The C128 Basic Interpreter

undefined

undefined

RETURN TO HOME
PAGE
READ INTRO

Translated from German from 128'er, number 1,
with some corrections and additions.
Original text by S. Baloui / ah.
Edited by Rene van Belzen.

To spare you the unnecessary work and sleepless nights, this article will show you how to incorporate Basic interpreter routines into your own machine language programs.

Programming assembly routines for Basic programs often requires to let the Basic interpreter pass parameters, to declare variables and to perform calculations on integer and real numbers. This article will show where these routines are located in the C128, and how to use them.

Pure machine language programs can do without string processing. In a wordprocessor, for instance, the text is represented as a continuous block, and the programmer himself manages the pointer to the instantaneous text position, page start, etc.

More problematic, though, are the subroutines in machine language programs that should replace time-critical Basic lines. Often, in these cases, the result of the machine language routine has to returned in the form of a declared variable to the calling Basic program. If the routine is to be executed with only a small overhead, the Basic program should be able to determine in which variable the result is stored when a "SYS..." command is issued.

In both cases, i.e. for passing variables to the machine language program, and for returning the result in the form of a variable, we need several routines of the Basic interpreter. There are lots of applications where these routines are needed:

a) Sorting routines, which should be able to sort any type of array, and have to be supplied with the array name. Furthermore, sorting routines have to be informed where in the array to start sorting and where to stop, with routines we will discuss further on in this article.

b) The beloved INPUT routines, which should replace the built-in command "INPUT X$". Comfortable INPUT routines, for instance, should be able to accept the following parameters:

- column and line where the input should start
- length of the input field
- characters that are allowed as input
- characters that end the input
- string, in which the input value is to be returned to Basic

An INPUT routine should, of course, not store the input at any location, but, instead, it has to put it into a string, which the Basic program can use for further processing.

c) Search routines, which can search in arbitrary arrays (strings or numerical variables) to find a character string or number, for example, the index data stored in memory for database management. In this case, the assembler routine should process the appropriate array (if possible, even only from a predetermined index, which marks the start of the routine), compare every array field with the search criterium, and, at completion, declare a numerical variable, in which the result is passed, for instance, "X%=5" (the fifth array element corresponds to the search criterium).

Because, recently, I saw myself put before the task to convert these and other routines from C64 into C128 format, and thereby forced to investigate the Basic routines of the C128, it happens to be now that I am able to pass my recently aquired knowledge to you. It should prevent assembly programmers from having sleepless nights, in which he "plows through" the Basic ROM with the aid of the built-in MONITOR.

Before we come to declaring strings and variables, we should remind ourselves that all parameters that the calling Basic program passes should be read from the Basic text. The C128 offers here more comfort than the C64, because the SYS commands already is equipped with the possibility to pass parameters. It has the following format:

SYS (start address), (Accumulator), (X register), (Y register), (Status)

SYS issued this way, can pass one-byte values, with which the Accumulator and the other registers are loaded. In this article the status register will not be used. Please keep in mind, that reading the status register is only possible "with some disruption", whereby several of the flags can be tested, but other bits cannot. I will, therefore, not use more than three values, in the Accumulator, and the X and Y registers.

An example: Enter with the built-in MONITOR:

```
A 0B00 BRK
```

Now, enter in direct mode:

```
SYS DEC("0B00"),1,2,3
```

After the BRK the MONITOR will show the passed values in the registers:

Accumulator: 1
X register: 2
Y-register: 3

You are in no way forced to declare all parameters. If you only want to pass the value 20 to the Accumulator and 53 to the X register, this statement is sufficient:

```
SYS DEC("0B00"),20,53
```

If a register should be "left out", i.e. not be loaded with a defined value, you may simply leave out the appropriate value.

Example: Load Accumulator with 15, and Y register with 87:

```
SYS DEC("0B00"),15,,87
```

For many applications this form of parameter passing is not enough, for instance, for the passing of two-byte values, or a string variable. Even when only one-byte values should be passed, but the number of parameters exceeds three, we need routines to retrieve further values from the Basic text. The first and most simplest of these routines is GETBYT, which retrieves a single byte from the Basic text and puts it in the X register.

Before I describe the routine GETBYT, please enter the following program lines, save the program, and run it:

```
10 POKE DEC("2E"), DEC("20"): POKE DEC("2000"), 0: POKE DEC("30"), DEC("20"): CLR:
NEW
```

Next, go to the built-in MONITOR and enter the following lines:

INITIALIZATION

```
a 00b00  ad 00 ff lda $ff00
a 00b03  48       pha
a 00b04  a9 00    lda #$00
a 00b06  8d 00 ff sta $ff00
a 00b09  ad 06 d5 lda $d506
a 00b0c  09 06    ora #$06
a 00b0e  8d 06 d5 sta $d506
a 00b11  68       pla
a 00b12  8d 00 ff sta $ff00
a 00b15  60       rts
```

Save also this program and execute it with "`SYS DEC("0B00")`".

A problem with programming the C128 in assembler is the bankswitching. The routines of the Basic interpreter switch constantly between bank 0 (Basic text) and bank 1 (variables bank). We put our program in bank 0. When we execute a Basic interpreter routine, and it leaves bank 1 selected, our program residing in bank 0 does not exist, as far as the microprocessor is concerned. The result is an unavoidable crash.

The initialization program defines the range $0000-$1FFF as common memory area. No matter what, whether after a return from an interpreter routine bank 0 or bank 1 is selected, our program will always exist in the instantaneous bank and can therefore continue.

Please get used to, in all cases, first loading and executing this initialization (both the Basic program and the machine language program) after each power-up or reset of the computer. This way, you will avoid any (even after close examination) "unexplainable" crashes, which are not caused by the executed assembler routine, but, in fact, by the bankswitching.

1. CHKKOM ($795C)

Before we, like we intended, retrieve an additional parameter from the Basic text, we first need to examine the CHKKOM routine. It reads the next character of the Basic text (based on the instantaneous state of the program counter of the CHRGET/CHRGOT routine) and increments the program counter. The read character is tested, to see whether or not it is a comma. If it is, everything is in order and a RTS follows. If it happens to be another character, a jump is made into the print error routine, and a "SYNTAX ERROR" is output. Because it is common to use commas as separation marks between several passed parameters (like in the SYS command), it makes sense to use this routine before we execute the GETBYT routine.

a) Read a comma:

```
a 0b20  20 5c 79 jmp $795c
```

After the execution with "`SYS DEC("0B20"),0,0,0,0,`" or "`SYS DEC("0B20"),,,,,`" we get the READY prompt. The SYS statement has read all four parameters, followed by the jump to our own assembly program, which reads an additional comma, and then returned to Basic. An additional RTS was omitted, because it is included at the end of the CHKKOM routine. Please note that before any parameters may be passed, first the maximum of four

parameters that SYS is able to read should be supplied. If you would execute, for instance, "`SYS DEC("0B20"),`" this would lead to a SYNTAX ERROR, because the SYS command expects after a comma the value of the first parameter (Accumulator), which it does not find.

b) Read three commas:

```
a 00b20  20 5c 79 jsr $795c
a 00b23  20 5c 79 jsr $795c
a 00b26  4c 5c 79 jmp $795c
```

Execute with:

```
SYS DEC("0B20"),0,0,0,0,,,
```

or

```
SYS DEC("0B02"),,,,,,,
```

Any other form that deviates from this form (passing of the standard parameters, followed by the appropriate number of commas), will lead to a SYNTAX ERROR. The CHKKOM routine, therefore, is an excellent safety precaution against syntactic errors when the assembly routine is called.

2. GETBYT ($87F4 ($87F1))

GETBYT also uses the SYS command to pass the standard parameters. This routine gets a one-byte value from the Basic text and returns it in the X register. If the read value is outside the allowed range (0-255), an ILLEGAL QUANTITY ERROR is printed.

Accept four one-byte values

```
a 00b30  8d 00 0d sta $0d00
a 00b33  83 01 0d stx $0d01
a 00b36  8c 02 0d sty $0d02
a 00b39  20 5c 79 jsr $795c
a 00b3c  20 f1 87 jsr $87f4
a 00b3f  00       brk
```

After execution with "`SYS DEC("0B30"),1,2,3,0,4`" the four standard parameters are passed. Before we are ready to read the next value, we first have to save the already accepted values. In the example the register values are stored in $0D00-$0D02. Next, the following comma is read by CHKKOM and the one-byte value by GETBYT. After the BRK we see that the X register contains the value 4. If you dump the memory from $0D00 with MONITOR, you can establish that the values one, two, three were indeed stored in a row.

The GETBYT routine has two entry points, $87F4 and $87F1. Until now, only $87F4 was used. The difference between both entry points is, that in $87F1 a CHRGET subroutine is executed first, i.e. the character, to which the program counter points, is read, after which the program counter is incremented.

The entry point at $87F1 is meant for "lazy" programmers. The execution of CHRGET "reads over" the separation mark, avoiding the execution of CHKKOM. Because this separator is by no means checked, I would personally never rely on such an entry point, because if the user has mistyped the comma, he probably mistyped the following one-byte value as well.

For those "byte savers" among you I will present here a small example program that uses this entry point to read

five parameters:

```
a 00b40  8d 00 0d sta $0d00
a 00b43  8e 01 0d stx $0d01
a 00b46  8c 02 0d sty $0d02
a 00b49  20 f1 87 jsr $87f1
a 00b4c  8e 03 0d stx $0d03
a 00b4f  20 f1 87 jsr $87f1
a 00b52  00       brk
```

After the execution with "`SYS DEC("0B40"),1,2,3,0,4,5`" the X register contains the last passed value five, and the previous parameters are contained in $0D00-$0D03.

This example can be expanded as far as you want. The number of parameters that may be passed to an assembly program has no limits. However, you have to save every value before you execute the next CHKKOM and GETBYT.

The next step is to read two-byte values. Such a task is, for instance, necessary for graphics programs, because the resolution in x-direction is larger than 255, or processing arrays, in which the Basic program passes the number of the starting element (which index value might be larger than 255), where the search should begin. The passing of one two-byte value could be implemented by letting the Basic program separate the number into a low and high byte, for example:

```
10 X=1020
20 H=INT(X/256)
30 L=X-H*256
40 SYS DEC("0B00"),L,H
```

As you can see from this example, the SYS command also processes variables, not only directly entered numbers. A problem with the format is, however, that a assembly routine should replace time-critical parts in a Basic program. Would, for instance, this routine be place inside a loop, and be called continuously (with ever changing parameter values), the speed suffers enormously from the cumbersome preparation before the execution.

It is, therefore, advantageous, even with one two-byte value, to use the appropriate Basic interpreter routine. That this routine has to exist, is easily proven by the POKE command (`POKE XXXX,YY`), which indeed uses a two-byte value as a parameter.

3. FRNUM ($77D7)

The routine FRNUM gets a arbitrary numerical expression from the Basic text and converts it into a value. The result is stored in floating point accumulator 1 (FAC #1).

4. ADRFOR ($8815)

ADRFOR converts a floating point (or: real) number stored in FAC #1 into the address format, i.e. into a 16-bit or two-byte value. This address is both stored in registers (Y=low byte/A=high byte) as passed in addresses $16, $17 ($16=low byte, $17=high byte).

If we execute both routines after each another (after the separation mark from the previous parameter is read by CHKKOM), we get as well in the registers as in the memory locations a two-byte value:

```
a 00b60  20 5c 79 jsr $795c
a 00b63  20 d7 77 jsr $77d7
```

```
a 00b66  20 15 88 jsr $8815
a 00b69  00       brk
```

Execute this routine with "`SYS DEC("0B60"),0,0,0,0,1026`" or with "`SYS DEC("0B60"),,,,,1026`". The comma is read first, then the numerical expression is fetched by FRNUM and stored in FAC #1, after that, the contents of FAC #1 is converted into address format by ADRFOR. The Accumulator now contains a four, and the Y register the value two (hex: $0402; decimal: 1026). You will find the same values in $16, $17 in the form: low byte, high byte.

5. GETADR ($880F)

It is self-evident that the routines FRNUM and ADRFOR may be used seperately for other purposes than reading a two-byte value. For the case of the two-byte value it makes more sense, anyway, to use the routine GETADR. This routine executes in sequence CHKKOM, FRNUM and ADRFOR. This makes the previous example much simpler. Please note, that you should no longer retrieve the comma separator with CHKKOM, because it is already called from within GETADR:

```
a 00b70  20 0f 88 jsr $880f
a 00b73  00       brk
```

Execute this routine with "`SYS DEC("0B70"),0,0,0,0,1026`". You get the same values as before both in the registers and the memory locations $16, $17.

6. ADRBYT ($8803)

ADRBYT executes, one after another, the routines ADRBYT, CHKKOM and GETBYT. This routine reads in one go as well a two-byte value, as the next one-byte value, and is, therefore, perfectly suited to execute graphics routines, in which both an X coordinate (16-bit) and an Y coordinate (8-bit) should be passed:

```
a 00b80  20 5e 79 jsr $795c
a 00b83  20 03 88 jsr $8803
a 00b86  00       brk
```

Execution:

```
SYS DEC("0B80"),0,0,0,0,513,15
```

After execution the X register contains the one-byte value 15 (hex: $0f), returned by GETBYT. You will look invain if you try to find the two-byte value 513 in the Y and A registers. The routines CHKKOM and GETBYT that follow ADRBYT will overwrite these registers so, that you may only find the addresse value in memory locations $16, $17 (low/high).

Because I think it is astonishing how much preparation can be avoided if you execute the appropriate routines, I present a small program, that performs the passing of an address (16-bit) and a byte with only CHKKOM, FRNUM, ADRFOR and GETBYT:

```
a 0b30  jsr $795c  ;CHKKOM
a 0b33  jsr $77d7  ;FRNUM
a 0b36  jsr $8815  ;ADRFOR
a 0b39  jsr $795c  ;CHKKOM
a 0b3c  jsr $87f4  ;GETBYTE
a 0b3f  brk
```

If you compare this sequence of calls with the call to ADRBYT, you see that it really pays if you know enough about the different routines that pass numerical parameters, so you are able to pick only the ones you need.

7. GETPOS ($7AAF)

To conclude the different parameter pass routines, I present you with GETPOS, that is by far the most important routine of the Basic interpreter. The routines until now related only to explicit numerical values in the Basic text and to values of numerical variables.

The main difference between these routines and GETPOS is that GETPOS can be used in relation to both numerical (integer or real) and string variables. GETPOS retrieves no value, but delivers a pointer to a variable, or more exactly, a pointer into the variables table.

Every variable is described in this variables table, that in the C128 normally starts from $0400 in bank 1, just like the variables themselves. Every entry ("descriptor block") in this table is for non-dimensioned variables seven bytes long. The first two bytes are both characters that form the variable name, where for:

7.1 Variable Names

1) integer variable names bit 7 of both bytes is 1
2) real variable names bit 7 of both these bytes is 0
3) string variables bit 7 of the first byte is 0 and of the second byte is 1

If a variable name consists only of one letter, the second byte is $00 (or $80 in integers and strings).

The remaining 5 bytes (3-7) have the following meaning:

7.2 Single Variable Values

a) Real: (5 bytes floating point representation)
b) Int: (high byte) (low byte) (3 unused bytes)
c) String: (1string length byte) (2 string address bytes) (2 unused bytes)

Dimensioned arrays are managed differently. The variable name is not located in the variable area, but instead in the special array area. The descriptor block is as follows:

7.3 Array Descriptor Block

1) name (as before, see: 7.1), 2 bytes
2) total length of the descriptor block, 2 bytes (low, high)
3) number of dimensions (n), 1 byte
4) number of array elements in dimension n, 2 bytes (low, high)
5) same in dimension n-1, 2 bytes
...

Only then follow the actual variables themselves:

7.4 Array Variable Values

a) Real: (5 bytes floating point representation)
b) Int: (high byte) (low byte)
c) String: (1 string length byte) (2 string address bytes)

GETPOS supplies in every case a pointer to the first byte of the descriptors belonging to the requested variable, i.e. for non-dimensioned variables a pointer to the first byte of the variable name, for array variables, either a pointer to the first byte of the value itself (integer, real), or to the length of the string variable.

This pointer is returned both in the registers (A=low, Y=high), and in the memory locations $49, $4a (low/high byte).

Because GETPOS processes all variable types, a small demonstration program is enough to demonstrate the access to numerical and string variables, both dimensioned and not dimensioned:

```
a 00b90  20 5c 79 jsr $795c
a 00b93  20 af 7a jsr $7aaf
a 00b96  00       brk
```

The next examples can only be entered in direct mode:

7.5 Not Dimensioned Variables Examples

a) `CLR: A%=10: SYS DEC("0B90"),0,0,0,0,A%`
Go to the MONITOR and enter:

```
M 49 49
```

Result:

```
>00049 02 20 ...
```

The pointer $49, $4A points, therefore, to $2002 (in bank 1, the variable bank). Enter:

```
M 12002 12002
```

Result:

```
>12002 00 0A ...
```

The high byte is located in $2002 (zero), and the low byte in $2003 ($0A=10), together the integer number 10.

b) `CLR: A=10: SYS DEC("0B90"),0,0,0,0,A`

```
M 49 49
```

Result:

```
>00049 02 20 ...
M 12002 12002
```

Result:

```
>12002 84 20 00 00 00 ...
```

(which is the floating point representation of 10).

c) `CLR: A$="BASICINTERPRETER": SYS DEC("0B90"),0,0,0,0,A$`

```
M 49 49
```

Result:

```
>00049 02 20 ...
M 12002 12002
```

Result:

```
>12002 10 EE FE ...
```

In contrast to the numerical variables the pointer in $49, $4A does not point directly to the variable itself, but instead to the first descriptor byte (string length). The next two bytes indicate the address of the string. Therefore, enter:

```
M 1FEEE 1FEFD
```

Result:

```
>1FEEE 42 41 53 ...
```

(= ASCII representation of "BASICINTERPRETER").

7.6 Dimensioned Array Variables Examples

a) `CLR: A%(2)=10: SYS DEC("0B90"),0,0,0,0,A%(2)`

```
M 49 49
```

Result:

```
>00049 0B 20 ...
```

The pointer $49, $4A points, therefore, to $2002 (in bank 1, the variable bank). Enter:

```
M 1200B 1200B
```

Result:

```
>1200B 00 0A ...
```

The high byte is located in $200B (zero), and the low byte in $200C ($0A=10), together the integer number 10.

b) `CLR: A(2)=10: SYS DEC("0B90"),0,0,0,0,A(2)`

```
M 49 49
```

Result:

```
>00049 11 20 ...
M 12011 12011
```

Result:

```
>12011 84 20 00 00 00 ...
```

(which is the floating point representation of 10).

c) `CLR: A$="BASICINTERPRETER": SYS DEC("0B90"),0,0,0,0,A$`

```
M 49 49
```

Result:

```
>00049 0D 20 ...
M 1200D 1200D
```

Result:

```
>1200B 10 EE FE ...
M 1FEEE 1FEFD
```

Result:

```
>1FEEE 42 41 53 ...
```

(= ASCII representation of "BASICINTERPRETER").

As you can see, this routine is extraordinary versatile. Note the following points that apply to single as well as array variables. The pointer in memory locations $49, $4A points for numerical variables to the first byte of the number itself, for string variables to the string length descriptor of the string, followed by the low/high byte of the string address.

It may be of further interest to you that by calling this routine, memory locations $47, $48 communicate the variable name, and that GETPOS creates a variable if it does not yet exist. Therefore, there are no real reasons to avoid using GETPOS in one of your own assembly routines. A crash is not possible, due to the creation of variables!

7.7 End-of-Array Pointer

There is, however, a drawback of this routine compared to its C64 counterpart: GETPOS returns in the C64 a second pointer, which points to the start of an array when using it for dimensioned variables. This pointer may proof to be very useful in search routines, which has to search through an array field for a certain value or string. Such a routine would have to know when the end of the array is reached and the search should be ended. This end can be established through the mentioned pointer, because the array descriptor (amount of memory that the array uses), that is localized within the array description, is added to the array start (= array end).

GETPOS on the C128 does also has this pointer; it is, however, overwritten by other values before the completion of GETPOS. To be able to, even then, fall back to the descriptor of an array, we use a trick. Example: A routine should search a string from element A(53). The array end can be established by the following format:

```
SYS DEC("0BA0"),0,0,0,0,A$(0),A$(53)
```

The executed assembly routine is:

```
a 00ba0  20 5c 79 jsr $795c  ;CHKKOM
a 00ba3  20 af 7a jsr $7aaf  ;GETPOS of A$(0)
a 00ba6  38       sec
a 00ba7  e9 07    sbc #$07   ;subtract 7 of low byte
a 00ba9  85 fb    sta $fb    ;store low byte
a 00bab  b0 01    bcs $0bae  ;underflow?
a 00bad  88       dey        ;decrement high byte
a 00bae  84 fc    sty $fc    ;store high byte
a 00bb0  20 5c 79 jsr $795c  ;CHKKOM
a 00bb3  20 af 7a jsr $7aaf  ;GETPOS of A$(53)
a 00bb6  00       brk
```

This routine first gets the address of the array element A$(0). The pointer, therefore, points to the string length descriptor of the first element. Now, if we subtract from this pointer--that is stored as well in the registers (A=low byte and Y=high byte)--the value seven, the pointer then points to the first byte of the array description, that is, as already mentioned, seven bytes long.

The new pointer is stored in $FB, $FC, and next GETPOS is used to get to the actual start element we were interested in, to retrieve the string length and address.

Because $FB, $FC points to the first element of the array description, it is possible to establish, by adding the number of needed bytes for the array to the array start, resulting in end+1, i.e. the start of the next array. Because the amount of reserved memory is stored in bytes two and three of the array description, it is possible to address those bytes with indirect indexing.

Principle:

```
LDY #2
LDA ($FB),Y; addressing to low byte of required memory locations
INY
LDA ($FB),Y; addressing to high byte of required memory locations
```

8. Creating Every Type of Variable

We now turn our attention to the creation of every type of variable. For this creation we have to operate on the variables table, and--for string variables--operate on the string stack. If you think otherwise, because the variables table normally starts from $0400, and the range $0000-$1FFF was defined as common area by our initialization routine, then please have a look at the memory locations $2F, $2E, the pointer to the start of the variables table. This pointer points to $2000, due to the Basic initialization program. Therefore, the variables table is outside common area. To access the variables table, we first have to switch to bank 1.

8.1 New Initialization Program

If we use the built-in MONITOR, this operation is quite easy, because you simply put a one (bank 1) in front of the address, whereafter the MONITOR switches to the desired bank by itself. In our program we have to switch between bank 0 and 1 "by hand". I present another initialization program, containing two bankswitch routines:

```
a 00b00  ad 00 ff lda $ff00
a 00b03  48       pha
a 00b04  a9 00    lda #$00
a 00b06  8d 00 ff sta $ff00
a 00b09  ad 06 d5 lda $d506
a 00b0c  09 06    ora #$06
a 00b0e  8d 06 d5 sta $d506
a 00b11  68       pla
a 00b12  8d 00 ff sta $ff00
a 00b15  60       rts
a 00b16  08       php
a 00b17  48       pha
a 00b18  a9 00    lda #$00
a 00b1a  8d 00 ff sta $ff00
a 00b1d  68       pla
a 00b1e  28       plp
a 00b1f  60       rts
```

```
a 00b20  08       php
a 00b21  48       pha
a 00b22  a9 7f    lda #$7f
a 00b24  8d 00 ff sta $ff00
a 00b27  68       pla
a 00b28  28       plp
a 00b29  60       rts
```

The improved initialization program contains a routine to switch to bank 0 starting from $0B16, and a second routine to switch to bank 1, starting from $0B20. In both cases the contents of both registers that are modified during execution, Accumulator and Status, are saved and towards the end retrieved. Both routines may, therefore, be called from anywhere within your own routines, without having to be concerned about which actual register contents might be changed--and should, therefore, be saved before execution them.

This saving of registers is in particular important, when you, for instance, want to get a value from the variables table:

```
jsr bank1
lda vartab,x
cmp vartab,y
jsr bank0
```

After switching back to bank 0 all flags that are set by the CMP instruction remain unmodified.

9. Creating an Integer Variable

To create a integer variable you only need the already discussed routines of the Basic interpreter. We assume that our program, for instance, after searching through an array should return to Basic the index of an element that qualifies for the search criterium:

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  20 20 0b jsr $0b20
a 00b39  a0 00    ldy #$00
a 00b3b  a9 02    lda #$02
a 00b3d  91 49    sta ($49),y
a 00b3f  c8       iny
a 00b40  a9 04    lda #$04
a 00b42  91 49    sta ($49),y
a 00b44  20 16 0b jsr $0b16
a 00b47  60       rts
```

Execute this program with:

```
SYS DEC("0B30"),0,0,0,0,I%
```

However, remember to first enter the improved initialization program!

Our routine, now, reads the comma and calls GETPOS, which provides a pointer to the--newly created--variable, i.e. to the first byte of the integer value itself, that, of course, currently is equal to zero. After switching to bank 1, the variables bank, we write in the first byte of the new variable the value $02, in the second byte the value $04. Finally, we switch back to bank 0.

Please enter "`PRINT I%`" after the routine was executed. To anyone that expected a value of 1026, I recommend to

reread the little excursion about variable storage. Integer variables are stored in the unusual "high/low" format. The values $02, $04 that were delivered as a value, therefore, conform with the value $0204, or decimally printed 516.

10. Creating a Real Variable

If we want to return to a calling Basic program a real number result, we create this real variable analogous to what was described earlier. We get the variable address, switch to bank 1, store the real value into the real variable, starting from the address indicated, and switch back again to bank 0.

```
a 00b50  20 5c 79 jsr $795c
a 00b53  20 af 7a jsr $7aaf
a 00b56  20 20 0b jsr $0b20
a 00b59  a0 00    ldy #$00
a 00b5b  a9 84    lda #$84
a 00b5d  91 49    sta $(49),y
a 00b5f  c8       iny
a 00b60  a9 20    lda #$20
a 00b62  91 49    sta $(49),y
a 00b64  a9 00    lda #$00
a 00b66  c8       iny
a 00b67  91 49    sta $(49),y
a 00b69  c8       iny
a 00b6a  91 49    sta $(49),y
a 00b6c  c8       iny
a 00b6d  91 49    sta $(49),y
a 00b6f  20 16 0b jsr $0b16
a 00b72  60       rts
```

Start this program with:

```
SYS DEC("0B50"),0,0,0,0,R
```

Next, enter "PRINT R". We get the decimal value 10, or in floating point representation:

```
84 20 00 00 00
```

As you can see, it is really not complicated to create numerical variables on the C128. The issue here is to switch to bank 1 before you access the variables table. In contrast with the C64, in debugging, you should first examine whether the necessary steps were taken, because, otherwise, even the most thought-through program would simply not run on the C128.

11. The Data Type Flag ($0f/$10)

Before we concern ourselves about the creation of strings, I would like you to pay attention to two important zero page addresses, $0F and $10. When we execute the GETPOS routine, we obtain in these two memory locations information about the nature of the read variables.

As is mentioned in the manual, $0F contains value $00 for numerical variables, and $80 for string variables.

$10 contains in case of an integer variable the value $80, and $00 in case of a real variable.

12. A Small Example: SWAP

I want to demonstrate by a small example how we might use both memory locations. This example performs the same function as the SWAP command in many other Basic interpreters, i.e. the exchange of values between two variables.

Such a command is of paramount importance for sorting routines, in which variables are constantly swapped.

The command:

```
SWAP A$(2),A$(7)
```

is really much faster than the traditional:

```
S$=A$(2):A$(2)=A$(7):A$(7)=S$
```

and the second, even more essential advantage is that for exchanging strings variables, the strings do not have to be recreated at the end of the string stack, but, instead, can be performed by swapping the string descriptors, instead.

The execution of a sorting routine, which uses this kind of command, will, therefore, hardly ever be interrupted by a necessary garbage collection, decreasing the sorting time enormously.

Before you enter the following program lines, please make sure that you have installed the new initialization routine in memory. Our routine will need the subroutines that are located in this routine.

12.1 Get Pointer To the Variables

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  a5 49    lda $49
a 00b38  a6 4a    ldx $4a
a 00b3a  85 fb    sta $fb
a 00b3c  86 fc    stx $fc
a 00b3e  20 5c 79 jsr $795c
a 00b41  20 af 7a jsr $7aaf
```

The first part of the program offers nothing new. After the routine is called by " `SYS DEC("0B30"),0,0,0,0,A$,B$`" the pointer to A$ is fetched with GETPOS. This pointer is copied to $FB, $FC, because, next, we retrieve a pointer to B$, after which the memory locations $49, $4A contain the pointer to the descriptors of this variable.

12.2 Using the Type Flags

We use the type flags according to the mentioned scheme: If $0F is zero, it concerns a numerical variable, and $10 is tested for $00. If this test was positive, a real variable was read, and the Y register is loaded with $04, the start value of a loop, that counts from Y until zero, inclusive, and swaps the separate bytes of both variables (respectively in strings their descriptors).

If $0F was zero, but $10 was, instead, not equal to zero, then the Y register is loaded with $01 (integer variable).

If a string variable was read, we load the Y register with the value $02.

```
a 00b44  a5 0f    lda $0f
a 00b46  d0 0a    bne $0b52
```

```
a 00b48  a5 10    lda $10
a 00b4a  d0 03    bne $0b4f
a 00b4c  a0 04    ldy #$04
a 00b4e  2c a0 01 bit $01a0
a 00b51  2c a0 02 bit $02a0
a 00b54  20 20 0b jsr $0b20
```

This part of the program will be understood by experienced 6502/8502 programmers. It will probably present less experienced programmers with an unsolvable puzzle.

First we look at the contents of $0F. If it is equal to zero, it means a numerical variable was passed, and $10 is tested for zero. If it concerns a real variable, the Y register is loaded immediate with $04.

12.2.1 The Matter With the BIT Instruction

The byte $2C that follows the instruction LDY #$04 is the processor code for the BIT instruction with 16-bit addressing. The next two bytes, therefore, represent the address value.

The BIT instruction has no effect on the Y register. Because the next byte is another $2C, it is interpreted as a BIT instruction too.

Strangely enough, our program jumps "in between" the BIT instruction if an integer variable was passed (a 00b4a d0 03 bne $0b4f). Enter with the MONITOR:

```
d 0b4f
```

The two bytes that follow the BIT code are interpreted as "`LDY #$01`", which means that the Y register--as it should be for integer variables--is loaded with a one.

That is enough about the use of the BIT instruction. That it is commonly used, is illustrated by the frequent use of this method in the C128 ROM.

12.3 Exchanging the Variables

```
a 00b57  b1 49    lda ($49),y
a 00b59  aa       tax
a 00b5a  b1 fb    lda ($fb),y
a 00b5c  92 49    sta ($49),y
a 00b5e  8a       txa
a 00b5f  91 fb    sta ($fb),y
a 00b61  88       dey
a 00b62  10 f3    bpl $0b57
a 00b64  20 16 0b jsr $0b16
a 00b67  60       rts
```

The exchange of both variables is performed analogous to the mentioned Basic statements, however, byte-wise. As a temporary store the X register is used.

Note, that the routine does not check whether both variables are of the same type, which is a primary condition for a sensible exchange.

12.4 Complete Listing of "SWAP"

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  a5 49    lda $49
a 00b38  a6 4a    ldx $4a
a 00b3a  85 fb    sta $fb
a 00b3c  86 fc    stx $fc
a 00b3e  20 5c 79 jsr $795c
a 00b41  20 af 7a jsr $7aaf
a 00b44  a5 0f    lda $0f
a 00b46  d0 0a    bne $0b52
a 00b48  a5 10    lda $10
a 00b4a  d0 03    bne $0b4f
a 00b4c  a0 04    ldy #$04
a 00b4e  2c a0 01 bit $01a0
a 00b51  2c a0 02 bit $02a0
a 00b54  20 20 0b jsr $0b20
a 00b57  b1 49    lda ($49),y
a 00b59  aa       tax
a 00b5a  b1 fb    lda ($fb),y
a 00b5c  92 49    sta ($49),y
a 00b5e  8a       txa
a 00b5f  91 fb    sta ($fb),y
a 00b61  88       dey
a 00b62  10 f3    bpl $0b57
a 00b64  20 16 0b jsr $0b16
a 00b67  60       rts
```

To test this program, please enter these three lines:

```
A$="C128":B$="C64"
SYS DEC("0B30"),,,,,A$,B$
PRINT A$,B$
```

and you will see on screen the contents of swapped strings:

```
C64     C128
```

13. Creating a String Variable

The program examples until now were kept as simple as possible and should cause no problems in understanding. I will explain the creation of string variables with a more complex example program, an INPUT routine.

For developing this routine many of the already described routines are needed. And what is more, further routines and special zero page addresses will be discussed, whithout which the creation of strings is impossible.

The difficulty level of this artikel is increased by the complex program, though. However, in return, you will have received a runnable input routine--which is always useful--and you will know how to create strings on the C128--which is important in more than one application. Useful applications are, as mentioned:

- sorting routines
- INPUT routines

We can imagine numerous other routines that process strings. A nice exercise would be to create an INPUT# routine, which reads up to 255 arbitrary characters, and, therefore, circumvents the limitations of the normal INPUT# routine.

14. The INPUT routine

The INPUT routine should be able to do the following:

1) Put the cursor at an indicated position before the input starts.
2) During the input allow only characters that appear in a string that was passed as parameter.
3) In principle--as long as point 2 allows it--allow all characters, even those that would lead in the normal INPUT to a "REDO FROM START" error.
4) Copy the input from the screen, and return it into a freely selectable string, supplied by the Basic program.

Because input characters are allowed only if they are contained in a passed string, it is necessary to include in this string:

```
CHR$(29)  = cursor right
CHR$(157) = cursor left
CHR$(20)  = delete
CHR$(148) = insert
```

to obtain equal editing properties as in the normal INPUT command.

For example,

```
Z$="1234567890"+CHR$(29)+CHR$(157)+CHR$(20)+CHR$(128)
```

allows digits and editor keys as input.

Because this program is obviously more extensive, and, therefore, more difficult than the program examples so far, I will explain the program execution with an assembly listing. To be able to enter this with the built-in MONITOR you will find the normal disassembled format at the end of the explanation.

14.1 Zero Page Addresses

```
;**ZEROPAGE ADDRESSES**
STREND   = $35     ;POINTER TO END OF STRING STACK
POINTER  = $49     ;POINTER TO DESCRIPTORS
COLUMN   = $EC     ;CURSOR COLUMN
LINEPTR  = $E0     ;POINTER: COLUMN 0 OF INSTANT. CURSOR LINE
```

STREND points to the instantaneous end of the string stack, that grows downwards from the top, $FF00, as you may know ($FFFF in C64). POINTER is the pointer returned by GETPOS, that points to the descriptors of a variable read from the Basic text. COLUMN contains--when used in system routines that output characters--the instantaneous cursor column, and LINEPTR points the to start of the current line where the cursor is located. The current cursor position is, therefore, obtained by LINEPTR+COLUMN.

14.2 Own Variables

```
;**OWN VARIABLES**
BANK0    = $0B16
BANK1    = $0B20
```

```
ALOWLEN  = $FB     ;LENGTH OF STRING WITH ALLOWED CHARS
ALOWADR  = $FC     ;POINTER: ADDRESS OF STRING W/ ALLOWED CHARS
STARTPOS = $A3     ;POINTER: START POSITION OF INPUT
LENGTH   = $A5     ;MAXIMAL INPUT LENGTH
```

BANK0 and BANK1 are the entry points of our subroutines for bankswitching. In ALOWLEN and ALOWADR, ALOWADR+1 the descriptors of the string are stored, in which the characters allowed for INPUT were stored. STARTPOS, STARTPOS+1 is the cursor position passed, where the input should begin. In LENGTH we store the maximal input length that was passed by the Basic program as well.

14.3 Used Kernel Routines

These routines should need no further explanation, because they are equivalent to the corresponding routines in the C64, and, because of the Kernel jump table, have the same entry points.

```
;**KERNEL ROUTINES**
BSOUT    = $FFD2   ;OUTPUT CHARACTER IN ACCU
BASIN    = $FFCF   ;INPUT CHARACTER FROM LOG. FILE
CHKKOM   = $795C   ;CHECK FOR COMMA
GETPOS   = $7AAF   ;GET DESCRIPTOR POSITION OF VARIABLE
STRRES   = $9299   ;RESERVE MEMORY
GETIN    = $FFE4   ;GET A CHARACTER FROM KEYBOARD BUFFER
SETCRS   = $FFF0   ;SET CURSOR
OPEN     = $FFC0   ;OPEN FILE
CLOSE    = $FFC3   ;CLOSE FILE
CHKIN    = $FFC6   ;SET INPUT CHANNEL TO LOGICAL FILE
CLRCH    = $FFCC   ;SET INPUT TO KEYBOARD, OUTPUT TO SCREEN
SETFLS   = $FFBA   ;SET FILE PARAMETERS
SETNAM   = $FFBD   ;SET FILE NAME
.ORG $0D00         ;PROGRAM START (OR: *= $0D00, OR: .BA $0D00)
```

14.4 Save Parameters/Set Cursor

Our routine should be executed with the following Basic statement:

```
SYS DEC("0D00"), length, line, column, 0, allowstring, returnstring
```

For example:

```
SYS DEC("0D00"),20,10,5,0,A$,R$
```

means: the maximal input length is 20 characters, the input should start at column 5 of line 10, only the characters contained in A$ are allowed, and the input itself should be returned in R$ (do not forget in the definition of A$ to include the editor keys).

The parameters length/line/column that are passed to the register must be saved for later use:

```
;**SAVE PARAMETERS**
STA LENGTH
TXA
PHA
TYA
PHA
```

The cursor is now set at the line and column of the input start:

```
;**SET CURSOR**
CLC
JSR SETCRS  ;SET CURSOR
```

14.5 Get Descriptors

The next step is to get the descriptors of the string with the characters that are allowed for input:

```
;**GET DESCRIPTORS ALLOW STRING**
       JSR CHKKOM
       JSR GETPOS
       JSR BANK1
       LDY # 2
GETIT  LDA (POINTER),Y
       STA ALOWLEN,Y
       DEY
       BPL GETIT
       JSR BANK0
```

First the comma after the standard parameters is read, next, GETPOS is executed, that supplies in memory locations $49, $4A (=POINTER) a pointer to the following string (in the example A$), or creates this string if it did not yet exist.

We are reading one after another all three descriptor bytes (string length, low and high byte of the string address) and pass them into ALOWLEN, and ALOWLEN+1, ALOWLEN+2, or: ALOWADR, ALOWADR+1 (ALOWLEN+1=ALOWADR, ALENTH+2=ALOWADR+1, see: 14.2).

Please, take notice of the fact, that the C128-specific feature of bankswitching has to be taken into account! For reading the descriptors we access the variables table in bank 1, and should call bank 1 explicitely, in other words, use our subroutine to switch this bank. Because after return from GETPOS bank 0 is selected, we would otherwise read the appropriate addresses in bank 0. After this routine ends, we switch back to standard bank 0.

14.6 Input Loop/Invert Character Under Cursor

The problem of the input loop is explained easily: It calls GETIN, analogous to the Basic GET, i.e. reading a key from the keyboard (buffer), not waiting for a key to be pressed. The character is passed into the Accumulator. The input loop is exited only then, if the contents of the Accumulator is not equal to zero, i.e. only if a key was pressed.

```
;**INPUT LOOP**
GET   JSR INVERT   ;INVERT CHARACTER
GET1  JSR GETIN
      BEQ GET
      JSR INVERT
```

INVERT represents a subroutine that inverts the character at the instantaneous cursor position. Before the input, the momentary character is displayed in reverse video by executing INVERT, and after the input, it is inverted back, i.e. displayed in normal video. This routine is necessary, because GETIN, just like GET, does not produce a blinking cursor. INVERT does not display a blinking cursor, though, but at least a fixed cursor, that shows the user the instantaneous cursor position.

```
;**INVERT CHAR UNDER CURSOR**
```

```
INVERT  PHA
        LDY COLUMN
        LDA (LINEPTR),Y
        EOR # 128
        STA (LINEPTR),Y
        PLA
        RTS
```

Please notice, that the character passed in the Accumulator should not modified by INVERT, and is, therefore, saved on stack, and retrieved from stack before return. In the actual program, INVERT does not follow the input loop immediately, but, instead, is located at the end of the program.

14.7 Testing the Characters

After a character is input it should be tested whether it concerns a Carriage Return (RETURN, code $0D). If yes, then the input has ended and the string with input characters is stored (routine RETURN).

```
;**RETURN?**
CMP #$0D
BEQ RETURN
```

If RETURN was not pressed, it should be tested if the input character occurs in the string with allowed characters (in the example: A$) .

```
;**ALLOWED CHARACTER?**
         LDY ALOWLEN
         DEY
COMPARE  JSR BANK1
         CMP (ALOWADR),Y
         JSR BANK0
         BEQ OUTPUT
         DEY
         BPL COMPARE
         BMI GET         ;UNCONDITIONAL JUMP
```

Remember: in ALOWLEN the length of the character string, and in ALOWADR, ALOWADR+1 the descriptor pointer to the string itself is passed. To perform this testing we pick every single character of this allow string in a loop, and compare it with the input character stored in the Accumulator. Before the compare, we switch to bank 1, i.e. the bank that contains the variable table and string stack, and then back to bank 0. This solution is certainly not be the fastest one, however, the execution speed of the INPUT routine is fast enough for even the most rapid among typists.

This solution--which was already used once, i.e. in principle, after a routine has used bank 1, to switch back to standard bank 0--has the advantage of an exactly defined initial condition for routines that follow. There is, therefore, no need to consider which bank has been selected currently, because from the start of each new routine, bank 0 is always selected. Deviating from this "firm" solution, and taking possible error sources for granted, I recommend, though, especially for time-critical programs, like sorting routines.

After this little detour, back to the character testing: if even the last character of the allow string does not compare equal to the input character, the program jumps back to the key entry loop, because the entered key, obviously, was not allowed.

However, is the input character present in the allow string, a jump to character output follows:

;**CHARACTER OUTPUT** OUTPUT JSR BSOUT JMP GET

BSOUT outputs the character contained in the Accumulator. After the output, the return to the input loop follows.

14.8 Returning the Input Characters

Now--for you--the most interesting parts of the program follow. Until now, only earlier mentioned routines of the Basic interpreter were used. Novel was only the practical application to switch between banks 0 and 1. Next, you will learn how a string is created on the C128, and which routines and zero page addresses you will need. Transfer of the input characters takes place in several steps:
1) Set cursor at its start position for input, and calculate the pointer to the start position.
2) Determine the actual input length, i.e. remove all tailing spaces, and accept only input values which are no longer than the passed maximal input length.
3) Execute CHKKOM and GETPOS, i.e. get the pointer to the descriptors of the string, that is to be returned to Basic.
4) Reserve memory space for the string to be created, or test if enough space is available.
5) If no characters were entered, go to step 9.
6) Open the screen as logical file, and set the input channel to this file.
7) Read the input from screen, and transfer it to the return string.
8) Close file, and set input channel back to the keyboard.
9) Update the descriptors of the created string, and the end of the string stack.

14.8.1 Set Cursor at Start Position/Calculate Pointer

Column and line of the input start were saved on stack at the start of the program. This values are now retrieved, and the cursor is placed at the appropriate position:

```
;**CURSOR AT START POSITION**
RETURN PLA          ;GET SAVED
       TAY          ;START
       PLA          ;POSITION
       TAX
       CLC          ;AND PUT
       JSR SETCRS   ;CURSOR THERE
```

A pointer to this start position is calculated by, as was already mentioned, adding the instantaneous column to the pointer to the actual start of the instantaneous cursor line:

```
;**POINTER TO INPUT POSITION**
LDA LINEPTR+1   ;PRODUCE
STA STARTPOS+1  ;POINTER TO
LDA LINEPTR     ;INPUT START
CLC
ADC COLUMN
STA STARTPOS
BCC STARTP
INC STARTPOS+1
```

After this addition STARTPOS, STARTPOS+1 contains a pointer to the exact start of the input.

14.8.2 Determine Actual Input Length

To demonstrate what I mean by the "actual" input length, look at the following possible input (S=space). Assume that the Basic program passed a maximum length of ten characters:

```
C128SSSSSS.....
Commodore128SSSSSSS.....
```

In the first case only four characters which should be copied were entered. The next spaces, up to the maximal input length, should, however, not be copied in the return string (just like the Basic command INPUT does not). As actual (real) input length four should be returned. In the second example twelve characters were entered. Because the maximal input length is only ten, and more characters should not be copied, the routine should return as actual length ten.

```
;**DETERMINE ACTUAL INPUT LENGTH**
STARTP  LDY LENGTH
        DEY
LACT    LDA (STARTPOS),Y
        CMP # 32
        BNE OKAY         ;REMOVE
        DEY              ;TAILING
        CPY # 255        ;SPACES
        BNE LACT
```

The loop starts with a length-1, i.e. in example Y=9. Therefore, first the nineth character after the start position is selected, i.e. the tenth input character. Now this character is compared with SPACE (=code 32). If it is equal to SPACE, then the preceding character is compared with SPACE, and so on.

At the first "non-SPACE" the comparing is interrupted, and the routine jumps to OKAY. The Y register contains the actual length minus one.

The loop otherwise continues until Y becomes less than zero, i.e. -1, or 255. This means no input character was entered. For those who do not believe that -1 is the same as 255, add 1 to 255 (=256), and put the result in 8 bits. The nineth bit (1) is "chopped off", and 8 zero bits remain, which is equal to zero. Note, that even in the case of an input length of zero, the Y register still contains the length minus one.

14.8.3 Get String descriptors of Return String

Because the Y regiser contains the length minus one, it is first corrected (INY), before the actual length is stored. Next, CHKKOM and GETPOS are executed. This means we obtain in memory locations $49, $4A a pointer to the descriptor of the return string, which should return the input string to Basic.

```
;**PREPARATIONS FOR RETURN STRING**
OKAY  INY
      STY LENGTH  ;SAVE LENGTH
      JSR CHKKOM  ;POINTER ON
      JSR GETPOS  ;RETURN STRING
```

Reserve space for the string to be created (STRRES=$9299):

```
      LDA LENGTH
      JSR STRRES  ;RESERVE SPACE
```

Here the Accumulator is loaded with the input length, and the routine STRRES ($9299) is executed. It is imperative

that this routine is called before creating the string. It performs several tasks:

1) It tests if there is enough memory space available for the string to be created, indicated by its string length in the Accumulator. If not, a garbage collection is executed. If, even after that, there is still not enough memory available, an OUT OF MEMORY ERROR is printed.

2) STREND ($35) represents a counter to the bottom of the string stack. A new string is created at this instantaneous bottom, that is to be decremented by the value of the length of the string to be created. STRRES also takes care of the decrementing of the pointer STREND with the string length passed through the Accumulator. The new value of STREND is the pointer to the start of our string, from which point characters will be stored.

14.8.4 Do Not Read Characters From Screen If Nothing Was Entered

In the special case that no characters were entered, with forego the logical file, because we do not have to read any characters from the screen.

```
;**TEST FOR ZERO INPUT STRING LENGTH**
      LDA LENGTH  ;WAS THE STRING EMPTY?
      BEQ UPDATE  ;YES, THEN ONLY UPDATE DESCRIPTORS
```

14.8.5 Open Screen As Logical File/Set Input Channel To Logical File

It seems rather cumbersome to open the screen as a logical file and read the input characters. Would, however, the same method be used as in most C64 INPUT routines, i.e. read characters from screen without the help of operating system routines, this would require in the C128 quite a lot more effort: the screen characters should be converted into ASCII before they are stored in the string stack.

In the C64 this would require a very simple conversion routine. The C128 posesses several character sets, though, with (in the foreign version of the C128--Editor) special accent signs, special maths characters, etc., which would lead to an extremely complex and cumbersome conversion. Therefore, it much simple in the C128, whenever possible, to use the operating system routines, that perform this conversion automatically.

```
;**SCREEN: OPEN LOGICAL FILE**
      LDA # 3
      TAX
      TAY
      JSR SETLFS
      LDA # 0
      JSR SETNAM
      JSR OPEN
      LDX # 3
      JSR CHKIN
```

The screen is defined as file with logical file number and secondary address three, the length of the file name is set to zero, the file is opened, and the input channel is set to this logical file. Every call to the BASIN routine, which is used to read, therefore, reads characters from the screen.

14.8.6 Read Input/Create String

BASIN reads the character at the instantaneous cursor position and returns it to the Accumulator. BASIN is called in a loop, and the read character is stored at the updated bottom of the string stack (STREND). Before storing, bank 1

is selected, and afterwards bank 0. Please note, that switching to bank 1 once does not work, because BASIN, that accesses the screen, is located in bank 0, and therefore this bank must be selected before BASIN is called.

```
;**READ/CREATE STRING**
      LDY # 0
READ  JSR BASIN
      JSR BANK1
      STA (STREND),Y
      JSR BANK0
      INY
      CPY LENGTH
      BNE READ
```

14.8.7 Close File/Set Input Channel To Standard

By CLRCH input is restored to the standard device, the keyboard, and, next, the opened file is closed with CLOSE.

```
;**CLOSE FILE/STANDARD INPUT**
      JSR CLRCH
      LDA # 3
      JSR CLOSE
```

14.8.8 Update Descriptors

Suppose, Basic supplied R$ as return string. The string descriptors of R$ point unmodified to the old address and contains its old length. We should, therefore, update these descriptors, i.e. the length descriptor with the actual length (LENGTH) of the input string, and the address with the address we acquired through the STRRES routine, STREND, which is the bottom of the string stack, exactly where we stored our string.

```
;**UPDATE DESCRIPTORS**
        JSR BANK1
DESUPD  LDY # 0
        LDA LENGTH
        STA (POINTER),Y
DUPD    LDA STREND,Y
        INY
        STA (POINTER),Y
        CPY # 2
        BNE DUPD
        JSR BANK0
        RTS             ;RETURN TO BASIC
```

The input is completely copied and the program returns to Basic. As a reminder, I repeat for you the syntax of the call:

```
SYS DEC("0D00"), length, line, column, 0, allow$, return$
```

14.9 Complete Listing of "INPUT"

Here is the complete listing of the INPUT routine. Keep in mind that this routine assumes the 40-columns screen to be the active screen.

To test the routine, please enter in Basic (in direct mode, or in program lines, whatever you prefer):

```
A$="1234567890"+CHR$(29)+CHR$(157)+CHR$(20)+CHR$(148)
SYS DEC("0D00"),10,20,5,0,A$,R$
```

The cursor is positioned on column five of line 20; you may enter as many digits as you want, and edit the entry with CURSOR-LEFT, CURSOR-RIGHT, DEL and INST. After RETURN no more than ten digits are returned in variable R$.

```
a 00d00  85 a5    sta $a5
a 00d02  8a       txa
a 00d03  48       pha
a 00d04  98       tya
a 00d05  48       pha
a 00d06  18       clc
a 00d07  20 f0 ff jsr $fff0
a 00d0a  20 5c 79 jsr $795c
a 00d0d  20 af 7a jsr $7aaf
a 00d10  20 20 0b jsr $0b20
a 00d13  a0 02    ldy #$02
a 00d15  b1 49    lda ($49),y
a 00d17  99 fb 00 sta $00fb,y
a 00d1a  88       dey
a 00d1b  10 f8    bpl $0d15
a 00d1d  20 16 0b jsr $0b16
a 00d20  20 c3 0d jsr $0dc3
a 00d23  20 e4 ff jsr $ffe4
a 00d26  f0 fb    beq $0d23
a 00d28  20 c3 0d jsr $0dc3
a 00d2b  c9 0d    cmp #$0d
a 00d2d  f0 18    beq $0d47
a 00d2f  a4 fb    ldy $fb
a 00d31  88       dey
a 00d32  20 20 0b jsr $0b20
a 00d35  d1 fc    cmp ($fc),y
a 00d37  20 16 0b jsr $0b16
a 00d3a  f0 05    beq $0d41
a 00d3c  88       dey
a 00d3d  10 f3    bpl $0d32
a 00d3f  30 df    bmi $0d20
a 00d41  20 d2 ff jsr $ffd2
a 00d44  4c 20 0d jmp $0d20
a 00d47  68       pla
a 00d48  a8       tay
a 00d49  68       pla
a 00d4a  aa       tax
a 00d4b  18       clc
a 00d4c  20 f0 ff jsr $fff0
a 00d4f  a5 e1    lda $e1
a 00d51  85 a4    sta $a4
a 00d53  a5 e0    lda $e0
```

```
a 00d55  18       clc
a 00d56  65 ec    adc $ec
a 00d58  85 a3    sta $a3
a 00d5a  90 02    bcc $0d5e
a 00d5c  e6 a4    inc $a4
a 00d5e  a4 a5    ldy $a5
a 00d60  88       dey
a 00d61  b1 a3    lda ($a3),y
a 00d63  c9 20    cmp #$20
a 00d65  d0 05    bne $0d6c
a 00d67  88       dey
a 00d68  c0 ff    cpy #$ff
a 00d6a  d0 f5    bne $0d61
a 00d6c  c8       iny
a 00d6d  84 a5    sty $a5
a 00d6f  20 5c 79 jsr $795c
a 00d72  20 af 7a jsr $7aaf
a 00d75  a5 a5    lda $a5
a 00d77  20 99 92 jsr $9299
a 00d7a  a5 a5    lda $a5
a 00d7c  f0 2e    beq $0dac
a 00d7e  a9 03    lda #$03
a 00d80  aa       tax
a 00d81  a8       tay
a 00d82  20 ba ff jsr $ffba
a 00d85  a9 00    lda #$00
a 00d87  20 bd ff jsr $ffbd
a 00d8a  20 c0 ff jsr $ffc0
a 00d8d  a2 03    ldx #$03
a 00d8f  20 c6 ff jsr $ffc6
a 00d92  a0 00    ldy #$00
a 00d94  20 cf ff jsr $ffcf
a 00d97  20 20 0b jsr $0b20
a 00d9a  91 35    sta ($35),y
a 00d9c  20 16 0b jsr $0b16
a 00d9f  c8       iny
a 00da0  c4 a5    cpy $a5
a 00da2  d0 f0    bne $0d94
a 00da4  20 cc ff jsr $ffcc
a 00da7  a9 03    lda #$03
a 00da9  20 c3 ff jsr $ffc3
a 00dac  20 20 0b jsr $0b20
a 00daf  a0 00    ldy #$00
a 00db1  a5 a5    lda $a5
a 00db3  91 49    sta ($49),y
a 00db5  b9 35 00 lda $0035,y
a 00db8  c8       iny
a 00db9  91 49    sta ($49),y
a 00dbb  c0 02    cpy #$02
a 00dbd  d0 f6    bne $0db5
```

```
a 00dbf  20 16 0b jsr $0b16
a 00dc2  60       rts
a 00dc3  48       pha
a 00dc4  a4 ec    ldy $ec
a 00dc6  b1 e0    lda ($e0),y
a 00dc8  49 80    eor #$80
a 00dca  91 e0    sta ($e0),y
a 00dcc  68       pla
a 00dcd  60       rts
```

14.10 Some last remarks

To conclude the discussion of INPUT, here are some last remarks.

As you know, GETPOS can process several types of variables. Therefore, you can also supply array strings, as allow and return strings.

The description of this INPUT routine is now complete. You are able to retrieve any parameter from a Basic text, and also create any kind of variable from machine language. As a test, you could try to dress up this INPUT routine with more comfort.

A clever INPUT routine--in contrast to this routine, which only shows an alternative to the regular INPUT routine--would not allow cursor movements that would move the cursor outside the input field. If, for instance, the maximal input length is ten characters, the cursor should be allowed to move only between the input start position and the next nine columns.

Furthermore, INSERT and DELETE should only move the characters inside the input field, without affecting the remaining characters on that line. It should be also possible to define other characters than only RETURN to terminate the input.

(*Editors' Notes:* Even more functionality would be added if keys could be defined that would leave the original value assigned to the return string intact, e.g. SHIFT-RETURN in the normal INPUT routine, and if you could supply a standard value for the input field. The ultimate input routine should also contain a "clear field" feature, which at the initialization of the INPUT routine fills the entire input field with a predetermined character (e.g. SPACE). Of course, certain keys or key combination should be able to invoke this field filling feature "on the fly", i.e. by the user. To make the routine even more universal, the program should take into account the 80-column screen, and use a blinking cursor, instead of a fixed cursor. All this would require more detailed knowledge of how the C128 system routines use the zero page.)

15. Arithmetical Routines

Not only the variable and string routines of the Basic interpreter, also arithmetical routine are often interesting. For me personally, it is very pleasant to use for multiplying and dividing in machine language. Because the corresponding interpreter routines are not very fast, is may be necessary, for time-critical parts in your program, to write routines that fit your particular problem.

In less time-critical parts, however, the already available routine could be used. The arithmetical routines of the interpreter normally process real numbers. Machine language-only programs will often need only calculations on integer values. Even in these cases, the interpreter routines can be used, because, next, we will learn how to use routines that convert integer values into real values, and other routines that, after execution, change those values back into integer values.

Besides for these purely arithmetical routines, it is useful for many other routines to print a real number to screen. Please consider what it would it take, if we would, for instance, constantly print the column and line numbers at a certain position on the screen "manually". Another example could be the instantaneously reached score in a video game.

Because I have programmed both instances, before I knew the appropriate interpreter routines, I know how to appreciate those routine all the more. You could, one day, also be faced with a similar problem.

All arithmetical routines use the so-called "floating point accumulators" (FAC1 and FAC2). FAC1 is located in the C128 from address $63 to $69, and FAC2 from address $6a to $71. Often FAC1 is referred to as "FAC" and FAC2 as "ARG", because the latter is used as argument in operations with two floating point values. From now on, I will use these references.

I do not want to elaborate on the floating point format in this article, because exact knowledge is not necessary for the application of arithmetical routines, and because--I must confess--I am not very familiar with the subject.

It is important for us to recognize that both floating point accumulators are constructed equally. For our application examples we need the following addresses:

```
$63     = FAC, exponent
$64-$67 = FAC, mantissa
$68     = FAC, sign

$6A     = ARG, exponent
$6B-$6E = ARG, mantissa
$6F     = ARG, sign
```

In the next examples I will use three floating point numbers, on which the program example will operate:

```
dec.   hex     floating point format
   8   $08     84 80 00 00 00
  10   $0A     84 A0 00 00 00
4098   $1002   8D 80 00 00 00
```

(*Editors' note:* A careful reader might have noticed that the floating point representation of the number 10, here, does not exactly correspond with the earlier mentioned representation in variables, see: 7.5 (b) and 7.6 (b). Do not worry about this too much.

For those who are really interested: Actually, the representation in FAC and ARG is the "official" notation. A floating number is represented by an exponent (e), a mantissa (m) and a sign (s) in a 6-byte notation:
`eeeeeeee mmmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm ssssssss`
If a value is zero, all 48 bits are zero (0). For non-zero values the mantissa represents a fractional number between 0.1 (inclusive) and 1 (not inclusive). This means the first bit should always be one, if the number is not zero. The exponent can be either positive or negative and represents a power of 2. The first exponent bit represents the sign bit, where 0 means negative, and 1 means positive. The sign byte can be either all zeros, or all ones ($00 or $FF). So a non-zero real number is represented in FAC and ARG like this:
`eeeeeeee 1mmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm ssssssss`
In variables this notation is compacted. Because in non-zero values the first mantissa bit is always set in the "official" notation, we can use this bit to contain the sign of the number. The compacted notation then becomes:
`eeeeeeee smmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm`)

To explain the different arithmetical routines it would be enough to load FAC and ARG with the appropriate values with the built-in MONITOR, call the arithmetical routines and view the results with the same MONITOR. After I used

this method and got completely senseless results, I came to the conclusion that MONITOR uses FAC for its own purposes (for the pointer at the instantaneous addresses for assemble/disassemble).

Therefore, from now on, we are forced to load the floating point accumulator with a program, and also to save the result elsewhere, before we may use the MONITOR.

Please execute also this time the necessary initialization, if you had switched off or resetted the computer after the last examples. You may use the initialization program without the bankswitching subroutines, because we no longer need to access bank 1.

15.1 Save FAC Contents

The following machine language routine saves the 6-byte contents of the floating point accumulator number 1, which is modified by entering MONITOR:

```
a 00b30  a2 05    ldx #$05
a 00b32  b5 63    lda $63,x
a 00b34  9d 00 0d sta $0d00,x
a 00b37  ca       dex
a 00b38  10 f8    bpl $0b32
a 00b3a  60       rts
```

15.2 Convert Into Integer ($8CC7)

This routine converts a floating point value, located in FAC, into a two-byte integer number, that is returned in FAC (in $66/$67 = high/low):

```
a 00b40  a2 04    ldx #$05
a 00b42  bd 50 0b lda $0b50,x
a 00b45  95 63    sta $63,x
a 00b47  ca       dex
a 00b48  10 f8    bpl $0b42
a 00b4a  20 c7 8c jsr $8cc7
a 00b4d  4c 30 0b jmp $0b30
a 00b50  84 a0    sty $a0
a 00b52  00       brk
a 00b53  00       brk
a 00b54  00       brk
a 00b55  00       brk
```

Because the end of this listing might appear a bit unusual, I want to explain the last instructions in more detail: To convert an integer number into a floating point (real) number, first of all, it has to be stored in FAC1. In this example, I used the decimal number ten, which is located at the end of the program, in floating point format:

```
m b50 b55
>00b50 84 a0 00 00 00 00 ...
```

The program transfers this value to FAC1. After executing routine $8CC7, the corresponding integer number is located in the addresses $66, $67 (high/low), and is copied by the subroutine $0B30 to $0D00-$0D05, to avoid overwriting by the MONITOR.

Execute the routine with "`SYS DEC("0B40")`" and examine $0D00... with the MONITOR:

```
m d00 d05
>00d00 84 a0 00 00 0a 00 ...
```

The integer number $000A is located in $0D03 and $0D04 (fourth and fifth byte of the memory dump).

15.3 Convert 2-Byte Integer (with sign) Into Floating Point Format ($8C70)

There are two methods to represent integer values: with sign, or positive. As an assembly language programmer you probably would normally need only the positive representation (without sign), in wich all 16 bits are used to represent the absolute value of a number, and ranges between $0000 and $FFFF. In the representation with sign, the value ranges between -$8000 and +$7FFF.

The conversion routine for integers with sign is, for instance, used in the C64 to print FRE(0). In that case, however, it should not have been used, because values larger than $7FFF are printed as negative numbers (this might cause a novice programmer to believe the amount of memory available is negative, which would imply "a shortage"--Editors).

The integer values is passed to this routine through memory addresses $64, $65 (high/low). The X register should be loaded immediate with $90.

```
a 00b60  a9 01    lda #$10
a 00b62  a2 00    ldx #$00
a 00b64  85 64    sta $64
a 00b66  86 65    stx $65
a 00b68  a2 90    ldx #$90
a 00b6a  20 70 8c jsr $8c70
a 00b6d  4c 30 0b jmp $0b30
```

After execution with "`SYS DEC("0B60")`" the passed number +$1000 is stored as real number in FAC1. The routine $0B30 copies it to $0D00-$0D05, which can be examined with the MONITOR.

15.4 Convert 2-Byte Integer (positive) Into Floating Point Format ($8C75)

You will probably use this routine most often, in which the--more common--positive integer value is converted into a real number value.The routine "expects" the integer value to be converted in memory locations $64, $65 and the X register should be loaded immediate with $90 as well. Furthermore, the carry flag should be set before the routine is called.

```
a 00b80  a9 ff    lda #$ff
a 00b82  a2 01    ldx #$01
a 00b84  85 64    sta $64
a 00b86  86 65    stx $65
a 00b88  a2 90    ldx #$90
a 00b8a  38       sec
a 00b8b  20 75 8c jsr $8c75
a 00b8e  4c 30 0b jmp $0b30
```

This demonstration program passes the value $FF01. After execution with "`SYS DEC("0B80")`" the stored real number value can be found in $0D00-$0D05:

```
m d00 d05
>00d00 90 ff 01 00 00 00 ...
```

If you perform calculations in your programs with integer numbers, you will probably ask yourself what could be the advantage of converting integer values into real number values. An example could be to display your integer number on screen. Such a task is only possible by using two other routines, but only then, if the integer number is converted first into a real number:

15.5 Convert FAC1 into ASCII ($8E42)

This routine returns a real number value in FAC1 as a string (ASCII format), in--again--FAC1. This conversion routine is particularly interesting in relation to this routine:

15.6 Print String in FAC1 ($55E2)

$55E2 prints the string that is located in FAC1 to the screen, at the instantaneous cursor location.

Both this routine and the previous (see: 15.5), do not need any more than the stored string, or the stored real number, respectively; no additional input parameters are required.

We now posses sufficient knowledge to display any integer number, as is necessary in wordprocessing, for instance. First we pass a positive integer value through memory locations $64, $65, and call $8C75, wich converts it into floating point format. Next, $8E42 is called, which converts the number into a string, that is printed with the call to $55E2, at the instantaneous cursor location:

```
a 00ba0  a9 ff    lda #$ff
a 00ba2  a2 01    ldx #$01
a 00ba4  85 64    sta $64
a 00ba6  86 65    stx $65
a 00ba8  a2 90    ldx #$90
a 00baa  38       sec
a 00bab  20 75 8c jsr $8c75
a 00bae  20 42 83 jsr $8e42
a 00bb1  4c e2 55 jmp $55e2
```

This demonstration program uses the number $FF01, i.e. 65281 in decimal. When you execute it with "`SYS DEC("0BA0")`", precisely this number is printed on screen. It is very convenient, when using these routines, that although the floating point format is needed, the programmer needs not to be bothered, only to pass an integer value at the appropriate memory location.

(Editors' note: If you replace the "JSR $8C75" with "JSR $8C70"--and also remove the "SEC"--you will see the signed integer result.)

15.7 Print 2-Byte Integer (positive, $8E32)

The conversion and output routines, which were called in sequence, can be used separately for other purposes. To display an integer value on screen, a more suitable routine exists, that starts from $8E32. This routine is fed with the integer value in the Accumulator (low value) and X register (high value). It saves this value in memory locations $64, $65, and calls in sequence the already described (see: 15.4, 15.5, 15.6) routines, that are necessary for printing to screen. The preparations are limited to loading the registers with the integer value. After that, the routine may be called:

```
a 00bc0  a9 ff    lda #$ff
a 00bc2  a2 01    ldx #$01
a 00bc4  4c 32 8e jmp $8e32
```

These three program lines perform the same function as the previous demonstration program (see: 15.6). Together they also print the value $FF01, i.e. the decimal number 65281, to the screen display. The execution is performed by "`SYS DEC("0BC0")`".

16. Calculations With Two Arguments

All routines described next perform calculations on two real numbers (addition, division, etc.), that are located in FAC1 and FAC2, and store the result in FAC1. Because we now know how to convert integer into real numbers, we can use two integers for these calculations as well, by converting those integers into real numbers with the appropriate routines.

This application especially saves us writing routines, if we want to do many multiplications and divisions. In time-critical parts of your program you should, however, certainly avoid floating point arithmetic, because integer numbers have to be converted every time, and because of the rather luxerious floating point routines that immediately follow. In such cases, writing your own integer routines is more becoming.

It is even more pointless to use the routines that have been just introduced for special cases, e.g. the multiplication or division of a (two-byte) integer by two, four, etc., because this can be accomplished with only a few rotate and shift instructions. Therefore, please remember this: In arithmetical operations on integer numbers, which are used most often in assembly language programs, you should use the built-in floating point arithmetic only in cases that are not time-critical!

16.1 FAC = FAC + ARG ($8848)

This routine adds two real numbers that are stored in FAC1 and FAC2.

```
a 00bd0  a2 04    ldx #$05
a 00bd2  bd e8 0b lda $0be8,x
a 00bd5  95 63    sta $63,x
a 00bd7  bd ed 0b lda $0bee,x
a 00bda  95 6a    sta $6a,x
a 00bdc  ca       dex
a 00bdd  10 f3    bpl $0bd2
a 00bdf  20 48 88 jsr $8848
;FAC = FAC + ARG
a 00be2  20 42 83 jsr $8e42
a 00be5  4c e2 55 jmp $55e2
a 00be8  84 80    sty $80
a 00bea  00       brk
a 00beb  00       brk
a 00bec  00       brk
a 00bed  00       brk
a 00bee  8d 80 10 sta $1080
a 00bf1  00       brk
a 00bf2  00       brk
a 00bf3  00       brk
```

Execute this routine with "`SYS DEC("0BD0")`". The real numbers ($08=8 and $1002=4098) are located at the end of the program, and are copied to FAC and ARG byte-by-byte. Next, the addition routine and the routines to convert FAC into ASCII and to print the ASCII string in FAC are executed. We obtain the printed result 4106 = 8 + 4098.

16.2 FAC = ARG - FAC ($8831)

This routine also needs two real numbers to be stored in FAC and ARG. Please note that FAC is subtracted from ARG, and not the other way around. To demonstrate the subtraction, please replace the instruction "`JSR $8848`" with "`JSR $8831`":

```
  ...
  ...
  ...
a 00bdc  ca       dex
a 00bdd  10 f3    bpl $0bd2
a 00bdf  20 48 88 jsr $8831
;FAC = ARG - FAC
a 00be2  20 42 8e jsr $8e42
a 00be5  4c e2 55 jmp $55e2
  ...
  ...
  ...
```

As a result, we obtain 4090 = 4098 - 8.

16.3 FAC = ARG * FAC ($8A27)

The execution of this routine is no different from that of the previous arithmetical routine. Therefore, replace " `JSR $8831`" with "`JSR $8A27`":

```
  ...
  ...
  ...
a 00bdc  ca       dex
a 00bdd  10 f3    bpl $0bd2
a 00bdf  20 48 88 jsr $8a27
;FAC = ARG * FAC
a 00be2  20 42 8e jsr $8e42
a 00be5  4c e2 55 jmp $55e2
  ...
  ...
  ...
```

As a result, we obtain 32784 = 8 * 4098.

16.4 FAC = ARG / FAC ($8B4C)

Now change "`JSR $8A27`" into "`JSR $8B4C`". You obtain the result 512.25 = 4098 / 8. Please, note the sequence of dividend and divisor, which is--analogous to the subtraction--reversed:

```
  ...
  ...
  ...
a 00bdc  ca       dex
a 00bdd  10 f3    bpl $0bd2
a 00bdf  20 48 88 jsr $8b4c
```

```
;FAC = ARG / FAC
a 00be2  20 42 8e jsr $8e42
a 00be5  4c e2 55 jmp $55e2
  ...
  ...
  ...
```

To conclude the discussion of the arithmetical routines, I want to present you with two special routines, which are meant for special cases, need one preparation step less, and execute considerably faster than the routines that perform FAC=FAC*ARG or FAC=ARG/FAC.

16.5 FAC = FAC * 10 ($8B17)

Should your program ever need this operation, this routine will save you a conversion of 10 into floating point value:

```
a 00bd0  a2 04    ldx #$05
a 00bd2  bd e8 0b lda $0be3,x
a 00bd5  95 63    sta $63,x
a 00bd7  ca       dex
a 00bd8  10 f3    bpl $0bd2
a 00bda  20 48 88 jsr $8b17
;FAC = FAC * 10
a 00bdd  20 42 8e jsr $8e42
a 00be0  4c e2 55 jmp $55e2
a 00be3  84 80    sty $80
a 00be5  00       brk
a 00be6  00       brk
a 00be7  00       brk
a 00be8  00       brk
```

The execution with "`SYS DEC("0BD0")`" leads to the output of 80 = 8 * 10.

16.6 FAC = FAC / 10 ($8B38)

```
  ...
  ...
  ...
a 00bd7  ca       dex
a 00bd8  10 f3    bpl $0bd2
a 00bda  20 48 88 jsr $8b38
;FAC = FAC / 10
a 00bdd  20 42 8e jsr $8e42
a 00be0  4c e2 55 jmp $55e2
  ...
  ...
  ...
```

After execution, we obtain an output of 0.8 = 8/10.

16.7 Practical Algorithm

If you want to use floating point arithmetic for integer values, you can use this basic algorithm:

1) Load the Accumulator and X register with the first argument.
2) Execute your own subroutine, which uses the registers' contents to pass the value to the interpreter routine that converts an integer into a real value in FAC (also see: 15.2, 15.3, and 17).
3) Call, again, your own subroutine that copies FAC into ARG.
4) Load the Accumulator and X register with the second argument.
5) Execute the subroutine mentioned in (2).
6) Execute the appropriate arithmetical routine.
7) Execute your own subroutine that converts the floating point value in FAC into an integer value (also see: 15.2), and returns the result in memory locations $66, $67 through the Accumulator and the X register.

After you have written these small subroutines, you only need these instructions to multiply $00A5 by $02B2:

```
LDA #$B2
LDX #$02
JSR ROUTINE1      ;convert $02B2 into real number
JSR ROUTINE2      ;copy FAC into ARG
LDA #$A5
LDX #$00
JSR ROUTINE1      ;convert $00A5 into real number
JSR ARITHMETICALROUTINE
                  ;execute operation
JSR ROUTINE3      ;convert real number into integer
                  ;and copy into A,X
```

17. The Most Important Interpreter Routines

Routine	Parameters In	Parameters Out	C128
Get Parameters From Basic Text:			
CHKKOM	...(comma)	none	$795C
GETBYT	...(byte)	byte in X register	$87F4
GETBYT W/ CHRGET	...(byte)	byte in X register	$87F1
FRNUM	...(num. expression)	result in FAC1	$77D7
ADRFOR	FAC1: real number	address in Y/A, and $16, $17	$8815
GETADR	...(address)	address in Y/A, and $16, $17	$880F
ADRBYT	...(address)(byte)	byte in X, address in $16, $17	$8803
Create Variables:			
GETPOS	...(arbitrary variable)	pointer to var. in A/Y, and $49, $4a; variable name in $47, $48	$7AAF
STRRES	string length in A	poss. OUT OF MEMORY; decrement STREND ($35, $36) with length	$9299
TYPEFLAGS	none	$0F: $00=numerical; $FF=string	$0F

		$10: $00=real; $80=integer	$10
Floating Point Routines:			
REAL TO INTEGER	real number in FAC	integer in $66, $67	$8CC7
INT. W/ SIGN TO REAL	integer in $64, $65; X=$90	real number in FAC	$8C70
INTEGER TO REAL	integer in $64, $65; X=$90;SEC	real number in FAC	$8C75
FAC INTO ASCII	real number in FAC	number string in FAC	$8E42
PRINT FAC STRING	number string in FAC	print on screen	$55E2
PRINT INTEGER	integer in A/X	print on screen	$8E32
FAC = FAC + ARG	real number in FAC/ARG	result in FAC	$8848
FAC = ARG - FAC	real number in FAC/ARG	result in FAC	$8831
FAC = ARG * FAC	real number in FAC/ARG	result in FAC	$8A27
FAC = ARG / FAC	real number in FAC/ARG	result in FAC	$8B4C
FAC = FAC * 10	real number in FAC	result in FAC	$8B17
FAC = FAC / 10	real number in FAC	result in FAC	$8B38

TOP