HOW TO PROGRAM YOUR Commodore-64®

in 6502/10 Machine Language

Introduction to Machine Language for the BASIC Programmer

Sam D. Roberts

HOW TO PROGRAM

YOUR

Commodore-64
in 6502/10
Machine Language

Introduction to
Machine Language
for the
BASIC Programmer

PREFACE

Few features of a home computer confuse the novice computer owner more than software. Many of these new owners have studied the system manuals, they have possibly read articles or even books on microcomputers. Many of them already programmed their Commodore-64 computer in BASIC, FORTH, PILOT or another high level language. After a while, they will find out that the language used is too slow for their needs (animation, sound, graphics, to name just a few applications). They also want to know more about the internal things happening in the computer. They are most likely aware of the ubiquitous 0's and 1's that control the computer. But how do those ubiquitous digits relate to the information displayed on the screen and to the language of the computer. How can they be put to work ?

The subject of this book ist to teach you how to program your C-64 computer in 6502 (6510) machine language. You may use a machine language monitor (like 64 MON, Supermon or the Macrofire Editor/Assembler with its built in monitor), to enter and start the programs listed in this book. Later on we will find out that it is too cumbersom to do the assembly by hand. We than use an assembler for our programs and we will learn how to call machine language subroutines from BASIC.

TABLE OF CONTENTS

1

PROGRAMMING IN MACHINE-LANGUAGE WITH THE MICROPROCESSOR 6510

Part 1

Most people don't realize that BASIC commands like IF or THEN actually are sequences of commands in machine-language. This introduction is meant for those who want to leave BASIC and go deeper into their computer.

The 6510 microprocessor and its commands are the subjects of this introduction. Once you understood how this microprocessor works it is not very difficult to learn another one. In this section we will talk about some rudiments.

The 6510 microprocessor is software compatible to the wellknown 6502 microprocessor. That means that both microprocessors use the same instruction set. The only difference that we have to pay attention to is, that the 6510 has an output register at address 0000 and the data-direction-register for that output register at address 0001.

The first thing you need for programming in machine-language is the monitor. This is not the television, but the operating system that takes control over the computer after power-up.

The monitor is very important for programming in machine-language. It contains the routines needed most, such as outputs to, and inputs from, a device.

To get into the monitor you have to enter a certain command. With the APPLE II the command would be : CALL - 151 (in BASIC), or "M" after power up with OHIO C1P. The AIM 65 is in the monitor automatically after power up. With the COMMODORE 64 you need the 64MON cartridge, or the MACROFIRE program from HOFACKER, if you want to program in machine language. When using MACROFIRE the command for getting into the monitor from the editor is CTRL-P.

The samples in this booklet are written for the machine-language monitor for COMMODORE 64, or the machine language monitor, which is included in the MACROFIRE program.

Programs in machine-language work directly in the computers memory. Each command is stored at a certain address. This address is the memory location where the first statement to be executed is stored. To start a machine-language program the startaddress of that progam has to be stored in the progam counter of the microprocessor.

The statements for the microprocessor are one, two, or three bytes long. One byte is eight bits broad and, therefore, one word for a eight bit processor. The first byte contains the operation code. Figure 1 shows the different commands available on the 6510 microprocessor. The left column in that figure shows the mnemonics for the commands (assembler-code). One or two address bytes can follow the operation code. There are several ways for addressing, which will be explained later.

Examples of statements :

1.
Load the accumulator with the contents of memory location $1000 ($ means : the following number is hexadecimal).

assembler code : LDA $1000
hex-code : AD 00 10

This statement is three bytes long. With the 6510 the addresses are specified with first the lower, then the higher byte.

2.
Compare the contents of the accumulator with the contents of the very next location.

assembler code : CMP #$7F
hex-code : C9 7F

This is a two-byte statement. The #-sign means immediate addressing. The operation referes to the memory location which immediately follows the command.

3.
Shift the contents of the accumulator to the left one position.

assembler-code : ASL
hex-code : 0A

This is a one-byte statement, no address is needed in this case.

Notes to part 1 :

* monitor
* address
* program counter
* statement
* 1-, 2-, and 3-byte commands

Commands	symb. Code	Operation	Adressing modes IMM.	ABS	ABS,X	ABS,Y	Z0	Z0,X	Z0,Y	(IND,X)	(IND),Y	REL	IND	ACCU	IMPL	condition codes N	Z	C	1	D	V
Transport	LDA	M → A	A9	AD	BD	B9	A5	B5		A1	B1					X	X	–	–	–	–
	LDX	M → X	A2	AE		BE	A6		B6							X	X	–	–	–	–
	LDY	M → Y	A0	AC	BC		A4	B4								X	X	–	–	–	–
	STA	A → M		8D	9D	99	85	95		81	91					–	–	–	–	–	–
	STX	X → M		8E			86		96							–	–	–	–	–	–
	STY	Y → M		8C			84	94								–	–	–	–	–	–
	TAX	A → X													AA	X	X	–	–	–	–
	TAY	A → Y													A8	X	X	–	–	–	–
	TXA	X → A													8A	X	X	–	–	–	–
	TYA	Y → A													98	X	X	–	–	–	–
	TXS	X → S													9A	–	–	–	–	–	–
	TSX	S → X													BA	X	X	–	–	–	–
	PLA	S+1 → S, Ms → A													68	X	X	–	–	–	–
	PHA	A → Ms, S–1 → S													48	–	–	–	–	–	–
	PLP	S+1 → S, Ms → P													28						
	PHP	P → Ms, S–1 → S													08	–	–	–	–	–	–
arithmetic-	ADC	A+M+C → A	69	6D	7D	79	65	75		61	71					X	X	X	–	–	X
	SBC	A–M–$\overline{C}$ → A	E9	ED	FD	F9	E5	F5		E1	F1					X	X	X	–	–	X
	INC	M+1 → M		EE	FE		E6	F6								X	X	–	–	–	–
	DEC	M–1 → M		CE	DE		C6	D6								X	X	–	–	–	–
	INX	X+1 → X													E8	X	X	–	–	–	–
	DEX	X–1 → X													CA	X	X	–	–	–	–
	INY	Y+1 → Y													C8	X	X	–	–	–	–
	DEY	Y–1 → Y													88	X	X	–	–	–	–
logic-	AND	A ∧ M → A	29	2D	3D	39	25	35		21	31					X	X	–	–	–	–
	ORA	A ∨ M → A	09	0D	1D	19	05	15		01	11					X	X	–	–	–	–
	EOR	A ⊻ M → A	49	4D	5D	59	45	55		41	51					X	X	–	–	–	–
compare-	CMP	A–M	C9	CD	DD	D9	C5	D5		C1	D1					X	X	X	–	–	–
	CPX	X–M	E0	EC			E4									X	X	X	–	–	–
	CPY	Y–M	C0	CC			C4									X	X	X	–	–	–
	BIT	A ∧ M		2C			24									7	X	–	–	–	6
branch-	BCC	BRANCH ON C=0										90				–	–	–	–	–	–
	BCS	BRANCH ON C=1										B0				–	–	–	–	–	–
	BEQ	BRANCH ON Z=1										F0				–	–	–	–	–	–
	BNE	BRANCH ON Z=0										D0				–	–	–	–	–	–
	BMI	BRANCH ON N=1										30				–	–	–	–	–	–
	BPL	BRANCH ON N=0										10				–	–	–	–	–	–
	BVC	BRANCH ON V=0										50				–	–	–	–	–	–
	BVS	BRANCH ON V=1										70				–	–	–	–	–	–
	JMP			4C									6C			–	–	–	–	–	–
	JSR			20												–	–	–	–	–	–
SHIFT-	ASL			0E	1E		06	16						0A		X	X	X	–	–	–
	LSR			4E	5E		46	56						4A		0	X	X	–	–	–
	ROL			2E	3E		26	36						2A		X	X	X	–	–	–
	ROR			6E	7E		66	76						6A		X	X	X	–	–	–
Status-Register	CLC	C=0													18	–	–	0	–	–	–
	CLD	D=0													D8	–	–	–	–	0	–
	CLI	I=0													58	–	–	–	0	–	–
	CLV	V=0													B8	–	–	–	–	–	0
	SEC	C=1													38	–	–	1	–	–	–
	SED	D=1													F8	–	–	–	–	1	–
	SEI	I=1													78	–	–	–	1	–	–
Misc.	NOP	NO OPER													EA	–	–	–	–	–	–
	RTS	RETURN F. SUB													60	–	–	–	–	–	–
	RTI	RETURN F. INT													40						
	BRK	BREAK													00	–	–	–	1	–	–

Instruction Set

READ THIS!
PRTBYT

The examples in this book are written for the COMMODORE 64. They work in conjunction with a machine-language monitor.

The samples use some routines which are stored in the COMMODORE kernal ROM. Two examples are the output of a character to the screen (called CHROUT, starting at $FFD2), and the input of a character from the keyboard (called CHRIN, starting at $FFCF).

Some programs contain the command JSR PRTBYT. This subroutine calls a routine for output of the contents of the accumulator in the form of two hexadecimal bytes. This routine has to be entered together with the program that calls that routine. PRTBYT starts at address $C000 and is called by the OP-code 20 00 C0.

The rest of the programs start at address $C100. This is an unused part of memory and may be used for short programs or for storage of data. Our examples are short so that they fit in this area.

Here is the routine PRTBYT :

```
                 BYTE     EQU      $C023
                 CHROUT   EQU      $FFD2
                          ORG      $C000
C000: 8D23C0 PRTBYT   STA      BYTE
C003: 4A                  LSR
C004: 4A                  LSR
C005: 4A                  LSR
C006: 4A                  LSR
C007: 2014C0              JSR      OUTPUT
C00A: AD23C0              LDA      BYTE
C00D: 2014C0              JSR      OUTPUT
C010: AD23C0              LDA      BYTE
C013: 60                  RTS
C014: 290F   OUTPUT   AND      #$0F
C016: C90A                CMP      #$0A
C018: 18                  CLC
C019: 3002                BMI      $C01D
C01B: 6907                ADC      #$07
C01D: 6930                ADC      #$30
C01F: 4CD2FF              JMP      CHROUT
C022: 00                  BRK

PHYSICAL ENDADDRESS: $C023

*** NO WARNINGS

BYTE              $C023
PRTBYT            $C000     UNUSED
CHROUT            $FFD2
OUTPUT            $C014
```

When used as a subroutine, location $C022 has to be changed into RTS (hex-byte 60).

To enter the above program (the hex-bytes) use a machine-language monitor.

2

Part 2

2-1 Programming model of the 6510 CPU

By looking at the hardware structure of a microprocessor you get a survey of what statements it can execute. The structure of the 6510 is shown in figure 2-1. There are four eight-bit registers :
the accumulator, the X-register, the Y-register, and the status register. The program counter is 16 bit long and can represent addresses from 0 to 65535.

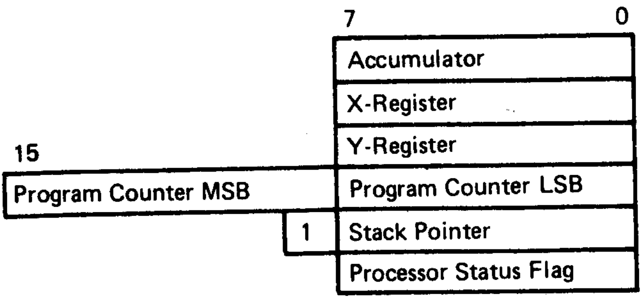


Figure 2-1
programming model of the 6510

Next is a stack pointer. The stack pointer points to a special part of the memory, the stack, at addresses $100 to $1FF. Only eight bits are used for addressing, the ninth bit always is one.

What are all these registers for ?

The main register is the accumulator. This is where all calculations are executed and the results of all calculations are stored. For addressing, one of the index registers may be used. These registers can be used as counters. For example the statement INX increments the contents of the X-register by one. The index register can also be used to indicate addresses. These features will be used in later sample programs.

The status register indicates the present status of the processor. Each bit marks a result of an operation.

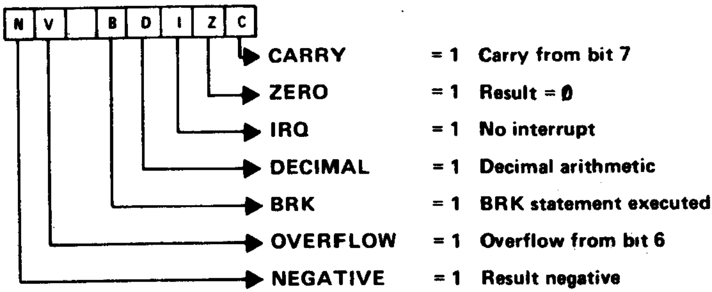


Figure 2-2
bits of the status register

The zero flag becomes 1, if the contents of the accumulator becomes zero. The carry flag becomes 1, if a carry from bit 7 to bit 8 occurres.

The right column of figure 1 shows which operations affect the bits in the status register (X indicates : change possible). For example a LDA statement can change bits N and Z; the statement STA can't change any bit of the status register.

The stackpointer points to a free area in the stack. You can store the contents of the accumulator there with PHA (push accumulator; one byte statement) then the stackpointer will be set to the next memory location. PLA (pull accumulator) sets the pointer back one location. At this time the contents of that location will be transfered to the accumulator.

Note : the top of the stack is address $1FF. The stack builds up to address $100. Another important task of the stack is to hold the current address in case of a jump to a subroutine. At the return from the subroutine this address is transferred back to the program counter. The program counter always holds the address of the command to be executed next. Only jump-instructions change the contents of the program counter.

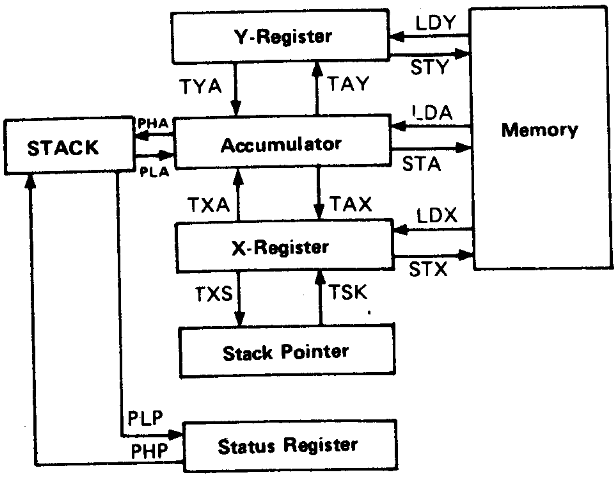


Figure 2-3
Transfer of data between registers and memory

Figure 2-3 shows all commands available for transferring data between the registers and memory. As you can see the 6510 has no command for transferring data between the registers, or to exchange the contents of X- and Y-register as is possible with other processors.

If you know how to program one processor and wish to program another one, you should study the logical structure, concerning the effects of the commands.

2-2
A first example and the paper-pencil-method

The addition of two numbers is quite simple in a higher programming language :

```
                                ORG   $C100
10 A=5            A905          LDA   #$05
20 B=3            18            CLC
30 C=A+B          6903          ADC   #$03
40 PRINT C        2000C0        JSR   $C000          ;(PRTBYT)
50 END            00            BRK
```

To do the same job in machine language it is necessary to answer the following questions first :

Where are the numbers stored ?

Are the numbers of type fixed point or floating point ?

Is there a routine existing in the monitor, which prints the contents of a memory location ?

Here is the program in machine-language :

LDA #$05	Load the accumulator with 05 (direct addressing). The number 05 is stored immediately after the operation code and is of the fixed point type

CLC	clear the carry bit for the next operation
ADC #$03	add with carry 03 (immediate). Result is in the accumulator.
JSR PRTBYT	PRTBYT is a subroutine that prints the contents of the accumulator on the screen as two hex-numbers
BRK	stop here

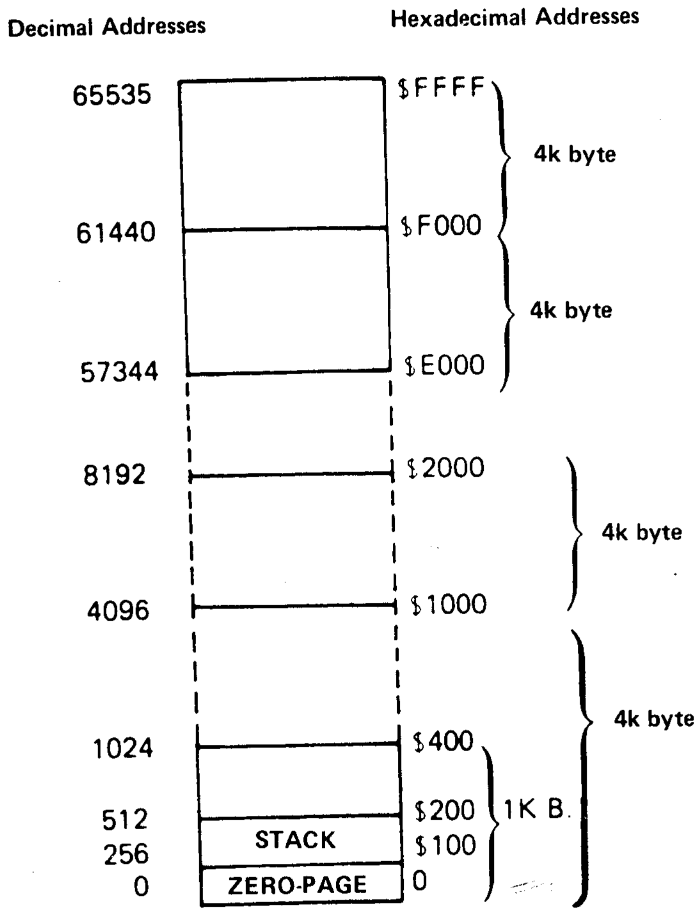
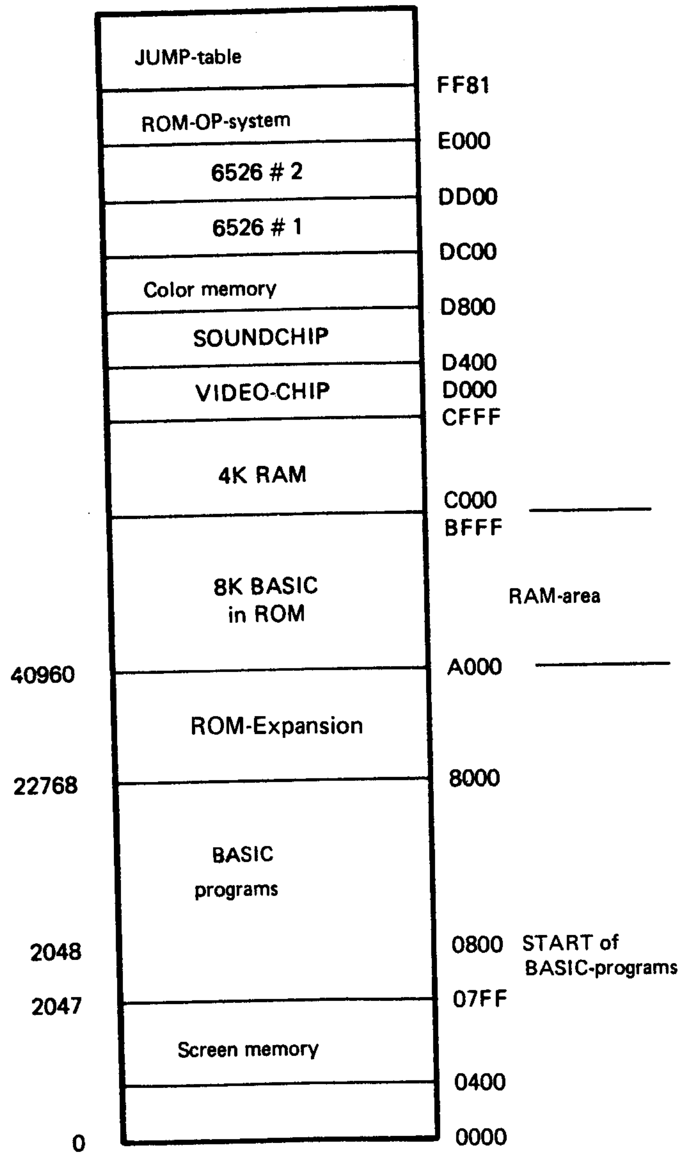


Figure 2.4: Decimal and hexadecimal addressing of a 64 k byte memory

Figure 2-4 shows a survey of the memory. On the left side are the addresses in decimal and on the right side they are in hexadecimal form. The addresses from 0 to $400 represent 1k of memory. The addresses from $1000 to $2000 represent 4k. Now we want to translate the program into machine language by using the paper and pencil method. This is the lowest level of programming, but it is useful in learning the programming in machine language.

The first problem is where to start the program. On principle the program can start anywhere in memory. There are however two certain areas which you should not use. First is the zero-page, a very useful area with simplified addressing, second is the stack. (remember that the stack is used by the processor itself !). For these reasons the addresses from 0 to $1FF are not available.

With the COMMODORE 64 the standard memory map looks as follows :



COMMODORE 64 standard memory map

Let's place our program at $C100. The RAM at addresses $C000 through $CFFF is always available to the user with the COMMODORE 64.

Now we can translate the first command. If you look at the table you will find that LDA has the code A9. Adjacent to that the first line looks as follows :

```
$C100 A9 05     LDA #$05
```

A9 is the operation code and 05 is the number which follows immediately. This command is two bytes long. The next line is at $C102.

```
$C102 18        CLC
```

18 is the code for clear carry. It can be found in table 1 under status register statements. The line after that is add with carry (ADC). The carry bit has to be cleared in this case, otherwise the result of the addition could be wrong.

```
$C103 69 03     ADC #$03
```

69 is the code for addition with immediate addressing. It can be found in table 1 under arithmetic statements. The next command calls the subroutine PRTBYT for output to the screen. This subroutine starts at address $C000 with our programs. Therefore the line for output looks as follows :

```
$C105 20 00 C0 JSR PRTBYT
```

20 is the code for JSR (JUMP SUBROUTINE).

Remember : with the 6510 processor you first have to enter the lower byte (LSB, least significant byte), then the higher byte of the address (MSB, most significant byte). After which we stop the program with :

```
$C108 00        BRK
```

Most computers jump back into the monitor after they hit a BRK-instruction.

The whole program looks like this for the COMMODORE 64 :

```
$C100 A9 05     LDA #$05
$C102 18        CLC
$C103 69 03     ADC #$03
$C105 20 00 C0 JSR PRTBYT
$C108 00        BRK
```

Thus a dump of these locations looks as follows :

```
$C100: A9 05 18 69 03 20 00 C0
$C108: 00
```

At this point we will not talk about how to enter that program, rather we will discuss different techniques of addressing. Let's assume that there is the same job, but the two numbers are stored in two zero-page locations. The number 5 is stored at location $10 and the number 3 is stored at location $11. Our program would look as follows :

```
$C100 A5 10     LDA $10 ;load the accumulator with
                the contents of location $10

$C102 18        CLC     ;clear carry bit

$C103 65 11     ADC $11 ;add contents of location
                        $11

$C105 20 00 C0 JSR PRTBYT ;output to screen

$C108 00        BRK     ;stop
```

A5 is the code for LDA with the contents of a zero-page location.

In the next example we assume, that the numbers are stored anywhere in memory, for example at $200A and at $3005. The program would look as follows :

```
$C100 AD 0A 20 LDA $200A ;load the contents of
                          location $200A

$C103 18       CLC       ;clear carry bit

$C104 6D 05 30 ADC $3005 ;add contents of location
                          $3005

$C107 20 00 C0 JSR PRTBYT;output to screen

$C10A 00       BRK       ;stop
```

In this case AD is the code for LDA with the contents of an absolute address. The code for ADC the contents of an absolute address is 6D. This last program is two bytes longer than the prior one. If possible, in order to shorten the program, the zero-page should be used for auxiliary cells, but take into consideration, that with the COMMODORE 64 only the zero-page locations $02, $FB, $FC, $FD, and $FE are available to the user, the other locations are used by BASIC, or by the operating system.

Notes to part 2:

* programming model of the 6510
* CPU register
* zero-page addressing
* absolute addressing

3

Part 3

In part 2 we talked about a program which flows off straight. In this part we will talk about programs which contain branches.

3-1 Programs with branches

There are many programs which contain loops that have to be traveled through until a certain condition becomes complied with. As an example the condition can be whether the contents of a memory location or a register is equal to zero, or whether a number in a register is greater than, or equal to, or smaller than, the contents of a memory location. The bits in the status register are influenced by operations or comparisons (see figure 2-2). Whether branch commands are executed or not, depends on the status of certain bits.

An example of this is a delay loop. The contents of the X-register is decremented until it is zero.

Here is the program for that :

```
LDX #$0A   ;load the X-register with A0

M DEX      ;decrement X-register by one

BNE M      ;jump back to M, if not zero

BRK        ;stop program, if X-register=0
```

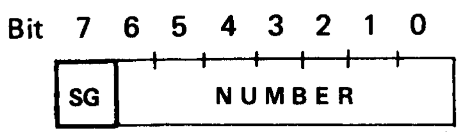
In machine-language it looks as follows :

```
                         ORG     $C100
C100: A2A0               LDX     #$A0
C102: CA      M          DEX
C103: D0FD               BNE     M
C105: 00                 BRK
```

Location C104 has been left open. The number of bytes the program has to jump back belongs to there.

The branch commands use the so-called relative addressing. This means the current contents of the program counter becomes increased or decreased by a certain number. The program then continues at the new address. What is the current contents of the program counter ? The program counter of the 6510 always points to the next command; in our example this is the BRK-command at location C105. To get back to location C102 we have to decrement the program counter by 3. Therefore the hexadecimal equivalent of -3 has to be stored at location C104.

How are negative numbers displayed ?
Bit 7 is used to determine, whether a number is positive or negative.



If bit 7 is 1, then the number is negative, if bit 7 is zero, then the number is positive.

Positive numbers are :

```
  0 = $00 = %0000 0000
  1 = $01 = %0000 0001
  2 = $02 = %0000 0010
  .
  .
  .
127 = $7F = %0111 1111
```

Negative numbers are described by the complement on two. To complement a number means to turn around all bits of that number : ones become zeros, zeros become ones. With the complement on two, one is added after that. For example the number -1 :

+1 = %0000 0001 ; the complemented number :
%1111 1110
addition of 1 results in : %1111 1111 = $FF

Negative numbers are :

```
  -1 = $FF = %1111 1111
  -2 = $FE = %1111 1110
  -3 = $FD = %1111 1101
  .
  .
  .
-128 = $80 = %1000 0000
```

Thus relative branches can range from -128 to +127.

Complete program :

```
C100 A2 A0     LDX #$A0
C102 CA      M DEX
C103 D0 FD     BNE M
C105 00        BRK
```

You also can use the following tables :

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Table 3-1 Forward branch

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Table 3-2 Backward branch

Most mistakes happen with the calculation of bytes for relative jumps, when assembling by hand !

3-3 Comparisons

Comparisons always happen between a register (accumulator, X- or Y-register) and a memory location. Bits N (negative), Z (zero), and C (carry) are influenced by comparisons.

Figure 3-3 shows how :

Comparison	N	Z	C
A, X, Y < M	1*	0	0
A, X, Y = M	0	1	1
A, X, Y > M	0*	0	1

* comparison with twos complement

Figure 3-3 Flags with comparisons

If the contents of the accumulator (or X-register, Y-register) is smaller than the contents of a memory location, then the zero flag and the carry flag become 0. For these two flags the numbers can be between 0 and 255. For the N flag the numbers are compared in the twos complement. These numbers can be from -128 to +127.

For example :
The contents of the accumulator is $FD, the contents of a memory location is 00. A comparison A > M (252-00) causes C to become 1 and Z to become 0. Here are different possibilities to branch :

```
A <  M     BCC    LABEL

A <= M     BCC    LABEL
           BEQ    LABEL

A =  M     BEQ    LABEL

A >= M     BCS    LABEL

A >  M     BEQ    NOT LABEL
           BCS    LABEL
```

The following program is a simple example for comparisons and branches. We want to input a character from the keyboard and check whether or not it is a hexadecimal number (0-9, A-F). If the character is hexadecimal, then we want to store it in location INP with address $FE. If not, we want to leave the program ($00 in INP).

For the input we use subroutine CHRIN, which is included in most monitors. This subroutine checks whether or not a key is pressed. If a key is pressed, the program returns from the subroutine with the ASCII character in the accumulator.

With the COMMODORE 64 the program returns from this subroutine after the RETURN key has been pressed.

Figure 3-4 shows the ASCII characters

LSD \ MSB		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N	↑	n	~
F	1111	SI	VS	/	?	O	←	o	DEL

ASCII characters

```
                    ORG     $C100
           CHRIN    EQU     $FFCF
           AUX      EQU     $FE
C100: A900          LDA     #0
C102: 85FE          STA     AUX
C104: 20CFFF        JSR     CHRIN
C107: C930          CMP     #$30
C109: 9013          BCC     L2
C10B: C947          CMP     #$47
C10D: B00F          BCS     L2
C10F: C93A          CMP     #$3A
C111: 9007          BCC     L1
C113: C941          CMP     #$41
C115: 9007          BCC     L2
C117: 18            CLC
C118: 6909          ADC     #9
C11A: 290F   L1     AND     #$0F
C11C: 85FE          STA     AUX
C11E: 00     L2     BRK

PHYSICAL ENDADDRESS: $C11F

*** NO WARNINGS

CHRIN           $FFCF
L1              $C11A
AUX             $FE
L2              $C11E
```

Figure 3-5 program ASCII HEX

Try to assemble the program by hand and calculate the jumps. This is a very good mental exercise. Compare your branch statements with those in the program before you start the program.

Notes to part 3 :

* program branch
* positive and negative numbers

* relative addressing
* comparisons

4

Part 4

In this section we will talk about the use of subroutines. Subroutines are independent parts of programs. They are called by the statement JSR (JUMP SUBROUTINE). With RTS (RETURN FROM SUBROUTINE) you return to the main program.

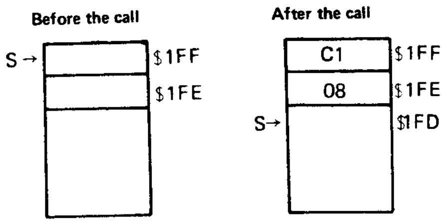
4-1 How to call a subroutine

As an example we use the instruction JSR CHRIN from the program ASCII HEX.
The first lines there are :

```
C100 A9 00       LDA #$00
C102 85 FE       STA $FE
C104 20 CF FF    JSR CHRIN
C107 C9 30       CMP #$30
```

Location C104 contains the command for jump to subroutine. With the execution of this statement the address of the command to be executed after that (decremented by one) is stored in the stack.

The stack



The stack is a defined part of memory of 6502 sytems. The TOS (top of stack) is at address $1FF. The stack pointer always points to the next available location in the stack.

It is possible to jump from one subroutine into another one. Figure 4-3 shows the model for that.

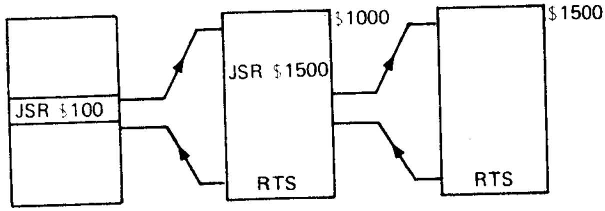


Figure 4-3 nested subroutines

The stack could hold up to 128 return addresses of subroutines at a time, but you will never need that many.

4-2 Saving the contents of registers

Most subroutines change the contents of the registers. If these contents are needed later (after RTS), they have to be saved.
This can be done either in the main program or in the subroutine. If you know what registers are changed by the subroutine, then you can save the contents at an unused location. The easiest way though, is to save the contents of all registes within the subroutine. The beginning of that subroutine then looks as follows :

```
PHA   ;ACCU -> STACK
TXA   ;X -> ACCU
PHA   ;ACCU -> STACK
TYA   ;Y -> ACCU
PHA   ;ACCU -> STACK
```

Prior to the RTS command, you have to restore the old contents of the registers. The end of the subroutine will look as follows :

```
PLA  ;LOAD Y
TAY  ;
PLA  ;LOAD X
TAX  ;
PLA  ;LOAD ACCU
RTS  ;JUMP BACK
```

The contents of the registers could also be stored in auxiliary locations instead of the stack.

4-3 Exchange of data between main program and subroutine

There are three ways to exchange data between main program and subroutine.

1. Exchange via the registers. For example most keyboard input routines have the character in the accumulator at the return.

2. Exchange via the stack. This technique is used often when machine language programs are used together with high level languages (for example PASCAL).

3. The main program and the subroutine use a common memory area for the data.

The method you should use depends on the problem to be solved. If the whole program is written by one programmer, then he will use the method he likes best. If more than one programmer works together then they have to arrange the kind of exchange.

Advantages with the use of subroutines :
Longer programs become split into smaller parts.

The shorter parts are easier to understand and debugging becomes easier. You can build up a library of subroutines and can use these subroutines later.

4-4 Indirect jumps and indirect jumps to subroutines.

```
SPECL:  LDA     CART        ,CHECK FOR RAM OR CART
        BNE     ENSPEC      ,GO IF NOTHING OR MAYBE RAM
        INC     CART        ,NOW DO RAM CHECK
        LDA     CART        ;IS IT ROM?
        BNE     ENSPEC      ;NO
        LDA     CARTFG      ,YES,
        AND     #$80        ;MASK OFF SPECIAL BIT
        BEQ     ENSPEC      ;BIT SET?
        JMP     (CARTAD)    ;YES, GO RUN CARTRIDGE

        CHECK FOR AMOUNT OF RAM
```

This is an indirect jump

```
F23F   AD FC BF
F242   D0 12
F244   EE FC BF
F247   AD FC BF
F24A   D0 0A
F24C   AD FD BF
F24F   29 80
F251   F0 03
F253   6C FE BF
```

5

Part 5

5-1 Indexed addressing

Example for indexed addressing :
We have stored data (numbers and letters) at memory locations $4000 – $401F. We now want to transfer this data to another area starting at $5000. This could be done by the following program :

```
LDA $4000
STA $5000
LDA $4001
STA $5001
LDA $4002
STA $5002
.
.
.
LDA $401F
STA $501F
```

This program is long and tedious. Six bytes are consumed for the transfer of one byte, which means the whole program is 32*6 = 192 bytes long. With indexed addressing this program becomes short and simple. With the statement LDA $4000,X you load the accumulator with the contents of the memory location whose address is the sum of address $4000 and the contents of the X-register.

For example :
If X=1, the contents of location $4001 will be stored in the accumulator;
If X=2, the contents of location $4002 will be stored in the accumulator.

It is also possible to use the Y-register. The statement then would be : LDA $4000,Y.

Here is the program :

```
                           ORG      $C100
              FROM         EQU      $4000
              TO           EQU      $5000
              CLR          EQU      $00
C100: A200    MOVE         LDX      #CLR
C102: BD0040  M            LDA      FROM,X
C105: 9D0050               STA      TO,X
C108: E8                   INX
C109: E020                 CPX      #$20
C10B: D0F5                 BNE      M
C10D: 00                   BRK

PHYSICAL ENDADDRESS: $C10E

*** NO WARNINGS

FROM                $4000
CLR                 $00
M                   $C102
TO                  $5000
MOVE                $C100      UNUSED
```

Figure 5-1

First the X-register is loaded with zero. After that the accumulator is loaded : LDA $4000,X then the contents are stored at $5000, X. INX increments the X-register. It is then checked, to see whether all data has been transferred already.

We want to transfer the contents of locations $4000 - $401F. The first location that should not be tranfered is $4020. If the contents of the X-register became $20 after INX, the program should stop.

In the comments above, $4000 means the address of that location; ($4000) means the contents of that location.

Both index registers are 8 bit long. For that reason it is possible to index from 0 to 255. Thus we can transfer a maximum of 256 bytes with this method. For the transfer of larger areas we have to use a different technique which will be discussed later.

Here is another example :
We want to exchange the contents of locations $4000 with $40FF, $4001 with $40FE, $4002 with $40FD , etc. (figure 5-2).

First we load X with 0 and Y with FF. Then we load the contents of $4000 and store it in the stack. After that we load the contents of $40FF and store it at $4000 and next we store the value in the stack at $40FF. Lastly the Y-register is decremented and the X-register is incremented. The exchange is done when X = $80.

```
                        ORG     $C100
               ADDRESS  EQU     $4000
C100: A200              LDX     #$00
C102: A0FF              LDY     #$FF
C104: BD0040 M          LDA     ADDRESS,X
C107: 48                PHA
C108: B90040            LDA     ADDRESS,Y
C10B: 9D0040            STA     ADDRESS,X
C10E: 68                PLA
C10F: 990040            STA     ADDRESS,Y
C112: 88                DEY
C113: E8                INX
```

```
C114: E080          CPX     #$80
C116: D0EC          BNE     M
C118: 00            BRK

PHYSICAL ENDADDRESS: $C119

*** NO WARNINGS

ADDRESS         $4000
M               $C104
```

Figure 5-2

The effective address with indexed addressing is the sum of the programmed address plus the contents of the index register used. The carry flag is noted with these calculations. (The carry flag will be set, if a carry appears with the calculations). With X = $FF the contents of the accumulator will be stored at $41DF, with the command STA $40E0,X.

The 6510 has two more ways of addressing, which consist of indirect and indexed addressing.

Note : The final address with indirect addressing is not the programmed address, but contents of that address. For example : JMP ($2000) means a jump to $3AFF, if the contents of $2000 and $2001 are $3AFF.

5-2 Indexed indirect addressing

With this kind of addressing the programmed address always is an address of the zero page, with the index register always the X-register. For example LDA ($10,X).
The final address can be calculated by adding the contents of the X-register to $10. The contents of this and the following address is the

effective address.

Example :
Contents of locations $0E - $15

```
(0E) = FF
(0F) = 0F
(10) = 00
(11) = 11
(12) = 2F
(13) = 30
(14) = 00
(15) = 47
```

If X = 0, then LDA ($10,X) loads the contents of location $1100; if X = 2, then LDA ($10,X) loads the contents of $302F, X = 4 causes the contents of $4700 to be loaded. No attention is payed to a carry occurring during the calculation of the address. For this reason the contents of location $0FFF will be loaded, if X = $FE.

5-3 Indirect indexed addressing

With this kind of addressing the programmed address is in the zero page also. Only register Y can be used as an index register in this case.
Example : STA ($10),Y.

To find out the final address, add the contents of locations $10 and $11 to the contents of register Y.
Example :

```
($10) = 3E
($11) = 2F
```

If Y = 0, then contents of the accumulator would be stored at location $2F3E.

The last two addressing modes are used mainly as indirect addressing, with X = 0 respectively Y =

0. It then follows that LDA ($10,X) means : load the accumulator with the contents of the memory location, whose address is stored in $10 and $11. Analogous with the statement LDA ($10),Y if Y = 0.

If the contents of these addresses are changed, you can load the accumulator with the contents of different locations. We will use this technique to do a blocktransfer of not just 256, but 4k byte from $4000 to $5000.

```
                         ORG    $C100
                 CLR     EQU    $00
                 LOS     EQU    $FB
                 LOD     EQU    $FD
                 HIS     EQU    $FC
                 HID     EQU    $FE
C100: A200               LDX    #CLR
C102: 86FB               STX    LOS
C104: 86FD               STX    LOD
C106: A940               LDA    #$40
C108: 85FC               STA    HIS
C10A: A950               LDA    #$50
C10C: 85FE               STA    HID
C10E: A1FB       M       LDA    (LOS,X)
C110: 81FD               STA    (LOD,X)
C112: E6FB               INC    LOS
C114: E6FD               INC    LOD
C116: D0F6               BNE    M
C118: E6FC               INC    HIS
C11A: E6FE               INC    HID
C11C: A5FC               LDA    HIS
C11E: C950               CMP    #$50
C120: D0EC               BNE    M
C122: 00                 BRK

PHYSICAL ENDADDRESS: $C123

*** NO WARNINGS

CLR                $00
```

```
LOD     $FD
HID     $FE
LOS     $FB
HIS     $FC
M       $C10E
```

Figure 5-3

In this program first the addresses for START ($FB, $FC) and DESTINATION ($FD, $FE) are defined. Second we load the accumulator with the contents of $4000 by LDA ($FB,X) and store it at $5000 with STA ($FD, X). Then we increment $FB and $FD by 1 until we reach the first address not to be moved.

Try the following two programs as an exercise :

1. Program FILL. A part of memory with the start address in $FB, $FC and the end address in $FD, $FE is to be filled with the hex number, which is stored in $02.

2. Program MOVE. A block of data (start address in $F9, $FA; end address in $FB, $FC) should be moved to another area (start address in $FD, $FE). This block may be at any location, even within the area of the block to be moved itself. This is not possible by the techniques used before.

Notes to part 5 :

* indexed addressing
* indexed indirect addressing
* indirect indexed addressing
* transfer of data within memory

NOTES

6

Part 6

In this chapter we will talk about the input of data (characters, numbers) into the computer. The data should be entered with the keyboard. All computers with a keyboard are equipped with a subroutine for the input of a character from the keyboard. Most times this routine is called GETCHR or CHRIN. Usually the ASCII code or a a similar code (for example ATASCII on the ATARI) is used with these characters. An 'A' in the ASCII code for instance is $41. This coding is used, for example, with the C1P and the PET. The APPLE computer uses $C1 (all normal displayed characters have bit 8 = 1). It follows that you have to be careful if you want to transfer machine language programs from one computer to another one !
With the COMMODORE 64 a check, whether 'A' was pressed looks as follows :

```
JSR CHRIN
CMP #$41
```

With the APPLE the same would look as follows :

```
JSR GETCHR
CMP #$C1
```

If the input of data is used very often, then a 'menu' is sometimes used. This technique, that you will know from BASIC, is possible also in machine-language. A text is displayed on the screen and the program waits for an input from the keyboard. It then branches depending on the input. We will show the whole program in a flowchart. A flowchart explains the structure of a program through the use of graphic symbols.

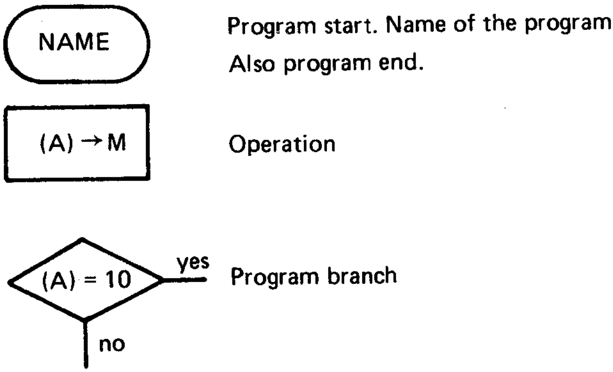


Figure 6–1 elements of a flowchart

The flowchart in figure 6–2 shows the structure of our program. The program first prints the text and then waits for a key to be pressed. If A, B, or E has been pressed, the program branches to the matching part. If another key has been pressed, the computer will beep and wait for another input.

This may sound simple to you, but a menu always should consider these two things :

1. The end of the program should be layed down. This means a stop of the program other than with RESET or switching off should be possible.

2. Input errors should be tied up; a warning should appear on the screen or an acustic sign (bell) should mark the error.

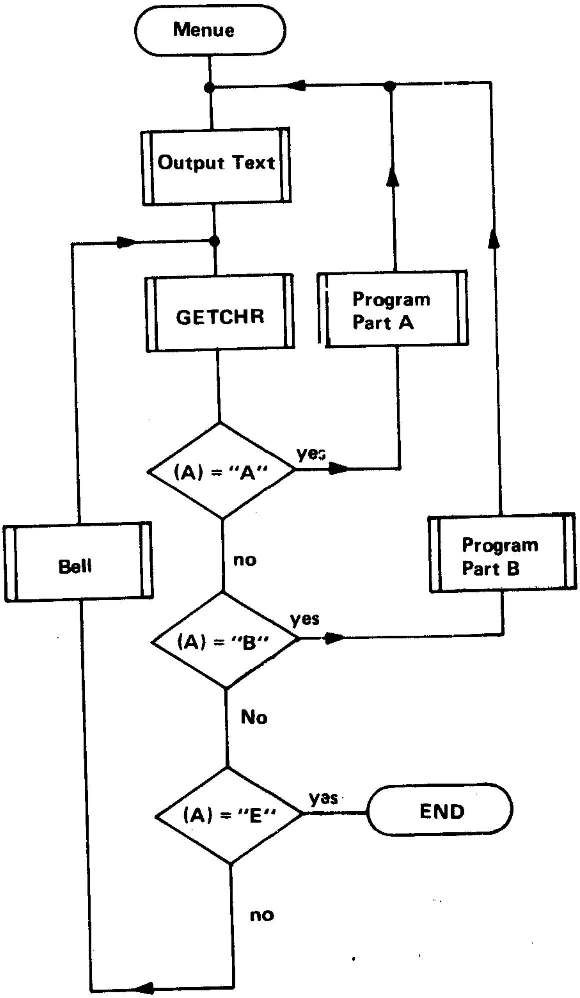


Figure 6-2 Flowchart of a menu program

Here is the program.
First the screen is cleared, then the text is printed. The text is stored at memory locations starting at $C140 and is printed by the subroutine TXTOUT.
The listing contains a few commands which are not CPU statements. These pseudo statements are for the assembler. We will talk about pseudo opcodes later.

```
            * MENU

                        ORG     $C100
            PLOT        EQU     $FFF0
            CHRIN       EQU     $FFCF
            CHROUT      EQU     $FFD2
C100: A993  MENU        LDA     #$93
C102: 20D2FF            JSR     CHROUT
C105: 202EC1 MENU1      JSR     TXTOUT
C108: A900              LDA     #$00
C10A: 20CFFF            JSR     CHRIN
C10D: C941              CMP     #$41
C10F: D006              BNE     MENU2
C111: 206AC1            JSR     A
C114: 18                CLC
C115: 90EE              BCC     MENU1
C117: C942  MENU2       CMP     #$42
C119: D006              BNE     MENU3
C11B: 207EC1            JSR     B
C11E: 18                CLC
C11F: 90E4              BCC     MENU1
C121: C945  MENU3       CMP     #$45
C123: D001              BNE     MENU4
C125: 00                BRK
C126: A907  MENU4       LDA     #$07
C128: 20D2FF            JSR     CHROUT
C12B: 18                CLC
C12C: 90D7              BCC     MENU1
C12E: A202  TXTOUT      LDX     #$02
C130: A003              LDY     #$03
C132: 18                CLC
C133: 20F0FF            JSR     PLOT
C136: A200              LDX     #0
C138: BD47C1 TX         LDA     TEXT,X
C13B: C99B              CMP     #$9B
C13D: F007              BEQ     TE
C13F: 20D2FF            JSR     CHROUT
C142: E8                INX
C143: 4C38C1            JMP     TX
C146: 60    TE          RTS
C147: 50524F TEXT       ASC     "PROGRAM (A)   "
C14A: 475241
C14D: 4D2028
```

```
C150: 412920
C153: 20
C154: 50524F                 ASC      "PROGRAM (B)  "
C157: 475241
C15A: 4D2028
C15D: 422920
C160: 20
C161: 454E44                 ASC      "END (E) "
C164: 202845
C167: 2920
C169: 9B                     DFB      $9B
C16A: A90D    A              LDA      #$0D
C16C: 20D2FF                 JSR      CHROUT
C16F: A205                   LDX      #5
C171: A941    AA             LDA      #$41
C173: 86FE                   STX      $FE
C175: 20D2FF                 JSR      CHROUT
C178: A6FE                   LDX      $FE
C17A: CA                     DEX
C17B: D0F4                   BNE      AA
C17D: 60                     RTS
C17E: A90D    B              LDA      #$0D
C180: 20D2FF                 JSR      CHROUT
C183: A205                   LDX      #5
C185: A942    BB             LDA      #$42
C187: 86FE                   STX      $FE
C189: 20D2FF                 JSR      CHROUT
C18C: A6FE                   LDX      $FE
C18E: CA                     DEX
C18F: D0F4                   BNE      BB
C191: 60                     RTS

PHYSICAL ENDADDRESS: $C192

*** NO WARNINGS

PLOT                $FFF0
CHROUT              $FFD2
MENU1               $C105
MENU3               $C121
TXTOUT              $C12E
TE                  $C146
A                   $C16A
```

B	$C17E	
CHRIN	$FFCF	
MENU	$C100	UNUSED
MENU2	$C117	
MENU4	$C126	
TX	$C138	
TEXT	$C147	
AA	$C171	
BB	$C185	

Figure 6-3 A menu program

Notes to part 6:
* input of text
* logic flowchart
* elements of a logic flowchart

7

Part 7

This chapter deals with the input of numbers.

7-1 Input of a hex number

For the input we use subroutine CHRIN. Subroutine PACK then checks the input [0 - 9, A - F]. If the character is not a hex number, then the program leaves the input mode, having the ASCII character in the accumulator. The following figure shows the logic flowchart of PACK.

The ASCII character has to be in the accumulator, when the subroutine is entered. First the character is compared to 0, then to F. If it is smaller than 0 or greater than F, it is not a hexadecimal number. For the other characters between 0 and F, two other comparisons are to be made. If the character is smaller than ':', then it is a number between 0 and 9. If it is not smaller than A, then it is a number between A and F. In this case 9 will be added to the number. 'A' is $41. With the addition of 9 the lower four bits then represent a 10. By shifting the contents of the accumulator to the left four times this number gets into the four higher bits. Next the contents of the accumulator and locations INL and INH are shifted left by ROL (four times). Bit 7 gets shifted to bit 0 via the carry bit. After that the four lower bits of the accumulator are the four lower bits of location INL. The program for that is shown in figure 7-2.

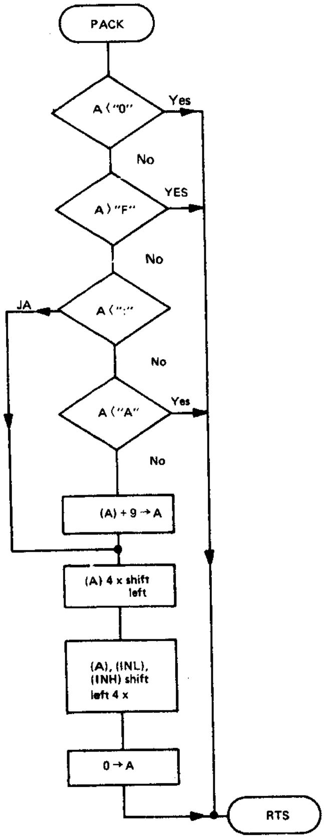


Figure 7-1 Logic flowchart of PACK

The program for the input is shown in figure 7-3. The two memory locations INL and INH are set to 0. For this reason you only have to enter 4F for number 004F. For the input we use subroutine CHRIN. GETWD (start address $C124) will be executed, until a non-hexadecimal number is entered.

7-2 Input of a decimal number

Now we want to enter a decimal number and convert it into a hexadecimal number.

```
                        * PACKHEX

                                ORG      $C100
                        CHRIN   EQU      $FFCF
                        PRTBYT  EQU      $C000
                        INL     EQU      $FB
                        INH     EQU      $FC
C100: C930              PACK    CMP      #$30
C102: 301F                      BMI      PACKEND
C104: C946                      CMP      #$46
C106: 101B                      BPL      PACKEND
C108: C93A                      CMP      #$3A
C10A: 3007                      BMI      CALC
C10C: C941                      CMP      #$41
C10E: 3013                      BMI      PACKEND
C110: 18                        CLC
C111: 6909                      ADC      #$09
C113: 0A                CALC    ASL
C114: 0A                        ASL
C115: 0A                        ASL
C116: 0A                        ASL
C117: A004                      LDY      #$04
C119: 2A                M1      ROL
C11A: 26FB                      ROL      INL
C11C: 26FC                      ROL      INH
C11E: 88                        DEY
C11F: D0F8                      BNE      M1
C121: A900                      LDA      #$00
C123: 60                PACKEND RTS
```

Figure 7-2 PACK

```
C124: A900       HEXINP  LDA      #$00
C126: 85FB               STA      INL
C128: 85FC               STA      INH
C12A: 20CFFF M2          JSR      CHRIN
```

```
C12D: 2000C1                JSR      PACK
C130: D009                  BNE      INPEND
C132: A5FB                  LDA      INL
C134: 290F                  AND      #$0F
C136: 2000C0                JSR      PRTBYT
C139: 10EF                  BPL      M2
C13B: 60        INPEND      RTS
C13C: 00                    BRK

PHYSICAL ENDADDRESS: $C13D

*** NO WARNINGS

CHRIN               $FFCF
INL                 $FB
PACK                $C100
M1                  $C119
HEXINP              $C124       UNUSED
INPEND              $C13B
PRTBYT              $C000
INH                 $FC
CALC                $C113
PACKEND             $C123
M2                  $C12A
```

Figure 7-3 Input of a hex number

The character entered is checked to see if it is a digit, inclusive, 0 through 9. The content of the input buffer is then multiplied by 10 and the new number is added.

Since the 6510 CPU doesn't have a command for multiplication we have to do that another way. One way would be to add the number 10 times. We however, use a different technique. A shift left command corresponds with a multiplication by two.

```
Example :    6 = %00000110
                 %00001100 = 12
```

The number is stored and shifted left two times, which means a multiplication by 4. Next the original number is added so that we now have five times the original number. The final step in multiplying by 10 consists of one more shift left. The program to do this is shown in figure 7-4.

```
                      ORG     $C100
              D0      EQU     $02
              D1      EQU     $FB
              D2      EQU     $FC
              D3      EQU     $FD
              D4      EQU     $FE
              CHRIN   EQU     $FFCF
              CHROUT  EQU     $FFD2
C100: A900    DEZINP  LDA     #$00
C102: 8502            STA     D0
C104: 85FB            STA     D1
C106: 20CFFF  L1      JSR     CHRIN
C109: 20D2FF          JSR     CHROUT
C10C: C930            CMP     #$30
C10E: 303B            BMI     L5
C110: C939            CMP     #$39
C112: 1037            BPL     L5
C114: 290F            AND     #$0F
C116: 2024C1          JSR     L3
C119: 18              CLC
C11A: 6502            ADC     D0
C11C: 8502            STA     D0
C11E: 9002            BCC     L2
C120: E6FB            INC     D1
C122: 90E2    L2      BCC     L1
C124: 85FC    L3      STA     D2
C126: A502            LDA     D0
C128: 85FD            STA     D3
C12A: A5FB            LDA     D1
C12C: 85FE            STA     D4
C12E: 2602            ROL     D0
C130: 26FB            ROL     D1
C132: 2602            ROL     D0
C134: 26FB            ROL     D1
```

```
C136: A502                LDA     D0
C138: 18                  CLC
C139: 65FD                ADC     D3
C13B: 8502                STA     D0
C13D: A5FB                LDA     D1
C13F: 65FE                ADC     D4
C141: 2602                ROL     D0
C143: 26FB                ROL     D1
C145: B003                BCS     L4
C147: A5FC                LDA     D2
C149: 60                  RTS
C14A: 00        L4        BRK
C14B: A99B      L5        LDA     #$9B
C14D: 20D2FF              JSR     CHROUT
C150: A5FB                LDA     D1
C152: 2000C0              JSR     $C000
C155: A502                LDA     D0
C157: 2000C0              JSR     $C000
C15A: 00                  BRK

PHYSICAL ENDADDRESS: $C15B

*** NO WARNINGS

D0                  $02
D2                  $FC
D4                  $FE
CHROUT              $FFD2
L1                  $C106
L3                  $C124
L5                  $C14B
D1                  $FB
D3                  $FD
CHRIN               $FFCF
DEZINP              $C100      UNUSED
L2                  $C122
L4                  $C14A
```

Figure 7-4 : Input of a decimal number

The program PACK (figure 7-2) uses a loop four times with ROL, ROL INL, ROL INH. This corresponds with a multiplication by 16, which is necessary with the input of hexadecimal numbers.

Notes to part 7 :

* input of a hexadecimal number
* input of a decimal number
* multiplication by 10

NOTES

8

Part 8

When you program in machine language you will use an assembler most times. An assembler is a program, which translates the mnemonic code into machine code. For example it will translate LDA #$05 into the two bytes A9 05.

An assembler also allows you to use symbolic names. If the name PORTA appears in a program, the assembler has to write in the address previously defined for PORTA. It also has to take notice of labels.

For example :

```
LDA PORTA
BNE M1
LDA PORTB
M1 STA HFZ
.
.
```

The assembler automatically calculates the number of bytes from BNE M1 to the label M1.

Assemblers usually consist of two parts. The first part is a text editor for entering the source-code.

There are text editors, where the source-code has to be entered with line numbers, while others don't require them. With most assemblers, labels have to start with a letter and have to be in the first position. Commands have to be in the second position. Labels and names usually can be up to six characters long.

After the source code has been entered, the assembler translates it into machine-code. To do that it needs additional information, so-called pseudo-commands. These pseudo-commands only affect the assembler, not the program itself. Unfortunately these commands are different on most assemblers, but most assemblers use the following pseudo-commands :

1. ORG

The command ORG (ORIGIN) defines the start address of the machine-code.

ORG $4000

means, that the code of the first line translated will start at location $4000.

This address also is the base address for the program starting there. All absolute addresses refer to that address. An ORG command always has to be at the beginning of the assembler text, but it is possible to change it within the text.

Example :

```
ORG $2000
<TEXT 1>
ORG $500
<TEXT 2>
```

The code of text 1 starts at address $2000. The code of text 2 starts at address $500. The machine code is often called the object code.

2. OBJ

The command OBJ allows you to store the machine-code at a different location in memory.

Example :

```
ORG $3000
OBJ $2500
```

The program will be translated with all absolute addresses referring to $3000, but the machine-code will be stored at addresses starting at $2500. If you want to start the program later, you first have to move it to $3000 with a blocktransfer.

With MACROFIRE the same command looks as follows :

```
ORG $3000,$2500
        :     :
        :  physical adddress
        :
     logical address
```

3. END

The command END shows the assembler that the text to be translated ends here.

4. EQU

With this command a certain address gets a symbolic name.

Example : PORTA EQU $COCO

The symbolic name PORTA corresponds with the address $COCO.
In this case PORTA is used as a label and, by that, has to be in the first position in the text.

Some assemblers need an extra command for addresses from the zero-page.

```
HFZ EPZ $10
```

The name HFZ corresponds with address $10 of the zero-page.
Some assemblers use the equal sign (=) instead of EQU.

5. HEX

With command HEX you can store hexadecimal numbers within a program.

Example :

```
DATA HEX 00AFFC05
```

The numbers 00 AF FC 05 are stored in four consecutive locations starting at the symbolic address DATA.

6. ASC

If you want to store text within a program, you can use command ASC.

Example : TEXT ASC "THIS IS A TEXT"

The text between the quotation marks is stored in ASCII code at address TEXT.

Some assemblers use the command BYT.

BYT 0045AF corresponds with HEX 0045AF.

BYT "TEXT" corresponds with ASC "TEXT".

For more information on the different pseudo commands please check with the manual for the assembler.

It is possible to do calculations in the address section. The following program portion shows a pseudo instruction :

```
DATA HEX 00AFFC05
```

The command LDA DATA will load 00, LDA DATA+2 will load FC.

Be careful, if you use address calculation with relative jumps.

```
BNE *+2
```

The above example causes the program to jump two bytes, but not two lines in the text.
With some assemblers the * is a pseudo command, or a pseudo address. It tells you the present value in the program counter.

Example :

```
LDA HFZ
BNE *+2
LDA #$FF
STA HFZ
```

If the contents of HFZ is different from zero, then the command LDA #$FF is jumped.
Some assemblers allow all four basic arithmetic operations, but in most cases addition and subtraction will be enough.

The following is offered to the reader as a programming hint :

When in the program there is line : H EQU $2F

then LDA H means, load the accumulator with the contents of $2F, but LDA #H means, load the accumulator with $2F.

Notes to part 8 :

* pseudo commands
* address calculations

9

Part 9

In this, the last chapter we will discuss some helpful suggestions and short cuts.
There are some programs, where you want the program to determine, where in memory it is located. This becomes necessary with programs which contain absolute addresses, but can run at any location in memory. With the APPLE for example, this trick is used to determine into which slot a peripheral board is plugged. Since there is no command which enables you to read the program counter, we use the following trick :
The program contains a JSR-command right to a RTS in the monitor. The present address is thereby written to the stack. You have to take into consideration, however, that the lower byte of the address is lowered by one. Figure 9-1 shows the stack pointer before, during, and after the jump to the subroutine.

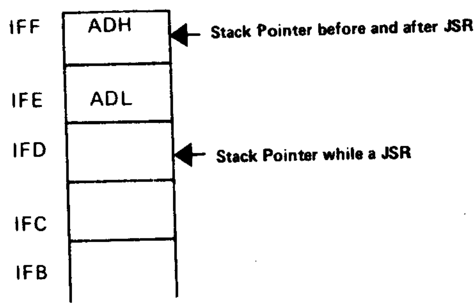


Figure 9-1 : stack pointer during JSR

After the return to the main program you can bring the contents of the stack pointer to register X with TSX. Then you can access address ADH as shown in figure 2.

You also can program another way, with an indirect jump JMP (ADR) as follows :

Let's assume, that the indirect jump should go to $2010. This can be done with the following program :

```
LDA #$20
PHA
LDA #$0F
PHA
RTS
```

You can find this technique in the operating system of C-64. Usually an indirect jump is programmed the following way :

```
LDA #$10
STA ADR
LDA #$20
STA ADR+1
JMP (ADR)
```

If you use an address in the zero page, then the first program is four bytes shorter. If you use any address, then the first program is six bytes shorter than the second one. Here is a comparison of the execution times :

LDA #$20	2	LDA #$10	2	2
PHA	3	STA ADR	3	4
LDA #$0F	2	LDA #$20	2	2
PHA	3	STA ADR+1	3	4
RTS	6	JMP (ADR)	5	5
	16		15	16

The numbers, after the commands, means the number of machine cycles required for this command. For the second program, the first column is an address in the zero page. The second column is for any address. You can find the number of cycles for the single commands in the reference card of the 6510 (6502) microprocessor.

Usually one doesn't think much about execution time, exept with loops which occure frequently.
To that a comparison of two program parts for relocation of data. Only the part which is different is compared. The rest is the same with both programs.

1st program

```
LDA (FROM,X)     6
STA (TO,X)       6
INC FROM         5
BNE M            2 (+1)
INC FROM+1       5
M INC TO         5
BNE M1           2 (+1)
INC TO+1         5
M1              ------
                36
```

The program needs 36 cycles, if no branches are executed. If a branch is executed, then one more cycle is used.

2nd program

```
MEM LDA FROM     4
STA TO           4
INC MEM+1        5
BNE M            2 (+1)
INC MEM+2        5
M INC MEM+4      5
BNE M1           2 (+1)
INC MEM+5        5
M1              ------
                32
```

The second program requires four cycles less, but it is a program that changes itself. Location MEM+1 contains the lower byte and location MEM+2 contains the higher byte of the command LDA FROM. This program does not work in ROM, it has to be in RAM.
The savings of 4 cycles, which corresponds with 4 microseconds if the clock frequency is 1 megahertz, doesn't look great, but it accumulates with the transfer of large quantities of data.

If, in a subroutine, there is a call of another subroutine immediately before the RTS command, then you can save seven cycles, if you replace the JSR command by a JMP command,
rather than :

```
JSR TO
RTS
```

use just :

```
JMP TO
```

The RTS command in subroutine TO brings you back to the same location as the RTS after JSR TO.

The processor 6510 has an indirect jump :
JMP (ADR), but no indirect jump to a subroutine :
JSR (ADR).

This is needed, if you want to jump to different subroutines, depending upon conditions, similar to the ON...GOTO instruction in BASIC.

If the program is in RAM, then you could use a self-modifying program, which changes the address after JSR. If the program is in ROM, then you can use the following trick.
Somewhere in memory there is a command
JMP1 JMP(ADR) 6C XX XX.
Instead of XX XX you write in the address of the

subroutine to be executed. You call the subroutine with

```
JSR JMP1
```

The RTS command in the subroutine brings you back to the command following JSR JMP1.

NOTES

KERNAL Routines 10

KERNAL-Routines

In most programs listed in this book you will find a call of the two subroutines CHRIN and CHROUT. These routines, among other ones, are resident in the ROM of your COMMODORE 64. Following is a description of the most important of these routines.

CHRIN, Input of a character ($FFCF)

This routine waits for an input from a device. Unless set otherwise this device is the keyboard. The routine stores all characters entered in the system input buffer (starting at $200). The routine returns to the main program with the last character entered in the accumulator after a carriage return has been received.

CHROUT, Output of a character ($FFD2)

This routine sends the contents of the accumulator (ASCII character) to a device. Unless set otherwise this device is the screen. For example if you want to print an 'A' on the screen, use :

```
LDA #$41
JSR CHROUT
```

GETIN, Input of a character ($FFE4)

This routine gets a character from the keyboard queue, which can contain up to ten characters. If the queue is empty, the program returns with 0 in the accumulator.

PLOT, Place cursor ($FFF0)

This routine allows you to place the cursor at a certain location on the screen, or to read the present location of the cursor. If you call the routine with the carry flag set, then register Y will contain the column number and register X will contain the row number of the cursor position after the return from the subroutine. A call of the routine with the carry bit clear will place the cursor at a position determined by the contents of registers Y and X. For example :

```
LDY #$5
LDX #$8
CLC
JSR PLOT
```

This will place the cursor at column 5, row 8.

RDTIM, Read clock ($FFDE)

This routine returns the present reading of the system clock as three bytes, with the most significant byte in the accumulator, the next significant byte in the X register, and the least significant byte in the Y register.

SETTIM, Set clock ($FFDB)

This routine sets the system clock to the time defined by the contents of the accumulator, register X, and register Y.

Calling Machine Language Routines From BASIC

Calling Machine Language Routines From BASIC

There are two commands that allow you to call a machine language program from a BASIC program. These commands are SYS X and USR(X).

Command SYS X jumps to the machine language program located at address X (decimal). For example if you want to call a machine language program which you have placed at $C000 use : SYS 49152.

Command USR (X) calls a machine language program at an address defined by the contents of locations 785 (lower byte) and 786 (higher byte). For example if you have two machine language programs, one located at $C000, which should be called if variable V is less than 10, and another one located at $C800, which should be called, if variable V is equal or greater than 10, use the following BASIC program (the machine language program has to be in memory when you call it, of course) :

```
.
.
200 IF V>9 THEN 230
210 POKE 785,0:POKE 786,192
```

```
220 X=USR(0):GOTO 250
230 POKE 785,0:POKE 786,200
240 X=USR(0)
   .
   .
```

Command USR(X) allows to hand over a parameter to the machine language program, for example the command X=USR(10) will hand over the number 10 to the machine language program through the floating point accumulator (starting at address $61) and a value can be returned to the variable X, if the machine language program places the value in the floating point accumulator before returning to BASIC.

Note :
In both cases the machine language program has to end with an RTS ($60), in order to return to BASIC.

Where to put machine language programs

As said earlier the COMMODORE 64, in its standard memory configuration, reserves the RAM at addresses $C000 through $CFFF (49152 through 53247) for your machine language programs. In case these 4k of RAM are not enough, you have to "steal" something from the area that's normally reserved for BASIC programs, by defining a new address for top of memory (normally $9FFF). To do that you have to POKE the new address into locations 51, 52, and 55, 56. For example if you need additional 2k of RAM, set top of memory to $9800 by the following command :

```
POKE 51,0:POKE 52,152:POKE 55,0:POKE 56,152:CLR
```

This gives you a total of 6k of RAM for your machine language programs :

$9800-$9FFF (2k) and
$C000-$CFFF (4k).

If you are using an assembler, check the manual for where you can place your machine language programs, so that your program will not overlap with the assembler program.

NOTES

Examples In Machine Language

Examples In Machine Language

The following short programs are examples in machine language, together with their equivalent BASIC programs.

The first program prints one row of character C at the top of the screen.

The second program fills the screen with the character entered.

The third program allows you to change colors. If you enter 'B', the background color will change, if you enter 'S', the screen color will change, if you enter 'R', the original colors will be restored.

CROW

```
                 *CROW

                           ORG        $C100
                 CHROUT    EQU        $FFD2
                 CHRIN     EQU        $FFCF
                 AUX       EPZ        $FB
C100: 4C08C1               JMP        START
C103: A993       CLEAR     LDA        #$93
C105: 4CD2FF               JMP        CHROUT
```

```
C108: 2003C1 START     JSR     CLEAR
C10B: A228             LDX     #40
C10D: 86FB   S1        STX     AUX
C10F: A943             LDA     'C'
C111: 20D2FF           JSR     CHROUT
C114: A6FB             LDX     AUX
C116: CA               DEX
C117: D0F4             BNE     S1
C119: 20CFFF           JSR     CHRIN
C11C: 00               BRK

PHYSICAL ENDADDRESS: $C11D

*** NO WARNINGS

CHROUT              $FFD2
AUX                 $FB
START               $C108
CHRIN               $FFCF
CLEAR               $C103
S1                  $C10D
```

CROWBAS

```
100 REM ROW OF CHARACTER C
110 PRINT""
120 FORX=1TO40
130 PRINT"C";
140 NEXTX
150 END
```

```
                        *SCREENCH

                                  ORG      $C100
                        CHROUT    EQU      $FFD2
                        CHRIN     EQU      $FFCF
                        AUX1      EPZ      $FB
                        AUX2      EPZ      $FE
C100: 4C08C1                      JMP      START
C103: A993              CLEAR     LDA      #$93
C105: 4CD2FF                      JMP      CHROUT
C108: 20CFFF            START     JSR      CHRIN
C10B: 85FE                        STA      AUX2
C10D: 2003C1                      JSR      CLEAR
C110: A019                        LDY      #25
C112: A228              S0        LDX      #40
C114: 86FB              S1        STX      AUX1
C116: A5FE                        LDA      AUX2
C118: 20D2FF                      JSR      CHROUT
C11B: A6FB                        LDX      AUX1
C11D: CA                          DEX
C11E: D0F4                        BNE      S1
C120: 88                          DEY
C121: D0EF                        BNE      S0
C123: 20CFFF                      JSR      CHRIN
C126: 00                          BRK

PHYSICAL ENDADDRESS: $C127

*** NO WARNINGS

CHROUT            $FFD2
AUX1              $FB
CLEAR             $C103
S0                $C112
CHRIN             $FFCF
AUX2              $FE
START             $C108
S1                $C114
```

SCREENCHBAS

```
100 REM SCREEN FULL OF CHARACTER
110 PRINT""
120 GET A$:IF A$=""THEN 120
130 FOR Y=1 TO 25
140 FOR X=1 TO 40
150 PRINT A$;
160 NEXT X
170 NEXT Y
180 GOTO 180
```

SETCOL

```
                   *SETCOL

                          ORG    $C100
                   CHRIN  EQU    $FFCF
                   COLOR  EQU    $D020
                   AUX    EPZ    $FB
C100: 4C0EC1              JMP    START
C103: AD20D0 COLSAV       LDA    COLOR
C106: 85FB                STA    AUX
C108: AD21D0              LDA    COLOR+1
C10B: 85FC                STA    AUX+1
C10D: 60                  RTS
C10E: 2003C1 START        JSR    COLSAV
C111: 20CFFF S0           JSR    CHRIN
C114: C942                CMP    'B'
C116: D003                BNE    S1
C118: 202CC1              JSR    BCOLOR
C11B: C953   S1           CMP    'S'
C11D: D003                BNE    S2
C11F: 2048C1              JSR    SCOLOR
C122: C952   S2           CMP    'R'
C124: D003                BNE    S3
C126: 4C64C1              JMP    RCOLOR
C129: 18     S3           CLC
C12A: 90E5                BCC    S0
C12C: AD20D0 BCOLOR       LDA    COLOR
C12F: 290F                AND    #$0F
C131: C90F                CMP    #$0F
```

```
C133: D009              BNE     B1
C135: AD20D0            LDA     COLOR
C138: 29F0              AND     #$F0
C13A: 8D20D0            STA     COLOR
C13D: 60                RTS
C13E: AD20D0 B1         LDA     COLOR
C141: 18                CLC
C142: 6901              ADC     #1
C144: 8D20D0            STA     COLOR
C147: 60                RTS
C148: AD21D0 SCOLOR     LDA     COLOR+1
C14B: 290F              AND     #$0F
C14D: C90F              CMP     #$0F
C14F: D009              BNE     SC1
C151: AD21D0            LDA     COLOR+1
C154: 29F0              AND     #$F0
C156: 8D21D0            STA     COLOR+1
C159: 60                RTS
C15A: AD21D0 SC1        LDA     COLOR+1
C15D: 18                CLC
C15E: 6901              ADC     #1
C160: 8D21D0            STA     COLOR+1
C163: 60                RTS
C164: A5FB   RCOLOR     LDA     AUX
C166: 8D20D0            STA     COLOR
C169: A5FC              LDA     AUX+1
C16B: 8D21D0            STA     COLOR+1
C16E: 00                BRK

PHYSICAL ENDADDRESS: $C16F

*** NO WARNINGS

CHRIN           $FFCF
AUX             $FB
START           $C10E
S1              $C11B
S3              $C129
B1              $C13E
SC1             $C15A
COLOR           $D020
COLSAV          $C103
S0              $C111
```

S2	$C122
BCOLOR	$C12C
SCOLOR	$C148
RCOLOR	$C164

SETCOLBAS

```
100 REM BORDER AND SCREEN COLOR
110 BO=53280
120 SC=53281
130 A=PEEK(BO)
140 B=PEEK(SC)
150 GET A$:IF A$=""THEN 150
160 IFA$<>"B"THEN200
170 IF(PEEK(BO)AND15)=15THENPOKEBO,PEEK(BO)AND240:
    GOTO150
180 POKEBO,PEEK(BO)+1
190 GOTO150
200 IF A$<>"S"THEN 240
210 IF(PEEK(SC)AND15)=15THENPOKESC,PEEK(SC)AND240:
    GOTO150
220 POKESC,PEEK(SC)+1
230 GOTO150
240 IFA$<>"R"THEN150
250 POKEBO,A
260 POKESC,B
270 END
```

RELOCATOR

RELOCATOR

This program allows you to move machine code from one part of memory to another one. You can chose between a blocktransfer, where every byte is transfered to its new location without change, or a relocation, where it is checked, whether there are absolute addresses, and if there are any they are converted for the new location in memory. For example if you relocate a program from addresses \$4000 through \$4100 to \$5000 and there is a command JMP \$4020, this command will be changed into JMP \$5020 by the relocator.

When relocating a program, you have to check for tables and text in your program, because the relocator may interpret parts therof as opcode and change it.

Before you start the program at address \$C100 you have to define the start address, the end address, and the destination address of the program to be relocated. You also have to define the lower and upper address of memory available. This will protect certain areas of memory from being overwritten by tranfered program.

Here is a table of the zero page locations, that have to be set before starting the program.

Memory location		Label	Remarks
7C		RFLAG	0=relocate 1=blocktransfer
7D	LSB	TEST1	lower address of
7E	MSB		memory available
7F	LSB	TEST2	upper address of
80	MSB		memory available
81	LSB	START	start address of
82	MSB		program to be moved
83	LSB	STOP	end address of
84	MSB		program to be moved
85	LSB	BEG	destination address
86	MSB		

```
                  *RELOC

                           ORG     $C100
                  RFLAG    EQU     $7C
                  TEST1    EQU     $7D
                  TEST2    EQU     $7F
                  START    EQU     $81
                  STOP     EQU     $83
                  BEG      EQU     $85
                  OPTR     EQU     $87
                  TEMP2    EQU     $89
                  NPTR     EQU     $8B
                  TEMP1    EQU     $8D
C100: A205        BEGIN    LDX     #$5
C102: B581        S10      LDA     START,X
C104: 9587                 STA     OPTR,X
C106: CA                   DEX
C107: 10F9                 BPL     S10
C109: E8                   INX
C10A: A57C        MOVE     LDA     RFLAG
```

```
C10C: F006              BEQ     M01
C10E: 204EC1            JSR     MOV1
C111: 4C5FC1            JMP     DONE
C114: A187    M01       LDA     (OPTR,X)
C116: A8                TAY
C117: D006              BNE     M02
C119: 2052C1            JSR     SKIP
C11C: 4C5FC1            JMP     DONE
C11F: 204EC1  M02       JSR     MOV1
C122: C920              CMP     #$20
C124: D003              BNE     BYTE1
C126: 4C79C1            JMP     BYTE3
C129: 98      BYTE1     TYA
C12A: 299F              AND     #$9F
C12C: F031              BEQ     DONE
C12E: 98                TYA
C12F: 291D              AND     #$1D
C131: C908              CMP     #$8
C133: F02A              BEQ     DONE
C135: C918              CMP     #$18
C137: F026              BEQ     DONE
C139: 98                TYA
C13A: 291C              AND     #$1C
C13C: C91C              CMP     #$1C
C13E: F039              BEQ     BYTE3
C140: C918              CMP     #$18
C142: F035              BEQ     BYTE3
C144: C90C              CMP     #$0C
C146: F031              BEQ     BYTE3
C148: 204EC1            JSR     MOV1
C14B: 4C5FC1            JMP     DONE
C14E: A187    MOV1      LDA     (OPTR,X)
C150: 818B              STA     (NPTR,X)
C152: 20D9C1  SKIP      JSR     IOPTR
C155: 20E0C1            JSR     INPTR
C158: 60                RTS
C159: 204EC1  MOV2      JSR     MOV1
C15C: 204EC1            JSR     MOV1
C15F: A587    DONE      LDA     OPTR
C161: 858D              STA     TEMP1
C163: A588              LDA     OPTR+1
C165: 858E              STA     TEMP1+1
C167: A583              LDA     STOP
```

```
C169:  8589              STA    TEMP2
C16B:  A584              LDA    STOP+1
C16D:  858A              STA    TEMP2+1
C16F:  20CEC1            JSR    TEST
C172:  9096              BCC    MOVE
C174:  F094              BEQ    MOVE
C176:  00                BRK
C177:  EA                NOP
C178:  EA                NOP
C179:  A187     BYTE3    LDA    (OPTR,X)
C17B:  858D              STA    TEMP1
C17D:  20D9C1            JSR    IOPTR
C180:  A187              LDA    (OPTR,X)
C182:  858E              STA    TEMP1+1
C184:  20E7C1            JSR    DOPTR
C187:  A57D              LDA    TEST1
C189:  8589              STA    TEMP2
C18B:  A57E              LDA    TEST1+1
C18D:  858A              STA    TEMP2+1
C18F:  20CEC1            JSR    TEST
C192:  F002              BEQ    B10
C194:  90C3              BCC    MOV2
C196:  A57F     B10      LDA    TEST2
C198:  8589              STA    TEMP2
C19A:  A580              LDA    TEST2+1
C19C:  858A              STA    TEMP2+1
C19E:  20CEC1            JSR    TEST
C1A1:  F002              BEQ    B20
C1A3:  B0B4              BCS    MOV2
C1A5:  38       B20      SEC
C1A6:  A187              LDA    (OPTR,X)
C1A8:  E581              SBC    START
C1AA:  8589              STA    TEMP2
C1AC:  20D9C1            JSR    IOPTR
C1AF:  A187              LDA    (OPTR,X)
C1B1:  E582              SBC    START+1
C1B3:  858A              STA    TEMP2+1
C1B5:  20D9C1            JSR    IOPTR
C1B8:  18                CLC
C1B9:  A589              LDA    TEMP2
C1BB:  6585              ADC    BEG
C1BD:  818B              STA    (NPTR,X)
C1BF:  20E0C1            JSR    INPTR
```

```
C1C2: A58A              LDA     TEMP2+1
C1C4: 6586              ADC     BEG+1
C1C6: 818B              STA     (NPTR,X)
C1C8: 20E0C1            JSR     INPTR
C1CB: 4C5FC1            JMP     DONE
C1CE: A58E    TEST      LDA     TEMP1+1
C1D0: C58A              CMP     TEMP2+1
C1D2: D004              BNE     T10
C1D4: A58D              LDA     TEMP1
C1D6: C589              CMP     TEMP2
C1D8: 60      T10       RTS
C1D9: E687    IOPTR     INC     OPTR
C1DB: D002              BNE     INC10
C1DD: E688              INC     OPTR+1
C1DF: 60      INC10     RTS
C1E0: E68B    INPTR     INC     NPTR
C1E2: D002              BNE     INC20
C1E4: E68C              INC     NPTR+1
C1E6: 60      INC20     RTS
C1E7: C687    DOPTR     DEC     OPTR
C1E9: A587              LDA     OPTR
C1EB: C9FF              CMP     #$FF
C1ED: D002              BNE     D10
C1EF: C688              DEC     OPTR+1
C1F1: 60      D10       RTS

PHYSICAL ENDADDRESS: $C1F2

*** NO WARNINGS

RFLAG           $7C
TEST2           $7F
STOP            $83
OPTR            $87
NPTR            $8B
BEGIN           $C100     UNUSED
MOVE            $C10A
MO2             $C11F
MOV1            $C14E
MOV2            $C159
BYTE3           $C179
B20             $C1A5
T10             $C1D8
```

INC10	$C1DF
INC20	$C1E6
D10	$C1F1
TEST1	$7D
START	$81
BEG	$85
TEMP2	$89
TEMP1	$8D
S10	$C102
M01	$C114
BYTE1	$C129
SKIP	$C152
DONE	$C15F
B10	$C196
TEST	$C1CE
IOPTR	$C1D9
INPTR	$C1E0
DOPTR	$C1E7

Random Number Generator

Random Number Generator

Randomness is required for many games like dice-games, maze-games, etc.

The programs listed below are based on a pseudo random shift register approach. Two bytes are used as a shift register (RNDM and RNDM+1). At least one of the locations RNDM or RNDM+1 has to be non-zero. Before starting the program, use the monitor to set one of these locations to a non-zero value.

After assembly you can start the program from the monitor with the G(OTO C100 command. The program will generate one random character and display its ASCII equivalent.

If called from BASIC the BRK command has to be replaced by an RTS command.

```
                  *RANDOM

                             ORG  $C100
                  CHROUT     EQU  $FFD2
                  RNDM       EPZ  $FB
C100: A5FE        RANDOM     LDA  $FE        ;SET ITERATIONS
C102: 48          R1         PHA             ;SAVE COUNTER
```

```
C103: A5FB        LDA RNDM    ;GET BYTE
C105: 2A          ROL
C106: 45FB        EOR RNDM    ;XOR BITS 13 & 14
C108: 2A          ROL
C109: 2A          ROL
C10A: 26FC        ROL RNDM+1 ;SHIFT BYTE
C10C: 26FB        ROL RNDM    ;SHIFT 2. BYTE
C10E: 68          PLA         ;GET COUNTER
C10F: 18          CLC
C110: 69FF        ADC #$FF    ;DECREMENT
C112: D0EE        BNE R1      ;IF NOT DONE DO AGAIN
C114: A5FB        LDA RNDM    ;GET RANDOM BYTE
C116: 20D2FF      JSR CHROUT ;PRINT
C119: 00          BRK

PHYSICAL ENDADDRESS: $C11A

*** NO WARNINGS

CHROUT          $FFD2
RANDOM          $C100    UNUSED
RNDM            $FB
R1              $C102
```

The following program is also a random number generator, but it will print 10 random characters rather than one.

Note : If you count less than 10 random characters then one character was a control character, for example DEL or HOME.

```
            *RANDOM10

                      ORG $C100
            CHROUT    EQU $FFD2
            RNDM      EPZ $FB
            COUNTER   EPZ $FD
C100: A900            LDA #0
C102: 85FD            STA COUNTER
```

```
C104: A5FE   RANDOM  LDA $FE      ;SET ITERATIONS
C106: 48     R1      PHA          ;SAVE COUNTER
C107: A5FB           LDA RNDM     ;GET BYTE
C109: 2A             ROL
C10A: 45FB           EOR RNDM     ;XOR BITS 13 & 14
C10C: 2A             ROL
C10D: 2A             ROL
C10E: 26FC           ROL RNDM+1   ;SHIFT BYTE
C110: 26FB           ROL RNDM     ;SHIFT 2. BYTE
C112: 68             PLA          ;GET COUNTER
C113: 18             CLC
C114: 69FF           ADC #$FF     ;DECREMENT
C116: D0EE           BNE R1       ;IF NOT DONE DO AGAIN
C118: A5FB           LDA RNDM     ;GET RANDOM BYTE
C11A: 20D2FF         JSR CHROUT   ;PRINT
C11D: E6FD           INC COUNTER
C11F: A90B           LDA #$0B
C121: C5FD           CMP COUNTER
C123: D0DF           BNE RANDOM
C125: 00             BRK

PHYSICAL ENDADDRESS: $C126

*** NO WARNINGS

CHROUT          $FFD2
COUNTER         $FD
R1              $C106
RNDM            $FB
RANDOM          $C104
```

Number Systems A

CHAPTER A : NUMBER SYSTEMS

In this chapter we will develop some straightforward mathematics, based on daily experience, which will make it much simpler to model the internal workings of microcomputers.

Decimal numbers
Quantity
Binary Numbers, BITS, and BYTES
Hexadecimal Numbers

DECIMAL NUMBERS, AND THE CONCEPT OF QUANTITY...

Western culture has adopted the ten arabic symbols: 0,1,2,3,4,5,6,7,8, and 9 to represent various quantities. Many other symbols are available to describe a particular quantity. For example, 'three' may be symbolized as three, 3, trois (French), III (Roman Numerals), etc.

With the exception of the Roman Numerals, the above examples refer to the DECIMAL, or BASE-TEN number system which we use daily. The base-ten system is charaterized by the ten symbols which are available to use in constructing symbolic representations of various quantities. For large (multi-digit) numbers, we combine several symbols, and assign each symbol a multiplier based upon it's position within the series of symbols. For example, we represent the number of eggs in a carton with the symbols '12'. The symbol on the far right side is in what we call the 'unit' position. The next symbol to the left is in what we call the 'tens' position, and represents the number of complete groups of ten eggs. The total number of eggs is equal to ten times the number in the tens position, plus one times the number in the unit's position. Were there another symbol to the left, that symbol would be multiplied by ten, and then ten again. (i.e. multiplied by one-hundred). Were there a symbol still further to the left, then that symbol would be accompanied by yet another multiplication by ten. (i.e. multiplied by one-thousand).

Summarizing, the base-ten (or decimal) number system is characterized by:

1). A basic set of TEN symbols (0-9).
2). Each digit positioned left of the unit position are accompanied by a multiplier, and that multiplier increases by a factor of TEN for every additional digit postion to the left.
3). Decimal numbers are NOT the only method of representing a quantity.

We will now explore some number systems commonly used in
association with computer systems. (They are harder for us, but
easier for the computer!).

BINARY NUMBERS...

Generally, computers do not deal directly with the symbols of the decimal number system. The computer is made up of combinations of circuits capable of presenting only two basic symbols (as opposed to ten). Logic circuits inside the computer represent one symbol with a high level voltage (often about five volts), and the other symbol with a low level voltage (often about zero volts).These states are often described with the symbols 'high' or '1' for the high voltage level, and the symbols 'low' or '0' for the low voltage level. Multiple digit binary numbers can therefore be represented by multiple wires, with each wire at either a '1' or a '0' voltage level. By drawing a parallel to the base-ten number system, we may define this to be a BASE-TWO (or BINARY) number system, summarized by the following characteristics:

1). A basic set of TWO symbols (1,2).
2). Each digit positioned left of the unit position are accompanied by a multiplier, and that multiplier increases by a factor of TWO for every additional digit postion to the left.

Significance of digit position, decimal numbers versus binary numbers:

DECIMAL(10000'S) (1000'S) (100'S) (10'S) (1'S)
BINARY (16'S) (8'S) (4'S) (2'S) (1'S)

Some examples of binary numbers follow.

TRIAL QUANTITY	BASE-2 (BINARY)	EXPLANATION OF BINARY
NONE	0	0 IN UNIT'S PLACE
ONE	1	1 IN UNIT'S PLACE
TWO	10	2 TIMES ONE IN TWO'S PLACE, PLUS ONE IN UNIT'S PLACE.
THREE	11	2 TIMES ONE IN TWO'S PLACE, PLUS ONE IN UNIT'S PLACE.
FOUR	100	2 TIMES 2 TIMES ONE IN FOUR'S PLACE, PLUS TWO TIMES ZERO IN TWO'S PLACE, PLUS ZERO IN UNIT'S PLACE.
FIVE	101	AS ABOVE, BUT ONE IN UNITS PLACE.

THIRTEEN	1101	AS ABOVE, BUT ADD 2 TIMES 2 TIMES 2 TIMES ONE IN THE EIGHT'S PLACE.

Note that in the decimal system, symbol position was used to represent multipliers of 1, 10, 100, 1000, 10000, etc. In the binary number system, symbol position is used to indicate multipliers of 1, 2, 4, 8, 16, 32, 64, 128, 256, etc.

Using the above multipliers, you should be able to convert the following binary numbers (left column) into the decimal numbers in the righthand column.

BINARY NUMBER SYMBOL	DECIMAL NUMBER SYMBOL
110	6
101000	40
1000000	64
111111	63
111110	62
111101	61
1111111	127

There is no real trick to reading binary numbers. If you desire to get the numbers into decimal form, then there is no avoiding the process of multiplying the appropriate digits by 1, 2, 4, 8, 16, etc., and adding up the results.

One digit of a binary number, or one wire in the computer, can represent only one of two possible states. Thus one digit certainly does not contain a great abundance of information. It is therefore appropriate that we refer to one digit of a binary number as a BIT. A bit may be either a one or a zero. Carrying this madness one more step, we refer to a group of 8 BITS (an 8 digit binary number) as a BYTE.

It is important to note that the binary number system is simply an alternative way to write a number, just as Roman Numerals provide an alternative way to write a number. In all cases, a given SYMBOL represents a QUANTITY, and the method we choose to write it is of secondary importance.

Hexadecimal Numbers

HEXADECIMAL NUMBERS...

The preceeding discussion of binary numbers demonstrated that binary symbols for large quantities become very cumbersome, due to the very large number of digits which must be used. This is the natural consequence of having only two possible symbols per digit. In the decimal number system, we had ten symbols available, and large quantities could be represented with relatively few digits. Ideally, we need a number system which provides us with a large number of symbols, while retaining a simple relationship to the on/off world of individual wires within the computer.

Note that a four bit number (four digit binary number) may represent any quantity from zero (0000) to fifteen (1111), for a total of sixteen possible combinations. Now suppose we assign a SINGLE letter or number to each of these combinations, as shown in the righthand column of the table below.

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

Don't be taken aback by the use of letter symbols to represent numbers. After all, we are making the rules here, and if we wish to use the symbol 'D' to represent a quantity of thirteen, then so be it.

The above sixteen symbols (0-9, and A-F) are the sixteen basic symbols of the HEXADECIMAL (or BASE-SIXTEEN!) number system. For multiple digit numbers, we once again start with the UNITS position. But now, each time we move one digit position to the left, we add a multiplication by sixteen.

DECIMAL	BINARY	HEXADECIMAL	EXPLANATION
15	1111	F	15 IN UNIT'S PLACE.
16	1 0000	10	1 IN 16'S PLACE.
17	1 0001	11	1 IN 16'S PLACE, PLUS 1 IN UNIT'S PLACE.
42	10 1010	2A	2 IN 16'S PLACE, PLUS 10 IN UNIT'S PLACE.
255	1111 1111	FF	15 IN 16'S PLACE, PLUS 15 IN UNIT'S PLACE.
256	1 0000 0000	100	1 IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS ZERO IN UNIT'S PLACE.
769	11 0000 0001	301	THREE IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS 1 IN UNIT'S PLACE.
783	11 0000 1111	30F	THREE IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS 15 IN UNIT'S PLACE.

The HEXADECIMAL (BASE-SIXTEEN) number system may be summarized by the following charateristics:

1). A basic set of 16 symbols (0-9,A-F).
2). Each digit positioned left of the unit position is accompanied by a multiplier, and that multiplier increases by a factor of sixteen for every additional digit positio to the left.
(i.e. Multipliers of 1,16,256,4096, etc. are used).

Note that binary representations may be very easily converted to hexadecimal representations via the following steps:

1). Group the binary number into groups of four bits, starting with the unit's position, and proceeding right to left.
2). Write the hexadecimal symbol for
2). Substitute the appropriate hexadecimal symbol for each four-bit group from the original number.
3). Simply reverse this process to convert hexadecimal numbers into binary numbers, four bits at a time.

Hexadecimal numbers provide an extremely compact means of expressing multiple-bit binary numbers.

When reading a multiple digit number, it is not always immediately clear whether it is a binary, decimal, or hexadecimal representation. The symbol '1101' might be interpreted as a binary number (thirteen), a decimal number (one-thousand one-hundred and one), or as a hexadecimal number (four-thousand three-hundred and fifty-three = 1 X 4096 + 1 X 256 + 0 X 16 + 1 X 1). The number '1301' is clearly not a binary representation (it contains a '3'), but it could be interpreted as either a decimal or hexadecimal number.

In those instances when binary numbers are used, the writer usually calls attention to this fact, either by using a subscript '2', or by enclosing the notation 'binary' in the text of his discussion. Hexadecimal numbers are often distinguished from decimal numbers by preceding the hexadecimal number with a dollar sign, or by suffixing the hexadecimal number with a capital H. (i.e. $43C7, $7FFF, $4020, 1AD7H, F371H, 9564H). The dollar sign convention is the one adopted by most users of computers based on the 6502 microprocessor chip,including Ohio Scientific Instruments, and is the convention used in this book.

CHAPTER A PROBLEMS...

1). Convert the following binary numbers into decimal representations.

```
1111 1111
0111 1111
 111 1111
   1 0000
1000 1000
0100 0101
1111 1110
```

(ANSWERS: 255, 127, 127, 16,
136, 69, 254).

2). Convert the binary numbers given in problem number (1) into hexadecimal numbers.

(ANSWERS: $FF, $7F, $7F, $10, $88,
$45, $FE).

HEX-DEZ CONVERSION IN
Maschine Language

Here is a subroutine in machine language for conversion of hexadecimal to decimal numbers. The first listing shows you a printout from the Editor.The second listimg is the assembly print out. The hexadecimal number has to be in the accumulator (higher byte) and in the X-register (lower byte) when you jump into the subroutine.

EXAMPLE

Type in the the listing and assemble to the screen using the pseudoop OUT LNM, 3 The sourcecode now is in RAM starting at location C100 hex. Type <CTRL>-<P> to enter the monitor and write a little programm into RAM starting at location C000 hex.

```
C000 A9
C001 10
C002 A2
C003 1F
C004 20
C005 00
C006 C1
C007 00
```

Start this program in the monitor with G C000.

This program puts the hexnumber 101F into the accumulator and into the X-Register and jumps to our HEXDEZ subroutime. The result, the decimal number , is in the X-register and the Y-register. 101F hex = 4127 dec.

```
        OUT LNM,3
        ORG $C100
        STA $02
        STX $03
        LDA #$00
        STA $04
```

```
        STA $05
        STA $06
        SED
        LDY #$10
LOOP2   LDX #$03
        ASL $03
```

```
        ROL  $02                BNE  LOOP2
LOOP1   LDA  $03,X              CLD
        ADC  $03,X              LDA  $04
        STA  $03,X              LDX  $05
        DEX                     LDY  $06
        BNE  LOOP1              RTS
        DEY
```

```
                                ORG  $C100
C100: 8502                      STA  $02
C102: 8603                      STX  $03
C104: A900                      LDA  #$00
C106: 8504                      STA  $04
C108: 8505                      STA  $05
C10A: 8506                      STA  $06
C10C: F8                        SED
C10D: A010                      LDY  #$10
C10F: A203          LOOP2       LDX  #$03
C111: 0603                      ASL  $03
C113: 2602                      ROL  $02
C115: B503          LOOP1       LDA  $03,X
C117: 7503                      ADC  $03,X
C119: 9503                      STA  $03,X
C11B: CA                        DEX
C11C: D0F7                      BNE  LOOP1
C11E: 88                        DEY
C11F: D0EE                      BNE  LOOP2
C121: D8                        CLD
C122: A504                      LDA  $04
C124: A605                      LDX  $05
C126: A406                      LDY  $06
C128: 60                        RTS

PHYSICAL ENDADDRESS: $C129

*** NO WARNINGS

LOOP2               $C10F
LOOP1               $C115
```

Digital Concepts B

CHAPTER TWO: DIGITAL CONCEPTS

In this chapter we present an overview of digital logic concepts, and the kinds of electronic devices used to accomplish logical operations and data storage within your computer.

LOGIC IN PROGRAMMING AND COMPUTER HARDWARE

"...a computer is like a brain, a dumb brain, it doesn't do anything unless you program it first, and then it just follows your instructions one after another..."
-reaction of ten-year-old to computers.

People program computers to perform sequences of logical operations. A computer program consists of a sequence of instructions for the computer. Often we wish the computer to decide between alternative courses of action, based upon some information which is external to the program. For example, a computer might be programmed to control the signal lights at a railway crossing. Sensor switches would be placed some distance down the railway, such that they can detect an oncoming train. The computer program might read something like:

1. START HERE
2. CHECK TO SEE IF A TRAIN IS COMING
3. IF A TRAIN IS COMING, THEN SKIP AHEAD TO LINE 5 OF THE INSTRUCTIONS
4. GO BACK TO STEP 2 OF THE INSTRUCTIONS
5. CHECK TO SEE IF THE SAFETY BARRIER IS LOWERED
6. IF THE SAFETY BARRIER IS UP, THEN LOWER IT
7. CHECK TO SEE IF THE TRAIN IS STILL HERE
8. IF THE TRAIN IS STILL HERE, OR, IF ANOTHER TRAIN IS COMING, THEN GO BACK TO STEP 7 OF THE INSTRUCTIONS
9. RAISE THE SAFETY BARRIER
10. GO BACK TO STEP 2 OF THE INSTRUCTIONS

The above PROGRAM acts upon the DATA (or information) supplied by the train sensor switch. Another example would be the word-processor program upon which this manuscript is being typed. That program decides which letter to code into computer memory, based upon which one of the keyboard switches are pressed by the typist. Each of these examples also has means provided to output some result to the real world. In the case of the railway crossing, the computer has control of the position of the safety barrier, and uses that barrier to inform people of it's decision regarding the presence or absence of oncoming trains. The word processor program has control of a CRT (picture tube) upon which it displays the text input by the typist. It also outputs this text to computer memory, from whence the typist may command that it be recalled, corrected, and output to a printer. In summary, the computer executes a SEQUENCE of LOGICAL instructions upon some source of DATA input (switches, keyboards, memory, etc.), and produces some consistant OUTPUT as a result. In the remainder of this chapter, we will examine some of the fundamental electronic hardware used to accomplish logical operations within the computer.

LOGIC OPERATIONS AND LOGIC GATES...

Consider the following statements:

If (A is true) Then (Z is true)
If (A is false) Then (Z is False)

We shall assume A, Z, etc. are all either true or false, with nothing in-between being possible. With the above two statements, we have completely defined the condition of the OUTPUT Z, for all possible conditions of the input A. Suppose that we wish to model statements such as the above two, using electronic circuits. Let us define:

1. TRUE is to be represented by any voltage in the range from +2 volts to +5 volts. (i.e. HIGH).
2. FALSE is to be represented by any voltage in the range from 0 volts to +1/2 volt. (i.e. LOW).

Now consider a short piece of plain copper wire, the left end labeled "INPUT--A", and the right end labeled "OUTPUT--Z." This piece of wire will certainly model our original logical statements, as re-written:

1. If (A is HIGH) then (Z is HIGH). Certainly, if we connect a 'HIGH' voltage input to point A, then the wire will carry this same high voltage to the output at point Z.

2. If (A is LOW) then (Z is LOW). Once again, the input from A is carried directly to the output at Z.

There is almost always another way to accomplish any given task, and the above example is no exeception. There are electronic circuits other than our piece of wire which we could connect from A to Z, and obtain the same result. The need for these should become apparent as we continue.

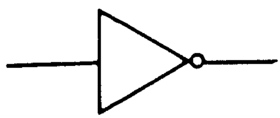
Consider the statements:

1. If (A is true), then (Z is false)
2. If (A is false), then (Z is true)
   (i.e. Z is always the opposite of A).

We cannot model this more complicated situation with only a piece of wire. We must use a readily available electronic circuit called a "NOT-gate", or "INVERTER." These devices are manufactured by many firms in many different forms. For the time being, it is perfectly sufficient to imagine a small box with two wires sticking out. One wire is our familiar input A, and the other wire is our output Z. If we put a high level on the input of an inverter, then we will get a low level at the output. A low level on the input yields a high level at the output. Forcing some signal INTO the output pin is forbidden, but the output of one inverter could certainly control the input to a second inverter. Clearly the output of inverter #2 would be exactly the same as the input to inverter #1. (This is a combination which could replace the copper wire in our earlier example).

There is a standard symbol used to represent an inverter. It is shown below in Figure 2.1.

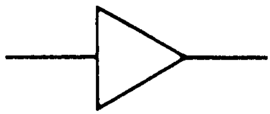
<<<<<<<<<<<<<<<FIGURE 2.1>>>>>>>>>>>>>>>>>>
<<<<<<<<<<<LOGIC INVERTER SYMBOL>>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>

There is a standard symbol used to represent a circuit which behaves as our copper wire did. This symbol represents a logic circuit whose single output duplicates it's single input. It is shown below in Figure 2.2. Note the absence of the "bubble" at the output, as compared with the inverter in figure 2.1. The bubble symbolizes the inversion process.

<<<<<<<<<<<<<<<<FIGURE 2.2>>>>>>>>>>>>>>>>>>>>
<<<<<<<<<<LOGIC BUFFER SYMBOL>>>>>>>>>>>>>>?



<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>

In certain situations we desire to connect the inputs of a number of different logic gates too the output of a single logic gate. If this number becomes too large the output of an ordinary gate might become overloaded. To prevent this we could connect the single output involved to the inputs of a pair of identical logic buffers. We could then distribute the large number of logic gate inputs between the two buffer outputs. Each buffer would have to drive only half the total number of inputs, and would not overload. More or larger buffers could be used if nessesary.

Consider the following statement:

If (A is true) OR (B is true), then (Z is true). (Otherwise Z is false).

This describes a single output (Z) controlled by two inputs (A and B). It is convenient to examine the possible outputs at Z, for all possible input combinations, through the use of a "truth table." A truth table for the current example is shown below in Figure 2.3. Note that a '1' is used to represent a 'true' condition, and that our electronic circuits would represent this with the 'high' voltage level.

TRUTH TABLE
Z = (A OR B)

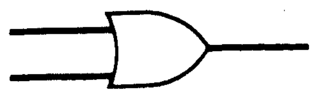
INPUT A	INPUT B	OUTPUT Z
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 2.3

In figure 2.3 we have described the operation of a "two-input OR-gate." This logical building block may be thought of as a box with THREE wires protruding. The three wires are inputs A, B, and output Z. Such circuits are readily available, and your microcomputer contains many, many of them. Note that we might also create a "Three-input OR-gate," which might have three inputs A, B, C, and output Z. In this case, output Z would become 'true' if any one OR more of the inputs became 'true.'

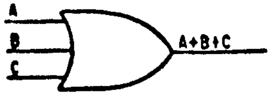
The logical symbol for a two-input OR-gate is shown in Figure 2.4, together with the symbol for a 3-input OR.

<<<<<<<<<<<<FIGURE 2.4A>>>>>>>>>>>>>>>>
<<<<<<<<<2-INPUT OR GATE SYMBOL>>>>>>>>>>



>>

<<<<<<<<<<<<FIGURE 2.4B>>>>>>>>>>>>>>>>>
<<<<<3-INPUT OR GATE SYMBOL>>>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>

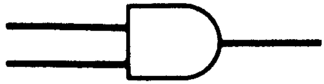
In the last example, we described how a logical output was based upon the truth of one OR another input. Frequently we wish to base some output upon the simultaneous truth of two inputs. For example:

If (a train is coming) AND (the safety barrier is up), then (lower the safety barrier).

If (A is true) AND (B is true) then (Z is true).

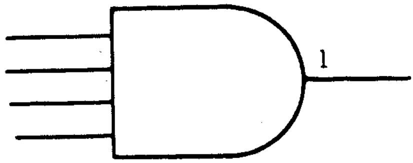
As in the case of the OR gate, we could just as easily base the truth of an output upon the simultaneous truth of three (or many more) inputs. Once again, the AND-gate is a readily available electronic circuit, supplied with two or more inputs as desired. The standard logic symbols for both two and three input AND-gates are shown below in Figure 2.5.

<<<<<<<<<<<<<<FIGURE 2.5A>>>>>>>>>>>>>>>>>
<<<<<<<<SYMBOL FOR 2-INPUT AND GATE>>>>>>>
<<<<<<<<<<<<<<<TITLES>>>>>>>>>>>>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<FIGURE 2.5B>>>>>>>>>>>>>>>>
<<<<<<<<SYMBOL FOR 4-INPUT AND GATE>>>>>>>
<<<<<<<<<<<<<<<TITLES>>>>>>>>>>>>>>>>>>>>



In summary, we have presented three principle types of logic gates. These are the AND, OR, and NOT gates. Each of these gates is readily available, usually packaged as several gates within a single plastic or ceramic cube, with input and output wires protruding in neat rows. In addition to the input and output wires, each package has at least two wires which must be connected to a source of power in order to operate it's internal circuitry. In the very common "Transistor-Transistor-Logic" (or "TTL") family which we describe, the inputs recognize voltages above 2 volts as a "true" or "1." The inputs recognize voltages below about 1/2 volt as "false" or "0." The voltages in the "no man's land" between 1/2 volt and 2 volts are illegal, and result in unpredictable performance of the gate circuit. Furthermore, voltages less than 0 (negative voltages), and voltages greater than 5 volts are excessive, and will damage the inputs. When a gate senses that it should send it's output high (or true), it will force the output to some voltage in the legal region between 2 and 5 volts. Otherwise the gate holds the output false, with a voltage between 0 and about 1/2 volt. Note that the output levels of a gate will always fall within the legal, recognizable voltage areas of an input. Thus it is possible to chain these simple gates together to perform complex logical operations built upon combinations of OR's, AND's, and NOT's acting upon some initial input(s).

Problem: Given four logic inputs A, B, C, and D, which are available on four wires within a computer, design a circuit which will set one logic output true if and only if ABCD=1010. (i.e. A=1, B=0, etc.).

Solution: Let's call our final output 'Z'. We wish to build a circuit such that:

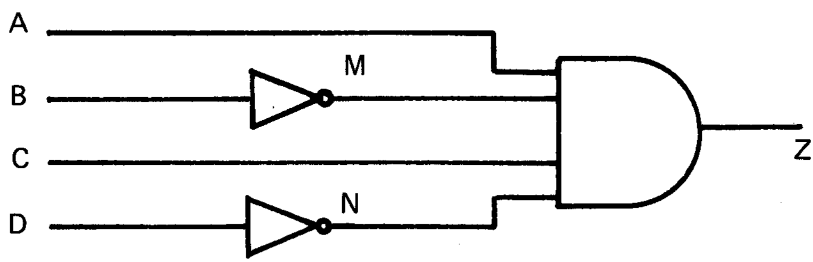
```
IF (A IS TRUE ), AND
   (B IS FALSE), AND
   (C IS TRUE ), AND
   (D IS FALSE), THEN (Z IS TRUE)
```

The B and D terms make it impossible to solve this problem with only a four-input AND-gate. However, if we put inverters on B and D then we might define two new signals:

M=NOT-B (i.e. M is the inverse of B).
N=NOT-D
We use these signals to write:

```
IF (A IS TRUE ), AND
   (M IS TRUE ), AND
   (C IS TRUE ), AND
   (N IS TRUE ), THEN (Z IS TRUE)
```

Our design uses two inverters to derive M and N from B and D respectively. M, N, A, and C are then combined with a four-input AND-gate. This combination is shown in Figure 2.6.



<<<<<<<<<<<<<FIGURE 2.6>>>>>>>>>>>>>>>>>
<<COMBINATIONAL LOGIC EXAMPLE SKETCH>>>>>

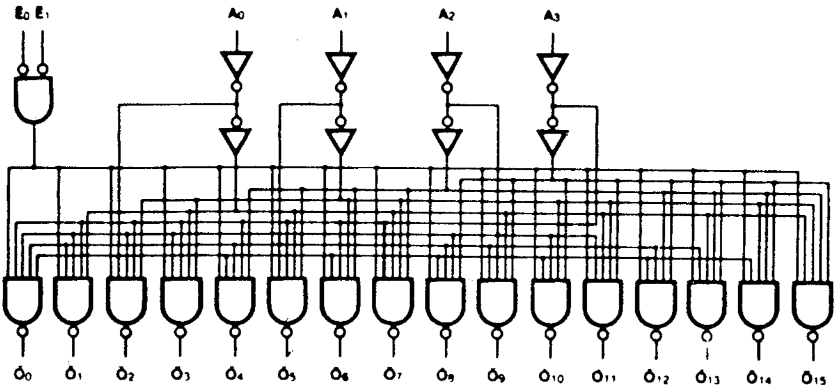
Figure 2.6 is an example of a decoder circuit. The circuit decodes a complex input, and generates a particular output for one possible state of the input. If we regard the four-bit input ABCD as a four bit binary number, then our decoder circuit decodes a count of ten. (Binary 1010). Recall that a four-bit binary number has sixteen possible combinations, zero thru fifteen. It is perfectly possible to design a decoder with four input lines, and sixteen outputs. Each output would represent exactly one of the sixteen possible combinations of the four-bit binary input. Since the input must, of course, be in one and only one of these possible states, it follows that one and only one of the output pins will be true at any one time. Figure 2.7 contains a truth table for such a circuit. Figure 2.8 contains a circuit diagram. The inputs are labeled ABCD, and the sixteen outputs are labeled Y0 thru Y15.

TRUTH TABLE: 4-INPUT 16-OUTPUT DECODER

INPUT:	OUTPUTS Y-															
ABCD	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0010	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0011	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0101	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0110	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0111	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1001	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1010	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1011	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1100	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1101	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

FIGURE 2.7

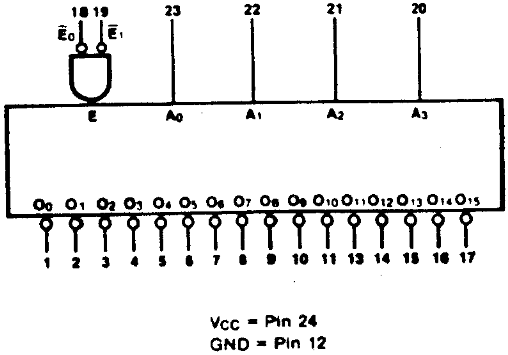
<<<<<<<<<<<<<<<FIGURE 2.8>>>>>>>>>>>>>>>
<<<CIRCUIT DIAGRAM. 4 TO 16 DECODER>>>>>>
<<<<<<<<<<POSITIVE LOGIC OUTPUTS>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>

Decoders such as the one shown in Figure 2.8 are available within a single package. Such a package measures about 2/3 inch wide, 2-1/2 inches long, and 1/8 inch high. There are 24 pins extending from the package. These connections consist of the 4 main inputs, 16 outputs, 2 power supply connections, and 2 "enable" inputs. Both of the enable inputs must be true, else NONE of the outputs will go true, irrespective of the state of the 4 main inputs. Smaller packages are available which function as 3-to-8 decoders and 2-to-4 decoders. The outputs of these devices are often inverted by comparison with the decoder example above. (i.e. The one and only selected output will be "low", and all others will be "high"). Figure 2.9 shows a sketch of a typical TTL integrated circuit containing a few logic gates.

<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>
<<<<<<<<<<FIGURE 2.9>>>>>>>>>>>>>>>>
<<<<<<TTL PACKAGE SKETCH>>>>>>>>>>>>
<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>



DECODERS AND MEMORY...

Decoders are important to the operation of the memory arrays in your computer. Memory consists of a large number of locations wherein the computer may store or recall either "1's", or "0's", as needed. In "8-bit" computers, these locations are grouped into sets of 8-bit BYTES as mentioned in chapter one. Each byte has a unique "ADDRESS", often compared to a post office box number.

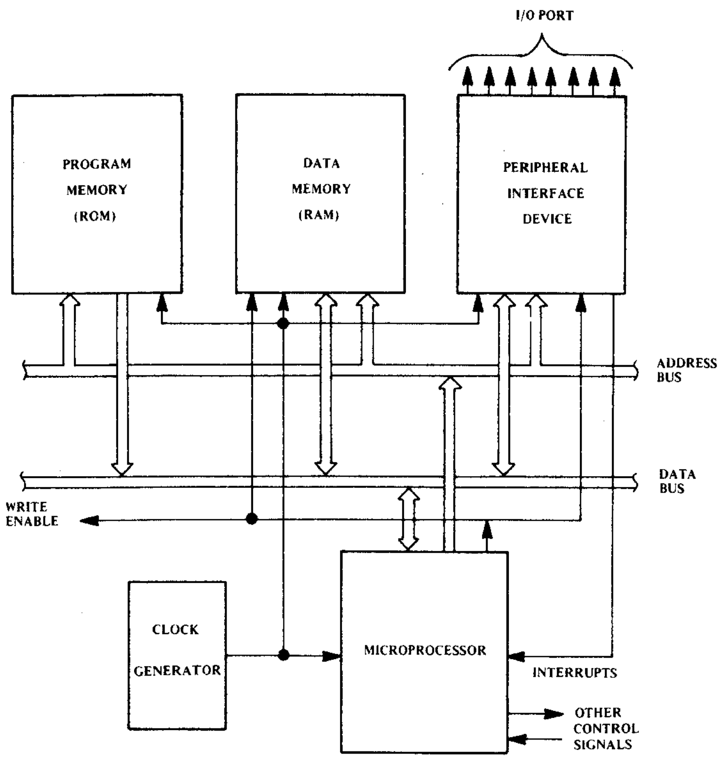
The computer's central processing unit (CPU) accesses a particular byte via the following process.

1. CPU sets a READ/WRITE control line to the proper state (high or low) to indicate a read memory or write to memory operation.

2. CPU outputs the unique address of the byte in question. The address is output in binary form onto a set of wires called "the ADDRESS BUS." Most small microcomputers use a sixteen wire address bus.

There are 65536 possible combinations of the sixteen address lines, meaning that the CPU is capable of distinguishing and controlling 65536 bytes of information. (Or 8 X 65536 = 524288 bits). a 16-to-65536 decoder. Most of this decoding is accomplished inside the memory integrated circuits, so it is not nessesary to imagine an integrated circuit with over 65000 pins protruding! In the case of a read operation, this decoder allows the 8 bits contained in a single location to be output to the CPU via a set of 8 wires called "the DATA BUS." In the case of a write operation, data passes FROM the CPU INTO the 8 bits of memory indicated by the address bus.

<<<<<FIGURE 2.10 CPU BUS SYSTEM>>>>>



<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>

NAND, NOR, AND EXCLUSIVE-OR GATES...

Consider the effect of adding an inverter to the output of an AND gate. If we call the two inputs A and B, and the final output Z, then we might describe the resulting logic function as:

If (A is true) AND (B is true),
Then (Z is FALSE).

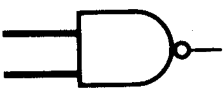
We call this logic function a "NAND GATE". We might write Z = A NAND B in this case. If we added yet another inverter, we would be back to a simple AND function. It turns out that it is easier to make NAND gates than AND gates. For this reason NAND gates are cheaper and more common.

As in the case of the NAND gate, an OR gate with an inverted output is called a NOR gate. Once again, this is a very common form of gate. NAND gates are drawn as AND gates with an inversion bubble at the output. NOR gates are drawn as OR gates with and inversion bubble at the output. (See Figures 2.11 and 2.12 for NAND and NOR standard logic symbols).

In the case of 2-input OR gates, the output was true if EITHER or BOTH inputs were true. The "exclusive-OR" gate excludes the case where BOTH inputs are true. Its performance could be stated:

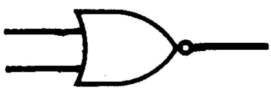
If ((A is true) OR (B is true)) AND
((A is false) OR (B is false)),
Then (Z IS TRUE).

The standard logic symbol for the exclusive-OR gate is shown in Figure 2.13.



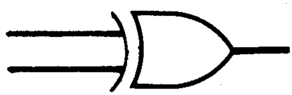
NAND

Fig. 2. 11



NOR

FIG. 2.12



EXCLUSIV OR

Fig. 2. 13

EN