    $0B14
0B04    LDA    #$00
0B06    STA    $FF00
0B09    LDA    $D020
0B0C    EOR    #$FF
0B0E    STA    $D020
0B11    JMP    $0380
0B14    CMP    #$3A
0B16    BCS    $0B1B
0B18    JMP    $0390
0B1B    STA    $FF03
0B1E    RTS
```

This time the CHRGET routine was changed at address $038D. There we jump to address $0B00 (this time not JSR, but JMP). The instruction at address $0390 (STA $FF03) was taken from the old CHRGET routine. The NOP instruction assigns the free byte in the CHRGET routine. Now to the part after address $0B00:

First the character read is compared with the value $FF (255). This value is the token for Pi (π). If the character is not Pi (π), you continue from address $0B14. There the CHRGET routine continues. First the character is compared with that of the semi-colon. If it is larger, Bank 14 will be activated (just like the CHRGET routine); then it will jump to the address where the CHRGET routine was called. If the ASCII value of the character is smaller than the value of the semi-colon, it continues with the JMP instruction to the normal CHRGET routine.

Let's imagine the entered character was Pi (π). In this case you continue the program at address $0B04. Here, Bank 15 switches on to have access to the I/O (in this case the VIC). Then, like the other routine, the value for the screen border is inverted. Then there's the usual jump back to the CHRGET routine, to read the next character.

Now enter a Pi (π) and press the RETURN key. The border color of the forty column screen will change. When you enter Pi (π) again, the frame color goes back to normal.

Contain your curiosity and don't experiment with the Pi (π) character (for example, what happens when you enter it in a program line). First, enter the following assembler listing:


```
0380    INC   $3D
0382    BNE   $0386
0384    INC   $3E
0386    STA   $FF01
0389    LDY   #$00
038B    LDA   ($3D), Y
038D    JMP   $0B00
0390    STA   $FF03
0393    NOP
- - - - - - - - - - - - - - - -
0B00    CMP   #$FF
0B02    BNE   $0B14
0B04    LDA   #$00
0B06    STA   $FF00
0B09    LDA   $D020
0B0C    EOR   #$FF
0B0E    STA   $D020
0B11    JMP   $0380
0B14    CMP   #$88
0B16    BNE   $0B28
0B18    LDA   #$00
0B1A    STA   $FF00
0B1D    LDA   #$00
0B20    EOR   #$FF
0B22    STA   $D021
0B25    JMP   $0380
0B28    CMP   #$3A
0B2A    BCS   $0B2F
0B2C    JMP   $0390
0B2F    STA   $FF03
0B32    RTS
```

Two new commands are inserted by this listing:

a) Pi (π): Here you invert the value for the border color, just like in the previous routine.
b) LET: This command inverts the value for the background color.

8.3. THE "BEHAVIOR" OF THE NEW COMMANDS

Enter the character Pi (π) with a line number and you will see that the command will be executed immediately. It will not be carried out in the program. Now try out the same with the command LET. The command will not be executed right away, but will be executed in a program RUN. As you can see the computer can distinguish between normal and new BASIC commands.

Now enter Pi (π) in a line again, but type in a semi-colon at the start of the line. Again, the command will be executed immediately, but the line will be maintained. That also works when you insert new commands between old commands. Enter the semi-colon and the command LET in the line. In this case you also maintain the line, and when you start the program the border and background color will change. Try the following line:

```
10 LET GOTO 10
```

Even though it looks wrong, it functions well: the value for the background color will change quickly (this effect is really interesting).

That works in the following format also:

```
10 GOTO LET 10
```

This works because the interpretation of the BASIC lines will go to the (changed) CHRGET routine. The token for GOTO will be read first. Because this value does not equal one of the values of the new commands,

it jumps back from the character analysis and branchs to the GOTO command. There it will jump to the CHRGET routine again, to get the line number. But this time the character equals the value of the new command, which is $88 for LET. So, the color is changed and then jumped back to the CHRGET routine. There the next character, the number 1, is read and jumps back to the GOTO command. The command LET is executed but otherwise it is ignored. Of course, that works behind any other command. But there are some limitations:

a) The new command cannot be in other commands. The following example does not work:

```
10 GO LET TO 10
```

That is because GOTO becomes changed into a token first. In this case the GOTO command will not be identified, so it won't get changed to a token. But you can place the new command in numbers:

```
10 A=100LET0
20 PRINT A
```

Here A receives the value 1000.

b) The new command is not identified in strings.

```
10 PRINT "LET"
```

will print LET without changing the background color.

8.4. SEVERAL ADDITIONAL COMMANDS

If you want to add more commands, the listing above would be too tedious. If you wanted to check for every new command, the command extension would grow too large. There is an easier method; as follows:

```
0380    INC   #3D
0382    BNE   $0386
0384    INC   $3E
0386    STA   $FF01
0389    LDY   #$00
038B    LDA   ($3D), Y
038D    JMP   $0B00
0390    STA   $FF03
0393    NOP
- - - - - - - - - - - - - - -
0B00    LDY   #$03
0B02    DEY
0B03    BMI   $0B0C
0B05    CMP   $0B23, Y
0B08    BEQ   $0B17
0B0A    BNE   $0B02
0B0C    CMP   #$3A
0B0E    BCS   $0B13
0B10    JMP   $0390
0B13    STA   $FF03
0B16    RTS
0B17    TYA
0B18    ASL
0B19    TAY
0B1A    LDA   $0B27, Y
0B1D    PHA
0B1E    LDA   $0B26, Y
0B21    PHA
0B22    RTS
0B23    $40,  $88,  $FF

0B26    $2B,  $0B
0B28    $43,  $0B
```



```
0B2A    $53,  $0B

0B2C    LDA   #$00
0B2E    STA   $FF00
0B31    LDA   $D020
0B34    EOR   #$FF
0B36    STA   $D020
0B39    LDA   $D021
0B3C    EOR   #$FF
0B3E    STA   $D021
0B41    JMP   $0380
0B44    LDA   $D021
0B46    STA   $FF00
0B49    LDA   $D021
0B4C    EOR   #$FF
0B4D    STA   $D021
0B51    JMP   $0380
0B54    LDA   #$00
0B56    STA   $FF00
0B59    LDA   $D020
0B5C    EOR   #$FF
0B5E    STA   $D020
0B61    JMP   $0380
```

In the part after $0B00 three commands are defined:

a) Pi (π): Functions like above
b) LET: Functions like above
c) @ = Pi (π) and LET

The principles are easy to understand. The codes of the new commands are stored in a table, and the addresses where the commands should be executed in another table. Take a close look at the listing again:

$0B00: Here you load the number of new commands in the X-register.
$0B02: Decrement by one.
$0B03: When all commands are carried out you jump to address $0B0C.

This BRANCH command also checks if the X-register contains the value zero (you also check if X=2).

$0B05: Here the entered character is compared with a byte from table $0B23 - $0B25. This table contains the values for the new commands.

$0B08: Here you jump to address $0B17 when the entered character is equal with one of the new commands.

$0B0A: If it was not equal, jump to $0B02 and compare the entered character with the next command.

$0B0C: You continue with this address when the entered character does not equal any of the new commands. At address $0B16 the routine is the same as in the normal CHRGET - Routine.

$0B17: You continue here when the entered character was equal with one command. The number of the command (0 - 2) will be placed in the X-register,

$0B18: Multiply by two

$0B19: and place back in the X-register again.

$0B1A: Here is the high-byte of the address where the new command starts, pulled in from the accumulator.

$0B1D: Stored on the stack.

$0B1E: Here you get the accompanying low-byte.

$0B21: And also stored on the stack.

You can see for yourself that with the separate X register values, the following low/high-bytes become stored on the stack:

X-Reg.	Low - Byte	High - Byte	Command
0	$2B	$0B	@
1	$43	$0B	LET
2	$53	$0B	Pi

This address must always show one byte below the address where the routine really starts. So the command LET starts not at $0B43, but at **$0B44. The reason for this is in the next command:**

$0B22: Here is the return from the subroutine. But from which subroutine? $0B00 was called with JMP, and not with JSR. Now by RTS the jump address is called from the stack; first the low-byte, then the high-byte. One address was stored in the stack by the previous instructions, and called in the program counter. This is incremented to get the next command. That is why the address had to be one byte lower than the correct address. You fool the computer, because you tell it, that a JSR instruction brought it to address $0B22 (which is before the start of the respective command).

$0B23 - $0B25: Table with codes of new commands

$0B26 - $0B2B: Table with addresses of new commands (-1!)

The new commands, which start at $0B2C, will not be explained again.

You shouldn't think you can use only commands of one byte lengths, or commands that already exist. Of course you can use your own longer

commands. The length of these commands can be different. When you examine the three tables, please notice that the first contains start addresses of the strings which are to be compared, the second contains those strings, and the third has the addresses of the commands.

And here's a last tip; always place an identifying mark in front of your commands, such as @. That makes it easier to tell your own commands from the commands of the Interpreter. This will make your work easier and faster.

CHAPTER 9

BANKING

9.1. THEORETICAL BASICS

We don't want to write a book about theory, but you should know some of the basics about memory management.

A few years ago the VIC-20 in the standard version had only 5K of RAM. It had 3583 Bytes available for BASIC programs. It had a 20KByte of ROM, which was extendable to 32K RAM and 24K ROM. Together you had 32K + 24K = 56K memory available. This could be addressed with the 6502 processor,without any problems, because it had 16 address lines available which allows for access to 2^16 = 65536 = 64K of memory.

In a short time memory became so inexpensive that computers could be packed with memory and still remain within the price range of home computers. But how was the additional memory supposed to be addressed? To install a CPU with more than 16 address lines would result in an excessive price increase.

Engineers worked out three methods to use more memory than can be addressed. For C-128 owners only one of these is interesting; its called *BANKING* or *BANK SWITCHING*. In this method, memory chips are installed "above each other" , which contain the total amount of addressable memory. This means that two memory banks are located in the exact same address range. But only one memory bank is used at a time. To choose a

particular bank, you can electronically (under program control) switch between those banks. This makes it very easy for the user, but creates a problem for the manufacturer of the computer system.

If, after the bank switching, the program returns to the wrong bank, and hence the incorrect program code, the computer will "hang up". It would be possible to switch only a certain range (for example, the operating system) instead of the total memory range. But that would mean a loss in memory size again. There is a possibility to switch the whole memory with an easy hardware solution (just like in the C-64, when you read the address $A000, then you read from the ROM, but when you write at address $A000 you write in the RAM).

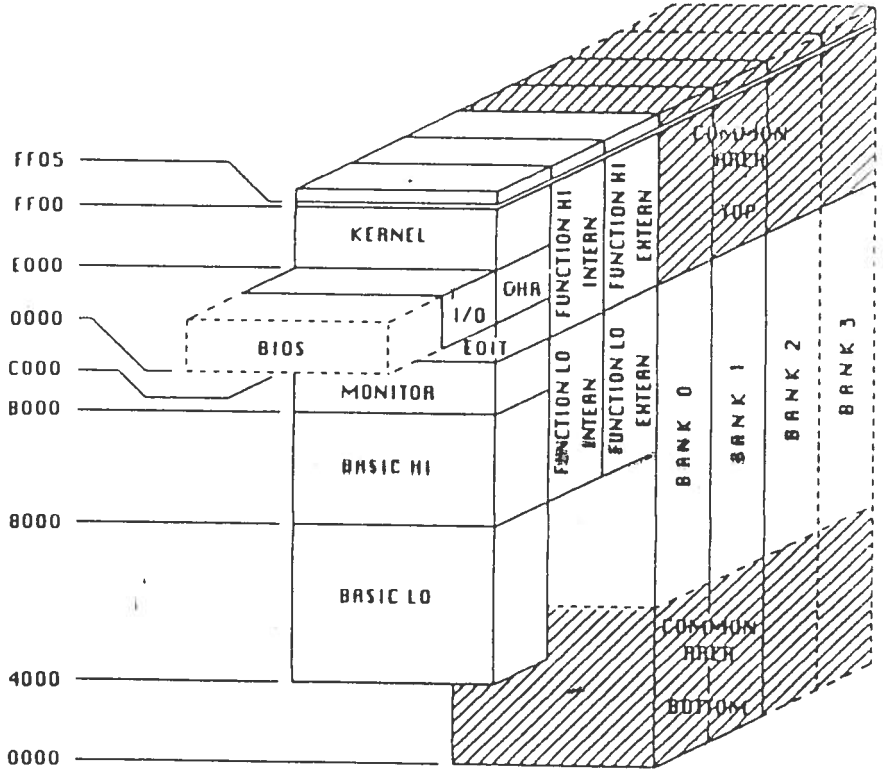
This is enough theory on bank switching and memory management. Let's return to the C-128 memory management.

9.2. BANKING WITH THE C-128

The Commodore advertisments for the C-128 state that it has 128K Bytes RAM (expandable to 512K Bytes) and 48K Bytes of ROM. How can this be possible if the microprocessor can only access 64K bytes at one time? The memory is overlayed and the proper memory bank is selected when necessary. The picture on the next page should clarify this.

But how is the memory management accomplished?

If you examine the memory diagram you will notice that the C-128 uses banking somewhat like the C-64 (the C-128 contains more overlayed memory banks. To manage this big mess, you need an additional integrated circuit. In the C-64 this was the AM (Address Manager). Because Commodore owns a company that makes integrated circuits (MOS Technology), they quickly designed a new integrated circuit called MMU 8722 (Memory Management Unit). The MMU controls which bank is currently accessed and makes sure that the computer always calls the information from the right bank.



9.3. SWITCHING THE BANKS WITH THE MMU

To switch the banks you use only one register of the MMU, register 0. It is called the CR (Configuration Register). The bits have the following function:

Bit	Function
0	**Selects the range $D000 - $E000 :**
	0 = I/O 1 = ROM / RAM
1	Selects the range $4000 - $7FFF :
	0 = ROM 1 = RAM
2 & 3	Selects the range $8000 - $BFFF :
	00 = System ROM 01 = internal function ROM
	10 = external function ROM 11 = RAM
4 & 5	Selects the range $C000 - $FFFF :
	00 = System ROM 01 = internal function ROM
	10 = external function ROM 11 = RAM
6 & 7	Selects the RAM - Memory :
	00 = RAM - Bank 0 01 = RAM - Bank 1
	10 = RAM - Bank 2 11 = RAM - Bank 3

Here are some further explanations:

a) Bits 4 and 5 are dependent on bit 0. This means when bits 4 and 5 are set and 0 is not, then the area from $C000 - $FFFF is not completely RAM. From $D000 - $E000 is the I/O (Input/Output). RAM would be there, when Bit 0 would be set to 1. If Bit 0 is 0, the I/O is switched on independently of bits 4 and 5. But bits 4 and 5 control the memory layout from $C000 - $FFFF.

b) In the present versions of the C-128 bits 6 & 7 make sense in only the first two modes because the RAM Banks 2 & 3 are not available yet.

c) In the area from $FF00 - $FF04 the system memory is located, independent from this register. The first of these five bytes ($FF00) is register 0 of the MMU. You are probably wondering why. The answer is very simple. Imagine you have the I/O switched off, for example, to access to the character generator. Now you would like to turn on the I/O, to change the screen color, but how do you switch it back on? The registers of the MMU are also in the I/O (excuse us for holding that information back so long): from $D500 - $D50B! Now the memory cell $FF00 takes action. Because this register has the same function as register 0 you can switch the I/O back on.

Because banking wouldn't be very user friendly if it could only be done using the registers of the MMU, the programmers of the BASIC interpreter implemented a new command: the BANK command. The SYNTAX is as follows:

```
BANK nr
```

The `nr` is a number from 0 to 15 that activates the respective memory configuration. On the next page is a chart showing the different memory configurations of each bank.

Nr:	Contents of $FF00	Memory Configuration
0	$3F = %00111111	RAM - Bank 0
1	$7F = %01111111	RAM - Bank 1
2	$BF = %10111111	RAM - Bank 2
3	$FF = %11111111	RAM - Bank 3
4	$16 = %00010110	Function - ROM internal, with RAM - Bank 0 and I/O
5	$56 = %01010110	Like above, but RAM - Bank 1
6	$96 = %10010110	Like above, but RAM - Bank 2
7	$D6 = %11010110	Like above, but RAM - Bank 3
8	$2A = %00101010	Function - ROM external, with RAM - Bank 0 and I/O
9	$6A = %01101010	Like above, but RAM - Bank 1
10	$AA = %10101010	Like above, but RAM - Bank 2
11	$EA = %11101010	Like above, but RAM - Bank 3
12	$06 = %00000110	Function - ROM internal low , with Kernal, RAM - Bank 0 and I/O
13	$0A = %00001010	Function - ROM external low , with Kernal, RAM - Bank 0 and I/O
14	$01 = %00000001	Kernal with BASIC, Character - generator and RAM - Bank 0
15	$00 = %00000000	Kernal with BASIC, I/O and RAM - Bank 0

The command "BANK 15" is the the configuration when the computer is turned on.

CHAPTER 10

AUTOSTART

10.1 AUTOSTART FROM THE DISK DRIVE

Many first time Commodore computer users have difficulty in starting their first program; they have not yet learned "computerese". On some computer systems, MS-DOS computers for example, it is possible to start a program by simply turning the computer on and inserting a disk in the disk drive. To start a program with the CP/M and MS-DOS operating systems all you have to do is type in the name of the program and press the <RETURN> key.

In the C-64 it was a little harder. You had to enter LOAD, the name of the program, "8" (maybe even ",8,1"). But admittedly the C-64 was designed for hobbyist programmers, people with some experience in using computers. The C-128 is aimed at a different group, the business user, and because of this it is equipped with the CP/M operating system. As a result it is possible to load programs automatically from disc, starting them when the computer is first turned on. This "autostart" routine is also called by resetting the computer.

This "autostart" routine holds several possibilities. You can load any number of blocks from disk (but they have to start in sequence from track 1, sector 1) before you load any program. After loading you can execute the program. So you could load in a new language (for example FORTH), then a program in the new language and then switch on the new

language. The same can be done with BASIC programs. We'll only know how many possibilities there are after the C-128 is on the market for a while.

In the next sectionwe will explain the "autostart" routine in detail. If you only want to use the "autostart", you don't have to study this section **simply use the programs presented here. But if you want to know more** about how these procedures operate, you should continue reading.

10.1.1. THE BOOT-CALL ROUTINE

The boot-call routine is located in the operating system from $F890 - $F98A. You should call it with address $FF53; this is where it is located in the kernal jump table.

First the listing of the routine:

```
F890   STA $BF
F892   STX $BA
F894   TXA
F895   JSR $F23D
- - - - - - - - - - - - - - -
F898   LDX #$00
F89A   STX $9F
F89C   STX $C2
F89E   INX
F89F   STX $C1
- - - - - - - - - - - - - - -
F8A1   INY
F8A2   BNE $F8A1
F8A4   INX
F8A5   BNE $F8A1
- - - - - - - - - - - - - - -
F8A7   LDX #$0C
F8A9   LDA $FA08,X
```



```
F8B0   BPL $F8A9
- - - - - - - - - - - - - - -
F8B2   LDA $BF
F8B4   STA $0106
F8B7   LDA #$00
F8B9   LDX #$0F
F8BB   JSR $F73F
F8BE   LDA #$01
F8C0   LDX #$15
F8C2   LDY #$FA
F8C4   JSR $F731
F8C7   LDA #$00
F8C9   LDY #$0F
F8CB   LDX $BA
F8CD   JSR $F738
F8D0   JSR $FFC0
F8D3   BCS $F8EB
- - - - - - - - - - - - - - -
F8D5   LDA #$01
F8D7   LDX #$16
F8D9   LDY #$FA
F8DB   JSR $F731
F8DE   LDA #$0D
F8E0   TAY
F8E1   LDX $BA
F8E3   JSR $F738
F8E6   JSR $FFC0
F8E9   BCC $F8EE
- - - - - - - - - - - - - - -
F8EB   JMP $F98B
- - - - - - - - - - - - - - -
F8EE   LDA #$00
F8F0   LDY #$0B
F8F2   STA $AC
F8F4   STY $AD
- - - - - - - - - - - - - - -
F8F6   JSR $F9D5
- - - - - - - - - - - - - - -
F8F9   LDX #$00
F8FB   LDA $0B00,X
F8FF   CMP $E2C4,X
F901   BNE $F8EB
F903   INX
```

```
F904   CPX #$03
F906   BCC $F8FB
- - - - - - - - - - - - - - -
F908   JSR $FA17
F90E   4F 54 49 4E
F912   47 20 00
- - - - - - - - - - - - - - -
F915   LDA $0B00,X
F918   STA $A9,X
F91A   INX
F91B   CPX #$07
F91D   BCC $F917
- - - - - - - - - - - - - - -
F91F   LDA $0B00,X
F922   BEQ $F92A
F924   JSR $FFD2
F927   INX
F928   BNE $F91F
F92A   STX $9E
- - - - - - - - - - - - - - -
F92C   JSR $FA17
F92F   2E 2E 2E 0D
F933   00
- - - - - - - - - - - - - - -
F934   LDA $AE
F936   STA $C6
- - - - - - - - - - - - - - -
F938   LDA $AF
F93A   BEQ $F945
F93C   DEC $AF
F93E   JSR $F9B3
F941   INC $AD
F943   BNE $F938
F945   JSR $F98B
- - - - - - - - - - - - - - -
F948   LDX $9E
F94A   .BYTE $2C
F945   INC $9F
F94D   INX
F94E   LDA 0B00,X
F951   BNE $F94B
F953   INX
F954   STX $04
```



```
F953   INX
F954   STX $04
- - - - - - - - - - - - - - -
F956   LDX $9E
F958   LDA #$3A
F95A   STA #0B00,X
F95D   DEX
F95E   LDA $BF
F960   STA $0B00,X
F963   STX $9E
- - - - - - - - - - - - - - -
F965   LDX $9F
F967   BEQ $F97E
F969   INX
F96A   INX
F96B   TXA
F96C   LDX $9E
F96E   LDY #$0B
F970   JSR $F731
F973   LDA #$00
F975   TAX
F976   JSR $F73F
F979   LDA #$00
F97B   JSR $F269
- - - - - - - - - - - - - - -
F97E   LDA #$0B
F980   STA $03
F982   LDA #$0F
F984   STA $02
F986   JSR $02CD
- - - - - - - - - - - - - - -
F989   CLC
F98A   RTS
```

The first two instructions store the contents of the accumulator and the X-register in the zero page. The function of these two bytes are: the accumulator represents the device address of the disk drive, from where you load a program. The X-register contains the device address of the disk drive, where you will search the disk for an autostart sequence. In most cases these bytes are identical. During a reset both register contents

are set to 8. You execute an autostart only on the disk drive with a device address of 8. A programmer can jump to this routine in many different ways, for example, with `SYS DEC ("FF53"),9,9`. This command searches a disk drive with device address 9 for the autostart track and sector.

The subprogram, called in address $F895, closes all files with the device address in the accumulator. The function is clear: the commands and data, which are supposed to be sent to the disk, wouldn't reach the disk quickly enough if a file was still open.

The files close in the following manner: first a table is searched which contains the device address of the present open files for the determined device number. The computer finds the device number in this table ($0362 - $036B) then it finds the respective file number (the information in both tables are in the same place). This file is closed with the kernal routine CLOSE ($FFC3). This happens with all open files (the amount is in $98). There is a third related table at ($0376 - $037F), this contains the secondary addresses of the open files (in the same sequence as the other two tables).

The next five instructions of the boot-call-routine set the following zero page bytes to the following values:

```
$9F   (159):  00
$C1   (193):  01
$C2   (194):  00
```

The address $9F is a counter and will be used later.

In $C1 is the track number, in $C2 is the sector number of the block of the disk where the information to be loaded is located. The information for the "autostart" boot-call is on track 1, sector 0--the first block on the disk. This block may not be used for any other data storage if you wish to make an "autostart" disk. The ideal solution is, to use only "fresh" formatted disks for "autostart" programs. Or check with a disk monitor to see if the block is used or not on disks that contain data.

Track 1 and sector 0 are not only used for the autostart routine, but also to load the CP/M operating system. CP/M uses the autostart routine to load and execute the new operating system. The next four commands (from $F8A1 - $F8A6) are a delay loop. Then 13 bytes from the operating system (from $FA08 - $FA14) are copied into the RAM (from $0100 - $010C). These bytes have the following values:

```
FA08:  30 30 20 31 30 20 30 20 33 31 3A 31 55
```

When you translate these characters to ASCII format and put them in sequence you will receive the following command:

```
U1:  13 0 01 00"
```

If you are familiar with disk drive commands, the command sequence will make sense to you. If you are not familiar with the disk drive commands, please be patient.

The two instructions from $F8B2 - $F8B6 store the contents from $BF to a string. At the start of the routine the address $BF was loaded with the chosen device address. That means, you write the device address instead of the single zero into the string.

The next eleven instructions have the following function:

Value	Location	Function
#00	$C6	Bank - Number for LOAD/SAVE/VERIFY
#0F	$C7	Bank - Number for present file name
#01	$B7	Length of present Filename
#15	$BB	Low - byte address of file name
#FA	$BC	High - byte address of file name
#00	$B8	Logic data -file number
#0F	$B9	Secondary address
#BA	$BA	Device address

So, the file name is in address $FA15. In this address is the value $49 (73). This value is the ASCII value of the character "I". Then the instruction following (JSR $FFC0) calls the kernal routine OPEN. These assembler instructions are the same as the following BASIC command:

```
OPEN 0, Deviceaddress, 15, "I"
```

The "I" stands for the disk command "INITIALIZE". This command reads the BAM (Block Availability Map) of the inserted floppy disk. This command is used to check the disks for different IDs. The ID on the disk tells the drive if a new disk has been inserted since the last time it read the BAM. The Block Availability Map keeps track of the free space on the

disk. If you switch disks that have the same ID, the drive still thinks the old disk is inserted; this can can cause a great deal of trouble. The programmers of the C-128 didn't trust that all of your disks would have different IDs, so they inserted this command.

Using the file number zero you can send commands to the disk drive. In address $F8D3 is a BRANCH instruction. This jumps to address $F8EB if the disk drive didn't report its presence on the serial bus (for example, when it is switched off). In this case you don't get an error message. Try it out with the following command (you probably don't have your disk drive switched to device address 10).

```
SYS DEC ("FF53"), 10,10
```

"READY" will appear again.

When the disk drive reports its presence on the serial bus, you continue at address $F8D5. There you set more parameters:

Value	Comes after	Function
#01	$B7	Length of the present file name
#16	$BB	Low - byte address of file name
#FA	$BC	High - byte address of file name
#0D	$B8	Logic data - file number
#0D	$B9	Secondary address
#BA	$BA	Device address

The value $23 (35) is in address $FA16, which is the pointer in $BB/$BC. The filename is "#".

The assembler instructions (with the call of the OPEN Routine in $F8E6) is the same as the following BASIC command:

OPEN 13, Deviceaddress, 13, " # "

In case you are not familiar with the disk drive, this reserves a buffer in the disk drive, for the information to be read from the disk. You have access to this buffer over channel 13.

Here you leave the boot-call routine if the disk is not present in the disk drive.

The four instructions from $F8EE - $F8F5 set the bytes $AC/$AD to $0B00 (the cassette buffer). These two zero page addresses are used as a pointer; the cassette buffer is used to store the characters read from the disk.

Address $F8F6 sends a command to the disk drive and characters are called accordingly. This important routine is listed and documented below:

```
F9D5   LDX  #$00       ; Logic Filenumber
F9D7   JSR  $FFC9      ; Output device with Device
                         address of file number 0
F9DA   LDX  #$0C       ; Amount of Chars to send-1
F9DC   LDA  $0100,X    ; Call Character
F9DF   JSR  $FFD2      ; And Send (BSOUT)
F9E2   DEX             ; All Characters ?
F9E3   BPL  $F9DC      ; No
F9E5   JSR  $FFCC      ; Close Present I/O Channel
                         at IEC - Bus (CLRCH)
F9E8   LDX  #$0D       ; Logic Filenumber
```



```
F9EA   JSR  $FFC6      ; Device address of file
                         with number 13
F9ED   LDY  #$00       ; Counter to zero
F9EF   JSR  $FFCF      ; Enter Character (BASIN)
F9F2   JSR  $F7BC      ; And store in Address
                         determined by $AC/$AD (in
                         bank determined at $C6)
F9F5   INY             ; Raise offset for pointer
                         $AC/$AD
F9F6   BNE  $F9EF      ; Still not all characters
                         (1 Block)
F9F8   JMP  $FFCC      ; Close present I/O Channel
                         at IEC-Bus and Jump back
```

In this case the message "U1:13 0 01 00" is sent and the block at track 1, sector 0 is read into the buffer. This buffer is read and the bytes from $0B00 - $0BFF are stored. Then you continue the boot-call routine.

Next the contents of the addresses $E2C4 - $E2C6 are compared with the first three bytes that were read in. If one of these bytes is different, you leave the routine just as if the disk wasn't present. This means these three bytes contain the information determining whether an autostart will be performed or not! The three bytes in the operating system have the following values:

```
$E2C4 : $43 (67) "C"
$E2C5 : $42 (66) "B"
$E2C6 : $4D (77) "M"
```

The first three bytes from track 1, sector 0 together build the string "CBM".

If all the bytes are equal, the message "(CR) BOOTING" will be output on the screen. This uses the kernal routine PRIMM.

Then the contents of the addresses $0B03 - $0B06 are copied to the addresses $AC - $AF. The contents of the addresses $0B03 - $0B06 equal the bytes 3 - 6 of the block called from disk. We'll explain the function of these bytes later.

In the next section, locations $0B07 to the next zero (0) are interpreted in ASCII and the respective characters are output. The location where the zero appeared is stored in $9E.

Then the PRIMM routine is called again and the text "...(CR)" is output. The next two instructions copy the contents of address $AE to $C6. Additionally, the address $AE was loaded with the contents of $0B05, byte 5 of the block read from the disk. Also, the address $C6 was used before, in a previous routine. There the bank where the block was supposed to be loaded was determined by the content of $C6.

In the next section the content of $AF is called. If it is different than zero one will be subtracted from the value. If it is zero, you jump to another subprogram:

```
F9B3    LDX  $C2      ; Sector Number
F9B5    INX           ; Raise one
F9B6    CPX  #$15     ; Already Sector 21?
F9B8    BCC  $F9BE    ; No
F9BA    LDX  #$00     ; yes, then Sector = 0
F9BC    INC  $C1      ; And Track No. raise one
F9BE    STX  $C2      ; Sector No. store again
F9C0    TXA           ; Sector No. in ACCU
F9C1    JSR  $F9FB    ; Sector No.to ASCII code
```



```
F9C4    STA  $0100    ; Lower Nibble
F9C7    STX  $0101    ; Upper  Nibble
F9CA    LDA  $C1      ; Get Track No.
F9CC    JSR  $F9FB    ; Change to ASCII characters
F9CF    STA  $0103    ; Lower Nibble
F9D2    STA  $0104    ; Upper Nibble
- - - - - - - - - - - - - - - - - - - - - - - - -
F9D5  - F9FA          ;  Documented above
- - - - - - - - - - - - - - - - - - - - - - - - -
F9FB    LDX  #$30     ; Upper Nibble = "0"
F9FD    SEC           ; Set carry-flag for subtrac.
F9FE    SBC  #$0A     ; Is number 0-9?
FA00    BCC  $FA05    ; yes, then end
FA02    INX           ; No, then raise number
FA03    BCS  $F9FE    ; And try again
FA05    ADC  #$3A     ; Change low nibble to ASCII
FA07    RTS
```

The subprogram changes numbers to ASCII values; it only functions with two decimal numbers. Don't forget this when you use the routine for your own purposes.

So, you may ask, what is the function of the rest of the boot-call routine?

The content from $AF determines how many blocks are to be read from dis; $AC - $AD determines, which address those blocks are to be stored (you can't skip pages, the blocks are stored in sequence) and $AE determines the bank where the blocks are stored. The blocks go in sequence, track 1, sector 1 then track 1, sector 2 etc..

But there is more. Regardless of if there are more blocks read or not, you close the opened files in address $F945.

Then you repeat the counter. Which counter? You might remember that in $9E was stored the location where the string ended. Increment this counter and get the character at which the counter shows. If it is zero you leave this section of the boot-call routine. If it is not zero, increment the indicator by one. If zero, increment the indicator again and store it in $04.

In the next section you use the indicator $9E as an offset. At the location of the first zero goes the ASCII value for the colon and before that the content of $BF. The new counter is now stored. Then, the length of the file name is placed into the X-register. If it is zero, you jump over this part, because you know another program is supposed to be called. But, if it is not zero, it is incremented by two, because the colon and the contents of $BF have been added. Then more parameters are set:

Value	Comes After	Function
ACCU	$B7	Length of present file name
$9E	$BB	Low-byte address of file name
#0B	$BC	High-byte address of file name
#00	$CC	Bank-No. for LOAD/SAVE/VERIFY
#00	$C7	BANK-No. for Present file name

In address $F979 the accumulator is loaded with a zero to signal that a LOAD is required. The subprogram, called in the next instruction, is also used by VERIFY. In that case, the flag would be 1.

Then you load the program. You continue at address $F97E, whether a program was loaded or not.

Next in address $0004, the value $0B is stored then in $0002 the value $0F. Then you jump to the subprogram which is also used by the kernal routine. This call does the following: you select the bank, with the number in $0002. Then you jump to the address, determined through $0003/$0004. This program will be executed. Should this subprogram be left with a RTS, the program continues in the bank from which you called the program (in this case address $F989).

In address $F989 you clear the carry-flag. This indicates that the boot-call routine was left properly (if the disk drive did not report, the carry-flag would be set, see address $F8D3). The routine is left with a RTS.

10.1.2. USING THE BOOT - CALL

By examining the boot-call routine in the previous chapter, we found out that the first block on a disk, track 1/sector 0, must have the following byte assignment:

1. Byte: $43 (67) ASCII-Value for "C"
2. Byte: $42 (66) ASCII-Value for "B"
3. Byte: $4D (77) ASCII-Value for "M"
4. Byte: Low-byte of address, from which more blocks are stored (comes in address $AC)
5. Byte: High-Byte of this address (comes after $AD)
6. Byte: Bank-number, in which the blocks are to be stored (comes after $AE)

7. Byte: Number of blocks to read (comes after $AF)

---

8. Byte: To the first zero: Text, given out after BOOTING. Behind the first zero to the second: The name of the program, which should be loaded after loading the blocks. Behind the second zero up to the end program which shall be executed after loading.

Let's try out the new knowledge and create an "autostart" disk.:

```
10 OPEN 1,8,15
20 OPEN 2,8,13,"#"
25 PRINT #1,"B-F 0 1 0"
30 PRINT #1,"B-P 13 0"
40 PRINT #2,"CBM"
50 PRINT #1,"U2 13 0 1 0"
60 PRINT #1,"B-A 0 1 0"
70 CLOSE 2
80 CLOSE 1
```

If you are not familiar with disk drive commands here is an explanation of the program. First you open a data file, giving you access to the command channel of the disk drive. Then you reserve a buffer in the disk drive. This is because data cannot be written directly to the disk, it has to go in a data buffer. In line 25 a block is marked as free. Important! Any program on that block will be erased!

In line 30 a pointer is set to the first byte of this buffer. After this, the character sequence "CBM" is written into this buffer. The instruction in line 50 will write the data buffer to the disk on the block at track 1, sector 0. In line 60 this block is marked as "used" and the files are closed.

Now call the boot - call routine:

```
SYS DEC ("FF53"),8,8 (or SYS DEC ("F890"),8,8)
```

The disk drive starts and after a short time the text "BOOTING..." will be output on the screen. Then the computer will probably jump into the monitor. Why? Because we assigned the first three bytes of the block, only the bytes that permit the autostart. The rest of the block was probably filled with zeros, so the computer did not get any more text and didn't search for any more blocks or programs. After that the computer tried to execute a machine language program, but found only zeros. These zeros represent the machine language instruction BRK, so it jumped to the monitor (the normal reaction of the C-128 to the BREAK instruction).

Let's enter some more values into our "autostart" program.

```
10 DIM BY (255)
20 BY(0)=ASC("C"): BY(1)=ASC("B"): BY(2)=ASC("M")
30 BY(3)=0: BY(4)=0: BY(5)=0: BY(6)=0
40 PRINT CHR$(147);"TEXT BEHIND 'BOOTING':"
50 FOR ZA=7 TO 252
60 GET KEY A$: IF A$=CHR$(13) THEN 80
70 BY(ZA)=ASC(A$): NEXT ZA
80 BY(ZA)=0
90 ZA=ZA+1: BY(ZA)=0
100 ZA=ZA+1: BY(ZA)=96
110 OPEN 1,8,15
120 OPEN 2,8,13,"#"
125 PRINT #1,"B-F 0 1 0"
130 PRINT #1,"B-P 13 0"
140 FOR ZA=0 TO 255
150 PRINT #2,CHR$(BY(ZA));
160 NEXT ZA
170 PRINT #1,"U2 13 0 1 0"
```

```
180 PRINT #1,"B-A 0 1 0"
190 CLOSE 2
200 CLOSE 1
```

Here you first create an array with 256 elements. The first seven bytes of this array (later written to the autostart block) are set, so that the autostart can be identified without loading any more blocks. Then you can enter any text, but no more than 246 characters. That happens in the following lines:

Line 80:	End of string to output
Line 90:	End of program name (0 characters, no program is loaded)
Line 100:	Program to execute (the command: RTS)

Then the bytes are written to disk. Start the boot-call routine again after running this program. Your text is printed and you are returned to BASIC.

Now for a program using the autostart. When you run the following program, the block at track 1, sector 0 will be written so that it starts a (BASIC) program with name "HELLO" automatically. The assignment of the bytes is as follows:

Byte 0 - 6	: Bits for autostart routine
Byte 7 - 26	: Text to print out
Byte 27	: Zero as end mark
Byte 28 - 32	: Name of program to load ("HELLO")
Byte 33	: Zero as end mark
Byte 34 - 52	: Machine Program to Execute (RUN)

Now the complete "autostart" program:

```
10 OPEN 1,8,15
20 OPEN 2,8,13,"#"
25 PRINT #1,"B-F 0 1 0"
30 PRINT #1,"B-P 13 0"
40 FOR ZA=0 TO 52
50 READ A
60 PRINT #2.CHR$(A)
70 NEXT ZA
80 PRINT #1,"U2 13 0 1 0"
90 PRINT #1,"B-A 0 1 0"
100 CLOSE 2
110 CLOSE 1
1000 DATA 67,66,77,0,0,0,0,
1010 DATA 13,40,67,41,32,49,57,56,53,32,66
1020 DATA 89,32,65,66,65,67,85,83,32
1030 DATA  0
1040 DATA 72,69,76,76,79
1050 DATA  0
1060 DATA 162,2,189,50,11,157,74,3,202,16,247
1070 DATA 169,3,133,208,96,82,213,13
```

The assembler program (data in lines 1060 & 1070) is as follows:

```
      1  LDX #$02          ; Counter
Loop  2  LDA table,X       ; Get Character
      3  STA $034A,X       ; And in Keyboard Buffer
      4  DEX               ; One more Character?
      5  BPL Loop          ; Yes
      6  LDA #$03          ; Amount of characters
      7  STA $D0           ; In Keyboard Buffer
      8  RTS               ; End of Routine
Table 9  $52, $D5, $0D     ; ASCII for R,Shift+U
                           ; And RETURN
```

The machine language routine also stores RUN in the keyboard buffer.

After you have modified your disk with the above program, call the boot-call routine again or reset your computer. The text will be output, but then you receive a "FILE NOT FOUND" message. Of course, this is because the file name "HELLO" is not on the disk. What could a "HELLO" program look like? Like this for example :

```
10 PRINT CHR$(147)
20 PRINT"ENTER CHOICE:"
30 PRINT: PRINT"(1) DIRECTORY"
40 PRINT: PRINT"(2) LOAD PROGRAM"
50 PRINT: PRINT"(3) DISC-OPERATION"
60 PRINT: PRINT: PRINT"(4) FINISH"
70 GET KEY A$: A=VAL(A$)
80 ON A GOTO 100,200,300,400
90 GOTO 70
100 PRINT CHR$(147)
110 DIRECTORY
120 PRINT: PRINT"- PRESS ANY KEY TO CONTINUE-"
130 GET KEY A$: GOTO 10
200 PRINT CHR$(147)
210 INPUT "NAME OF PROGRAM"-,NA$
220 LOAD NA$,8,1
300 PRINT CHR$(147)
310 INPUT "COMMAND"-,BF$
320 OPEN 1,8,15
330 PRINT#1 ,BF$
340 CLOSE 1
350 GOTO 10
400 PRINT CHR$(147)
410 PRINT"ARE YOU SURE (Y/N)?"
420 GET KEY A$: IF A$="N" THEN 10
430 IF A$<>"Y" THEN 420
```

But there are many other possibilities. You could change the name of the program called from the boot-call to automatically load your favorite game or the program you work with most.

The boot-call routine can also load more blocks. You can load a different operating system, a different language, etc, etc..

The respective data has to be in sequence on the disc (starting at track 1, sector 1) and is loaded in the memory in sequence. The bytes that determine this loading are the bytes 3 - 6 on block track 1, sector 0 (see above).

No doubt this routine will be used very often, probably by every commercial program written for the C-128. But you can use it for your own purposes.

10.2. AUTOSTART BY CARTRIDGE

Who doesn't know them, those little boxes, which are inserted in the expansion port. They are called cartridges. This automatic start by cartridge has two advantages:

1. It is user friendly; the user doesn't have to enter inconvenient SYS commands.
2. The cartridge is harder to copy.

But how does this autostart work? In the C-64 it worked as follows.

When the computer was reset it examined the memory locations $8004 - $8008. If it found the ASCII values of the character sequence "CBM80", it did an indirect jump to the address pointed to in $8000/$8001; which means it called the contents of $8000 as the low byte of the address and

the contents of $8001 as the high byte. Then the computer jumped to this address.

The C-128 works about the same. After a reset, the computer jumps to a subprogram that compares some bytes with fixed values. You will find this routine in memory from $E1F0 - $E241:

```
E1F0  LDX #$F5
E1F2  LDY #$FF
E1F4  STX $C3
E1F6  STY $C4
E1F8  LDA #$C3
E1FA  STA $02AA
E1FD  LDY #$02
E1FF  LDX #$7F
E201  JSR $02A2
E204  CMP $E2C4,Y
E207  BNE $E224
E209  DEY
E20A  BPL $E201
E20C  LDX #$F8
E20E  LDY #$FF
E210  STX $C3
E212  STY $C4
E214  LDY #$01
E216  LDX #$7F
E218  JSR $02A2
E21B  STA $0002,Y
E21E  DEY
E21F  BPL $E218
E221  JMP ($0002)
- - - - - - - - - - - - - - - -
E224  LDA #$40
E226  STA $FF00
E229  LDA #$24
E22B  LDY #$E2
E22D  STA $FFF8
E230  STY $FFF9
E233  LDX #$03
E235  LDA $E2C3,X
```



```
E238  STA $FFF4,X
E23B  DEX
E23C  BNE $E235
E23E  STX $FF00
E241  RTS
```

First the vector at ($C3/$C4) points to address $FFF5. Then it jumps to a routine which calls any address from any bank. Here it calls the contents of address $FFF7 in Bank 1. This byte will be compared with the content of address $E2C6. If these two bytes not identical it jumps to address $E224. If they are equal, it tests the next bytes. The following addresses are compared:

$FFF5 with $E2C4 : $43 (="C")
$FFF6 with $E2C5 : $42 (="B")
$FFF7 with $E2C6 : $4D (="M")

So, unlike the C-64, only three bytes are tested, which are enough. These three bytes equal the string "CBM", which is also used in the autostart routine of the disk drive. If these are identical, you call the contents of the addresses $FFF8 and $FFF9 and store it in the address $0002 and $0003. Over this vector an indirect jump is carried out. To execute an autostart, the following bytes must be assigned as follows:

$FFF5: $43 ("C")
$FFF6: $42 ("B")
$FFF7: $4D ("M")

$FFF8: Low-Byte determined jump - address
$FFF9: High-Byte determined jump - address

The routine jumped to in case of an error must be in Bank 15. Jumping to a routine at $A000 in the RAM of Bank 0 doesn't work, because the BASIC interpreter is in Bank 15. But you can jump to a routine in the cassette buffer, which is shared by the respective memory configuration.

Here is a short explanation before we show some examples of the uses of the cartridge autostart. When the bytes are not equal you don't jump to BASIC as in the C-64.

At address $E224 the first two instructions select the memory configuration for Bank 15. The only difference is that you select RAM Bank 1 instead of RAM Bank 0 (remember: RAM BANK 15 turns on all the system ROMs). Then you store the following values in the addresses $FFF5 - $FFF9:

```
$FFF5:  $43 ("C")
$FFF6:  $42 ("B")
$FFF7:  $4D ("M")
$FFF8:  $24
$FFF9:  $E2
```

That's why an autostart is performed after every reset, but only with the normal reset routine.

Now take a look at these bytes: enter the monitor and enter `M 1FFF5`. As you can see, the addresses $FFF5 - $FFF9 are assigned with the values above. Now enter your own address for the autostart: change byte $24 to $4D. Now change byte $E2 to $FF. You now will jump to address $FF4D instead of address $E224 when a reset occurs. This

address initializes the C-64 mode. Try it out: push the reset button. After a short time (the 80 column screen might start to blink) you are in the C-64 mode.

And another thing: switch the computer off for a very short time and then back on. As you can see, the computer comes on in the C-64 mode! No, don't be afraid; it is not broken. In the C-128 the RAMs don't clear as fast as in the C-64. That's why the values from $FFF5 - $FFF9 stayed in the computer. Turn your computer off for a little bit longer and you will come back in the C-128 mode. The longer "life" of the RAMs has two advantages: first, a power loss does not affect the operation of the computer; and second, your programs stay in the computer after you switched it off for a short time.

Of course, you can also start BASIC programs with a reset. You have to change the vector at $FFF8 - $FFF9 to your own routine, which simulates the RUN command. In the C-64 it was possible with the following assembler commands:

```
JSR $A659    ;CHRGET On Program start + CLR
JMP $A7AE    :Interpreter loop
```

We can accomplish the same thing on the C-128 in a different manner. Because the C-128 has a keyboard buffer, we store the string RUN and RETURN in the buffer and then quit the program. This starts the BASIC program in memory. An assembler listing would look as follows:

```
        LDX #$02            ;Counter
Loop    LDA table,X         ;GET CHARACTER
        STA $034A,X         ;In the Keyboard Buffer
        DEX                 ;One more Character?
        BPL Loop            ;Yes
        LDA #$03            ;Number of characters
        STA $D0             ;In the Keyboard Buffer
        RTS

Table $52, $D5, $0D         ;ASCII for R,SHIFT+U
                            ;And RETURN
```


CHAPTER 11

C-128 MEMORY

11.1 IMPORTANT ADDRESSES

The C-128's new BASIC is one of the most complete BASICs available on a microcomputer today. But with the C-128 you can do a lot more than the standard BASIC commands allow--without using machine language. The key is the zero page, which is made up of the first 256 memory locations. They contain a lot of pointers and addresses used by the operating system. Modifying these pointers and addresses will let you create your own special operating system, allowing you to do things not possible with normal BASIC commands.

The Commodore 128 has several zero pages, because its operating system is more complex than those used by previous Commodore computers.

The following pages contain interesting memory locations and suggestions for using them. Don't be afraid of destroying your computer when modifying the memory locations. About the worst you can do is lock up the computer, but usually a reset will put everything back to normal. If that fails, then you'll have to turn off the computer and turn it back on.

Beside the addresses of the C-128 you will find the corresponding C-64 addresses in parenthesis. This means you can translate your old C-64 programs to the new C-128. Both modes are fairly compatible.

Here are some interesting memory locations and their functions.

(decimal location) (comment)

45 - 46 (43-44) BASIC Start (Bank O)

This memory location contains the start address of where the BASIC programs are stored in the memory (in Low-byte/High-byte formula, Bank O). The command:

```
PRINT  PEEK(45)+256*PEEK(46)
```

prints this address. You can also move the start-of-BASIC address. To do that type:

```
POKE45, lo:POKE 46, hi:POKE (lo+256*hi)-1,0:NEW
```

This command sequence has to be entered in direct mode. `lo` and `hi` are the low-byte/high-byte for the new start address. These bytes are calculated as follows:

```
HI=INT(address/256): LO=address-(256*HI)
```

You erase the program in the memory if you shift the start of BASIC. Remember that you usually can't shift the start-of-BASIC. This memory area is reserved for the operating system!

47 - 48 (45-46) VARIABLE Start (Bank 1)

These two bytes point to the start of variable-memory (in bank 1). This pointer can be read or manipulated like the BASIC start (above).

59 - 60 (57-58) Current BASIC-Line Number

The current BASIC line number is stored in these bytes. Therefore the readout of these addresses makes sense only in the program mode:

```
10 PRINT "This line #";PEEK(59)+256*PEEK(60);
"in use!"
```

65 - 66 (63-64) Current DATA line number

This pointer is interesting if you use the commands READ and DATA in your programs. It contains the line number of the line where you READ the last DATA element; the following program demonstrates this:

```
 10 READ A: IF A=1 THEN END
 20 PRINT"THE ELEMENT A IS IN LINE ";
 30 PRINT PEEK(65)+256*PEEK(66);"!"
 40 GOTO 10
 50 DATA 3,6,4,8,4,6,2,
 57 DATA 33,6,4,2,4,2,4,
 99 REM TEST PROGRAM
167 DATA 3,7,4,9,6,0,0,
190 DATA 5,7,5,-1
```

This address is also useful in finding errors. You find the error with the pointer that gives you the line number where the error occurred. If your program contains a string and your program allows numerical DATA only, the following BASIC program is handy:

```
10 READ A$:IF ASC(A$)<50 OR ASC(A$)>60 THEN1000
20 A=VAL(A$):IF A=-1 THEN END
30 PRINT A,:GOTO 10
40 DATA 4,3,7,54,3,5,2,444
50 DATA 3,5656,a,3,d,4,2,2,2,:REM Error line
60 DATA 4,6,3,5,6,
70 '
1000 REM ERROR MESSAGE
1010 PRINT"IN LINE";PEEK(65)+256*PEEK(66);
1020 PRINT"ERROR FOUND. ONLY NUMBERS ";
1030 PRINT"ARE ALLOWED !"
1040 PRINT"CHANGE THIS!":END
```

208 (198) Number of keys pressed

This pointer contains the number of characters you entered with the keyboard.

```
POKE 208,0
```

clears the keyboard buffer.

213 (203) Keyboard reading

The value of the key actually pressed is stored in this address. It has the value 88 if no key is pressed.

```
10 PRINT CHR$(147);
20 PRINT PEEK(213):GOTO 10
```

prints the value of the key pressed.

215 (---) 40/80-column Flag

This address shows you which screen is active. Bit 7 is set in the 80-column-mode, PEEK(215) = 128:

```
10 A=PEEK(215)
```

The following addresses refer to the 40-column and 80-column screens. These memory locations are revised when switching to the other screen. Therefore, you always find the value for the active screen in these locations.

241 (646) Actual character color

There are 16 colors available for characters on the screen. Normally control-characters are used to turn on the colors. But this is impractical and hard to follow in program listings. Often it is easier to use this memory location. It contains the value of the actual character color (0-15) and is very easy to manipulate.

```
10 A=INT(RND(1)*16) :REM Random number 0-15
20 POKE 241, A:Set character
30 PRINT"*";
40 GOTO 10
```

Here is a short summary of the color combinations:

0	: Black	1	: White	2	: Red
3	: Cyan	4	: Violet	5	: Light green
6	: Blue	7	: Yellow	8	: Light red
9	: Brown	10	: Light red	11	: Grey 1
12	: Grey 2	13	: Light green	14	: Light blue
15	: Grey 3		(Only for a 40-column screen)		

0	: Black	1	: Grey 2	2	: Blue
3	: Light blue	4	: Green	5	: Light green
6	: Grey 1	7	: Cyan	8	: Red
9	: Light red	10	: Dark violet	11	: Violet
12	: Brown	13	: Yellow	14	: Grey 3
15	: White		(Only for a 80-column screen)		

You don't have these colors if you work with a monochrome screen. The colors are transformed into different shades of green, grey, etc. There are only 5 steps of brightness available for all 16 colors. You have to use colors with different brightness if you want to use them on a monochrome screen. Here is the order of brightness:

1. Black
2. Red, Blue, Brown, Grey 1
3. Violet, Orange, Light red, Grey 2, Light blue
4. Turquoise, Yellow, Light green, Grey 3
5. White

The 80-column screen has additional features. The bits 4-7 have the following functions:

Bit 4 : Flashing
Bit 5 : Underline
Bit 6 : Reverse
Bit 7 : 2nd set of characters

Bits 4 and 5 only affect the next PRINT statement after the POKE command.

```
POKE 241, 15+2^4 : PRINT "TEST"
```


Puts a white, flashing "TEST" on the screen.

```
POKE 241, 2+2^5+2^7 : PRINT "HELLO"
```

The above prints a blue-underlined "HELLO". This means you can mix the different functions! The functions "reverse" (bit 6) and "2nd set of characters" (bit 7) are constant functions; they refer to all following PRINT orders and to the direct mode. A RVS-mode like this cannot be turned off by CRTL+9!

243 (199) RVS-Flag

Corresponding to address 241, this address determines the kind of characters that are to be shown on the screen. Reverse or normal (there is also a control-character for this). 0 = normal, 1 = reverse:

```
10 POKE 243,0:PRINT "NORMAL and.....";
20 POKE 243,1:PRINT ".....REVERSE!"
```

244 (212) Quote-Mode-Flag

Quote-mode means control-characters are not active between quotation marks, but instead display control characters on the screen. If you turn the quote-mode on (=1) the following control characters in a PRINT statement are ignored and their control characters are printed on the screen.

```
POKE 244, 1 : PRINT"{3crsrdwns}{rvson}HELLO !"
```

247 (657) C=/SHIFT Stop-Flag, CRTL-S

You know how to change the character set by pressing the <SHIFT> key and the Commodore key (<C=>). You can also do this while a program is being executed. It doesn't disturb the running program, only what you see on the screen. This can be disabled:

```
POKE 247,128    (locked)
POKE 247,0      (normal)
POKE 247,64     (disable CTRL-S)
```

248 (---) Scrolling Stop-Flag

Once you have reached the lowest screen-line and print on the next line, the screen contents are shifted one row up. You can eliminate this effect:

```
POKE 248, 128 (Scrolling off)
POKE 248, 0     (Normal)
```

249 (---) Beep-Tone Stop-Flag

Here it is possible to disable the beep-tone. It is generated by:

```
PRINT CHR$(7)   (Program-mode)   or
CTRL+G          (Direct-mode)
```


Try this:

```
POKE 249, 128   (off)
POKE 249, 0     (Enables the beep-tone again)
```

By the way: SYS 51602 generates the tone. But you can do more with the tone than just give a signal.

```
FOR X=0 TO 100:SYS 51602:NEXT X
```

This gives you a longer tone, depending on the X-value.

All the following addresses are common and not tied to the active screen. They are not in the true zero page.

842-852 (631-640) Keyboard-Buffer

These ten bytes are the buffer for keys entered from the keyboard. They can be used for many programming applications. There are too many possibilities to describe here, but we'll give you one example:

```
10 POKE 842, ASC("L"): POKE 843,ASC("I")+128
20 POKE 844,13
30 POKE 208, 3
40 END
```

1024-2023 (1024-2023) 40-Column Screen Memory

These 1000 bytes contain the 40-column screen. This area is not used if you work with a 80-column screen; you can use it for own machine language programs. Otherwise:

```
POKE 1024 +X + 40 * Y,Z

X=Column
Y=Row
Z=Character-code
```

places a character on the 40-column screen.

2592 (649) Maximum Length of the Keyboard-Buffer

The length of the keyboard-buffer is defined with this address.

```
POKE 2592,0    (turns off the buffer, keys are no longer stored)
POKE 2592,10   (normal)
```

The length of the keyboard-buffer should not exceed the maximum 10 because this is the maximum of available space reserved in memory!

2593 (---) CTRL-S Flag

A running program stops when <CTRL-S> is pressed, and continues when any other key is pressed. Many larger computers have this function. It is very useful if you only want to interrupt the program

for a short while to have a closer look (or have a quick breakfast). But you can use this address in a completely different way. You simulate the <CTRL-S>. The program stops and waits for the next key pressed.

```
10 PRINT "PRESS ANY KEY "
20 POKE 2593,1
30 PRINT "OK"
```

2594 (650) REPEAT-Flag

The C-128 has a REPEAT-function for all keys. This means if you depress a key for a short period of time the character will be printed continuously.

```
POKE 2594, 0      (space bar and cursor keys)
POKE 2594,64      (no keys with repeat function)
POKE 2594,128     (normal)
```

2595 (651) REPEAT-delay

The computer continues normal printing after a short delay. The delay time can be adjusted with this address, so you can choose a long delay. No delay makes sense as a controller (with a GET).

POKE 2595,X
X=delay time

2598 (---) VIC Cursor-Mode

This address controls the flash of the cursor, but only the cursor controlled by VIC chip. This means the address will work only in 40-column mode.

Bit 6 : 1=solid
0=flash

```
POKE 2598,0      (Cursor flashes, normal)
POKE 2598,64     (Cursor doesn't flash)
```

2599 (204) VIC Cursor on/off-Flag

This address works only with the 40-column screen. You can activate the cursor with this address. The following demonstrates what you can do with this address:

Bit 0 : 1=Cursor on
0=Cursor flash

```
10 POKE 2599, 0:REM CURSOR ON
20 PRINT  "HELLO!";
30 FOR  T=1 TO 5000:  NEXT T
40 PRINT  "THAT'S IT!"
50 END
```

This can be used effectively with the command GETKEY. This command waits for a key press and then stores the character. It is different from the INPUT command in that the cursor does not flash to get the user's attention. Here is an example:


```
10  POKE 2599, 0
20  GETKEY A$
30  (. . . )
```

This works with the GET command too.

2603 (---) VDC Cursor-Mode

You can influence the cursor of the 80-column screen at address 2598. The bits of this address mean:

Bit 0 1=solid
Bit 2-3 Display in pixel rows
Bit 4-5 Cursor off
Bit 6 1=flash

```
POKE 2603, 2^0+2^1+2^2
```

This activates the underline cursor. The thickness of the cursor depends on the bits 2 and 3.

2604 (---) VIC Pointer to Screen-RAM / character set

The content of this address are transferred automatically to address 53272 of the VIC. The contents determine the start addresses of the screen-RAMs and the character generator.

2606 (648) VDC High-Byte of the screen RAMs

This address is responsible for the 80-column screen again. Here you can shift the screen-memory with the registers 12 and 13 of the VDC.

2607 (--) VDC High-Byte of the color memory (attribute-RAM)

The High-Byte of the 80-column attribute-RAM is stored here. It is related to the previous address because you can move the color-memory in the 80-column mode.

2619 (648) VIC Hi-Byte of the screen RAMs

This address has the same function as the address 2606 at the VDC. You find the start address of the 40-column screen-RAM here.

2816-3327 (828-1029) Cassette buffer

This relatively large area is used by the system only if you work with the Datasette. This area is not used if you use the disk drive. It offers room for machine language routines.

4096-4105 (---) Assignment-length of the function keys

Each of these ten bytes is reserved for one of the ten function keys (<F1-8>, <HELP> and <SHIFT+RUN/STOP>). The length of the function key assignment is stored in these bytes. Set the corresponding byte to zero if you want to turn off the assignment of a function key (for example: if you want to read a key in your own program).

```
POKE 4096,0   (erases assignment of F1)
```

4624-4625 (55-56) Pointer pointing to end-of-BASIC

The highest address of a BASIC program is stored in these bytes (in bank 0):

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

The size of your program is easily determined:

```
PRINT (PEEK(4624)+256*PEEK(4625))-
(PEEK(45)+256*PEEK(46))
```

In case you want to write your own machine language program at the end of the memory in bank 0 (BASIC-memory) you can limit the BASIC-memory with these two addresses so that your machine language program is protected from being overwritten.

11.2. JUMP TABLE

Important to every machine language programmer is a solid understanding of the operating system. This allows him/her to create shorter programs in the least amount of time. He/she simply takes prepared routines, instead of writing them. We don't want to write a second *COMMODORE-128 INTERNALS*, but we'll choose some popular routines and show you how to use them.

There are several jump commands with powerful routines included in both the BASIC interpreter and the operating systems of the C-128 . Let's go to the most important and best known routines first:

11.2.1 KERNAL

The C-128, like every other Commodore computer, has a Kernal jump table at the end of the ROM. It provides easy access to important kernal routines and allows quick program conversions to other Commodore computers. This table is greatly extended compared to the C-64 or VIC-20. We want to introduce the section of the table from $FF4D-$FF7F.

There are some routines designed exclusively for the C-128. There is, for example, the C-64-mode routine that turns the C-128 into a C-64. There are also some routines to read different memory banks.

Another interesting aspect is that Commodore has assigned a byte with the value zero between the old and new kernal table ($FF80). This is the number of the Kernal table. There is also a new address directly in front of the addresses for NMI, Reset and the IRQ. This address, at $FFF8/$FFF9, points to address $E224 and is called C-128 mode. Let's examine the secrets of the new jump table.

Kernal-address:	$FF4D (65357)
Name:	C64MODE
Real jump-address:	$E24B (57931)
Function:	Transforms the C-128 into the C-64

The name of this function is self-explanatory.

Kernal address:	$FF50 (65360)
Name:	DMA-CALL
Real jump address:	$F7A5 (63397)
Function:	access to the DMA-controller

You can control the DMA-controller (Direct Memory Access) with this routine. This routine becomes interesting with versions of the C-128 with more than 128K RAM, because it works only with the extended RAM.

Kernal-address: $FF53 (65363)
Name: BOOT-CALL
Real jump-address: $F890 (63632)
Function: Load and start program from disk

With this routine you can load and start a program immediately from the disk. It is possible to start programs after turning on the computer, because this routine is called with a reset.

Kernal-address: $FF56 (65366)
Name: PHOENIX
Real jump-address: F867 (63591)
Function: Start external-ROM's

You can start external ROM's (cartridges) with this routine. Immediately after PHOENIX is done, it jumps to the BOOT-CALL routine. This routine is also called with a reset.

Kernal-address: $FF59 (65369)
Name: LKUPLA
Real jump-address: $F79D (63389)
Function: Gets entry for file number

If you call this routine you enter the file number in the accumulator and you get back:

Accumulator: File number
X-Register: Device address
Y-Register: Secondary address

It returns with a set Carry-Flag if the file number was not found. Otherwise this Flag is reset. This routine uses a different routine at $F202. This routine looks to see if a given file number already exists in the X-Register. It returns with a reset Zero-Flag if the file number already exists. Otherwise the Zero-Flag is set.

Kernal-address: $FF5C (65372)
Name: LKUPSA
Real jump-address: $F786 (63366)
Function: Look up secondary-address

This function is comparable to the last one. But the secondary-address is in the Y-Register. This routine does not look for several equal files with the same secondary-address. It looks up the entry of the first found file belonging to the secondary-address. After the return, all three registers contain the same values as the routine before:

Accumulator: File number
X-Register : Device address
Y-Register : Secondary address

The flags are set the same way.

Kernal address: $FF5F (65375)
Name: SWAPPER
Real jump-address: $C02A (49194)
Function: Switch between 80 / 40 column-screen

The screen is switched if you execute this routine. If you were in the 40 column mode and this routine is called, your input now appears on the 80-column-screen. We will now have a closer look at this routine. Beginning with address $FF5F it jumps to address $CD2E (why make it easy if you can make it complicated?) At $CD2E the following is executed:

```
CD2E    LDX #$1A
CD30    LDY  $0A40,X
CD33    LDA  $E0,X
CD35    STA  $0A40,X
CD38    TYA
CD39    STA  $E0,X
CD3B    DEX
CD3C    BPL  $CD30
CD3E    LDX  #$0D
CD40    LDY  $0A60,X
CD43    LDA  $0354,X
CD46    STA  $0A60,X
CD49    TYA
CD4A    STA  $0354,X
CD4D    DEX
CD4E    BPL  $CD40
CD50    LDA  $D7
CD52    EOR  #$80
CD54    STA  $D7
CD56    RTS
```

In the first section (up to $CD3D) the memory-blocks $E0-$FA and $0A40-$0A5A are exchanged. In locations $0A40 to $0A5A are

stored the screen-values for the screen not currently being used (cursor position, insert-mode-flag, quote-mode-flag, etc.). The routine continues in the same manner when you leave the current screen to switch back to the other screen. Try it out. Get in the quote-mode by entering a quotation mark. Press the cursor keys a few times. Now enter ESC+X (press slowly, one after the other) and enter something on the other screen. Now change back to the other screen and press the cursor keys a few times. As you can see, you are still in the quote-mode. Change to the other screen and enter:

```
POKE DEC("0A54"),0
```

Go back to other screen and press a cursor key; no control-character is printed, but the cursor moves. Why? Because the above POKE sets the quote-mode-flag to zero. This register was copied when you switched back to the other screen and the quote-mode was gone. Therefore, you can predetermine with POKEs how your other screen will look after you switch back. Also, have a look at the addresses in the description of the zero page ($E0-$FA). We'll give you a formula to calculate an address on the closed screen in relation to the one in use:

`new address=old address +4200` (fantastic isn't it ?)

Let's go back to the SWAPPER routine. The next eight commands exchange two more blocks.

a) $0354 - $035E (Tab -Stops) with $0A60-$0A6A
b) $035F - $0361 (Line-links) with $0A6B-$A6D

Between the addresses $CD50-$CD55 the screen is changed. Therefore bit 2 of $D7 is inverted and this ends the routine.

Kernal-address:	$FF62 (65378)
Name:	DLCHR
Real jump-address:	$CO27 (49191)
Function:	Initialize the VDC character-generator

This routine copies the VDC's character-generators for 40 column to that for a 80 column. The character-generator of the VDC doubles its size (to 8K) because it copies eight zero-bytes between each character.

The next seven routines are the most important routines for accessing different banks.

Kernal-address:	$FF68 (65384)
Name:	SETBAK
Real jump-address:	$F73F (63295)
Function:	Verify the bank for SAVE / LOAD / VERIFY

This routine has only 3 commands:

```
$F73F   STA   $C6   ;.  bank number, where the program is located.
$F741   STX   $C7   ;   bank number, where the file name is located.
$F743         RTS
```

Vector $BB / $BC contain the address of the file names.

Kernal-address:	$FF6B (65387)
Name:	GETCFG
Real jump-address:	$F7EC (63468)
Function:	Get configuration number.

You have to load the X-register with the number of the desired bank (0-15) before you jump to this routine. You get back a byte in the accumulator. This byte corresponds to the assignment of the MMU's configuration register, so that the desired bank can be chosen. You only have to store it in the address $D500 ($FF00). The chapter about the MMU tells you which memory configurations follow the single bank numbers.

Kernal-address:	$FF6E (65390)
Name:	JSRFAR
Real jump-address:	$02CD (717)
Function:	Starting a subprogram in any bank

This function lets you call a subprogram in any bank. The computer returns to the bank that originally called the routine after it finishes the subprogram. Before calling (you must use JSR) you have to load the following memory locations and registers with the corresponding contents:

$0002:	Number of the desired bank
$0003:	High-byte of the desired address
$0004:	Low-byte of the desired address
$0005:	Desired status-register (to turn off the IRQ, turning on the decimal -modes, etc.)

$0006:	Desired accu
$0007:	Desired X-register
$0008:	Desired Y-register
$0009:	Number of the calling bank

The following addresses contain the new register contents after the return:

$0005:	Status
$0006:	Accu
$0007:	X-register
$0008:	Y-register
$0009:	Stack-pointer

What you do with these values is up to you. You also can load the memory location $0006 into the X-register and save a transfer-command.

Kernal-address:	$FF71 (65393)
Name:	JMPFAR
Real jump-address:	$02E3 (739)
Function:	Jump to a bank

With this routine a jump to any bank is executed. This routine is also used by JSRFAR. Therefore the call is very similar:

$0002:	Number of the desired bank
$0003:	High-byte of the desired address
$0004:	Low-byte of the desired address

$0005:	Desired status-register
$0006:	Desired accu
$0007:	Desired X-register
$0008:	Desired Y-register

Here the call has to be made with the JMP command.

Kernal-address:	$FF74 (65369)
Name:	INDFET
Real jump-address:	$F7D0 (63440)
Function:	Get a byte from any bank

With this routine you can get the contents of any bank. Call the routine as follows:

$02B9: Low-byte of the vector pointing to the desired memory location.The vector has to be initialized (it has to contain the desired address split in Low-and High-byte). This vector must be in the zero page.

Accu: Byte, where you want to enter the desired address

X-Reg: Number of the desired bank

Y-Reg: Offset that is eventually added to the address (normally zero, if you don't ask for an index).

Kernal-address: $FF77 (65399)
Name: INDSTA
Real address jump: $F7DA (63450)
Function: Stores a byte in any bank

This is the opposite of INDFET. To store a byte the following addresses and registers have to be set:

Accu: Low-byte of the vector
(it points to the desired address like INDFET).
X-Reg: Number of the desired bank
Y-Reg: Offset

Kernal-address: $FF7A (65402)
Name: INDCMP
Real jump-address: $F7E3 (63459)
Function: Compare accumulator and byte in any bank

This function compares the contents of an address with the contents of the accumulator in any desired bank. The usual flags for comparing are set (Zero-Flag=1 if both are equal, etc.) The following memory locations have to be set before calling:

$02C8: Vector low-byte pointing to the desired address
Accu: Character to be compared with the contents of the addressed memory location.
X-Reg: Number of the desired bank

Kernal-address: $FF7D (65405)
Name: PRIMM
Real jump-address: $FA17 (64023)
Function: Print a string

The old part of the Kernal contains a routine to print out a single character (BSOUT). You can print out a string with this routine in a simple but clever way. The characters to be printed are simply placed behind the call to the routine. This results in the following:

```
JSR    $FF7D
(any text, a zero for an end mark.)
RTS
```

But how can this be? How does the computer reach the finishing RTS? Let's have a look at the routine again:

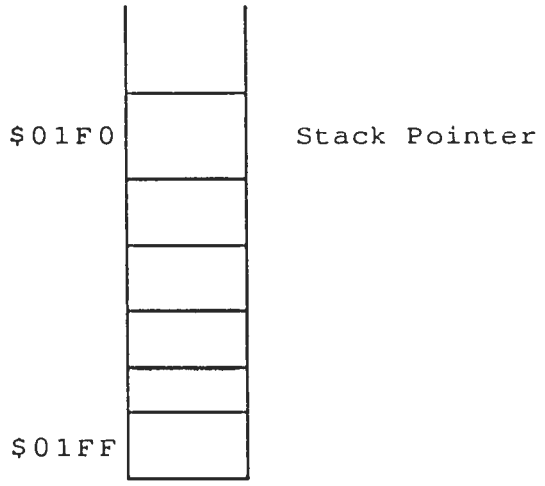
```
FA17     PHA
FA18     TXA
FA19     PHA
FA1A     TYA
FA1B     PHA
FA1C     LDY #$00
FA1E     TSX
FA1F     INC $0104,X
FA22     BNE $FA27
FA24     INC $0105,X
FA27     LDA $0104,X
FA2A     STA $CE
FA2C     LDA $0105,X
FA2F     STA $CF
FA31     LDA ($CE),Y
FA33     BEQ $FA3A
FA35     JSR $FFD2
FA38     BCC $FA1F
```

```
FA3A      PLA
FA3B      TAY
FA3C      PLA
FA3D      TAX
FA3E      PLA
FA3F      RTS
```

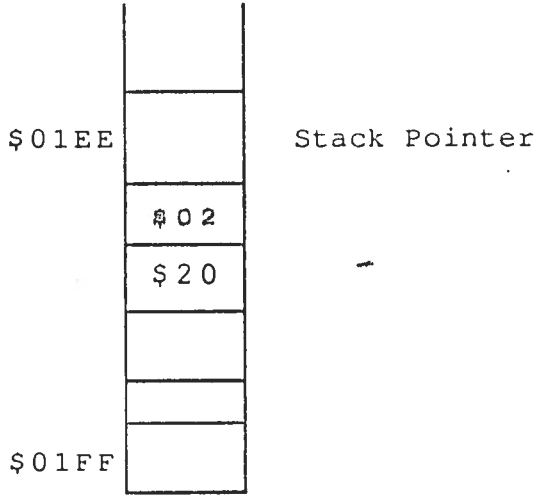
First the contents of the accumulator and the X-Registers are stored in the stack, then the Y-Register, which is later used with indirect indexed address, as an offset, is stored in the X-Register.

Now the important part of the routine begins. The stack pointer points to the next free location on the stack. To understand what happens you have to know how the stack is built as well as its position. In the C-128 it is located in memory from $0139 to $01FF. The stack pointer first points to $01FF and becomes smaller as more elements are put on the stack.

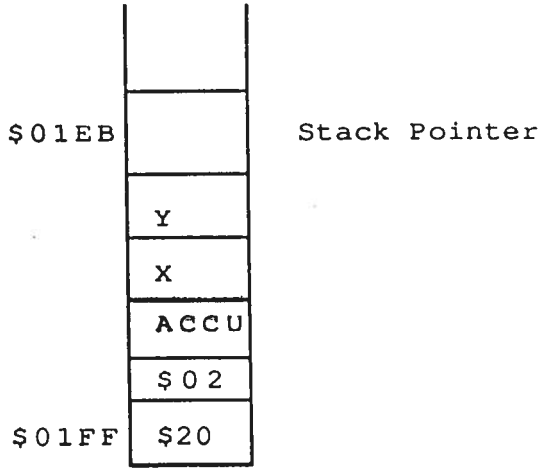
We'll simulate how the stack reacts when you call it with this routine. Let's say the stack pointer contains the value $F0 and as a result points to the address $01F0. The stack looks something like the picture on the next page:



Now the routine is started. First one number is placed on the stack, for the hi-byte of address $2000. The program counter (this register has the pointer, and the next byte of the word) places the next word on the stack, first the hi-byte then the low byte. This results in the following picture:

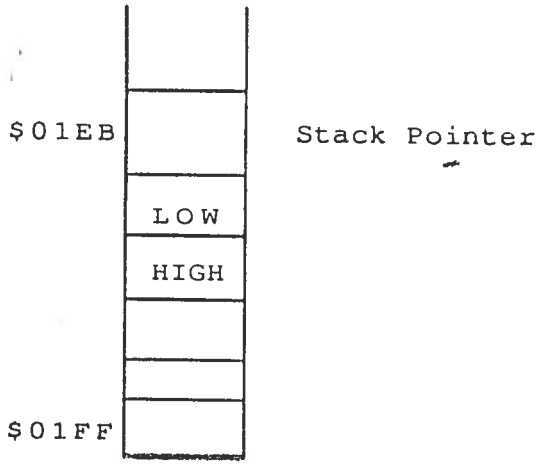


Then the routine fills the next three memory registers and the stack looks like this:



At address $FA1F one byte of the stack is pointed to. But which one? Let's calculate it. The stack pointer (and therefore the X-register) contain the value $EB. When you add $0104 you get $01EF. The low-byte of the return address is raised by one. The BRANCH command checks if the low-byte is zero. In this case, the byte at address $0105 + $EB=$01F0, (which is the high-byte of the return address) is raised by one. The following four commands set the vector $CE / $CF to the first byte of the string to print.

Now a character is read, checked for zero and printed out if it is not a zero. With the following BRANCH command all characters are printed. The register-contents are carried back when the loop is finished. The result is depicted on the next page:



Finally the processor reaches the RTS command and gets the return address from the stack. This address is incremented with every character read so that now it points to the last byte of the string, the end mark zero. Now the processor increments the program-counter and continues on.

The programmers had some good ideas, as you can see. The routine has important advantages. It doesn't require you to load the registers with pointers to the desired string, as is normal. This makes the program shorter,faster and requires less memory. Increasing the speed and reducing the memory requirements is always the goal of a goo systems programmer.

Those were the routines of the Kernal. We continue with the next section.

11.2.2. VECTOR-LOAD-TABLE

There is a table from $42CE-$4309 in the BASIC-interpreter with which you can easily get memory-locations, defined by vectors. You use these routines as follows: You enter the desired address into the vector and then call the routine belonging to the vector. But this index has some disadvantages:

1. You cannot use every vector, those you can use are listed in the index below.
2. You can not get a byte from any bank.
3. It is not always returned to the original bank.

Look at the index below to see if you can get a byte from a certain bank and return to the same bank! If not you must use the kernal-routine INDFET ($FF74).

Address	Vector	Byte-of	After the return
$42CE	$50/$51	Bank 1	Bank 14 (RAM-Bank 1)
$42D3	$3F/$40	Bank 1	Bank 14 (RAM-Bank1)
$42D8	$52/$53	Bank 1	Bank 14 (RAM-Bank 1)
$42DD	$5C/$5D	Bank 0	Bank 14
$42E2	$5C/$5D	Bank 1	Bank 14 (RAM-Bank 1)
$42E7	$66/$67	Bank 1	Bank 14 (RAM-Bank 1)
$42EC	$61/$62	Bank 0	Bank 14
$42F1	$70/$71	Bank 0	Bank 14
$42F6	$70/$71	Bank 1	Bank 14 (RAM-Bank 1)
$42FB	$50/$51	Bank 1	Bank 14 (RAM-Bank 1)
$4300	$61/$62	Bank 1	Bank 14 (RAM-Bank 1)
$4305	$24/$25	Bank 0	Bank 14

"Bank 14 (RAM-bank 1)" means: The memory shows the configuration as bank 14 after the switch is over. But RAM-bank 1 is turned on instead of

RAM-bank 0. These routines switch RAM and ROM only; the contents of the RAM-bank are never changed.

11.2.3 KERNAL CALLS

This index is found in the BASIC-interpreter also. Some kernal routines are called here, but Bank 15 is turned on first (the one exception is $928D; it turns on Bank 14).

Address	Kernal-Routine
$9251	Get status ($FFB7)
$9257	Set ($FFBA)
$925D	Set filename parameter ($FFBD)
$9263	BASIN ($FFCF)
$9269	BSOUT ($FFD2)
$926F	CLRCH ($FFCC)
$9275	CLOSE ($FFC3)
$927B	CLALL ($FFE7)
$9281	PRIMM ($FFE7)
$9287	SETBNK ($FF68)
$928D	set/get ($FFF0)
$9293	Requests stop-key ($FFE1)

11.3. FREE MEMORY

When using machine language programs you have to store them so they are not destroyed by BASIC or the operating system. The BASIC-RAM can be used only if you don't want to enter a BASIC program in the memory (or you set the start-of-BASIC higher, or the end-of-BASIC lower). It is important to know which addresses you can use in zero page. Some addressing modes are only available with pointers in the zero page. These free areas will be shown here. Let's first talk about the zero page.

11.3.1 FREE ZERO PAGE MEMORY

The addresses $FA-$FE in the zero page are always available, as well as the addresses $4E/$4F. These latter two addresses are used only by the boot-call routine (see chapter 10.1).

These are all the free addresses in the zero page. But six addresses are better than none. Other pages used by the operating-system also have free addresses. But using them really makes no sense because you also can use every other address in the memory.

11.3.2 USABLE MEMORY FOR MACHINE LANGUAGE

The C-64 often used two memory locations: the cassette buffer for short programs and the area $C000-$CFFF. A machine language program was usually not placed at the end-of-BASIC or even at the start of BASIC. The C-128 has more possibilities. First there is the cassette buffer, from $0B00-$0BFF. Directly behind the tape buffer are the two buffers for the RS232 interface:

1. $0C00 - $0CFF = RS232 input buffer
2. $0D00 - $0DFF = RS232 output buffer

This area is used by very few C-128 users. It offers 768 bytes of storage for machine language programs ($0B00 - $0DFF).

But that's not all. Under normal circumstances the area from $1300 to $1BFF is available. This is nine pages or 2304 bytes of RAM! This area was originally meant for ROM-cartridges.

There is another possibility: put your programs in the RAM-bank 1. This is meant for variables only. You no doubt realize that 64K for variables is a little much; you can use some of this for machine language programs.

The next chapter will show you how to make use of this memory. The C-128 offers many possibilities to arrange your programs in the memory. It is also useful to know that you can do graphics without using your RAM, because the VDC has its own 16K of RAM-memory.

CHAPTER 12

CHANGING THE OPERATING SYSTEM

Because the C-128 is also a C-64, there are routines to switch it into the C-64 mode. BASIC 7.0 has the command GO 64. The new kernal jump table has a routine called "C64 MODE". Let's have a closer look at this routine.

```
E24B   LDA #$E3
E24D   STA $01
E24F   LDA #$2F
E251   STA $00
E253   LDX #$08
E255   LDA $E262,X
E258   STA $01,X
E25A   DEX
E25B   BNE $E255
E25D   STX $D030
E260   JMP $0002
- - - - - - - - - - - - - - - - - - - - - - - - -
E263   LDA #$F7
E265   STA $D505
E268   JMP ($FFFC)
```

The first four commands switch the processor-port. The next five commands are a loop. First the addresses $E263 - $E26A to $0002 - $0009 are copied. The command in address $E25D changes the clock speed to 1 MHz. Therefore zero is stored in register 48 of the new VIC (the X-register was set to zero by the loop before). Row by row, the commands copied by the loop are executed. The switching is now made with the first two commands. The memory-location $D505 belongs to the MMU and has the following significance.

Bit-number	Function at 0	Function at 1
0	Z-80	6502 on
1 & 2	not used now	--
3	FSDIR control bit (for disk)	
4	C-64	C-128
5	C-64	C-128
6	C-128	C-64
7	40-column	80-column

Bits 3 and 4 are read only. The mode is not changed by writing to these bits. These bits represent only the polarity of two pins at the expansion port:

Bit 4 : GAME
Bit 5 : EXROM

These pins are checked every reset. If one of them returns a logic zero, then the C-64 mode is activated. Therefore the C-64 mode is activated immediately if the C-64 cartridges are connected. You also can build a special push button to activate the C-64 mode when desired.

Bit 7 is also read only. This bit gives you the status of the <40/80-DISPLAY> key. When this key is depressed, this bit will be 1. This bit does not show you the actual mode; it is not altered if you change the mode with <ESC+X>. The most important bit is bit 6. The C-128 goes into the C-64 mode if this bit is set (1). Now we can understand the two commands in the C-64 mode routine. The hexadecimal value $F7 is binary 11110111. Bit 6 is set.

The last command finally does the reset. After the switching, before the reset, all ROMs of the C-128 are turned off and those of the C-64 are turned on. If the last three commands were not in RAM Bank 0, but in either ROM

or RAM Bank 1, the computer would lock up. Try it, jump to the routine at $E263. The commands are in the RAM and the processor (now a 6510) continues at address $0007 and does a reset (the program counter stayed as it was, of course).

We want to make sure that the RAM Bank 0 is really used by both processors. Let's enter the following:

```
POKE 8192 , 170 : POKE 8193 , 85
GO 64  (and then Y for Yes, of course)
PRINT PEEK(8192);PEEK(8193)
```

You get back the values entered on the screen! If the memory is not erased, then you should be able to transfer complete programs to the C-64! Indeed this works. Try entering a small program, and changing the mode with the command GO 64. Now enter the following POKE's:

```
POKE 43, 1: POKE 44, 28
```

Your program appears back on the screen. You can even execute it. However, commands that the C-64 cannot execute are not allowed in these programs.

It is even possible to continue a program started in the C-128 mode in the C-64 mode without a break. It's necessary to do the largest part of the reset-routine first. Otherwise a lot would go wrong: the IRQ-Vector is not defined, the screens would not be set to normal values, etc...

An example:

```
2000   A9 F7      LDA   #$F7    ; Switch
2002   8D 05 D5   STA   $D505   ; to 64-mode
2005   A2 FF      LDX   #$FF    ; Reset-routine
2007   78         SEI           ; Up to jump
2008   9A         TXS           ; To BASIC-cold start
2009   D8         CLD           ; and test at ROM in
200A   8E 16 D0   STX   $D016   ; $8000
200D   20 A3 FD   JSR   $FDA3
2010   20 50 FD   JSR   $FD50
2013   20 15 FD   JSR   $FD15
2016   20 5B FF   JSR   $FF5B
2019   58         CLI
201A   EE 20 D0   INC   $D020   ; Raise bkgnd color
201d   4C 1A 20   JMP   $201A   ; once again
```

Now call the routine with the command SYS DEC ("2000"). After a moment you get a colorful screen--in the C-64 mode.

None of this is easy with BASIC programs. Try it once!

CHAPTER 13

THE C-64 MODE ON THE COMMODORE 128

One real advantage of the C-128 is its compatibility with the C-64. When you switch to the C-64-mode, you're able to use any and all software available for the C-64.

The Commodore people had to take care that all C-64 programs could really be run on the C-128 in the C-64-mode. Therefore they switched the complete system configuration of the C-128 to the C-64. The C-64-mode offers you only the normal BASIC V2.0., thus eliminating the bank switching and the Z-80 processor features. But some features of the C-128 can still be used in C-64 mode.

13.1 HIGH-SPEED ON THE C-64

The C-128 has a video controller, VIC 8564, for the 40-column screen. The Commodore 64 had a VIC 6564. The new VIC is upward compatible to the C-64's VIC 6564. This means it accomplishes the same tasks as the old 6564. There are two new registers added and these registers can also be modified in the C-64 mode! Register 48 plays a special role.

13.1.1 Register 48: Processor-clock

As with the C-64, the 128's clock controls the entire computer. With register 48 you can double the speed of the C-128 from 1 MHz to 2 MHz.

The great thing is that you can double the speed in C-64 mode as well.

This means that all of your "old" C-64 programs can run twice as fast. It also speeds up time consuming routines like the garbage collection. But there is one catch: The VIC uses clock gaps in the system to get a character out of the video RAM to refresh the screen. With the doubled clock speed these gaps are only half size, too short a time for the VIC to refresh the screen. You'll get a lot of strange things on the screen, instead of the normal display. Therefore, you should use the double clock only if the screen content does not matter to you. The routine is done like this:

```
POKE 53296,1    (2   MHz, fast)
POKE 53296,0    (1   MHz, normal)
```

To see what this looks like, enter the following program:

```
10   TI$="000000"
20   :
30   FOR X=1 TO 10000
40   NEXT X
50   :
60   PRINT TI$
```

This program needs 14 seconds to complete the loop. It works faster with the following lines added:

```
20 POKE 53296 , 1 :   REM FAST
50 POKE 53296 , 0 :   REM NORMAL
```

The same loop now takes only 7 seconds!

13.2 80-COLUMN CONTROLLER ACCESS

Yes, you now can access the 80-column controller in the C-64 mode. Finally, a full 80 characters per line. The operating system of the C-64 is not prepared for this controller, but it should not be a problem to copy these routines out of the C-128.

Any examples of this would be beyond the scope of this book. But here are a few suggestions. How about a graphic extension to get access to the 80-character screen in the '64 mode? Or an operating system expansion (in the IRQ) that will be able to write on the 80-character-screen?

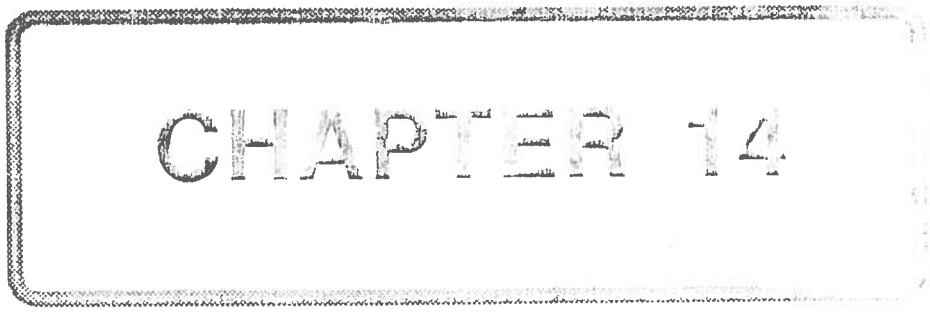
13.3 NUMERIC KEYPAD ON THE C-64

This is more of a suggestion than a tip. We'll activate the numeric keypad with a single POKE command in the C-64 mode. Unfortunately, you'll see that the characters don't fit the keys. We didn't have enough time to figure this out before press time. Maybe *you* can activate the numeric keypad **completely!**

Here's the POKE:

```
POKE 53295,248
```

The memory location 53295 corresponds to register 47 of the new VIC 8564--one of the new registers. Here you enter the same value as in the C-128 mode.



TOKEN TABLE

BASIC commands are not stored as a character sequence in the BASIC-RAM, but turned into TOKENs. A special routine compares the entered BASIC line with commands stored in the BASIC-interpreter. If the command is found, the BASIC-interpreter gives a special character sequence instead. The bytes for RUN--82, 85, and 78--are transferred to the byte 138 ($8A).

Try entering NEW and then the line 10 RUN. Now enter the monitor and type in M1C00. You get the beginning section of the BASIC-RAM. But where is the RUN? If you look closely you'll find the value $8A in the sixth position, which is the token for RUN.

Tokens save a lot of memory for BASIC programs. Take the command for DIRECTORY. Stored character by character it would use 9 bytes. Changed into a token it only uses one.

It follows that tokens also make programs run faster. There is only one byte to compare instead of 9. The transfer does not waste any time, because the computer is in input waiting loops for most of the time you're entering BASIC program text.

The tokens start at $80 (128); this lets the computer quickly check if the characters read are BASIC commands or simply text characters. But this layout has one disadvantage--only 128 BASIC commands are available.

Many TOKENs not used on the C-64; the C-128 uses them all. But this still was not enough for the new set of BASIC 7.0. commands. The 128's programmers had a simple solution for using some TOKENs twice. They gave these TOKENs a second byte. The command BANK has the tokens $FE and $02 now. With this method you can store a lot more commands.

Now the token table:

BASIC-Command	Value (hex.)	Value (dec.)
END	$80	128
FOR	$81	129
NEXT	$82	130
DATA	$83	131
INPUT#	$84	132
INPUT	$85	133
DIM	$86	134
READ	$87	135
LET	$88	136
GOTO	$89	137
RUN	$8A	138
IF	$8B	139
RESTORE	$8C	140
GOSUB	$8D	141
RETURN	$8E	142
REM	$8F	143
STOP	$90	144
ON	$91	145
WAIT	$92	146
LOAD	$93	147
SAVE	$94	148
VERIFY	$95	149
DEF	$96	150
POKE	$97	151
PRINT#	$98	152
PRINT	$99	153
CONT	$9A	154

BASIC-Command	Value (hex.)	Value (dec.)
LIST	$9B	155
CLR	$9C	156
CMD	$9D	157
SYS	$9E	158
OPEN	$9F	159
CLOSE	$A0	160
GET	$A1	161
NEW	$A2	162
TAB(	$A3	163
TO	$A4	164
FN	$A5	165
SPC(	$A6	166
THEN	$A7	167
NOT	$A8	168
STEP	$A9	169
+	$AA	170
-	$AB	171
*	$AC	172
/	$AD	173
^	$AE	174
AND	$AF	175
OR	$B0	176
>	$B1	177
=	$B2	178
<	$B3	179
SGN	$B4	180
INT	$B5	181
ABS	$B6	182
USR	$B7	183
FRE	$B8	184
POS	$B9	185
SQR	$BA	186
RND	$BB	187
LOG	$BC	188
EXP	$BD	189
COS	$BE	190
SIN	$BF	191
TAN	$C0	192
ATN	$C1	193
PEEK	$C2	194

BASIC-Command	Value (hex.)	Value (dec.)
LEN	$C3	195
STR$	**$C4**	**196**
VAL	**$C5**	**197**
ASC	$C6	198
CHR$	$C7	199
LEFT$	$C8	200
RIGHT$	$C9	201
MID$	$CA	202
GO	$CB	203
RGR	$CC	204
RCLR	$CD	205
Two-byte token	$CE	
JOY	$CF	207
RDOT	$D0	208
DEC	$D1	209
HEX$	$D2	210
ERR$	$D3	211
INSTR	$D4	212
ELSE	$D5	213
RESUME	$D6	214
TRAP	$D7	215
TRON	$D8	216
TROFF	**$D9**	**217**
SOUND	$DA	218
VOL	$DB	219
AUTO	$DC	220
PUDEF	$DD	221
GRAPHIC	$DE	222
PAINT	$DF	223
CHAR	$E0	224
BOX	$E1	225
CIRCLE	$E2	226
GSHAPE	$E3	227
SSHAPE	$E4	228
DRAW	$E5	229
LOCATE	$E6	230
COLOR	$E7	231
SCNCLR	$E8	232
SCALE	$E9	233
HELP	$EA	234

BASIC-Command	Value (hex.)	Value (dec.)
DO	$EB	235
LOOP	**$EC**	**236**
EXIT	**$ED**	**237**
DIRECTORY	$EE	238
DSAVE	$EF	239
DLOAD	$F0	240
HEADER	$F1	241
SCRATCH	$F2	242
COLLECT	$F3	243
COPY	$F4	244
RENAME	$F5	245
BACKUP	$F6	246
DELETE	$F7	247
RENUMBER	$F8	248
KEY	$F9	249
MONITOR	$FA	250
USING	$FB	251
UNTIL	$FC	252
WHILE	$FD	253
Two-byte token	$FE	
π (Pi)	$FF	255

Next is the index for double byte TOKENs. They have two preceding values: $CE and $FE. Index for commands with $CE as the first TOKEN:

BASIC-Command	First byte	Value (hex.)	Value (dec.)
POT	$CE	$02	2
BUMP	$CE	$03	3
PEN	$CE	$04	4
RSPPOS	$CE	$05	5
RSPRITE	$CE	$06	6
RSPCOLOR	$CE	$07	7
XOR	$CE	$08	8
RWINDOW	$CE	$09	9
POINTER	$CE	$0A	10

Index for commands with $FE as the first TOKEN:

BASIC-Command	First byte	Value (hex.)	Value (dec.)
BANK	$FE	$02	2
FILTER	$FE	$03	3
PLAY	$FE	$04	4
TEMPO	$FE	$05	5
MOVSPR	$FE	$06	6
SPRITE	$FE	$07	7
SPRCOLOR	$FE	$08	8
RREG	$FE	$09	9
ENVELOPE	$FE	$0A	10
SLEEP	$FE	$0B	11
CATALOG	$FE	$0C	12
DOPEN	$FE	$0D	13
APPEND	$FE	$0E	14
DCLOSE	$FE	$0F	15
BSAVE	$FE	$10	16
BLOAD	$FE	$11	17
RECORD	$FE	$12	18
CONCAT	$FE	$13	19
DVERIFY	$FE	$14	20
DCLEAR	$FE	$15	21
SPRSAV	$FE	$16	22
COLLISION	$FE	$17	23
BEGIN	$FE	$18	24
BEND	$FE	$19	25
WINDOW	$FE	$1A	26
BOOT	$FE	$1B	27
WIDTH	$FE	$1C	28
SPRDEF	$FE	$1D	29
QUIT	$FE	$1E	30
STASH	$FE	$1F	31
FETCH	$FE	$21	33
SWAP	$FE	$23	35
OFF	$FE	$24	36
FAST	$FE	$25	37
SLOW	$FE	$26	38

Optional Diskette

For your convenience, the program listings contained in this book are available on a 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 £5.00+ £0.50 (£1.50 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose check, money order or credit card information. Mail your order to:

First Publishing Ltd
20B Horseshoe Park
Horseshoe Rd.
Pangbourne, Berks.
Tel: 07357 5244

Tricks and Tips for the C-128

Tricks and Tips for the C-128 is a tremendous treasure trove of programming techniques and 'tricks' for every C-128 owner. This book not only contains plenty of example programs, but also explains in a simple to understand manner the operation and programming of the computer.

Contents include:

- Graphics on the C-128
- Working with more than one screen
- Graphics with the 80 column screen
- Simulating multiple windows
- Listing Converter
- Software protection on the C-128
- Changing the keyboard
- The MMU (Memory Management Unit)
- Important memory locations
- Changing the operating system
- Sprite handling
- Custom character sets
- Autostart
- The 80 column controller
- Modified INPUT
- Line insertion
- Banking
- Kernal routines
- Key pad in C-64 mode
- C-64 mode of the C-128

Other titles in this series:
1. THE ANATOMY OF THE COMMODORE 128
2. ANATOMY OF THE 1571 DISK DRIVE

£12.95