Advanced Machine Language Book for the COMMODORE 64

A Data Becker Book from First Publishing Limited

FIRST PUBLISHING LTD

ADVANCED MACHINE LANGUAGE BOOK

FOR THE

COMMODORE 64

BY: Lothar Englisch

A DATA BECKER BOOK

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

ISBN 0-948015-055

Table of Contents

Introduction

The material in this book builds on the fundamentals of machine language programming as found in the book The Machine Language Book for the Commodore 64. In this book we will show you how to make use of many the Commodore 64's special features and capabilities using machine language.

The book is divided into three major sections. The first section concerns the internal representation of numbers on the Commodore 64 and describes in detail how the computer performs calculations and how its math routines can be used from machine language. In addition to the conversion of numbers between the various formats, the main emphasis of this section lies in writing arithmetic functions which can be used from BASIC with the help of the USR function.

The second section deals with a specialty of machine language: interrupts. After explaining some of the terms, interrupts are discussed in detail. Many sample programs illustrate the variety of uses for interrupt handling. At the close of this section, a machine language program demonstrates how BASIC subroutines can be controlled with interrupts.

The third and final section presents the concept of vectors in both the BASIC interpreter and kernal. The individual vectors are described and the procedure for adding your own commands is explained. The implementation of the REPEAT-UNTIL structure is used to demonstrate this.

SECTION 1 Numbers and arithmetic

1.1 Number representation on the Commodore 64

Your Commodore 64 uses two methods to represent numbers internally:

You are already familiar with the first type of representation, which is used for variables of type INTEGER. These variables can contain only whole numbers from -32768 to +32767 and can be represented in two bytes (16 bits). The upper-most of these 16 bits is used to represent the sign of the number.

Decimal	Binary	Hex
-32768	1 000 0000 0000 0000	80 00
-32767	1 000 0000 0000 0001	80 01
-32766	1 000 0000 0000 0010	80 02
-32765	1 000 0000 0000 0011	80 03
	...	
-2	1 111 1111 1111 1110	FF FE
-1	1 111 1111 1111 1111	FF FF
0	0 000 0000 0000 0000	00 00
1	0 000 0000 0000 0001	00 01
2	0 000 0000 0000 0010	00 02
	...	
32766	0 111 1111 1111 1110	7F FE
32767	0 111 1111 1111 1111	7F FF

This table illustrates how signed 16-bit numbers are represented. You can see that it is similar to the representation of signed 8-bit numbers which can contain the value0 -128 to +127 and are used for such things as relative addressing.

Integers are not suited for calculations which require fractional values or a large value range. Floating-point numbers are used for this purpose. You may be acquainted with this type of representation from such things as pocket calculators that use scientific notation. Let's take a closer look at floating-point.

Since we are familiar with the decimal system, we'll begin with it. If we want to represent a number, we first see how many times the base of this number system, 10, is contained within the number as a factor and divide the number into two parts. A example should clarify this:

```
 15 = 1.5 * 10^1
230 = 2.3 * 10^2
```

When we extend the exponential representation to include negative exponents, we can represent all of the numbers:

```
  5 = 5.0 * 10^0
0.7 = 7.0 * 10^-1
```

Since we know the base of the number system, a number is then represented by its mantissa, 7.0 in the last example, and the exponent, here -1. This is called normalized representation. The factor in front of the exponent is always a value between 1 and the base of the number system, or 10 in our case. The calculation rules of mathematics also apply for these numbers: For example, two normalized floating-point numbers can be multiplied together by multiplying the mantissas and adding the exponents. If the product of the mantissas is greater than 10, a factor of ten is added to the exponent of the product and the mantissa is adjusted so that it lies in the range 1-10. If we multiply the last two example numbers together, it looks like this:

5 * 10^0 times 7 * 10^-1

By multiplying the mantissas we get 35; the sum of the exponents is -1. The result is therefore 35 * 10^-1. This number must now be normalized since the mantissa is greater than 10. We get 3.5 * 10^0, or simply 3.5. The normalization can be thought of simply as moving the decimal point. In our example, we moved the decimal place one position to the left and compensated by increasing the exponent by one. When shifting the decimal place to the right, the exponent must be correspondingly decremented.

If we want to add our numbers, we know from mathematics that only numbers with the same exponent can be added. The exponents must therefore be made equal.

If we make both exponents equivalent to the larger, the procedure goes like this:

From 7.0 * 10^-1 we get 0.7 * 10^0. Now we need to add the mantissas:

5.0 + 0.7 = 5.7 * 10^0

Since the number is already normalized, we have as result 5.7 times 10^0 or simply 5.7.

If we want to put this process on a microprocessor, we must give a bit of thought to how this can best be realized. The processor can work only with binary numbers, so we first want to convert this procedure to binary numbers.

Let's select 2 as the base of our number system. Before we can implement floating-point numbers on the microprocessor, we should first decide what value range our numbers will have and to what degree of accuracy the numbers will be stored. It becomes clear very quickly that, using exponential representation, the exponent is the key to the value range, while the mantissa determines to how many places a number can be represented. We'll return later to the subjects of accuracy and representing decimal numbers in floating point format.

A floating-point number in binary representation has the following appearance:

```
     1.011101 * 2^10010
or   1.011101 * 2^18
```

which is

```
     1 * 2^18 = 262144
   + 0 * 2^17 =      0
   + 1 * 2^16 =  65536
   + 1 * 2^15 =  32768
   + 1 * 2^14 =  16384
   + 0 * 2^13 =      0
   + 1 * 2^12 =   4096
   ------------------
              = 380928
```

Fractional binary numbers can also be used. For example:

```
    1.011 * 2^0

  + 1 * 2^0  =       1
  + 0 * 2^-1 =       0
  + 1 * 2^-2 =       0.25
  + 1 * 2^-3 =       0.125
  ------------------------
                     1.375
```

If, however, we want to represent numbers which are smaller than one (the exponents of which are therefore less than zero) we must find a form for representing such exponents. We recall how we have stored negative numbers in the past. One possibility is two's complement. If we set aside one byte, 8 bits, for our exponents, we can represent powers of two from -128 to +127. To find out what decimal range of values this will allow us to represent, we need only form the corresponding power of two:

```
    2 ^  127  = 1.7 * 10^38
    2 ^ -128  = 3.9 * 10^-39
```

Thus if we reserve one byte for the exponent and work with powers of 2 from -128 to +127, we can work with numbers which in the decimal system have 38 places before the decimal point or which begin at the 39th place after the decimal point. These numbers then cover the value range which we are used to in normal calculations in BASIC.

The Commodore 64 does not use two's complement for its floating point numbers, but rather an offset. One adds the number 129 or $81 (hexadecimal) to every exponent and views the result as a sign-less positive number. This simplifies the manipulation of exponents in practice. The following table gives the assignment of the stored exponent to the actual power of two. We use the hexadecimal representation for the sake of simplicity.

Representation	Exponent	Value
$00	see text	0
$01	-128	3.9 * 10^-40
$02	-127	5.9 * 10^-39
$03	-126	1.2 * 10^-38
$7F	-2	0.25
$80	-1	0.5
$81	0	1
$82	1	2
$83	2	4
$FE	125	4.3 * 10^37
$FF	126	8.5 * 10^37

If the stored value for the exponent is zero, the number is by convention also zero.

Now that we have taken care of the exponents, we can give some thought to the mantissa.

Since the mantissa determines the number's accuracy, we must decide how accurately we want to store our numbers. The Commodore 64 uses 4 bytes for the mantissa. This allows us to represent 32 binary digits. What sort of accuracy does this correspond to for decimal numbers?

We compare the decimal values of two binary floating point numbers which differ only in the last place.

1.111 1111 1111 1111 1111 1111 1111 1111

and

1.111 1111 1111 1111 1111 1111 1111 1110

The two numbers are different in the last place, which has a place value of 2^-31. This is, in decimal, approximately

4.6566129 * 10^-10

or

0.46566129 * 10^-9

The two numbers have a value of somewhat less than 2; they differ by 5 in the 10th decimal place. We therefore conclude that we get a decimal accuracy of about 9 places from a mantissa of 4 bytes. This should suffice for most applications. The accuracy of 9 places is a relative accuracy, independent of the exponent. If we normalize the decimal numbers so that a digit between 1 and 9 is in front of the decimal point, we can still represent and differentiate between numbers which differ only in the ninth place after the decimal.

At this point we can use an exponent between -128 and +126 as well as mantissa of 4 bytes which allows a decimal accuracy of 9 places. What we still lack is the ability to account for the sign of the mantissa. If we are tricky, we can account for the sign of the mantissa without losing any accuracy.

Our mantissa will always displayed as normalized, meaning that a digit between one and one less than the base of the number system will always appear in front of the decimal point. Using the binary system, the only digit that can appear is the digit 1. We therefore assume this and do not save it, but use this bit for the sign. The usual convention applies, where a "0" indicates a positive number, while a "1" denotes a negative number.

Now we have all the information we need in order to convert decimal numbers into binary floating-point format. Let's try this with some numbers.

```
1 = 1 * 2^0

  = 1.000 0000 0000 0000 0000 0000 0000 0000 0000 * 2^0
```

We replace the first 1 in front of the point ("binary point") with the sign, account for the offset ($81) on the exponent and get

```
0000 0000 0000 0000 0000 0000 0000 0000  1000 0001
```

If we write the exponent first, as is done when storing floating-point numbers in the computer, we get the following picture:

```
1000 0001  0000 0000 0000 0000 0000 0000 0000 0000
```

To make it easier to read, we convert the number to hexadecimal:

```
81 00 00 00
```

This is the representation of the floating-point number 1.0. We will now try to represent the number 10.0. We divide it into powers of two as follows:

```
10 = 8 + 2
   = 2^1 + 2^3
   = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0
   = 1.01  * 2^3 binary
```

With the exponent and complete mantissa, we get the following result:

```
1000 0100  0010 0000 0000 0000 0000 0000 0000 0000
```

or

```
84 20 00 00 00
```

We will take a negative number, -5.5

```
-5.5 = - (4 + 1 + 0.5)
     = - (2^2 + 2^0 + 2^-1)
     = - (1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^-1)
     = - 1.011 * 2^2 binary

=> 1000 0011  1011 0000 0000 0000 0000 0000 0000 0000
```

or

```
83 B0 00 00 00
```

Negative numbers can be easily recognized because the first byte of the mantissa is always greater than or to equal $80.

With this knowledge we can easily calculate the decimal value of any floating-point number. If we designate the individual bytes as follows

```
EX M1 M2 M3 M4
83 B0 00 00 00
```

this formula gives us the value:

```
X = -SGN (M1 AND 128) * 2^(EX-129) *
     (1+ ((M1 AND 127)+(M2+(M3+M4/256)/256)/256)/128)
```

You can see clearly that the sign is derived from the most significant bit of the most significant byte of the mantissa (M1). The offset of 129 is taken into account on the power of two. The weighting of the individual bytes is taken into account in the mantissa; subsequent bytes have only one 256th the value of the preceding byte. Let us try out our formula with the above floating-point number.

```
X = -SGN (176 AND 128) * 2^(131-129) *
    (1+ ((176 AND 127)+(0+(0+0/256)/256)/256)/128)
```

We get the value -5.5 back again.

Up to now we have had no problems in converting decimal numbers to binary floating-point numbers. We will now try to convert the value 0.4.

We proceed systematically and subtract the largest power of two contained in the number.

```
   0.4                       Power of two
-  0.25                          -2
  =====
   0.15
-  0.125                         -3
  ======
   0.025
-  0.015625                      -6
  =========
   0.009375
-  0.0078125                     -7
  ==========
   0.0015625
-  0.0009765625                 -10
  =============
   0.0005859375
-  0.00048828125                -11
  ==============
   0.00009765625
-  0.00006103515625             -14
  =================
   0.00003662109275             etc.
```

We can continue this calculation as long as we want to; the remainder of a division will never be zero. We receive the periodic value

1. 1001 1001 1001 1001 1001 1001 1001 1001 1001 ... * 2^-2

We cannot represent the number 0.4 exactly as a binary floating-point number. We must stop the succession of digits at the 31st place behind the binary point and then get

1. 1001 1001 1001 1001 1001 1001 1001 100 * 2^-2

In order to increase the accuracy somewhat we will not truncate the digits, but rather round the number up or down. Binary values are rounded up when the last (here 32nd) digit is a 1; the number remains the same for a 0. In our case we must round up.

1. 1001 1001 1001 1001 1001 1001 1001 101 * 2^-2

If we now take the exponent and sign into account, we get the following:

0111 1111 0100 1100 11001 1100 1100 1100 1100 1101

or in hexadecimal

7F 4C CC CC CD

The fact that we cannot exactly represent all decimal numbers with binary floating-point numbers is not just a defect in base 2, it is a typical phenomenon when converting from one number system to another. Try to represent the fraction 1/3 in the decimal system--it cannot be done exactly. The succession of digits

0.33333 33333 33333 .....

must be truncated somewhere. This is not necessary in a number system with base 3, however. We get simply

0.1

which we interpret as 1 * 3^-1 or exactly one third.

Now that we have heard about the fundamentals of floating-point numbers, we want use them. Since a large part of the built-in BASIC interpreter is concerned with conversion between various number formats as well as floating-point arithmetic, it makes sense to learn how to use these routines.

The BASIC interpreter has two floating-point accumulators, usually shortened to FAC, in which floating-point numbers are stored. FAC #1 is used for all floating-point operations. If an operation such as addition requires two operands, the second is placed in FAC #2. The result is always returned in FAC #1. Floating-point accumulator #1 is often designated only as FAC and FAC #2 is then called ARG (argument). The numbers are not stored in the shortened 5-byte form in these floating-point accumulators. Instead, an additional byte is used for the sign. The bit in front of the binary point which is otherwise replaced by the sign is then reconstructed. Furthermore, a rounding byte is used in order to facilitate rounding with various operators. The floating-point accumulators use the following memory locations in the zero-page:

	FAC	ARG
Exponent	$61	$69
Mantissa 1	$62	$6A
Mantissa 2	$63	$6B
Mantissa 3	$64	$6C
Mantissa 4	$65	$6D
Sign	$66	$6E
Rounding byte	$70	
Sign-comparison byte	$6F	

The sign-comparison byte is required for operations with two operands and is $00 for equivalent signs and $FF for different signs.

The BASIC interpreter has numerous routines which manipulate floating-point numbers. We will begin with the routine which reads a decimal number and converts it to a floating-point number. This routine is used for every number input. First we will take a brief look at a routine called "CHRGET" which reads a character from an input line or from the BASIC program. The routine is located in the zero-page and has the task of reading a character and executing various comparisons. The routine has a second entry point by the name of "CHRGOT" which allows the character last read to be gotten again.

```
CHRGET  INC TXTPTR
        BNE CHRGOT
        INC TXTPTR+1
CHRGOT  LDA TEXT
        CMP #":"
        BCS EXIT
        CMP #" "
```

```
           BEQ CHRGET
           SEC
           SBC #$30
           SEC
           SBC #$D0
EXIT       RTS
```

The power of this routine and the reason it must be located in RAM is that it is self-modifying. The address of TXTPTR, the pointer to the current position from which the character will be fetched, is found in the routine itself. This will be immediately clear if we look at the machine code for this routine.

```
0073   E6 7A       INC $7A
0075   D0 02       BNE $0079
0077   E6 7B       INC $7B
0079   AD 02 02    LDA $0202
007C   C9 3A       CMP #$3A
007E   B0 10       BCS $008A
0080   C9 20       CMP #$20
0082   F0 EF       BEQ $0073
0084   38          SEC
0085   E9 30       SBC #$30
0087   38          SEC
0088   E9 D0       SBC #$D0
008A   60          RTS
```

When we call the routine CHRGET, the operand of the LDA instruction at CHRGOT is incremented by one and then the contents of this memory location are placed into the accumulator. Several comparison instructions follow.

If the ASCII code of the character in the accumulator is greater than or equal to that for a colon, then control passes directly to the RTS instruction. In this case, the carry flag is set. If the character was a colon, the zero flag is also set. Since the colon denotes the end of a statement, this can easily be tested for with the zero flag. If the character's ASCII code is less than that for a colon, the code is next compared to a space (ASCII 32). If the test is positive, control is returned to CHRGET--the next character is fetched. Spaces are thereby ignored by the interpreter. The next two subtractions do not change the value in the accumulator, but they do have an effect on the carry flag. The carry flag is cleared if the character is an ASCII digit between "0" and "9", corresponding to $30 and $39.

Let's review the points of this routine: the CHRGET routine increments the text pointer TXTPTR and returns the current character in the accumulator. If the character is a colon or a zero byte, which indicates the end of a statement or the end of the line, respectively, the zero flag is set. If the character was a digit, the carry flag is cleared.

We now come to our conversion routine. Before we can call this routine, the accumulator must contain the first character of the number and the flags must be set according to the CHRGET routine. The text pointer TXTPTR must naturally point to our number. The following short program reads a number and converts it into floating point format.

```
100:    033C                         .OPT P,00
105:    033C                         *=   828
110:    007A          TXTPTR         =    $7A
120:    0079          CHRGOT         =    $79
```

```
130:     BCF3              ASCFLOAT =    $BCF3
140:     033C A9 4B                 LDA #<NUMBER
150:     033E A0 03                 LDY #>NUMBER
160:     0340 85 7A                 STA TXTPTR
170:     0342 84 7B                 STY TXTPTR+1
180:     0344 20 79 00              JSR CHRGOT
190:     0347 20 F3 BC              JSR ASCFLOAT
200:     034A 00                    BRK
210:     034B 31 2E 32   NUMBER  .ASC "1.2345"
220:     0351 00                    .BYT 0
```

If we assemble this routine and execute it from the monitor with

```
G 033C
```

the number 1.2345 is converted to floating-point format and placed in FAC #1, which we can see with

```
M 0061 0066
```

We get the following values:

```
>: 0061 81 9E 04 18 93 00
```

We try our 0.4 again. We must place the digits at address $034B and terminate them with a zero byte:

```
M 034B 034B
```

```
>: 034B 30 2E 34 00
```

We get the result

```
>: 0061 7F CC CC CC CC 00
```

The sign is saved separately as the sixth byte and is zero for positive numbers. We can also work with numbers represented as powers of ten with this number conversion, such as -1.4E-7 or 1E12. We will take negative number as our next example, -1E8. Now get

```
>: 0061 9B BE BC 20 00 FF
```

This time the negative sign is denoted by $FF.

Let us return briefly to the result of the value 0.4. We got a value that was one less in the last place than was the case for the manual conversion. No automatic rounding is performed by our routine; the rounding byte is used only to indicate if an overflow is present in the next places. Enter 0.4 again and note the value of the rounding byte at location $70. We get $80. This means that the last place of the result must be incremented by one. There is a routine at our disposal which does this for us. If we add this to our program, the converted value is automatically rounded.

```
100:    033C                      .OPT P,00
105:    033C                      *=   828
110:    007A          TXTPTR   =   $7A
120:    0079          CHRGOT   =   $79
130:    BCF3          ASCFLOAT =   $BCF3
140:    BC1B          ROUND    =   $BC1B
150:    033C A9 4B                LDA #<NUMBER
160:    033E A0 03                LDY #>NUMBER
170:    0340 85 7A                STA TXTPTR
```

```
0342 84 7B                     STY TXTPTR+1
0344 20 79 00                  JSR CHRGOT
0347 20 F3 BC                  JSR ASCFLOAT
034A 20 1B BC                  JSR ROUND
034D 00                        BRK
034E 31 2E 32     NUMBER       .ASC "1.2345"
0351 00                        .BYT 0
```

If we take a look at the FAC, we have the desired result.

```
>: 0061 7F CC CC CC CD 00
```

The rounding byte is naturally cleared by rounding, something of which you can easily convince yourself.

Now that we have converted the digit string to an internal floating-point number, let's reverse the procedure by converting a floating-point number back into a string of decimal digits. This task is performed by the routine FLOATASC, located at address $BDDD. Calling this routine converts a number to a string which is placed at address $0100. Let us try this by writing the following values into the FAC:

```
>: 0061 90 8F 00 00 00 80
```

We take a look at the result after calling the routine:

```
>M 0100 0107
>: 0100 2D 33 36 36 30 38 00   -36608
```

The above value in the FAC therefore represents the decimal number -36608. After calling this routine, the accumulator and Y register contain the address at which the string was placed ($100), here A=0 and Y=1 (low byte, high byte). Now we can output the string on the screen. Another routine is already built into the BASIC interpreter: STROUT, with address $AB1E.

```
JSR  FLOATASC ;convert FAC to ASCII string at $100
LDA  #<$100   ;least significant address of string
LDY  #>$100   ;most significant address of string
JSR  STROUT   ;print string pointed to by A,Y
```

Before we start performing calculations with our floating-point numbers, we first want to become acquainted with the various BASIC interpreter routines which perform conversions from various whole-number formats to floating-point format. This is particularly important for our machine language programs because all of the arithmetic operations within the BASIC interpreter are carried out in floating point, but input and output for these routines often require or expect numbers in INTEGER format.

1.2 Conversion to floating-point format

1.2.1 Signed one-byte values

The following routine converts a signed one-byte value into floating-point format. The result will therefore be a number between -128 and +127. The byte value is passed in the accumulator.

```
LDA #BYTE
JSR $BC3C
```

A value of $80 will be converted to -128, $FF to -1, $7F to 127, and so on.

1.2.2 Unsigned one-byte values

If the sign is not to be taken into account (the byte is to be treated as unsigned, having a value 0-255), the following conversion routine must be used:

```
LDY #BYTE
JSR $B3A2
```

This routine converts $00 to zero, $80 to 128, and $FF to 255.

1.2.3 Signed two-byte values

A two-byte value with sign can be converted with the following routine:

```
LDY #LOW
LDA #HIGH
JSR $B395
```

The least-significant byte must be placed in the Y register while the accumulator contains the most-significant byte.

The following examples demonstrate the conversion:

A	Y	Floating-point value
00	00	0
00	01	1
00	FF	255
01	00	256
7F	FF	32767
80	00	-32768
FF	FF	-1

1.2.4 Unsigned two-byte values

If the sign of a two-byte value is to be ignored, the following routine is used:

```
LDY #LOW
LDA #HIGH
STY $63
STA $62
LDX #$90
SEC
JSR $BC49
```

This conversion assumes that the number is unsigned and returns values from 0-65535.

A	Y	Floating-point value
00	00	0
00	01	1
00	FF	255
01	00	256
7F	FF	32767
80	00	32768
FF	FF	65535

1.2.5 Signed three-byte values

Although three-byte values are rarely used in practice, the routines for converting such data into floating-point format should be mentioned.

```
LDA #LOW
LDX #MID
LDY #HIGH
STY $62
STX $63
STA $64
LDA $62
EOR #$FF
ASL A
LDA #$0
STA $65
LDX #$98
```

```
JSR $BC4F
```

The conversion table looks like this:

```
Y  X  A    Floating-point value
00 00 00   0
00 00 FF   255
00 FF FF   65535
7F FF FF   8388607
80 00 00   -8388608
FF FF FF   -1
```

We can cover a value range from -8,388,608 to 8388607 with 3-byte (24-bit) values.

1.2.6 Unsigned three-byte values

If the sign is not to be used, the following routine can be used.

```
LDA #LOW
LDX #MID
LDY #HIGH
JSR $AF87
JSR $AF7E
```

Here we can represent values between 0 and 2^24-1 = 16,777,215.

Y X A	Floating-point format
00 00 00	0
00 00 FF	255
00 FF FF	65535
7F FF FF	8388607
80 00 00	8388608
FF FF FF	16277215

1.2.7 Signed 4-byte values

For the sake of completeness, the conversion of 32-bit integer values is also presented here. The routines look similar. Because 4 bytes must be passed, the routine expects that these values will be stored in the FAC from address $62 (MSB) to $65 (LSB).

```
LDA $62
EOR #$FF
ASL A
LDA #0
LDX #$A0
JSR $BC4F
```

We get the following conversion table:

$62 63 64 65	Floating point value	
00 00 00 00	0	
00 00 00 FF	255	
00 00 FF FF	65535	
00 FF FF FF	16777215	
7F FF FF FF	2147483647	(2.14748365E+09)
80 00 00 00	-2147483648	(-2.14748365E+09)

```
FF FF FF FF   -1
```

1.2.8 Unsigned 4-byte values

This conversion routine concludes the presentation. Here too the values must be placed in the FAC.

```
SEC
LDA #0
LDX #$A0
JSR $BC4F
```

The value range from 0 to 2^32-1 = 4,294,967,295 can be used.

```
$62 63 64 65   Floating-point value
 00 00 00 00   0
 00 00 00 FF   255
 00 00 FF FF   65535
 00 FF FF FF   16777215
 7F FF FF FF   2147483647    (2.14748365E+09)
 80 00 00 00   2147483648    (2.14748365E+09)
 FF FF FF FF   4294967295    (4.2949673E+09)
```

The routines described here are useful if you want to use one to four-byte values from your own machine language routines as arguments for the floating-point routines in the BASIC interpreter. The reverse procedure--converting from floating-point values to integer numbers--will now be discussed.

1.3 Conversion to integer format

Only one routine is required for the conversion from floating-point to integer format. The result of this conversion is a signed 4-byte number. If the number to be converted is in the FAC, the conversion is executed with

```
JSR $BC9B
```

Because only numbers which are smaller than 2^31 can be converted to integer values without error, the exponent of the number should be checked to see that it is smaller than $A0, The result of the conversion is stored at $62 (most significant byte, including sign) to $65 (least-significant byte). Let us try an example.

The FAC contains the floating-point value 10:

```
        EX M1 M2 M3 M4 SGN
>: 0061 84 A0 00 00 00 00
```

After the JSR $BC9B we get

```
>: 0061 84 00 00 00 0A 00
```

If the FAC does not contain a whole number, the fractional portion will be truncated as with the INT function. For example, if the FAC contains 321.25:

```
        EX M1 M2 M3 M4 SGN
>: 0061 89 A0 A0 00 00 00
```

We get the result

```
>: 0061 89 00 00 01 41 00
```

or $41 + $100 = 65 + 256 = 321. With negative fractional numbers, the result will be next-smallest whole number, so that -0.5 becomes -1.

```
        EX M1 M2 M3 M4 SGN
>: 0061 80 80 00 00 00 FF
```

We get the result

```
>:0061 80 FF FF FF FF FF
```

or -1.

We will later become acquainted with BASIC interpreter routines which perform range checks before the conversion to integer format, on the ranges 0 to 255 or -32768 to 32767, for example.

1.4 BASIC math routines

Now that we have covered input, output, and conversion of numbers, it is time that we execute the first calculations.

The interpreter has five basic arithmetic operations, each having two operands, which are addition, subtraction, multiplication, division, and exponentiation. If we want to use these functions, the first operand must be in the FAC while the second is expected in ARG. After calling the appropriate routine, the result is left in the FAC. These are the addresses of the routines:

ADDITION	FAC := ARG + FAC	$B86A
SUBTRACTION	FAC := ARG - FAC	$B853
MULTIPLICATION	FAC := ARG * FAC	$BA2B
DIVISION	FAC := ARG / FAC	$BB12
EXPONENTIATION	FAC := ARG ^ FAC	$BF7B

Before calling these routines, the accumulator must contain the exponent of the number in the FAC ($61). If this exponent is zero, the number in the FAC is by convention also zero and special cases can be handled (ARG + 0 = ARG; ARG * 0 = 0; ARG / 0 results in "DIVISION BY ZERO"; ARG ^ 0 yields 1). Now let's multiply two values, such as 7*13 = 91.

```
 7 =  83 E0 00 00 00 00
13 =  84 D0 00 00 00 00
```

We place the values in the floating-point accumulators, load the accumulator with the exponent of the FAC, and call the routine.

```
>: 0061 83 E0 00 00 00 00
>: 0069 84 D0 00 00 00 00

>, 1000    A5 61        LDA $61
>, 1002    20 2B BA     JSR $BA2B
>, 1005    00           BRK

>G 1000

B*
    PC  IRQ  SR AC XR YR SP   NV-BDIZC
>; 1006 EA31 A0 87 B6 00 F8   10100000

>: 0061 87 B6 00 00 00 00 00
```

Now we can convert the result into a decimal number.

```
  1.0110110 * 2^6 = 1011011
  = 64 + 16 + 8 + 2 + 1 = 91
```

Next we will try exponentiation. 3^7 should equal 2187.

```
   3 =  82 C0 00 00 00 00
   7 =  8E E0 00 00 00 00
```

We can pass the values and call the exponentiation routine.

```
>: 0061 83 E0 00 00 00 00
>: 0069 82 C0 00 00 00 00
```

```
>, 1000    A5 61       LDA $61
>, 1002    20 7B BF    JSR $BF7B
>, 1005    00          BRK

>G 1000

B*
    PC  IRQ  SR AC XR YR SP   NV-BDIZC
>; 1006 EA31 22 00 61 00 F8   10100000

>: 0061 8C 88 B0 00 02 00 00
```

We get

```
1.000 1000 1011 0000 0000 0000 0000 0010 * 2^11 =
1000 1000 1011. 0000 0000 0000 0000 0010

= 2^11 + 2^7 + 2^3 + 2^1 + 2^0 + 2^-19
= 2048 + 128 + 8   + 2   + 1   + 1.9*10^-6
= 2187.0000019
```

You see that the result is not exact--there is a deviation in the last two places. Since only 9 significant digits are displayed when converting from binary to decimal, we receive from the following instruction

```
PRINT 3^7
```

the result 2187, although the calculation

```
PRINT 3^7 - 2187
```

results in

   1.90734863E-06

which reveals the discrepancy. If we analyze the routine for exponentiation in greater detail, we see that the following algorithm is used:

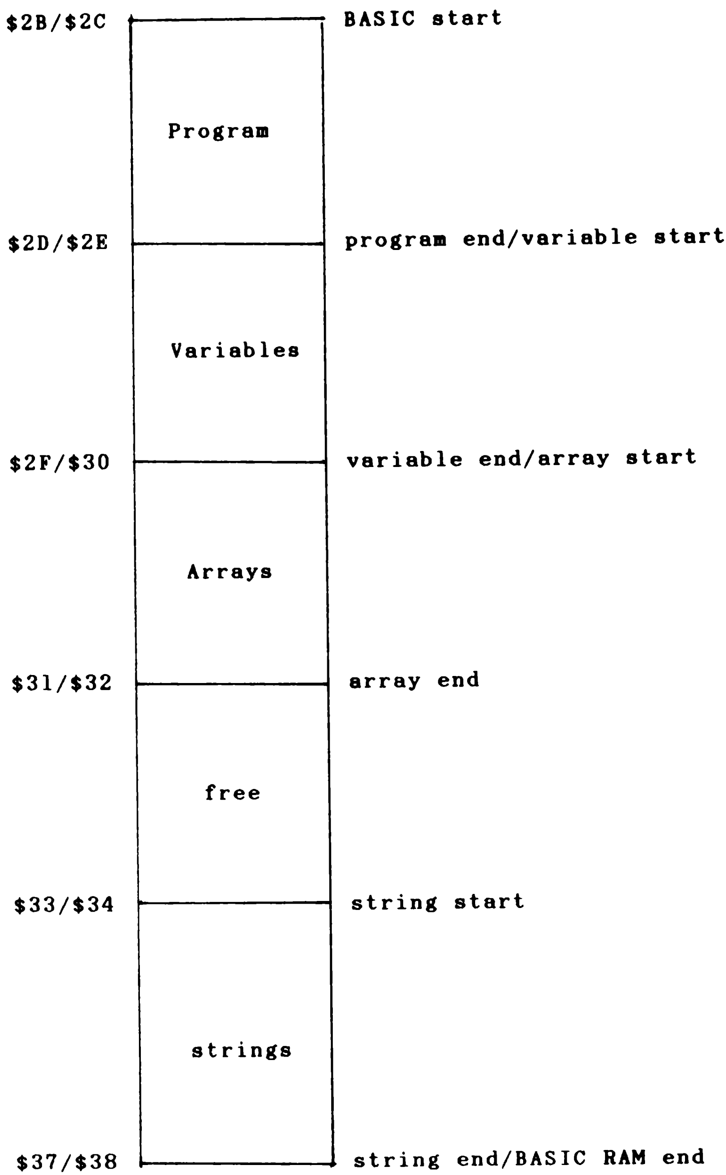
```
A ^ B  =>  EXP( B * LOG( A ))
```

Because the BASIC interpreter can only calculate approximations for the EXP and LOG functions--as we will see later--it is no wonder that exponentiation returns a discrepancy. Since two other functions must be calculated for the exponentiation function, this routine is also one of the slowest arithmetic routines. It requires more than 50 milliseconds on average. Therefore it is advisable to perform exponentiations with simple, integer exponents with repeated multiplication, both for the sake of speed and accuracy.

```
3 ^ 2 should be calculated as 3 * 3
```

The multiplication here is more than 20 times faster. A summary of execution times will be presented later.

So that we can make practical use of our knowledge, we will first take a look at the way the BASIC interpreter manages variables. A number of pointers exist in the zero-page for managing variables. These pointers determine the areas for the BASIC program, normal variables, indexed variables, and strings. The variable pointers are arranged as follows.


$2B/$2C
BASIC start
Program
$2D/$2E
program end/variable start
Variables
$2F/$30
variable end/array start
Arrays
$31/$32
array end
free
$33/$34
string start
strings
$37/$38
string end/BASIC RAM end

After turning on the computer, the start of the BASIC text area is set to $0801 = 2049 and the end to $A000 = 40960. When you enter a program line

```
10 A=1
```

the following is allocated:

At the BASIC start $0801 is

- address of the next BASIC line
- line number (LSB, MSB)
- tokenized program line
- $00 signifying the end of this line

From the monitor this looks like:

```
>M 0800 080F
>: 0800 00 09 08 0A 00 41 B2 31
>: 0808 00 00 00
```

The program pointers have the following values:

```
>M 002B 0037
>: 002B 01 08 0B 08 0B 08 0B 08
>: 0033 00 A0 00 00 00 A0
```

We will try to interpret these contents. At address ($2B/$2C) = $0801 is the address of the next program line in the format lo/hi, or $0809. Then follows the line number, also in lo/hi format = $000A = 10. Next is the program text $41 = "A", $B2 is the interpreter code for "=", while $31 is "1" in ASCII code. A zero byte serves to mark the end of the line. The next program line follows the same scheme. But because we entered only one program line, we find $0000 as the address of the next program line. By convention this denotes the end of the program. The address following, $080B, is stored in ($2D/$2E) and denotes both the end of the program and the start of the normal variables. Since we have not defined any variables, the pointers for the variable end and array end have the same value. If we execute the program with RUN, the variable A is allocated.

```
>M 0800 0810
>: 0800 00 09 08 0A 00 41 B2 31
>: 0808 00 00 00 41 00 81 00 00
>: 0810 00 00

>M 002B 0037
>: 002B 01 08 0B 08 12 08 12 08
>: 0033 00 A0 00 00 00 A0
```

Now the start-of-variables pointer ($2D/$2E) points to $080B and the end-of-variables pointer ($2F/$30) to $0812. Thus the variable table is $0812 - $080B = $0007 = 7 bytes long and has the following contents:

```
>: 080B 41 00 81 00 00 00 00
```

Variable entries are generally 7 bytes long. The first two bytes represent the name of the variable, in this case $41 $00 = A. Variable names which are only one character long contain a zero as the second character. Following the name is the floating point representation of the value in the abbreviated 5-byte form in which the sign is the first bit of the mantissa. The value 81 00 00 00 00 has a decimal equivalent of 1.

We will now take a look at what happens when we use integer variables. We change our program line to

```
10 A%=1
```

```
>M 002B 0037
>: 002B 01 08 0C 08 13 08 13 08
>: 0033 00 A0 00 00 00 A0

>M 0800 0810
>: 0800 00 0A 08 0A 00 41 25 B2
>: 0808 31 00 00 00 C1 80 00 01
>: 0810 00 00 00
```

The program has become one byte longer because of the percent sign. The variable entry is still 7 bytes long. Do recognize the name A or A% in the table? If you compare the bit pattern $C1 $80 with $41 $00, you see that the most significant bit (bit 7) of both bytes is set. This is how integer variables are denoted. The next two bytes contain the 16-bit integer value $0001, in which the most-significant byte comes first. The next three bytes are unused for integer variables. Therefore when you work with normal integer variables, there is no space savings. Using integer variables does not increase the speed either--in fact, just the opposite since all of the math operations are performed in floating-point arithmetic and that additional conversion are necessary.

Let us move on to the string variables. Enter the following program line:

```
10 A$="STRING"
```

RUN the program and take a look at the result with a monitor.

```
>M 002B 0037
>: 002B 01 08 13 08 1A 08 1A 08
>: 0033 00 A0 00 00 00 A0

>M 0800 0810
>: 0800 00 11 08 0A 00 41 24 B2
>: 0808 22 53 54 52 49 4E 47 22
>: 0810 00 00 00 41 80 06 09 08
>: 0818 00 00
```

If you take a look at the pointer for the string area, you see that nothing has been altered. The variable table begins at $0813 and looks like this:

```
>: 0813 41 80 06 09 08 00 00
```

The first two bytes again represent the name of the variable. You have probably already noticed that the most significant bit of the second byte of the variable name is set in order to denote a string variable--$41 $00 becomes $41 $80. The next three values can be interpreted as follows: The first value, $06, gives the length of the string: 6 characters. The next two bytes point to the address at which the string can be found: $0809. Thus they point to an area within the program, directly behind the first quotation mark. This is also the reason that the string area is still empty. The situation changes if we modify the string, however, as we see in the next example:

```
10 A$="STRING"
20 A$=LEFT$(A$,3)
```

```
>M 002B 0037
>: 002B 01 08 22 08 29 08 29 08
>: 0033 FD 9F 00 A0 00 A0

>M 0800 0810
>: 0800 00 11 08 0A 00 41 24 B2
>: 0808 22 53 54 52 49 4E 47 22
>: 0810 00 20 08 14 00 41 24 B2
>: 0818 C8 28 41 24 2C 33 29 00
>: 0820 00 00 41 80 03 FD 9F 00
>: 0828 00
```

The variable table begins at $0822.

```
>: 0822 41 80 03 FD 9F 00 00
```

Following the variable name is the length (3 this time) and the address of the string, $9FFD, which is also the lower boundary of the string storage. If we look there, we find our new string "STR".

```
>: 9FFD 53 54 52
```

How are variable arrays organized? Erase the current program (with NEW) and enter the following:

```
10 DIM A(500)
RUN
```

We get the following storage allocation:

```
>M 002B 0037
>: 002B 01 08 10 08 10 08 E0 11
>: 0033 00 A0 00 00 00 A0
```

Since no non-array variables are defined, the starting and ending pointers have the same value, $0810. This is also the start of the array area. The array area extends to $11E0, and is therefore $11E0 - $0810 = $09D0 = 2512 bytes long. The start looks like this:

```
>M 0810 0820
>: 0810 41 00 D0 09 01 01 F5 00
>: 0818 00 00 00 00 00 00 00 00
>: 0820 00 00 00 00 00 00 00 00
```

The name of the array is encoded in the first two bytes. The following two bytes contain the length of the memory occupied by the array, $09D0, which we calculated above. The next "01" indicates that the array has one dimension. Next is the number of array elements, $01F5 = 501. There are five hundred and one because an element exists with the index 0, A(0). Finally, the values of the array elements are stored starting with the zero element. If we enter A(0)=10:A(1)=11 in the direct mode, the representation appears as follows:

```
>M 0810 0820
>: 0810 41 00 D0 09 01 01 F5 84
>: 0818 20 00 00 00 84 30 00 00
>: 0820 00 00 00 00 00 00 00 00
```

84 20 00 00 00 => 10; 84 30 00 00 00 => 11

Now let's see how multi-dimensional arrays are stored. Enter

```
DIM B(1,2,3)
```

in the direct mode. The array table starts at $0803 and looks like this:

```
>M 002B 0037
>: 002B 01 08 03 08 03 08 86 08
>: 0033 00 A0 00 00 00 A0

>M 0803 0813
>: 0803 42 00 83 00 03 00 04 00
>: 080B 03 00 02 00 00 00 00 00
>: 0813 00 00 00 00 00 00 00 00
```

We recognize the name "B" ($42). The length of the array table is $0083 = 131 bytes this time. Then comes a 3 which indicates that the array is three-dimensional. Next are the index boundaries, beginning with the last index $0004, then $0003, and $0002 corresponding to 3, 2, and 1. How are these values allocated? This is the order in which the individual array elements are stored:

```
B(0,0,0)
B(1,0,0)
B(0,1,0)
B(1,1,0)
B(0,2,0)
B(1,2,0)
```

B(0,0,1)
B(1,0,1)
B(0,1,1)
B(1,1,1)
B(0,2,1)
B(1,2,1)
B(0,0,2)
B(1,0,2)
B(0,1,2)
B(1,1,2)
B(0,2,2)
B(1,2,2)
B(0,0,3)
B(1,0,3)
B(0,1,3)
B(1,1,3)
B(0,2,3)
B(1,2,3)

If we use arrays with integer variables, only 2 bytes are reserved for each array element, resulting in a space savings compared to floating point arrays. Only three bytes per element are used for string arrays. The first byte represents the length of the individual string element and the next two bytes give the actual memory address of the string. No space is used for the strings themselves until they are actually assigned values. Using this information we can state the space requirements of any array:

$$M = 5 + 2*N + T * PROD(N_i+1)$$

M is the required memory space of the entire array, N is the number of dimensions, T is the specified space requirement per element (2 for integer, 5 for real, and 3 for string) and PROD(N_1+1) the product of the index boundaries + 1.

The following examples should clarify the formula:

The constant 5 is based on 2 bytes for the name, 2 bytes for the length, and one byte for the number of dimensions. Two bytes are required for each dimension for the index boundaries. The space for the elements themselves is contained in the last term. Let's try our formula for the first array A(500).

P = 5 + 2*1 + 5*(501)
P = 2512 bytes

Our three dimensional array B(1,2,3) requires the following space in memory:

P = 5 + 2*3 + 5*(2*3*4)
P = 131 bytes

The array A%(10,10,10) requires the following memory space:

P = 5 + 2*3 + 2*(11*11*11)
P = 2673 bytes

A string array A$(100,100) would hardly fit into memory.

P = 5 + 2*2 + 3*(101*101)
P = 30603 bytes

The array table alone requires 30K bytes; there are only 8K bytes left for the 10201 elements.

1.5 BASIC floating-point routines

Now that we know how to execute the fundamental floating-point calculations in BASIC, it is time to look at the functions.

A function can be written in general as

Y = F(X)

in which X is the argument, F is the function, and Y is the result. The floating point functions are written such that the argument X must be placed in the FAC before the function can be called. The result of the function call is placed back into the FAC.

The BASIC interpreter contains a number of useful functions which we can use:

Name	Address	Calculation time	Description
ABS	$BC58	0.0 ms	absolute value
ATN	$E30E	44.6 ms	arctangent
COS	$E264	27.9 ms	cosine
EXP	$BFED	26.6 ms	power of e
FRE	$B37D	0.6 ms	free memory space
INT	$BCCC	0.9 ms	greatest-int function
LOG	$B9EA	22.2 ms	natural logarithm
POS	$B39E	0.3 ms	cursor column
RND	$E097	3.5 ms	random number
SGN	$BC39	0.4 ms	sign
SIN	$E26B	24.5 ms	sine
SQR	$BF71	51.2 ms	square root
TAN	$E2B4	49.8 ms	tangent

The calculation times were obtained using pi as the argument. As you can see from the table, the vary enormously. Above all, the so-called transcendental functions such as COS, EXP, LOG, SIN, TAN, and ATN require a relatively large amount of time. These functions cannot be calculated exactly using the four basic math operators. Most functions are approximated using polynomials, which are functions of the form

$$y = a_0 + a_1*x + a_2*x^2 + a_3*x^3 + a_4*x^4 + a_4*x^5 + ...$$

The more terms such an expression has, the more exact the result will be, but the longer the calculation will take.

If one wants to calculate a polynomial, such as a 5th degree polynomial

1 + 2 + 3 + 4 + 5 = 15 multiplications
and 5 additions would be necessary.

There is a different method of solution which goes under the name "Horner Scheme" (polynomial substitution). The above equation can be reworked as follows:

$$y = ((((a_5 * x + a_4) * x + a_3) * x + a_2) * x + a_1) * x + a_0$$

Here only 5 multiplications and 5 additions are necessary. In general, a polynomial of degree n requires n multiplications and n additions compared to n*(n-1)/2 multiplications and n additions.

The simplicity of this procedure can be demonstrated with a simple BASIC program.

```
100 Y = A(N)
110 FOR I = N-1 TO 0 STEP -1
120 Y = Y * X + A(I)
130 NEXT
```

The program calculates the value of a polynomial of nth degree for the value x and returns the result in y. The array A(0) to A(N) contains the coefficients a0 through aN.

This routine for polynomial evaluation is the heart of all of the transcendental functions which the BASIC interpreter must calculate.

To use this routine, the argument of the polynomial must be in the FAC. The polynomial coefficients must be in the following format in the memory:

```
polynomial degree n
coefficient of nth     degree
coefficient of (n-1)th degree
  ...
coefficient of 1st     degree
coefficient of 0th     degree
```

The degree of the polynomial is stored as a one-byte value, which must follow the coefficients as a 5-byte floating-point value. The address of this coefficient field must be passed when the routine is called. The low byte must be in the accumulator and the high byte in the Y register. With this knowledge we can write a routine to calculate polynomials.

It is relatively complicated to place floating-point values into object code with a normal assembler. We can assign the value to a variable, use a monitor to find the variable table, note the corresponding 5 bytes of the variable value and then insert this into the source text with the .BYT command. ASSEMBLER/MONITOR 64 allows you to insert floating-point constants directly into the source. This is done with the .FLP pseudo-op (FLoating Point). The assembler then performs the conversion into the internal 5-byte representation.

Let us put our knowledge into practice and calculate the following polynomial:

$$y = 0.7 + 2.5 * x + 8.2 * x^2 - 2.3 * x^3 + 0.5 * x^4$$

```
ASSEMBLER 64 V2.0    PAGE 1

100:   033C                       .OPT P,00
110:                   ;
120:                   ; POLYNOMIAL CALCULATION
130:                   ;
140:   033C                       *=   828      ; CASSETTE BUFFER
150:                   ;
160:   E059            POLYNOM    =    $E059
170:                   ;
180:   033C A9 43                 LDA #< COEFF
190:   033E A0 03                 LDY #> COEFF
200:   0340 4C 59 E0              JMP POLYNOM
210:                   ;
220:   0343 04         COEFF      .BYT 4        ; DEGREE OF POLY.
230:   0344 80 00 00              .FLP 0.5      ; A(4)
```

```
240:    0349 82 93 33          .FLP -2.3    ; A(3)
250:    034E 84 03 33          .FLP 8.2     ; A(2)
260:    0353 82 20 00          .FLP 2.5     ; A(1)
270:    0358 80 33 33          .FLP 0.7     ; A(0)
280:                   ;
]033C-035D
NO ERRORS
```

The entire routine consists of passing the starting address and calling the polynomial function; the coefficients of the polynomial then follow in decreasing order.

How can we use our new function? It obviously won't work well with the SYS command--how are we supposed to pass the parameters and get the function value back? We need a function similar to the built-in functions like SIN, EXP, and so on.

The interpreter has already taken this case into consideration. It offers the USR function which you can freely define. We need only inform the interpreter of the starting address of the function. This starting address is placed in the usual form, low/high byte, at the addresses 785/786 ($0311/$0312).

```
POKE 785,828AND255 : POKE 786,828/256
```

Now enter the following, after the program has been assembled and the above line typed in:

```
? USR(1)
```

You get the value 9.6. A check of the formula confirms the correctness of the result.

$$y = 0.7 + 2.5 + 8.2 - 2.3 + 0.5 = 9.6$$

The following loop can be added for additional checks.

```
FOR I=-5 TO 5 : PRINT USR(I) : NEXT

    793.2
    397.1
    169.6
    54.9
    9.2
    .7
    9.6
    28.1
    60.4
    122.7
    243.2
```

This method for calculating polynomials is recommended whenever a program must repeatedly calculate the same polynomial. The execution time of this function at 12.5 ms is even shorter than many built-in functions. The calculation in BASIC requires about 45 ms. The more complicated the formula is, the faster the machine language version will run in comparison.

As you can gather from the above example, the coefficients, including their signs, must be in descending order (meaning that the coefficient of the highest power of x is first). If a power of x is missing in the polynomial, a zero must be inserted as its value.

The next example will calculate the factorial function. Factorial is a function which is first defined only for positive integer values and which consists of the product of all integers from one to the given number. For example

5! = 1 * 2 * 3 * 4 * 5 = 120

or

7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040

In mathematics, the function is also extended to include non-integers, which can again be approximated through a polynomial. This polynomial is only defined for values between zero and one, however; function values of other arguments must be counted backwards. For example:

4.3! = 4.3 * 3.3 * 2.3 * 1.3 * 0.3!

The factorial of 0.3 can be calculated with an eighth degree polynomial having the following coefficients:

a_0 = 1
a_1 = -.57719 1652
a_2 = .98820 6891
a_3 = -.89705 6937
a_4 = .91820 6857
a_5 = -.75670 4078
a_6 = .48219 9394
a_7 = -.19352 7818
a_8 = .03586 8343

We can now write a program to calculate this polynomial.

```
ASSEMBLER 64 V2.0    PAGE  1

100:   033C                     .OPT P1,00
110:                    ;
120:                    ; POLYNOMIAL FOR FACTORIAL CALCULATION
130:                    ;
140:   033C                     *=    828     ; CASSETTE BUFFER
150:                    ;
160:   E059             POLYNOM  =    $E059
170:                    ;
180:   033C A9 43               LDA #< COEFF
190:   033E A0 03               LDY #> COEFF
200:   0340 4C 59 E0            JMP POLYNOM
210:                    ;
220:   0343 08          COEFF   .BYT 8        ;8TH DEGREE POLY.
230:   0344 7C 12 EA            .FLP .035868343
240:   0349 7E C6 2C            .FLP -.193527818
250:   034E 7F 76 E2            .FLP .482199394
260:   0353 80 C1 B7            .FLP -.756704078
270:   0358 80 6B 0F            .FLP .918206857
280:   035D 80 E5 A5            .FLP -.897056937
290:   0362 80 7C FB            .FLP .988206891
300:   0367 80 93 C2            .FLP -.577191652
310:   036C 81 00 00            .FLP 1
]033C-0371
NO ERRORS
```

We can calculate the factorial values for arguments between 0 and 1 with PRINT USR(X). For example:

```
?USR(.1) =>  0.951350564
?USR(.5) =>  0.886227246
```

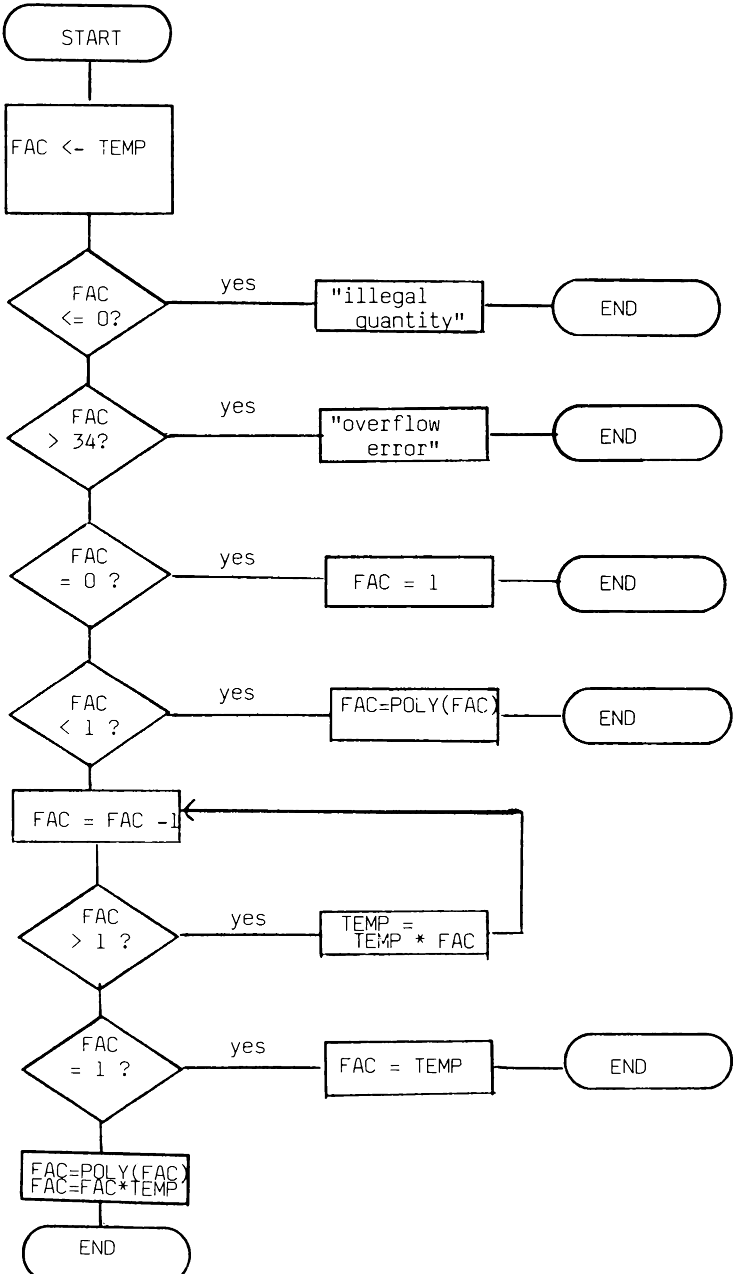
We can also calculate the factorial values for numbers outside of this range with a small BASIC routine.

```
10 INPUT "ARGUMENT"; X
20 IF X<0 OR X>33 THEN 10
30 IF X=0 THEN Y=1 : GOTO 70
40 Y=X : IF X<1 THEN Y = USR(X) : GOTO 70
50 X=X-1 : IF X>1 THEN Y=Y*X : GOTO 50
60 IF X<>1 THEN Y = Y * USR (X)
70 PRINT "FACTORIAL =";Y
```

Line 20 prevents negative values from being entered as well as values which have a factorial greater then 1E38. The argument 0 returns 1 by definition (line 30). In line 50 the argument is multiplied by the running product and decremented by one until it is less than or equal to one. A check is made in line 60 to determine if the argument is an integer. If this is not the case, the polynomial value must yet be multiplied by the result. Finally, the result is printed in line 70. For example:

```
0    =>  1
1    =>  1
1.5  =>  1.32934087
2    =>  2
3    =>  6
0.5  =>  .886227246
7.35 =>  10287.3151
```

Now that we have calculated the polynomial with a machine language routine of our own, we want to try to replace the entire BASIC program with a machine language program. By so doing we will become acquainted with more of the floating-point arithmetic routines. On the next page is a flow chart of the program operation.


START
FAC <- TEMP
FAC <= 0?
yes
"illegal quantity"
END
FAC > 34?
yes
"overflow error"
END
FAC = 0 ?
yes
FAC = 1
END
FAC < 1 ?
yes
FAC=POLY(FAC)
END
FAC = FAC -1
FAC > 1 ?
yes
TEMP = TEMP * FAC
FAC = 1 ?
yes
FAC = TEMP
END
FAC=POLY(FAC)
FAC=FAC*TEMP
END

Let's try our new function out. (Do not forget to first set the USR vector at address 785/786 to our routine--after turning the computer on this vector always points to "ILLEGAL QUANTITY").

```
?USR(0)   =>  1
?USR(1)   =>  1
?USR(2)   =>  2
?USR(3)   =>  6
?USR(.5)  =>  .886227246
?USR(4.5) =>  52.3427967
?USR(-1)  =>  ILLEGAL QUANTITY ERROR
?USR(40)  =>  OVERFLOW ERROR
```

What we had to do with a relatively complicated BASIC program before can now be done quickly and easily, simply by calling a function. We used some new routines in the machine language program which we want to discuss briefly.

FACMEM - This routine stores the contents of the floating point accumulator FAC at the address given in the X (low byte) and Y (high byte) register. The contents of the FAC are stored in the abbreviated 5-byte form.

MEMFAC - performs the opposite task. It gets a floating point number from memory and puts it in the FAC. This time the A register must contain the low byte of the address and the Y register the high byte.

COMPARE - We can compare two floating-point numbers to each other with this BASIC interpreter routine. The first number is in memory and is addressed through A (low byte) and Y (high byte). The second number must be in the FAC. If both numbers are the same, the accumulator (not the floating-point accumulator!) contains a zero and the Z flag is set. If the first value is smaller than the number in the FAC, the accumulator contains -1 ($FF) and the N flag is set. If the number in the FAC was smaller, the accumulator contains 1 and the N flag is cleared. This routine was used extensively in our program.

MEMPLUS - This routine consists of two subroutines. First the floating-point number pointed to by A and Y (low/high) is placed in ARG and then the routine for adding the FAC and ARG is called, which leaves the result in the FAC.

MEMMULT - This routine serves to multiply a number in memory with the FAC. The logic corresponds to that of MEMPLUS.

The addresses OVERFLOW and ILLQUAN call the appropriate routines for outputting error messages. It was unnecessary to check to see if the argument was greater than 34 in our case because this error message would automatically appear in the course of the multiplications.

The function for polynomial calculation can be put into yet another form.

$$y = a_0 * x + a_1 * x^3 + a_2 * x^5 + a_3 * x^7 + \ldots$$

This function is derived from the normal polynomial calculation by taking x^2 as the argument and multiplying the result by x once again.

$$y = x * (a_0 + a_1 * (x^2) + a_2 * (x^2)^2 + a_3 * (x^2)^3 + \ldots)$$

This routine is used for most built-in functions because the approximation polynomial is often in this form. The argument must usually first be brought into a specific value range for which the function is defined and then the result is modified corresponding to the original value.

We will calculate the following formula with this routine:

$$y = 6 * x + 0.5 * x^3 + - 0.11 * x^7$$

Note that a term is missing from the sequence (that with exponent 5), which we must replace with zero as the coefficient.

```
ASSEMBLER 64 V2.0     PAGE  1

100:    033C                    .OPT P,00
110:    033C                    *=   828
120:                  ;
130:    E043          POLY2     =    $E043
140:                  ;
150:    033C A9 43              LDA #< COEFF
160:    033E A0 03              LDY #> COEFF
170:    0340 4C 43 E0           JMP POLY2
180:                  ;
190:    0343 03       COEFF     .BYT 3     ; DEGREE OF POLY.
200:    0344 7D E1 47           .FLP -.11
210:    0349 00 00 00           .FLP 0
220:    034E 80 00 00           .FLP .5
230:    0353 83 40 00           .FLP 6
]033C-0358
NO ERRORS
```

Note that here the degree of the polynomial comes from the number of the highest power, not the highest power, because we have taken x out of the parentheses and use x2 as the argument.

Here are a few function values for a check:

```
USR(0)   = 0
USR(1)   = 6.39
USR(2)   = 1.92
USR(.75) = 4.69625427
```

At the close of our discussion of floating-point numbers we want to take up a problem which occurs often in programming: sorting number arrays. We will try to implement the following algorithm in machine language.

```
100 FOR I=1 TO N : FL=0
110 FOR J=N TO I STEP -1
120 IF A(J-1)>A(J) THEN H=A(J):A(J)=A(J-1):A(J-1)=H:FL=1
130 NEXT J
140 IF FL=0 THEN RETURN
150 NEXT I: RETURN
```

The program sorts the array A(N) and can be called as a subroutine with GOSUB 100. The program uses a bubble-sort algorithm. Two successive array elements are compared with each other. If the first element is greater than the second, the two elements are exchanged and a flag is set. This occurs in two nested loops. If no exchange occurs during the course of the inner loop, the array is sorted. In this case the flag remains zero and the sorting process ends prematurely. Otherwise, the smallest value will be found in A(0) after the first pass. The next pass compares elements 1 through N, then 2 through N, and so on. Do you remember how BASIC array elements are stored? There is a pointer which indicates the start of the array table. So that we do not have to search through this table to find the right array, we will agree that the array to sorted must have only one dimension and that it must also be the first array in the table.

```
100:   033C                       .OPT P1
110:   002F          ARRTAB       =    $2F
120:   0057                       *=   $57
130:   0057          IPNT         *=   *+2
140:   0059          JPNT         *=   *+2
150:   005B          JPNT1        *=   *+2
160:                 ;
170:   BBA2          MEMFAC       =    $BBA2
180:   BC5B          COMPARE      =    $BC5B
190:                 ;
200:   033C                       *=   828
210:                 ;
220:   033C A5 2F                 LDA  ARRTAB
230:   033E 18                    CLC
240:   033F A0 02                 LDY  #2
250:   0341 71 2F                 ADC  (ARRTAB),Y   ;ADD ARRAY
                                                      LENGTH
260:   0343 8D D9 03              STA  NPNT      ;POINTER N TO
                                                  ARRAY END
270:   0346 C8                    INY
```

```
280:   0347 A5 30               LDA  ARRTAB+1
290:   0349 71 2F               ADC  (ARRTAB),Y
300:   034B 8D DA 03            STA  NPNT+1
310:   034E AD D9 03            LDA  NPNT
320:   0351 38                  SEC
330:   0352 E9 05               SBC  #5
340:   0354 8D D9 03            STA  NPNT
350:   0357 B0 03               BCS  L1
360:   0359 CE DA 03            DEC  NPNT+1
370:                  ;
380:   035C A5 2F     L1        LDA  ARRTAB
390:   035E 18                  CLC
400:   035F 69 07               ADC  #7
410:   0361 85 57               STA  IPNT     ; POINTER I TO
                                                A(0)
420:   0363 A5 30               LDA  ARRTAB+1
430:   0365 69 00               ADC  #0
440:   0367 85 58               STA  IPNT+1
450:                  ;
460:   0369 A0 00     ILOOP     LDY  #0
470:   036B 8C D8 03            STY  FLAG     ;CLEAR FLAG
480:   036E AD D9 03            LDA  NPNT
490:   0371 85 59               STA  JPNT     ;J=N
500:   0373 AD DA 03            LDA  NPNT+1
510:   0376 85 5A               STA  JPNT+1
520:                  ;
530:   0378 A5 59     JLOOP     LDA  JPNT
540:   037A 38                  SEC
550:   037B E9 05               SBC  #5
```

```
560:   037D 85 5B              STA  JPNT1    ;POINTER J-1
570:   037F AA                 TAX
580:   0380 A5 5A              LDA  JPNT+1
590:   0382 E9 00              SBC  #0
600:   0384 85 5C              STA  JPNT1+1
610:   0386 A8                 TAY
620:   0387 8A                 TXA
630:   0388 20 A2 BB           JSR  MEMFAC   ;A(J-1) TO FAC
640:                  ;
650:   038B A5 59              LDA  JPNT
660:   038D A4 5A              LDY  JPNT+1
670:   038F 20 5B BC           JSR  COMPARE  ;COMPARE TO A(J)
680:   0392 30 12              BMI  NOSWAP
690:                  ;
700:   0394 A0 04              LDY  #4
705:   0396 8C D8 03           STY  FLAG     ;SET FLAG
710:   0399 B1 59     SWAP     LDA  (JPNT),Y
720:   039B AA                 TAX
730:   039C B1 5B              LDA  (JPNT1),Y
740:   039E 91 59              STA  (JPNT),Y  ;EXCHANGE A(J)
750:   03A0 8A                 TXA           ;AND A(J-1)
760:   03A1 91 5B              STA  (JPNT1),Y
770:   03A3 88                 DEY
780:   03A4 10 F3              BPL  SWAP
790:                  ;
800:   03A6 A5 59     NOSWAP   LDA  JPNT
810:   03A8 38                 SEC
820:   03A9 E9 05              SBC  #5
825:   03AB 85 59              STA  JPNT
```

```
03AD B0 02              BCS   TESTJ
03AF C6 5A              DEC   JPNT+1
                  ;
03B1 C5 57        TESTJ CMP   IPNT
03B3 D0 C3              BNE   JLOOP
03B5 A5 5A              LDA   JPNT+1   ;I=J
03B7 C5 58              CMP   IPNT+1
03B9 D0 BD              BNE   JLOOP
                  ;
03BB AD D8 03           LDA   FLAG     ;NO EXCHANGES?
03BE F0 17              BEQ   END
                  ;
03C0 A5 57              LDA   IPNT
03C2 18                 CLC
03C3 69 05              ADC   #5       ;I=I+1
03C5 85 57              STA   IPNT
03C7 90 02              BCC   TESTI
03C9 E6 58              INC   IPNT+1
                  ;
03CB CD D9 03     TESTI CMP   NPNT
03CE D0 99              BNE   ILOOP
03D0 A5 58              LDA   IPNT+1   ;I=N?
03D2 CD DA 03           CMP   NPNT+1
03D5 D0 92              BNE   ILOOP
                  ;
03D7 60           END   RTS
                  ;
03D8              FLAG  *=    *+1
03D9              NPNT  *=    *+2
]033C-03DB
NO ERRORS
```

This assembly language program takes over the task of the previous BASIC program. As said before, the array to be sorted must be one-dimensional. The program does not check to see if the array is allocated or if it is one dimensional--that is the responsibility of the user.

To sort an array, all that is required is to call the routine with

SYS 828

In order to get an idea of the speed of the program, we filled various large arrays with random numbers and first sorted them with BASIC and then with machine language. The results are found in the following table.

N	BASIC	Machine lang. routine
10	1"	0.0"
50	24"	0.4"
100	1' 37"	1.5"
200	6' 33"	6.3"
500	41'	38.7"
1000	2h 44'	2' 33.4"

You can see from the table that approximately four times as much time is required for twice as many elements to be sorted. If you must sort large arrays in BASIC, there comes a point at which the time requirement enters the hours range. Here our machine language is a good sixty times faster. If you have very large arrays and the machine language routine still takes too long for you, you must use a more efficient routine such as quicksort.

As an exercise, you might like to try to modify our routine so that it can sort integer arrays. What must be changed? For one, the different space requirement of an element must be taken into account--2 bytes must be added or subtracted as necessary instead of 5 bytes. For another, we should perform the comparison of the elements ourselves. We can compare the two-byte values directly instead of converting the integers to floating-point and then executing the floating-point comparison. In addition, the routine will be faster than the floating-point sort routine.

As a reference for your own applications, we present a table of all of the functions and operations of the BASIC interpreter which pertain to arithmetic.

Name	Address	Pointer to constants	Preparation	FAC	Function
MEMARG	$BA8C	A/Y	-	-	ARG := constant
FACARG	$BBFC	-	-	+	FAC := ARG
DIV	$BB12	-	A = EXP	+	FAC := ARG/FAC
MEMDIV	$BB0F	A/Y	-	+	FAC := cons/FAC
TIME10	$BAE2	-	-	+	FAC := FAC*10
DIV10	$BAFE	-	-	+	FAC := FAC/10
PLUS05	$B849	-	-	+	FAC := FAC+0.5
MEMFAC	$BBA2	A/Y	-	+	FAC := constant
FACARG	$BC0C	-	-	-	ARG := FAC
FACMEM	$BBD4	X/Y	-	-	constant := FAC
MINUS	$B853	-	A = EXP	+	FAC := ARG-FAC
MEMMIN	$B850	A/Y	-	+	FAC := cons/FAC
MULT	$BA2B	-	A = EXP	+	FAC := ARG*FAC
MEMMUL	$BA28	A/Y	-	+	FAC := cons*FAC

PLUS	$B86A	-	A = EXP	+	FAC := ARG+FAC
MEMPLU	$B867	A/Y	-	+	FAC := cons+FAC
POWER	$BF7B	-	A = EXP	+	FAC := ARG^FAC
PWRMEM	$BF78	A/Y	-	+	FAC := ARG^cons
POLY	$E059	A/Y	-	+	FAC := polynom.
POLY2	$E043	A/Y	-	+	FAC := polynom2
OR	$AFE6	-	-	+	FAC:=ARG OR FAC
AND	$AFE9	-	-	+	FAC:=ARG AND FAC
NOT	$AED4	-	-	+	FAC := NOT FAC
COMPAR	$BC5B	A/Y	-	-	comp FAC w/ cons
ROUND	$BC1B	-	-	+	round FAC
CHGSGN	$BFB4	-	-	+	FAC := -FAC

Conversions and standard functions are not listed since they were detailed in other places.

The "+" in the FAC column indicates that the contents of the FAC are changed; a "-" indicates that they remain the same. If an operation uses both the ARG and FAC, the accumulator should be loaded with the exponent of the FAC ($61) before the call.

With the logical operations AND, OR, and NOT the arguments are first converted to 16-bit integers, then the operation is executed bitwise, the result converted back to a floating-point number and placed back into the FAC.

The BASIC interpreter contains a number of floating point numbers which you can use for your own applications. They are listed in the following table.

Address	Constant	Decimal value	Significance
$AEA8	82 49 0F DA A1	3.14159265	PI
$B1A5	90 80 00 00 00	-32768	
$B9BC	81 00 00 00 00	1	
$B9C2	7F 5E 56 CB 79	.434255942	
$B9C7	80 13 9B 0B 64	.576584541	
$B9CC	80 76 38 93 16	.961800759	
$B9D1	82 38 AA 3B 20	2.88539007	
$B9D6	80 35 04 F3 34	.707106781	1/SQR(2)
$B9DB	81 35 04 F3 34	1.41421356	SQR(2)
$B9E0	80 80 00 00 00	-.5	
$B9E5	80 31 72 17 F8	.693147181	LOG(2)
$BAF9	84 20 00 00 00	10	
$BDB3	9B 3E BC 1F FD	99999999.9	
$BDB8	9E 6E 6B 27 FD	999999999	
$BDBD	9E 6E 6B 28 00	1E9	
$BFBF	81 38 AA 3B 29	1.44269504	1/LOG(2)
$BFC5	71 34 58 3E 56	2.14987637E-5	
$BFCA	74 16 7E B3 1B	1.4352314E-4	
$BFCF	77 2F EE E3 85	1.34226348E-3	
$BFD4	7A 1D 84 1C 2A	9.614011701E-3	
$BFD9	7C 63 59 58 0A	.0555051269	
$BFDE	7E 75 FD E7 C6	.240226385	
$BFE3	80 31 72 18 10	.693147186	
$BFE8	81 00 00 00 00	1	
$E08D	98 35 44 7A 00	11879546	
$E092	68 28 B1 46 00	3.92767774E-4	
$E2E0	81 49 0F DA A2	1.57079633	PI / 2
$E2E5	83 49 0F DA A2	6.28318531	PI * 2
$E2EA	7F 00 00 00 00	.25	
$E2F0	84 E6 1A 2D 1B	-14.3813907	
$E2F5	86 28 07 FB F8	42.0077971	

$E2FA	87 99 68 89 01	-76.7041703	
$E2FF	87 23 35 DF E1	81.6052237	
$E304	86 A5 5D E7 28	-41.3147021	
$E309	83 49 0F DA A2	6.28318531	PI * 2
$E33F	76 B3 83 BD D3	-6.84793912E-4	
$E344	79 1E F4 A6 F5	4.85094216E-3	
$E349	7B 83 FC B0 10	-.0161117015	
$E34E	7C 0C 1F 67 CA	.034209638	
$E353	7C DE 53 CB C1	-.054279133	
$E358	7D 14 64 70 4C	.0724571965	
$E35D	7D B7 EA 51 7A	-.0898019185	
$E362	7D 63 30 88 7E	.110932413	
$E367	7E 92 44 99 3A	-.142839808	
$E36C	7E 4C CC 91 C7	.19999912	
$E371	7F AA AA AA 13	-.33333316	
$E376	81 00 00 00 00	1	

SECTION 2 Interrupts

2.1 Interrupt programming

One area avoided by many machine language programmers is the programming of interrupts. We want to demonstrate the principles and prove that any fear of this subject is completely unfounded. We will explain what an interrupt is and what possibilities are opened up to the machine language programmer by using such new techniques.

First we must explain what we mean by the term "interrupted." What is interrupted, and how? Quite simple--the machine language program currently being executed is interrupted. This interruption is hardware-generated and can occur at any place within the program. What can interrupt a machine language program? To find this out we must give some consideration to the hardware construction of the processor.

The 6502 or 6510 microprocessor is housed within a 40-pin package, two pins of which have the designations

IRQ and NMI

These are abbreviations for Interrupt ReQuest and Non-Maskable Interrupt. If a signal from the outside is sent to one of these pins, the following events occur:

1. Signal on NMI pin

The processor finishes executing the current instruction and then attends to the interrupt:

1) The current value of the program counter is placed on the stack (first high byte then low byte).
2) The processor status register (the flags) is then pushed onto the stack.
3) The processor reads the contents of the addresses \$FFFA and \$FFFB and, interpreting them as the new value of the program counter, executes an indirect jump: JMP (\$FFFA). The program at this address will then be executed.

This program "services" the interrupt request.

2. Signal on IRQ pin

Here something similar happens. The current instruction is completed when the interrupt is registered. With IRQ, however, the processor first checks the state of the interrupt flag (bit 3 in the status register). Two cases are possible:

a) If this flag is set, the interrupt request is ignored and the program continues running.
b) If the flag was not set, the same procedure is executed as for NMI:
    1) The contents of the program counter and the flags are saved on the stack.
    2) The I flag is set so that any interrupt requests occurring during the interrupt service routine will be ignored.
    3) The processor gets the new value of the program counter from addresses \$FFFE and \$FFFF. The value to which these addresses point is used as the new value of the program counter.

How can we return to the interrupted program? There is a special machine language instruction for this purpose.

RTI - ReTurn from Interrupt

This instruction reverses the interrupt procedure. The value of the status register is fetched from the stack, the contents of the program counter is pulled from the stack and the program continues execution at this address. The interrupted program does not "notice" any of these activities. The processor saves only the status register--the other registers, if they are used by the interrupt routine, must be saved by the interrupt service routine before they are used and then restored before the return with RTI. For example

```
INTERRUPT       PHA   ; save accumulator
                TXA
                PHA   ; save X register
                TYA
                PHA   ; save Y register

                ...   ; interrupt service routine

                PLA
                TAY   ; restore Y register
                PLA
                TAX   ; restore X register
                PLA   ; restore accumulator
                RTI   ; return to interrupted program
```

The structure of an interrupt service routine is similar to that of a normal subroutine. The principle difference is that a subroutine is always called by the main program from specific place, whereas the interrupt routine is called from the outside by hardware and can be called at any time, from anywhere. In contrast to a subroutine call, the current contents of the processor status register are saved in addition to the return address. If this were not done, the interrupted program could not continue to function normally when control was returned to it. Now to the most important question:

How can an interrupt be generated?

There are several ways that this can happen in the Commodore 64. We will take a look at the ways in which an IRQ can be generated. The

video controller VIC 6569

and the I/O interface

CIA 6526

can both generate IRQs. The CIA here is the CIA at address $DC00.

A non-maskable interrupt (NMI) can be generated by

CIA 6526 (address $DD00)

as well as

the RESTORE key

In order to successfully program our own interrupt routines, a detailed knowledge of the capabilities and features of the peripheral interfaces is indispensable. We will discuss these interfaces in sufficient detail for our programming. More information can be obtained from the book The Anatomy of the Commodore 64.

2.2 The CIA 6526

The CIA (Complex Interface Adapter) 6526 is an interface module of the 65XX family which offers two 8-bit input /output ports, a serial 8-bit shift register, two cascadable 16-bit timers, a real time clock and several control lines.

The CIA has 16 registers which are addressed as successive memory locations by the microprocessor. The Commodore 64 has two of these chips; the first is located at addresses $DC00 to $DC0F, the second at $DD00 to $DD0F.

On the next few pages you will find a short description of these 16 control registers which we will get into in greater detail in the programs.

Register 0 Port A data register
Access: READ/WRITE
The contents of this register reflect the condition of the input/output port A.

Register 1 Port B data register
Access: READ/WRITE
The contents of this register reflect the condition of the input/output port B.

Register 2 Data direction register A
Access: READ/WRITE
The eight lines of data port A can be switched to input or output with this register. The corresponding bit of the data direction register must be 0 for input or 1 for output.

Register 3 Data direction register B
Access: READ/WRITE
This register has the same function as register 2, except for port B.

Register 4 Timer A LSB
Access: READ
When reading this register it returns the current condition of timer A (LSB).
Access: WRITE
By means of a write command to this register one can load the least-significant byte of the value from which the timer is to count down to zero.

Register 5 Timer A MSB

Access: READ

When reading, the contents of this register give the current condition of timer A (MSB).

Access: WRITE

One can load the high byte of the value from which timer A is to count down by writing to this register.

Register 6 Timer B LSB

This register corresponds in function to register 4, but applies to timer B.

Register 7 Timer B MSB

This register corresponds in function to register 5, except that it applies to timer B.

Register 8 Time of day (real-time clock) tenths of a second

Access: READ

When reading this register, bits 0-3 return the current state of the real-time clock, specifically, the tenths of a second in BCD format. Bits 4-7 are always zero.

Access: WRITE

By writing to register 8 you can, depending on the preselection of control register B (register 15), either set the tenths of a second on the real-time clock or select the alarm time. The tenths of a second must be given in BCD format, in which bits 4-7 must be zero.

Register 9 Time of day, seconds

Access: READ

By reading this register you get the seconds of the current clock time in BCD format. Bits 0-3 represent the one's place and bits 4-7 the ten's place.

Access: WRITE

You can either set the clock time or select the alarm time through a write access to this register, similar to register 8. The seconds count must be in BCD format.

Register 10 Time of day, minutes

Register 10 is organized similarly to register 9, but pertains to minutes.

Register 11 Time of day, hours

Access: READ

Reading this register returns the current hour value of the real-time clock. Bits 0-3 represents the one's place. Because the clock counts only from one to twelve, only one bit is necessary for the ten's place, namely bit 4. Bit 7 corresponds to the American time representation as a flag for before noon (AM, bit 7=0) or after noon (PM, bit 7=1).

Access: WRITE

The write access occurs in the same way as for the other real-time clock registers, although the significance of the individual bits is the same as for the read access.

Register 12 Serial shift register

Data is written to this register which will be shifted bit-by-bit out the serial port. By reading, the data shifted in can be read.

Register 13 Interrupt control register

Access:	READ (interrupt data)
Bit 0	timer A time-out
Bit 1	timer B time-out
Bit 2	clock time = alarm time
Bit 3	shift register full (for input) or empty (for output)
Bit 4	signal on FLAG pin
Bit 5-6	always zero
Bit 7	Bit seven is set if at least one of the bits 0-4 is set in both the interrupt control registers.

NOTE: READING THIS REGISTER ERASES IT!

Access: WRITE (interrupt mask)

The significance of bits 0-4 is the same as above. If bit seven is set in addition, one can enable the interrupt for the selected function. If bit 7 is cleared, a one bit disables the corresponding interrupt possibility.

Register 14 Control register A

Access:	READ/WRITE
Bit 0	0= timer A stop, 1= timer A start
Bit 1	1= timer A time-out is signaled on PB6
Bit 2	0= every timer A time-out creates a high signal on PB6, 1= every time-out on timer A inverts the state of PB6.

	Bit 3	1= timer A counts once from initial value to zero and stops (one shot), 0= timer A starts automatically after every time-out (continuous mode).
	Bit 4	1= absolute loading of a new value on timer A.
	Bit 5	0= timer counts system clock pulses, 1= timer counts pulses on CNT.
	Bit 6	0= serial port is input, 1= serial port is output.
	Bit 7	0= real time clock runs at 60 Hz, 1= real-time clock runs at 50 Hz.
Register 16	Control register B	
	Access:	READ/WRITE
	Bits 0-4	same meaning as the corresponding bits in control register A, but for timer B and PB7.
	Bits 5-6	These bits determine the trigger source of timer B. 00= timer B counts system clock pulses, 01= timer B counts CNT pulses, 10= timer B counts time-outs on timer A, 11= timer B counts time-outs on timer A when CNT=1.
	Bit 7	0= set, clock time, 1= set alarm time.

2.3 Using system interrupt

The simplest option for programming your own interrupt service routine is to add it to the system interrupt. What generates the system interrupt and what tasks does it perform?

The system interrupt is controlled by a timer in CIA 1. A timer is simply a counter which is decremented by one each system clock cycle. When the timer counts down to zero (also known as "timing-out"), it sends a signal to the IRQ input on the processor. The program will be interrupted and control passed to an interrupt routine found at $EA31. The timer consists of two 8-bit registers and can therefore count up to approximately 2^16 microseconds or 65 milliseconds. The system interrupt is generated every sixtieth of a second, that is, approximately every 16 ms.

What tasks does this routine perform? The first task is to check to see if the STOP key is pressed. If this is the case, a flag in the zero-page is set. This flag is checked before the execution of every BASIC program. If it is set, the BASIC program is stopped. The routine for checking the STOP key increments the internal clock TI which returns the time in sixtieths of a second.

The second task concerns the cursor. If the computer is in the direct mode or is awaiting input, it flashes the cursor. Every twentieth time the interrupt routine is called, the character over which the cursor is positioned is reversed. Thus the cursor blinks 20/60=3 times per second.

Another task is the supervision of the datasette. If the datasette is not under program control (LOAD or SAVE, for example), the motor is switched on or off depending on whether a key on the datasette is pressed or not.

The last and perhaps most important task of the interrupt routine consists of reading the keyboard. If a key is pressed, the key code is determined and the value placed in the keyboard buffer. The keyboard buffer is ten characters long. It is thereby possible to press several keys "outside" of an input routine which then appear on the screen when the program expects the characters. The number of characters in the keyboard buffer is also saved. When these tasks are finished, control exits the interrupt routine and returns to the interrupted program.

As we mentioned already, the processor gets the address of the interrupt routine from the memory locations $FFFE and $FFFF, which are in ROM. How can we change these values? Let's take a look at exactly what happens when an interrupt occurs. The address to which the interrupt vector points is $FF48.

```
******************************    IRQ jump point
FF48   48         PHA
FF49   8A         TXA
FF4A   48         PHA              save registers
FF4B   98         TYA
FF4C   48         PHA
FF4D   BA         TSX
FF4E   BD 04 01   LDA $0104,X      get break flag from stack
FF51   29 10      AND #$10         and test
FF53   F0 03      BEQ $FF58        not set?
FF55   6C 16 03   JMP ($0316)      BREAK routine
FF58   6C 14 03   JMP ($0314)      interrupt routine
```

First the contents of the registers are saved on the stack. Then the contents of the status register, which are automatically saved on the stack by the processor during an interrupt, are read and bit 4 is isolated. This is the BREAK flag which is set by a BRK command. The BRK command simulates an interrupt call in software. In order to distinguish it from a hardware interrupt, the BREAK flag in the status register is set. The appropriate indirect jump is made based on this. If the flag was set, a jump will be made over the vector at $0316/$0317, else via the vector at $0314/$0315.

The vector $0314/$0315 is the actual interrupt vector and normally points to the previously mentioned address $EA31.

If we want to execute additional tasks inside the interrupt routine, we can proceed in the following manner:

We change the interrupt vector so that it points to our own routine. When our routine is finished, we jump to the normal system interrupt routine so that these tasks can be performed. Using this procedure we can execute a second "job" sixty times per second, independent of the main program. This routine must naturally not last longer than one sixtieth of second, otherwise there will be no time for the main program. A long interrupt routine is characterized by a slowing down of the main program.

What could the computer execute 60 times per second? Here your imagination is the only limiting factor. You could, for example, flash the screen or text on the screen, similar to the way the cursor is flashed. So that the blinking does not go too fast, a counter must be used so that the event occurs only once every given number of interrupt calls.

```
100:  C000                         .OPT P1
110:                   ;
120:                   ;FLASH BACKGROUND/BORDER
130:                   ;
140:  C000                         *=    $C000
150:                   ;
160:  D020             BORDER      =     $D020    ;BORDER
170:  D021             BACK        =     $D021    ;BACKGROUND
180:  EA31             IRQROUT     =     $EA31
190:  0314             IRQVEC      =     $314
200:                   ;
210:  001E             NUMBER      =     30       ;EVERY HALF SECOND
220:                   ;
230:  C000 78          INIT        SEI            ;DISABLE INTERPETS
240:  C001 A9 0D                   LDA   #<BLINK
250:  C003 A0 C0                   LDY   #>BLINK
260:  C005 8D 14 03                STA   IRQVEC   ;IRQ-VECTOR TO
                                                   SCREEN
270:  C008 8C 15 03                STY   IRQVEC+1
280:  C00B 58                      CLI
290:  C00C 60                      RTS
300:                   ;
310:  C00D CE 26 C0 BLINK          DEC   COUNT    ;DECREMENT COUNTER
320:  C010 D0 11                   BNE   DONE
330:  C012 A9 1E                   LDA   #NUMBER
340:  C014 8D 26 C0                STA   COUNT    ;RESET COUNTER
350:                   ;EXCHANGE COLORS
360:  C017 AE 21 D0                LDX   BACK
370:  C01A AD 20 D0                LDA   BORDER
380:  C01D 8D 21 D0                STA   BACK
```

```
390:    C020 8E 20 D0          STX  BORDER
400:                   ;
410:    C023 4C 31 EA DONE     JMP  IRQROUT
420:                   ;
430:    C026 1E        COUNT    .BYT NUMBER  ;COUNTER
]C000-C027
NO ERRORS
```

Let's take a closer look at the above program. The routine INIT takes care of the initialization and sets the interrupt vector to the blink routine. Note that interrupts otherwise possible while the vector is being changed are blocked by the SEI instruction. If such an interrupt were to be generated when the low byte pointed to the new value while the high byte still pointed to the old routine, the processor would branch to an undefined place in memory and would in all likelihood "crash." If the I bit is set, interrupts can be enabled with CLI and we return with RTS. Now the new interrupt routine is active.

The following happens at the next interrupt call: First, the memory location COUNT is decremented by one. If this does not yield a value of zero, execution branches to the label DONE and the normal interrupt routine is executed from there. If, however, the counter was zero, it is first reset to value 30 and the background and border colors are exchanged, creating the flash effect.

We can activate our routine by calling it with SYS 12*4096. Immediately the screen begins to flash twice a second. This interrupt routine runs completely independently of a BASIC or machine language program until the interrupt vector is set back to the old routine. This is done by pressing the RUN/STOP-RESTORE keys, for example.

We can easily change the flash frequency with the label NUMBER; it gives the number of sixtieths of a second between color changes.

As a second example of interrupt routines, we want to change the cursor attributes. The cursor should not blink, but only be represented as an inverted character. We cannot simply place our new routine ahead of the normal interrupt routine. We must replace the part which pertains to the cursor blinking.

```
******************************   Interrupt routine
EA31   20 EA FF   JSR $FFEA      stop key, increment time
EA34   A5 CC      LDA $CC        blink flag for cursor
EA36   D0 29      BNE $EA61      not blinking, then continue
EA38   C6 CD      DEC $CD        decrement blink counter
EA3A   D0 25      BNE $EA61      not zero, then continue
EA3C   A9 14      LDA #$14       set blink counter back to 20
EA3E   85 CD      STA $CD        and save
EA40   A4 D3      LDY $D3        cursor column
EA42   46 CF      LSR $CF        blink switch zero then C=1
EA44   AE 87 02   LDX $0287      color under cursor
EA47   B1 D1      LDA ($D1),Y    set character code
EA49   B0 11      BCS $EA5C      blink switch on, continue
EA4B   E6 CF      INC $CF        blink switch on
EA4D   85 CE      STA $CE        save character under cursor
EA4F   20 24 EA   JSR $EA24      calculate pntr in color RAM
EA52   B1 F3      LDA ($F3),Y    get color code
EA54   8D 87 02   STA $0287      and save
EA57   AE 86 02   LDX $0286      color code under cursor
EA5A   A4 CE      LDA $CE        character under cursor
```

```
EA5C    49 80       EOR #$80        invert RVS bit
EA5E    20 1C EA    JSR $EA1C       set cursor char and color
```

The cursor blinking is realized as follows. First a check is made to see if the cursor is active. If not, the following part is skipped. Otherwise the blink counter is decremented. If it is not zero, the remaining portion will skipped. Otherwise, the phase of the cursor is checked to see if it is in the inverted phase. The current or stored character is inverted and displayed depending on this. The same happens in the color RAM with the character color and the current cursor color.

We want to modify the routine so that we have a steady cursor. We can do this with the following program.

```
100:  C000                 .OPT P1
110:            ;
120:            ;MODIFY CURSOR
130:            ;
140:  FFEA      STOP     =   $FFEA   ;READ STOP KEY
150:  00CC      CURSFLAG =   $CC     ;FLAG FOR VISIBLE
                                      CURSOR
160:  00CF      REVERSE  =   $CF     ;FLAG FOR INVERTED
                                      CHARACTER
170:  0287      CURSCOL  =   $287    ;COLOR UNDER
                                      CURSOR
180:  00CE      CURSCHAR =   $CE     ;CHARACTER UNDER
                                      CURSOR
190:  00D1      CHAR     =   $D1     ;POINTER IN VIDEO
                                      RAM
200:  00F3      COLOR    =   $F3     ;POINTER IN COLOR
```

```
210:  EA24            SETCOL   =    $EA24    ;SET POINTER TO
                                              COLOR RAM
220:  00D3            COLUMN   =    $D3      ;CURSOR COLUMN
230:  0286            COLSTR   =    $286     ;CURSOR COLOR
240:  0314            IRQVEC   =    $314     ;IRQ VECTOR
250:  EA61            CONTIRQ  =    $EA61    ;CONTINUE IRQ
260:                  ;
270:  C000 78         INIT     SEI           ;DISABLE INTER-
                                              RUPTS
280:  C001 A9 0D               LDA  #<NEWCURS
290:  C003 A0 C0               LDY  #>NEWCURS
300:  C005 8D 14 03            STA  IRQVEC   ;IRQ VECTOR TO
                                              NEW ROUTINE
310:  C008 8C 15 03            STY  IRQVEC+1
320:  C00B 58                  CLI
330:  C00C 60                  RTS
340:                  ;
350:  C00D 20 EA FF NEWCURS    JSR  STOP     ;TEST STOP KEY
360:  C010 A5 CC               LDA   CURSFLAG  ;CURSOR
                                                 VISIBLE?
370:  C012 D0 1D               BNE  NOCURSOR  ;NO
380:  C014 A4 D3               LDY  COLUMN   ;CURSOR COLUMN
390:  C016 A5 CF               LDA  REVERSE   ;CHARACTER AL-
                                               READY REVERSED?
400:  C018 D0 17               BNE  NOCURSOR  ;YES
410:  C01A E6 CF               INC  REVERSE   ;SET FLAG FOR
                                               REVERSE
420:  C01C 20 24 EA            JSR  SETCOL   ;POINTER IN COLOR
                                              RAM
430:  C01F B1 D1               LDA  (CHAR),Y  ;POINTER AT
                                                CURSOR POSITION
440:  C021 85 CE               STA  CURSCHAR  ;SAVE
450:  C023 49 80               EOR  #$80     ;FLIP RVS BIT
```

```
460:   C025 91 D1                STA  (CHAR),Y  ;AND IN VIDEO
                                                 RAM
470:   C027 B1 F3                LDA  (COLOR),Y  ;COLOR
480:   C029 8D 87 02             STA  CURSCOL  ;SAVE
490:   C02C AD 86 02             LDA  COLSTR  ;CURSOR COLOR
500:   C02F 91 F3                STA  (COLOR),Y  ;SET
510:   C031 4C 61 EA NOCURSOR JMP  CONTIRQ  ;CONTINUE IRQ
]C000-C034
NO ERRORS
```

When you activate this routine with SYS 12*4096, the cursor is simply replaced with an inverted character. You can modify this routine according to your own taste; the cursor color need not be the same as the character color, for instance--it could always be one color. Instead of the inverted representation you can do something different, such as display a line. It would also be possible to leave the character unchanged and simply alternate between two colors. You should consider these examples only as suggestions for your own experiments with the interrupt routine--the possibilities are numerous.

Here we can briefly discuss a method of inhibiting the STOP key. Because the test for the STOP key is the first thing done in the interrupt routine, we can bypass this test by changing the interrupt vector to point beyond it. A running BASIC program can no longer be stopped with the STOP key:

```
POKE 788, PEEK(788)+3
```

The vector is simple incremented by three bytes so that the test is bypassed. A disadvantage of this method is that the internal clock TI and TI$ no longer run. This is because the routine that tests the STOP key also keeps the clock updated.

An additional application of the system routine is the execution of a certain action upon a keypress. It is possible, for example to call a hardcopy routine which outputs the screen contents to a printer by pressing a function key.

The interrupt routine can check to see if the key was pressed. If this is the case, a routine can be called which performs the special task. Here too, many applications are possible, such as switching between two screen pages. Here is an example of this:

```
100:  033C                .OPT P1
110:            ;
120:            ;SWITCH SCREEN PAGES
130:            ;
140:  0003      PNT1     =   3
150:  0005      PNT2     =   5
160:  DD00      VIDEOMAP =   $DD00   ;16K VIDEO RANGE
170:  0288      VIDEOPGE =   648
180:  0314      IRQVEC   =   $314
190:  EA31      IRQOLD   =   $EA31
200:  D000      CHARGEN  =   $D000   ;CHARACTER GEN-
                                      ERATOR
210:  D800      COLOR    =   $D800   ;COLOR RAM
220:  C000      COLOR2   =   $C000   ;STORAGE FOR COLOR
                                      RAM
230:  0001      PORT     =   1       ;PROCESSOR PORT
240:  028D      CTRL     =   653     ;FLAG FOR CONTROL
250:  00C5      KEY      =   $C5     ;LAST KEY
```

```
260:    0004             F1        =    4          ;MATRIX NUMBER OF
                                                    F1 KEY
270:                     ;
280:    033C                       *=   828
290:                     ;
300:    033C 78          INIT      SEI
310:    033D 20 94 03              JSR  SETCHAR    ;COPY CHARACTER
                                                    GENERATOR
320:    0340 A9 4C                 LDA  #<TEST
330:    0342 A0 03                 LDY  #>TEST
340:    0344 8D 14 03              STA  IRQVEC     ;POINTER TO NEW
                                                    ROUTINE
350:    0347 8C 15 03              STY  IRQVEC+1
360:    034A 58                    CLI
370:    034B 60                    RTS
380:                     ;
390:    034C AD 8D 02    TEST      LDA  CTRL       ;CONTROL KEY
                                                    PRESSED?
400:    034F 29 04                 AND  #%100
410:    0351 F0 09                 BEQ  NOSWITCH   ;NO
420:    0353 A5 C5                 LDA  KEY        ;F1 PRESSED
430:    0355 C9 04                 CMP  #F1
440:    0357 D0 03                 BNE  NOSWITCH   ;NO
450:    0359 20 5F 03              JSR  SWITCH     ;EXCHANGE PAGES
460:    035C 4C 31 EA    NOSWITCH  JMP  IRQOLD
470:                     ;
480:    035F A0 00       SWITCH    LDY  #0
490:    0361 84 03                 STY  PNT1
500:    0363 84 05                 STY  PNT2
510:    0365 A9 D8                 LDA  #>COLOR    ;POINTER TO COLOI
                                                    RAM
520:    0367 85 04                 STA  PNT1+1
530:    0369 A9 C0                 LDA  #>COLOR2   ;POINTER TO
```

```
                                                STORAGE FOR COLOR
540:   036B 85 06            STA  PNT2+1
550:   036D A2 04            LDX  #4        ;NUMBER OF PAGES
560:   036F B1 03   SWAP     LDA  (PNT1),Y
570:   0371 48               PHA
580:   0372 B1 05            LDA  (PNT2),Y
590:   0374 91 03            STA  (PNT1),Y   ;SWAP COLOR
                                              STORAGE
600:   0376 68               PLA
610:   0377 91 05            STA  (PNT2),Y
620:   0379 C8               INY
630:   037A D0 F3            BNE  SWAP
640:   037C E6 04            INC  PNT1+1
650:   037E E6 06            INC  PNT2+1
660:   0380 CA               DEX            ;NEXT PAGE
670:   0381 D0 EC            BNE  SWAP
680:   0383 AD 00 DD         LDA  VIDEOMAP
690:   0386 49 03            EOR  #%11      ;ACCESS ADDRESS
                                             FOR VIC
700:   0388 8D 00 DD         STA  VIDEOMAP
710:   038B AD 88 02         LDA  VIDEOPGE
720:   038E 49 C0            EOR  #$C0      ;SCREEN PAGE
730:   0390 8D 88 02         STA  VIDEOPGE
740:   0393 60               RTS
750:                ;
760:   0394 A0 00   SETCHAR  LDY  #0
770:   0396 84 03            STY  PNT1
780:   0398 A9 D0            LDA  #>CHARGEN
782:   039A 85 04            STA  PNT1+1
784:   039C A2 10            LDX  #$10
786:   039E A9 33   LOOP     LDA  #$33
790:   03A0 85 01            STA  PORT      ;TURN ON CHARACTER
                                             GENERATOR
```

```
800:    03A2 B1 03          LDA  (PNT1),Y
810:    03A4 48             PHA
820:    03A5 A9 30          LDA  #$30
830:    03A7 85 01          STA  PORT     ;TURN ON RAM
840:    03A9 68             PLA
850:    03AA 91 03          STA  (PNT1),Y
860:    03AC C8             INY
870:    03AD D0 EF          BNE  LOOP
880:    03AF E6 04          INC  PNT1+1   ;NEXT PAGE
890:    03B1 CA             DEX
900:    03B2 D0 EA          BNE  LOOP
910:    03B4 A9 37          LDA  #$37     ;STANDARD CONFIG-
                                           URATION
920:    03B6 85 01          STA  PORT
930:    03B8 60             RTS
]033C-03B9
NO ERRORS
```

This program allows us to switch between two screen pages. The first page lies as usual at $0400, while we have placed the second page at address $C400. It is also possible for the second page to have its own sprites. The sprite pointers must be at address $C7F8. The base address of this screen is therefore located at $C000. For example, the address space from $C800 to $CFFF is available for storing sprites and offers room for 32 different sprite patterns (sprite numbers 32 to 63). Because the video controller always expects the color RAM to be at address $D800, we can store the color of the currently invisible page at $C000 to $C3FF. Furthermore, we must take into consideration the fact that the VIC in the upper 16K from $C000 to $FFFF cannot access the character generator ROM. We therefore copy the character generator from ROM to the RAM at the same addresses.

The actual interrupt routine checks bit 2 in the flag for the control key. If this bit is set, the control key was pressed. If the F1 key was also pressed, the routine which exchanges the color storage and sets the parameters for displaying the other screen page is called. Finally, a branch is made to the normal interrupt routine.

When we assemble our program and activate it with SYS 828, we can switch to the second screen page simply by pressing the CTRL and F1 keys at the same time. The first time you make the switch, you should clear the screen because random values will be left in the video RAM. Pressing the two keys again returns you to the original screen. The cursor will remain at the same place.

As an additional suggestion, you could try to display the clock time during the interrupt routine. The time will then appear on the screen at all times, independent of other program activities. You can find such a routine in the book <u>Tricks and Tips for the Commodore 64</u>.

Another equally interesting possibility for an interrupt routine involves sprites. One or more sprites can be moved during each interrupt. Programming the sound chip can also be done in an interrupt routine. Here sound sequences or entire pieces of music can be played completely independently of other programs. You can see that the possibilities which are offered to you are virtually unlimited. Before we begin to generate our own interrupts, we will present two more routines which are tied to the system interrupt.

You may find the following program quite useful if you use the user port to interface with devices of your own. The program is tied into the system interrupt and gives you a continual display of the individual bits of the user port. The direction register is displayed in the first screen line. This allows you to see which lines are set for input (=0) or output (=1). In the next line the states of the user port lines are displayed; a 0 indicates a low signal, a high signal is displayed as a 1. Both displays are also given in hexadecimal form.

```
100:  033C                        .OPT P1
110:                   ;
120:                   ;USER PORT DISPLAY
130:                   ;
140:  DD00             CIA2      =    $DD00
150:  DD01             USERPORT  =    CIA2+1
160:  DD03             DIRECTION=     CIA2+3
170:                   ;
180:  0288             VIDEOPGE  =    648
190:  D800             COLORRAM  =    $D800
200:                   ;
210:  0007             COLOR     =    7         ;YELLOW
220:                   ;
230:  0314             IRQVEC    =    $314
240:  EA31             IRQOLD    =    $EA31
250:  00FB             PNT       =    $FB
260:                   ;
270:  033C                       *=   828
280:  033C 78          INIT      SEI
290:  033D AD 14 03              LDA  IRQVEC
300:  0340 49 7E                 EOR  #< IRQOLD ^ DISP
310:  0342 8D 14 03              STA  IRQVEC
```

```
320:    0345 AD 15 03             LDA  IRQVEC+1
330:    0348 49 E9                EOR  #> IRQOLD ^ DISP
340:    034A 8D 15 03             STA  IRQVEC+1
350:    034D 58                   CLI
360:    034E 60                   RTS
370:                    ;
380:    034F A5 FB      DISP      LDA  PNT
390:    0351 48                   PHA            ;SAVE POINTER
400:    0352 A5 FC                LDA  PNT+1
410:    0354 48                   PHA
420:    0355 A9 1C                LDA  #28
430:    0357 85 FB                STA  PNT       ;POINTER TO VIDEO
                                                  RAM
440:    0359 AD 88 02             LDA  VIDEOPGE
450:    035C 85 FC                STA  PNT+1
460:    035E AD 03 DD             LDA  DIRECTION
470:    0361 A0 00                LDY  #0        ;DIRECTION IN TOP
                                                  LINE
480:    0363 20 77 03             JSR  DISPLAY   ;DISPLAY
490:    0366 AD 01 DD             LDA  USERPORT
500:    0369 A0 28                LDY  #40       ;USER PORT IN
                                                  SECOND LINE
510:    036B 20 77 03             JSR  DISPLAY   ;DISPLAY
520:    036E 68                   PLA
530:    036F 85 FC                STA  PNT+1     ;GET POINTER BACK
540:    0371 68                   PLA
550:    0372 85 FB                STA  PNT
560:    0374 4C 31 EA             JMP  IRQOLD    ;TO NORMAL IRQ
570:                    ;
580:    0377 48         DISPLAY   PHA            ;SAVE VALUE FOR
                                                  HEX DISPLAY
590:    0378 A2 08                LDX  #8
600:    037A 0A         LOOP      ASL            ;HIGHEST BIT IN
```

```
                                                           CARRY
610:     037B 48                      PHA
620:     037C A9 30                   LDA   #"0"      ;DISPLAY ZERO
630:     037E 90 02                   BCC   NULL
640:     0380 A9 31                   LDA   #"1"      ;DIPLAY ONE WHEN
                                                       C=1
650:     0382 91 FB         NULL      STA   (PNT),Y
660:     0384 A9 07                   LDA   #COLOR    ;AND SET COLOR
670:     0386 99 1C D8                STA   COLORRAM+28,Y
680:     0389 C8                      INY
690:     038A 68                      PLA
700:     038B CA                      DEX             ;NEXT BIT
710:     038C D0 EC                   BNE   LOOP
720:                        ;
730:                        ;HEX DIPLAY
740:                        ;
750:     038E C8                      INY
760:     038F 68                      PLA
770:     0390 48                      PHA
780:     0391 4A                      LSR
790:     0392 4A                      LSR             ;SHIFT UPPER
                                                       NYBBLE DOWN
800:     0393 4A                      LSR
810:     0394 4A                      LSR
820:     0395 20 99 03                JSR   ASCII     ;HIGH NYBBLE
830:     0398 68                      PLA
840:     0399 29 0F         ASCII     AND   #%1111
850:     039B C9 0A                   CMP   #10
860:     039D 90 02                   BCC   SMALLER
870:     039F 69 06                   ADC   #6
880:     03A1 69 30         SMALLER   ADC   #"0"      ;CONVERT TO ASCII
890:     03A3 29 3F                   AND   #%111111   ;CONVERT TO
                                                         SCREEN CODE
```

```
900:    03A5 91 FB              STA  (PNT),Y
910:    03A7 A9 07              LDA  #COLOR  ;AND SET COLOR
920:    03A9 99 1C D8           STA  COLORRAM+28,Y
930:    03AC C8                 INY
940:    03AD 60                 RTS
]033C-03AE
NO ERRORS
```

We have done the initialization somewhat differently. We eXclusive-OR the old value of the IRQ vector with the new value and thereby arrive at a switch between the old value $EA31 and our new routine DISPLAY with every call of SYS 828. Thus if you want to turn the display off, simply enter SYS 828 again and the interrupt vector will be set back to $EA31.

The program itself consists of a main program which at the start saves the necessary memory locations on the stack so that other programs which might use these addresses are not disturbed. Then the pointer PNT is set to the 28th column of the first screen line, the value of the data direction register loaded, and the subroutine for display called. After this, Y is set to 40 so that the display routine writes one line lower, and the contents of the user port are passed. Now the pointers are restored and the branch is made to the normal IRQ routine.

The display routine prints the value in the accumulator first in binary and then in hexadecimal. We use a loop for the 8 bit positions in the binary representation. During each pass through the loop, the uppermost bit is shifted into the carry with ASL. If this bit was a "1," then the carry is set and we output a "1" on the screen, otherwise a

"0". After the binary display, the values temporarily stored on the stack are displayed in hexadecimal. The upper nybble (four bits) is right-shifted into the lower nybble, then converted to ASCII and displayed on the screen. The same is then done for the lower nybble.

When you activate the routine with SYS 828, the following representation appears on the screen, for example:

```
00000000 00
FFFFFFFF FF
```

This is the value after the computer is turned on. The user port is set to input and the open inputs yield a high signal. Switch the user port to output and write 100 to it.

```
POKE 56579, 255
POKE 56577, 100
```

You get the following display:

```
FFFFFFFF FF
01100100 64
```

The bits 2, 5, and 6 are set; this corresponds to the hex number $64 or decimal 100.

The next routine is similar to the previous. It provides us with a continual display of the remaining free memory space. We accomplish this like the FRE function each interrupt. We simply calculate the difference between the end of the array table and the start of the strings. In contrast to the real FRE function, no garbage collection is performed in the interrupt routine. If we want to display the free space in decimal, it requires a conversion to floating-point format and again to ASCII representation. This takes time, although we could put up with that. The main disadvantage of such a method is that we must save all used memory locations on the stack because the interrupt can stop the program at any place. We would have to save 20 or more memory places, which, for one, requires time, and for another, requires quite a bit of space on the stack. We will therefore display the free memory space in hexadecimal format. This is equally informative and significantly faster.

```
100:   033C                        .OPT P1
110:                    ;
120:                    ;DISPLAY FREE MEMORY SPACE
130:                    ;
140:   0031             ARRAYEND =    $31
150:   0033             STRGSTRT =    $33
160:                    ;
170:   0400             VIDEO    =    1024
180:   D800             COLOR    =    $D800
190:                    ;
200:   0007             CLRCODE  =    7          ;YELLOW
210:                    ;
220:   0314             IRQVEC   =    $314
230:   EA31             IRQOLD   =    $EA31
240:                    ;
250:   C000             INIT     *=   828
260:   033C 78                   SEI
270:   033D A9 49                LDA  #<FREE
280:   033F A0 03                LDY  #>FREE
290:   0341 8D 14 03             STA  IRQVEC
300:   0344 8C 15 03             STY  IRQVEC+1
```

```
310:    0347 58                     CLI
320:    0348 60                     RTS
330:                    ;
340:    0349 38         FREE        SEC
350:    034A A5 33                  LDA   STRGSTRT
360:    034C E5 31                  SBC   ARRAYEND
370:    034E 08                     PHP
380:    034F A0 25                  LDY   #37
390:    0351 20 61 03               JSR   DISPLAY
400:    0354 28                     PLP
410:    0355 A5 34                  LDA   STRGSTRT+1
420:    0357 E5 32                  SBC   ARRAYEND+1
430:    0359 A0 23                  LDY   #35
440:    035B 20 61 03               JSR   DISPLAY
450:    035E 4C 31 EA               JMP   IRQOLD
460:    0361 48         DISPLAY     PHA
470:    0362 4A                     LSR
480:    0363 4A                     LSR
490:    0364 4A                     LSR
500:    0365 4A                     LSR
510:    0366 20 6A 03               JSR   ASCII
520:    0369 68                     PLA
530:    036A 29 0F      ASCII       AND   #%1111
540:    036C C9 0A                  CMP   #10
550:    036E 90 02                  BCC   SMALLER
560:    0370 69 06                  ADC   #6
570:    0372 69 30      SMALLER     ADC   #"0"
580:    0374 29 3F                  AND   #%111111
590:    0376 99 00 04               STA   VIDEO,Y
600:    0379 A9 07                  LDA   #CLRCODE
610:    037B 99 00 D8               STA   COLOR,Y
620:    037E C8                     INY
630:    037F 60                     RTS
```

```
]033C-0380
NO ERRORS
```

The amount of free space is displayed continually on the screen after calling the routine with SYS 828. Try the following BASIC program and watch the display.

```
100 DIM A$(200)
110 FOR I= 1 TO 200 : A$(I) = CHR$(1) : NEXT
```

You can watch the free memory space get smaller and smaller. Now enter ?FRE(0). During the approximately 4 seconds which this function requires, you can see that the free memory space changes constantly.

If you work with ASSEMBLER 64, you can see how the symbol table is created in pass 1 because it uses the same pointers as BASIC.

2.4 Video controller interrupts

Now that we have learned how to use the timer controlled system interrupt for our own purposes, we want to be able to generate interrupts ourselves and execute corresponding routines.

We will take a look at the chips which are capable of generating interrupt requests. These include the two CIA 6526s, of which CIA 1 can generate an IRQ and CIA 2 an NMI. The video controller VIC 6567 can also generate an interrupt. The registers necessary for the interrupt will be described here.

Register 18 Access READ

A read access to this register returns the number of the raster line currently being displayed on the screen. Because the raster line number can be larger than 255, bit 7 of register 17 is used for the carry.

Access WRITE

When your write to this register, you can set the raster line at which the VIC will generate an interrupt request.

Register 25 Interrupt Request Register

This register signals an interrupt request by the VIC. The individual bits stand for various interrupt sources.

Bit 0 The video controller reached the raster line which was set in register 18.

Bit 1 A sprite collided with a background character. The number of the sprite is recorded in register 31.

Bit 2	Two sprites collided. The numbers of the sprites involved are saved in register 30.
Bit 3	A strobe was generated by the light pen. The X and Y positions are recorded in registers 19 and 20.
Bit 7	This bit is set whenever any of the others are set.

Register 26 Interrupt Mask Register

The significances of the bits correspond to the those in register 25. An interrupt request is generated only if the corresponding bit in the IMR is set and the interrupts are enabled.

Register 30 Sprite-sprite collision

If two sprites collide, the bits are set according to the numbers of the sprites involved. Bit 2 in register 25 is also set. These bits must be reset after reading the results.

Register 31 Sprite-background collision

If a sprite encounters a background character, the number of the sprite is recorded in this register and bit 1 of register 25 is set at the same time. This register must also be reset after use.

The video controller can generate interrupts based on four different events:

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

The video controller records these conditions in register 25 if one of the four events occurs. The Interrupt Mask Register (IMR) decides whether an interrupt request to the processor will generated. If a bit is set in this register, the corresponding event will cause an interrupt request to be generated. This register may be read from and written to like a RAM memory location. If a bit is to be set or cleared, that is, an interrupt is to be enabled or disabled, the appropriate procedure must be followed.

Setting a bit

Set the desired bit and also bit 7. Then write the resulting value to the interrupt mask register. If, for example, you want to permit an interrupt by a sprite-sprite collision (bit 2):

```
LDA #%10000100
STA IMR
```

You set the desired bit and bit number 7. The other bits (0, 1, and 3) will not be changed.

Clearing a bit

If you want to disable an interrupt, the corresponding bit must be cleared. You must set the desired bit, but bit 7 must be cleared. For example, to disable the sprite-sprite collision:

```
LDA #%00000100
STA IMR
```

Here too the unset bits remain unchanged. It is not possible to read the interrupt mask register. If a program requires the value of the interrupt mask, it can be stored in RAM at the same time it is written to the VIC in order to save the value.

A second peculiarity must be taken into account for the Interrupt Request Register (IRR). If the video controller has generated an interrupt request, this register must be reset, otherwise another interrupt will be generated immediately following the exit from the interrupt routine. A set bit can be cleared in the same manner as is done in the interrupt mask register, simply by writing this bit back into the interrupt request register. This can be done most easily by reading the value and immediately writing it back. For example:

```
LDA IRR
STA IRR
```

Now the bit pattern is in the accumulator and the individual bits can be tested by masking. This is always necessary whenever several interrupt sources are active, such as the normal system interrupt through the timer and an additional interrupt via the video controller. Because both interrupts must go over the same vector, we must first determine in our interrupt service routine what source generated the request and then branch accordingly. An example will make all of this quite clear if it sounds a bit confusing at the moment.

We want to use the raster interrupt in order to display 16 sprites on the screen at the same time. Since the video controller can only display 8 sprites at a time, we must display each set of 8 sprites in succession.

The whole thing functions as follows:

Eight sprites are to be displayed in the upper half of the screen. If the video controller has displayed the upper half, we generate an interrupt. In the interrupt routine we set the parameters for the sprites which are to be displayed in the lower half of the screen. At the same time, we must prepare the next raster interrupt for the end of the screen so that we can again switch back to the upper 8 sprites.

```
100:   033C                     .OPT P1
110:                 ;
120:                 ;RASTER INTERRUPT
130:                 ;
140:   D000          VIC      =    $D000    ;VIDEO CONTROLLER
160:   D001          SPRITEY  =    VIC+1    ;SPRITE Y-COORD-
                                             INATE
165:   D012          RASTER   =    VIC+18   ;RASTER LINE
170:   D019          IRR      =    VIC+25   ;INTERRUPT REQUEST
                                             REGISTER
180:   D01A          IMR      =    VIC+26   ;INTERRUPT MASK
                                             REGISTER
190:   0064          LINE1    =    100      ;FIRST LINE
200:   00C8          LINE2    =    200      ;SECOND LINE
202:   005A          YCOORD1  =    90       ;FIRST Y-COORD-
                                             INATE
203:   00AA          YCOORD2  =    170      ;SECOND Y-COORD-
```

```
                                                INATE
210:                    ;
220:     0314           IRQVEC     =    $314
230:     EA31           IRQOLD     =    $EA31
240:                    ;
300:     033C                      *=   828
310:     033C 78        INIT       SEI
320:     033D A9 64                LDA  #LINE1   ;FIRST INTERRUPT
330:     033F 8D 12 D0             STA  RASTER   ;AT LINE 100
340:     0342 AD 11 D0             LDA  RASTER-1
350:     0345 29 7F                AND  #%01111111  ;ERASE HIGH
                                                    BIT
360:     0347 8D 11 D0             STA  RASTER-1
370:     034A A9 81                LDA  #%10000001  ;INTERRUPT BY
380:     034C 8D 1A D0             STA  IMR      ;RASTER LINE
390:     034F A9 5B                LDA  #<TESTIRQ
400:     0351 A0 03                LDY  #>TESTIRQ
410:     0353 8D 14 03             STA  IRQVEC   ;VECTOR TO NEW
420:     0356 8C 15 03             STY  IRQVEC+1  ;ROUTINE
430:     0359 58                   CLI
440:     035A 60                   RTS
450:                    ;
460:     035B AD 19 D0 TESTIRQ     LDA  IRR      ;READ REGISTER
470:     035E 8D 19 D0             STA  IRR      ;AND ERASE
480:     0361 29 01                AND  #%1      ;IRQ BY RASTER
                                                 LINE
490:     0363 D0 03                BNE  OK       ;YES
500:     0365 4C 31 EA             JMP  IRQOLD   ;NORMAL IRQ
510:                    ;
520:     0368 AD 12 D0 OK          LDA  RASTER   ;CURRENT LINE
530:     036B C9 C8                CMP  #LINE2   ;>= SECOND LINE?
540:     036D B0 16                BCS  SECOND   ;YES
545:                    ;
```

```
550:    036F A0 C8                LDY  #LINE2  ;NEXT IRQ AT 2ND
                                                LINE
555:    0371 A9 AA                LDA  #YCOORD2  ;NEW SPRITE
                                                  COORDINATE
560:    0373 8C 12 D0 BACK        STY  RASTER  ;SET RASTER LINE
570:    0376 A2 0E                LDX  #14
590:    0378 9D 01 D0 LOOP1       STA  SPRITEY,X  ;SPRITE COORD-
                                                   INATES
600:    037B CA                   DEX          ;CHANGE
610:    037C CA                   DEX
620:    037D 10 F9                BPL  LOOP1
630:                    ;
640:    037F 68                   PLA          ;GET REGISTERS
                                                BACK
650:    0380 A8                   TAY
660:    0381 68                   PLA
670:    0382 AA                   TAX
680:    0383 68                   PLA
690:    0384 40                   RTI
700:                    ;
710:    0385 A0 64      SECOND    LDY  #LINE1  ;PARAMETERS FOR
                                                FIRST LINE
720:    0387 A9 5A                LDA  #YCOORD1
730:    0389 4C 73 03             JMP  BACK
]033C-038C
NO ERRORS
```

In order to test our routine, you can activate 8 sprites with the following program. When you then start the interrupt routine with SYS 828, 16 sprites suddenly appear on the screen. Eight are at the y-coordinate 90 and the other 8 at the y-coordinate 170. Each time the upper 8 sprites are displayed we change the sprite parameters in the interrupt routine so that the video controller can display the same sprites again in the lower half of the screen.

```
100 FOR I=0 TO 7:POKE 2040+I,12:NEXT
110 V=53248
120 POKE V+21,255
130 FOR I=0 TO 7:POKEV+2*I,(I+1)*30:POKEV+2*I+1,70:NEXT
140 FOR I=0 TO 7:POKEV+39+I,1:NEXT
```

In addition to the sprite coordinates, you can change all of the other sprite parameters as well, such as the color or size. You can also change the sprite pointers so that other sprite patterns can be displayed, even multicolor.

You can do more than display 16 sprites. If you change the display mode in the raster interrupt routine, you can display a split screen. The top half could display high-resolution graphics while text appears in the lower half. If you place the line number at which a raster interrupt is to be generated into a specific memory location, you can even continually change it from BASIC with a POKE loop so that the border changes. Superimposed effects can also be achieved in this manner. As you can see, there are many possibilities here also.

2.5 CIA 6526 Interrupts

Now that we are acquainted with the interrupts generated by the video controller, we want to look at the CIA 6526, which has very diverse interrupt sources.

The CIA 6526 is a universal input/output interface chip with two parallel 8-bit ports, a serial shift register, two 16-bit timers, a real-time clock as well as several handshake lines.

The two parallel 8-bit ports serve to input and output data. Of the total of four ports contained in the two CIAs, three are used by the system; the two ports of CIA 1 are used for reading the keyboard and joysticks. Port A of CIA 2 yields the 16K address selection for the video controller (bit 0 and 1); bit 2 is free, while bits 3 to 7 are used for the serial bus. Port B of CIA 2 is available to the user through the user port, provided you have not inserted an RS-232 cartridge in the user port.

The timers are used as follows by the operating system:

CIA 1	Timer A	60 Hz system interrupt
	Timer B	serial bus (time-out)
		read and write datasette
CIA 2	Timer A	send RS 232
	Timer B	receive RS 232

If you want to use the timers for your own purposes, you can use the CIA 2. CIA 2 does not generate an IRQ, however, but an NMI. If you do not want to use the serial bus at the same time as your routine, you can use timer B in CIA 1 and thereby generate an IRQ. In special cases you can even dispense with the system interrupt and use timer A.

The real-time clocks are not used by the operating system; there are therefore two of them at your disposal. You can generate either an IRQ (CIA 1) or an NMI (CIA 2) with the alarm time.

The serial shift registers can also be used freely. The line FLAG which serves as a handshake input, sets the corresponding bit in the interrupt control register of CIA 2 on a trailing edge.

The input/output and handshake lines are used primarily for connecting custom peripheral devices. Interrupt programming is often required in such applications. We will later describe interfacing a printer to the user port as an example; the primary aim is to include the routine in the operating system so that the device can be addressed by the usual BASIC commands OPEN, PRINT#, etc.

The next example uses the real-time clock to make an alarm clock. We will use CIA 2 which will generate an NMI when the alarm time is reached.

```
100:   033C                      .OPT P1
110:               ;
120:               ;ALARM WITH REAL-TIME CLOCK IN CIA2
130:               ;
140:   DD00        CIA2     =    $DD00    ;BASE ADDRESS CIA
150:   DD08        TOD10    =    CIA2+8   ;TENTHS OF A
                                           SECOND
```

```
160:   DD09           TODSEC   =    CIA2+9   ;SECONDS
170:   DD0A           TODMIN   =    CIA2+10  ;MINUTES
180:   DD0B           TODSTD   =    CIA2+11  ;HOURS
190:                  ;
200:   DD0D           ICR      =    CIA2+13  ;INTERRUPT CON-
                                              TROL REGISTER
210:   DD0E           CRA      =    CIA2+14  ;CONTROL REG-
                                              ISTER A
220:   DD0F           CRB      =    CIA2+15  ;CONTROL REG-
                                              ISTER B
230:   D020           BORDER   =    $D020    ;BORDER COLOR
240:   0002           RED      =    2
250:                  ;
260:   0318           NMI      =    $318     ;NMI-VECTOR
270:   FE56           CONTNMI  =    $FE56    ;OLD NMI
280:                  ;
290:                  ;TIME 12H 00' 00.0"
300:   0000           TENTHS   =    0
310:   0000           SECONDS  =    $00
320:   0000           MINUTES  =    $00
330:   0001           HOURS    =    $01
340:                  ;
350:                  ;ALARM TIME 12H 00' 05.0"
360:   0000           ALARM.10 =    0
370:   0005           ALARM.SC =    $05
380:   0000           ALARM.MN =    $00
390:   0001           ALARM.HR =    $01
400:                  ;
410:   033C                    *=   828
420:                  ;
430:                  ;SET CLOCK TIME
440:   033C AD 0E DD           LDA  CRA
450:   033F 09 00              ORA  #$00     ;CLOCK TIME 60 HZ
```

```
460:   0341 8D 0E DD            STA  CRA
470:                     ;
480:   0344 AD 0F DD            LDA  CRB
490:   0347 29 7F               AND  #$7F      ;SET CLOCK TIME
500:   0349 8D 0F DD            STA  CRB
510:                     ;
520:   034C A5 01               LDA  HOURS
530:   034E 8D 0B DD            STA  TODSTD
540:   0351 A9 00               LDA  #MINUTES
550:   0353 8D 0A DD            STA  TODMIN
560:   0356 A9 00               LDA  #SECONDS
570:   0358 8D 09 DD            STA  TODSEC
580:   035B A9 00               LDA  #TENTHS
590:   035D 8D 08 DD            STA  TOD10
600:                     ;
610:   0360 AD 0F DD            LDA  CRB
620:   0363 09 80               ORA  #$80      ;SET ALARM TIME
630:   0365 8D 0F DD            STA  CRB
640:                     ;
650:   0368 A9 01               LDA  #ALARM.HR
660:   036A 8D 0B DD            STA  TODSTD
670:   036D A9 00               LDA  #ALARM.MN
680:   036F 8D 0A DD            STA  TODMIN
690:   0372 A9 05               LDA  #ALARM.SC
700:   0374 8D 09 DD            STA  TODSEC
710:   0377 A9 00               LDA  #ALARM.10
720:   0379 8D 08 DD            STA  TOD10
730:                     ;
740:   037C A9 84               LDA  #%10000100  ;ALARM
750:   037E 8D 0D DD            STA  ICR       ;FREE NMI
760:                     ;
770:   0381 A9 8C               LDA  #<TEST
780:   0383 A0 03               LDY  #>TEST
```

```
790:    0385 8D 18 03           STA  NMI      ;NEW NMI VECTOR
800:    0388 8C 19 03           STY  NMI+1
810:    038B 60                 RTS
820:                   ;
830:    038C 48        TEST     PHA
840:    038D 8A                 TXA
850:    038E 48                 PHA           ;SAVE REGISTERS
860:    038F 98                 TYA
870:    0390 48                 PHA
880:    0391 AC 0D DD           LDY  ICR
890:    0394 98                 TYA
900:    0395 29 04              AND  #%100    ;ALARM BIT SET?
910:    0397 D0 03              BNE  ALARM    ;YES
920:    0399 4C 56 FE           JMP  CONTNMI
930:                   ;
940:    039C A9 02     ALARM    LDA  #RED
950:    039E 8D 20 D0           STA  BORDER   ;BORDER COLOR TO
                                               RED
960:                   ;
970:    03A1 68                 PLA
980:    03A2 A8                 TAY
990:    03A3 68                 PLA
1000:   03A4 AA                 TAX
1010:   03A5 68                 PLA
1020:   03A6 40                 RTI
]033C-03A7
NO ERRORS
```

The program first defines the addresses of the real-time clock and the control register in the CIA 2. Then the clock time is set to 12 o'clock and the alarm time to 12 o'clock and 5 seconds. The program first sets the real-time clock to 60 Hz so that the clock runs correctly. Then bit 7 in control register B is cleared in order to inform the CIA that we want to input the clock time, which we proceed to do. Now we set bit 7, program the alarm time, and enable the alarm NMI in the interrupt control register. Bit 2 as well as bit 7 must be set in order to do this. Finally, we must set the NMI vector to our new routine and the initialization is completed.

The actual NMI routine does not have much to do. First the registers are saved on the stack, then the interrupt control register is read and bit 2 checked. If the bit was set, the alarm time was reached. We respond by setting the border color of the screen to red. The registers are restored and control is returned to the interrupted program. If the NMI was not generated by the alarm time, we jump to the NMI routine in the kernal. There a check is made to see if the STOP key was pressed in addition to the RESTORE key which generated the interrupt. If this test is positive, a warm start is executed.

Naturally, you can change the action which occurs when the alarm time is reached. For example, your routine could sound a tone through the sound chip. You should also add an easy way of setting the clock and alarm times. The real-time clock is very accurate over a long period of time because it runs in synchronization with the AC power lines.

2.6 Using the timer

Each CIA contains two 16-bit timers. An interrupt can be generated when the timer times out. These timers are used heavily by the operating system and are decremented by one with each system clock pulse. If the value zero is reached, the corresponding bit in the interrupt request register is set and--if the mask in the interrupt control register permits it--an IRQ or NMI is generated. The American version of the Commodore 64 has a clock frequency of approximately 1.02 MHz, resulting in a clock period of about .98 microseconds or close to 1 microsecond. Because the timer can be loaded with a 16-bit value, times up to 65,535 clock periods or approximately 65 milliseconds (about a fiftieth of a second) can be attained. Timer A of CIA 1 is loaded with $4295 or 17045, for example, which corresponds to one sixtieth of a second. European PAL versions have a clock frequency of 985 KHz, resulting in a clock period of 1.015 microseconds. The timer is loaded with the value $4025 or 16421, which corresponds to one sixtieth of a second at the slower speed.

There are various operating methods for using the timer such as the "one shot" and "continuous" modes. In the one-shot mode the timer counts down only once from the initial value to zero and then stops. In the continuous mode, the timer is automatically reloaded with the starting value and started again when it times out. In addition to generating an interrupt, the timer can also generate a pulse on the user port after time-out. This could be used as the clock signal for a peripheral device. In addition, the timers can be used as counters. In this mode, the system clock does not do the decrementing. Instead, an external signal causes the timer to be decremented. One can also couple the timers. One timer counts the number of times the other reaches zero. This allows the two to be used as a 32-bit timer, so that times up to 2^32 clock cycles or about 4,360 seconds (1 hour and 12 minutes) can be recorded.

At the close of our chapter on interrupt programming, we want to write a machine language program that allows us to control BASIC subroutines with interrupts. We will learn something about the use of the timers as well as the operation of the BASIC interpreter.

We will introduce a new BASIC command which allows us to execute a normal BASIC subroutine when a certain time has elapsed. First a bit of background information.

The BASIC interpreter uses a main loop when executing a BASIC program to analyze and execute each statement. After each statement, a check is made to see if the STOP key was pressed. If it was pressed, the main loop is exited and control returns to the direct mode. The reading of the STOP key occurs via a jump vector. We can change this vector so that it points to a new routine. In this new routine we can check to see if the condition for executing our interrupt program has been met. In other words, to see if the timer has timed out. In order to recognize this, an interrupt routine sets a flag based on the state of the timer, which can then be read by the previous routine.

The new BASIC command specifies which BASIC routine is to be executed after an interrupt. An additional parameter specifies the time at which the interrupt should be generated. The command looks like this.

!GOSUB 1000,100

The exclamation point is used to differentiate the new command from the normal GOSUB command. The 1000 is the first line number of the subroutine and the 100 is the time at which the interrupt will be generated. The time increments are fiftieths of a second. We load a timer with this value (for one fiftieth of a second). We load the second value (the second parameter of the command) into the next timer, using the two of them together as a 32-bit timer. We can then program times from a fiftieth of second to 65535 fiftieths of a second, which is 0.02 to 1311 seconds (21 minutes and 51 seconds).

Our program consists of three routines in addition to the initialization. The first modifies the BASIC interpreter so that it understands our new command. The second routine checks (after each statement) to see if the time-out flag is set and if so, branches to the BASIC subroutine. The third routine is the interrupt (actually NMI) routine which sets the flag for the second routine after the timer times out.

```
100:   CC00                      .OPT P1
110:   CC00                      .SYM 2
130:               ;
140:               ;INTERRUPT ROUTINE FOR BASIC
150:               ;
160:   0308        EXEC     =     $308     ;EXECUTE VECTOR
                                            FOR STATEMENT
170:   0318        NMI      =     $318     ;NMI VECTOR
180:   0328        STOP     =     $328     ;STOP VECTOR
190:               ;
```

```
200:   DD00        CIA2      =   $DD00
210:   DD04        TIMERA    =   CIA2+4    ;TIMER A
220:   DD06        TIMERB    =   CIA2+6    ;TIMER B
230:   DD0D        ICR       =   CIA2+13    ;INTERRUPT CON-
                                             TROL REGISTER
240:   DD0E        CRA       =   CIA2+14    ;CONTROL REG-
                                             ISTER A
250:   DD0F        CRB       =   CIA2+15    ;CONTROL REG-
                                             ISTER B
260:               ;
270:   FE56        CONTNMI   =   $FE56     ;CONTINUE OLD NMI
280:               ;
290:   4FB0        TIME      =   20400     ;=20 MILLISECONDS
300:   0014        LO        =   $14       ;LINE NUMBER LO
310:   0015        HI        =   LO+1
320:   005F        LINEADDR  =   $5F       ;ADDRESS OF BASIC
                                            LINE
330:   0039        LINENO    =   $39       ;RUNNING LINE
                                            NUMBER
340:   0073        CHRGET    =   $73
350:   0079        CHRGOT    =   CHRGET+6
360:   007A        TXTPTR    =   CHRGOT+1
370:   008D        GOSUB     =   $8D       ;GOSUB TOKEN
380:   AF08        SYNTAX    =   $AF08     ;SYNTAX ERROR
390:   A8E3        UNDEFD    =   $A8E3     ;UNDEF'D STATEMENT
                                            ERROR
400:   B248        ILLQUAN   =   $B248     ;ILLEGAL QUANTITY
                                            ERROR
410:   A7AE        INTER     =   $A7AE     ;INTERPRETER LOOP
420:   A96B        GETLIN    =   $A96B     ;GET LINE NUMBER
430:   A613        GETADDR   =   $A613     ;SEARCH LINE
440:   AEFD        CHKCOM    =   $AEFD     ;TEST COMMA
450:   A7E7        EXECOLD   =   $A7E7     ;EXECUTE STATEMENT
```

```
460:  AD8A              FRMNUM    =    $AD8A    ;GET NUMERICAL
                                                 VALUE
470:  B7F7              INTEGER   =    $B7F7    ;AND CONVERT TO
                                                 INTEGER
480:  A3FB              TESTSTACK=     $A3FB    ;CHECK FOR SPACE
                                                 IN STACK
490:  F6ED              TESTOLD   =    $F6ED    ;CHECK STOP KEY
500:  FE47              NMIOLD    =    $FE47    ;OLD NMI VECTOR
510:                    ;
520:  CC00                        *=   $CC00
530:  CC00 A9 10        INIT      LDA  #<TESTSTAT
540:  CC02 A0 CC                  LDY  #>TESTSTAT
550:  CC04 8D 08 03               STA  EXEC     ;ROUTINE FOR
                                                 DECODING
560:  CC07 8C 09 03               STY  EXEC+1   ;OFF '!'
570:  CC0A A9 00                  LDA  #0
580:  CC0C 8D F7 CC               STA  FLAG     ;ERASE FLAG
590:  CC0F 60                     RTS
600:                    ;
610:  CC10 20 73 00 TESTSTAT JSR  CHRGET   ;GET NEXT CHAR-
                                                 ACTER
620:  CC13 C9 21                  CMP  #"!"
630:  CC15 F0 06                  BEQ  TSTGOSUB
640:  CC17 20 79 00               JSR  CHRGOT   ;REPLACE FLAGS
650:  CC1A 4C E7 A7               JMP  EXECOLD   ;AND CONTINUE AS
                                                  NORMAL
660:                    ;
670:  CC1D 20 73 00 TSTGOSUB JSR  CHRGET   ;NEXT CHARACTER
680:  CC20 C9 8D                  CMP  #GOSUB   ;GOSUB CODE?
690:  CC22 F0 03                  BEQ  OK       ;YES
700:  CC24 4C 08 AF               JMP  SYNTAX   ;SYNTAX ERROR
710:  CC27 20 73 00 OK            JSR  CHRGET   ;NEXT CHARACTER
720:  CC2A F0 68                  BEQ  IRQOFF   ;LINE END, THEN
```

```
                                                SWITCH IRQ ADD
CC2C 20 6B A9              JSR  GETLIN   ;GET LINE NUMBER
CC2F 20 13 A6              JSR  GETADDR  ;GET LINE ADDRESS
CC32 B0 03                 BCS  FOUND    ;FOUND?
CC34 4C E3 A8              JMP  UNDEFD   ;NO, UNDEF'D
                                          STATEMENT ERROR
CC37 A5 5F     FOUND       LDA  LINEADDR  ;LINE ADDRESS
CC39 E9 01                 SBC  #1       ;MINUS 1
CC3B 8D F8 CC              STA  LINESTR  ;SAVE
CC3E A5 60                 LDA  LINEADDR+1
CC40 E9 00                 SBC  #0       ;HIGH BYTE
CC42 8D F9 CC              STA  LINESTR+1
CC45 20 FD AE              JSR  CHKCOM   ;CHECK FOR COMMA
CC48 20 8A AD              JSR  FRMNUM   ;NEXT VALUE
CC4B 20 F7 B7              JSR  INTEGER  ;CONVERT TO
                                           INTEGER
CC4E A5 14                 LDA  LO
CC50 05 15                 ORA  HI       ;LOW AND HIGH BYTE
                                          ZERO?
CC52 D0 03                 BNE  OK1      ;NO
CC54 4C 48 B2              JMP  ILLQUAN  ;ILLEGAL QUANTITY
                                           ERROR
CC57 A5 15     OK1         LDA  HI
CC59 8D 07 DD              STA  TIMERB+1
CC5C A5 14                 LDA  LO       ;LOAD VALUE INTO
                                          TIMER B
CC5E 8D 06 DD              STA  TIMERB
CC61 A9 4F                 LDA  #>TIME   ;LOAD TIMER A
CC63 8D 05 DD              STA  TIMERA+1
CC66 A9 B0                 LDA  #<TIME   ;WITH 20MS
CC68 8D 04 DD              STA  TIMERA
CC6B A9 11                 LDA  #%00010001  ;START TIMER A
CC6D 8D 0E DD              STA  CRA
```

```
1000:  CC70 A9 51               LDA  #%01010001  ;START TIMER B
1010:  CC72 8D 0F DD            STA  CRB        ;TRIGGERED BY
                                                 TIMER A
1020:  CC75 AD 0D DD            LDA  ICR        ;ERASE ICR
1030:  CC78 A9 82               LDA  #%10000010  ;NMI FOR
                                                     TIMER B
1040:  CC7A 8D 0D DD            STA  ICR        ;FREE
1050:  CC7D A9 C9               LDA  #<TESTTIME
1060:  CC7F A0 CC               LDY  #> TESTTIME
1070:  CC81 8D 28 03            STA  STOP       ;GET STOP VECTOR
1080:  CC84 8C 29 03            STY  STOP+1
1090:  CC87 A9 B0               LDA  #<NMIROUT
1100:  CC89 A0 CC               LDY  #>NMIROUT
1110:  CC8B 8D 18 03            STA  NMI        ;SET NMI VECTOR
1120:  CC8E 8C 19 03            STY  NMI+1
1130:  CC91 4C AE A7            JMP  INTER      ;TO INTERPRETER
                                                 LOOP
1140:                  ;
1150:  CC94 A9 7F      IRQOFF   LDA  #%01111111
1160:  CC96 8D 0D DD            STA  ICR        ;ALL INTERRUPTS
                                                 OFF
1170:  CC99 A9 ED               LDA  #<TESTOLD
1180:  CC9B A0 F6               LDY  #>TESTOLD
1190:  CC9D 8D 28 03            STA  STOP       ;STOP VECTOR TO
                                                 OLD VALUE
1200:  CCA0 8C 29 03            STY  STOP+1
1210:  CCA3 A9 47               LDA  #<NMIOLD
1220:  CCA5 A0 FE               LDY  #>NMIOLD
1230:  CCA7 8D 18 03            STA  NMI        ;NMI VALUE TO OLD
                                                 VECTOR
1240:  CCAA 8C 19 03            STY  NMI+1
1250:  CCAD 4C AE A7            JMP  INTER      ;TO INTERPRETER
                                                 LOOP
```

```
1260:                   ;
1270:  CCB0 48          NMIROUT  PHA
1280:  CCB1 8A                   TXA
1290:  CCB2 48                   PHA
1300:  CCB3 98                   TYA
1310:  CCB4 48                   PHA
1320:  CCB5 AC 0D DD             LDY  ICR
1330:  CCB8 98                   TYA
1340:  CCB9 29 02                AND  #%10     ;TIMER B TIMED
                                                OUT?
1350:  CCBB D0 03                BNE  TIMEOUT  ;YES
1360:  CCBD 4C 56 FE             JMP  CONTNMI  ;OTHERWISE NORMAL
                                                 NMI
1370:                   ;
1380:  CCC0 EE F7 CC TIMEOUT  INC  FLAG     ;SET FLAG
1390:  CCC3 68                   PLA
1400:  CCC4 A8                   TAY
1410:  CCC5 68                   PLA
1420:  CCC6 AA                   TAX
1430:  CCC7 68                   PLA
1440:  CCC8 40                   RTI
1450:                   ;
1460:  CCC9 AD F7 CC TESTTIME LDA  FLAG     ;FLAG SET?
1470:  CCCC D0 03                BNE  TIMEIRQ  ;YES
1480:  CCCE 4C ED F6             JMP  TESTOLD
1490:                   ;
1500:  CCD1 CE F7 CC TIMEIRQ  DEC  FLAG     ;ERASE FLAG AGAIN
1510:  CCD4 68                   PLA
1520:  CCD5 68                   PLA            ;RETURN ADDRESS
                                                FROM STACK
1530:  CCD6 A9 03                LDA  #3
1540:  CCD8 20 FB A3             JSR  TESTSTACK  ;STILL ENOUGH
                                                   STACK SPACE
```

```
1550:   CCDB A5 7B                LDA  TXTPTR+1
1560:   CCDD 48                   PHA          ;CHRGET POINTER
                                                TO STACK
1570:   CCDE A5 7A                LDA  TXTPTR
1580:   CCE0 48                   PHA
1590:   CCE1 A5 3A                LDA  LINENO+1
1600:   CCE3 48                   PHA          ;ACTUAL LINE
                                                NUMBER ON STACK
1610:   CCE4 A5 39                LDA  LINENO
1620:   CCE6 48                   PHA
1630:   CCE7 A9 8D                LDA  #GOSUB
1640:   CCE9 48                   PHA          ;GOSUB CODE ON
                                                STACK
1650:   CCEA AD F8 CC             LDA  LINESTR
1660:   CCED 85 7A                STA  TXTPTR  ;ADDRESS OF SUB-
                                                ROUTINE
1670:   CCEF AD F9 CC             LDA  LINESTR+1
1680:   CCF2 85 7B                STA  TXTPTR+1
1690:   CCF4 4C B1 A7             JMP  INTER+3  ;TO INTERPRETER
                                                 LOOP
1700:                   ;
1710:   CCF7            FLAG      *=   *+1
1720:   CCF8            LINESTR   *=   *+2
]CC00-CCFA
NO ERRORS
```

```
SYMBOL-TABLE:
LINESTR   CCF8   FLAG       CCF7
TIMEIRQ   CCD1   TESTTIME   CCC9
TIMEOUT   CCC0   NMIROUT    CCB0
IRQOFF    CC94   OK1        CC57
FOUND     CC37   OK         CC27
TSTGOSUB  CC1D   TESTSTAT   CC10
```

INIT	CC00	NMIOLD	FE47
TESTOLD	F6ED	TESTSTAC	A3FB
INTEGER	B7F7	FRMNUM	AD8A
EXECOLD	A7E7	CHKCOM	AEFD
GETADDR	A613	GETLIN	A96B
INTER	A7AE	ILLQUAN	B248
UNDEFD	A8E3	SYNTAX	AF08
GOSUB	008D	TXTPTR	007A
CHRGOT	0079	CHRGET	0073
LINENO	0039	LINEADDR	005F
HI	0015	LO	0014
TIME	4FB0	CONTNMI	FE56
CRB	DD0F	CRA	DD0E
ICR	DD0D	TIMERB	DD06
TIMERA	DD04	CIA2	DD00
STOP	0328	NMI	0318
EXEC	0308		

45 SYMBOLS DEFINED

Before we come to the detailed description of the program, here is a small demonstration program.

```
100 SYS 52224 : REM INITIALIZE EXPANSION
110 !GOSUB 200,50
120 I=I+1 : PRINT I : IF I<100 GOTO 120
130 !GOSUB
140 END
200 J=J+1 : PRINT "IRQ CALL #" J : RETURN
```

When you start this program with RUN, the new command is added by the SYS in line 100. Line 110 defines the subroutine at line 200 as the interrupt program, which is executed every second (50 fiftieths). The actual main program is in line 120 and outputs the number from 1 to 100. When this loop is ended, the interrupt routine is switched off by !GOSUB without any parameters and the program ends. The interrupt routine is at line 200. It displays a running count of the number of calls before returning to the main program with RETURN.

If you run the program, numbers from 1 to 100 will be printed, but the output will be interrupted five times with the message

IRQ CALL # 1

through

IRQ CALL # 5

If you change the second parameter in line 110, you can set the frequency at which the subroutine is called. Values from 1 to 65535 are allowed. The smaller the value is, the more often the interrupt routine will be called. The time required to execute the BASIC interrupt routine may not be longer than the time between calls, otherwise the interrupt routine will interrupt itself and the BASIC stack will overflow. For example, if your replace line 110 with

```
110 !GOSUB 200,1
```

you will receive the following output:

```
1
IRQ CALL # 1
IRQ CALL # 2
```

```
...
IRQ CALL # 22
IRQ CALL # 23

?OUT OF MEMORY  ERROR IN 200
```

Now to description of the machine language program.

Constants are defined in lines 100 to 500. These concern the NMI and BASIC vectors. Then follow the registers in the CIA 2 which are necessary for the timer interrupt. Line 290 defines our time increment. After this are BASIC addresses from the zero page as well as error messages and ROM addresses used by the BASIC interpreter. The initialization is performed in lines 520 to 590. Here the vector which points to the routine for decoding and executing a BASIC statement is redirected to our own routine. This routine gets the next character from the BASIC text and compares it with the exclamation point. If this character is not found, the original values of the flags are restored with the CHRGOT routine and a jump is made to the point in the interpreter where statements are normally processed. If, on the other hand, an exclamation point is found, we get the next character and check if it is the code for GOSUB. If not, then we output "SYNTAX ERROR." If so, then it is our new command. The next character is fetched. If it is the end of the line, a branch is made to the routine which disables the interrupt and resets the vectors to their original values. Otherwise the line number is determined and its address obtained. After a check is made to see if this line really exists (signaled by a set carry flag), the line address is decremented by one and saved. Now a test can be made for the comma and the second parameter fetched. The second parameter determines the duration between interrupts. If it is not zero, timer B is loaded with it. Timer A is loaded with the value for a fiftieth of a second and both timers are started. Timer B is programmed such that it is decremented each time timer A reaches zero (times out). The NMI for timer B is then enabled by writing the corresponding bit pattern into the control register. Finally, the STOP and NMI vectors are set to the new routines before we jump back to the interpreter loop.

From line 1150 to 1250 you find the routine which turns the interrupt off following a !GOSUB command without parameters. It also sets the vectors back to the original values. The actual NMI routine is perfomed by lines 1270 to 1390. The registers are first saved and the status of timer B is tested by reading the interrupt control register, to see if the timer generated the NMI. If this was the case, a flag is set and the NMI routine exited. Otherwise execution branches to the normal NMI routine.

The most important subroutine, called by the BASIC interpreter after each statement, is found at line 1410. Here a check is made to see if the appropriate flag from the NMI is set indicating that the time is up. If the test is negative, a branch is made to the normal routine which checks the STOP key. If the time was up, the flag is cleared and the actual return address is pulled from the stack. The BASIC GOSUB command is then imitated. After the program determines that there is enough room left on the stack, the pointer to the BASIC text as well as the current line number are saved on the stack. In order to distinguish this from a FOR-NEXT loop which also places its parameters on the stack, the GOSUB code is pushed onto the stack. Next the address of the subroutine is determined as saved by the definition, loaded into the BASIC text pointer and a branch is again made to the interpreter loop. The BASIC interpreter executes the subroutine and can correctly return to the interrupted program when it encounters the RETURN command.

The program ends with the definition of two variables. The .SYM pseudo-op in line 120 produces the symbol table shown at the end of the listing which includes all of the symbols used together with their values.

This new command offers you possibilities in BASIC which could previously be attained only in machine language. You can now execute time-controlled subroutines in BASIC with a time span from 20 milliseconds to 21 minutes to choose from. This is one example of interrupt control from within BASIC. To illustrate the routine, here is a program which flashes the screen by exchanging the background and border colors.

```
100 SYS 52224
110 F1 = 53280 : F2 = F1 + 1
120 !GOSUB 1000,30
130 FOR I=1 TO 1000 : PRINT I, : NEXT
140 !GOSUB : END
1000 A=PEEK(F1) : POKE F1,PEEK(F2) : POKE F2,A : RETURN
```

The BASIC interrupt routine should always be deactivated with !GOSUB before the end of the program. If you later try to list or save a program with the interrupt routine active, it will interrupt this process because these routines check for the STOP key.

The next example program displays the Commodore 64's character set in normal and reverse and then switches, on interrupt, between the standard text representation and the extended color mode. This is done by setting bit 6 in register 17 of the video controller. In this mode only 64 characters instead of 256 can be displayed. The upper two bits of the screen memory now serve to select one of four different background colors for each character. These colors are placed in registers 33 to 36 of the video controller (addresses 53281 to 53284).

```
100 SYS 52224
110 !GOSUB 170,25
120 X=18
130 PRINT CHR$(X);:X=X+128 AND 255
140 FOR I=32 TO 127:PRINT CHR$(I);:NEXT
150 FOR I=160 TO 255:PRINT CHR$(I);:NEXT
160 PRINT:GOTO130
170 A=PEEK(53248+17):POKE53248+17,(AOR64)ANDNOT(AAND64)
180 RETURN
```

SECTION 3 Beyond BASIC

3.1 Kernal and BASIC extensions

One advantage that the Commodore 64 has over its "big" brothers, the CBM 8032 and 8096 is that the BASIC interpreter and operating system kernal can be easily "expanded" with your own customized routines.

By expand we mean that we can extend the capabilities by adding new or enhanced commands to BASIC. It is no longer necessary to access each new command with PEEK, POKE, or SYS. There are two ways to do this.

Because the entire address space of the Commodore 64 of 64K is equipped with RAM, you can easily make changes in the BASIC and operating system by copying the BASIC interpreter and/or the kernal ROM into the RAM lying at the same address. Then you can make the desired changes and "switch on" this RAM version of BASIC by means of the processor port at address 1. This method has both advantages and disadvantages compared to the method described later.

The advantage of this method is that you have complete freedom in making changes. This freedom is so extensive that a completely different language can be used in place of BASIC, or a completely new operating system can be constructed. This RAM area is otherwise often used for such things as graphics storage. The disadvantage of this method lies in that this RAM area is no longer available for other purposes. A variant of this method is the use of one or two EPROMs in the address range from $8000 to $9FFF or from

$8000 to $BFFF which contains a BASIC extension, another language, or a user-specified program. A cartridge in the cartridge slot is necessary for this, however.

A second method does not require additional ROM but rather uses entry points in the system software in order to modify the most important functions. These key positions are accessed via so-called jump vectors which can be changed by the user. An indirect jump instruction used at this point. For example

```
JMP (VECTOR)
```

The low and high bytes of the actual jump address are stored at the address vector. These vectors are initialized when the computer is turned on and usually point directly behind the indirect jump command in the BASIC interpreter. If we want to change a certain function, we write our own routine and change the appropriate jump vector so that it points to our new routine. The principle is similar to that which we learned for interrupt vectors.

The following table gives information concerning what bit pattern must be written to address 1 in order to get the appropriate memory configuration when using the "RAM method":

Bit 2	1	0	dec	\$A000 - \$BFFF	\$D000 - \$DFFF	\$E000 - \$FFFF
1	1	1	7	BASIC	I/O	KERNAL
1	1	0	6	RAM	I/O	KERNAL
1	0	1	5	RAM	I/O	RAM
1	0	0	4	RAM	RAM	RAM
0	1	1	3	BASIC	CHAR GEN	KERNAL
0	1	0	2	RAM	CHAR GEN	KERNAL
0	0	1	1	RAM	CHAR GEN	RAM
0	0	0	0	RAM	RAM	RAM

This table contains all possible combinations for the memory configuration. Combinations 4 and 0 have the same result; the complete address space is switched to RAM. You can see from the table that BASIC can be exchanged for RAM independently, but the kernal ROM must be switch out together with the BASIC ROM. This should be noted if the kernal is to be replaced. The address area at \$D000-\$DFFF has three functions: it is the I/O area, which is divided as follows:

\$D000 - \$D3FF	VIC 6567
\$D400 - \$D7FF	SID 6581
\$D800 - \$DBFF	color RAM
\$DC00 - \$DCFF	CIA 1 6526
\$DD00 - \$DDFF	CIA 2 6526
\$DE00 - \$DEFF	I/O 1 for expansion
\$DF00 - \$DFFF	I/O 2 for expansion

In addition, the character generator can be addressed at this address. Third, this area is allocated with RAM which can only be addressed when the entire memory is switched to RAM.

3.2 The BASIC vectors

The BASIC interpreter has six vectors which make it possible to add new routines. These vectors are placed in page 3 and have the following use:

Vector	Address	Significance
$0300/$0301	$E38B	BASIC warm start and error entry point
$0302/$0303	$A483	input delay loop
$0304/$0305	$A57C	conversion to interpreter code
$0306/$0307	$A71A	convert interpreter code to text
$0308/$0309	$A7E4	execute BASIC command
$030A/$030B	$AE86	evaluate BASIC expression

With the help of these 6 vectors you have an easily accessible way of changing the BASIC interpreter. We will become acquainted with the significance of each vector and use them for extensions and enhancements.

In order to draw the greatest usefulness from this section, you may want to consult the ROM listing of the '64 found in The Anatomy of the Commodore 64 as we go along. This allows you to trace exactly what happens in the BASIC interpreter.

The warm start and error vector $300/$301

This vector is used when the END of the program or an error is encountered. If an error occurs, the X register contains the error number. These numbers range from 1 to 29 and have the following meaning:

No.	Error message
1	TOO MANY FILES
2	FILE OPEN
3	FILE NOT OPEN
4	FILE NOT FOUND
5	DEVICE NOT PRESENT
6	NOT INPUT FILE
7	NOT OUTPUT FILE
8	MISSING FILENAME
9	ILLEGAL DEVICE NUMBER
10	NEXT WITHOUT FOR
11	SYNTAX
12	RETURN WITHOUT GOSUB
13	OUT OF DATA
14	ILLEGAL QUANTITY
15	OVERFLOW
16	OUT OF MEMORY
17	UNDEF'D STATEMENT
18	BAD SUBSCRIPT
19	REDIM'D ARRAY
20	DIVISION BY ZERO
21	ILLEGAL DIRECT
22	TYPE MISMATCH
23	STRING TOO LONG
24	FILE DATA
25	FORMULA TOO COMPLEX
26	CAN'T CONTINUE
27	UNDEF'D FUNCTION
28	VERIFY
29	LOAD

Error messages from 1 to 9 are input/output related errors and are issued by the operating system (kernal). Errors 10 to 29 are generated by the BASIC interpreter. If an error is recognized by the BASIC interpreter, the X register is loaded with the error number and a jump is made to address $A437 by way of the indirect jump JMP ($0300). If the program is ended with END as normal, however, the X register is loaded with a negative value ($80) in order to distinguish it from an error message. This is checked in the error routine, the error output bypassed, the message "READY." displayed, and a branch made to the input-wait loop.

We can use the error vector for a variety of purposes. For one, we could change the text of the error messages, or prevent the program from breaking off when an error is discovered, but to branch to a specified BASIC line where the error can be caught or perhaps corrected. Some enhanced versions of BASIC have a command for this purpose such as:

```
ON ERROR GOTO ...
```

Such a command can, for example, be used to catch errors generated by peripheral devices.

The input-wait loop $302/$303

When the computer displays the READY. prompt after END or an error message, it goes to the warm-start vector at $300. Then it jumps to the vector $302/$303. In this routine, the computer waits for the input of a line terminated by the <RETURN> key.

This line is checked to make sure that it is not longer than 88 characters (the length of the BASIC input buffer located from $200 to $258.) If this length is exceeded, the error message "STRING TOO LONG" is displayed. The first character of the inputted line determines how the line will be treated. If the first character is a digit, the interpreter assumes that we want to enter a new BASIC line. In this case, the entire line number is read and a check is made to see if this line exists. If so, the old line is deleted from the program. If nothing follows the line number, then the line is to be simply deleted and a branch is made back to the start of the loop. If additional text follows the line number, this text is converted to interpreter codes, and the program line is inserted into the BASIC text, and a branch is again made to the start of the loop.

If the first character entered was not a digit, the line is interpreted as a BASIC command in the direct mode. The line is converted to interpreter codes and a branch is made to the place in the interpreter where a BASIC command is executed.

We can also use this vector to extend or enhance BASIC. For example, it is possible to take program input from a sequential disk file or from the user port, from another computer connected there, instead of the keyboard. This greatly simplifies the transfer of BASIC programs from other computers. The slow and error-prone typing-in of listings is no longer necessary. With direct coupling of two computers, the sending computer need only list its program over the interface. The RS 232 is best suited for this purpose since most computers have this interface available to them.

An additional application of this vector is the AUTO command. This command eases the input of programs by automatically placing the next line number at the start of the line and positioning the cursor behind it.

Conversion to interpreter codes $304/$305

As you probably know, a program line is not saved as it was entered. Instead, each command word is shortened to a single-byte value called a token. This has two advantages over storing the entire text of that word. First, it saves memory. Instead of 5 bytes for the word "PRINT" only one byte is require for the token. The second advantage is noticeable during program execution. When the BASIC interpreter is executing a program and comes across a token, it can immediately execute the appropriate command. If the command were saved in text (ASCII) form, the complete word would have to be read. Then the interpreter would have to read through its command table and see if the word is present in its table as a command word. The program would take considerably longer to run without tokens. If, on the other hand, the program line is converted to tokens, this conversion is only necessary once and not each time the command is executed.

If we want to convert new commands to tokens, we can change this vector. Our routine must then compare the word read from the input with the table of the new command words. If a new command is found, the command word is replaced by its token in the program or command line.

Conversion of the interpreter code to text (ASCII) $306/$307

This vector performs the opposite task of the one above. If we want to list a program, we must convert the tokens back into text. The token value is used as a pointer in the command word table. This vector is used only by the LIST command. We must change it when we use our own interpreter codes so that the new commands can be listed properly. An additional application is to change the operation of the LIST command. We could for example make a listing more readable by placing a space after each command word, or by indenting loop structures. It is also possible to start a new line for each new statement separated by a colon.

Execute a command $308/$309

This vector is one of the most important. It points to the place in the interpreter where a BASIC command is executed. Normally, this routine gets a character from the BASIC text and check to see if it is a token. If the character is not a token, the interpreter assumes that it is an assignment of the form "A = ..." and branches to the LET command. If it is a token, its value is used as an index in a table containing the BASIC commands. These commands are executed as subroutines and after execution can branch back to the start of the interpreter loop where the next statement can be handled in the same way.

With the help of this vector one can easily add custom BASIC commands to the interpreter. These can be designated by a special character such as an exclamation point (!). We can then check for this character in our routine and execute the new command when found.

If we have added our own tokens for our new commands using the previously described vector $304/$305, a special character is no longer necessary. Instead, we can first check for our new tokens and branch to the original routine for executing commands if the new command is not found.

Evaluate a BASIC expression $30A/$30B

This vector is to a function what the previous vector was to a command. This vector is used when an element of an expression is to be calculated. This element can be a number, a BASIC variable, or a function. If we want to add new functions, we must add them using this vector. Numeric as well as string functions can be handled. You must also modify this vector if you save variables in other forms. This allows such things as hex and binary constants.

```
100:   033C                 .OPT P1
110:               ;
120:               ;INPUT OF HEX AND BINARY NUMBERS
130:               ;
140:   030A        EXPRESSIO=    $30A     ;VECTOR FOR EXP-
                                          RESSION EVALUATION
150:   AE8D        OLDVECT  =    $AE8D    ;OLD ROUTINE
160:               ;
170:   000D        TYP      =    13       ;VARIABLE TYPE
180:   0073        CHRGET   =    $73
190:   0079        CHRGOT   =    CHRGET+6
200:               ;
210:   BD7E        ADDDIGIT =    $BD7E    ;ADD ONE-BYTE
                                           DIGIT TO FAC
220:               ;
230:   005D        FLOAT    =    $5D      ;RANGE FOR FLOAT-
```

```
                                               ING POINT NUMBERS
240:   0061            EXP       =    $61      ;EXPONENT FROM FAC
250:                   ;
260:   B97E            OVERFLOW =    $B97E    ;OVERFLOW ERROR
270:                   ;
280:   033C                      *=   828
290:                   ;
300:   033C A9 47      INIT      LDA  #<TEST
310:   033E A0 03                LDY  #>TEST
320:   0340 8D 0A 03             STA  EXPRESSION  ;SET VECTOR
                                                    TO NEW ROUTINE
330:   0343 8C 0B 03             STY  EXPRESSION+1
340:   0346 60                   RTS
350:                   ;
360:   0347 A9 00      TEST      LDA  #0
370:   0349 85 0D                STA  TYP      ;TYPE FLAG TO
                                                NUMERIC
380:   034B 20 73 00             JSR  CHRGET   ;GET NEXT CHAR-
                                                ACTER
390:   034E C9 24                CMP  #"$"     ;HEX NUMBER?
400:   0350 F0 0A                BEQ  HEXNUMBER
410:   0352 C9 25                CMP  #"%"     ;BINARY NUMBER?
420:   0354 F0 41                BEQ  BINNUMBER
430:                   ;
440:   0356 20 79 00             JSR  CHRGOT   ;REPLACE FLAGS
450:   0359 4C 8D AE             JMP  OLDVECT   ;AND GO TO OLD
                                                 EVALUATION
460:                   ;
470:   035C 20 8D 03 HEXNUMBERJSR  CLRFAC   ;CLEAR FAC
480:   035F 20 73 00 GETNEXT   JSR  CHRGET   ;GET NEXT CHAR-
                                                ACTER
490:   0362 90 0B                BCC  DIGIT    ;DIGIT
500:   0364 C9 41                CMP  #"A"
```

```
510:   0366 90 1F                BCC  END       ;LESS THAN "A"?
520:   0368 C9 47                CMP  #"F"+1
530:   036A B0 1B                BCS  END       ;GREATER THAN "F"?
540:   036C 38                   SEC
550:   036D E9 07                SBC  #7        ;TAKE OFFSET INTO
                                                 ACCOUNT
560:   036F 38       DIGIT       SEC
565:   0370 E9 30                SBC  #"0"      ;CONVERT TO HEX
570:   0372 48                   PHA            ;SAVE CHARACTER
580:   0373 A5 61                LDA  EXP
585:   0375 F0 07                BEQ  STILLZERO  ;IS FAC STILL
                                                    ZERO?
590:   0377 18                   CLC
600:   0378 69 04                ADC  #4        ;EXPONENT + 4=>
                                                 NUMBER * 16
610:   037A B0 0E                BCS  OVER      ;NUMBER TOO LARGE!
620:   037C 85 61                STA  EXP
630:   037E 68       STILLZEROPLA               ;GET DIGIT BACK
640:   037F F0 DE                BEQ  GETNEXT  ;ZERO, THEN ADD-
                                                ITION UNNECESSARY
650:   0381 20 7E BD             JSR  ADDDIGIT  ;ADD DIGIT TO
                                                  FAC
660:   0384 4C 5F 03             JMP  GETNEXT
670:                 ;
680:   0387 4C 79 00 END         JMP  CHRGOT
690:                 ;
700:   038A 4C 7E B9 OVER        JMP  OVERFLOW
710:                 ;
720:   038D A9 00    CLRFAC      LDA  #0
730:   038F A2 0A                LDX  #10
740:   0391 95 5D    LOOP        STA  FLOAT,X  ;CLEAR FLOATING
                                                 POINT AREA
750:   0393 CA                   DEX
```

```
760:    0394 10 FB              BPL  LOOP
770:    0396 60                 RTS
780:                      ;
790:    0397 20 8D 03 BINNUMBERJSR  CLRFAC   ;CLEAR FAC
800:    039A 20 73 00 GETBIN    JSR  CHRGET   ;GET NEXT CHAR-
                                              ACTER
810:    039D C9 32              CMP  #"2"
820:    039F B0 E6              BCS  END      ;GREATER THAN "1"?
830:    03A1 C9 30              CMP  #"0"
840:    03A3 90 E2              BCC  END      ;LESS THAN "0"?
850:    03A5 E9 30              SBC  #"0"     ;FROM ASCII TO HEX
860:    03A7 48                 PHA
870:    03A8 A5 61              LDA  EXP      ;IS NUMBER STILL
                                              ZERO?
880:    03AA F0 04              BEQ  ZERO
890:    03AC E6 61              INC  EXP      ;DOUBLE NUMBER
900:    03AE F0 DA              BEQ  OVER     ;TOO LARGE?
910:    03B0 68       ZERO      PLA
920:    03B1 F0 E7              BEQ  GETBIN   ;DON'T ADD ZERO
930:    03B3 20 7E BD           JSR  ADDDIGIT  ;ADD DIGIT
940:    03B6 4C 9A 03           JMP  GETBIN   ;AND GET NEXT
                                              DIGIT
]033C-03B9
NO ERRORS
```

This routine works in the same way as the subroutine for processing decimal digits, but it is simpler and easier to understand because no fractions or exponents need to be taken into consideration. When you activate the program with SYS 828, you can enter numbers in either hexadecimal or binary format in addition to the usual decimal form. For example:

```
? $FFFF   returns   65535

? %101010 returns   42
```

You are not limited to four digit hex numbers. The entire range of floating point numbers is available. This means that a hex number may have a maximum of 31 places. For example

```
? $FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

returns

```
2.12676479E+37
```

The entire value range cannot be used in a single binary input line; a number of 78 binary digits has a value of about 3E+23.

With this command expansion you can use hex and binary numbers not only in PRINT statements but wherever decimal numbers were previously necessary. This is particularly interesting in connection with POKE, PEEK, and SYS commands. The address $D000 for the video controller is somewhat easier to remember that 53248. For example, sprite 3 can be activated with

```
POKE $D015, PEEK($D015) OR %1000
```

instead of

```
POKE 53248+21, PEEK(53248+21) OR 8
```

There are a few problems with the hex input. Enter

```
? $ABCDEF
```

and you will get a "SYNTAX ERROR." Why? If you look at the number closely you may recognize that it contains the command word "DEF" for the definition of functions. Since the interpreter first converts the input line into tokens, the string "DEF" gets converted to the appropriate token and our new function returns a SYNTAX ERROR. We can easily get around this by adding a space:

```
? $ABCD EF
```

Now we get the correct value 11259375. It is possible to insert the space because the CHRGET routine ignores spaces. This is also the case for normal decimal numbers.

Let us take a closer look at the operation of the routine.

After the usual initialization which sets the vector to our new routine, the flag denoting the variable type is cleared (set to numeric) as per the interpreter routine. Now the next character can be fetched and tested. If it is a dollar or percent sign, with which hex and binary numbers are designated, respectively, a branch is made to our new routine. If this was not the case, the flags are reset with CHRGOT and execution continues with the original evaluation routine of the interpreter. We proceed as follows in the new routine to convert a hex number:

First, the floating-point accumulator is cleared because we will construct our result in it. The next character is fetched and checked to see if it is a digit or a letter from "A" to "F". If this condition is met, the character is converted to the corresponding hex value; for example, "1" becomes the value $01 and "A" becomes $0A. The value in the floating-point accumulator are multiplied by 16, provided it is not zero. We perform this multiplication in the simplest and fastest way. Instead of calling a floating-point multiplication routine, which takes at least a millisecond, we can see that multiplying by 16 is the same as incrementing the power of two by 4: 16 = 2^4. We therefore simply add four to the FAC exponent, which takes only a few microseconds. After we are satisfied that no overflow occurred, we get the character just read and add it to the FAC. If the number is zero, we can skip the addition. This process is done in a loop until the CHRGET routine reads a character which is not part of the number.

The conversion of a binary number follows the same pattern and is even simpler. Here we simply increment the exponent by one instead of multiplying by two. The additional procedures are the same as those for converting the hex numbers.

3.3 Structured Programming

Throughout this book we have examined the operation of the BASIC interpreter, especially the execution of simple commands. We have not examined the concept of programming structures. The interpreter recognizes only two sets of commands for structured programming:

GOSUB ... RETURN

and

FOR ... NEXT

In order to make use of these structures, the interpreter must know where to jump when executing the RETURN command after a GOSUB to a subroutine so that the main program can continue as normal. With the NEXT command, the end value and step size must be known in addition to the address of the start of the loop so that the interpreter can determine when to end the loop. The parameters required for RETURN and NEXT could be stored at a predetermined place in memory. But what happens if we want to nest several subroutines or loops?

Care must be taken to ensure that the parameters for the last used structure (RETURN or NEXT) can be accessed. What was stored last must be gotten back first. We are familiar with this principle from the stack: LAST IN - FIRST OUT. Therefore the BASIC interpreter simply uses the stack to store the parameters of the program structures.

What data must be saved on the stack by a GOSUB command? First, the address following the GOSUB call must certainly be saved. In addition, the current line number must be placed on the stack so that it has the correct value upon return. In order that one can later distinguish the data for a GOSUB command from that of a FOR command, the GOSUB identification code is also placed on the stack. A complete data set on the stack looks like this:

```
Stack pointer
before GOSUB command ----> program pointer hi
                           program pointer lo
                           line number hi
                           line number lo
                           GOSUB code $8D
Stack pointer
after GOSUB command ----->
```

The GOSUB command thus requires 5 bytes of space on the stack. Because the 6510 stack pointer is only 8 bits long, it can address only one page, from $100 to $1FF. It is clear then that subroutines cannot be nested to any desired depth. A maximum of 256/5 = 51 nested subroutines are possible. Since the stack is also used for other purposes as well, fewer are actually allowed. Before the execution of a GOSUB command, a subroutine is called which checks to see if enough space is left on the stack. When calling this subroutine, one half the number of required memory locations is placed into the accumulator. This must be 3 for the GOSUB command; therefore the subroutine tests for 6 bytes. If the required space is not available, the message "OUT OF MEMORY" is given. This message is unfortunately worded the same as the message printed when the memory space for variables has been used up. A message such as "STACK OVERFLOW" would be more appropriate.

The BASIC interpreter has only the area from $013E to $1FA at its disposal in the stack. The memory range from $0100 to $0110 is used for converting floating point numbers to strings and the space from $0111 to $013E is used for error correction when reading from the cassette.

What happens during a RETURN command? First a check is made to see if the top stack element is the code for GOSUB. If this is not the case, the error message "RETURN WITHOUT GOSUB" is given. Otherwise the next four bytes are fetched from the stack and the parameters for line number and program pointer are taken care of. The stack pointer now points to the element to which it pointed before the GOSUB call. A jump is made to the interpreter loop and the program execution automatically continues with the statement following the GOSUB command.

The principle is similar for a FOR-NEXT loop, but somewhat more complicated because of the number of parameters which must be temporarily stored. The required parameters are stored on the stack in the following order:

```
Stack pointer
before FOR command ----> program pointer hi
                         program pointer lo
                         line number hi
                         line number lo
                         mantissa 4
                         mantissa 3
                         mantissa 2     TO value
                         mantissa 1
                         exponent
                         sign
                         mantissa 4
                         mantissa 3
                         mantissa 2     STEP value
                         mantissa 1
                         exponent
                         variable address hi
                         variable address lo
                         FOR code $81
Stack pointer
after FOR command ----->
```

You can see that a FOR-NEXT loop requires 18 bytes of storage on the stack. The following happens with a NEXT command: First a check is made to see if the top stack element is the FOR code $81. If this is not the case, the error message "NEXT WITHOUT FOR" is given. If a variable follows the NEXT command, the address of the variable is determined and compared with the variable address on the stack. If they are the same or there is no variable name given, the variable value is placed in the FAC and the STEP value from the stack is added. This value is saved as the new variable value and can be compared with the end value on the stack. • The sign of the STEP value can determine whether the loop will be ended or not. If the loop can be ended, the stack pointer is incremented by 18 in order to remove the parameters from the stack and a jump made to the interpreter loop where the next statement can be executed. If, on the other hand, the end value was not reached, the line number and program counter are taken from the stack. The stack pointer remains unchanged however, so that the data remain for the next NEXT command.

If a variable name whose address is not saved on the stack follows the NEXT command, the stack pointer is incremented by 18 to see if another FOR-NEXT data set is present on the stack. This automatically takes care of nested loops.

With this knowledge we can add a new structure to BASIC. If you have done any programming in Pascal, you are probably acquainted with the REPEAT...UNTIL loop. This is a program structure which runs until the end criterium is met. For example

```
REPEAT
I=I+1
UNTIL I=10
```

Here the loop is executed until the end condition of I=10 is fulfilled. This structure can be used for a variety of purposes. As with the FOR-NEXT loop, the contents of the loop are executed at least once. Waiting for a key press can also be accomplished with this loop.

```
REPEAT : GET A$ : UNTIL A$<>""
```

or simpler

```
REPEAT : UNTIL PEEK(197)<>64
```

Here the computer waits until the memory location 197 contains a value other than 64, indicating that a key was pressed.

The following machine language program implements this structure in BASIC.

```
100:  033C                        .OPT P1
120:              ;
130:              ;REPEAT-UNTIL LOOP
140:              ;
150:  0308        COMMAND   =    $308      ;EXECUTE VECTOR
                                            FOR COMMAND
160:              ;
170:  A7E7        CMD.OLD   =    $A7E7     ;OLD ROUTINE
180:  0022        ADDR      =    $22       ;ADDRESS FOR
                                            ERROR MESSAGE
190:  0039        LINENO    =    $39       ;ACTUAL LINE
                                            NUMBER
200:  0073        CHRGET    =    $73
210:  0079        CHRGOT    =    CHRGET+6
220:  007A        TXTPTR    =    CHRGOT+1
230:  0100        STACK     =    $100      ;PROCESSOR STACK
240:  A445        ERROR     =    $A445     ;OUTPUT ERROR
                                            MESSAGE
250:              ;
260:  A3FB        TESTSTACK=     $A3FB     ;TEST FOR SPACE
```

```
                                                  IN STACK
270:   AD8A            FRMNUM    =    $AD8A    ;GET NUMERICAL
                                                  EXPRESSION
280:   A7AE            INTER     =    $A7AE    ;INTERPRETER LOOP
290:   AF08            SYNTAX    =    $AF08    ;SYNTAX ERROR
300:   A906            NEXTSTAT  =    $A906    ;SEARCH FOR NEXT
                                                  STATEMENT
310:                   ;
320:   033C                      *=   828
330:   033C A9 47      INIT      LDA  #<TEST
340:   033E A0 03                LDY  #>TEST
350:   0340 8D 08 03             STA  COMMAND   ;VECTOR TO NEW
                                                  ROUTINE
360:   0343 8C 09 03             STY  COMMAND+1
370:   0346 60                   RTS
380:                   ;
390:   0347 20 73 00 TEST        JSR  CHRGET   ;GET NEXT CHAR-
                                                  ACTER
400:   034A C9 21                CMP  #"!"
410:   034C F0 06                BEQ  NEWCMD   ;NEW COMMAND?
420:                   ;
430:   034E 20 79 00             JSR  CHRGOT   ;REPLACE FLAGS
440:   0351 4C E7 A7             JMP  CMD.OLD   ;AND EXECUTE OLD
                                                  COMMANDS
450:                   ;
460:   0354 20 73 00 NEWCMD      JSR  CHRGET   ;NEXT CHARACTER
470:   0357 C9 52                CMP  #"R"     ;REPEAT COMMAND
480:   0359 F0 07                BEQ  REPEAT
490:   035B C9 55                CMP  #"U"     ;UNTIL COMMAND
500:   035D F0 24                BEQ  UNTIL
510:   035F 4C 08 AF SYNERR      JMP  SYNTAX   ;OTHERWISE SYNTAX
                                                  ERROR
520:                   ;
```

```
0362 20 73 00 REPEAT   JSR  CHRGET   ;POINTER TO NEXT
                                      CHARACTER
0365 A9 03             LDA  #3
0367 20 FB A3          JSR  TESTSTACK  ;ENOUGH SPACE
                                        IN STACK?
036A 20 06 A9          JSR  NEXTSTAT  ;SEARCH FOR
                                       NEXT STATEMENT
036D 18                CLC
036E 98                TYA           ;OFFSET TO NEXT
                                      COMMAND
036F 65 7A             ADC  TXTPTR   ;ADD
0371 48                PHA           ;AND ONTO STACK
0372 A5 7B             LDA  TXTPTR+1
0374 69 00             ADC  #0
0376 48                PHA
0377 A5 39             LDA  LINENO   ;LINE NUMBER
0379 48                PHA           ;ON STACK
037A A5 3A             LDA  LINENO+1
037C 48                PHA
037D A9 52             LDA  #"R"     ;AND REPEAT CODE
037F 48                PHA           ;ON STACK
0380 4C AE A7          JMP  INTER    ;TO INTERPRETER
                                      LOOP
                  ;
0383 20 73 00 UNTIL    JSR  CHRGET   ;CONDITION
                                      FOLLOWS?
0386 F0 D7             BEQ  SYNERR   ;NO THEN ERROR
0388 20 8A AD          JSR  FRMNUM   ;EVALUATE CONDI-
                                      TION
038B A8                TAY           ;SAVE RESULT
038C BA                TSX           ;STACK POINTER
                                      TO X
038D BD 01 01          LDA  STACK+1,X  ;LAST STACK
```

```
                                                   ENTRY
780:   0390 C9 52               CMP  #"R"      ;AND TEST FOR
                                                REPEAT CODE
790:   0392 D0 23               BNE  RPTERR    ;NO, THEN ERROR
                                                MESSAGE
800:   0394 98                  TYA
810:   0395 D0 17               BNE  RPTENDE   ;EXPRESSION TRUE,
                                                END LOOP
820:                   ;
830:   0397 BD 02 01            LDA  STACK+2,X
840:   039A 85 3A               STA  LINENO+1  ;GET LINE NUMBER
850:   039C BD 03 01            LDA  STACK+3,X
860:   039F 85 39               STA  LINENO
870:   03A1 BD 04 01            LDA  STACK+4,X
880:   03A4 85 7B               STA  TXTPTR+1  ;AND PROGRAM
                                                POINTER
890:   03A6 BD 05 01            LDA  STACK+5,X ;FROM STACK
900:   03A9 85 7A               STA  TXTPTR
910:   03AB 4C AE A7            JMP  INTER     ;TO INTERPRETER
                                                LOOP
920:                   ;
930:   03AE 8A         RPTENDE  TXA            ;STACK POINTER
940:   03AF 18                  CLC
950:   03B0 69 05               ADC  #5        ;INCREMENT BY 5
960:   03B2 AA                  TAX
970:   03B3 9A                  TXS
980:   03B4 4C AE A7            JMP  INTER     ;AND TO INTER-
                                                PRETER LOOP
990:                   ;
1000:  03B7 A9 C0      RPTERR   LDA  #<TEXT
1010:  03B9 85 22               STA  ADDR      ;SET POINTER TO
                                                ERROR MESSAGE
1020:  03BB A9 03               LDA  #>TEXT
```

```
1030:  03BD 4C 45 A4          JMP  ERROR
1040:                  ;
1050:  03C0 55 4E 54 TEXT     .ASC "UNTIL WITHOUT REPEAT"
]033C-03D4
NO ERRORS
```

Now let's see how our new commands are used. For the sake of simplicity we have designated our new commands with a prefixed exclamation point "!" and an "R" for REPEAT and a "U" for UNTIL. When you have the assembly language program assembled and activated with SYS 828, you can try it out with the following program:

```
100 I=0
110 !R
120 I=I+1 : PRINT I
130 !U I=10
```

The program prints the numbers from 1 to 10. Nested loops are also possible.

```
100 I=0
110 !R
120 I=I+1 : PRINT "I=" ; I : J=0
130 !R
140 J=J+1 : PRINT "J=" ; J
150 !U J=3
160 !U I=3
```

In these nested loops the counter I runs from 1 to 3 and the counter J in the inner loop also from 1 to 3. The above problem could be solved more simply with two nested FOR-NEXT loops. The main applications area of the REPEAT through the loop is not known when the loop started, but will be determined during the loop. The stop criterium might be a pressed key, for example. This program structure is also very useful for iterations such as calculating a square root using the Newton method.

```
100 INPUT "INPUT ";A
110 X1 = A
120 !R
130 X0 = X1
140 X1 = (X0 + A/X0)/2
150 !U ABS (X1-X0) < 1E-8
160 PRINT "THE ROOT IS ";X1
```

Here an approximation is calculated until the difference between two successive values is less than 10^-8. Try this with a few values and compare the result with that of the SQR function.

Endless loops can also be constructed with this structure, by using an ending criterium which is never true. For example

```
110 !R
110 PRINT TI
120 !U 1=0
```

This loop will never be exited by the program.

The REPEAT...UNTIL loop runs faster than an IF...GOTO construction because the line number to which GOTO is directed must be searched for each time. With the UNTIL command, this address needs only to be fetched from the stack.

In addition, the program is easier to read and understand because the intentions of the programmer come through more clearly.

We now come to a description of the machine language program. We proceed in much the same manner as the other programs structures discussed earlier. After the usual initialization in which the vector for command evaluation is changed to point to our routine, we first test to see if a new command was used. If no exclamation point was found, control is returned to the original command evaluation routine. Otherwise the next character is fetched and checked to see if it is "U" or "R". The routines REPEAT and UNTIL are branched to accordingly. If neither of these two characters were read, we jump to the error message "SYNTAX ERROR."

For the REPEAT command we set the program pointer to the next character by a call to CHRGET and check to see that enough space is left on the stack. We use the routine NEXTSTAT to search for the next command, the relative address of which we get back in the Y register. We add this value to the program counter and place it on the stack. The line number is also placed on the stack. To denote the data set as a REPEAT command, we also push the letter "R" on the stack. The data set in the stack is constructed according to the GOSUB command. The work is now done and we return to the interpreter loop.

The UNTIL command checks to see that a condition follows and evaluates it. The result is saved in the Y register. Now we load the X register with the stack pointer and compare the top stack element with "R", the designator for REPEAT. If this element was not an R, we output the message

"UNTIL WITHOUT REPEAT." Note that the last character of the error message must be shifted (bit 7 set). This is how the error message output routine determines the end of the message. If we did find an R, the next action is dependent on the result of the condition. If the condition was not met, we load the program pointer and line number from the stack and jump to the interpreter loop. Note that the data is not taken from the stack with PLA but with LDA STACK,X, after the stack register was first copied into the X register. This retains the value of the stack pointer and the data remain for the next UNTIL command. If, however, the condition was satisfied, we simply increment the stack pointer by 5. This has the effect of removing the data set from the stack and we continue with the next command.

3.4 Using new keywords

The easiest way to add new commands to the BASIC interpreter is to give the command a name by which you can access it. Internally this keyword is stored in the form of a token, an interpreter code which can range in value from $80 to $FF.

The Commodore 64 BASIC uses the tokens from $80 to $CB for itself, as well as $FF for pi. If we want to add new keywords, the interpreter codes from $CC (204) to $FE (254) are available to us. We could therefore add up to 54 new commands. Let us consider what is necessary in order to do this.

First, there must be a routine which converts a line of BASIC text into the new tokens upon input. The routine for executing the commands must recognize the new token and call the appropriate routine to execute this new command. So that we can list our program, the LIST program must also be changed to output the ASCII form of the new commands when it finds the token. The most convenient way to do all this is to place our new keywords and the addresses of the corresponding routines in a table, exactly as the interpreter does with the standard commands.

Recall from our discussion about BASIC vectors that four vectors are necessary for these tasks. We have already used the vectors for BASIC command execution ($308) and function calculation ($30A). For converting keywords into tokens we must use the vector $304. To convert tokens back into keywords with the LIST command we must use the vector $306.

Once we have written these routines, it is quite easy to add new keywords. We need only place the keyword together with the address of the routine which executes the command in a table.

This procedure is also faster in execution because no special characters such as "!" need be added for recognition of the new command. In the program itself, the command "REPEAT" looks better than "!R".

Before we venture to write a routine which converts new keywords to tokens, we will first take a look at how the BASIC interpreter handles this. In order to do this we have re-assembled the ROM routine for you here. If we follow the principle, it is not hard to change the routine to add our own tokens.

```
A57C                     .OPT P1
             ;
             ;ROM ROUTINE FOR CONVERSION TO TOKENS
             ;
             ;SPECIAL TOKENS
             ;
0083         DATA   =   $83
008F         REM    =   $8F
0099         PRINT  =   $99
             ;
0008         CHAR   =   8        ;ACTUAL CHARACTER
000B         COUNT  =   11       ;COUNTER FOR COM-
                                  MAND WORDS
0071         PNT    =   $71      ;POINTER IN LINE
                                BEING CONVERTED FROM
```

```
230:    0022              QUOTE     =    $22
240:    000F              FLAG      =    15        ;FLAG FOR DATA
                                                    AND REM
250:    007A              TXTPTR    =    $7A       ;POINTER IN LINE
                                                  BEING CONVERTED TO
260:    0200              BUFFER    =    $200      ;INPUT BUFFER
270:                      ;
280:    A09E              TABLE     =    $A09E     ;TABLE OF COMMAND
                                                    WORDS
290:                      ;
300:    A57C                        *=   $A57C     ;ROM ROUTINE
310:                      ;
320:    A57C A6 7A                  LDX  TXTPTR    ;POINTER TO FIRST
                                                    CHARACTER
330:    A57E A0 04                  LDY  #4        ;POINTER TO LINE
                                                    CONVERTED FROM
340:    A580 84 0F                  STY  FLAG      ;CLEAR FLAG
350:    A582 BD 00 02 NEXTCHAR LDA  BUFFER,X  ;GET CHARACTER
                                                      FROM BUFFER
360:    A585 10 07                  BPL  NORMAL
370:    A587 C9 FF                  CMP  #$FF      ;CODE FOR 'PI'
380:    A589 F0 3E                  BEQ  TAKCHAR   ;YES, TAKE IT
390:    A58B E8                     INX            ;OTHERWISE IGNORE
                                                    CHARACTER
400:    A58C D0 F4                  BNE  NEXTCHAR
410:                      ;
420:    A58E C9 20        NORMAL    CMP  #" "      ;SPACE?
430:    A590 F0 37                  BEQ  TAKCHAR   ;TAKE IT
440:    A592 85 08                  STA  CHAR      ;SAVE CHARACTER
450:    A594 C9 22                  CMP  #QUOTE    ;QUOTE?
460:    A596 F0 56                  BEQ  GETCHAR   ;YES
470:    A598 24 0F                  BIT  FLAG      ;TEST FLAG
480:    A59A 70 2D                  BVS  TAKCHAR   ;TAKE AS DATA
```

```
                                                  MODE
490:   A59C C9 3F               CMP  #"?"     ;QUESTION MARK
500:   A59E D0 04               BNE  SKIP
510:   A5A0 A9 99               LDA  #PRINT   ;REPLACE WITH
                                               PRINT CODE
520:   A5A2 D0 25               BNE  TAKCHAR
530:   A5A4 C9 30     SKIP      CMP  #"0"     ;LESS THAN '0'
540:   A5A6 90 04               BCC  SKIP1
550:   A5A8 C9 3C               CMP  #"<"     ;LESS THAN '<'?
560:   A5AA 90 1D               BCC  TAKCHAR   ;YES, TAKE CHAR-
                                                ACTER
570:   A5AC 84 71     SKIP1     STY  PNT      ;SAVE POINTER IN
                                               LINE
580:   A5AE A0 00               LDY  #0
590:   A5B0 84 0B               STY  COUNT    ;COUNTER FOR COM-
                                           MAND WORDS TO ZERO
600:   A5B2 88                  DEY
610:   A5B3 86 7A               STX  TXTPTR   ;SAVE LINE POINTER
620:   A5B5 CA                  DEX
630:                  ;
640:   A5B6 C8        CMPLOOP   INY           ;POINTER IN COM-
                                               MAND TABLE
650:   A5B7 E8                  INX           ;AND INCREMENT
                                               LINE POINTER
660:   A5B8 BD 00 02 TESTNEXT LDA  BUFFER,X  ;GET CHARACTER
                                                 FROM BUFFER
670:   A5BB 38                  SEC
680:   A5BC F9 9E A0            SBC  TABLE,Y   ;COMPARE WITH
                                                COMMAND WORD
690:   A5BF F0 F5               BEQ  CMPLOOP   ;SAME, THEN NEXT
                                                CHARACTER
700:   A5C1 C9 80               CMP  #$80     ;LAST LETTER?
710:   A5C3 F0 30               BEQ  NEXTCMD ;OTHERWISE POINT-
```

```
                                                 ER TO NEXT COMMAND
720:   A5C5 05 0B              ORA  COUNT    ;FOUND #+$80=INTER
725:   A5C7 A4 71     TAKCHAR1 LDY  PNT      ;GET POINTER BACK
730:                  ;
740:   A5C9 E8        TAKCHAR  INX
750:   A5CA C8                 INY
760:   A5CB 99 FB 01           STA  BUFFER-5,Y  ;SAVE CODE
770:   A5CE B9 FB 01           LDA  BUFFER-5,Y  ;RESTORE
                                                 FLAGS
780:   A5D1 F0 36              BEQ  END      ;LINE END?
790:   A5D3 38                 SEC
800:   A5D4 E9 3A              SBC  #":"     ;SEPARATOR?
810:   A5D6 F0 04              BEQ  SKIP2    ;CLEAR DATA FLAG
820:   A5D8 C9 49              CMP  #DATA-":"  ;CODE FOR
                                                'DATA'
830:   A5DA D0 02              BNE  SKIP3
840:   A5DC 85 0F     SKIP2    STA  FLAG     ;SET BIT 6 FOR
                                             'DATA'
850:   A5DE 38        SKIP3    SEC
860:   A5DF E9 55              SBC  #REM-":"  ;CODE FOR 'REM'
870:   A5E1 D0 9F              BNE  NEXTCHAR  ;NO, GET NEXT
                                               CHARACTER
880:   A5E3 85 08              STA  CHAR     ;SAVE ZERO BYTE
                                              FOR 'REM'
890:   A5E5 BD 00 02  REMLOOP  LDA  BUFFER,X
900:   A5E8 F0 DF              BEQ  TAKCHAR  ;LINE END, TAKE
                                              CHARACTER
910:   A5EA C5 08              CMP  CHAR     ;NEXT '"' OR REM
                                              OR DATA
920:   A5EC F0 DB              BEQ  TAKCHAR  ;YES?
930:   A5EE C8        GETCHAR  INY
940:   A5EF 99 FB 01           STA  BUFFER-5,Y  ;TAKE CHAR-
                                                 ACTER
```

```
950:    A5F2 E8                 INX
960:    A5F3 D0 F0              BNE  REMLOOP
970:                  ;
980:    A5F5 A6 7A    NEXTCMD   LDX  TXTPTR   ;LINE POINTER TO
                                               START OF WORD
990:    A5F7 E6 0B              INC  COUNT    ;COUNTER TO NEXT
                                               COMMAND WORD
1000:   A5F9 C8       CONTINUE  INY
1010:   A5FA B9 9D A0           LDA  TABLE-1,Y
1020:   A5FD 10 FA              BPL  CONTINUE  ;WORD NOT DONE!
1030:   A5FF B9 9E A0           LDA  TABLE,Y
1040:   A602 D0 B4              BNE  TESTNEXT  ;TEST FOR NEXT
                                                COMMAND WORD
1050:                 ;
1060:   A604 BD 00 02           LDA  BUFFER,X
1070:   A607 10 BE              BPL  TAKCHAR1  ;TAKE CHARACTER
                                                AS SUCH
1080:                 ;
1090:   A609 99 FC 01 END       STA  BUFFER-4,Y  ;END BUFFER
                                                  WITH ZERO
1100:                 ;
1110:   A60C C6 7B              DEC  TXTPTR+1
1120:   A60E A9 FF              LDA  #$FF     ;TXTPTR TO $01FF,
                                               BUFFER-1
1130:   A610 85 7A              STA  TXTPTR
1140:   A612 60                 RTS
]A57C-A613
NO ERRORS
```

When a line of BASIC text is to be converted to tokens, it must be placed into the BASIC input buffer located at $200 to $258. The pointer TXTPTR ($7A/$7B) must point to the first character following the line number. The X register is loaded with this pointer. The X register serves throughout the entire routine as a pointer to the text to be converted. The Y register is used as a pointer in the converted line. After the FLAG is cleared, the first character of the line is examined. If the ASCII value of this character is greater than $7F, it is checked for the code 255 for pi. If this test was positive, the character is accepted as such. All other characters with bit 7 set are ignored; the pointer is incremented and the next character is tested. If the character is a normal unshifted character, it is checked for a special character. A space is accepted unchanged. Otherwise the current character is saved in CHAR. If the character is a quotation mark ("), a branch is made to GETCHAR where characters are read until another quotation mark is encountered. A DATA statement is recognized by checking FLAG. If a DATA command is active, text is accepted unchanged. The code "?" is next replaced with "PRINT". After the digits and the characters ";" and ":" are filtered out and accepted unchanged, comes the actual conversion to tokens.

The pointer in the line being converted (X register) is stored in PNT and the counter for the token number of the keyword is initialized. The comparison is executed at the label CMPLOOP. The current character in the buffer is compared to the first letter in the keyword table. If the characters are equal, the next character is compared to the second letter. If these are equal, the difference is checked to see if it is $80. This value denotes the character as the last character of a command in the keyword table because it is stored with bit 7 set. If this is true, the accumulator contains the difference $80. By logical ORing with the command number COUNT you get the token number, which is then saved. If the characters were not equal, however, the start of the next keyword is found by NEXTCMD and the counter for the number of keywords is incremented by one. If we are not at the end of the table, a branch is made back to the compare loop where the next word from the table is compared. If the end of the table was found (denoted by a zero byte), the current character is accepted unchanged.

After either the token or the unchanged character is stored by the routine at label TAKCHAR, the special characters are handled. If the colon is found, FLAG is cleared so that it can be be set again by another DATA statement. If the REM command was found, the current character is saved as a zero and all of the characters up to a zero (end of line) are accepted unchanged by REMLOOP. At the end of the routine (label END), the converted buffer is terminated with a zero and TXTPTR set to one character before the buffer.

If we now want to convert our own keywords to tokens, we must ensure that the table containing our own commands is searched after the command table in ROM. In addition we must determine which tokens we want to use for the new keywords. The tokens starting at $CC should be used since they follow the existing commands directly.

```
C000                    .OPT P1
                ;
                ;ROUTINE FOR USING CUSTOM TOKENS
                ;
                ;SPECIAL TOKENS
                ;
0083            DATA    =       $83
008F            REM     =       $8F
0099            PRINT   =       $99
```

```
190:                    ;
200:   0008             CHAR     =    8
210:   000B             COUNT    =    11
220:   0071             PNT      =    $71
230:   0022             QUOTE    =    $22
240:   000F             FLAG     =    15
250:   007A             TXTPTR   =    $7A
260:   0200             BUFFER   =    $200      ;INPUT BUFFER
270:                    ;
280:   A09E             TABLE    =    $A09E     ;TABLE OF COMMAND
                                                 WORDS
290:                    ;
300:   C000                      *=   $C000     ;NEW ROUTINE
310:                    ;
320:   C000 A6 7A                LDX  TXTPTR    ;POINTER TO FIRST
                                                 CHARACTER
330:   C002 A0 04                LDY  #4        ;POINTER WITHIN
                                             LINE BEING CONVERTED
340:   C004 84 0F                STY  FLAG      ;FLAG FOR SPECIAL
                                                 CHARACTERS
350:   C006 BD 00 02 NEXTCHAR LDA  BUFFER,X   ;GET CHARACTER
                                                   FROM BUFFER
360:   C009 10 07                BPL  NORMAL
370:   C00B C9 FF                CMP  #$FF      ;CODE FOR 'PI'?
380:   C00D F0 3E                BEQ  TAKCHAR   ;YES, TAKE CODE
                                                  AS SUCH
390:   C00F E8                   INX            ;OTHERWISE IGNORE
                                                 CHARACTER
400:   C010 D0 F4                BNE  NEXTCHAR
410:                    ;
420:   C012 C9 20       NORMAL   CMP  #" "      ;SPACE?
430:   C014 F0 37                BEQ  TAKCHAR   ;TAKE AS SUCH
440:   C016 85 08                STA  CHAR      ;SAVE CHARACTER
```

```
450:   C018 C9 22              CMP  #QUOTE   ;QUOTE?
460:   C01A F0 55              BEQ  GETCHAR
470:   C01C 24 0F              BIT  FLAG     ;DATA MODE?
480:   C01E 70 2D              BVS  TAKCHAR  ;YES, TAKE AS
                                              SUCH
490:   C020 C9 3F              CMP  #"?"     ;QUESTION MARK?
500:   C022 D0 04              BNE  SKIP
510:   C024 A9 99              LDA  #PRINT   ;REPLACE WITH
                                              PRINT CODE
520:   C026 D0 25              BNE  TAKCHAR
530:   C028 C9 30     SKIP     CMP  #"0"     ;SMALLER THAN '0'?
540:   C02A 90 04              BCC  SKIP1
550:   C02C C9 3C              CMP  #"<"     ;LESS THAN '<'?
560:   C02E 90 1D              BCC  TAKCHAR  ;YES, TAKE CHAR-
                                              ACTER AS SUCH
570:   C030 84 71     SKIP1    STY  PNT      ;SAVE LINE POINTER
580:   C032 A0 00              LDY  #0
590:   C034 84 0B              STY  COUNT    ;COUNTER FOR COM-
                                              MAND WORDS TO 0
600:   C036 88                 DEY
610:   C037 86 7A              STX  TXTPTR
620:   C039 CA                 DEX
630:                  ;
640:   C03A C8        CMPLOOP  INY
650:   C03B E8                 INX           ;INCREMENT POINTER
660:   C03C BD 00 02  TESTNEXT LDA  BUFFER,X ;GET CHARACTER
                                              FROM BUFFER
670:   C03F 38                 SEC
680:   C040 F9 9E A0           SBC  TABLE,Y  ;COMPARE WITH
                                              COMMAND WORDS
690:   C043 F0 F5              BEQ  CMPLOOP  ;SAME, THEN NEXT
                                              CHARACTER
700:   C045 C9 80              CMP  #$80     ;LAST LETTER?
```

```
710:  C047 D0 2F              BNE  NEXTCMD   ;OTHERWISE POINT-
                                             ER TO NEXT COMMAND
720:  C049 05 0B              ORA  COUNT     ;#+$80 = INTER-
                                              PRETER CODE
730:  C04B A4 71    TAKCHAR1  LDY  PNT       ;GET POINTER BACK
740:                ;
750:  C04D E8       TAKCHAR   INX
760:  C04E C8                 INY
770:  C04F 99 FB 01           STA  BUFFER-5,Y  ;SAVE CODE
780:  C052 C9 00              CMP  #0        ;GET FLAGS BACK
790:  C054 F0 38              BEQ  END       ;END OF LINE?
800:  C056 38                 SEC
810:  C057 E9 3A              SBC  #":"      ;SEPARATOR?
820:  C059 F0 04              BEQ  SKIP2
830:  C05B C9 49              CMP  #DATA-":"   ;CODE FOR
                                                'DATA'?
840:  C05D D0 02              BNE  SKIP3
850:  C05F 85 0F    SKIP2     STA  FLAG      ;SET BIT 6 FOR
                                              'DATA'
860:  C061 38       SKIP3     SEC
870:  C062 E9 55              SBC  #REM-":"   ;CODE FOR 'REM'?
880:  C064 D0 A0              BNE  NEXTCHAR   ;NO, GET NEXT
                                               CHARACTER
890:  C066 85 08              STA  CHAR      ;SAVE CHARACTER
900:  C068 BD 00 02 REMLOOP   LDA  BUFFER,X
910:  C06B F0 E0              BEQ  TAKCHAR   ;END OF LINE,
                                              TAKE AS SUCH
920:  C06D C5 08              CMP  CHAR      ;NEXT 'N' OR REM
                                              OR DATA
930:  C06F F0 DC              BEQ  TAKCHAR   ;YES
940:  C071 C8       GETCHAR   INY
950:  C072 99 FB 01           STA  BUFFER-5,Y  ;TAKE CHAR-
                                                ACTER
```

```
960:   C075 E8                  INX
970:   C076 D0 F0               BNE  REMLOOP
980:                  ;
990:   C078 A6 7A     NEXTCMD   LDX  TXTPTR
1000:  C07A E6 0B               INC  COUNT     ;POINTER TO NEXT
                                                COMMAND WORD
1010:  C07C C8        CONTINUE  INY
1020:  C07D B9 9D A0            LDA  TABLE-1,Y  ;NEXT LETTER
1030:  C080 10 FA               BPL  CONTINUE  ;WORD NOT DONE?
1040:  C082 B9 9E A0            LDA  TABLE,Y
1050:  C085 D0 B5               BNE  TESTNEXT  ;TEST FOR NEXT
                                                 COMMAND WORD
1060:  C087 F0 0F               BEQ  NEWTOK   ;USE NEW TABLE
1070:                  ;
1080:  C089 BD 00 02  NOTFOUND  LDA  BUFFER,X
1090:  C08C 10 BD               BPL  TAKCHAR1  ;TAKE CHR AS
                                                 SUCH
1100:                  ;
1110:  C08E 99 FD 01  END       STA  BUFFER-3,Y  ;LINK BYTE
                                          ZERO FOR DIRECT MODE
1120:                  ;
1130:  C091 C6 7B               DEC  TXTPTR+1
1140:  C093 A9 FF               LDA  #$FF      ;TXTPTR TO $01FF,
                                                BUFFER-1
1150:  C095 85 7A               STA  TXTPTR
1160:  C097 60                  RTS
1170:                  ;
1180:                  ;WORK ON NEW COMMAND
1190:  C098 A0 00     NEWTOK    LDY  #0        ;POINTER TO START
                                                OF NEW TABLE
1200:  C09A B9 C3 C0            LDA  NEWTAB,Y  ;GET FIRST CHAR-
                                                 ACTER FROM TABLE
1210:  C09D D0 02               BNE  NEWTEST
```

```
1220:                  ;
1230:   C09F C8        NEWCMP    INY
1240:   C0A0 E8                  INX
1250:   C0A1 BD 00 02  NEWTEST   LDA  BUFFER,X   ;COMPARE ROUTINE
                                                  FOR NEW
1260:   C0A4 38                  SEC            ;COMMAND TABLE
1270:   C0A5 F9 C3 C0            SBC  NEWTAB,Y
1280:   C0A8 F0 F5               BEQ  NEWCMP
1290:   C0AA C9 80               CMP  #$80
1300:   C0AC D0 04               BNE  NEXTNEW   ;TEST FOR NEXT
                                                 COMMAND
1310:   C0AE 05 0B               ORA  COUNT     ;FOUND
1320:   C0B0 D0 99               BNE  TAKCHAR1   ;ABSOLUTE JUMP
1330:                  ;
1340:   C0B2 A6 7A     NEXTNEW   LDX  TXTPTR
1350:   C0B4 E6 0B               INC  COUNT     ;INCREMENT TOKEN
                                                NUMBER
1360:   C0B6 C8        CONT1     INY
1370:   C0B7 B9 C2 C0            LDA  NEWTAB-1,Y   ;POINTER TO
                                                NEXT COMMAND WORD
1380:   C0BA 10 FA               BPL  CONT1
1390:   C0BC B9 C3 C0            LDA  NEWTAB,Y
1400:   C0BF D0 E0               BNE  NEWTEST   ;COMPARE TO
                                                 INPUT LINE
1410:   C0C1 F0 C6               BEQ  NOTFOUND   ;END OF NEW
                                                  TABLE
1420:                  ;
1430:   C0C3 52 45 50  NEWTAB    .ASC "REPEAT"   ;TABLE OF
                                                NEW COMMAND WORDS
1440:   C0C9 55 4E 54            .ASC "UNTIL"
1450:   C0CE 43 4F 4D            .ASC "COMMAND"
1460:   C0D5 00                  .BYT 0         ;END OF TABLE
]C000-C0D6
```

NO ERRORS

With these routines we can convert our own keywords to tokens. When entering the new keywords in the new table you must be sure that the last character of each command is entered with bit 7 set. This is done by pressing shift when the entering the character. In our assembly listing this is represented as an underlined character. The new commands can also be abbreviated as desired. reP could be used for repeat or uN for until.

With our procedure you can assign the new keywords a token value from $CC to $FE. This gives us a maximum of 51 new command words. Because this table is indexed with an 8-bit register, the total length of the commands may not be longer than 255 characters. The end of the table must be denoted by a zero byte.

In order to activate our new routine we must set the vector $304/$305 to the routine. Before we do this, we first want to write a routine which allows us to LIST the new keywords. The BASIC vector $306/$307 is used for this. Converting tokens to ASCII text takes place here; the organizational work such as taking care of the line end and line numbers is handled by the list routine. Let us take a look at the interpreter LIST routine.

```
100:  A71A                       .OPT P1
110:                 ;
120:                 ;INTERPRETER LIST ROUTINE
130:                 ;
140:  000F           QUOTFLG  =    15        ;FLAG FOR QUOTE
                                              MODE
150:  0049           PNT      =    $49
160:  A09E           TABLE    =    $A09E     ;INTERPRETER COM-
                                              MAND TABLE
170:  AB47           CHAROUT  =    $AB47     ;OUTPUT CHARACTER
180:                 ;
190:  A71A                    *=   $A71A
200:  A71A 10 D7              BPL  $A6F3     ;NO INTERPRETER
                                              CODE, SO OUTPUT
210:  A71C C9 FF              CMP  #$FF
220:  A71E F0 D3              BEQ  $A6F3     ;CODE FOR PI,
                                              OUTPUT
-30:  A720 24 0F              BIT  QUOTFLG    ;QUOTE MODE?
240:  A722 30 CF              BMI  $A6F3     ;YES, OUTPUT
                                              UNCHANGED
250:  A724 38                 SEC
260:  A725 E9 7F              SBC  #$7F      ;SUBRACT OFFSET
270:  A727 AA                 TAX            ;SAVE CODE AS
                                              COUNTER
280:  A728 84 49              STY  PNT       ;SAVE POINTER
290:  A72A A0 FF              LDY  #-1
300:  A72C CA        NEXT     DEX
310:  A72D F0 08              BEQ  FOUND     ;XTH COMMAND WORD
                                              FOUND?
320:  A72F C8        LOOP     INY
330:  A730 B9 9E A0           LDA  TABLE,Y
340:  A733 10 FA              BPL  LOOP      ;WORD NOT DONE?
350:  A735 30 F5              BMI  NEXT      ;NEXT WORD
```

```
                     ;
A737 C8        FOUND    INY
A738 B9 9E A0           LDA  TABLE,Y  ;GET LETTER
A73B 30 B2              BMI  $A6EF    ;LAST CHARACTER?
A73D 20 47 AB           JSR  CHAROUT  ;OUTPUT CHARACTER
A740 D0 F5              BNE  FOUND    ;ABSOLUTE JUMP
]A71A-A742
NO ERRORS
```

The routine checks for interpreter codes (is bit 7 set?). The special code for pi is left unchanged. This is also ignored in the quote mode. First we search for the keyword. The token is brought into the range 1-76 by subtracting $7F. Then the keyword table is searched and the token number is decremented by one at the end of each keyword, which is denoted by the set bit 7. When the number counts down to zero, we have found the appropriate word in the table. Now we output all characters until we encounter the one with bit 7 set. In this case we branch back to the list routine. There bit 7 is cleared as the character is printed.

If we have new tokens to list, we need only check to see if the token is greater than $CB. If this is the case, we can search for the keyword in our new table using the same method, otherwise we leave the work for the original routine.

```
C000                    .OPT P1
               ;
               ;LIST ROUTINE FOR NEW COMMANDS
               ;
000F           QUOTFLG  =    15
```

```
150:   0049              PNT      =     $49
160:   A09E              TABLE    =     $A09E    ;COMMAND TABLE
170:   AB47              CHAROUT  =     $AB47    ;OUTPUT CHARACTER
180:                     ;
190:   C000 10 0F                 BPL   OUT      ;NO TOKEN, SO
                                                  OUTPUT
200:   C002 24 0F                 BIT   QUOTFLG  ;QUOTE MODE?
210:   C004 30 0B                 BMI   OUT      ;SO OUTPUT
220:   C006 C9 FF                 CMP   #$FF     ;PI?
230:   C008 F0 07                 BEQ   OUT      ;SO OUTPUT
240:   C00A C9 CC                 CMP   #$CC     ;NEW TOKEN?
250:   C00C B0 06                 BCS   NEWLIST  ;YES
260:                     ;
270:   C00E 4C 24 A7              JMP   $A724    ;LIST OLD TOKENS
280:   C011 4C F3 A6 OUT          JMP   $A6F3    ;OUTPUT BYTE
290:                     ;
300:   C014 38           NEWLIST  SEC
310:   C015 E9 CB                 SBC   #$CB     ;SUBTRACT OFFSET
320:   C017 AA                    TAX            ;CODE AS COUNTER
330:   C018 84 49                 STY   PNT
340:   C01A A0 FF                 LDY   #-1
350:   C01C CA           NEXT     DEX            ;WORD FOUND?
360:   C01D F0 08                 BEQ   FOUND    ;YES
370:   C01F C8           LOOP     INY
380:   C020 B9 35 C0              LDA   NEWTAB,Y
390:   C023 10 FA                 BPL   LOOP     ;EXPECT END OF
                                                  WORD
400:   C025 30 F5                 BMI   NEXT     ;NEXT WORD
410:                     ;
420:   C027 C8           FOUND    INY
430:   C028 B9 35 C0              LDA   NEWTAB,Y ;COMMAND WORD
440:   C02B 30 05                 BMI   OLDEND   ;AT END?
450:   C02D 20 47 AB              JSR   CHAROUT  ;OUTPUT CHARACTER
```

```
C030 D0 F5                  BNE  FOUND   ;AND CONTINUE
                       ;
C032 4C EF A6 OLDEND   JMP  $A6EF   ;TO OLD ROUTINE
                       ;
C035 52 45 50 NEWTAB   .ASC "REPEAT"  ;COMMAND
                                              TABLE
C03B 55 4E 54          .ASC "UNTIL"
C040 43 4F 4D          .ASC "COMMAND"
C047 00                .BYT 0
]C000-C048
NO ERRORS
```

When we change the LIST vector $306-$307 to point to this routine, we can list our new commands correctly. The keyword table NEWTAB is naturally identical to the table in the routine for creating new tokens and can be shared. In a practical application you should assemble the two routines together and create a single initialization program which changes both vectors appropriately.

We need routines which allow us to process the new commands from the BASIC interpreter which can call the new commands and functions. This happens, as we know, by using the vector $308/$309 for commands and $30A/$30B for functions. To simplify the processing, the new commands should be assigned tokens such that they form a block. The routines can then verify that the token lies in the range of new commands or functions. The token value can be used as a pointer to a table which contains the starting addresses of the routines which perform the new commands. This is the same procedure which the built-in interpreter uses. We now present a universal routine which handles the processing of new tokens. You need only establish the range of the new commands and functions and place the starting addresses of the corresponding routines in the table.

```
100:  C000                  .OPT P1
110:              ;
120:              ;INCLUDING NEW TOKENS
130:              ;
140:  0308        CMDVEC   =  $308     ;COMMAND VECTOR
150:  030A        FUNVEC   =  $30A     ;FUNCTION VECTOR
160:              ;
170:  000D        TYPFLAG  =  13       ;FLAG NUMERIC/
                                        STRING
180:  0073        CHRGET   =  $73
190:  0079        CHRGOT   =  CHRGET+6
200:  007A        TXTPTR   =  CHRGOT+1
210:  A7ED        EXECOLD  =  $A7ED    ;OLD COMMAND
                                        EXECUTION
215:  A7AE        INTER    =  $A7AE    ;INTERPRETER LOOP
217:  AE8D        FUNCTOLD =  $AE8D    ;OLD FUNCTION
                                        CALCULATION
218:  AEF1        GETTERM  =  $AEF1    ;GET EXPRESSION
                                        IN PARENTHESES
219:  AD8D        CHECKNUM =  $AD8D    ;TEST FOR NUM-
                                        ERICAL RESULT
220:  0054        JUMP     =  $54      ;JUMP COMMAND FOR
                                        FUNCTIONS
300:  00CC        CMDSTART =  $CC      ;FIRST COMMAND
                                        TOKEN
310:  00E0        CMDEND   =  $E0      ;LAST COMMAND
                                        TOKEN
320:              ;
330:  00E1        FUNSTART =  $E1      ;FIRST FUNCTION
```

```
                                                     TOKEN
340:   00FE           FUNEND    =    $FE       ;LAST FUNCTION
                                                     TOKEN
350:                  ;
400:   C000 A9 15     INIT      LDA  #<NEWCMD
410:   C002 A0 C0               LDY  #>NEWCMD
420:   C004 8D 08 03            STA  CMDVEC    ;COMMAND VECTOR
430:   C007 8C 09 03            STY  CMDVEC+1
440:                  ;
450:   C00A A9 3C               LDA  #<NEWFUN
460:   C00C A0 C0               LDY  #>NEWFUN
470:   C00E 8D 0A 03            STA  FUNVEC    ;FUNCTION VECTOR
480:   C011 8C 0B 03            STY  FUNVEC+1
490:   C014 60                  RTS
500:                  ;
510:   C015 20 73 00 NEWCMD     JSR  CHRGET    ;NOT  TAKEN
520:   C018 20 1E C0            JSR  TESTCMD    ;EXECUTE COMMAND
530:   C01B 4C AE A7            JMP  INTER     ;BACK TO INTER-
                                                PRETER LOOP
550:   C01E C9 CC     TESTCMD   CMP  #CMDSTART
560:   C020 90 04               BCC  OLDCMD    ;OLD COMMAND?
570:   C022 C9 E1               CMP  #CMDEND+1
580:   C024 90 06               BCC  OKNEW     ;EXECUTE NEW
                                                COMMAND
590:   C026 20 79 00 OLDCMD     JSR  CHRGOT    ;REPLACE FLAGS
600:   C029 4C ED A7            JMP  EXECOLD    ;AND EXECUTE OLD
                                                 COMMAND
610:                  ;
620:   C02C 38        OKNEW     SEC            ;NEW COMMANDS
630:   C02D E9 CC               SBC  #CMDSTART   ;SUBRACT OFFSET
640:   C02F 0A                  ASL            ;TIMES 2
650:   C030 AA                  TAX
660:   C031 BD 6F C0            LDA  CMDTAB+1,X   ;HIGH BYTE
```

```
670:   C034 48                PHA             ;RETURN ADDRESS
                                               ON STACK
680:   C035 BD 6E C0          LDA  CMDTAB,X
690:   C038 48                PHA             ;LOW BYTE
700:   C039 4C 73 00          JMP  CHRGET     ;GET NEXT
                                               CHARACTER
710:                 ;
720:   C03C A9 00    NEWFUN   LDA  #0
730:   C03E 85 0D             STA  TYPFLAG    ;TYPE TO NUMERIC
740:   C040 20 73 00          JSR  CHRGET     ;GET TOKEN
750:   C043 C9 E1             CMP  #FUNSTART
760:   C045 90 04             BCC  OLDFUN     ;OLD FUNCTION?
770:   C047 C9 FF             CMP  #FUNEND+1
780:   C049 90 06             BCC  OK1NEW
790:   C04B 20 79 00 OLDFUN   JSR  CHRGOT     ;REPLACE FLAGS
800:   C04E 4C 8D AE          JMP  FUNCTOLD   ;CALCULATE OLD
                                                FUNCTION
810:                 ;
820:   C051 38       OK1NEW   SEC             ;NEW FUNCTION
830:   C052 E9 E1             SBC  #FUNSTART  ;SUBTRACT
                                                 OFFSET
840:   C054 0A                ASL
850:   C055 48                PHA             ;SAVE POINTER TO
                                               TABLE
860:   C056 20 73 00          JSR  CHRGET     ;GET NEXT
                                               CHARACTER
870:   C059 20 F1 AE          JSR  GETTERM    ;GET FUNCTION
                                                ARGUMENT
880:   C05C 68                PLA
890:   C05D AA                TAX             ;POINTER AS INDEX
900:   C05E B9 72 C0          LDA  FUNTAB,Y   ;LOW ADDRESS
910:   C061 85 55             STA  JUMP+1
920:   C063 B9 73 C0          LDA  FUNTAB+1,Y ;HIGH ADDRESS
```

```
930:   C066 85 56              STA  JUMP+2
940:   C068 20 54 00           JSR  JUMP     ;EXECUTE FUNCTION
950:   C06B 4C 8D AD           JMP  CHECKNUM  ;TEST RESULT
                                               FOR NUMERIC
960:                   ;
970:                   ;
980:   C06E            CMDTAB  .WOR CMD1-1   ;TABLE OF COMMAND
                                              ADDRESSES -1
990:   C06E                    .WOR CMD2-1
1000:                  ;....
1010:  C06E            FUNTAB  .WOR FUN1     ;TABLE OF FUNCTION
                                              ADDRESSES
1020:  C06E                    .WOR FUN2
]C000-C06E
```

If you want to use this routine, you need only place the numbers of the first and last new tokens in lines 300 and 310 and the corresponding numbers for numerical functions in lines 330 and 340. A table is placed at lines 950 on so that the routine knows where the new commands are located. This table contains the address of the routines which execute the commands. Because the routines are called with RTS by first placing the return address on the stack, one is subtracted from the addresses because the return address is automatically incremented by one by the RTS command. This is not necessary for functions which are called using the normal JSR call.

3.5 Operating system vectors

We shall review the important functions which use operating system jump vectors that can be changed. In addition to the hardware vectors IRQ, BRK, and NMI which we have already looked at, we will discuss all of the elementary input/output functions which use these vectors. These functions are addressed over the kernal routines at $FXXX. The following table contains a list of these vectors and the addresses to which these vectors point after power-up.

Vector	Address	Significance
$0314/$0315	$EA31	IRQ vector
$0316/$0317	$FE66	BRK vector
$0318/$0319	$FE47	NMI vector
$031A/$031B	$F34A	OPEN vector
$031C/$031D	$F291	CLOSE vector
$031E/$031F	$F20E	CHKIN vector
$0320/$0321	$F250	CKOUT vector
$0322/$0323	$F333	CLRCH vector
$0324/$0325	$F157	BASIN vector
$0326/$0327	$F1CA	BSOUT vector
$0328/$0329	$F6ED	STOP vector
$032A/$032B	$F13E	GET vector
$032C/$032D	$FE66	warmstart vector (unused)
$032E/$032F	$F4A5	LOAD vector
$0330/$0331	$F5ED	SAVE vector

We will become acquainted with the significance of the vectors and the functions of the routines to which they pertain. With this knowledge we can then write our own input/output functions.

OPEN - JSR $FFC0

This routine performs the same task as the BASIC command by the same name. The parameters used by the equivalent BASIC command must be taken care of before the routine is called. There are two other routines which are used to do this.

SETFLS - JSR $FFBA

This routine sets the parameters for the logical file number, device number, and secondary address. The parameters are passed in the processor registers:

```
      LDA LF ; logical file number
      LDX DN ; device number
      LDY SA ; secondary address
      JSR SETFLS ; set parameters
```

The routine SETNAM - JSR $FFBD exists for passing the filename. You must provide the length as well as the address of the filename. If no filename is used, the length is given as zero.

```
      LDA #NAME1-NAME ; length of the name
      LDX #<NAME ; low byte of the address
      LDY #>NAME ; high byte of the address
      JSR SETNAM ; pass parameters
      ...
NAME .ASC "FILENAME"
NAME1 = * ; end of the name
```

Once these two routines have done their work, the OPEN routine can be called.

```
     JSR OPEN
```

This opens the logical file. The following procedure permits one to recognize any errors which may occur. The carry flag is used as an error flag. If the flag is cleared after the routine call, the routine was executed without error. If an error did occur, however, the carry flag will be set and the accumulator will contain the error number. These error numbers have the following meanings:

No.	Meaning
0	halt via STOP key
1	too many files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing filename
9	illegal device number
240	RS 232 open/close

The carry flag should be tested after a kernal routine call in order to check the error status.

```
     JSR OPEN ; open file
     BCC OK   ; everything OK?
     JMP ERROR
OK   ...
```

The error numbers correspond to the error messages which we are already acquainted with from BASIC. A new error number occurs upon OPEN or CLOSE with device number 2, the RS 232 interface. As you may know, two 256-byte buffers are allocated when an RS 232 channel is opened. These buffers are placed at the top end of the BASIC area. This normally results in the end-of-BASIC being moved from $A000 to $9E00. Since strings are normally placed in this area, this area is no longer available. In order to inform the BASIC interpreter of this situation, the error flag is set and the error number 240 is passed. Upon receipt of this error, the interpreter executes a CLR command, thereby clearing all of the variables. These buffers are freed upon CLOSEing the channel and the variables are again cleared. If you use the RS-232 interface in your BASIC programs, the OPEN command should be one of the first statements in the program and the CLOSE command should be executed last. This ensures that no variables will be lost during the course of the program.

As an alternative, you could also change the OPEN routine. You could simply place the buffers in the area beginning at $C000 when opening the RS-232 interface. This has no effect on the BASIC program area and the CLR command can be dispensed with.

The carry flag is also used as an error flag for the I/O routines that will be discussed shortly and the accumulator also contains the error number.

The operating system even has its own routine for outputting error messages. The output appears in the form

```
I/O ERROR #X
```

in which X is the error number (1 to 9). The program is not stopped when an error is encountered. We can activate the error output by calling the routine SETMSG - JSR $FF90 with a value of $40 in the accumulator (bit 6 set). The error messages can be turned off by calling SETMSG with a value of zero in the accumulator.

An additional function of the routine SETMSG is to distinguish between program mode and the direct mode. Bit 7 is used for this. If bit 7 is cleared, the program mode is designated and status messages of the operating system such as "SEARCHING FOR", "LOADING", and "SAVING" are suppressed.

CLOSE - JSR $FFC3

The CLOSE routine requires only one parameter: the logical file number, passed in the accumulator.

```
LDA LF
JSR CLOSE
```

No error messages can occur when using the CLOSE command. An exception to this is the closing of an RS-232 channel. Here the buffer is freed and the BASIC interpreter executes a CLR command. An attempt to close an unopened file does not result in an error message.

CHKIN - JSR $FFC6

This command serves to redirect the input from the keyboard to an opened file. If you want to read data from the diskette, you must first open the file and then use this file as input with CHKIN. The logical file number must be in the X register for the call.

```
LDX LF
JSR CHKIN
```

Here too, errors are recognized through the set carry flag. If the file was not previously opened, we get "FILE NOT OPEN"; if you try to read a cassette file, a "NOT INPUT FILE" error results. The actual input is performed by the routine BASIN, introduced later.

CKOUT - JSR $FFC9

The routine CKOUT is to output what CHKIN is to input. It allows the output to be redirected to a previously opened file. The CKOUT routine corresponds to the BASIC command CMD. The logical file number is again passed in the X register.

```
LDX LF
JSR CKOUT
```

The possible errors correspond to those for CHKIN. An attempt to write to a tape file results in "NOT OUTPUT FILE." The output is performed with BSOUT.

BASIN - JSR $FFCF

This routine can be compared to the INPUT command in BASIC. If you have not redirected the input with CHKIN, you can get characters from the keyboard or from the screen. If you call BASIN from within a machine language program, the cursor appears on the screen and you can enter characters until you press RETURN. BASIN returns, in the accumulator, the first character entered. Each additional call of BASIN gets an additional character until RETURN (CHR$(13)) is encountered. This allows you to make full use of the screen editor. If, however, you want characters from an opened file, corresponding to the INPUT# command, you must first call CHKIN which redirects input from this file. The BASIN routine then gets a character from this file upon each call and returns this character in the accumulator.

BSOUT - JSR $FFD2

We can output characters with the BSOUT routine. The character in the accumulator will be printed on the screen. For example:

```
LDA #$41
JSR BSOUT
```

This prints the character with the ASCII value $41 or 65 (the letter A) on the screen. You can also output control characters or color codes, exactly as with the BASIC command PRINT CHR$(X);. A new-line, as is possible in BASIC with a PRINT command without a terminating semicolon, must be explicitly specified in machine language.

```
LDA #13 ; carriage return
JSR BSOUT ;output
```

If you do not want to output the characters on the screen, but rather to the printer or to a disk file, you must first open the appropriate file and use the routine CKOUT. This routes the output to the file and all calls of BSOUT output the character not to the screen, but to that file. Error messages such as "DEVICE NOT PRESENT" may occur if the device on the serial bus does not answer.

CLRCH - JSR $FFCC

The routine CLRCH has the opposite function as CHKIN and CKOUT. While these routines redirect input or output to a logical file, CLRCH resets the standard I/O devices--the keyboard and the screen. If you want to get 10 characters from logical file 2 from the disk, the appropriate program fragment looks like this:

```
     LDX #2 ; logical file number
     JSR CHKIN ; input from file #2
     LDY #0
LOOP JSR BASIN ; get character from the disk
     STA STORE,Y ; and store
     INY
     CPY #10 ; 10 characters?
     BNE LOOP ; no
     JSR CLRCH ; back to standard input
```

The logical file 2 must be opened before using this fragment. The input is routed from the file with CHKIN, ten characters are read with BASIN and stored, and the standard input is re-established from the keyboard with CLRCH. The file remains open; closing must be done explicitly with CLOSE.

GET - JSR $FFE4

This routine corresponds to the GET routine of BASIC. You can get a character from the keyboard with it. If no key is pressed at the time the routine is called, a zero is returned, exactly as in BASIC where a null string is returned if no key is pressed. A loop to wait for a keypress is constructed as follows:

```
LOOP JSR GET
     BEQ LOOP
```

The loop waits until a key is pressed. The GET command can also be used on a logical file. As with BASIN, the logical file must first be set with CHKIN. The GET command on a file works the same way as the BASIN routine. After a GET on a logical file a call to CLRCH is necessary in order to reactivate the standard input.

CLALL - JSR $FFE7

This routine performs the same tasks as CLRCH. In addition, however, the number of open files is set to zero. This has the effect of closing all of the files. The corresponding CLOSE routine is not called. A file opened for writing on the disk is not closed properly. This routine is called by the BASIC interpreter for each RUN command.

LOAD - JSR $FFD5

This is the operating system LOAD routine. Before calling this routine, the device number, secondary address, and filename must be set. This can be done with the routines SETFLS and SETNAM which were discussed in connection with the OPEN command. A program can be loaded at the address from which it was saved and which is stored in the disk or datasette file, or it can be loaded at an address passed to the LOAD command, depending on the secondary address. With a secondary address of zero, the file (program) is loaded at the address passed in the X (LSB) and Y (MSB) registers. The contents of the accumulator determines if a load or a verify is to be executed.

```
LDA #0 ; flag for LOAD
LDX #<ADDRESS ; start address
LDY #>ADDRESS
JSR LOAD
STX ENDADDR ; end address LSB
STY ENDADDR+1 ; MSB
```

For the case in which the secondary address is zero, the program is loaded at the address given by ADDRESS. The ending address of the loaded program is returned in the X and Y registers. If the program is not to be loaded but only compared with the program in memory (verified), a 1 must be passed in the accumulator.

```
LDA #1 ; flag for VERIFY
JSR LOAD
```

If the secondary address is one, the file is loaded at the address specified within the file itself and we need not pass the start address in X and Y. For VERIFY, a verify failure is denoted by a status value (STATUS is located at address $90) other than zero. Bit 6 (value 64) must be masked out since this signals the end of the program.

```
      LDA STATUS
      AND #%10111111 ; mask EOF BIT
      BEQ OK
      JMP ERROR
OK    ...
```

SAVE - JSR $FFD8

With the SAVE routine it is possible to save a section of memory to a peripheral device. The device number and the filename must again be previously specified with SETFLS and SETNAM. The routine itself must be given the starting address and ending address+1 of the area to be saved. The ending address plus one must be contained in the X and Y registers. The accumulator must contain a pointer to the zero page address at which the low and high bytes of the starting address are stored. If for example we want to save the area from $1234 to $1FFF, the call looks like this:

```
      LDA #<$1234
      STA START
      LDA #>$1234
      STA START+1
      LDX #<$1FFF+1
      LDY #>$1FFF+1
      LDA #START
      JSR SAVE
```

First the starting address is placed in the zero page locations START and START+1. The ending address plus one is placed in the X (LSB) and Y (MSB) registers and the accumulator is loaded with the address of START. Note that immediate addressing is used because the address itself, not its contents, is intended.

Error messages such as "DEVICE NOT PRESENT" or "MISSING FILENAME" may occur when saving to diskette or "ILLEGAL DEVICE NUMBER" for an attempt to save to the keyboard, screen, or RS-232.

Before we try to write our own input/output routines, we will briefly review the operation of some operating system kernal routines.

OPEN

For the OPEN command the parameters for the logical file number, device number, and secondary address are placed in a table. This table has ten positions. An attempt to open more than 10 files will generate the error message "TOO MANY FILES." The rest of the procedure is dependent on the device number. If the device is the keyboard (0) or the screen (3), any filename is ignored and the routine ends. For the datasette (1) a tape file is opened either for reading (secondary address = 0) or for writing (secondary address = 1) based on the secondary address. Secondary address 2 leads to opening a write file and is handled differently only by the CLOSE command. For reading, the tape file with the filename given in the OPEN command is searched for. If no name is given, the first file found is opened. For writing, a file with the provided name (if any) is opened.

If the device address is 2, RS-232 transmission is prepared. As already mentioned, two 256-byte buffers for input and output are allocated at the upper end of the BASIC storage. The secondary address is ignored. The first two characters of the "filename" are copied to $293 and $294. From these parameters the number of bits per word (5-8) is calculated and stored in $298. The corresponding baud rate values with which the timer in CIA 2 must be loaded are determined from the first character of the filename by means of a table and saved in $295/$296. If the X line handshake was specified, a check is made to see if the signal DSR (Data Set Ready) is present. In the absence of this signal the appropriate bit in the RS-232 status ($297) is set. Otherwise the status is always cleared by the OPEN command.

Device addresses greater than 3 refer to the serial bus. If the secondary address and filename are missing, as with OPEN 1,4 for the printer, only an entry is made in the table. The absence of the secondary address must be made known to the routine SETFLS by using a negative value ($FF) for the secondary address. Otherwise the OPEN command is sent over the serial bus. After the device is addressed with LISTEN, the secondary address plus $F0 is sent. The connected device interprets this as an OPEN command. If a filename was specified, it is sent at the end before the transmission is ended with UNLISTEN.

CLOSE

The CLOSE command ends the transmissions and clears the corresponding table entries in the computer. The rest of the procedure is again determined by the device address. For files on the keyboard and screen, nothing more is done. If a tape file is to be closed, the procedure is further dependent on the secondary address. If the file was opened for reading (secondary address = 0), nothing more need be done. For writing, the current contents of the cassette buffer are written to the tape. For secondary address 2, an EOT

(End Of Tape) block is also written. For an RS-232 transmission, the activities are terminated and the two buffers are deallocated. If a file on the serial bus is to be closed, the computer sends the secondary address (if there was one) plus $E0, which is interpreted as a CLOSE command.

CHKIN

If the input is to be taken from a file, the computer detemines the device number and secondary address from the logical file number and takes additional steps dependent upon this. With the datasette, a check is made to see if the file is a read file (secondary address = 0), otherwise the error message "NOT INPUT FILE" is generated. For devices on the serial bus, a TALK command and then the secondary address are sent. The device is thereby ready to send data. The number of the device from which input is to be expected is stored independent of the device until the normal input is re-enabled with CLRCH.

CKOUT

The CKOUT command functions like the CHKIN command. For the datasette, a check is made to see if the secondary address is greater than zero (otherwise a "NOT OUTPUT FILE" error). A LISTEN command and the secondary address are sent. The connected device is then ready to receive data.

BASIN

Here a character is fetched from the keyboard, the datasette, the RS-232 interface, or the serial bus depending on the active device selected with CHKIN.

BSOUT

This routine sends the character in the accumulator to the device previously determined with CKOUT. The screen serves as the standard device.

CLRCH

The CLRCH command cancels the CHKIN and CKOUT I/O redirections. The values 0 for keyboard input and 3 for screen output are again entered. If devices were active on the serial bus, an UNTALK or UNLISTEN command is sent in order to inform the devices of the end of the transmission.

3.6 Printer spooling

As an example of the use of the input/output vectors of the operating system, we present a routine that emulates a Centronics-compatible interface on the user port and also allows for printer spooling.

Spooling is the outputting of characters to the printer in the "background," while the computer performs other tasks. From this description, it should be quite clear that this must be handled by an interrupt routine. In order that the normal PRINT output not have to wait until the printer is ready for each character, we will write the character in a buffer. The interrupt program checks each time to see if characters are still in the buffer. If this is so and the printer is ready to accept more data, characters are sent until either the printer is no longer ready or there are no more characters left to be sent.

```
100:   CC00                   .OPT P1
110:               ;
120:               ;PRINTER SPOOLING
130:               ;
140:               ;I/O VECTORS
150:   031A        OPEN    =    $31A    ;OPEN VECTOR
160:   031C        CLOSE   =    $31C    ;CLOSE VECTOR
170:   0326        BSOUT   =    $326    ;BSOUT VECTOR
180:               ;
190:   00F7        WPNT    =    $F7     ;WRITE-POINTER
                                         WITHIN BUFFER
200:   00F9        RPNT    =    $F9     ;READ-POINTER
                                         WITHIN BUFFER
210:               ;
```

```
220:   0098        NRFLS    =  $98       ;NUMBER OF OPEN
                                          FILES
230:   00B8        LF       =  $B8       ;LOGICAL FILE
                                          NUMBER
240:   00BA        FA       =  $BA       ;DEVICE ADDRESS
250:   00B9        SA       =  $B9       ;SECONDARY ADDRESS
260:   0259        LFTAB    =  $259      ;TABLE OF LOGICAL
                                          FILE NUMBERS
270:   0263        FATAB    =  LFTAB+10   ;TABLE OF DEVICE
                                           ADDRESSES
280:   026D        SATAB    =  FATAB+10   ;TABLE OF
                                       SECONDARY ADDRESSES
290:   009E        CHAR     =  $9E       ;CHARACTER TO BE
                                          OUTPUT
300:   0001        CONFIG   =  1         ;MEMORY DIVISION
310:   009A        OUTDEV   =  $9A       ;DEVICE NUMBER
                                          FOR OUTPUT
320:   0314        IRQVEC   =  $314      ;IRQ VECTOR
330:   EA31        IRQOLD   =  $EA31     ;OLD IRQ ROUTINE
340:               ;
350:   F34A        OPENOLD  =  $F34A
360:   F1CA        BSOUTOLD =  $F1CA
370:   F31F        SETPARA  =  $F31F
380:   F314        SEARCHLF =  $F314
390:   F30F        SRCHLFX  =  $F30F
400:   F2A1        OLDCLOSE =  $F2A1
410:   F2F1        CONTCLS  =  $F2F1
420:   F6FE        FILEOPEN =  $F6FE
430:   F64B        TOOMANY  =  $F64B
440:   F291        CLOSEOLD =  $F291
450:   DD00        CIA      =  $DD00     ;CIA2
460:   DD00        PORTA    =  CIA       ;PA2 FOR STROBE
470:   DD01        PORTB    =  CIA+1     ;PORT B FOR DATA
```

```
480:   DD03            DIRECTION=    CIA+3    ;DATA DIRECTION
                                               REGISTER
490:   DD0D            ICR       =   CIA+13   ;INTERRUPT CONTROL
                                               REGISTER
500:                   ;
510:   E000            BUFFER    =   $E000    ;PRINTER BUFFER
                                               UNDER KERNAL
520:                   ;
530:   CC00                      *=  $CC00
540:   CC00 A9 0B      INIT      LDA #<OPENNEW
550:   CC02 A0 CC                LDY #>OPENNEW
560:   CC04 8D 1A 03             STA OPEN     ;RESET OPEN VECTOR
570:   CC07 8C 1B 03             STY OPEN+1
580:   CC0A 60                   RTS
590:                   ;
600:   CC0B A6 B8      OPENNEW   LDX LF       ;LOGICAL FILE
                                               NUMBER
610:   CC0D F0 05                BEQ ERROR    ;ZERO NOT ALLOWED
620:   CC0F 20 0F F3             JSR SRCHLFX   ;SEARCH FOR FILE
-                                               DATA
630:   CC12 D0 03                BNE OK2      ;NOT FOUND, OK
640:   CC14 4C FE F6   ERROR     JMP FILEOPEN   ;OTHERWISE 'FILE
                                                 OPEN' ERROR
650:   CC17 A6 98      OK2       LDX NRFLS    ;NUMBER OF OPEN
                                               FILES
660:   CC19 E0 0A                CPX #10
670:   CC1B 90 03                BCC OK       ;LESS THAN 10, OK
680:   CC1D 4C 4B F6             JMP TOOMANY   ;'TOO MANY FILES'
690:   CC20 A5 BA      OK        LDA FA       ;DEVICE NUMBER
700:   CC22 C9 04                CMP #4       ;EQUAL TO 4?
710:   CC24 F0 03                BEQ SPOOL    ;YES, SPOOLING
720:   CC26 4C 4A F3             JMP OPENOLD
730:   CC29 E6 98      SPOOL     INC NRFLS    ;INCREMENT NUMBER
```

```
CC2B 9D 63 02      STA  FATAB,X   ;DEVICE ADDRESS
                                   IN TABLE
CC2E A5 B8         LDA  LF
CC30 9D 59 02      STA  LFTAB,X   ;LOGICAL FILE
                                   NUMBER
CC33 A9 FF         LDA  #-1
CC35 9D 6D 02      STA  SATAB,X   ;NO SECONDARY
                                   ADDRESS
CC38 A9 E0         LDA  #>BUFFER
CC3A 85 F8         STA  WPNT+1   ;WRITE-POINTER
CC3C 85 FA         STA  RPNT+1   ;AND READ-POINTER
CC3E A9 00         LDA  #0       ;TO START BUFFER
CC40 85 F7         STA  WPNT
CC42 85 F9         STA  RPNT
CC44 A9 FF         LDA  #$FF
CC46 8D 03 DD      STA  DIRECTION  ;USER PORT TO
                                    OUTPUT
CC49 AD 00 DD      LDA  PORTA
CC4C 09 04         ORA  #%100    ;STROBE HI
CC4E 8D 00 DD      STA  PORTA
CC51 A9 B5         LDA  #<BSOUTNEW
CC53 A0 CC         LDY  #>BSOUTNEW
CC55 8D 26 03      STA  BSOUT    ;BSOUT VECTOR TO
                                  NEW ROUTINE
CC58 8C 27 03      STY  BSOUT+1
CC5B A9 DD         LDA  #<CLOSENEW
CC5D A0 CC         LDY  #>CLOSENEW
CC5F 8D 1C 03      STA  CLOSE    ;CLOSE VECTOR TO
                                  NEW ROUTINE
CC62 8C 1D 03      STY  CLOSE+1
CC65 A9 73         LDA  #<SPOOLING
CC67 A0 CC         LDY  #>SPOOLING
CC69 78            SEI
```

```
CC6A 8D 14 03           STA  IRQVEC   ;IRQ VECTOR TO
                                       SPOOL ROUTINE
CC6D 8C 15 03           STY  IRQVEC+1
CC70 58                 CLI
CC71 18                 CLC           ;ERASE ERROR FLAG
CC72 60                 RTS
                   ;
CC73 A5 01     SPOOLING LDA  CONFIG
CC75 48                 PHA
CC76 A9 35              LDA  #$35     ;SELECT RAM
CC78 85 01              STA  CONFIG
CC7A A5 F9     TESTNEXT LDA  RPNT     ;COMPARE WRITE
                                       POINTER
CC7C C5 F7              CMP  WPNT     ;WITH READ PRINTER
CC7E D0 06              BNE  SENDCHAR  ;NOT EQUAL, THEN
                                        OUTPUT CHARACTER
CC80 A5 FA              LDA  RPNT+1
CC82 C5 F8              CMP  WPNT+1
CC84 F0 29              BEQ  EXIT
CC86 A9 10     SENDCHAR LDA  #%10000   ;BIT MASK FOR
                                        FLAG
CC88 2C 0D DD           BIT  ICR      ;PRINTER READY?
CC8B F0 22              BEQ  EXIT     ;NO
CC8D A0 00              LDY  #0
CC8F B1 F9              LDA  (RPNT),Y  ;CHARACTER TO
                                         OUTPUT
CC91 8D 01 DD           STA  PORTB    ;GIVE TO PORT
CC94 AD 00 DD           LDA  PORTA
CC97 29 FB              AND  #%11111011  ;STROBE LO
CC99 8D 00 DD           STA  PORTA
CC9C 09 04              ORA  #%000000100  ;AND HI AGAIN
CC9E 8D 00 DD           STA  PORTA
CCA1 E6 F9              INC  RPNT
```

```
1290:  CCA3 D0 D5                BNE  TESTNEXT  ;INCREMENT-READ
                                                 POINTER
1300:  CCA5 E6 FA                INC  RPNT+1
1310:  CCA7 D0 D1                BNE  TESTNEXT
1320:  CCA9 A9 E0                LDA  #>BUFFER
1330:  CCAB 85 FA                STA  RPNT+1
1340:  CCAD D0 CB                BNE  TESTNEXT  ;SEND NEXT
                                                 CHARACTER
1350:                   ;
1360:  CCAF 68          EXIT     PLA
1370:  CCB0 85 01                STA  CONFIG  ;OLD MEMORY
                                               DIVISION
1380:  CCB2 4C 31 EA             JMP  IRQOLD  ;TO ADD IRQ
1390:                   ;
1400:  CCB5 48          BSOUTNEW PHA          ;SAVE CHARACTER
1410:  CCB6 A5 9A                LDA  OUTDEV  ;DEVICE ADDRESS
1420:  CCB8 C9 04                CMP  #4      ;EQUAL TO 4?
1430:  CCBA F0 04                BEQ  OK1     ;YES
1440:  CCBC 68                   PLA
1450:  CCBD 4C CA F1             JMP  BSOUTOLD  ;TO OLD OUTPUT
1460:  CCC0 68          OK1      PLA          ;CHARACTER BACK
1470:  CCC1 85 9E                STA  CHAR    ;AND SAVE
1480:  CCC3 98                   TYA
1490:  CCC4 48                   PHA          ;SAVE Y
1500:  CCC5 A5 9E                LDA  CHAR    ;CHARACTER
1510:  CCC7 A0 00                LDY  #0
1520:  CCC9 91 F7                STA  (WPNT),Y  ;WRITE IN BUFFER
1530:  CCCB E6 F7                INC  WPNT
1540:  CCCD D0 08                BNE  NOINC   ;INCREMENT BUFFER
                                               POINTER
1550:  CCCF E6 F8                INC  WPNT+1
1560:  CCD1 D0 04                BNE  NOINC
1570:  CCD3 A9 E0                LDA  #>BUFFER  ;BUFFER POINTER
```

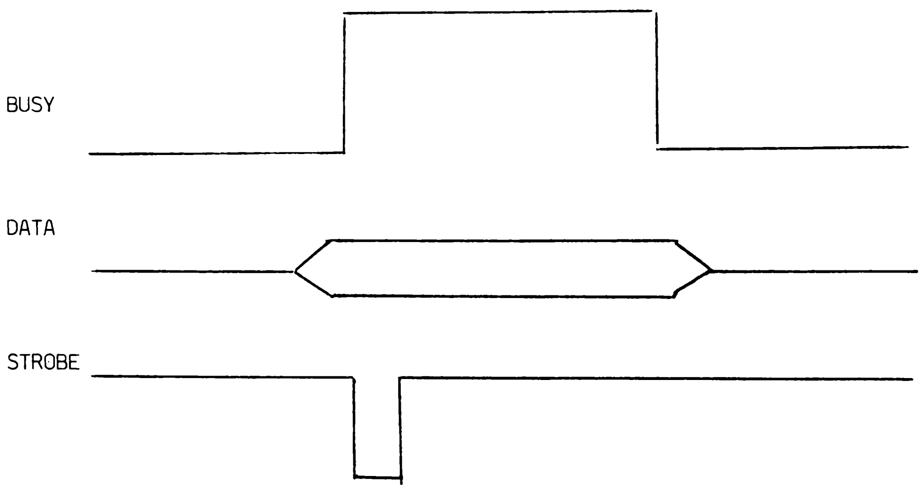
```
                                                    TO START
1580:  CCD5 85 F8              STA  WPNT+1
1590:  CCD7 68        NOINC    PLA
1600:  CCD8 A8                 TAY            ;Y BACK
1610:  CCD9 A5 9E              LDA  CHAR
1620:  CCDB 18        DONE     CLC            ;CLEAR ERROR FLAG
1630:  CCDC 60                 RTS
1640:                 ;
1650:  CCDD 20 14 F3 CLOSENEW JSR  SEARCHLF  ;SEARCH FOR FILE
                                               DATA
1660:  CCE0 D0 F9              BNE  DONE      ;NO FILE OPEN,
                                              DONE
1670:  CCE2 20 1F F3           JSR  SETPARA  ;GET FILE
                                              PARAMETER
1680:  CCE5 8A                 TXA
1690:  CCE6 48                 PHA            ;SAVE X REGISTER
1700:  CCE7 A5 BA              LDA  FA        ;DEVICE ADDRESS
1710:  CCE9 C9 04              CMP  #4        ;4?
1720:  CCEB F0 03              BEQ  CLOSE1
1730:  CCED 4C A1 F2           JMP  OLDCLOSE  ;OLD CLOSE
                                               ROUTINE
1740:  CCF0 A9 CA    CLOSE1   LDA  #<BSOUTOLD
1750:  CCF2 A2 F1              LDX  #>BSOUTOLD
1760:  CCF4 8D 26 03           STA  BSOUT     ;VECTOR TO ADD
                                              BSOUT ROUTINE
1770:  CCF7 8E 27 03           STX  BSOUT+1
1780:  CCFA A9 91              LDA  #<CLOSEOLD
1790:  CCFC A2 F2              LDX  #>CLOSEOLD
1800:  CCFE 8D 1C 03           STA  CLOSE     ;VECTOR TO OLD
                                              CLOSE ROUTINE
1810:  CD01 8E 1D 03           STX  CLOSE+1
1820:  CD04 A9 31              LDA  #<IRQOLD
1830:  CD06 A2 EA              LDX  #>IRQOLD
```

```
CD08 78           SEI
CD09 8D 14 03     STA  IRQVEC   ;REPLACE OLD IRQ
CD0C 8E 15 03     STX  IRQVEC+1
CD0F 58           CLI
CD10 4C F1 F2     JMP  CONTCLS   ;END CLOSE
                                  NORMALLY

]CC00-CD13
NO ERRORS
```

Before we come to the description of the routine, we should first learn something about the operation of the Centronics interface for a better understanding of the printer output.

A Centronics interface is a parallel interface, meaning that 8 bits (a complete byte) are always sent in parallel. In order that the computer and printer be able to agree on the time of the transmission, two "handshake" lines are used. The first line is called STROBE and is controlled by the computer. The line floats high, meaning that it is normally logically high. If the computer wants to send a character to the printer, it places the data on the data lines and signals the printer through a short low impulse on the STROBE line meaning that the data is ready for it. The printer accepts the data and forces the BUSY line high until it has processed the character and is ready to accept the next. Before the computer can send the next character, it must first wait until the BUSY line returns to low. The CIA 2 of the Commodore 64 is used for the interface. Port B, the user port, serves to transmit the data. The STROBE signal goes over the PA2 line (bit 2 of port A) and the BUSY line of the printer is connected to the FLAG line of the user port. Bit 4 in the interrupt control register of the CIA is automatically set by high to low transition. We can therefore recognize exactly when the printer is ready to receive data. The following timing diagram represents the relationship graphically.



Now to the description of our program. After the definition of the addresses we find first the initialization which sets the OPEN vector to our new routine in the usual manner. The routine itself begins in the same way as the operating system routine with the test of the logical file number. If it is zero, we output an error message. Otherwise we search for an open file with this number. If no file with the same number was opened, we can check to see if ten files are already open. If so, then the capacity of the file table is exhausted and we output the error message "TOO MANY FILES." Otherwise we check the device number. If the device number is not four, we jump to the normal OPEN routine.

Otherwise we increment the number of open files and enter the logical file number, device number, and secondary address in the appropriate tables. The buffer pointers are set to the start of the buffer. We use the 8K from $E000 to $FFFF under the operating system as the buffer. Then the user port is switched to output and the STROBE signal is forced high. Now the vectors for BSOUT and CLOSE are set to our new routines. The actual spooling is done during the interrupt; we change the interrupt vector to point to the routine SPOOLING. After that, the carry flag is cleared and we can return with RTS.

The spool routine, which is tied into the system interrupt, first switches the memory configuration to RAM and checks to see if there is a character to output in the buffer. This is the case if the write pointer, which is incremented by the routine BSOUT by each write to the buffer, is not the same as the read pointer. If the printer is now ready to accept characters, we get a byte from the buffer and place it on the user port. We notify the printer that we have sent it a valid character by toggling the STROBE line to low and back to high again. Now we increment the read pointer so that the next character can be sent from the buffer.

We now branch to the start of the routine and output the next character. The loop is executed until either no characters are left to be sent or the printer is no longer ready to accept them. At the label EXIT, the normal memory configuration is switched back on and the normal interrupt routine is executed.

The routine BSOUTNEW tests to see if the output is going to device 4. In this case, the character is written into the buffer and the buffer pointer is incremented. The routine does not destroy any register contents and is exited with the cleared carry flag to indicate that no errors occurred.

In CLOSENEW, the vectors for BSOUT and CLOSE are reset to the original addresses if the device address four is recognized. The interrupt vector is also set to its old value. The output of any characters still in the buffer is terminated. A loop must be inserted which waits until the buffer pointers for reading and writing are the same in order to avoid this.

A cable is necessary to connect the Commodore 64 user port to the Centronics interface of the printer. The following lines must be connected:

USER PORT	-	CENTRONICS
A	GND	16
B	FLAG-BUSY	11
C	D0	2
D	D1	3
E	D2	4
F	D3	5
H	D4	6
J	D5	7
K	D6	8
L	D7	9
M	PA2-STROBE	1

Since most printers with a Centronics interface use the ASCII character set which is different from the Commodore 64's character set, the output can also include a conversion to ASCII codes.

The following must be noted when starting. Connect the printer and the computer with the cable and turn on first the computer and then the printer. This guarantees that the printer will be in the READY condition and will set the FLAG bit in the CIA. Now you can load the machine language program and initialize it with SYS 52224. After OPEN 1,4, all data sent via PRINT#1 are written to the buffer, whose contents are sent to the printer in the interrupt routine. Writing to the buffer is done very quickly so that your application program does not have to wait for the printer.

3.7 Table of BASIC keywords and their tokens

Token		Command	Address	Token		Command	Address
$80	128	END	$A831	$9F	159	OPEN	$E1BE
$81	129	FOR	$A742	$A0	160	CLOSE	$E1C7
$82	130	NEXT	$AD1E	$A1	161	GET	$AB7B
$83	131	DATA	$A8F8	$A2	162	NEW	$A642
$84	132	INPUT#	$ABA5	$A3	163	TAB(	-
$85	133	INPUT	$ABBF	$A4	164	TO	-
$86	134	DIM	$B081	$A5	165	FN	-
$87	135	READ	$AC06	$A6	166	SPC(	-
$88	136	LET	$A905	$A7	167	THEN	-
$89	137	GOTO	$A8A0	$A8	168	NOT	-
$8A	138	RUN	$A871	$A9	169	STEP	-
$8B	139	IF	$A928	$AA	170	+	$B86A
$8C	140	RESTORE	$A81D	$AB	171	-	$B853
$8D	141	GOSUB	$A883	$AC	172	*	$BA2B
$8E	142	RETURN	$A8D2	$AD	173	/	$BB12
$8F	143	REM	$A93B	$AE	174	^	$BF7B
$90	144	STOP	$A82F	$AF	175	AND	$AFE9
$91	145	ON	$A94B	$B0	176	OR	$AFE6
$92	146	WAIT	$B82D	$B1	177	>	-
$93	147	LOAD	$E168	$B2	178	=	-
$94	148	SAVE	$E156	$B3	179	<	-
$95	149	VERIFY	$E165	$B4	180	SGN	$BC39
$96	150	DEF	$B3B3	$B5	181	INT	$BCCC
$97	151	POKE	$B824	$B6	182	ABS	$BC58
$98	152	PRINT#	$AA80	$B7	183	USR	$0310
$99	153	PRINT	$AAA0	$B8	184	FRE	$B37D
$9A	154	CONT	$A69C	$B9	185	POS	$B39E
$9B	155	LIST	$A69C	$BA	186	SQR	$BF71
$9C	156	CLR	$A65E	$BB	187	RND	$E097
$9D	157	CMD	$AA86	$BC	188	LOG	$B9EA
$9E	158	SYS	$E12A	$BD	189	EXP	$BFED

Token		Command	Address
$BE	190	COS	$E264
$BF	191	SIN	$E26B
$C0	192	TAN	$E2B4
$C1	193	ATN	$E30E
$C2	194	PEEK	$B80D
$C3	195	LEN	$B77C
$C4	196	STR$	$B465
$C5	197	VAL	$B7AD
$C6	198	ASC	$B78B
$C7	199	CHR$	$B6EC
$C8	200	LEFT$	$B700
$C9	201	RIGHT$	$B72C
$CA	202	MID$	$B737
$CB	203	GO	-

The table is constructed such that the command words come first ($80-$A2), then the special words which are used in combination with other commands ($A3-$A9). The operators are next ($AA-$B0), followed by the comparison operators ($B1-$B3) and the BASIC functions ($B4-$CA). The code for GO which allows GOTO to be written as GO TO concludes the table. Behind the command words are the addresses of the corresponding routines in ROM, whenever possible.

FIRST PUBLISHING LTD