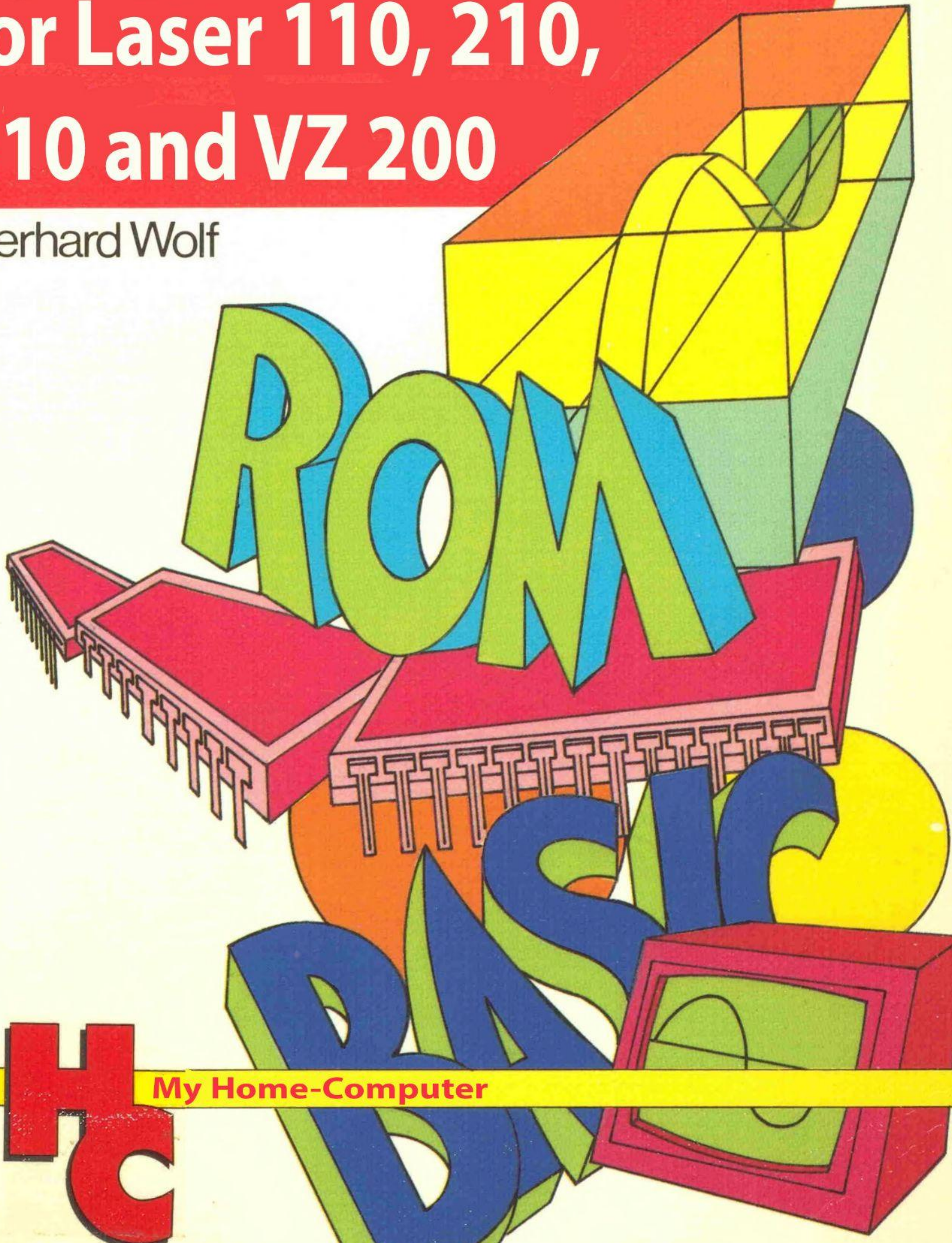
active computing

The BASIC-Interpreter for Laser 110, 210, 310 and VZ 200

Gerhard Wolf



HC
My Home-Computer

Gerhard Wolf

The BASIC Interpreter for Laser 110, 210, 310 and VZ 200

HC - My Home computer

Gerhard Wolf

The BASIC Interpreter for Laser 110, 210, 310 and VZ 200

Structure and working method

VOGEL BOOK PUBLISHING
WURZBURG

Published by the same author:
ROM listings for Laser 110, 210, 310 and VZ 200

(HC - My Home Computer)
ISBN 3-8023-0852-2

The Laser DOS in the Laser 110, 210, 310 and VZ 200

(HC - My Home Computer)
ISBN 3-8023-0868-9

CIP short title recording of the German Library

Wolt, Gerhard:
The BASIC Interpreter for Laser 110, 210, 310 and VZ 200: Structure working method / Gerhard Wolf. - 1st edition - Würzburg: Vogel, 1985.
(HC - My Home Computer)
ISBN 3-8023-0874-3

ISBN 3-8023-0874-3
1st edition. 1985

All rights, including the translation, reserved. No part of the work may be in in any form (print, photocopy, microfilm or any other process) Reproduced or reproduced without the written permission of the publisher processed, duplicated or distributed using electronic systems become. These are those expressly mentioned in 88 53, 54 UrhG exceptions are not affected.
Printed in Germany
Copyright 1985 by Vogel-Buchverlag Würzburg

Cover design: Bernd Schröder, Böhl
Manufacture: Alois Erdl KG, Trostberg

0. Introduction........ 9
1. What is an operating system?........ 11
2. General information about the LASER 110-310 operating system........ 13
3. The Memory allocation........ 15
4. Input/Output Area (6800H = 6FFFH)........ 19
Keyboard input:........ 20
Cassette Input........ 20
Screen control........ 20
Speaker output........ 21
Cassette output........ 21
Vertical SYNC-Pulse........ 21
5. The screen memory........ 23
6. The communication area........ 25
7. The free storage........ 27
8. Operating system features........ 31
System initialization........ 31
Input routine........ 34
Interpretation and execution control........ 37
The execution routines........ 41
Arithmetic and mathematical functions........ 43
Internal representation of the data........ 46
Input/output driver........ 49
9. Addresses and tables of the BASIC interpreter........ 55
Internal tables........ 56
BASIC keyword table (1650H - 1821H)........ 56
Address tables of the execution routines........ 58
Address table of the BASIC instructions (1822H - 1899H)........ 58
Address table of the BASIC functions (1608H - 164FH)........ 59
Ranking of arithmetic operations........ 60
Arithmetic routines........ 60
Data conversion (type matching)........ 61
Error messages........ 62
External tables........ 63
The BASIC communication area (7800H - 7AE8H)........ 63
The string cache (783BH - 78D2H)........ 73
Table of types for every variable (7901H - 791A)........ 74
Screen Row Status - Table (7AD7H - 7AE6H)........ 75
Programs data (Program Statement Table = PST)........ 75

The variables - table....78

10. The use of the BASIC Stack....83

Stack usage in a FOR/NEXT loop....83

Stack usage for a GOSUB instruction....84

11. Expression analysis....85

12. Function derivatives....91

Sine....92

Exponential function....93

Arctangent....94

Natural logarithm....95

13. Subroutines of the BASIC interpreter....97

Input/output routines....97

Reading from the keyboard....98

CALL 2BH Evaluate keyboard....98

CALL 49H Waiting for keyboard input....99

CALL 3E3H Reading a line....99

Display characters on the screen....101

CALL 33AH Display character on screen....101

CALL28A7H Output a line....102

CALL 1C9H Clear the screen....103

Direct output to screen memory....104

Output characters to the printer....105

CALL 3BH Print a character....105

CALL 5C4H Determine printer status....106

The cassettes - input/output....107

CALL 3511H Write a byte to the cassette....108

CALL 3558H Write file header to cassette....108

CALL 3AE8H Query BREAK key....110

CALL 388EH Create checksum....110

CALL 3775H Read one byte from cassette....110

CALL 35E1H Search for file on the cassette....111

CALL 358CH Transfer file name....113

CALL 386BH Load start and end address....113

Speaker - output....113

CALL 345CH Emit a single tone....114

CALL 2BF5H Play a melody....114

Conversion routines....115

Data type conversion....115

CALL 0A7FH Floating point number into integer....115

CALL 0AB1H Integer to single precision number... 116
CALL 0ADBH Integer to double precision number... 117
ASCII string to numerical representation... 118
CALL 1E5AH Convert ASCII string to integer... 118
CALL 0E6CH Convert ASCII string to binary value of any type... 119
CALL 0E65H ASCII string to double precision... 120
Convert binary value to ASCII string... 120
CALL 0FAFH Convert content from HL to ASCII... 120
CALL 132FH Convert integer to ASCII... 121
CALL 0FBEH Convert Floating point value into ASCII string... 121
Arithmetic routines... 123
Routines for processing integers... 123
CALL 0BD2H Add two integers... 123
CALL 0BC7H Subtract two integers... 124
CALL 0BF2H Multiplication of 2 integers... 125
CALL 2490H Division of integers... 126
CALL 0A39H Comparison of two integers... 126
Single precision aritimetic operations... 127
CALL 0716H Single precision addition... 127
CALL 0847H Single precision multiplication... 128
CALL 2490H Single precision division... 129
CALL 0A0CH Comparison of two single precision values... 129
Double precision arithmetic operations... 130
CALL 0C77H Double precision addition... 130
CALL 0C70H Double precision subtraction... 130
CALL 0DA1H Double precision multiplication... 131
CALL 0DE5H Double precision division... 132
CALL 0A4FH Comparison of two double precision values... 132
Mathematical routines... 133
CALL 0977H Determine absolute value ABS(N)... 133
CALL 0B37H Finding the next lower integer INT(N)... 134
CALL 15BDH Determine the arc tangent ATN (N)... 135
CALL 1541H Find the cosine of an angle COS (N)... 135
CALL 1547H Find the sine of an angle SIN (N)... 136
CALL 1439H Finding the exponential function e^x EXP (N)... 137
CALL 0809H Natural logarithm LOG (N)... 139
CALL 13E7H Find root of N SQR (N)... 139
CALL 14C9H Generate random number RND (N)... 140
RESTART - vectors... 141

RST 08H Check a character....141
RST 10H Find next valid character....142
RST 18H Compare DE with HL....143
RST 20H Determine data type.... 143
Transfer routines.... 144
CALL 09B4H Single precision number from BC/DE to work area 1....144
CALL 09B1H Single precision number from address HL to work area 1....145
CALL 09CBH Single precision number from work area 1 to memory.... 146
CALL 09C2H Single precision number from memory into BC/DE regs....146
CALL 09BFH Single precision number from workspace 1 to BC/DE regs.... 147
CALL 09A4H Transfer workspace 1 to the stack....147
CALL 09D3H Variable transfer routine.... 148
CALL 29C8H Transferring a string variable....148
BASIC functions.... 149
CALL 1B2CH Determine line in the program....150
CALL 260DH Get the address of a variable.... 151
CALL 1EB1H GOSUB Emulation.... 151

0. Introduction

The small computers in the LASER 110, 210, 310 and VZ200 series owe their popularity to the comfortable and comprehensive BASIC interpreter, which is located in memory modules (ROMs) inside the computer and is fully available to all users in all its functions after switching on.

This is a slightly modified variant of the well-known and globally used MICROSOFT BASIC, which is part of basic operating software. This operating software consists of a basic operating system and the extensive BASIC interpreter.

A floppy disk operating system (D0S = Disk Operating System) serves as an extension. This is also permanently stored in ROM modules, but is not housed in the main computer, but rather in the floppy disk controller.
By connecting the floppy disk controller to the system bus, these ROM modules are activated and add the software required to operate the disk drives to the basic system.

A further increase in performance can be achieved with an EXTENDED BASIC package, which was developed in Germany and expands the language range of the built-in BASIC interpreter to more than 38 commands, including powerful ones such as PLOT, PAINT, CIRCLE, RENUMBER and many others.

The aim of this book is to describe the essential functions of the BASIC ROM so that you can better understand the internal processes in your computer and make optimal use of all functions. The book is also intended to give assembler/machine program experts the opportunity to use BASIC ROM functions in their own programs, be it to carry out simple data conversions, use the input/output interfaces or mathematical functions (e.g. sine , cosine, etc.) not having to program it yourself.

Not every routine can be described in every detail. However, the description contained here should be sufficient to be able to carry out further detailed investigations in the ROM listing itself.

A detailed and documented ROM listing and a detailed description of the diskette operating system for LASER 110-310 and VZ200 have been published as independent volumes by Vogel-Verlag.

My thanks go to Mr. Dieter Effkemann for his support and work on Chapter 12 "Functional Derivations" and to my son Rainer for testing the program examples and careful proofreading.

1. What is an operating system?

A computer without any kind of operating system is a useless box filled with electronic components.

The need for an operating system arises solely from the need to establish a means of communication between the computer and its environment. This includes, among other things, the constant monitoring of all inputs, e.g. the keyboard, in order to be able to receive instructions and data and also the output of data, e.g. on a connected screen, to be able to display or transmit results.

If you type in a BASIC program on your LASER computer and then want it to be executed, there must already be a program in the computer that first accepts your entries and puts them in the correct location in the memory. The running of the program also requires strong support from this internal program. One such program present in the computer is the operating system.

There are now thousands of different operating systems, from simple systems in the ROM of home and personal computers to complex structures that can occupy entire disk units in large-scale computing systems.

The differences are due on the one hand to the different computers with different hardware equipment, but on the other hand also to the requirements placed on such a system. The operating system of a computer for process control certainly looks completely different than that for a commercial application, although both may run on the same hardware.

For this reason, no precise definition of an operating system can be drawn up.

Despite all the differences, the basic building blocks of the operating systems are similar and, in general, the following functional building blocks can be recognized in most systems:

1. A monitor program that constantly monitors all system inputs (e.g. the keyboard query).

2. Device driver routines that satisfy the specific physical needs of the connected peripheral devices such as keyboard, monitor, cartridge, floppy disk or printer.

3. General service routines that initialize system functions after entering certain commands (in LASER, e.g. LIST, CLOAD).

4. Language converters (compilers, assemblers, interpreters) that enable the use of a programming language (e.g. BASIC, PASCAL, FORTRAN).

5. Runtime support routines for the respective programming language.
   These include the arithmetic and mathematical functions for which there are ready-made routines in the operating system that are only called by the respective language.

6. Utility routines for device and memory management. These maintain internal tables, manage the various memory areas and control access to the input/output devices.

The following sections will identify and describe these components within the LASER operating system.

Since the operating software of the LASER computer is located in ROM modules within the computer, all of its functions are available immediately after switching on.

For other computers and operating systems, the operating software must first be read from an external memory (disk, tape, diskette, cassette). But this also requires a small program. This is called the boot loader (IPL = Initial Program Loader), which exists somewhere in the system or must be entered manually.

2. General information about the LASER 110-310 operating system

The operating system of the LASER computers 110-310 and the VZ200 is a self-contained program package that can run independently (stand alone system),

It includes a BASIC interpreter as a language converter as well as the necessary support and auxiliary routines for executing BASIC programs. It also offers the option of saving programs to cassettes and reloading them from cassettes.

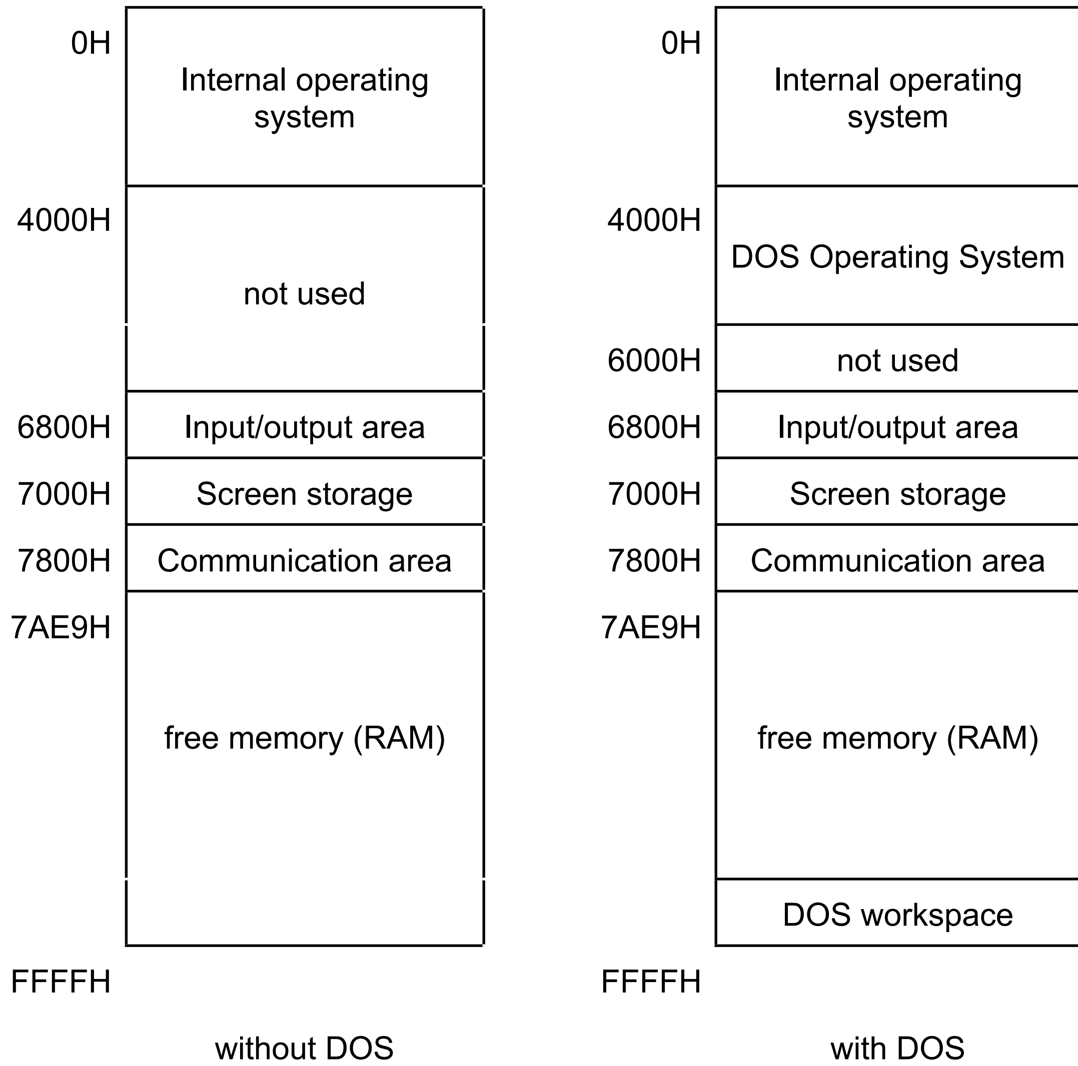
The floppy disk operating system is only available when the floppy disk controller is connected. It expands the language scope of the BASIC interpreter to include instructions for diskette editing and offers the option of saving programs to diskettes and loading them again from there.

If a floppy disk drive is connected, the floppy disk operating system is linked to the internal operating system when the computer is switched on.

The floppy disk operating system has its own interpreter for recognizing and processing the additional commands, but makes intensive use of support and auxiliary routines from the internal system and is therefore not an independently executable program unit.

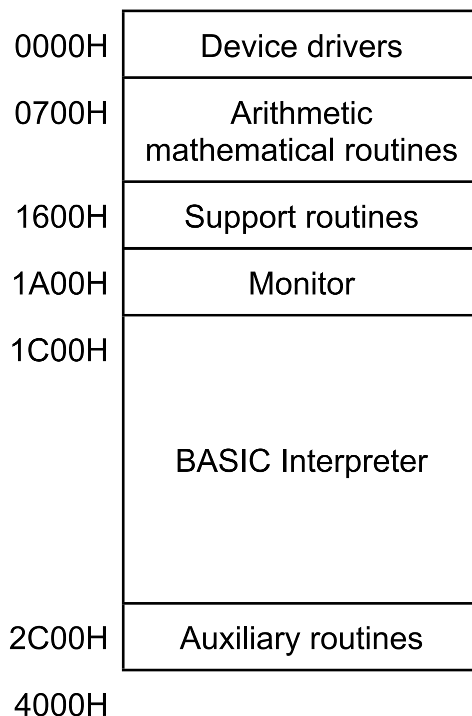
EXTENDED BASIC is a language extension of the BASIC interpreter and offers the convenience of additional commands, especially in the graphics area and in program development. Depending on the level of expansion of your computer, it is available on cassette or floppy disk and must be loaded into the computer in addition to the internal operating system.

3. The Memory allocation



The first 16K are dedicated exclusively to the internal operating system.

If you consider the 5 basic building blocks of an operating system, the situation in this area is roughly as shown below. However, no clear separation was carried out; individual code elements, which were assigned to a specific basic module based on their function, were scattered wildly in the memory. Especially at the end of the ROM area you will find a number of “backpacks” that were created there during later version changes.



If a floppy disk system is connected, the following 8K are occupied by the floppy disk operating system, otherwise the address area up to 6800H is not occupied by memory modules.

This is followed by an area titled 'Input/Output Area'. There is no memory chip hidden behind these addresses. With appropriate addressing, very direct interfaces to input/output blocks are addressed. For example, the keyboard is wired into this address space as a matrix, whereby each individual key can be queried directly.

This is followed by the screen memory with 2K. Behind it is a 2K RAM component that is constantly read and displayed by the image generator.

In the following communication area, work areas, address pointers and administration tables are created and used by the BASIC interpreter.

The free memory is located next to the communication area. This is the area in which the operating system stores BASIC programs and associated variables. You can also load your assembler or machine programs there and run them.

When the floppy disk drive is connected, a 318-byte long work area is created at the end of the DOS memory during system initialization, which fulfills similar functions for the DOS as the communication area in BASIC. If no floppy disk drive is connected, the free memory extends to the end of the RAM.

4. Input/Output Area (6800H = 6FFFH)

This memory area is used to directly control input/output modules such as keyboard, speakers, cassette and image generator.

When addressing input or output, the respective bits have different meanings.

The keyboard matrix, the cassette input and the vertical SYNC pulse of the image generator are read in via these addresses.

On the output side, the image generator, the loudspeaker and the cassette are controlled and addressed via this address area. Since an output value cannot be read back, the operating system always keeps a current copy of the output value in the address 783BH of the communication area and uses this as a reference if further output is to be made.

Except for keyboard input, it is irrelevant which address of the address space you address; only the top 5 address bits are evaluated.

Input:

Bit	**Used by**
7	Vertical Sync Pulse (CPU Interrupt line)
6	Cassette Input
5 .. 0	Keyboard

Ouput:

Bit	**Used by**
7 .. 6	not used
5	loudspeaker
4	image generator
3	image generator
2	cassette
1	cassette
0	loudspeaker

Keyboard input:

The keyboard forms a matrix of 8 rows and 6 columns in the address space 6800H to 68FFH, which is organized as follows:

A8 - A15 = 68

A7	A6	A5	A4	A3	A2	A1	A0	D0	D1	D2	D3	D4	D5	
0	1	1	1	1	1	1	1	**H**	**L**	**:**	**K**	**;**	**J**	= 687FH
1	0	1	1	1	1	1	1	**Y**	**O**	**Ret**	**I**	**P**	**U**	= 68FBH
1	1	0	1	1	1	1	1	**6**	**9**	**-**	**8**	**9**	**7**	= 68DFH
1	1	1	0	1	1	1	1	**N**	**.**		**,**	**Spc**	**M**	= 68EFH
1	1	1	1	0	1	1	1	**5**	**2**		**3**	**1**	**4**	= 68F7H
1	1	1	1	1	0	1	1	**B**	**X**	**Shift**	**C**	**Z**	**V**	= 68FBH
1	1	1	1	1	1	0	1	**G**	**S**	**Ctrl**	**D**	**A**	**F**	= 68FDH
1	1	1	1	1	1	1	0	**T**	**W**		**E**	**Q**	**R**	= 68FEH

You can see that a '0" in an address bit determines the line to be read. If one of the data bits read = 0, the corresponding key at the intersection of the matrix is pressed

If more than one address bit is set to 0, several lines will be read out at the same time. By addressing 6800H you can read the matrix at once.

Cassette Input

The pulses from the cassette recorder are read in via bit 6 of the address 6800H (or any other in the range 6800H-6FFFH).

Screen control

Control information can be transferred to the image generator via the address range 6800H-6FFFH.

Bit 3 = Screen mode
0 - Text
1 - Graphic

Bit 4 = Display Color
0 - Green
1 - Red

Speaker output

Use bits 0 and 5 to control the built-in small speaker. The two bits on the outside must always be complementary, otherwise you will never produce a sound {i.e. bit 0=1 and bit 5=0 or vice versa). By switching these bits at a certain frequency, the pitch is determined.

Cassette output

You output data bits to the cassette via bit positions 1 and 2 of the address space 6800H - 6FFFH, whereby both bits should always be the same.

For all three outputs (Image Generator, Speaker, Cassette), you must always respect the previous status and only change the bits you want to change. Therefore, the ROM routines always maintain a current copy of the output byte in address 783BH of the communication area. You should also take this into account and understand it if you ever want to spend something directly on these building blocks.

Vertical SYNC-Pulse

The vertical SYNC pulse generated by the image generator (every 20 ms for PAL) is normally used to generate an interrupt in the CPU. As described in more detail later, this interrupt is used to synchronize the transfer to the video RAM in order to obtain a flicker-free image.

This pulse can also be checked via bit 7 of the address range 6800H-6FFFH, even if the interrupts are switched off (disabled). You can use this to be able to carry out screen synchronization within your own program even when interrupts are switched off, without having to program your own interrupt handling.

5. The screen memory

The characters to be displayed on the screen must be entered into the screen memory. This memory area is constantly scanned by the image generator and the image information is displayed 1:1 on the screen. The screen memory occupies the address space 7000H-77FFH (= 2 KB).

In text mode only the first 512 bytes are used, where each byte can hold a character to be displayed. This corresponds to an output capacity of 16 lines of 32 characters.

Please note that custom screen codes are used, which do not always correspond to the ASCII code.

Two display colors can be selected via the screen control, these are
green and red. There are two types of display, whereby the selected color is used as a background color or as a font color. The display type is controlled via address 7818H of the communication area. You can also switch between the display types using the inverse character display.

Block graphics in 8 different colors are also possible within the text output.

Character table:

00 = @	0B = K	16 = V	21 = !	2C = ,	37 = 7
01 = A	0C = L	17 = W	22 = “	2D = -	38 = 8
02 = B	0D = M	18 = X	23 = #	2E = .	39 = 9
03 = C	0E = N	19 = Y	24 = $	2F = /	3A = :
04 = D	0F = O	1A = Z	25 = %	30 = 0	3B = ;
05 = E	10 = P	1B = [	26 = &	31 = 1	3C = <
06 = F	11 = Q	1C = \	27 = ‘	32 = 2	3D = =
07 = G	12 = R	1D =]	28 = (	33 = 3	3E = >
08 = H	13 = S	1E = ^	29 =)	34 = 4	3F = ?
09 = I	14 = T	1F = _	2A = *	35 = 5	
0A = J	15 = U	20 =	2B = +	36 = 6	

Codes 40H to 7FH represent the same characters in reverse.

A block graphic character is identified by bit 7 = 1.
Bit positions 4, 5 and 6 determine one of eight colors:

Bit 6 5 4	
0 0 0	green
0 0 1	yellow
0 1 0	blue
0 1 1	red
1 0 0	buff
1 0 1	cyan
1 1 0	magenta
1 1 1	orange

The shape of the lock graphic is determined in bit positions 0 - 3.

Each of the four bit positions is assigned to a quarter of a character position. The figure below shows the assignment of the individual bits
represents:

3	2
1	0

In graphics mode, the total 2K of the screen memory is used to achieve a resolution of 128 x 64 points (pixels).

The information of four pixels is stored in each byte, with 2 bits assigned to each individual pixel.

These 2 bits can be used to display four different colors, and you can choose between two sets of colors using the screen control (display color).

6800H		Bit 4 = 0	Bit 4 = 1	
	00	green	buff	(Background)
	01	yellow	cyan	
	10	blue	magenta	
	11	red	orange	

6. The communication area

The communication area is the operating system's notepad and is located in the address range 7800H to 7AE9H.

It contains tables, pointers and addresses that are created and managed there by the operating system.

In addition, there are work areas and buffer areas that are required when performing arithmetic operations and input/output operations.

There are a whole series of “RAM expansion outputs” within the operating system. These are CALL calls to a 3-byte area in the communication area. Normally the outputs in the communication area are occupied by a RETurn.

You can connect your own connection routines to the various operating system functions via the RAM expansion outputs. EXTENDED BASIC uses this very intensively; DOS also connects to the BASIC interpreter via such an extension output.

Some auxiliary routines, which must be modified before being called, have also been placed in the communication area. They are brought there from the ROM after the computer is switched on during the system initialization. One of these routines is a subroutine of the division function. This is modified from the ROM division routine and called to perform special subtractions and comparisons.

A description of each individual field in the communication area can be found in Chapter 9 "Addresses and tables of the BASIC interpreter".

7. The free storage

The free memory is available for you to load and run your own BASIC and machine programs.

After switching on the computer, the free memory extends from the end of the communication area to the end of the physical memory, or to the beginning of the D0S work area if the floppy disk drive is activated.

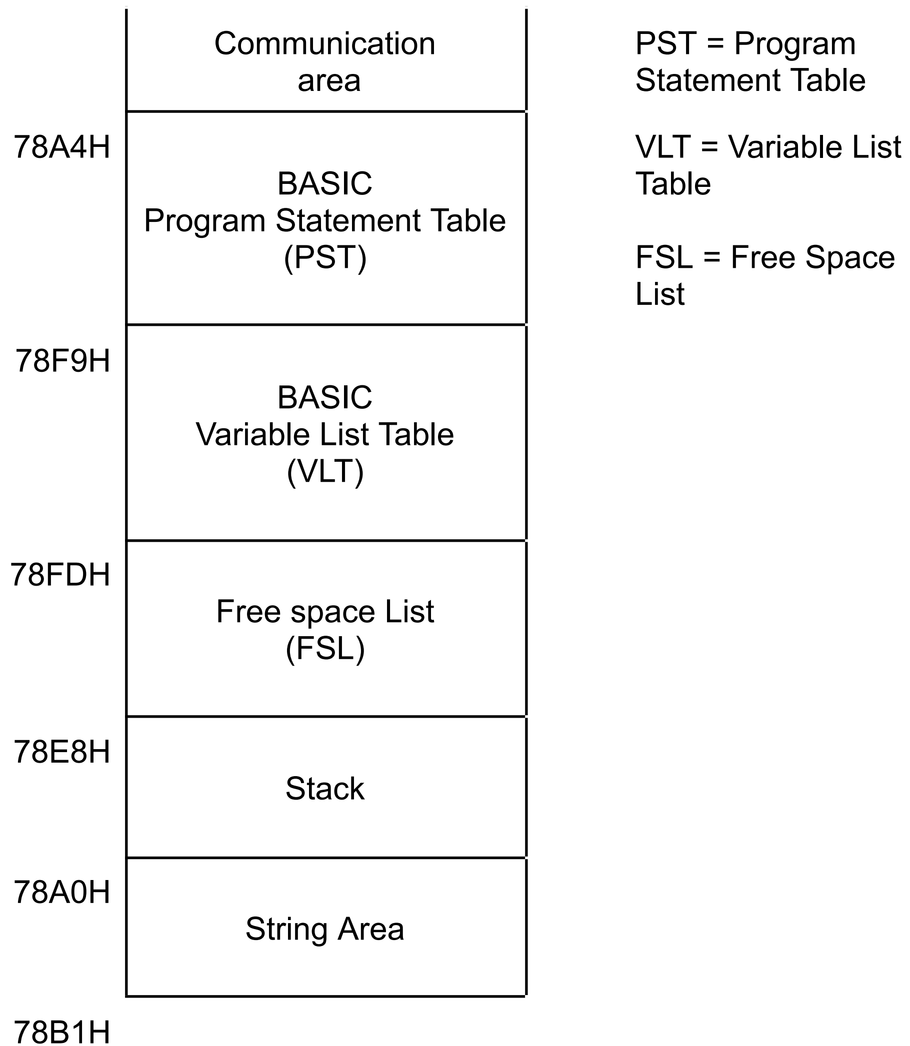
The limits of the free memory are displayed in two address pointers of the communication area:

> 78A4H/78A5H contains the start address
> 78B1H/78B2H contains the end address

By manipulating these pointers you can reduce the free memory from above or below. This is useful if you want to protect a known memory area from the BASIC interpreter (see DOS workspace).

If you use the free memory exclusively for machine programs, the entire space is available to you without restrictions; the machine program is responsible for managing this space itself.

With BASIC programs, the BASIC interpreter takes over the memory management of the free memory area. This is divided into different sub-areas and structures and managed dynamically.



Please note that the areas shown above do not have fixed addresses assigned to them. These areas are dynamic, i.e. their sizes and boundaries are fluid and are constantly adapted to current needs.

Look at the Programs table. This contains the loaded BASIC program. If you insert a program line there, the program table expands and at the same time the beginning of the variable table, which is directly connected to the programs table, moves. If you define a new variable in your program, the variable table will enlarge,

Since the boundaries of the individual areas are fluid, their current. Addresses are kept in address pointers of the communication area. This allows the tables to be moved according to the requirements of the program and at the same time offers the user the possibility of manipulation.

The Programa table (PST) contains the individual lines of the BASIC program in compressed format. Compressed means that within the lines the BASIC key type is replaced by 1-byte hexadecimal herts {TOKEN).

The start address of the program table is contained in the address pointer 78A4H of the communication area. After loading a program, it remains constant. If the beginning of the free memory is not manipulated (EXTENDED BASIC does something like this, for example), you will find the entry 7AE9H there.

The ending address of the Program table varies with the size of the program. It is identical to the start address of the variable table and can be found in the address pointer at 78F9H.

The variable table (VLT) contains names and values of all variables defined in a BASIC program. It is divided into two sections, Section 1 contains the simple variables and Section 2 contains dimensioned variables, i.e. matrices.

Variable names and values are entered into the variable table in the order in which they appear during the execution of a program. Each new variable that appears increases the size of the table. Once variables have been defined, they remain in the table until the system is reinitialized, i.e. you cannot delete them, but only change their value.

From the end of the variable table to the beginning of the stack area is the remaining free memory, which originally extended from the end of the communication area to the end of the memory. You can see from the picture that this free memory is restricted from above (lower address space) by the programs table and the variable table and from below (upper address space) by the string area and the stack area. This works fine until the end of the variable table collides with the stack area. In such a case, the error message "OUT OF MEMORY" is output.

The stack area is very dynamic. It constantly serves the operating system as a point memory for return addresses and register contents. Each CALL, PUSH or RST command increases the stack area by 2 bytes, each RET or POP instruction reduces the stack area accordingly.

The BASIC interpreter also writes entire memory blocks onto the stack. Each FOR/NEXT loop creates an 18-byte block on the stack and each GOSUB instruction creates a 7-byte block.

The last section shown behind the stack area is the string area. With the exception of the text variables firmly defined in the program (contained in quotation marks in the program text), all text variables {=strings) are stored in this area. For text variables, the variable table contains the reference address to the string area or the program table where the corresponding text can be found.

The length of the string area is defined as 50 bytes during initialization. However, you can vary this using the CLEAR command.

8. Operating system features

The basic building blocks of an operating system have already been discussed several times in the previous sections. Using this breakdown as a guide, the LASER operating system can be divided into the following seven functional areas:

1. System initialization
2. Input routine
3. Interpretation and execution control
4. Command and command execution
5. Arithmetic and mathematical routines
6. Input/output driver
7. System functions

Each of these functional areas will now be examined in more detail.

System initialization

In the first section of this book it was mentioned that the operating system of the LASER is located in ROM components and is available immediately after switching on. This is only the half truth. Although this system does not require any additional components that have to be loaded from anywhere, some initialization functions must be carried out, such as setting up the communication area, before the operating system can be used.

What happens now during system initialization?

After switching on, the computer always begins its program execution at the absolute address 0 of the ROM area. Then the image generator is switched to text mode and branched to address 0674H.

At 674H, the ROM contents from address 06D2H to address 0707H are transferred to the communication area from 7800H to 7835H. This initializes the addresses for the restart vectors RST 8, RST 10, RST 18 and RST 20 so that a jump to the current ROM routines can be made from there. The restart vectors RST 28, RST 30 and RST 38 are assigned RET commands (no function).

The "Device Control Blocks" for keyboard, screen and printer are also filled with output values (7815H - 782CH). The memory area 7836H to 785CH is deleted (00H).

For further initialization, a branch is made to address 75H. The following functions are carried out there:

The ROM range from 18F7H to 191DH is transferred to the communication range from 7880H to 78A6H. This area contains, among other things, the auxiliary routine for division. Some switches and pointers are also initialized, including the start address of the program table at 78A4H.

The input/output buffer starting at address 79E8H is initialized by entering its start address into the buffer pointer 78A7H and writing a header of 3AH-00H-2CH in front of the buffer. This buffer is used, among other things, to temporarily store each program line to be input and output.

Address vectors for special diskette commands from address 7952H to 79A5H become address 01D2H with a jump command
(JP 01D2H) preassigned. This causes the error message "DISK COMMAND ERROR" when one of these vectors is accessed. These disk vectors are a relic from an earlier use of this operating system and are not used by the LASER computer.

The RAM expansion outputs 79A6H to 79E2H are preset with RET (C9H).

Address 7AE8H (directly in front of the free memory) is loaded with 00H
and the stack pointer is first initialized with 79F8H (in the input/output buffer). This happens because the initialization now continues with CALL calls and a stack is required to store the return address.

A subroutine at 1B8FH (within the NEW command routine) is called, where the stack pointer is set to 7B4CH, an address is the free memory area.

The string buffer (from address 78B5H) is marked as empty by setting the address pointer at 78B3H to the first entry.

The screen is set as the output device and, if necessary, a carriage return is output to a connected printer.

The indexing lock is reset and the end of the stack area is marked 0.

The image memory is deleted (CLS).

The memory end address is determined and stored in 78B1H. For the string area, 50 bytes are reserved at the end of the free memory and the start address of the string area is stored in 78A0H.

The subroutine 1B4DH (NEW) is called.

There the TRACE function and the AUTO mode are switched off. The beginning of the Programs table is initialized to 00H-00H and marks it as empty.

The address pointer to the variable table (=end address of the program table) at 78F9H is set to the start address of the program table + 2. Since the variable table is also empty, the end addresses of both parts at 78FBH and 78FDH are set to the same value.

The TypeCode table starting from 7901H is set to "single precision" for all variables.

The stack area is set up before the string area.

A subroutine is then called at 3484H. There it is checked (CALL 3FA0H) whether the CTRL key was pressed at the same time as switching on (green background) and the background flags 7818H and 7819H are set accordingly. To create the correct background, the screen is cleared again.

The basic setting 20H is set via the input/output addresses (6800H) and is also noted in the memory at 783BH. This means "display color = green" and "text mode".

The basic values for the screen cursor delay counter (783AH) and the flash counter (7841H) are set.

The color value is set to "yellow" (7846H) and the interrupt vector in 787DH to RETurn (C9H).

Now the introduction text "VIDEO TECHNOLOGY..." is finally output and the Z80 is switched to interrupt mode 1. This means that with maskable interrupts the program address 38H is jumped to. If interrupts are permitted (EI), such an interrupt occurs every 20 milliseconds.

The final phase of initialization takes place from address 068EH. There it is checked one after the other whether there is a ROM cartridge slot in the address areas 4000H, &6000H or 8000H (e.g. the DOS from address 4000H). Such a cassette insert must begin at the above addresses with the byte sequence AAH-55H-E7H-18H.

If this byte sequence is detected at one of the addresses, it branches to the following address, otherwise the program in the BASIC input routine continues.

In this way, among other things, it is determined whether a floppy disk system is connected. The floppy disk additional routines are housed in ROMs in the floppy disk controller. The addressing starts at address 4000H, with the first four memory locations being occupied by AAH-55H-E7H-18H. If the diskette system is connected, this byte sequence is found at 4000H and the program for initializing the DOS continues from address 4004H.

Input routine

The input routine is similar in all operating systems. Their function is to accept keyboard input and react to received commands.

In the operating system of the LASER computers 110, 210, 310 and the VZ200, both system commands and BASIC program lines are processed by the input routine.

The entry into the input routine occurs at address 1A19H, also known as the start of the BASIC main loop or BASIC warm start address.

The message "READY" is initially output there and the RAM expansion output at 78ACH is called (initialized with RET).

The functions of the input routine can be divided as follows!

1. Read line from the keyboard
2. Replace BASIC keywords in the line with TOKENs
3. Check whether a direct command was entered (BASIC line without line number).
   If yes, go to 6.
4. Transfer line to program table.
5. back to 1
6. Interpret and execute

The input loop starts at address 1A33H. If the system is in AUTO mode, the line number and, if this is already present in the program, also the line content are output.

With a CALL to the routine at 03E3H, a line is taken over from the keyboard and transferred to the input/output buffer starting at address 79E8H. If the line entry was completed with the BREAK key, the loop immediately returns to the beginning of the loop (1A33H), i.e. the line entry is ignored.

An entered line number is converted from ASCII to binary format (CALL 1E5AH).

In the subroutine from 1BC0H onwards, the input line is examined and all BASIC keywords contained therein are replaced by 1-byte hexadecimal identifiers, the "TOKENS".

There is then a jump to the RAM expansion output at 79B2H, a good opportunity if you want to handle the input line yourself (connecting a machine routine). Among other things, EXTENDED BASIC was added here to recognize and implement the additional commands.

From 1AA4H system checks whether a line number has been entered. If not, it is a direct command. In this case, the system immediately branches to interpretation and execution control (1D5AH).

If a line number is available, the line entered is added or inserted in the correct position in the program table.

If a line with the same number already exists, it will be deleted beforehand using the routine at 2BE4H.

The end address of the program table (78F9H) is increased by the length of the entered line (1AC2H....) and with the routine at 1955H space is made for the new line by moving the lines behind it with a higher line number up in the memory.

From 1AD0H the preparation and transfer of the line from the input/output buffer to the program table begins.

If the line entered is empty, this is noted with 1ABFH and the transfer routine is skipped.

The reference addresses for line concatenation are updated throughout the program by the routine at 1AFCH. Finally, 1B5DH in the NEW routine is called to delete the variables table and reset some flags and identifiers so that it is no longer possible to continue the program with CONT after inserting or changing a line.

Then there is a jump back to the beginning of the loop to read the next line.

The AUTO command is only available in EXTENDED BASIC, but can also be switched on in normal BASIC using a POKE 30945.1 if you are satisfied with the default values (initial value = 10, step size = 10).

In AUTO mode, the line number to be output is provided in address 78E2H/78E3H and the increment value at 78E4H,

The lines are read in AUTO mode in section 1A3FH to 1A76H. The line number is first output from 78E2H/78E3H and, if this line already exists (search with 1B2CH), also the line content with 2E53H.

Reading is also done via 03E3H. If the BREAK button is pressed, in addition to ignoring any input, the AUTO mode is also switched off. After reading the line (from 1A60H), the line number stored at 78E2H/78E3H is increased by the value from 78E4H.

The further processing of the line takes place, as with the input without AUTO, from 1A81N.

Annotation:

On the LASER 110, 210, 310 and VZ200, a number of BASIC keywords are not encoded, although the necessary execution routines and TOKENs are present (see command table from 1650H). This was probably done for copyright reasons.
EXTENDED BASIC opens and makes available some of these commands and functions, along with many additional ones.

Interpretation and execution control

System commands and BASIC commands are executed through interpretation.

Interpretation means that all commands and all BASIC lines are only analyzed by the operating system at execution time and the necessary operations are initiated.

Such a procedure is common practice at the system command level. A command is read in, interpreted and the corresponding execution routine is triggered.

Executing a program using interpretation is generally limited to home and personal computers and only for a few languages, such as BASIC or LOGO.

The alternative to interpretation is to compile beforehand using a compiler.

Compilers translate the source code, which is the input lines in the corresponding language (FORTRAN, COBOL, PASCAL, PL1, etc.), into directly executable machine code, called object code.

Such object code is loaded into memory and started at execution time using a loader (part of the operating system). Once started, such a program runs almost completely independently of the operating system.

The LASER computers 110, 210, 310 and the VZ200 work in BASIC exclusively using the interpretation method. If you want to use directly executable machine code, you must have it generated using an assembler or enter it directly.

First of all, the main tool of an interpreter is comparison. An entered program line of a BASIC program is checked character by character and searched for BASIC keywords such as IF, THEN, FOR, NEXT, GOTO, etc. Each keyword found is replaced with a unique hexadecimal digit called a TOKEN (e.g. CLS = 84H, IF = 8FH, GOSUB = 91H, CLOAD = B9H). The program line treated in this way is then transferred to the program table.

This function takes place when the program is entered and relieves the execution control of this laborious and time-consuming work.

At the time of execution, the execution control addresses each line again and checks for the presence of these TOKENs. For almost every TOKEN, the BASIC interpreter has its own execution routine, which is called when the corresponding TOKEN is found to execute the function hidden behind it.

These routines for command and command execution then first carry out a further formal (syntactic) check:

- is the number of parameters correct?
- were the correct data types used?
- Are the commas in the right place?
- Are parameters enclosed in parentheses if necessary? etc.

With a compiler, these functions would all be eliminated at execution time because they already take place during compilation.

With LASER, the execution control is called when a command or program line without a line number was entered or after a RUN command was recognized.

With a RUN command, a complete BASIC program stored in the Programs table is executed.

Execution control starts at address 1D1EH and ends at address 1D77H. The entry occurs at address 1D5AH.

The following steps are carried out when a program line is executed:

1. Load the first character of the current line from the program table.
   Once the end of the program table has been reached, you return to the input routine.

2. If the tracer is switched on (TRON active), the line number is first displayed on the screen (<nn>).

3. If the character is not a TOKEN, go to step 7.

4. If the character > ’BBH’, it must be exactly “FAH? (MID$), otherwise it is not allowed at the beginning of the line and a SYNTAX ERROR will be generated.

5. If the character is less than 'BCH', it is used as an index for a jump table of the execution routines.

6. The corresponding execution routine is called and after execution it jumps back to step 1.

7. If the character is not a TOKEN, it must be a value assignment.
   The specified variable is determined; if it does not exist, it is added to the variable table,

8. The value expression is parsed and the value is assigned to the variable.

9. Back to step 1

Annotation:

For LASER, the TRON and TROFF commands are only available in EXTENDED BASIC. However, you can replace these with POKE commands in normal BASIC.

```
POKE 31003,1    =    TRON
POKE 31003,0    =    TROFF
```

The execution routine begins by loading the first character of the line to be processed.

The address 1D1EH is then packed onto the stack. This is the address to which all execution routines should return after successful completion of their operations.

If the character read is not a TOKEN (< 80H), it should be a value assignment, e.g. B=5.

The routine for performing value assignments starts at 1F21H. This is the same address that you would end up at if the TOKEN 8CH had stood for LET before the assignment. For this reason you can also leave out the LET.

The allocation routine expects that the address pointer to the line to be processed is directly in front of the variable name. The variable table is searched for an entry with the same name; if not present, the name is added to the table. There must be an equal sign after the variable name and then the value expression. The value expression is parsed in the routine starting from 2337H. The determined value is then converted into the correct type of the specified variable and saved at the variable address.

If a TOKEN is found at the beginning of the line, it is checked whether it is a valid TOKEN. Only TOKENs 80H to BBH are valid at the beginning of an instruction. The TOKEN BCH through F9H can only be used as part of a value assignment or a command sequence.

> Example: 8FH (IF) ‘expression’ CAH (THEN) xxxx
>
> The THEN TOKEN may not appear at the beginning of a line, but only after an IF statement.

The only exception to this is the MID$ TOKEN 'FAH', which cannot easily be used with the LASER because it uses one of the unused RAM expansion outputs in the communication area. However, new, self-made BASIC commands can be cleverly connected to the interpreter, e.g. a SORT or similar.

A TOKEN between 80H and BBH is used as an index into the jump table starting at address 1822H, in which the start address of the execution routine is stored for each valid instruction. The program is then continued at the address taken from there.

The values behind the TOKENs up to the end of the statement are the parameters required for the respective execution routine. Each execution routine knows the expected parameters and checks them for completeness and correct format. The end of a command's parameters must also match the end of the statement.

After an execution routine completes, control returns to execution control
handed over (1D1EH). There it is first checked whether the end of the instruction has been reached. The end of a statement is either a line end identifier 00H or a statement separator ':'. When a statement separator is reached, the following statement on the same line is interpreted and executed in the same way.

When the end of the line is reached, the next line in the program table is addressed and passed to execution control.

However, with a system command or a direct command there is no “next” line; these instructions are not executed from the program table, but directly from the input/output buffer. A program end 00H-00H is simulated at the end of the buffer. In addition, a line number of FFH-FFH or 65535 is added to identify such an instruction.

The RUN command tells the execution control that it has to take its instructions from the Programs table.

When the end of a BASIC program is reached, be it with or without an END statement, the END execution routine jumps back to the input routine.

An error detected during execution causes an appropriate error message to be output and also a return to the input routine.

The execution routines

The actual functions of the individual commands are carried out in the execution routines. There is a separate execution routine for each system command (CLOAD, CSAVE, CLEAR, RUN, etc.) and for each BASIC command (FOR; IF, GOSUB, GOTO, etc.). In addition, execution routines for all mathematical functions such as SIN, COS, ATN, LOG etc. are available.

These execution routines further analyze the instruction from the point where execution control detected a TOKEN. The statement is examined from left to right for special characters, such as commas or brackets or TOKENs. Each instruction has its own special parameter structure, so that a further formal check takes place here first.

In many cases, the execution routines also fulfill control functions by calling a whole series of internal subroutines to fulfill their function, which they share with other execution routines.

A good example of such an internal sub-routine is the expression analysis at 2337H. This routine is called by all execution routines that allow expressions in their parameters. Examples of such execution routines are those for processing IF, FOR and PRINT.

Expression analysis calls additional internal subroutines, such as 260DH, to manipulate variables within an expression. Since there are also indexed variables that allow an expression as indexing, this routine may have to call the expression analysis subroutine from which it was just called. This is called recursion.

An example of a value assignment that causes such a recursion:

    ST = E(BC/CD(3,F))/D(DE-1)

Other internal subroutines are:

- Prefix to the end of the statement (1F05H)
- Find a FOR or GOSUB data block on the stack (1936H)
- write an entry to the string buffer (2865H)
- and many others.

Mixing results are usually stored in a work area of the coaunication area (X register = 791DH).

All execution routines (except MID$) are started with the following register contents!

A	-	the character following the TOKEN
Flags	-	CARRY = numeric char
		ZERO = end of statement ':' or end of line X'00'
BC	-	Start address of the execution routine
DE	-	Address in jump table entry + 1 for the TOKEN
HL	-	Address of the character located in A in the instruction

A table of all system commands and BASIC commands with the address of their execution routines is included in Chapter 9.

Arithmetic and mathematical functions

First, some considerations of the computational capabilities of the Z80 and the BASIC interpreter.

The Z80 only supports 8-bit and 16-bit additions and subtractions internally. It does not perform any multiplication or division and certainly no floating-point arithmetic.

The Z80's register set for performing arithmetic consists of seven 16-bit register pairs (AF, BC, DE, HL, IX, IY, SP}, plus four shadow register pairs (AF', BC', DE', HL'), however, last four pairs can only be used for temporary storage.

All arithmetic operations must fundamentally take place between these registers. Register memory operations are only possible in a few exceptional cases via indirect addressing.

The operations of the registers among themselves are also limited. Especially in 16-bit arithmetic, only very specific register constellations are permitted.

The BASIC interpreter supports all arithmetic operations, be it addition, subtraction, multiplication or division, for three different types of variables:

- Integer variable (Integer)
- Single precision float variable (Single)
- Double precision float variable (Double)

This is achieved through internal subroutines that replace the Z80's missing hardware capabilities via software.

However, due to the complexity of the software, mixed operations are not supported, i.e. only variables of the same type can be linked together. Using unequal types of variables in an operation would lead to unpredictable results, creating a new type with random value.

The three variable types that the BASIC interpreter supports have the following format:

Integer	16 bit	1 bit sign, 15 bits data
Single	32 bit	8 bits Exponent, 24 bits mantissa with sign
Double	56 bit	8 bits Exponent, 48 bits mantissa with sign

This shows that the hardware registers are not sufficient to hold, let alone process, two single or double precision variables.

For this reason, two work areas were set up in the communications area, which are used as intermediate and working memory (register replacement).

These are work area 1 (referred to as the X register), which is the area 791DH to 7924H and the work area 2 (Y register), which extends from 7927H to 792EH.

These two workspaces have the following format:

Address	Integer	Single	Double
791D / 7927	—-	—-	LSB
791E / 7928	—-	—-	NSB
791F / 7929	—-	—-	NSB
7920 / 792A	—-	—-	NSB
7921 / 792B	LSB	LSB	NSB
7922 / 792C	MSB	NSB	NSB
7923 / 792D	—-	MSB	MSB
7924 / 792E	—-	EXP	EXP

(The addresses refer to work area 1; however, work area 2 is structured in the same way)

LSB = Least Significant Byte
NSB = Next Significant Byte
MSB = Most Significant Byte
EXP = Exponent

The various arithmetic operations for the three variable types have the following register/workspace assignment:

Integer Variables:

1.Operand		2.0perand	Operation	Result	Execution routine
HL	+	DE	Addition	HL	0BD2H
HL	-	DE	Subtraction	HL	0BC7H
HL	*	DE	Multiplication	HL	0BF2H
DE	/	HL	Division	ARB1	2490H

Single precision Variables

1.Operand		2.0perand	Operation	Result	Execution routine
ARB1	+	BCDE	Addition	ARB1	0716H
ARB1	-	BCDE	Subtraction	ARB1	0713H
ARB1	*	BCDE	Multiplication	ARB1	0847H
ARB1	/	BCDE	Division	ARB1	08A2H

The two register pairs BC and DE are used to record the 2nd operand in single precision operations.

Double precision Variables

1.Operand		2.0perand	Operation	Result	Execution routine
ARB1	+	ARB2	Addition	ARB1	0C77H
ARB1	-	ARB2	Subtraction	ARB1	0C70H
ARB1	*	ARB2	Multiplication	ARB1	0DA1H
ARB1	/	ARB2	Division	ARB1	0DE5H

Because mixed operations are not allowed, integer values can only be processed with other integer values. The same applies, of course, to single or double precision values.

Since four different arithmetic operations are possible for each type (+ - * /) and there are three different data types, there are twelve different arithmetic routines. There are also 3 routines for type-dependent arithmetic comparison operations.

An address table of the 15 routines is contained in the ROM starting at address 18ABH.

Each of these routines knows the type of values to be processed and expects them to be available in the correct registers or areas.

However, this does not apply to the mathematical routines, as these only work with one input value, which must always be provided in work area 1 (X register).

A problem now arises for the mathematical routines. You need to call arithmetic operations internally, but in workspace 1 you cannot see the type of argument provided there. For this reason, another byte was allocated in the communication area (78AFH), which provides information about the type of value stored in work area 1. This byte is also called the Type Flag.

One of the four following codes is entered there!

Code	Type of data
02	Integer
03	Text (string)
04	Single
08	Double

The type code corresponds exactly to the length of the value stored in workspace 1 for the specific data type.

With a few exceptions, the mathematical routines allow all different data types (see detailed description of the routines).

Internal representation of the data

To better understand what was said above, it will be explained here how the data is actually displayed internally in the LASER.

Integer variables are represented in 16 bits, with bit 15 containing the sign and bits 0 - 14 containing the value. The largest positive value that can be represented is 32767 (decimal) or 7FFF (hexadecinmal). The smallest negative value that can be represented is -32768 (decimal) or 8000 (hexadecinal).

Bit	15 = 0	positive number
	15 = 1	negative number

Positive range:	0000 .. 7FFF (hex) == 0 .. 32767 (dec)
Negative range:	FFFF .. 8000 (hex) == -1 .. -32768 (dec)

Note that negative values are represented in their complement.

Floating point variables are processed by BASIC in two different types:

- Single precision variable
- Double precision variable

These Types have an eight-byte signed exponent.

For single precision variables, the mantissa has a signed value in 24 bits (3 bytes) and for double precision variables, it has a signed value in 56 bits (7 bytes).

Both types have the same format, they only differ in the mantissa length.

Bit	31	Sign of the exponent 1 = positive (the decimal point must go to the right) 0 = negative (the decimal point must go to the left)
Bits	30..24	Value of the exponent = number of places to move the decimal point.
Bit	23	Sign of the mantissa 0 = positive 1 = negative
Bits	22..0	Value of the mantissa, left-bound (normalized), i.e. the most significant bit is in bit position 23. Negative values are displayed like positive ones, with the only difference that bit 23 is set.

The numbers are represented in the form

Number = mantissa * 2^EXP with: 0.5 <= mantissa < 1

The mantissa is 24 or 56 bits long, although the first bit is not saved because it is always 1. The sign of the mantissa is put in its place.

The exponent is always saved with an offset of 128 (= 80H), which results in the sign of the exponent in the most significant bit position.

Examples:	0.5	= 0.5 * 2 ^ 0 ⇒	Exp = 80, M = 00 00 00
	-4	= -0.5 * 2 ^ 3 ⇒	Exp = 83, M = 80 00 00
	-0.25	= -0.5 * 2 ^ -1 ⇒	Exp = 7F, M = 80 00 00

The largest number that can be represented with all its digits in single precision is 2^24-1 or 8388607 (dec.) or 7FFFFF (hex.). At double precision with the larger mantissa, the largest number that can be represented accurately is 2^56-1 or 3.578*10^16 (dec.) or 7FFFFFFFFFFFFF (hex).

However, these values, 8388607 or 3.578*10^16, do not represent the largest value that can be used for calculations, but only the largest that can be represented without loss of precision. This is because the exponent for both types can take values between 2^-128 and 2^127. Theoretically, the comma (in binary representation) can be shifted 127 places to the right or 128 places to the left, even though there are only 24 or 56 bits in the mantissa.

Depending on the type of data and the calculation, this is usually sufficient. The only thing that matters is the number of valid digits in a value.

Annotation:

With the LASER 110, 210, 310 and the VZ200 it is not so easy to work with variables of double precision. In the basic version of the BASIC interpreter you can only choose between single precision variables and integer variables.

- Variable names without addition = single precision
- Variable names with the suffix '%' = integer variable

However, there is a little trick in which you can define very specific variable names with double precision using a POKE XXXX;8 in the type table of the communication area (from 7901H).

Input/output driver

Drivers have the task of creating an interface between the logical intent of an application (a program) and the physical conditions of a special input/output device.

Here’s an example!
A program wants to output a very specific character, e.g. the letter 'A', to the cassette.
This character is provided in the ASCII code in A register of the Z80 and the driver is called.
The driver takes the character and transmits it bit by bit serially to the cassette output (bit 1.2 of the address 6800H) in the pulse sequence determined by the recording method.

On all LASER computers there are drivers for the keyboard, the screen, a parallel printer and the cassette.

Keyboard and printer drivers are addressed via a special device control block (Device Control Block = DCB) on the 7815H and 7825H, respectively. There is also a fuselage DCB on the 781DH for the screen, but it is not used by the LASER except for cursor management.

The DCBs hold counters and pointers for the specific device, such as the paper size and line counter for the printer or the cursor position for the screen. The driver addresses can also be found there.

The DCBs are set up in the communication area during system initialization and filled with standard values.

A driver is called for each character to be transmitted. Drivers do not recognize records or files, they cannot block or unblock characters. Such functions must be carried out by the calling program itself. In BASIC interpreters, for example, the routines for PRINT and INPUT do this.

When writing to the cassette, the PRINT command first creates a header of 255 * 80H and 5 * FEH, the data identifier F2H and the file name. Each variable is then transferred as an ASCII string. Commas separate the individual variables and a carriage return character (CR) completes the transfer.

An INPUT command first looks for the header, checks data identifiers and file names. All variables are then transferred to the input/output buffer one after the other until a carriage return character (CR = 0DH) is detected.

Each individual variable is then converted back into the correct format and transferred to the variable table.

The Keyboard Driver starts at address 2EF4H and extends to address 3014H, including an ECHO routine that displays the entered characters directly on the screen. There is also a routine from 0507H that handles the actuation of several keys at the same time, the so-called 'ROLLOVER'.

The keyboard driver can be called directly from the device control block (DCB) with CALL 2BH if a single character is to be fetched from the keyboard. This is what the routine of the BASIC command INKEY$ does, for example. With this type of query, however, no ECHO output is carried out on the screen.

However, the keyboard query is also integrated in the interrupt service routine (see Display Driver description). This type of query is mainly used when a whole line is to be read in via the screen editor at 3E3H and the entered text is to be displayed on the screen.

The Display Driver extends from 3039H to 342FH and consists of various subroutines.

A special problem needs to be taken into account here. The VIDEO RAM is accessed by two different blocks. On the one hand, the image generator constantly scans the VIDEO RAM and transfers the characters contained there to the screen; on the other hand, the Z80 has to write the information to be displayed into the VIDEO RAM. If this happens unsynchronized next to each other, you will get an unsightly flickering screen.

This is remedied by outputting to the VIDEO RAM only during the screen's blank phase. This process is controlled and synchronized via the vertical synchronization signal of the image generator. This is used to generate an interrupt. Screen output is first buffered in a special buffer from 7AAFH and only transferred to the VIDEO RAM in the interrupt service routine, i.e. during the blank phase of the screen.

The interrupt service routine is located at 2EB8H. In addition to the previously described buffered screen output (via 3F7BH and 30E8H), it also provides additional information.

- the flashing cursor display (2EDCH),
- the keyboard query (2EFDH) and
- the ECHO output to the screen (301BH) with a beep sound (3430H)

The interrupt occurs every 20 milliseconds in the PAL system. Before executing the above functions, a jump is made via the RAM expansion output 787DH, a good opportunity to work with the interrupt yourself, e.g. to implement a software clock.

To manage the screen, there is a table at the end of the BASIC communication area (from address 7AD7H) that provides information about the status of each individual line in 16 1-byte entries.

80H - Line is a single line of 32 characters
81H - Line is the first of a double line of 64 characters
00H - Line is second of a double line

This table in conjunction with two status flags at 7838H and 7839H are the basis of the screen editor at 03E3H.

The Printer Driver starts at address 058DH and continues at 3AB6H. This is the process when a normal ASCII character is to be output and the driver is called via the DCB.

A special type of output is the COPY command (from 3912H), which is used to print out the entire screen content. If you have a "Seikosha GP100" printer or similar connected, inverted characters (setting table from 39B4H), block graphics and even images of high-resolution graphics can also be printed out, with the colors replaced by different shades of gray.

A "Seikosha GP 100" compatible printer must meet the following conditions:

- Switching code text => graphics = 08H
- Switching code graphics => text = 0FH
- Single needle control for 7 needles

The Cassette Driver spans from 34A9H to 389CH and consists of many individual routines that are called depending on whether there is a cassette input or output.

The cassette recordings have the following format:

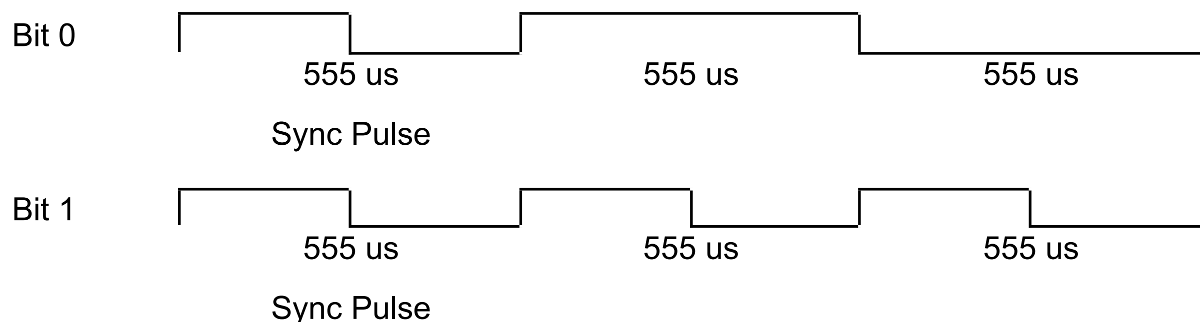
1. BASIC and machine programs

255 x 80H	-	Synchronization bytes
5 x FEH	-	Synchronization bytes
1 byte	-	Program Type 00 = BASIC 01 = machine program
max 15 bytes	-	Program name
00H	-	End identifier for name field
2 bytes	-	Program start address
2 bytes	-	Program end address + 1
n bytes	-	Program text/code
2 bytes	-	Checksum
20 x 00H	-	End identifier

2. Data files (from BASIC programs with PRINT#)

255 x 80H	-	Synchronization bytes
5 x FEH	-	Synchronization bytes
02H	-	Identifier for a data file
max 15 bytes	-	Program name
00H	-	End identifier for name field
n bytes	-	String variables, separated by comma
0DH	-	End identifier (Carriage Return)

The cassette recording takes place bit serially in the following signal format:



First, a synchronization pulse lasting 555 microseconds is issued. A binary zero is followed by a long pulse lasting a total of 1110 microseconds; a binary one is followed by two pulses of 555 microseconds each. This results in a total pulse length for a bit recording of 1665 microseconds and corresponds to a recording rate of 600 bits/sec. (Baud).

The messages on the last screen line generated by the cassette driver routine can be suppressed if necessary. You can do this via the address 784CH.

784CH = 0	Messages are displayed
784CH = 1	Messages are not displayed

9. Addresses and tables of the BASIC interpreter

0000H	Internal operating system (ROM)
4000H	free or occupied by DOS
6000H	free
6800H	Input/output area
7000H	Screen memory
7800H	BASIC communication area
(78A4H)	Program table (PST)
(78F9H)	Variable table-VLT simple variable
(78FBH)	Indexed variables
(78FDH)	Free storage
(78E8H)	BASIC - Stack
(78A0H)	String - range
(78B1H)	End of memory

nnnn = absolute location of the area in memory
(nnnn) = address pointer in communication area

Internal tables

The lists and tables that are permanently located in the ROM of the operating system are referred to as 'internal tables'. This also means that the content and position of these tables are fixed. They are used by the BASIC interpreter for syntactic checking, expression analysis, data conversion and the execution of certain commands (e.g. FOR, IF).

BASIC keyword table (1650H - 1821H)

This table contains all reserved words and characters of the BASIC interpreter, be they statements or functions.

Each entry contains one word, with 7th bit = 1 in the first character of the entry.

During the input phase, a read line is compared character by character with this table. If a text section of the line matches an entry, it is replaced by the so-called TOKEN. A TOKEN is a 1-byte hexadecimal value that is formed from entry number of the entry found. In addition, bit 7 is set.

Example:
CLS is the 5th entry in the table.
Since counting starts with 0, this results in an entry number of 4.
If bit 7 is switched on, this results in a TOKEN of 84H for CLS.
This means that every character string CLS in the program text is replaced by 84H, with the line length being shortened by 2 bytes.

But this alone is not enough. The head is rarely at the point where a new byte begins on the track. As a rule, it will start reading in the middle of a byte. However, since the data is stored consecutively bit by bit without gaps, it is impossible to identify the beginning of a byte. That is, first of all, a start of recording is found. One speaks here of a synchronization of the head.

If you examine the table, you will notice that a number of keywords are not encoded but consist of binary zeros. These are enclosed in parentheses and marked with an asterisk in the list below. However, as long as these are not DOS commands, the execution routines are mostly there.

In my opinion, the reason is to look for copyright in order to avoid too much similarity to another computer. With EXTENDED BASIC many of these commands are made available to the user again.

Keyword	TOKEN	Keyword	TOKEN	Keyword	TOKEN
END	80H	(NAME)	A9H *	OR	D3H
FOR	81H	(KILL)	AAH *	>	D4H
RESET	82H	(LSET)	ABH *	=	D5H
SET	83H	(RSET)	ACH *	<	D6H
CLS	84H	(SAVE)	ADH *	SGN	D7H
(CMD)	85H *	(SYSTEM)	AEH *	INT	D8H
(RANDOM)	86H *	LPRINT	AFH	ABS	D9H
NEXT	87H	(DEF)	B0H *	(FRE)	DAH *
DATA	88H	POKE	B1H	INP	DBH
INPUT	89H	PRINT	B2H	(POS)	DCH *
DIM	8AH	CONT	B3H	SQR	DDH
READ	8BH	LIST	B4H	RND	DEH
LET	8CH	LLIST	B5H	LOG	DFH
GOTO	8DH	(DELETE)	B6H *	EXP	E0H
RUN	8EH	(AUTO)	B7H *	COS	E1H
IF	8FH	CLEAR	B8H	SIN	E2H
RESTORE	90H	CLOAD	B9H	TAN	E3H
GOSUB	91H	CSAVE	BAH	ATN	E4H
RETURN	92H	NEW	BBH	PEEK	E5H
REM	93H	TAB(	BCH	(CVI)	E6H *
STOP	94H	TO	BDH	(CVS)	E7H *
ELSE	95H	(FN)	BEH *	(CVD)	E8H *
COPY	96H	USING	BFH	(EOF)	E9H *
COLOR	97H	(VARPTR)	C0H *	(LOC)	EAH *
VERIFY	98H	USR	C1H	(LOF)	EBH *
(DEFINT)	99H *	(ERL)	C2H *	(MKI$)	ECH *
(DEFSNG)	9AH *	(ERR)	C3H *	(MKS$)	EDH *
(DEFDBL)	9BH *	(STRING$)	C4H *	(MKD$)	EEH *
CRUN	9CH	(INSTR)	C5H *	(CINT)	EFH *
MODE	9DH	POINT	C6H	(CSNG)	F0H *
SOUND	9EH	(TIME$)	C7H *	(CDBL)	F1H *
(RESUME)	9FH *	(MEM)	C8H *	(FIX)	F2H *
OUT	A0H	INKEY$	C9H	LEN	F3H
(ON)	A1H *	THEN	CAH	STR$	F4H
(OPEN)	A2H *	NOT	CBH	VAL	F5H
(FIELD)	A3H *	STEP	CCH	ASC	F6H
(GET)	A4H *	+	CDH	CHR$	F7H
(PUT)	A5H *	-	CEH	LEFT$	F8H
(CLOSE)	A6H	*	CFH	RIGHT$	F9H
(LOAD)	A7H	/	D0H	MID$	FAH
				‘	FBH

Address tables of the execution routines

In ROM there are two different address tables for the execution routines. The first is used by execution control when a BASIC statement is to be executed. It contains addresses for the TOKEN execution routines 80H to BBH and is located in the address range 1822H - 1899H. The first TOKEN of an instruction (Bit7=0) is used as an index (0 - 59) for table access. The address of the execution routine is taken from the table and called. If the statement does not begin with a TOKEN, it branches to the value assignment routine (implicit LET).

The second address table from 1608H - 164FH contains addresses of routines for BASIC functions that may only appear on the right side of a value assignment (after the equal sign).

If a TOKEN is encountered in the range D7H to FAH during the expression analysis, it is used as an index (0 - 35) for the second address table and the address of the execution routine is taken from it.

No address table is required for the TOKEN BCH to D6H, as these are processed directly by the other execution routines when they occur.

Address table of the BASIC instructions (1822H - 1899H)

TOKEN	**Keyword**	**Address**	**TOKEN**	**Keyword**	**Address**
80	END	1DAE	9E	SOUND	2BF5
81	FOR	1CA1	9F	RESUME	1FAF
82	RESET	0138	A0	OUT	2AFB
83	SET	0135	A1	ON	1F6C
84	CLS	01C9	A2	OPEN	7979
85	CMD	7973	A3	FIELD	797C
86	RANDOM	01D3	A4	GET	797F
87	NEXT	22B6	A5	PUT	7982
88	DATA	1F05	A6	CLOSE	7985
89	INPUT	219A	A7	LOAD	7988
8A	DIM	2608	A8	MERGE	798B
8B	READ	21EF	A9	NAME	798E
8C	LET	1F21	AA	KILL	7991
8D	GOTO	1EC2	AB	LSET	7997
8E	RUN	1EA3	AC	RSET	799A

TOKEN **Keyword** **Address** **TOKEN** **Keyword** **Address**

8F	IF	2039	AD	SAVE	79A0
90	RESTORE	1D91	AE	SYSTEM	0000
91	GOSUB	1EB1	AF	LPRINT	2067
92	RETURN	1EDE	B0	DEF	795B
93	REM	1F07	B1	POKE	2CB1
94	STOP	1DA9	B2	PRINT	206F
95	ELSE	1F07	B3	CONT	1DE4
96	COPY	3912	B4	LIST	2B2E
97	COLOR	389D	B5	LLIST	2B29
98	VERIFY	3738	B6	DELETE	2BC6
99	DEFINT	1E03	B7	AUTO	2008
9A	DEFSNG	1E06	B8	CLEAR	1E7A
9B	DEFDBL	1E09	B9	CLOAD	3656
9C	CRUN	372E	BA	CSAVE	34A9
9D	MODE	2E63	BB	NEW	1B49

Address table of the BASIC functions (1608H - 164FH)

TOKEN	Keyword	Address	TOKEN	Keyword	Address
D7	SGN	098A	E9	EOF	7961
D8	INT	0B37	EA	LOC	7964
D9	ABS	0977	EB	LOF	7967
DA	FRE	27D4	EC	MKI$	796A
DB	INP	2AEF	ED	MKS$	796D
DC	POS	27F5	EE	MKD$	7970
DD	SQR	13E7	EF	CINT	0A7F
DE	RND	14C9	F0	CSNG	0AB1
DF	LOG	0809	F1	CDBL	0ADB
E0	EXP	1439	F2	FIX	0B26
E1	COS	1541	F3	LEN	2A03
E2	SIN	1547	F4	STR$	2836
E3	TAN	15A8	F5	VAL	2AC5
E4	ATN	15BD	F6	ASC	2A0F
E5	PEEK	2CAA	F7	CHR$	2A1F
E6	CVI	7952	F8	LEFT$	2A61
E7	CVS	7958	F9	RIGHT$	2A91
E8	CVD	795E	FA	MID$	2A9A

Ranking of arithmetic operations

The ranking of the various arithmetic operations in arithmetic expressions is also determined using a table located in the address range 1B9AH to 18A0H.

This table contains numeric values for the various operators that determine precedence.

During expression analysis, each operator/operand pair plus the rank value of the preceding operator is placed on the stack. If an operator with a higher rank value than the previous one is found, the operation is carried out immediately and the resulting intermediate result is put on the stack.

Operator	**Function**	**Rank Value**
+	Addition	79
-	Subtraction	79
*	Multiplication	7C
/	Division	7C
[	Exponentiation	7F
AND	Logical AND	50
OR	Logical OR	46

Arithmetic routines

For the three different types of numerical variables, the ROM at 18ABH-18C8H contains three tables with the start addresses of the associated arithmetic routines. These are used by expression analysis.

Function	Integer Variables	Single precision Variables	Double precision Variables
Addition	0BD2	0716	0C77
Subtraction	0BC7	0713	0C70
Multiplication	0BF2	0847	0DA1
Division	2490	08A2	0DE5
Comparison	0A39	0A0C	0A78

For completeness, the address of the routine for adding text variables (strings) is 298FH.

Data conversion (type matching)

To convert data into the various variable types, there is another table which, depending on the target type, contains the addresses of the corresponding conversion routines. These routines convert the value located in work area 1 (X register) into the desired data type. They are primarily used by expression analysis to be able to link values and intermediate results of different types.

The table is located at 18A1H-18AAH.

Conversion into	Address
Text (string) variable	0AF4 *
Integer variable	0A7F
Single precision variable	0AB1
Double precision variable	0ADE

* For text variables, the specified routine does not contain any conversion, but only a test as to whether the variable in work area 1 is really a text variable. If not, 'TYPE MISMATCH ERROR' is displayed.

Error messages

There are two tables with error messages in the ROM area.

The first table is located at 18C9H-18F6H and only contains a two-character abbreviation for each error message. It is a relic from another operational computer of this operating system and is not used by the LASER computer.

The table used by the LASER 110, 210, 310 and the VZ200 is included 3CECH-3E28H and contains the error messages in written form.

The error messages are determined using an error number, which is used as an index for table access.

Error number	Error code	Error text
00	NF	NEXT WITHOUT FOR
02	SN	SYNTAX ERROR
04	RG	RET’N WITHOUT GOSUB
06	OD	OUT OF DATA
08	FC	FUNCTION CODE ERROR
0A	OV	OVERFLOW
0C	OM	OUT OF MEMORY
0E	UL	UNDEF’D STATEMENT
10	BS	BAD SUBSCRIPT
12	DD	REDIM’D ARRAY
14	0/	DIVISION BY ZERO
16	ID	ILLEGAL DIRECT
18	TM	TYPE MISMATCH
1A	OS	OUT OF SPACE
1C	LS	STRING TOO LONG
1E	ST	FORMULA TOO COMPLEX
20	CN	CAN’T CONTINUE
22	NR	NO RESUME
24	RW	RESUME WITHOUT ERROR
26	UE	UNPRINTABLE ERROR
28	MO	MISSING OPERAND
2A	FD	BAD FILE DATA
2C	L3	DISK COMMAND ERROR

External tables

The data structures that are created and managed by the BASIC interpreter in the RAM area are referred to as 'external tables'. The characteristic of external tables is that their content can change and their location in the RAM area can also change.

For tables that change their location in memory, there are address pointers in the BASIC communication area so that they can be found and addressed again at any time.

The BASIC communication area (7800H - 7AE8H)

In the communication area, the BASIC interpreter creates and manages all the necessary address pointers and management tables, which can change during normal program processing or in which the user can also define or change their own process variables.

Think of the communications area as the BASIC interpreter's notepad.

The following list contains a description of each individual byte in this area.

Some tables within the communication area are described in detail following this list.

The range 7800H to 7835H is pre-assigned from the ROM area during system initialization.

```
7800  C3 96 1C    JP     1C96H        ; RST 8 - Vector
7803  C3 78 1D    JP     1D78H        ; RST 10 - Vector
7806  C3 90 1C    JP     1C90H        ; RST 18 - Vector
7809  C3 D9 25    JP     25D9H        ; RST 20 - Vector
780C  C9 00 00    RET                 ; RST 28 - Vector
780F  C9 00 00    RET                 ; RST 30 - Vector
7812  FB C9 00    EI ; RET            ; RST 38 - Vector
```

```
; Keyboard Device Control Block (DCB)
7815  01              ; DCB Identifier
7816  F4 2E           ; Driver Address
7818  00              ; Background Flag (0-green,1-black)
7819  00              ; act. background
781A  00
781B  4B 49           'KI'

; Screen Device Control Block (DCB) (unused in the LASER 110-310)
781D  00              ; DCB Identifier (deleted)
781E  00 00           ; Pointer to Programs start address used by CLOAD
7820  00 70           ; Cursor Address
7822  00
7823  00 00           ; Checksum for cassette input/output

; Printer Device Control Block (DCB)
7825  06              ; DCB Identifier
7826  8D 05           ; Driver Address
7828  43              ; Lines per Page + 1
7829  00              ; Line counter
782A  00
782B  50 52           'PR'

782D  C3 00 50        JP    5000H         ; unused
7830  C7 00 00        RST 0               ; unused

7833  3E 00           LD    A,0           ; with unknown DCB identifier A=0
7835  C9              RET

7836        ; Buffer B1 for 1st key code with multiple key presses at the same time
7837        ; Buffer B2 for 2nd key code with multiple key presses at the same time

7838        ; FLAG 1
            ; Bit 7 - CONTROL flag
            ; Bit 6 - REPEAT flag
            ; Bit 5 - WAIT flag
            ; Bit 4 - B2 Status flag
            ; Bit 3 - B1 Status flag
            ; Bit 2 - FUNCTION flag
            ; Bit 1 - INVERSE flag
            ; Bit 0 - SHIFT flag
```

```
7839            ; FLAG 2
                ; Bit 7 - unused
                ; Bit 6 - CRUN flag
                ; Bit 5 - Ini flag for buffered output
                ; Bit 4 - Flag for INPUT statement
                ; Bit 3 - VERIFY flag
                ; Bit 2 - BREAK flag
                ; Bit 1 - BUZZER flag
                ; Bit 0 - Carriage Return flag

783A            ; Time counter

783B            ; INPUT/OUTPUT Latch (shadow register)

783C            ; Copy of Character for cursor display

783D-7840       ; unused

7841            ; Cursor Blink Counter

7842-7843       ; Temporary storage for keyboard scan (row/column)

7844-7845       ; Temporary storage for keyboard scan (Matrix address)

7846            ; Color code

7847-784B       ; unused

784C            ; Output flag for message output for cassettes I/0
                ; (if > 0 - messages are suppressed)

784D-787C       ; unused

787D  C9 00 00    RET   ; RAM expansion output of the interrupt service routine
```

The 7880H-78A5H area is filled from the ROM area during initialization

Subprogram for division:

```
7880  D6 00       SUB   0      ; Subtraction Z2 - Z1
7882  6F          LD    L,A    ; is modified before each call.
7883  7C          LD    A,H
7884  DE 00       SBC   A,0
7886  67          LD    H,A
7887  78          LD    A,B
```

```
7888  DE 00       SBC  A,0
788A  47          LD   B,A
788B  3E 00       LD   A,0
788D  C9          RET

788E  4A 1E       ; USR starting address (initialized with FUNCTION CODE error)

7890  40 E6 4D    ; Multiplier for RND

; Subprogram for INP
7893  DB 00       IN   A, (0)
7895  C9          RET

; Subprogram for OUT
7896  D3 00       OUT  (0),A
7898  C9          RET

7899  00          ; INKEY$ cache

789A  00          ; last error code for ERR

789B  00          ; Printer position on the line

789C  00          ; Selected Out Device flag (0=Screen, 1=Printer, 80=cassette)

789D  40          ; BASIC Line length on screen (default is 64)

789E  30          ; last Tab position (default is 48)

789F  00          ; unused

78A0  47 7B       ; Starting address of the string area (defaut 7B47)

78A2  FE FF       ; Current BASIC Line Number

78A4  E9 7A       ; Start address of the program text (encoded data? or…)

78A6-78A7         ; Column pointer for output image (??)

78A7-78A8         ; Pointer to input/output buffer (from 79EBH)

78A9              ; Input flag (0 = cassette) ??

78AA-78AD         ; last random number
```

78AE ; Flag for DIM instruction

78AF ; Type of the value in the X register
; 02 - Integer
; 03 - String
; 04 - SIngle
; 08 - Double

78B0 ; Flag for intermediate code generation for DATA
; operation code during expression analysis

78B1-78B2 ; End address of the BASIC memory area

78B3-78B4 ; Pointer to string area

78B5-78D2 ; String buffer (10 x 3 bytes)
; (1 byte - length, 2 bytes - address in string area)

78D3-78D5 ; Current string variable

78D6-78D7 ; Pointer to the last free byte in the string area

78D8-78D9 ; General address buffer format flag for string
; output of a number

78DA-78DB ; DATA - line number

78DC ; Indexing blocking flag

78DD ; RESUME/RETURN Flag

78DE ; Intermediate buffer for PRINT USING
; DATA flag for INPUT etc.

78DF-78E0 ; general address memory
; e.g. program continuation at NEW
; Run variable for FOR/NEXT
; Address d. Variable table at LET

78E1 ; AUTO input - flag (0 - no AUTO)

78E2-78E3 ; AUTO - next line number

78E4-78E5 ; AUTO - increment value

```
78E6-78E7      ; Address of the current BASIC line (FFFF = direct command)

78E8-78E9      ; Pointer to the BASIC stack

78EA-78EB      ; Number of the BASIC line in which the last error occurred

78EC-78ED      ; Number of the BASIC line in which the last error occurred
               ; ( - Option for LIST)

78EE-78EF      ; Address of the BASIC line where the error occurred

78F0-78F1      ; Address of an error handling routine (ON ERROR)

78F2           ; Error - Flag (Error=255, RESUME=0)

78F3-78F4      ; Address of the decimal point in the print buffer

78F5-78F6      ; BASIC Line number where the last break occurred
               ;  (END, STOP, BREAK)

78F7-78F8      ; Address of the BASIC line where the last break occurred

78F9-78FA      ; Programa end address
               ; Start of the variable table

78FB-78FC      ; End address of the 1st part of the variable table
               ; Start of the arrays table (2nd part)

78FD-78FE      ; Starting address of free memory
               ; End of arrays table

78FF-7900      ; Pointer to DATA line

; Table of types for every variable

7901           ; A

7902           ; B

7903           ; C

7904           ; D
```

7905 ; E

7906 ; F

7907 ; G

7908 ; H

7909 ; I

790A ; J

790B ; K

790C ; L

790D ; M

790E ; N

790F ; O

7910 ; P

7911 ; Q

7912 ; R

7913 ; S

7914 ; T

7915 ; U

7916 ; V

7917 ; W

7918 ; X

7919 ; Y

7920 ; Z

7791B ; TRACE Flag (0 = TRON, AF = TROFF)

; X Register

791C ; 791C additional byte for right shift

	INT	STRING	SINGLE	DOUBLE
791D				LSB
791E				NSB
791F				NSB
7920				NSB
7921	LSB	ADR LSB	LSB	NSB
7922	MSB	ADR MSB	NSB	NSB
7923			MSB	MSB
7924			EXP	EXP

7925 ; Buffer for arithmetic operations. e.g., sign

7926-792E ; Y Register (structured like X register)

792F ; unused

7930-7949 ; Printer buffer

794A-7951 ; Additional register for double precision
; multiplication and division

RAM vectors for floppy disk commands
initialized with JP 012DH (DISK COMMAND Error)

7952 ; CVI statement
7955 ; FN statement
7958 ; CVS statement
795B ; DEF statement
795E ; CVD statement
7961 ; EOF statement
7964 ; LOC statement
7967 ; LOF statement
796A ; MKI$ statement
796D ; MKS$ statement
7970 ; MKD$ statement
7973 ; CMD statement
7976 ; TIME$ statement
7979 ; OPEN statement

```
797C                ; FIELD statement
797F                ; GET statement
7982                ; PUT statement
7985                ; CLOSE statement
7988                ; LOAD statement
798B                ; MERGE statement
798E                ; NAME statement
7991                ; KILL statement
7994                ; & statement
7997                ; LSET statement
799A                ; RSET statement
799D                ; INSTR statement
79A0                ; SAVE statement
79A3                ; LINE statement
```

RAM expansion hooks
initialized with C9H-00H-00H (RET)

```
79A6                ; from ERROR routine
79A9                ; from USR routine
79AC                ; start of BASIC loop
79AF-79B1           ; unused
79B2                ; from programm input
79B5                ; End of program input
79B8                ; End of program input
79BB                ; from NEW and END
79BE                ; Final query PRINT
79C1                ; Data output
79C4                ; Reading Keyboard
79C7                ; RUN execution
79CA                ; Start of PRINT statement
79CD                ; PRINT statement
79D0                ; PRINT statement
79D3                ; PRINT statement
79D6                ; INPUT statement
79D9                ; MID$ function
79DC                ; INPUT statement
79DF                ; READ + INPUT + LIST
79E2-79E4           ; unused

79E5  3A 00 2C      ; I/O buffer header
```

79EB-7A9C	; Input/output buffer (178 bytes)
79F8	; BASIC stack during initialization
7A9D-7AAD	; Program/file name - buffer for cassette input/output
7AAE	; Column display on screen

Zusätzlicher Ausgabepuffer für gepufferte
Bildschirmausgabe

7AAF	; Number of characters in the buffer
7AB0-7AB1	; Buffer pointer
7AB2-7AD1	; Buffer area
7AD2-7AD5	; 4 bytes for draw Graphics, SOUND and cassette I/O
7AD6	; Counter for the above buffer + length names for cassettes I/0

Flags for Editor lines on screen
(80=individual parts, 81=double line, 00=following line)

7AD7	; Line 1
7AD8	; Line 2
7AD9	; Line 3
7ADA	; Line 4
7ADB	; Line 5
7ADC	; Line 6
7ADD	; Line 7
7ADE	; Line 8
7ADF	; LIne 9
7AE0	; Line 10
7AE1	; Line 11
7AE2	; Line 12
7AE3	; Line 13
7AE4	; Line 14
7AE5	; Line 15
7AE6	; Line 16
7AE7	; unused
7AE8	; BASIC Programs start here

The string cache (783BH - 78D2H)

This is a table within the communication area. It is used by the BASIC interpreter to temporarily store text variables (strings) that occur during string additions or some PRINT operations.

The table consists of ten 3-byte entries that are stored one after the other. At the beginning of the table, at 78B3H, there is an address pointer to the next free entry. When the computer is initialized, it is set to the first entry in the table.

Each entry consists of a length byte and an address pointer to the String in String area or in the program table. Entries are assigned from top to bottom and released from bottom to top.

If the table overflows, the error message is displayed

FORMULA TOO COMPLEX

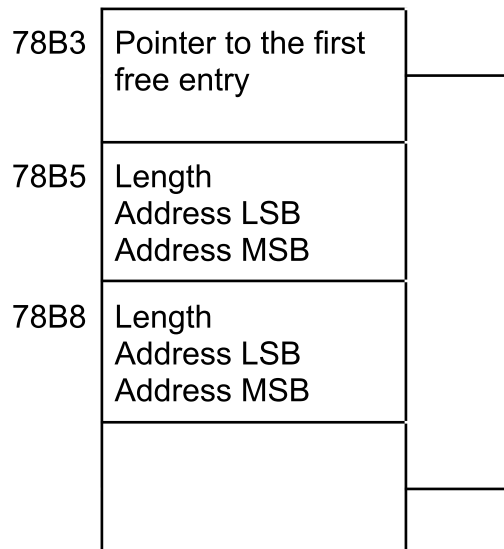


Table of types for every variable (7901H - 791A)

This table is used by the BASIC interpreter to determine the data type of a variable (integer, string, single precision, double precision).

The location of this table is also fixed in the communication area, so no address pointer is required for addressing. The content is changeable; in normal BASIC with the help of the POKE command, in EXTENDED BASIC with the DEFxxx declarations.

The table is 26 bytes long, which corresponds to the number of characters in the English alphabet. There is one byte for each letter from 'A' to 'Z'. The first letter of a variable name is used as an index for table access. Each entry contains a code that provides information about the coresponding variable type:

02 - Integer variable
03 - Text variable (string)
04 - Single precision variable
08 - Double precision variable

When the system is initialized, all entries are marked 04, i.e. as a variable marked with simple precision.

If a variable name already contains a type identifier (e.g. A$ or B1%), this has priority over a different table entry.

Address	Letter	Type at initialization
7901	A	04
7902	B	04
7903	C	04
7904	D	04
7905	E	04
7906	F	04
7907	G	04
7908	H	04
7909	I	04
790A	J	04
790B	K	04

790C	L	04
790D	M	04
790E	N	04
790F	O	04
7910	P	04
7911	Q	04
7912	R	04
7913	S	04
7914	T	04
7915	U	04
7916	V	04
7917	W	04
7918	X	04
7919	Y	04
791A	Z	04

Screen Row Status - Table (7AD7H - 7AE6H)

At the end of the communication area there is a 16-byte table that is required by the screen editor for line management. It contains a 1-byte entry for each of the 16 screen lines with the following content:

80 - this line is a single line of 32 characters.
81 - this line is the first of a double line of 64 characters
00 - this line is the second of a double line of 64 characters.

Programs data (Program Statement Table = PST)

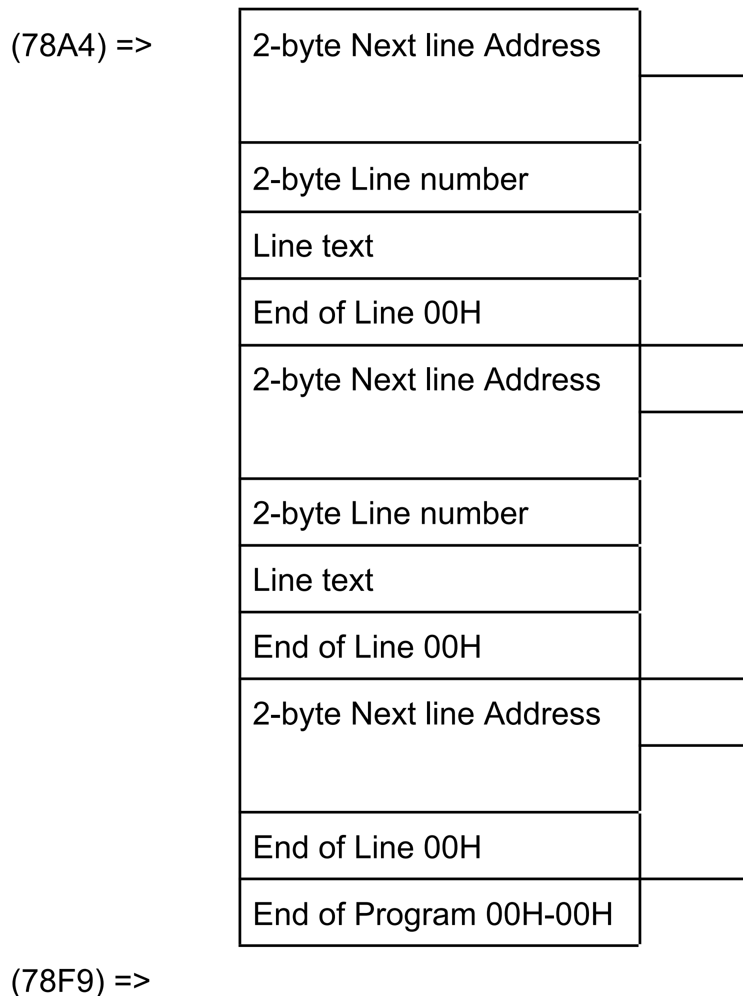
A BASIC program entered or read from a cassette or diskette is stored with all of its instruction lines in the program table. This usually lies directly after the communication area.

Since the program table/area is located in the free memory area of the RAM and its position can change if necessary, in the communication area there is a pointer to the start of the table address at 78A4H and a pointer to the end of the table+1 at 78F9H.

Entered program lines are compressed by the input routine, i.e. BASIC keywords are replaced by TOKENs and transferred to the program table. The individual program lines are saved in ascending order by line number, regardless of the order in which they were entered.

Each program line begins with an address pointer to the beginning of the next line. This is followed by the line number in 2-byte integer (whole number) format. Behind the line number is the actual line text, which ends with a 'Null' byte (00H). This ‘Null’ byte is also known as the “End of Statement” flag or EOS.

The end of the program is marked by two more 'Null' bytes after the last program line.



The content of the program table should be represented using a simple example of two program lines.

```
420 IF A = 25 THEN 500
430 A = A + 1
```

Pointer from previous line = 8132H

```
8132    45 81       ; Pointer to next line 8145H
8134    A4 01       ; Line number 01A4H (420 dec)
8136    8F          ; IF (BASIC token)
8137    20          ; space
8138    41          ; A (variable name)
8139    20          ; space
813A    D5          ; = (BASIC token)
813B    20          ; space
813C    32 35       ; 25 (number stored as char sequence)
813E    20          ; space
813F    CA          ; THEN (BASIC token)
8140    20          ; space
8141    35 30 30    ; 500 (number stored as char sequence)
8144    00          ; End of line (Null byte)
8145    53 81       ; Pointer to next line 8153H
8147    AE 01       ; Line number 01AEH (430 dec)
8149    41          ; A (variable name)
814A    20          ; space
814B    D5          ; = (BASIC token)
814C    20          ; space
814D    41          ; A (variable name)
814E    20          ; space
814F    CD          ; + (BASIC token)
8150    20          ; space
8151    31          ; 1 (number stored as char sequence)
8152    00          ; End of line (Null byte)
8153    …
```

The variables - table

This table contains all variables that have been defined and assigned in a BASIC program. The table is divided into two sections. Section 1 contains all simple variables, section 2 contains all dimensioned variables (arrays).

Three address pointers in the communication area provide information about the location of the variable table in the RAM area.

78F9H	- start address of the 1st section (simple variable)
7BFBH	- start address of the 2nd section (arrays)
78FDH	- end address of the variable table + 1 (= start of the free memory)

	Program data
(78F9H)	simple variables
(78FBH)	dimensioned variables (arrays)
(78FDH)	free memory area

Regardless of which section a variable is defined in, the first three bytes of each entry have the same format.
Byte 1 contains the type code of the variable (02, 03, 04 or 08), which corresponds to the length of the value part. Bytes 2 and 3 contain the variable name in the order 'second letter/first letter'.

In the first case (simple variable) this is followed by the value in the sequence LSB …NSB … MSB or in the case of text variables (strings) ‘length’ and ‘address’ to the string in the string area or in the program table.

Entries in section 2 have another header entry after the 3-byte header, which contains information about the size of the array.

The variables are entered into the variable table during the program execution, whenever a value is assigned or a matrix is created using a DIM instruction.

New variables are simply added within the sections; there is no alphabetical sorting order. Since entries are only made when a variable appears during program execution, it may happen that the entire second section has to be moved in memory to make room for a simple variable in the first section.

Example:
During the program process, the variables A, B and C(5) have already been defined and entered into the variable table.

If a variable D now follows, the matrix C(5) must be moved in the second section of the table so that D can be inserted at the end of the first section.

Matrices are stored in section 2 so that the indices are processed from left to right

Example: DIM(2,3) would be saved like this:

E(0,0)
E(1,0)
E(2,0)
…
E(0,3)
E(1,3)
E(2,3)

The position of each element in the matrix can be determined using the following formula:

(Example of a three-dimensional matrix)

INDEX = (DRI * GMI + DMI) * GLI + DLI

where:

GMI = limit middle index + 1
GLI = limit left index + 1
DRI = defined right index
DMI = defined middle index
DLI = defined left index

Example: A matrix was defined with DIM A(5,4,4)

We are looking for the element A(3,1,2).

That makes: GMI = 5, GLI = 6
DRI = 2, DMI = 1, DLI = 3

INDEX = (2 * 5 + 1) * 6 + 3 = 69

A(3,1,2) is the 69th element of the matrix.

The routine for calculating the indices can be found in the ROM at address 2795H.

Examples of different entries in the variable table (assuming table starts at 8000H)

1. Simple variables

```
; Variable statement C% = 100
8000   02               ; Variable type 02 (Integer)
8001   00 43            ; C - variable name
8002   64 00            ; 0064H (100 dec) variable value

; Variable statement D = -4
8004   04               ; Variable type 04 (single precision)
8005   00 44            ; D - variable name
8007   00 00 80 81      ; -4 dec variable value (float format)

; Variable statement A$ = “XYZ”
800B   03               ; Variable type 03 (string)
800C   00 41            ; A - variable name
800E   03               ; 3 - string length
800F   nn nn            ; address of string characters
```

2. One-dimensional array

```
; Variable statement DIM A(20)
9000   04               ; Variable type 04 (single precision)
9001   00 41            ; A - variable name
9002   nn nn            ; Length of array = Distance to the next array
9004   01               ; 1 - number of dimensions
9005   15 00            ; 0015H (21 dec) - max index + 1
9007   LSB NSB MSB EXP  ; A(0) value
900B   LSB NSB MSB EXP  ; A(1) value
900F   LSB NSB MSB EXP  ; A(2) value
…
xxxx   LSB NSB MSB EXP  ; A(20) value
xxx+4  …                ; next array variable data
```

3. Three-dimensional array

```
; Variable statement DIM A(4,5,9)
9000   04                 ; Variable type 04 (single precision)
9001   00 41              ; A - variable name
9002   nn nn              ; Length of array = Distance to the next array
9004   03                 ; 3 - number of dimensions
9005   0A 00              ; 000AH (10 dec) - max right index + 1
9007   06 00              ; 0006H (6 dec) - max middle index + 1
9009   05 00              ; 0005H (5 dec) - max left index + 1
; 300 4-byte enries
900B   LSB NSB MSB EXP    ; A(0,0,0) value
900F   LSB NSB MSB EXP    ; A(1,0,0) value
9013   LSB NSB MSB EXP    ; A(2,0,0) value
…
xxxx   LSB NSB MSB EXP    ; A(4,5,9) value
xxx+4  …                  ; next array variable data
```

10. The use of the BASIC Stack

Before the string area, at the end of the memory available for BASIC, is the stack area of the BASIC interpreter.

Normally, the communication area serves the BASIC interpreter as a notepad for temporarily storing work values and addresses. However, this does not always work well, as some routines may call themselves internally (recursion) and the previously saved intermediate results were overwritten. An indexed table would also help here, but the BASIC interpreter uses the stack to do this.

In addition to the normal buffering of register contents and return addresses, the stack is used by the BASIC interpreter primarily for three special functions:

- FOR/NEXT - loops
- GOSUB - Views
- Expression analysis

Stack usage in a FOR/NEXT loop

When a FOR statement occurs, all required variable addresses are packed onto the stack in a FOR block. When a NEXT statement occurs, the FOR block with the corresponding run variable is searched on the stack. This search routine is located at 1936H.

The stack is searched from back to front. If no suitable FOR block is found, the error message is displayed

NEXT WITHOUT FOR

Format of a FOR block!

```
0000  81                    ; FOR BASIC token
0001  LSB MBB               ; Address of loop variable
0003  LSB NSB NSB MSB       ; loop Step value
0007  LSB NSB NSB MSB       ; loop Final value
000B  LSB MSB               ; Line number with FOR statement (binary)
000D  LSB MSB               ; Address of the first loop statement
```

Stack usage for a GOSUB instruction

When a G0SUB instruction occurs, a 7-byte block is written to the stack, which is determined and evaluated by a subsequent RETURN.

Format of a GOSUB block:

```
0000  91                    ; GOSUB BASIC token
0001  LSB MSB               ; Line number with GOSUB instruction
0003  LSB MSB               ; Address of the GOSUB instruction
                            ; in the program table
```

11. Expression analysis

Expression analysis breaks down expressions into their individual elements and links them according to the precedence of the operators within the expression.

Each expression is searched and the most significant operation is carried out first. The resulting intermediate result is cached and the next higher operation in expression is determined and executed. This continues until the expression has been completely resolved.

An expression is searched from left to right. The search stops when an operator or the end of the expression is found. The variable to the left of the found operator (called the current variable), together
with the operator (an arithmetic shortcut symbol +, -, *, /, [ ) are called a set and either

- if the rank of the operator was greater than the previous one, the set is written to the stack as a record or
- if the rank of the operator was equal to or lower than the previous one, the variable is linked to the previous sentence from the stack.
  The previous sentence is removed from the stack and the result of the link is considered a new “current variable”.

This is repeated until a new record created in this way has been pushed onto the stack or there are no more values on the stack to link. In such a case the expression was completely dissolved.

The variable/operator sets are written to the stack in the following format:

Rank value of the preceding operator (for the 1st entry =0)		XXXXXX XXXXXX XXXXXX	
Continuation address after a link (usually 2346H)			
Value of the variable			
Type code of the variable			TOKEN of the operator after the variable (0=+, 1=-, 2=*, 3=/, 4=[, 5=AND, 6=0R)
Address of the link routine (for + - * / = 24086)			
Rank value of the operator		XXXXXX XXXXXX XXXXXX	

Checking whether a link should take place or not is relatively easy. The rank value of the operator in the last sentence is the last entry on the stack. It checks whether the new operator's rank value is the same or smaller. If not, the new operator and current variable are written to the stack as a new set. If yes, a link is performed,

In the case of a link, the last block is completely fetched from the stack and the link routine specified there (normally at 2406H) is started. There the previous variable is linked from the stack with the current variable according to the previous operator. The result becomes the new current variable, which forms a new sentence with the current operator.

After the link, the system jumps back to where it is checked again whether the new record now formed is lower in rank or equal to that of another record on the stack.

If there is no set left on the stack or if the existing set has a lower rank value than the current set, the newly formed set is written to the stack. Otherwise a link will take place again.

The end of an instruction or the occurrence of a non-arithmetic TOKEN always triggers a link.

The following example is intended to illustrate such an expression analysis in individual steps:

Expression: A = B + C * D / E [5

The search begins with the first character to the right of the equal sign and initially ends at the '+' sign.

'B' and '+' are written to the stack as the first sentence because there was no sentence there yet for comparison or the 0 stored there during initialization suggests a lower rank value.

The search continues and is interrupted again at '*'. The variable/operator set 'C *' is written to the stack as the second set because the rank value of the operator '*' is greater than that of the previous '+' on the stack.

The stack now looks like this:

```
00          XXXXXX
     2346
    Value of B
04          00                Set 1
     2406
79          XXXXXX
     2346
   Value of C
04          02                Set 2
     2406
7C          XXXXXX
```

Another interruption occurs at the character '/'. Now a join needs to be performed because the rank values of '*' (on the stack) and '/' (the new operator) are the same.

Set 2 is read from the stack and a branch is made to the link routine at 2406H. There the link between 'C *' and the current variable 'D' is carried out, i.e. 'C' is multiplied by 'D'. This results in a new current variable, the product of 'C * D'.

After the multiplication, the program continues at 2346H and the new variable/operator set 'C * D /' is compared in rank value with set 1 on the stack. Since the rank value of '/' is greater than that of the first set ('+'), 'C * D /' is pushed onto the stack as a 2nd set.

Now the stack has the following content:

```
00          XXXXXX
     2346
    Value of B
04          00                  Set 1
     2406
79          XXXXXX
     2346
   Result of C * D
04          03                  Set 2
     2406
7C          XXXXXX
```

The analysis of the expression continues after the '/' and stops again when the character '[' is reached. The current variable/operator set 'E[' is pushed onto the stack as set 3 because the rank value of '[' is greater than that of the previous '/',

Now the stack has the following content:

```
Set 1 and Set 2
   as before
     2346
    Value of E
04          04                  Set 3
     2406
7F          XXXXXX
```

If you continue the search process, the end of the expression will be found after the number '5'. As stated before, reaching the end of the statement triggers a shortcut.

Set 3 is popped from the stack and 'E [5' is calculated. The result is the new current variable.

Since the current operator is still reaching the end of the expression, sentence 2 and sentence 1 are also fetched from the stack one after the other and the calculated.

C * D / E [5

and last

B + C * D / E [5

is carried out.

If the program then continues at 2346H, there are no further elements of the expression on the stack and the program returns to the calling routine.

The expression has been evaluated, the result is in work area 1 of the communication area (X register) and is assigned from there to the variable 'A'.

12. Function derivatives

The BASIC interpreter supports 16 arithmetic functions, including the following 7 mathematical functions.

Sine	(sin)	(a)
Exponential function	(e)	(b)
Arc tangent	(arctan)	(c)
Natural logarithm	(ln)	(d)
Cosine	(cos)	(e)
Root function	(x)	(f)
Tangent	(tan)	(g)

The functions (e) to (g) can be calculated by the functions (a) to (d).

(1) $\cos \varphi = \sin (\varphi + \Pi/2)$ φ in radians

(2) $\tan \varphi = \sin \varphi / \cos \varphi$ φ in radians

(3) sqrt(x) = e ^ (½ ln x) e = Euler's number (2.71 .... }

If (3) is generalized, we get!

(4) x ^ y = e ^ (y ln x)

For internal calculation, the BASIC interpreter uses arithretic approximations of the functions (a) to (d). All other functions are calculated using the approximations and the functional relationships (1) to (4).

Sine

According to the laws of power series expansion, the sine can be expressed in a series.

$$(5)\quad \sin\varphi_0 = \sum_{n=0}^{\infty} (-1)^n \frac{\varphi^{2n+1}}{(2n+1)!} \qquad |\varphi| \leq \pi/2$$

Development point $\varphi_0 = 0$ in radians

$|f(x)|$ Sum $f(x)$

$$= \frac{\varphi^1}{1} - \frac{\varphi^3}{1*2*3} + \frac{\varphi^5}{1*2*3*4*5} - \ldots\ldots$$

The BASIC interpreter approximates the sine up to the 5th term. The angle is given in multiples of the circular arc (t).

$$(5.1)\quad \varphi = 2\pi t \qquad ;\ \varphi \text{ in radians}$$

$$(5.2)\quad \sin(2\pi t) = 2\pi t - \frac{(2\pi)^3}{3!} t^3 + \frac{(2\pi)^5}{5!} t^5 - \frac{(2\pi)^7}{7!} t^7 + \frac{(2\pi)^9}{9!} t^9$$

If x is given in degrees, t must also be calculated.

$$(5.3)\quad t = \frac{\varphi}{2\pi} = \frac{x}{360^\circ} \qquad ;x \text{ in degrees, } \varphi \text{ in radians, } |t| \leq 1/4$$

To determine the sign for angles between 0 and 360 degrees (| φ | < 2Π) t is calculated:

$$(5.4)\quad t = \frac{x}{360^\circ} \qquad ;\ 0^\circ < x \leq 90^\circ$$

$$(5.5)\quad t = \frac{180^\circ}{360^\circ} - \frac{x}{360^\circ} \qquad ;\ 90^\circ < x \leq 270^\circ$$

$$(5.6)\quad t = \frac{x}{360^\circ} - \frac{360^\circ}{360^\circ} \qquad ;\ 270^\circ < x \leq 360^\circ$$

For larger angles, the number of integer multiples of a circular arc must be subtracted and then the procedure must be followed as above.

The elements of the individual series members are accurate to four decimal places.

The maximum error of the approximation is < 0.0000035 (for |t| < 1/4). This means that the overall result can be specified with an accuracy of 5 digits.

Exponential function

The BASIC interpreter calculates the exponential function e ^ x for all values in range:

$$-88 < x < 88$$

The function approximation has two elements. One element is an integer exponent with base 2, the second is a series expansion in 8 terms.

(6) $$e^x = 2^{x \log_2 e}$$

(6.1) $$e^x = e^{-t} \left(2^{[\![x \log_2 e]\!] + 1} \right)$$; $[\![f(x)]\!]$ = **Step function largest integer in f(x), known as an integer value.**

$$0 < t \leqslant \ln 2$$

(6.2) $$e^x = T e^{-t}$$; $T = 2^{[\![x \log_2 e]\!] + 1}$

Here e^(-t) describes the difference between e^(x) and the next largest integer multiple of log 2 e as an exponent.

(6.3) $$t = -x + [\![x \log_2 e]\!] \ln 2 + \ln 2$$

(6.4) $$x = \ln e^x = \ln \left(e^{-t} (2^{[\![x \log_2 e]\!] + 1} \right)$$

$$= -t + \ln 2 \left([\![x \log_2 e]\!] + 1 \right) \quad ; 0 < t \leqslant \ln 2$$

The integer power to base 2 can be determined directly (binary calculation systems).

The power series for the second element is general:

$$(6.5) \qquad e^{-t} = 1 + \sum_{n=0}^{\infty} \frac{(-t)^{n+1}}{(n+1)!}$$

$$= 1 - x + \frac{t^2}{1*2} - \ldots\ldots\ldots$$

The results found from (6.5) and (6.2) give e^x to 5 decimal places.

Arctangent

The approximation of the inverse tangent is based on the series expansion up to the 9th term.

$$(7.1) \qquad \arctan x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)}$$

$$= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \ldots\ldots$$

If the output values are negative, the amount of x is calculated and the result is then inverted. For values x > 1, the approximation for 1/x must be calculated.
The result is as follows!

$$(7.2) \qquad \arctan x = \frac{\pi}{2} - \arctan \frac{1}{x} \qquad ; |x| > 1$$

For the values |x| < 1, the result results directly from the series development. The coefficients 5-7 have been corrected so that the maximum error is 0.026 (in radians).

$$(7.3) \qquad \arctan x = x - 0.33331\, x^3 + 0.199936\, x^5 - 0.142089\, x^7 + 0.106563\, x^9 - 0.0752896\, x^{11} + 0.0429096\, x^{13} - 0.01616157\, x^{15} + 0.00286623\, x^{17}$$

Natural logarithm

The approximation of the natural logarithm is calculated from three terms of the associated series.

$$(8.1) \qquad \ln\left(\frac{1+a}{1-a}\right) = \sum_{n=0}^{\infty} \frac{1}{2n+1} a^{2n+1} \qquad ; \text{für } |a| < 1$$

$$(8.2) \qquad x = \frac{1+a}{1-a} \qquad ; \text{für } |a| < 1$$

This results in:

$$(8.3) \qquad \ln x = 2\left[\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 \ldots\right]$$

This series converges for values x < 1. Therefore x must be transferred into these areas by scaling.

$$(8.4) \qquad x = y\, 2^n \qquad ; 1/2 \leq y < 1 \,,\; n = 0,1,2\ldots$$

The scaling factor is the next higher power of two with an integer exponent.

$$(8.5) \qquad \ln x = \ln(y\, 2^n) = \ln y + n \ln 2$$

$$(8.6) \qquad \ln x = \left(\frac{\ln y}{\ln 2} + n\right) \ln 2$$

$$= \ln 2 \left(\frac{\ln\left(\frac{y}{\ln 2}\right)}{\ln 2} + \frac{\ln(\ln 2)}{\ln 2} + n \right)$$

The constants of this formula are specified internally:

$$\ln 2 = 0.0707092$$

$$\frac{\ln(\ln 2)}{\ln 2} = -0.5$$

$$A = \frac{\ln\left(\frac{y}{\ln 2}\right)}{\ln 2} \tag{8.7}$$

The scaled value 'A' can be calculated using the approximation (8.8).

$$A = \frac{2}{\ln 2}\left(\frac{\frac{y}{\ln 2} - 1}{\frac{y}{\ln 2} - 1} + \frac{1}{3}(\ldots)^3 + \frac{1}{5}(\ldots)^5 \right) \tag{8.8}$$

$$\ln x = 0.707092\,(A - 0.5 + n) \tag{8.9}$$

The precision is given in four digits. This precision cannot be achieved for values of x near 0 or for very large values.

13. Subroutines of the BASIC interpreter

The BASIC interpreter of the LASER computers 110, 210, 310 and the VZ200 consists of a large number of self-contained routines that are called by the execution controller to perform specific functions. Many of these routines can also be used by machine programs, thus simplifying program creation.

In this chapter, a number of these routines are described and their integration into machine programs is shown using call examples.

When using each individual routine, it is important to have precise knowledge of the input and output requirements. Where do the input parameters have to be provided, in what data format do they have to be provided and where and how is the result provided? These are just some of the questions that arise. It is also important to know which registers will be changed by the routine to be called so that you can save them in advance.

In this context, it should also be remembered again that when connecting a floppy disk system, the register pair IY must not be changed.

Input/output routines

This section describes routines that deal with the interpreter's communication with its environment,

How can characters be read from the keyboard, displayed on the screen or output to a printer? How can you use the cassette interface from machine programs or output sounds on the small built-in speaker?

Reading from the keyboard

Of course, you are free to evaluate the keyboard matrix yourself. There are also ROM routines available that do this work for you.

CALL 2BH Evaluate keyboard

This routine evaluates the keyboard matrix once and transfers the result. When the button is pressed, the ASCII code is determined and entered into the A register.

You will immediately jump back to the calling routine, regardless of whether a key was pressed or not. If you want to read several characters one after the other, you have to take care of debouncing yourself.

The register pair DE and the A register in which the result is transmitted is changed.

Example:

```
        …
        …
        LD   BC,600H       ; Possibility of debouncing
        CALL 60H           ;
        PUSH DE            ; save DE on stack
        CALL 2BH           ; Read keyboard
        OR   A             ; was the button pressed?
        JP   Z,NOKEY       ; no, to the NOKEY branch
YES:    POP DE             ; restore DE from Stack
        …
        …
```

On return, the A register contains the ASCII code of the key pressed. If no key was pressed, the A register is empty (00H). In the example above, the reading routine is called and, depending on whether a key was pressed or not, either branches to the 'NOKEY' routine or the program continues with the 'YES' routine.

'CALL 2BH' is always useful if you want to see whether someone has rung the doorbell, i.e. whether a button has been pressed, as you walk past while the program is running.

CALL 49H Waiting for keyboard input

Here you will be asked to check whether a key was pressed or not. You don't get control back until someone actually presses a key. Otherwise the routine corresponds to the call 'CALL 2BH', which is also used internally.

Example:

The user should answer a question with 'Y' = Yes or 'N' = No.
The program waits for one of these two buttons to be pressed and branches accordingly to the yes or no routine.

```
        …
        …
LOOP    PUSH DE         ; save DE on stack
        CALL 49H        ; Wait for Press Key
        POP  DE         ; restore DE from Stack
        CP   'Y'        ; 'Y' key pressed?
        JR   Z,YES      ; yes, to YES branch
        CP   'N'        ; 'N' key pressed?
        JR   NZ,LOOP    ; no, keep waiting
NO      …               ; user pressed 'N'
        …
YES     …               ; user pressed 'Y'
        …
        …
```

The characters read in with 'CALL 2BH' or 'CALL 49H' are not displayed on the screen. You may have to do this yourself using one of the following output routines.

CALL 3E3H Reading a line

The routine at 3E3H serves you more comfortably than the previous two calls.

A complete line is read from the keyboard and displayed on the screen. The read line is then made available in the input/output buffer of the communication area for further processing.

The display on the screen begins at the current cursor position and the flashing cursor character is automatically output.

An input line can contain a maximum of 64 characters and must be completed with the <RETURN> key or the <CTRL-BREAK> keys.

Since this routine is used, among other things, to read BASIC programs, only the characters that are valid in it are permitted (letters, numbers, special characters). If you also want to read in block graphic characters or inverted characters, you must enclose them in quotation marks when entering them. If incorrect characters are entered, the text "SYNTAX ERROR" appears and the entry can be repeated.

In order for this routine to function, it is necessary that the interrupts are switched on (EI = enable interrupts).

After return, the read text is in the input/output buffer starting at address 79E8H. The register pair HL points to the byte before it (79E7H). The end of the text is marked by a character 00H.

You can use the carry flag to determine whether the text was completed using <RETURN> or <CTRL-BREAK>.

Carry = 0 completed with <RETURN>
Carry = 1 completed with <CTRL-BREAK>

All registers are changed by the routine.

Example:

A line of text must be read from the keyboard and transferred to the 'TEXT' field. Register B should then contain the text length.

```
            …
            EI                  ; enable interrupts
LOOP        CALL 3E3H           ; Read Line
            JR   C,LOOP         ; BREAK - read again
            INC  HL             ; HL at the start of the buffer
            LD   DE,TEXT        ; DE = address for text
            LD   B,0            ; Character count = 0
NEXT        LD   A,(HL)         ; A - first char from buffer
            OR   A              ; end of text?
            JR   Z,FINISH       ; yes, to FINISH branch
            LD   (DE),A         ; store char in Text memory
            INC  HL             ; address in buffer + 1
```

```
                INC   DE           ; address in Text Memory + 1
                INC   B            ; character count + 1
                JR    NEXT         ; repeat until 0 found
FINISH          …
                …
TEXT            DEFS 64            ; Text memory
```

When finished, the line entered is in the 'TEXT' field. Register B contains the text length.

Display characters on the screen

Three routines are suitable for displaying text on the screen. At 33AH
you can output a single character, with 28A7H and 2B75H a whole character string.

The screen memory for text display is 7000H-71FFH. You can also write something directly in this area, but you should pay attention to screen synchronization; but this possibility is also shown in an example.

CALL 33AH Display character on screen

The character provided in A register is displayed at the cursor position on the screen. The cursor is advanced by one character.

The normal ASCII codes should be used in the A register, not the LASER-internal screen codes.

In addition to the alphanumeric characters, control characters from the ASCII code table are also recognized and the corresponding screen functions are executed.

08H or 18H	Cursor one place to the left
09H or 19H	Move cursor one place to the right
0AH	Cursor down one line
0DH	Cursor to the beginning of the next line
15H	Insert a space (INSERT)
1BH	Cursor up one line
1CH	Cursor in the top left corner
1DH	Cursor at the beginning of the line
1FH	Clear screen
7FH	Delete characters at the cursor position (RUBOUT)

Register contents are not changed, they are saved by the routine when jumping and restored again when jumping back.

Example:

The letter 'A' should be displayed at the cursor position.

```
...
...
LD    A,'A'            ; Load character
CALL  33AH             ; and display on screen
...
...
```

CALL28A7H Output a line

or

CALL2B75H

If you have more than one character to output, you could call 33AH several times in a loop. The routines in 28A7H and 2B75H do this work for you and output a complete text to the screen with a single CALL call.

The start of the text address must be provided in the HL register pair, the end of the text must be marked with 00H, with 28A7H 0DH is also recognized as the end of the text,

The routine at 28A7H uses the string buffer in the communication area and the BASIC string area at the end of the memory to prepare text. You should therefore only use this routine if the communication and string areas are properly managed by your machine program.

The routine at 2B75H performs direct output and is not subject to the restrictions mentioned above. This routine offers you another level of comfort. By simply switching a byte, you can redirect the output from the screen to a connected printer or even to the tape recorder. This is byte 789CH in the communications area.

00H	=	screen
01H	=	printer
80H	=	cassette

But don't forget to put this byte back on the screen afterwards. Otherwise, later system output may also migrate to the other device.

Example:

The screen should be deleted and the text “MY LASER IS AWESOME” should then be displayed.

```
              …
              …
              LD     HL,TEXT          ; address of text to display
              CALL 2B75H              ; and display on screen
              …
              …
TEXT          DEFB 1FH                ; clear screen control char
              DEFM ‘MY LASER IS AWESOME’
              DEFB 0                  ; end of text
```

CALL 1C9H Clear the screen

You can easily get the screen clean with a CALL 1C9H. All that happens internally by outputing the control characters 1DH (cursor to the beginning of the image) and 1FH (clear screen) with two consecutive CALL 33AH.

Example:

```
              …
              …
              CALL 1C9H       ; clear screen
              …
              …
```

Direct output to screen memory

If you want to write directly to the screen memory (7000H-71FFH), you must not use the normal ASCII character set. The character to be output must be provided in LASER's internal encryption, but graphic characters of all colors can also be output directly (see Chapter 5). The process is also suitable for output in high-resolution graphics (7000H-77FFH). Please note that each byte contains information for four pixels.

However, such output should be synchronized with the image generator, otherwise the image quality will suffer (with frequent output, the image will be disturbed by horizontal lines).

The RAM expansion output of the interrupt service routine on 787DH offers one possibility for synchronization. There you can use a jump "C3 xx x' to your own routine. After outputting your text in this routine, you return to the normal interrupt service routine with a simple RETURN (C9H).

Another option, when the interrupt (DI) is switched off, is to query bit 7 in the input/output area 6800H-6FFFH (see Chapter 4). This bit is directly connected to the image generator's vertical sync signal.

Example:

The letter 'A' should be displayed at the beginning of the 3rd line of the screen.

```
            …
            DI                      ; disable interrupts
            LD    HL,7040H          ; address of video RAM for text
            LD    B,'A'             ; load character to be output
WAIT        LD    A,(6800H)         ; check vertical sync
            OR    A
            JP    P,WAIT            ; wait if Bit 7 = 0
            LD    (HL),B            ; output character
            …
            …
```

Output characters to the printer

ROM routines can be used from a machine program to output characters to a connected printer. One of these options has already been discussed with the screen output with the CALL 2B75H and switching the flag with 789CH. This allows entire lines of text to be transferred to the printer. Another possibility is described below.

With a "Seikosha GP100" printer (see Chapter 8), block graphic characters and inverted text characters can also be printed.

The routine for outputting individual characters automatically waits for the printer to “finish”. However, you also have the option of querying the printer status yourself.

CALL 3BH Print a character

The character passed in the A register (in ASCII code) is output to a connected printer.

A line counter is automatically included in the device control block on the 7825H-782CH, which is reset to 0 when there are 66 lines. These 66 lines per page are a system default value and when initializing the communication area this is set and located at address 7828H (number of lines/page + 1). 66 lines/page corresponds to 11 inches and therefore the American sheet format. The German DIN A4 format is approx. 12 inches, i.e. with German paper dimensions, if you want to use the sheet feed control, you should change this value to 72 lines/page (+1 = 49H).

In addition to normal ASCII text characters, control characters can also be transferred to the printer, e.g. escape sequences for font selection (see printer operating instructions)

The following control characters are already recognized and executed by the printer driver:

00H = The printer status is determined and when:
Bit 0 of the A register returned.
- Bit B = 0 - the printer is ready to print
- Bit B = 1 - the printer is not ready

The zero flag is set accordingly.

0BH = Absolute sheet feed.
The printer is moved to the top of the next page,

0CH = Conditional sheet feed.
The printer is only moved to the beginning of the next page if it is not already at the beginning of a page (line counter at 7829H = 0).

0DH = Carriage Return (CR)
A carriage return character (0DH) and a line feed character (0AH) are output.
Attention: You should set your printer so that it does not automatically perform a line feed after a carriage return (see the printer operating instructions).
Otherwise a blank line is always inserted.

0AH = Line feed.
Internally converted to a 0DH before execution.

CALL 5C4H Determine printer status

By calling this routine, the printer status can be queried directly (only the 'BUSY' line is monitored).

The status is transferred in bit 0 of the A register and the ZERO flag is set accordingly.

Bit 0=0	(Z flag = 1)	-	The printer is ready
Bit 0=1	(Z-Flag=0)	-	The printer is not ready

Example:

The text 'TEST A PRINTER OUTPUT' should be printed on the printer.
If it is not ready to print, the text “PRINTER NOT READY” should appear on the screen instead.

```
            ...
            ...
            CALL 54CH         ; Check printer status
            JR   NZ,ERROR     ; not ready
            LD   HL,PRTEXT    ; text address
LOOP        LD   A,(HL)       ; character of text
            OR   A            ; end of text?
            JR   Z,CONT       ; yes, continue rest of program
            CALL 3BH          ; print character
            INC  HL           ; address of next char
            JR   LOOP         ; repeat until end of text
CONT        ...
            ...
ERROR       LD   HL,ERRTXT    ; address of Error Text
            CALL 2B75H        ; display error text on screen
            ...
            ...
PRTEXT      DEFM 'TEST A PRINTER OUTPUT'
            DEFB 0
ERRTXT      DEFM 'PRINTER NOT READY'
            DEFB 0
```

The cassettes - input/output

Communication with a connected cassette recorder takes place via the input/output area 6800H-6FFFH (see Chapter 4). When reading, the information is taken over from the recorder via bit 6, and output to the cassette recorder via bits 1 and 2. In contrast to the screen and printer, the transmission takes place serially, i.e. for a character to be written or read (= 1 byte), 8 bits must be written or read one after the other.

There are a number of routines available that you can use when editing the cassette interface yourself. The standard format of a recording can be found in Chapter 8, section "The Cassette Driver".

Since bit recording is very time-critical, it is important to switch off the interrupts (DI) before any processing (reading or writing), otherwise you will not transmit any useful information.

Write to the cassette

CALL 3511H Write a byte to the cassette

With this routine, a byte is output serially bit by bit to the cassette recorder. The byte to be output must be provided in the A register.

CALL 3558H Write file header to cassette

This outputs a complete file header to the cassette recorder. This consists of the synchronization bytes (255 x 80H), the header (5 x FEH), the file identifier (F0H = BASIC, F1H = binary file, F2H = data file) and the file name (max. 15 characters).

The file name must be provided in a separate field, enclosed in quotation marks. HL must contain the starting address of this field. The file identifier must be passed in the C register.

If the carry bit is set when jumping back, the recording process was interrupted by the <CTRL-BREAK> keys.

Example:

The memory area from 8000H to 8FFFH should be output to the cassette recorder as a binary file called "TEST". A checksum should be written behind the data for later loading control.

```
…
…
DI                       ; disable interrupts
LD    C,0F1H             ; binary file type
LD    HL,NAME            ; address of filename text
CALL  3358H              ; output file header
JP    C,BREAK            ; interrupted by BREAK key
LD    BC,400             ; a short break in between
CALL  60H                ; call delay function
CALL  3AE8H              ; check if BREAK key is pressed
JP    C,BREAK            ; yes, abort
LD    IX,7823H           ; address of checksum bytes
```

```
        LD    HL,8000H       ; address of data to send
        LD    A,L            ; LSB of data address
        CALL  3511H          ; write 1 byte
        LD    (IX),A         ; store to checksum variable
        XOR   A              ; MSB of checksum must be 0
        LD    (IX+1),A       ; store checksum MSB
        LD    A,H            ; MSB of data address
        CALL  3511H          ; write 1 byte
        CALL  388E           ; add to checksum
        EX    DE,HL          ; address of data into DE
        LD    HL,8FFFH       ; address of last data byte
        INC   HL             ; +1 to save to cassette
        LD    A,L            ; LSB of above address
        CALL  3511H          ; write 1 byte
        CALL  388E           ; add to checksum
        LD    A,H            ; MSB of above address
        CALL  3511H          ; write 1 byte
        CALL  388E           ; add to checksum
LOOP    CALL  3AE8H          ; check if BREAK key is pressed
        JP    C,BREAK        ; yes, abort
        LD    A,(DE)         ; data byte to write
        INC   DE             ; increment address for next byte
        CALL  3511H          ; write 1 byte
        CALL  388E           ; add to checksum
        RST   18H            ; End reached? (compare DE and HL)
        JR    NZ,LOOP        ; no, write next byte
        LD    A,(IX)         ; LSB of checksum
        CALL  3511H          ; write 1 byte
        LD    A,(IX+1)       ; MSB of checksum
        EI                   ; enable interrupts
        …
        …
BREAK   …
        …
NAME    DEFM  “TEST”         ; filename
        DEFB  0
```

In the example above, some routines were called that have not yet been described.

CALL 3AE8H Query BREAK key

This routine checks whether the <CTRL> and <BREAK> keys were pressed at the same time (BREAK function). If this is the case, the CARRY bit is set.

CALL 388EH Create checksum

This routine is used by the cassette recording routine and the reading routine to determine the checksum of a recording. For this purpose, the two bytes 7823H and 7824H are available in the communication area, which are to be addressed and initialized with the register pair IX.

RST 18H Compare HL with DE

Here the register pair HL is logically compared with the register pair DE. The CARRY and ZERO flags are set according to the result of the comparison.

This routine is described in detail when discussing the RESTART procedures.

Reading from the cassette

CALL 3775H Read one byte from cassette

This reads a single byte from the cartridge and makes it available in the A register. The register pairs BC, DE and HL remain unchanged.

In the event of read errors, the CARRY bit is set.

Before reading a byte, the reading routine must be synchronized to a valid record.

When reading several bytes, the 600 baud rhythm must be adhered to.

CALL 35E1H Search for file on the cassette

This routine determines the beginning of a file on the cassette and synchronizes the reading routine to the recording.

The name of the file to be searched for must be stored in the communication area starting at address 7AB2H and ending with 00H. The routine at 358CH can also be used for this, to which the start address of the name field in HL must be passed.

If a file with the specified name is found on the cassette, the file identifier is transferred to field 7AB2H and can be checked there.

During the search process, messages about the search status are displayed in the last line of the screen:

WAITING - no synchronization bytes have been found yet.

FOUND X:Filename

A file with the specified name was found.
'X:' = file identifier
T = text file (e.g. BASIC program)
B = Binary file (e.g. machine programs)
D = data file

If the name specified does not correspond to the file you are looking for, the search process continues automatically and several FOUND messages may appear in a row.

The above messages can be suppressed by entering a value other than 0 in byte 784CH of the communication area.

If the BREAK button is pressed (CTRL-BREAK) during the search process, you will not jump back to the calling program, but to the BASIC main loop.

Example:

The memory area recorded in the previous example should be read in again. The checksum must be determined and checked at the end of the recording. The message output should be suppressed.

```
        ...
        ...
        DI                  ; disable interrupts
        LD    A,1           ; suppress message output
        LD    (784C),A
        LD    HL,NAME       ; address of filename text
        CALL  358CH         ; copy filename to Communication Area
NEXT    CALL  35E7H         ; search file on cassette
        LD    A,(7AD2)      ; Name found, check file Type
        CP    0F1H          ; is this binary file?
        JR    NZ,NEXT       ; no, search next file
        LD    IX,7823H      ; address of checksum
        CALL  3868H         ; Read start and end address
        JP    C,ERROR       ; Reading error!
        OR    A             ; clear Carry flag
        SBC   HL,DE         ; Determine program length
        JP    C,ERROR       ; start address > end address
        PUSH  HL
        POP   BC            ; program length into BC
LOOP    CALL  3775H         ; read 1 byte from cassette
        JP    C,ERROR       ; Reading error!
        LD    (DE),A        ; stroe byte into memory
        CALL  388EH         ; update checksum
        INC   DE            ; destination address + 1
        DEC   BC            ; bytes to read -1
        LD    A,C           ; = 0?
        OR    B             ; BC= 0?
        JR    NZ,LOOP       ; no, read next byte
        CALL  3775H         ; read LSB of checksum
        CP    (IX)          ; is the same as computed
        JP    NZ,ERROR      ; no, Reading error!
        CALL  3775H         ; read MSB of checksum
        CP    (IX+1)        ; is the same as computed
        JP    NZ,ERROR      ; no, Reading error!
        EI                  ; enable interrupts
        ...
        ...
ERROR   ...
        ...
NAME    DFM   '"TEST"'      ; filename (quoted)
```

Two auxiliary routines were used in the example, which will be explained briefly.

CALL 358CH Transfer file name

The routine is used to transfer a file name located in the program into the communication area starting at address 7A9DH.

The file name must be addressed with HL before being called and must be enclosed in quotation marks.

CALL 386BH Load start and end address

If the correct file has been found on the cassette, the start and end addresses of the memory area can be read from the cassette.

When returning, DE contains the start address and HL the end address + 1 of a text or binary file.

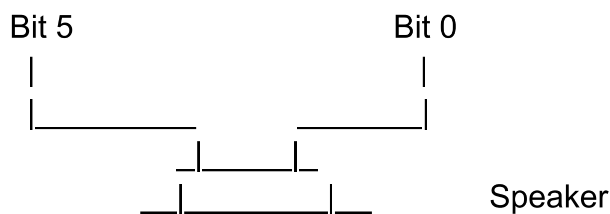
The checksum is initialized by this routine.

If the CARRY flag is set when returning, a read error occurred.

Speaker - output

You can also address the small built-in loudspeaker of the LASER computers and the VZ200 from machine programs.

As already described at the beginning, this loudspeaker is hard-wired to bits 0 and 5 of the input/output area 6800H-6FFFH.



These two bits must always be complementary. A tone is produced by switching bits at a specific frequency. However, you don't need to worry about having to do this yourself from machine programs. Two ROM routines allow you to output either a single tone or an entire melody with one call.

CALL 345CH Emit a single tone

If the correct file has been found on the cassette, the start and end addresses of the memory area can be read from the cassette.

When calling, the pulse length must be specified in the HL register pair (according to the table from 2CFH) and the tone duration in the BC register pair.

All registers are changed.

To produce a clean tone, interrupts should be turned off.

Example:

```
…
…
DI                      ; disable interrupts
LD    HL,0A0H           ; Load pulse length
LD    BC,6              ; Load tone duration
CALL  345CH             ; Output sound
EI                      ; enable interrupts
…
…
```

A high, short beep will be emitted.

The higher the value in HL, the darker the tone becomes. The duration of the sound also depends on the pitch. In order to achieve the same duration for two different tones, a larger value for BC must be selected for the higher tone.

CALL 2BF5H Play a melody

This routine allows a complete melody to be played in one call. The individual notes must be specified as with a BASIC SOUND command:

tone,length; tone,length; tone,length; …

The melody must be provided as an ASCII string in the program and addressed with the HL register pair. The tone and length must be separated by a comma; several tones can be specified one after the other, separated by semicolons.

Example: Playing a little melody

```
        ...
        ...
        LD   HL,MELODY        ; address of melody data
        CALL 3BF5H            ; Play melody
        ...
        ...
MELODY  DEFM '16,2;21,2;21,2;23,2;23,2;25,2;26,1;'
        DEFM '28,2;26,2;25,2;23,2;23,2;21,4;'
```

Conversion routines

Data type conversion

A series of subroutines serves only to provide the numeric data of the correct type before processing, i.e. to convert it from one data type to another.

The conversion routines expect the value to be converted in working register 1 of the communication area (X register) and the data type of this value in the type flag at address 78AFH.

The result is made available again in work area 1. After the conversion, the type flag 78AFH contains the identifier of the new data type.

CALL 0A7FH Floating point number into integer

The contents of workspace 1 are converted from a single or double precision variable to an integer.

All registers are changed.

There is no rounding.

Example:

The single precision value 2.88539 is to be converted to integer.

```
            ...
            ...
            LD    DE,INVALUE      ; address of value to convert
            LD    HL,7921H        ; address of workspace 1 (X Reg)
            LD    BC,4            ; 4 bytes to copy
            LDIR                  ; do copy value to X-Register
            LD    A,4             ; Type-Flag=4 (single precision)
            LD    (78AFH),A       ; set Type of value inside X-Register
            CALL  0A7FH           ; call conversion routine
            LD    HL,(7921)       ; HL - converted number
            LD    (VAR),HL        ; store new value
            ...
            ...
INVALUE     DEFB 45H,0AAH,38H,82H       ; value 2.88539 stored as
                                        ; floating point number
                                        ; LSB-NSB-NSB-MSB
            ...
VAR         DEFW 0                ; Result field contains the value
                                  ; after the conversion
```

CALL 0AB1H Integer to single precision number

The contents of work area 1 are converted from an integer to a single precision floating point number.

Example:

The number 18569 is to be converted from the integer format into a single precision floating point number and transferred to the VAR field.

```
            ...
            ...
            LD    A,89H           ; LSB of value 18569
            LD    (7921H),A       ; store in X-Register
            LD    A,48H           ; MSB of value 18569
            LD    (7922H),A       ; store in X-Register
            LD    A,2             ; Type-Flag=2 (integer)
            LD    (78AFH),A       ; set Type of value inside X-Register
```

```
            CALL 0AB1H          ; call conversion routine
            LD   HL,VAR         ; address of destination value
            CALL 9CBH           ; move value
            …
            …
VAR         DEFS 4              ; variable area
```

After conversion, VAR contains the value 00H-12H-11H-8FH (LSB-NSB-MSB-EXP). This corresponds to the number 18569 in single precision floating point form.

CALL 0ADBH Integer to double precision number

The contents of workspace 1 are converted from an integer to a double precision floating point number.

Example:

The number 457 is to be converted to a double precision floating point number.

```
            …
            …
            LD   A,91H          ; LSB of value 657
            LD   (7921H),A      ; store in X-Register
            LD   A,2            ; MSB of value 657
            LD   (7922H),A      ; store in X-Register
            LD   A,2            ; Type-Flag=2 (integer)
            LD   (78AFH),A      ; set Type of value inside X-Register
            CALL 0ADBH          ; call conversion routine
            LD   DE,VAR         ; address of destination value
            LD   HL,791DH       ; destination address workarea 2
            LD   BC,8           ; 8 bytes to copy
            LDIR
            …
            …
VAR         DEFS 8              ; variable area
```

After conversion, VAR contains the value 657 as a floating point number double precision (00H-0OH-00H-00H-00H-40H-24H-8AH)

ASCII string to numerical representation

The following three routines convert a numeric ASCII string into one of the three data types.

When entering, the register pair HL must point to the beginning of the string to be converted. The conversion ends when the first non-numeric character is reached.

All registers are changed.

CALL 1E5AH Convert ASCII string to integer

The ASCII string addressed with HL is converted into an integer.

The result is transferred into the register pair DE.

<u>Example:</u>

The ASCII string '16544' is to be converted into an integer.

```
        …
        …
        LD   HL,TEXT      ; address of text to convert
        CALL 1E5AH        ; convert to integer
        LD   (VAR),DE     ; store integer value
        …
        …
TEXT    DEFM ‘16544’      ; ASCII text
        DEFB 0            ; End of text byte
        …
VAR     DEFW0             ; Field to store integer value
```

After conversion, VAR contains the value 16544 as a 2-byte binary integer (= 40A0H).

CALL 0E6CH Convert ASCII string to binary value of any type

Converts the HL-addressed ASCII numeric string to one of the three binary data types.

If the value of the ASCII string is less than 32768 and the ASCII string does not contain a decimal point, no exponent specification 'E' or 'D' and no type identifier '#°' or '!', the conversion to a 2-byte integer takes place..

If the value is greater than 32767 or the ASCII string contains a decimal point, the exponent specification 'E' or the type identifier '!', it is converted into a single-precision floating point number.

If an exponent specification is 'D' or a type designation '#', the conversion takes place into a floating point number with double precision.

Example:

‘12345’	- Conversion to an integer
‘40516’	- Conversion to single precision
‘12.3’	- Conversion to single precision
‘12345!’	- Conversion to single precision
‘12345#’	- Conversion to double precision
‘123E10’	- Conversion to single precision
‘123D10’	- Conversion to double precision

The result is transferred to work area 1. The type flag shows the type of the result.

Example:

```
        …
        …
        LD   HL,TEXT        ; address of text to convert
        CALL 0E6CH          ; convert to binary value
        …
        …
TEXT    DEFM ‘24657’        ; ASCII text
        DEFB 0              ; End of text byte
        …
```

The string '24457' is converted to an integer and passed into workspace 1 (7921H-7922H). The type flag at 78AFH is set to 02H (= integer).

CALL 0E65H ASCII string to double precision

This is a prelude to the above routine at 0E6CH. This forces conversion to a floating point number with double precision, regardless of the size and configuration of the ASCII string.

Example:

```
                …
                …
                LD      HL,TEXT         ; address of text to convert
                CALL    0E65H           ; convert to double precision
                …
                …
TEXT            DEFM    '24657'         ; ASCII text
                DEFB    0               ; End of text byte
                …
```

Convert binary value to ASCII string

The three following routines convert a numeric value from the binary format into an ASCII string,

CALL 0FAFH Convert content from HL to ASCII

A binary value located in the HL register pair is converted into an ASCII string and displayed on the screen at the cursor position.

This routine is used by the BASIC interpreter to display the line number of a program line on the screen.

Example:

```
                …
                …
                LD      HL,3039H        ; value to convert
                CALL    0FAFH           ; convert to ASCII string
                …
                …
```

The value 12345 (= 3039H) is displayed on the screen.

CALL 132FH Convert integer to ASCII

An integer located in work area 1 is converted into an ASCII string and stored in memory at the location addressed with HL. Registers B and C should be set to a value greater than 6 at entry to suppress the insertion of commas or periods in the ASCII string.

The result is given an end identifier 00H

Example:

```
            …
            …
            LD      HL,456          ; value to convert
            LD      (7921H),HL      ; store in work area 1
            LD      BC,0606H        ; B and C >= 6
            LD      HL,STRING       ; addres of destination buffer
            CALL    132FH           ; convert to ASCII string
            …
            …
STRING      DEFS 6                  ; 6 bytes of buffer
```

After conversion, the result field contains the entry 30H-30H-34H-35H-36H-00H = 00456.

CALL 0FBEH Convert Floating point value into ASCII string

Converting a single or double precision floating point number to an ASCII string. The floating point number must be provided in work area 1. The generated ASCII string is passed to the print buffer for formatted number output at 7930H.

The ASCII string ends with 00H, HL points to the beginning of the buffer when exiting, DE points to the end of the generated ASCII string (00H). In the event of a field overflow, the character '%' is entered in the byte in front of the print buffer (792FH).

During this conversion, convenient formatting of the ASCII string to be generated can be requested. This formatting is controlled by entries in the A, B and C registers.

The individual bits of the A register have the following effects on formatting:

A Register:

Bit 7	= 0	- do not do any formatting
	= 1	- Carry out formatting according to the bits set below.
Bit 6	= 1	- A comma is inserted every 3 places to separate the thousands values.
Bit 5	= 1	- leading spaces of the ASCII string are replaced by '*'.
Bit 4	= 1	- The character '$' must be displayed before the number.
Bit 3	= 1	- A '+' sign must be displayed.
Bit 2	= 1	- The sign must be shown after the number.
Bit 1		- not used
Bit 0	= 1	- ASCII representation with expense output

B Register:

Number of characters to be output to the left of the decimal point.

C Register:

Number of characters to be output to the right of the decimal point.

When calling in 0FBDH instead of 0 FBEH, formatting is suppressed.

Example:

```
        …
        …
        LD      HL,VALUE        ; Conversion of an ASCII string
        CALL 0E6CH              ; into simple precision
        CALL 0FBDH              ; and back to an ASCII string
        …
        …
VALUE   DEFM ‘1234.56’          ; Initial value
        DEFB 0                  ; End of string terminator
```

The output ASCII string ’1234.56° is first converted into a single precision number. The CALL 0FBDH is used to convert back to the original ASCII string, which is now in the range 7930H. .... with a leading space and a trailing 00H.
Register HL contains entry 7930H and register DE contains entry 7938H.
The result field has the following content in hexadecimal:
20H-31H-32H-33H-34H-2EH-35H-36H-00H

Arithmetic routines

These routines perform arithmetic operations between two operands of the same data type,

The routines await the operands in the specified registers or work areas. The type flag should be set to the appropriate data type before calling.

The division routines use the division subroutine in the communications area (7880H-788DH); this must be there intact.

Routines for processing integers

The following 5 routines perform arithmetic operations between two 16-bit integers. The two operands are to be provided in the register pairs HL and DE. With one exception, the content of DE remains unchanged; the result is usually returned in the register pair HL.

CALL 0BD2H Add two integers

Adds the contents of the register pair DE to the contents of the register pair HL. The sum is handed over in HL.

However, if the sum exceeds 2^15 (32767) (overflow), both values are first converted to a single precision floating point number and the operation is repeated. In this case, the result is then available as a single precision value in working area 1 of the communication area. The type flag at 78AFH receives the entry '4'.

Example:

The integers stored in VAL1 and VAL2 must be added.

```
                …
                …
                LD      HL,VAL1         ; value 1
                LD      DE,VAL2         ; value 2
                LD      A,2             ; Type Flag = 2 (Integer)
                LD      (78AFH),A
                CALL    0BD2H           ; execute addition
                LD      A,(78AFH)       ; Type Flag after addition
                CP      2               ; is it still Integer?
                JR      NZ,SPVAL        ; no, now it's single precision
INTVAL          …                       ; yes
                …
                …
VAL1            EQU15                   ; value 1
VAL2            EQU40                   ; value 2
```

CALL 0BC7H Subtract two integers

Subtracts the value in DE from the value in HL. The difference is transferred in the register pair HL.

If an underflow occurs, i.e. the subtraction of two values of unequal sign results in a value > 2^15 (32767), both values are converted into single-precision floating point numbers before subtraction again. The difference is passed as a single precision value in workspace 1. Such a case can be recognized by the type flag, which is set from '2' to '4'.

Example:

VAL2 should be subtracted from VAL1.

```
        …
        LD    HL,VAL1     ; value 1
        LD    DE,VAL2     ; value 2
        LD    A,2         ; Type Flag = 2 (Integer)
        LD    (78AFH),A
        CALL 0BC7H        ; execute subtraction
        LD    A,(78AFH)   ; Type Flag after subtraction
        CP    2           ; is it still Integer?
        JR    NZ,SPVAL    ; no, now it's single precision
INTVAL  …                 ; yes
        …
VAL1    EQU50             ; value 1
VAL2    EQU30             ; value 2
```

CALL 0BF2H Multiplication of 2 integers

The content of HL is multiplied by the content of DE. The product is ready then in HL.

In case of an overflow (product > 2^15), both values are converted to single precision floating point numbers and the multiplication is performed again. In this case, the product is in work area 1, the type flag contains the value 4.

Example:

VAL2 should be subtracted from VAL1.

```
        …
        …
        LD    HL,VAL1     ; value 1
        LD    DE,VAL2     ; value 2
        LD    A,2         ; Type Flag = 2 (Integer)
        LD    (78AFH),A
        CALL 0BF2H        ; execute multiplication
        LD    A,(78AFH)   ; Type Flag after multiplication
        CP    2           ; is it still Integer?
        JR    NZ,SPVAL    ; no, now it's single precision
INTVAL  …                 ; yes
        …
```

```
VAL1        EQU18               ; value 1
VAL2        EQU12               ; value 2
```

CALL 2490H Division of integers

The content of DE is divided by HL.

Both values are converted to single precision floating point numbers before division. The quotient is also passed with single precision in working area 1. The type flag at 78AFH receives the entry '4'.

The contents of DE and HL will be destroyed.

Example:

VAL1 is to be divided by VAL2.

```
            …
            …
            LD   DE,VAL1        ; Load dividend
            LD   HL,VAL2        ; Load divisor
            CALL 2490H          ; execute division
            LD   A,(78AFH)      ; Type Flag after multiplication
            …
            …
VAL1        EQU80               ; value 1
VAL2        EQU4                ; value 2
```

CALL 0A39H Comparison of two integers

The contents of HL and DE are compared algebraically. Both register contents remain unchanged.

The result of the comparison is transferred to the A register and the status flags (Z = ZERO flag, C = CARRY flag):

HL > DE	A = 1	
HL = DE	A = 0	Z-Flag = 1
HL < DE	A = -1	C-Flag = 1, S-Flag = 1

Example:

The contents of VAL1 and VAL2 must be compared.

```
              ...
              LD    HL,(VAL1)          ; value 1
              LD    DE,(VAL2)          ; value 2
              CALL  0A39H              ; execute comparison
              JP    Z,EQUAL            ; when VAL1 = VAL2
              JP    C,LESSER           ; when VAL1 < VAL2
GREATER       ...                      ; when VAL1 > VAL2
              ...
VAL1          DEFW80                   ; value 1
VAL2          DEFW4                    ; value 2
```

Single precision aritimetic operations

Five additional routines are available to arithmetically compute single-precision floating point numbers.

These routines expect one argument in register pairs BC and DE and the second argument in work area 1 of the communication area. The result is generally made available in work area 1.

CALL 0716H   Single precision addition

Adding two single precision floating point numbers.

One summand must be provided in BC/DE, the second summand in work area 1. After addition, the sum is in work area 1.

Example:

```
              ...
              LD    HL,VAL1            ; address of value 1
              CALL  9B1H               ; transfer to workspace 1
              LD    HL,VAL2            ;  address of value 2
              CALL  9C2H               ; transfer to BC/DE registers
              CALL  716H               ; execute addition
              ...
VAL1          DEFS 4                   ; value 1
VAL2          DEFS 4                   ; value 2
```

The result of the addition is in work area 1.

CALL 0713H Single precision subtraction

Subtracts a single-precision floating point number in BC/DE from the content of workspace 1. The difference is then workspace 1.

Example:

```
              …
              LD    HL,VAL1        ; address of value 1
              CALL 9B1H            ; transfer to workspace 1
              LD    HL,VAL2        ;  address of value 2
              CALL 9C2H            ; transfer to BC/DE registers
              CALL 713H            ; execute subtraction
              …
VAL1          DEFS 4               ; value 1
VAL2          DEFS 4               ; value 2
```

CALL 0847H Single precision multiplication

Multiplies the single precision value generally found in working area 1 with the content of BC/DE. The product is then in work area 1.

Example:

```
              …
              LD    HL,VAL1        ; address of value 1
              CALL 9B1H            ; transfer to workspace 1
              LD    HL,VAL2        ;  address of value 2
              CALL 9C2H            ; transfer to BC/DE registers
              CALL 847H            ; execute multiplication
              …
VAL1          DEFS 4               ; value 1
VAL2          DEFS 4               ; value 2
```

CALL 2490H Single precision division

Divides a single precision floating point number in BC/DE by the contents of work area 1 (single precision). The quotient is then in working area 1.

Example:

```
              …
              LD     HL,VAL1          ; address of divisor
              CALL 9B1H               ; transfer to workspace 1
              LD     HL,VAL2          ;  address of dividend
              CALL 9C2H               ; transfer to BC/DE registers
              CALL 847H               ; execute division
              …
VAL1          DEFS 4                  ; value 1
VAL2          DEFS 4                  ; value 2
```

CALL 0A0CH Comparison of two single precision values

This routine performs an algebraic comparison of two floating point numbers. These are provided externally in the BC/DE registers and in work area 1.

The result of the comparison is transferred to the A register and the status flags (Z = ZERO flag, C = CARRY flag):

BC/DE > X-Reg	A = -1	C-Flag = 1, S-Flag = 1
BC/DE < X-Reg	A = 1	
BC/DE = X-Reg	A = 0	Z-Flag = 1

Example:

```
              …
              …
              LD     HL,VAL1          ; address of value 1
              CALL 9B1H               ; transfer to workspace 1
              LD     HL,VAL2          ; address of value 2
              CALL 9C2H               ; transfer to BC/DE registers
              CALL 0A0CH              ; execute comparison
              JP     Z,EQUAL          ; when VAL1 = VAL2
              JP     C,LESSER         ; when VAL1 < VAL2
GREATER       …                       ; when VAL1 > VAL2
              …
```

```
VAL1        DEFS 4              ; value 1
VAL2        DEFS 4              ; value 2
```

Double precision arithmetic operations

As for the previous two data types, five routines are available for the arithmetic combination of two double-precision floating-point numbers.

One argument must be provided in work area 1 (791DH-7924H) and the second argument in work area 2 (7927H-792EH). The result is always in work area 1.

CALL 0C77H Double precision addition

Here two double precision floating point numbers are added and the sum is passed to work area 1.

Example:

```
            …
            LD    A,8           ; Type-Flag 8 (double precision)
            LD    (78AFH),A
            LD    DE,VAL1       ; address of value 1
            LD    HL,79D1H      ; address of workspace 1 (X-Reg)
            CALL 9D3H           ; transfer to workspace 1
            LD    DE,VAL2       ;  address of value 2
            LD    HL,7927H      ; address of workspace 2 (Y-Reg)
            CALL 9D3H           ; transfer to workspace 2
            CALL 0C77H          ; execute addition
            …
            …
VAL1        DEFS 8              ; value 1
VAL2        DEFS 8              ; value 2
```

CALL 0C70H Double precision subtraction

Subtracts a double-precision floating point number in workspace 2 from the content of workspace 1. The difference is then in workspace 1.

Example:

```
        ...
        LD    A,8              ; Type-Flag 8 (double precision)
        LD    (78AFH),A
        LD    DE,VAL1          ; address of value 1
        LD    HL,79D1H         ; address of workspace 1 (X-Reg)
        CALL  9D3H             ; transfer to workspace 1
        LD    DE,VAL2          ;  address of value 2
        LD    HL,7927H         ; address of workspace 2 (Y-Reg)
        CALL  9D3H             ; transfer to workspace 2
        CALL  0C70H            ; execute subtraction
        ...
        ...
VAL1    DEFS  8                ; value 1
VAL2    DEFS  8                ; value 2
```

CALL 0DA1H Double precision multiplication

Multiplies the double precision floating point number in workspace 2 with the contents of workspace 1. The product is then in workspace 1.

Example:

```
        ...
        LD    A,8              ; Type-Flag 8 (double precision)
        LD    (78AFH),A
        LD    DE,VAL1          ; address of value 1
        LD    HL,79D1H         ; address of workspace 1 (X-Reg)
        CALL  9D3H             ; transfer to workspace 1
        LD    DE,VAL2          ;  address of value 2
        LD    HL,7927H         ; address of workspace 2 (Y-Reg)
        CALL  9D3H             ; transfer to workspace 2
        CALL  0DA1H            ; execute multiplication
        ...
        ...
VAL1    DEFS  8                ; value 1
VAL2    DEFS  8                ; value 2
```

CALL 0DE5H Double precision division

Divides a double-precision floating point number in workspace 1 by one in workspace 2. The quotient is then in workspace 1.

Example:

```
            …
            LD    A,8              ; Type-Flag 8 (double precision)
            LD    (78AFH),A
            LD    DE,VAL1          ; address of dividend
            LD    HL,79D1H         ; address of workspace 1 (X-Reg)
            CALL  9D3H             ; transfer to workspace 1
            LD    DE,VAL2          ;  address of divisor
            LD    HL,7927H         ; address of workspace 2 (Y-Reg)
            CALL  9D3H             ; transfer to workspace 2
            CALL  0DA1H            ; execute division
            …
            …
VAL1        DEFS  8                ; value 1
VAL2        DEFS  8                ; value 2
```

CALL 0A4FH Comparison of two double precision values

Compares two double precision floating point numbers. These are to be provided in work areas 1 and 2. The result of the comparison is displayed in the A register and the flag register.:

X-Reg < Y-Reg	A = 1	
X-Reg = Y-Reg	A = 0	Z-Flag = 1
X-Reg > Y-Reg	A = -1	C-Flag = 1, S-Flag = 1

Example:

```
            …
            …
            LD    A,8              ; Type-Flag 8 (double precision)
            LD    (78AFH),A
            LD    DE,VAL1          ; address of dividend
            LD    HL,79D1H         ; address of workspace 1 (X-Reg)
            CALL  9D3H             ; transfer to workspace 1
            LD    DE,VAL2          ;  address of divisor
```

```
              LD    HL,7927H          ; address of workspace 2 (Y-Reg)
              CALL  9D3H              ; transfer to workspace 2

              CALL  0A4FH             ; execute comparison
              JP    Z,EQUAL           ; when VAL1 = VAL2
              JP    C,LESSER          ; when VAL1 < VAL2
GREATER       …                       ; when VAL1 > VAL2
              …
VAL1          DEFS  8                 ; value 1
VAL2          DEFS  8                 ; value 2
```

Mathematical routines

The following routines are used to calculate mathematical functions. With one exception, these only receive one argument when called, which is to be passed in work area 1 of the communication area. The type of the argument must be specified in the type flag at 78AFH.

CALL 0977H Determine absolute value ABS(N)

The value located in workspace 1 is converted to its positive equivalent. The result is then also in work area 1.

If the negative integer -32768 is in work area 1, the result is transferred as a single-precision floating point number. The type flag at 78AFH is corrected accordingly.

All data types are permitted as arguments.

Example:

The absolute value of the single precision floating point number in field VAL1 is to be determined.

```
              …
              …
              LD    A,4               ; Type-Flag 4 (single precision)
              LD    (78AFH),A
              LD    HL,VAL1           ; address of value 1
              CALL  7B1H              ; transfer to workspace 1
```

```
                CALL 0977H              ; calculate absolute value
                ...
                ...
VAL1            DEFB 0F2H,80H,0BCH,87H        ; single precision = -94.3456
```

After the operation is completed, the value 94.3456 in single precision appears in work area 1.

CALL 0B37H Finding the next lower integer INT(N)

This routine determines the integer part of a floating point number. This must be provided in workspace 1, the type flag must indicate the correct data type.

If the value size allows it (-32768 to +32767), the result is returned in the integer data type, otherwise the data type remains unchanged. The type of the result can be determined in Type Flag at 78AFH.

The result is in work area 1.

Example:

The integer part of the single precision floating point number in value i must be determined.

```
                ...
                ...
                LD     A,4              ; Type-Flag 4 (single precision)
                LD     (78AFH),A
                LD     HL,VAL1          ; address of value 1
                CALL 7B1H               ; transfer to workspace 1
                CALL 0B37H              ; calculate integer part
                ...
                ...
VAL1            DEFB 00H,6FH,49HH,84H; single precision = 12.5896
```

The result, the number 12, is in work area 1 as the data type 'integer', the type flag at 78AFH has the entry '2'.

CALL 15BDH Determine the arc tangent ATN (N)

The corresponding angle in radians is determined from a tangent value stored in working area 1 as a floating point number. The result is made available as a floating point number in the workspace.

Example:

The angle in radians is to be determined from the tangent value stored in the 'TAN' field and transferred to the 'RAD' field.

```
        …
        …
        LD    A,4              ; Type-Flag 4 (single precision)
        LD    (78AFH),A
        LD    HL,TAN           ; tangent value
        CALL  9B1H             ; transfer to workspace 1
        CALL  15BDH            ; calculate arc tangent
        LD    HL,RAD           ; address of field
        LD    DE,7921H         ; address of workspace 1
        CALL  9D3H             ; copy into result field
        …
        …
TAN     DEFB 3AH,0CH,13HH,80H        ; tangent of 30 st (0.57735)
RAD     DEFS 4
```

After the above routine has been completed, the 'RAD' field contains the value of the angle 30° in radians (91H-0AH-06H-80H = 0.523598)

CALL 1541H Find the cosine of an angle COS (N)

Gets the cosine of an angle specified in radians.

The angle is to be provided as a floating point number in work area 1, the result is also passed as a floating point number in work area 1.

Example:

The cosine of the angle specified in the 'RAD' field must be determined and transferred to the 'COS' field.

```
                …
                …
                LD     A,4               ; Type-Flag 4 (single precision)
                LD     (78AFH),A
                LD     HL,RAD            ; angle value
                CALL 9B1H                ; transfer to workspace 1
                CALL 1541H               ; calculate cosine
                LD     HL,COS            ; address of field
                LD     DE,7921H          ; address of workspace 1
                CALL 9D3H                ; copy into result field
                …
                …
RAD             DEFB 91H,0AH,06H,80H     ; radians for 30 st (0.523598)
COS             DEFS 4
```

After the calculation, the cosine of the angle of 30° is in the 'COS' field (D7H-B3H-5DH-80H = 0.866025)

CALL 1547H Find the sine of an angle SIN (N)

The sine of an angle is determined.

The angle must be provided in radians as a floating point number in working area 1. The result is then also in work area 1.

Example:

The sine of the angle in the 'RAD' field is to be determined and transferred to the 'SIN' field.

```
                …
                …
                LD     A,4               ; Type-Flag 4 (single precision)
                LD     (78AFH),A
                LD     HL,RAD            ; angle value
                CALL 9B1H                ; transfer to workspace 1
                CALL 1547H               ; calculate cosine
```

```
              LD    HL,SIN              ; address of field
              LD    DE,7921H            ; address of workspace 1
              CALL  9D3H                ; copy into result field
              …
              …
RAD           DEFB  91H,0AH,06H,80H     ; radians for 30 st (0.523598)
SIN           DEFS  4
```

After the calculation, the ‘SIN’ field contains the sine of 30°.

CALL 1439H Finding the exponential function e^x EXP (N)

The argument N is to be provided as a single precision floating point number in workspace 1. The result is also transferred in single precision in work area 1.

Example:

e^1.5708 has to be determined.

```
              …
              …
              LD    A,4                 ; Type-Flag 4 (single precision)
              LD    (78AFH),A
              LD    HL,EXP              ; argument value
              CALL  9B1H                ; transfer to workspace 1
              CALL  1439H               ; calculate function
              LD    HL,RES              ; address of Result field
              LD    DE,7921H            ; address of workspace 1
              CALL  9D3H                ; copy into Result field
              …
              …
EXP           DEFB  0DBH,0FH,49H,81H          ; exponent (1.5708)
RES           DEFS  4                   ; Result field
```

JP 13F2H Raise X to the power of Y

The input values X and Y are to be provided as single precision floating point numbers.

The base ‘X' is to be transferred to the stack area, the expander 'Y' is to be provided in work area 1. After the calculation, the result is a single-precision floating point number in working area 1.

There is a special feature to note here. Since one of the arguments must be provided on the stack, the routine cannot be called with CALL, since the return address would then be the last entry on the stack. Rather, the return address must be written to the stack before the argument and the routine must be called with a JP. The following example illustrates this procedure.

Example:

16^4 should be calculated.

```
            …
            …
            LD    HL, RET            ; address to return from call
            PUSH  HL
            LD    A,4                ; Type-Flag 4 (single precision)
            LD    (78AFH),A
            LD    HL,BAS             ; address of base value
            CALL  9B1H               ; transfer to workspace 1
            CALL  9A4H               ; push workspace 1 on Stack
            LD    HL,EXP             ; address of exponent value
            CALL  9B1H               ; transfer to workspace 1
            JP    13F2               ; calculate function
RET         …
            …
            …
BAS         DEFB 00H,00H,00H,85H     ; base = 16
EXP         DEFB 00H,00H,00H,83H     ; exponent = 4
```

CALL 0809H Natural logarithm LOG (N)

Finds the natural logarithm of a single-precision floating-point number. The argument is to be provided in work area 1, the result is also passed as a single precision floating point number in work area 1.

Example:

The natural logarithm of 5 must be determined.

```
            …
            …
            LD   A,4                ; Type-Flag 4 (single precision)
            LD   (78AFH),A
            LD   HL,ARG             ; argument value
            CALL 9B1H               ; transfer to workspace 1
            CALL 0809H              ; calculate function
            LD   HL,LOG             ; address of Result field
            LD   DE,7921H           ; address of workspace 1
            CALL 9D3H               ; copy into Result field
            …
            …
ARG         DEFB 00H,00H,02H,83H    ; argument = 5
LOG         DEFS 4                  ; Result field
```

CALL 13E7H Find root of N SQR (N)

Finds the root of a value stored in workspace 1.
The result is also returned in workspace 1.

Example:

The root of 144 must be determined

```
            …
            …
            LD   A,4                ; Type-Flag 4 (single precision)
            LD   (78AFH),A
            LD   HL,ARG             ; argument value
            CALL 9B1H               ; transfer to workspace 1
            CALL 13E7H              ; calculate function
            …
ARG         DEFB 00H,00H,10H,88H    ; argument = 144
```

After the calculation, the result (= 12) is in work area 1.

CALL 14C9H Generate random number RND (N)

Generates a random number between 0 and 1 or between 1 and N, depending on the value 'N' that must be passed in workspace 1.

The generated random number is returned as a single precision floating point number in workspace 1 and the type flag is modified accordingly.

The passed argument 'N' determines the range of the random number. If N=0, a random number between 0 and 1 is generated. If N > 0, a random number between 1 and N is determined and passed as an integer.

Example:

A random number between 1 and 20 should be determined.

```
...
...
LD    A,4          ; Type-Flag 4 (single precision)
LD    (78AFH),A
LD    HL,20        ; N = 20
LD    (7921H),HL   ; transfer integer to workspace 1
CALL  14C9H        ; generate random number
...
...
```

After the calculation, a random number between 1 and 20 with single precision is in working area 1.

The random number generated in this way is not a real random number, but is formed from the last random number according to a fixed algorithm.

If more randomness is to be introduced, a new base value can be set with a CALL 1D3H, which is taken from the current status of the Z80 refresh register. The base value and the last generated random number are at 78AAH - 78ADH in communication area.

RESTART - vectors

In the lower address area of the Z80 there are so-called restart addresses, which can be reached using a special 'RST' command.

The RESTART vectors 8H to 30H are routed to RAM expansion outputs at the beginning of the communication area. From there, the vectors 8H - 20H jump into ROM routines, which also fulfill useful functions for a machine language or assembler programmer.

It is also possible to assign jump commands to the RAM expansion outputs in your own routines. However, it should be borne in mind that all of the other routines listed in this chapter can no longer be used so easily, as they occasionally also make use of the restart commands.

RST 08H Check a character

This routine is used by the BASIC interpreter to check the syntax of an input line. The character addressed by HL is compared with the character following the RST 8 command. If both are the same, the RST 10 routine is automatically called and from there it jumps back to the 2nd byte after the RST 8 command. The HL register pair contains the next good character determined by RST 10, the A register contains this character.

If the two characters do not match, an error message "SYNTAX ERROR" is generated and a jump is made to the input phase of the BASIC interpreter.

Example:

Checking a text addressed by HL to see whether it consists of the characters 'A=B'.

```
…
…
LD     HL,TEXT          ; address of text to check
RST    8
DEFB ‘A’
RST    8
DEFB ‘=’
RST    8
DEFB ‘B’
…
```

If one of the characters does not match, a SYNTAX ERROR is generated.

RST 10H Find next valid character

This routine is used by the BASIC interpreter to determine and accept the next valid character when analyzing an input line.

The register pair HL is used to address a text string. When jumping in, the address contained therein is incremented by 1. The next character addressed by HL is loaded into the A register and the carry flag is set according to the character type.

Carry = 0 - Character is not numeric
Carry = 1 - Character is numeric

The ZERO flag is set when the end of the text has been reached. This must be identified by a byte = 00H.

When editing the text string, spaces and the control characters 09H (TAB) and 0AH (LF) are not taken into account; they are simply ignored.

<u>Example:</u>

HL points to a program line that contains a value assignment. The line has been edited up to the equal sign. It must be checked whether this is followed by a variable or a constant.

```
          …
          …
NEXT      RST   10H          ; load next character
          JR    NC,NONUM     ; not numeric
          CALL  1E5AH        ; load constant value
          JP    CONTINUE     ; continue program
NONUM     CP    '+'          ; is it a '+' sign?
          JR    Z,NEXT       ; yes, load next character
          CP    '-'          ; is it a '-' sign?
          JR    Z,NEXT       ; yes, load next character
          CALL  260DH        ; Variable in variable table
                             ; determine
          …
          …
```

RST 18H Compare DE with HL

This routine carries out a logical comparison of the two register pairs DE and HL. It does not work correctly with signed integers; for signed integer values the routine 0A39H (arithmetic comparison HL:DE) must be used.

The result is displayed in the ZERO and CARRY flags. The contents of the A register are changed.

Carry = 1	-	HL < DE
Carry = 0	-	HL >= DE
Zero = 1	-	HL = DE
Zero = 0	-	HL <> DE

Example:

It must be checked whether a value in DE is in the range 200 - 500.

```
…
…
LD      HL,500          ; load upper limit value
RST     18H             ; compare with value in DE
JR      C,ERROR         ; value in DE > 500
LD      HL,199          ; load lower limit value
RST     18H             ; compare with value in DE
JR      NC,ERROR        ; value in DE <= 199
…
…
```

RST 20H Determine data type

The type flag at 78AFH is evaluated and a numeric value is returned in the A register depending on the data type displayed there. The flag register can also be evaluated.

Type	Status	A Register
Integer	NZ,C,M,E	-1
String	Z,C,P,E	0
Single prec.	NZ,C,P,O	1
Double prec.	NZ,NC,P,E	5

The value passed in A register corresponds to the type code in 78AFH - 3.

The routine is used by the BASIC interpreter to determine the data type of a value stored in workspace 1. But be careful, type flag and workspace 1 do not always have to be synchronous.

<u>Example:</u>

After adding two integers, it is necessary to check whether the result was passed as an integer or as a single-precision floating point number.

```
        …
        …
        LD   A,2           ; Type-Flag = Integer
        LD   (78AFH),A
        LD   DE, VAL1      ; value 1
        LD   HL, VAL2      ; value 2
        CALL 0BD2H         ; add values
        RST  20H           ; check type of resul value
        JP   M,INT         ; jump if Integer
SP      …                  ; single precision
        …
INT     …                  ; integer
        …
VAL1    DEFW2              ; value 1
VAL2    DEFW2              ; value 2
```

Transfer routines

This section describes some routines that transfer data of various types within memory or between registers and memory.

CALL 09B4H Single precision number from BC/DE to work area 1

Transfers a single precision floating point number from the register pairs BC/DE into work area 1 of the communication area.

The content of HL is destroyed, BC/DE remains unchanged.

Attention: The type flag at 78AFH is not updated.

Example:

Two single precision floating point numbers in memory must be added.

```
                ...
                ...
                LD      DE,(VAL1)       ; MSB + Exponent of 1st number
                LD      BC,(VAL1+2)     ; LSB + NSB of 1st number
                CALL 09B4H              ; transfer to work area 1
                LD      DE,(VAL2)       ; MSB + Exponent of 2nd number
                LD      BC,(VAL2+2)     ; LSB + NSB of 2nd number
                CALL 0716H              ; add numbers
                ...
                ...
VAL1            DEFS 4                  ; value 1
VAL2            DEFS 4                  ; value 2
```

VAL1 and VAL2 must contain the values in the sequence LSB-NSB-MSB-EXP.

CALL 09B1H Single precision number from address HL to work area 1

A single-precision floating point number in memory is transferred to work area 1. HL must contain the starting address of the memory area.

The contents of HL/BC/DE will be destroyed.

Example:

```
                ...
                ...
                LD      HL,VAL1         ; address of single prec. number
                CALL 09B1H              ; transfer to work area 1
                ...
                ...
VAL1            DEFS 4                  ; value 1
```

CALL 09CBH Single precision number from work area 1 to memory

A floating point number in work area 1 is transferred to the program memory. HL must contain the starting address of the memory area,

The contents of all registers are changed.

Example:

```
        …
        …
        LD   HL,VAL1        ; address for single prec. number
        CALL 09CBH          ; transfer from work area 1
        …
        …
VAL1    DEFS 4              ; value 1
```

CALL 09C2H Single precision number from memory into BC/DE regs

HL must contain the starting address of the memory area. The contents of all registers are changed.

Example:

Two single precision numbers are to be added. The result must be transferred to BC/DE.

```
        …
        …
        LD   HL,VAL1        ; address of value 1
        CALL 9B1H           ; transfer to work area 1
        LD   HL,VAL2        ; address of value 2
        CALL 9C2H           ; transfer to BC/DE
        CALL 0716H          ; add numbers
        LD   BC,(7923H)     ; MSB + Exponent of result
        LD   DE,(7921H)     ; LSB + NSB of result
        …
        …
VAL1    DEFS 4              ; value 1
```

```
VAL2        DEFS 4              ; value 2
```

VAL1 and VAL2 must contain the values in the sequence LSB-NSB-MSB-EXP.

CALL 09BFH Single precision number from workspace 1 to BC/DE regs

Transfers a single precision floating point number from the work area 1 into the register pairs BC/DE.

Attention, it is not checked whether work area 1 really contains a single-precision floating point number; this must be ensured by the calling program.

Example:

Two single precision floating point numbers are to be multiplied. The result must be provided in BC/DE.

```
            …
            …
            LD   HL,VAL1        ; address of value 1
            CALL 9B1H           ; transfer to work area 1
            LD   HL,VAL2        ; address of value 2
            CALL 9C2H           ; transfer to BC/DE
            CALL 847H           ; multiple numbers
            CALL 9BFH           ; result ito BC/DE
            …
            …
VAL1        DEFS 4              ; value 1
VAL2        DEFS 4              ; value 2
```

VAL1 and VAL2 must contain the values in the sequence LSB-NSB-MSB-EXP.

CALL 09A4H Transfer workspace 1 to the stack

Pushes a single-precision floating-point number from the workspace onto the stack. This is saved there in the order LSB-NSB-MSB-EXP.

All register contents remain unchanged.

It is not checked whether there really is a single-precision floating point number in work area 1.

This routine is needed, for example, if you want to use the exponentiation routine at 13F2H. There the base is provided on the stack as a single-precision floating point number.

CALL 09D3H Variable transfer routine

Transfers, depending on the data type, a value in the length of the type flag (78AFH) from the address specified in DE to the address specified in HL.

The registers A, DE and HL are changed.

Example:

A double precision variable with the name 'XY' should be determined in the variable table and the value of the variable should be transferred to the program.

```
        ...
        ...
        LD   HL,NAME     ; address variable name
        CALL 260DH       ; variable address into DE
        LD   HL,VAL2     ; address of value 2
        RST  20H         ; is it double precision?
        JR   NC,OK       ; yes, continue
        JP   ERROR       ; no
OK      LD   HL,DP       ; address for variable value
        CALL 9D3H        ; transfer variable
        ...
        ...
NAME    DEFM “XY’        ; name of the variable
        DEFB 0           ; end of text
DP      DEFS 8           ; place fo variable
```

CALL 29C8H Transferring a string variable

A string variable is transferred from the string area or the programs table into a program-internal memory area.

When entering, HL must contain the address of the string variable in the variable table and DE the receive address in the program.

For a string variable, an entry in the variable table has the following format:

1 byte	= length of string
2 bytes	= address of string data (text)

Example:

A string variable with the name 'A$' is to be transferred to an internal program field 'VAR'.

```
           …
           …
           LD    HL,NAME      ; address variable name
           CALL  260DH        ; variable address into DE
           RST   20H          ; is it string variable?
           JR    Z,OK         ; yes, continue
           JP    ERROR        ; no
OK         EX    DE,HL        ; address in Variable Table in HL
           LD    DE,VAR       ; address of buffer
           CALL  29C8H        ; transfer variable
           …
           …
NAME       DEFM  “A$’         ; name of the variable
           DEFB  0            ; end of text
DP         DEFS  255          ; buffer for text
```

BASIC functions

BASIC functions differ from the previous functions mainly in the intensive use of the communication area,

When using the following routines, care must be taken to ensure that the communication area is intact and has not been destroyed or otherwise used by the calling machine program.

The routines are particularly suitable for use in machine program subroutines that are called from a BASIC program.

CALL 1B2CH Determine line in the program

This routine searches the Programs table for a BASIC line with a given line number. The line number to be determined must be provided in the register pair DE.

When entering, HL must contain the address of the string variable in the variable table and DE the receive address in the program.

All registers are changed. When you jump back, the success of the action can be seen from the status flags. BC and HL contain corresponding address information.

Status	**Flags**	**Register contents**
Line found	C/Z	BC = Start address of the line in the program table
Line not found, line number too large	NC/Z	HL/BC = End address of program +1
Line not found, but found larger line numbers in programs.	NC/NZ	BC = Address of the line with the next highest line number HL = Address of the following line in the program.

Example:

The program line with line number 500 should be determined in the program. If present, the value 0 should be passed in the A register; if not present, the value 1 should be passed.

```
        …
        …
        LD   DE,500      ; line number
        CALL 1B2CH       ; search line in programm
        LD   A,1         ; default: line not found
        JR   NC,CONT     ; yes, continue
        XOR  A           ; A = 0: line found
CONT    …                ; continuation
```

```
                ...
```

The register pairs HL and BC contain the required address information according to the table above, depending on the return status.

CALL 260DH Get the address of a variable

This routine can be used to search the variable table for a specific variable. The name of the variable you are looking for must be provided in a program-internal field and addressed with HL.

If the variable does not exist, a new entry is made in the variable table. The value field is set = 0.

When returning, the register pair DE contains the address of the first value entry, the type flag at 78AFH indicates the type of the variable found.

Simple variables or elements of a matrix can be determined with this routine. For a matrix, the index must be added to the name as with a BASIC access, e.g. 'A(20)' for the 20th element of the matrix 'A'.

Example:

The address of the variable 'AB' should be determined.

```
        ...
        ...
        LD   HL,NAME        ; address variable name
        CALL 260DH          ; variable address into DE
        LD   (ADR),DE       ; store addres for future use
        ...
        ...
NAME    DEFM “AB’           ; name of the variable
        DEFB 0              ; end of text
ADR     DEFW 0              ; place for variable address
```

CALL 1EB1H GOSUB Emulation

This routine allows a BASIC subroutine to be called from a machine program. After executing the BASIC routine, the program continues with the command following the CALL.

All registers are changed.

When entering, HL must contain the start address of an ASCII string in which the first line number of the BASIC subroutine is specified.

Example:

A BASIC subroutine, starting at line number 800, is to be called from a machine program.

```
            ...
            ...
            LD    HL,LINENO       ; line number to gosub
            CALL  1EB1H           ; call BASIC subroutine
            ...
            ...
LINENO      DEFM  "800'           ; line number as text
            DEFB  0               ; end of text
```

This is just a small selection of the LASER-ROM routines available. By intensively studying the documented ROM list, a large number of other routines can be localized and made usable for calling from machine programs.

Die Einsteiger-Modelle für Schüler und Studenten

LASER 210, 8 KByte RAM,
erweiterbar um 16 oder 64 KByte,
8 Farben, Programmsprache BASIC.

LASER 310 mit gleicher Ausstattung wie Laser 210,
aber 18 KByte RAM und mit Schreibmaschinen-Tastatur.

Floppy Disk Controller für 2 Laufwerke
mit LASER-DOS, Speicherkapazität 80 KByte.

Generalimporteur: CE ○ TEC Trading GmbH
Lange Reihe 29, 2000 Hamburg 1

aktiv und kreativ computern

Früher oder später wird bei jedem Computerbesitzer der Wunsch nach einem Diskettenspeicher übermächtig. Zu groß sind die Vorteile des direkten Zugriffs gegenüber der sequentiellen Arbeitsweise. Nur so kann man alle Möglichkeiten seines Geräts voll ausschöpfen.

In diesem Band wird das Disketten-Betriebssystem des Laser-Computers in seinem Aufbau und seiner Anwendung detailliert beschrieben. Neben den BASIC-DOS-Befehlen werden auch die Schnittstelle und Anwendbarkeit in Maschinenprogrammen erläutert. Anwendungsbeispiele erleichtern den Einstieg in die Diskettenwelt.

VOGEL-BUCHVERLAG
WÜRZBURG

ISBN 3-8023-0868-9