Update
Tiger Basic™ 5.2

New functions of Version 5.2

Production: Wilke Technology GmbH
Krefelder Str. 147
D-52070 Aachen, Germany
www.wilke-technology.de
info@wilke-technology.de

Cover: Der Design Pool, www.derdesignpool.de
Print: Print Production, www.printproduction.de
Edition: 1st edition July 2004
Art-Code: BTI-MANUE5.2-1

BASIC-Tiger™, TINY-Tiger™, Econo-Tiger™, Tiger-II™, Tiger-Basic™, CoolBase™ are registered trademarks of Wilke Technology GmbH. TouchMemory™ is a registered trademark of Dallas Semiconductors, Windows™ is a registered trademark of Microsoft Corp.

Those designations of products and methods given in this publication, which are trademarks, were not especially indicated . Such names are trademarks of the respective trademark owner. It cannot be concluded from the absence of the ™ mark that the designation is not a registered trademark.

Editors, translators and authors of this publication created texts, figures and programs with due diligence. However, errors are still possible. Wilke Technology does not accept guarantees, legal responsibility or liability for consequences which were caused by incorrect statements. Any feedback concerning errors is welcome.

Information given in this manual is not a warranty of specific product features. Changes, which conduce to technical progress, remain reserved.

Contents

About this book

empty page

Preparations

In order to be able to work with the new version of the Tiger-Basic IDE, you need a PC (development computer) and the BASIC-Tiger target system connected to the PC.

PC system requirements

For using the Tiger-BASIC IDE 5.2 we recommend a PC with the following minimum equipment:

- Pentium (or equivalent) 500 MHz processor or better
- At least 40 MByte of available hard drive space
- SVGA 800 x 600 screen resolution or higher
- Mouse
- Windows 2000 / Me or higher
- 1 free COM port (COM1: ... COM4:)
- CD-ROM drive

BASIC-Tiger target system

Every Tiger proto board and every device containing a BASIC-Tiger computer is applicable as a target system, provided it has a RS-232 interface (Ch-1) and can be set to PC-Mode.

The following minimum memory capacity is recommended:

SRAM	128 KBytes
FLASH	512 KBytes

Safety instructions

BASIC-Tiger computer modules are used as embedded computers in different applications, devices and machines. They are typically used for communication, as user interfaces, for monitoring as well as for controlling devices and plants.

It has to be made sure at any rate that such applications are secured before running program tests and stopping or starting programs. The user has to take suitable measures for avoiding personal injury and material damage in any case, e.g. by switching off the devices’ power supply or by not working with real, confidential data.

Furthermore there are numerous development tools for this IDE, such as prototyping boards, development kits, adapters, cables and accessories. Such components merely serve as a support for developing and testing of computer software and hardware circuitries. Their only purpose is to save the competent developer time when constructing lab set ups and to inspire him to design applications himself.

Those components must not be installed or operated by amateurs. Lab setups are not supposed to control plants or devices, especially, if malfunctions can cause hazards or damages. Never handle high voltages or high currents with lab setups.

Before modifying circuitries and before opening the casing, always disconnect devices respectively breadboard constructions from the power supply. Sensitive components such as CMOS circuitries have to be handled in an antistatic environment, in order to avoid damages.

Typographic conventions

We use the following scripts and symbols which help you quickly recognise important information.

<table>
<tr><td colspan="6">Element</td><td>Meaning</td></tr>
<tr><td colspan="6"><code>Program listing</code></td><td>Tiger-BASIC program listing</td></tr>
<tr><td colspan="6">Instruction / Function</td><td>Tiger-BASIC instruction / function</td></tr>
<tr><td></td><td>B</td><td>W</td><td>L</td><td>S</td><td>F</td><td rowspan="4">Parameter description:
● Data type valid
- Data type not valid
B Data type BYTE
W Data type WORD
L Data type LONG
S Data type STRING
F Data type REAL (floating point)</td></tr>
<tr><td>Parameter 1</td><td>●</td><td>●</td><td>●</td><td>-</td><td>-</td></tr>
<tr><td>Parameter 2</td><td>-</td><td>-</td><td>-</td><td>●</td><td>-</td></tr>
<tr><td>Parameter 3</td><td>-</td><td>-</td><td>-</td><td>-</td><td>●</td></tr>
<tr><td colspan="6">Annotation</td><td>Annotation /accentuation</td></tr>
</table>

Instructions & Functions

empty page

Functions: Commented overview

Overview over the new functions in version 5.2:

Res$ = BFlag_Tab$ (These_Chars$, Is_Char_Flg, Is_NOT_Char_Flg)
Generates flag tables as a string of 256 bytes each

ADR = Bit_Map_Adjust (Bitmap$, Record_Size, ADR, Value)
Corrects data block address according to (in)valid block bitmap

Num = Bit_Map_Cnt (Bitmap$, Value)
Counts “0” bits or “1” bits of a bitmap

N = Bit_Map_RD (BMap$, Blk_Size, ADR) **' Read Bit from Bit-Map**
N = Bit_Map_WR (BMap$, Blk_Size, ADR, Val) **' Read + Write Bit**
' from Bit-Map
Reads one bit from a bitmap / writes bit to bitmap

R$ = Check_Keyword$ (Src$, Key_Word_List$, Sep$, Start_Pos, Tolerance)
Checks strings for keywords - e.g. for command words and arguments

Found_Ptr = Chk_Fnam (Src$, Start_Pos, Stepwidth, FName$, Method)
Checks string for the existence of file names

Concentrate$ (Source_Destin$, Initial_Skip, Take, Next_Skips)
Concentrates data fields which exist as e.g. record strings

ERG$ = Conv_Base64$ (Source$, Methode)
Converts data from/to Base64 format

Count_Patt$ (Src$, Patt$, Patt_ELen, Pos, Cnt_Len)
Checks string for the existence of given byte patterns

CRC (A$, Pos, Len, Start_Factor, Step, CRC)
Creates CRC checksum from string data

A$ = Cut_and_Paste$ (SRC$) **' cut away SRC$ completely**
R = Cut_and_PasteR (SRC$, Offset) **' cut out Real from String**
N = Cut_and_PasteN (SRC$, Offset, Len) **' cut out Num from String**
Additional “Cut_and_Paste” functions for string, numerical and floating point variables

ECC$ = Calc_ECC (Data$, Start_Pos, ECC_Method)
Calculates ECC (error correction code) for a data block

Flag = Correct_ECC (Data$, Start, ECC_stored, ECC_calculated, ECC_Method)
Corrects a data block potentially having a parity error

Pos_Ndx$ = Find_opt_Group$ (Num$, Group_Ndx_List$, Size_of_Group,
Size_of_Nums, Target_Val, Tolenrance_Band, Start_Pos, Target_Tol)
Finds the optimal group of cases which contain different quantities

```
I2CL_Setup ( Port, Clock_Pin, Data_Pin, Speed )
```

Specifies Tiger pins used for the I^2C bus

```
I2CL_Start ( Speed )
```

Generates start condition on the I^2C bus / ISO-7816 channel

```
I2CL_Stop ( Speed )
```

Generates stop condition on the I^2C bus / ISO-7816 channel

```
I2CL_Release ( Dummy )
```

Switches both bus lines to high impedance state (without stop condition)

```
A$ = I2CL_Read$ ( nob )                 ' Read Byte(s) from I2C-Bus
A$ = I2CL_Read$ ( nob, 7816 )           ' Read Byte(s) from ISO-7816 Bus
```

Reads specified number of bytes from I^2C bus / ISO-7816 bus

```
NAK = I2CL_Write ( A$ )                 ' Write A$ to I2C-Bus
NAK = I2CL_Write ( N1, N2 )             ' Write N2 Bytes of N1 to I2C-Bus
NAK = I2CL_Write ( A$, 7816 )           ' Write A$ to ISO-7816
NAK = I2CL_Write ( N1, N2, 7816 )       ' Write N2 Bytes of N1 to ISO-7816
```

Writes specified number of bytes to I^2C bus / ISO-7816 bus

```
Flag = I2CL_Result ( )
```

Reads result code of the latest I2CL function carried out

```
N$ = Insert$ ( Source$, S_Pos, Ins$, I_Pos, I_Len )
```

Inserts string (or a part of it) into a string

```
N = Key_Direct ( Port, Column, Mode, Speed )      ' Read key column
N = Key_Direct ( Port, Column, Mode )             ' Read key column
```

Reads up to 16 keys or switches directly via the data bus

```
A = BLookup# ( Index )
A = WLookup# ( Index )
A = LLookup# ( Index )
```

Fast access to data in “look up” tables in DATA FLASH area or strings

```
A = Pin ( ADR, Bit_Pos )
```

Reads a single pin of a port

```
FLG = PushN ( A$, NUM, 0..4 )           ' Push 0...4 Bytes of NUM to A$
FLG = PushR ( A$, REAL )                ' Push 1 REAL = 8 Bytes to A$
FLG = Push$ ( A$, Stri$, Pos, Len )     ' Push Len Bytes from Stri$
                                        ' starting at POS to A$
N = PopN ( A$, 0..4 )                   ' Pop 0...4 Bytes from A$
R = PopR ( A$ )                         ' Pop 8 Bytes from A$ to Real
X$ = Pop$ ( A$, Len )                   ' Pop a String from A$
```

PUSHes data (byte, word, long, real, string) to string and likewise POPs data from string to byte, word, long, real, string

Graphic_Reformat (Src$, Dest$, Width, High, Re_Format)

Reformats pixel structure of a pixel graphic from horizontal to vertical

Dest$ = Text_Reformat (Source$, Dest_Width, Dest_Cut_Wrap, Dest_Fill_Blank, Dest_Line_End)

Reformats ASCII text strings for output media (terminals, LCDs, printers,...)

New_Pos = Scan_or_Skip (Src$, Charset$, Pos, Scan_Skip)

Scans or skips given character collectives in strings

FLG = SPI_Setup (Clk_MOSI_Port, Clk_Pin, MOSI_Pin, SSI_Port, SSI_Pin, MISO_Port, MISO_Pin, MSB_First)

Specifies SPI bus for function SPI_IO (....)

Rec$ = SPI_IO$ (Transm$)

Sends and receives data simultaneously via SPI bus as specified in SPI_SETUP

Dest$ = Universal_Convert$ (Src$, Search_List$, Replace_List$)
Dest$ = Universal_Convert$ (Src$, Search_List$, Replace_List$, Start)
Dest$ = Universal_Convert$ (Src$, Search_List$, Replace_List$, Start, Len)

Universal string converter with search-string list and replace-string list

Data$ = Unpack_DC$ (Ctrl$, Src$, Pos, Len, Flag, Method)

Unpacks source data stream and divides it into 2 channels: DATA channel and CTRL channel

Err_Code = Update_Me (Data_Label, Option)

Replaces an existing program by a succeeding program version (update) and starts the new program

Flag = XSetup (Bus_Port, Ctrl_Port, Bit_ACLK, Bit_DLCK, Bit_INE, Ctrl2_Port, Bit_Bus_CE)

Defines bus and control signal lines for the XPort system

N = Xin (ADR) **' <ADR> <data>**

Reads 1 byte from the I/O extension port (XPort) “ADR”

X$ = Xin$ (ADR, Anz) **' <ADR> <data> <ADR+1> <data>...**
X$ = Xin$ (-ADR, Anz) **' <ADR> <data> <data> <data> ...**
X$ = Xin$ (256+, Anz) **' <data> <data> <data> ...**

Reads extension port with different types of bus access

Xout (ADR, N) **' <ADR> <data> - write 1 Byte**

Writes 1 byte to I/O extension port (XPort) “ADR”

```
Xout ( ADR, A$ )                 ' write many Bytes to subsequ. ADRs
Xout ( ADR, ““ )                 ' write ONLY 1 ADR cycle, NO data
Xout ( -ADR, A$ )                ' write many Bytes, 1 ADR cycle only
Xout ( 256+, A$ )                ' write many Bytes, NO ADR cycle
Xout ( ADR, Flash, FLen )        ' write many Bytes to subsequ. ADRs
Xout ( ADR, Flash, 0 )           ' write ONLY 1 ADR cycle, NO data
Xout ( -ADR, Flash, FLen )       ' write many Bytes, 1 ADR cycle only
Xout ( 256+, Flash, FLen )       ' write many Bytes, NO ADR cycle
```

Writes to I/O extension ports (XPorts) in different modes

```
XSet ( ADR, Bit_Pos )            ' set 1 Bit in XPort ADR: 00 ... FF
```

Sets one bit of an XPort output to high = 1

```
XRes ( ADR, Bit_Pos )            ' reset 1 Bit in XPort ADR: 00 ... FF
```

Sets one bit of an XPort output to low = 0

```
XInv ( ADR, Bit_Pos )            ' toggle 1 Bit in XPort ADR: 00 ... FF
```

Inverts one bit of an XPort output

```
A = XPin ( ADR, Bit_Pos )        ' read 1 Bit in XPort ADR: 00 ... FF
```

Reads a single pin of an XPort input

BFlag_Tab$ (i)

Res$ = BFlag_Tab$ (These_Chars$, Is_Char_Flg, Is_NOT_Char_Flg)

Function: BFlag_Tab$ generates flag tables as a string of 256 bytes each. Every byte of the flag string represents a character code according to its position in the flag string.

BFlag_Tab$ works in 3 different ways. The first one (i) is:

Every byte in the string can contain one of two available flag values, “Is_Char_Flg” or “Is_NOT_Char_Flg”, the position of which in the string is defined by the character code of parameter “These_Chars$”.

Example:

Hex table of the flag string Res$ being generated

```
Res$ = BFLAG_TAB$ ( "ABCDEFgh", 0FFH, 77H)

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 00 ... 0F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 10 ... 1F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 20 ... 2F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 30 ... 3F

77 FF FF FF FF FF FF 77 77 77 77 77 77 77 77 77        ' 40 ... 4F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 50 ... 5F
77 77 77 77 77 77 77 FF FF 77 77 77 77 77 77 77        ' 60 ... 6F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 70 ... 7F

77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 80 ... 8F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' 90 ... 9F
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' A0 ... AF
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' B0 ... BF

77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' C0 ... CF
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' D0 ... DF
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' E0 ... EF
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77        ' F0 ... FF
```

“FF” = Is_Char_Flg = flag for codes which belong to the character set.
“77” = Is_NOT_Char_Flg = flag value for codes which do NOT belong to the character set.

A flag string is always needed, if fast conversions, filters, masking, searching, scanning, skipping and similar procedures are involved. An example will be presented in the context of the Scan_or_Skip function.

BFlag_Tab$

Generate Byte Flag Table

Parameters:

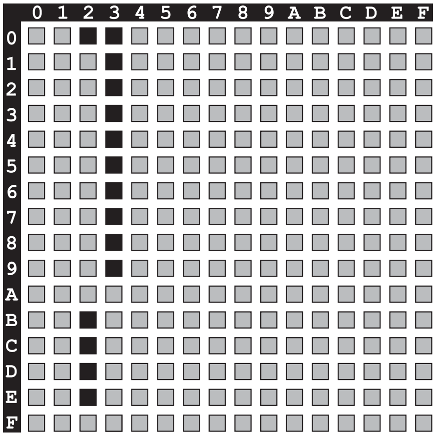
	B	W	L	S	F	
These_Chars$	-	-	-	●	-	String with all characters which are to belong to the character set
Is_Char_Flg	●	●	●	-	-	Flag value for identifying codes in “Res$”, which belong to the character set according to the definition in “These_Chars$” (i) or 256 (ii) or 256 + offset (iii)
Is_NOT_Char_Flg	●	●	●	-	-	Flag value for identifying codes in “Res$”, which do NOT belong to the character set according to the definition in “These_Chars$”
						Function value:
Res$	-	-	-	●	-	Flag string, exactly 256 Byte long (if sufficiently declared). Every string’s byte is a flag having one of the following two values: “Is_Char_Flg” or “Is_NOT_Char_Flg”.

2 examples:

```
ChSet$= "01234567890 ,.-+"
R$ = BFlag_Tab$ (ChSet$,7BH,00H)
```

```
ChSet$= "hello lady"
R$ = BFlag_Tab$ (ChSet$,0F1H,0FFH)
```

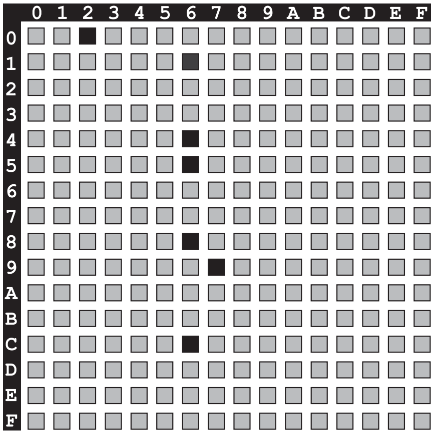
Flag configuration in R$ after executing BFlag_Tab$ functions



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	■	■	□	□	□	□	□	□	□	□	□	□	□	□
1	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
2	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
3	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
4	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
5	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
6	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
7	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
8	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
9	□	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□
A	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
B	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□
C	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□
D	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□
E	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□
F	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

□ = 00 ■ = 7B

R$



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	■	□	□	□	□	□	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
7	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
8	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□
9	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
A	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
B	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
C	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□
D	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
E	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
F	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

□ = FF ■ = F1

R$

BFlag_Tab$ (ii)

Res$ = BFlag_Tab$ (These_Chars$, 256, Is_NOT_Char_Flg)

Function: Here (ii) BFlag_Tab$ generates a flag string where each flag byte, marking a character of the character set, is generated with its own code. The characters which do NOT belong to the character set, are generated by the code given by “Is_NOT_Char_Flg” as presented in (i)

Example:

Hex table of the flag string Res$ being generated

```
Res$ = BFLAG_TAB$ ( "ABCDEFgh", 256, 2EH)

   0123456789ABCDEF

  "................"        ' 00 ... 0F
+ "................"        ' 10 ... 1F
+ "................"        ' 20 ... 2F
+ "................"        ' 30 ... 3F

+ ".ABCDEF........."        ' 40 ... 4F
+ "................"        ' 50 ... 5F
+ ".......gh......."        ' 60 ... 6F
+ "................"        ' 70 ... 7F

+ "................"        ' 80 ... 8F
+ "................"        ' 90 ... 9F
+ "................"        ' A0 ... AF
+ "................"        ' B0 ... BF

+ "................"        ' C0 ... CF
+ "................"        ' D0 ... DF
+ "................"        ' E0 ... EF
+ "................"        ' F0 ... FF
```

‹Code› = Flag value for codes which belong to the character set.

"." = ‹2EH› = Is_NOT_Char_Flg = Flag for codes which do NOT belong to the character set.

BFlag_Tab$ (iii)

Res$ = BFlag_Tab$ (These_Chars$, 256 + Offset, Is_NOT_Char_Flg)

Function: Here (iii) BFlag_Tab$ generates a flag string where each flag byte, marking a character of the character set, is generated with its own code + the value given in “Offset”. The characters which do NOT belong to the character set, are generated by the code given by “Is_NOT_Char_Flg” as presented in (i) and (ii).

Example:

Hex table of the flag string Res$ being generated

```
Res$ = BFLAG_TAB$ ( "ABCDEFgh", 256 + 3, 2DH)

    0123456789ABCDEF

    "----------------"       ' 00 ... 0F
  + "----------------"       ' 10 ... 1F
  + "----------------"       ' 20 ... 2F
  + "----------------"       ' 30 ... 3F

  + "-DEFGHI---------"       ' 40 ... 4F
  + "----------------"       ' 50 ... 5F
  + "-------jk-------"       ' 60 ... 6F
  + "----------------"       ' 70 ... 7F

  + "----------------"       ' 80 ... 8F
  + "----------------"       ' 90 ... 9F
  + "----------------"       ' A0 ... AF
  + "----------------"       ' B0 ... BF

  + "----------------"       ' C0 ... CF
  + "----------------"       ' D0 ... DF
  + "----------------"       ' E0 ... EF
  + "----------------"       ' F0 ... FF
```

‹Code› = Flag value for codes which belong to the character set.

"." = ‹2EH› = Is_NOT_Char_Flg = Flag for codes which do NOT belong to the character set.

Generate Byte Flag Table

BFlag_Tab$ can be used e.g. to detect characters which can be found in a text and to mark them by flags in a flag string.

Example:

```
Flag_Total$ = Fill$ ("00"%, 256)          ' initialize flag string
FOR EVER=0 TO 0 STEP 0                    ' <-- endless loop -->
  ...                                     ' read some text into "Text$"
  Flag$ = BFLAG_TAB$(Text$, 0FFH, 0)      '
  Flag_Total$ = OR$ (Flag$, Flag_Total$)  '
  ...                                     ' Loop-Exit Condition
NEXT                                      '
```

... which could turn out to look like the following flag string:

```
"...Just some text and 0123456789,.-+ ..."
```

Examplary flag configuration in Flag_Total$ after stopping the infinite loop

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	■	■	□	□	□	■	□	□	□	□	□	□	□	□
1	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
2	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
3	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
4	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
5	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
6	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
7	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
8	□	□	□	■	□	□	■	■	□	□	□	□	□	□	□	□
9	□	□	□	■	□	□	■	□	□	□	□	□	□	□	□	□
A	■	□	□	□	□	□	■	■	□	□	□	□	□	□	□	□
B	□	□	■	■	□	□	■	□	□	□	□	□	□	□	□	□
C	□	□	■	□	□	□	■	□	□	□	□	□	□	□	□	□
D	■	□	■	□	□	□	■	□	□	□	□	□	□	□	□	□
E	□	□	■	□	□	□	■	□	□	□	□	□	□	□	□	□
F	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□

□ = 00 ■ = FF

This table states that the text only contains small letters (no “y”), some punctuation marks as well as ‹CR› and ‹LF› codes.

Bit_Map_Adjust

ADDR = Bit_Map_Adjust (Bitmap$, Record_Size, ADDR, Value)

Function:	Corrects a data block address according to an invalid block respectively valid block. Application: Administration of memory blocks, FAT systems, RAM discs, flash discs, ...

Parameters:

	B	W	L	S	F	
Bitmap$	-	-	-	●	-	Bitmap; 1 bit each is a flag for a memory block of the size given in the parameter “Record_Size”. Valid respectively invalid memory blocks are marked in the bitmap (see “Value”).
Record_Size	●	●	●	-	-	Block size: 1 ... 7FFFH bytes
ADDR	●	●	●	-	-	Memory address: 0 ... 7FFF FFFFH
Value	●	●	●	-	-	0: “0” bit marks “valid block” 1: “1” bit marks “invalid block”
						Function value:
ADDR	●	●	●	-	-	corrected address to the next valid block address, where applicable:

0 ... nnnnnnnn	ADDR with or without correction
-1	Error: bitmap$ = empty
-2	Error: record size invalid
-3	Error: ADDR lies behind end of ADDR (possibly after correction)

Bit_Map_Adjust

Adjust ADDR According to Bitmap

Example:

```
ADDR = 8000H
Bitmap$ = "00 03 00 00 80 00 00 00 00 00 00 00 00 00 00 00"%
ADDR = Bit_Map_Adjust (Bitmap$, 1000H, ADDR, 1)
```

Bitmap$ shows 3 “invalid blocks”:

	7	6	5	4	3	2	1	0
0	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	■	■
2	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
4	■	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
7	□	□	□	□	□	□	□	□
8	□	□	□	□	□	□	□	□
9	□	□	□	□	□	□	□	□
A	□	□	□	□	□	□	□	□
B	□	□	□	□	□	□	□	□
C	□	□	□	□	□	□	□	□
D	□	□	□	□	□	□	□	□
E	□	□	□	□	□	□	□	□
F	□	□	□	□	□	□	□	□

1. invalid block = 8000H
2. invalid block = 9000H
3. invalid block = 2F000H

The input value of ADDR = 8000H lies in an “invalid block” according to the bitmap. The address ADDR is moved to the next valid block (A000H) by Bit_Map_Adjust.

Bit_Map_Adjust is used to administer memory blocks in mass storages. In a bitmap usually invalid/valid blocks or free / assigned blocks are administered.

An application of this function can be found in the FAT file system applications with SmartMedia storage media.

Bit_Map_Cnt

Num = Bit_Map_Cnt (Bitmap$, Value)

Function: Counts ”0” bits or “1” bits of a bitmap

Application: Administration of memory blocks, FAT systems, RAM discs, FLASH discs,...

Parameters:

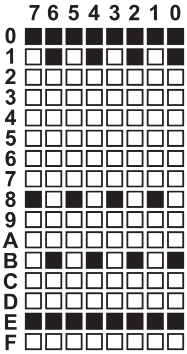
	B	W	L	S	F	
Bitmap$	-	-	-	●	-	Bitmap, 1 bit each is a flag for a memory block of the size given in parameter “Record_Size”. Valid respectively invalid memory blocks are marked in the bitmap (see “Value”).
Value	●	●	●	-	-	0: count “0” bits X: count “1” bits
						Function value:
Num	●	●	●	-	-	0 ... nnnnnnnn Number of counted bits -1 Error: Bitmap$ is empty

Example:

```
Bitmap$ = "FF 55 00 00 00 00 00 00 AA 00 00 55 00 00 FF 00"%
Num = Bit_Map_Cnt (Bitmap$, 1)
```

After executing the sequence stated above the value 28 is assigned to Num.

This complies with the number of “1” bits in Bitmap$.



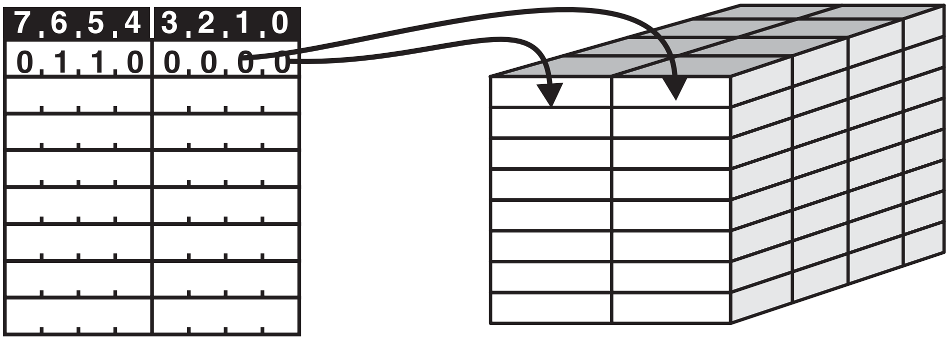
Bit_Map_RD
Bit_Map_WR

```
N = Bit_Map_RD (BMap$, Blk_Size, ADDR)       ' read Bit from Bit-Map
N = Bit_Map_WR (BMap$, Blk_Size, ADDR, Val)  ' read + write Bit from Bit-Map
```

Read bits from and write bits to bitmap

Function: Read bit from the bitmap / write bit to bitmap

Bitmaps are suited for e.g. administrating memory media, such as discs, and RAM or FLASH mass storages. In the mass storage one block, sector or cluster, which can be assigned or free, valid or invalid, is equivalent to 1bit each.



Bitmap
1 bit

=>

Mass storage organisation
1 block, sector, cluster, ...

Bit_Map_RD
Bit_Map_WR

Read + Write Bitmap

Parameters:

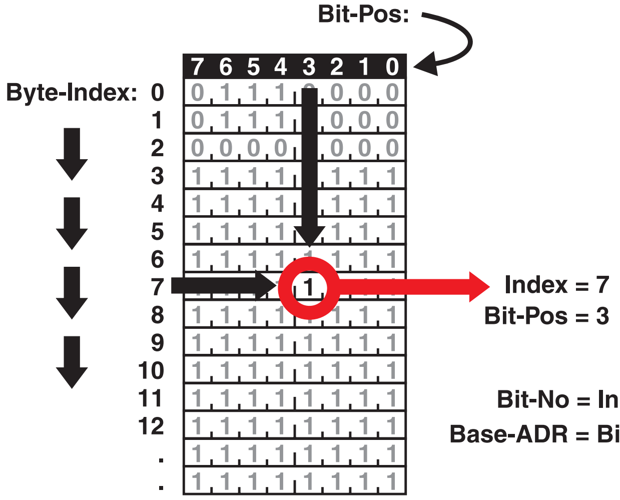
	B	W	L	S	F	
BMap$	-	-	-	●	-	Bitmap string, input and output. 1 Bit complies with 1 memory block, whose size is defined by “Blk_Size”.
Blk_Size	●	●	●	-	-	Size of a memory block (sectors, clusters ...) in bytes, values: 1 ... 7FFFH
ADDR	●	●	●	-	-	Address in mass storage: 0 ... nnnn nnnn
Val	●	●	●	-	-	Bit value, which is to be entered into the bitmap: 0 = write “0” bit X = write “1” bit
						Function value:
N	●	●	●	-	-	Former value of the according bit from the bitmap: 0 = Bit is/was =0 1 = Bit is/was =1 -1 = Error: BMap$ = empty -2 = Error: Block size invalid -3 = Error: ADDR not contained in this bitmap

The function value is always available, both for reading (Bit_Map_RD) and for writing (Bit_Map_WR). When writing, N states the bit value, which existed at this position before the writing procedure.

Bit_Map_RD
Bit_Map_WR

Read + Write Bitmap

The correlation between bit position in the bitmap (Bit-No), block size (Block-Size) and address (Base-ADR):



Bit-No = Index * 8 + Bit-Pos
Base-ADR = Bit-No * Block-Size

Bitmap

Example:

Block-Size = 16 kByte = 4000H

Byte-Index:	Bit-Pos:	Bit-No:	Block-No:	Block Start-ADDR:
0	0	0	0	0
0	1	1	1	4000H
0	2	2	2	8000H
0	3	3	3	C000H
0	4	4	4	10000H
.	.	.	.	.
.	.	.	.	.
1	1	9	9	24000H
1	2	0AH	0AH	28000H
1	3	0BH	0BH	2C000H
1	4	0CH	0CH	30000H
1	5	0DH	0DH	34000H
.	.	.	.	.
.	.	.	.	.
33H	2	198H	198H	660000H
34H	3	199H	199H	664000H
35H	4	19AH	19AH	668000H
.	.	.	.	.
.	.	.	.	.

Bit_Map_RD
Bit_Map_WR

Read + Write Bitmap

Discs and other mass storage systems use bitmap tables to mark allocated and free blocks. Instead of carrying out markings in the according memory block, only the respective bit is set accordingly in a bitmap (e.g. record allocation table).

Bit_Map_RD and Bit_Map_WR simplify and accelerate this access. Examples for using this function can be found, amongst others, in the application: FAT systems for SmartMedia FLASH cards.

Two short examples concerning the functions' effectiveness:

Example 1: Read bit from bitmap according Block_Size and ADDR:

```
BMap$ = "00 09 00 00"%                 ' Bitmap = 4 Bytes = 32 Bits
N1 = Bit_Map_RD (BMap$, 4000H, 20642H) ' read Bit from Bitmap
N2 = Bit_Map_RD (BMap$, 4000H, 28033H) ' read Bit from Bitmap

' Block_Size in Storage System = 4000H
'
' N1 = 1   (Bit-No in Bitmap =  8,  Bit-Value = "1")
' N2 = 0   (Bit-No in Bitmap = 0AH, Bit-Value = "0")
```

Example 2: Write bit to bitmap and read the bit's value before the writing procedure:

```
BMap$ = "00 09 00 00"%                    ' Bitmap = 4 Bytes = 32 Bits
N3 = Bit_Map_WR (BMap$, 4000H, 20642H,0) ' read + write Bit from/to Bitmap

' Block_Size in Storage System = 4000H
'
' N3 = 1   (Bit-No in Bitmap =  8,  Bit-Wert = "1")
'
' BMap$ = "00 08 00 00"%                 ' Bitmap = 4 Bytes = 32 Bits
'                                        ' 1 bit resetted to "0"
```

Check_Keyword$

R$ = Check_Keyword$ (Src$, Sep$, Key_Word_List$, Start_Pos, Tolerance)

Function: Checks strings for keywords - e.g. for commands and arguments.

Application: Text analyses, command interpreter, remote control via SMS, email, security applications, ...

Parameters:

	B	W	L	S	F	
Src$	-	-	-	●	-	Input string, which a valid keyword is to be searched in.
Sep$	-	-	-	●	-	Separator flag string: String with flags, exactly 256 bytes long 00 = this code is NOT a separator in the source string XX = this code is a separator in the source string This flag string marks those character codes in Src$, which are used as separators between keywords. When searching for keywords, these characters are missed out. They are never part of a keyword.
Key_Word_List$	-	-	-	●	-	String with a list of keywords. Format: "‹sep›Keyword_1‹sep›Keyword_2‹sep›..." e.g.: ".switch.do.make.set.on.off.to.from."
Start_Pos	●	●	●	-	-	From this start position Src$ is checked for keywords: 0 ... nnnn.

Tolerance	●	●	●	-	-	Marks the tolerance when detecting a keyword from the Key_Word_List:

Tolerance:	0	1	2	3	4	5	6	7	8	9
Exact match required:	●	○	○	○	○	○	○	○	○	○
1 matching character required:	○	●	○	○	○	○	○	○	○	○
2 matching characters required:	○	○	●	○	○	○	○	○	○	○
3 matching characters required:	○	○	○	●	○	○	○	○	○	○
4 matching characters required:	○	○	○	○	●	○	○	○	○	○
5 matching characters required:	○	○	○	○	○	●	○	○	○	○
6 matching characters required:	○	○	○	○	○	○	●	○	○	○
7 matching characters required:	○	○	○	○	○	○	○	●	○	○
8 matching characters required:	○	○	○	○	○	○	○	○	●	○
9 matching characters required:	○	○	○	○	○	○	○	○	○	●

Tolerance:	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
Accept no errors:	●	○	○	○	○	○	○	○	○	○
1 faulty character = OK:	○	●	○	○	○	○	○	○	○	○
2 faulty characters = OK:	○	○	○	○	●	○	○	○	○	○
3 faulty characters = OK:	○	○	○	○	○	○	○	●	○	○
1 character too much = OK:	○	○	●	○	○	○	○	○	○	○
2 characters too much = OK:	○	○	○	○	○	●	○	○	○	○
3 characters too much = OK:	○	○	○	○	○	○	○	○	●	○
1 character too little = OK:	○	○	○	●	○	○	○	○	○	○
2 characters too little = OK:	○	○	○	○	○	○	●	○	○	○
3 characters too little = OK:	○	○	○	○	○	○	○	○	○	●

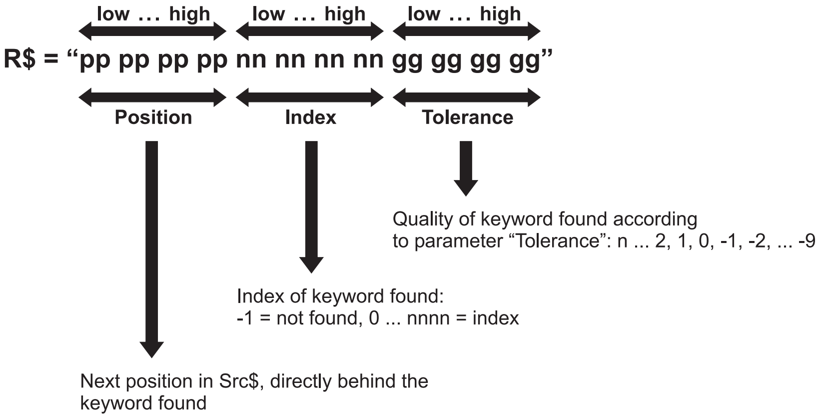
Function value:

R$	-	-	-	●	-	Result string with information about the possibly found next keyword.

Length = 12 bytes = 3 LONG numerical values:

1. LONG = Position in Src$
2. LONG = Index of the keyword found
3. LONG = Tolerance of the keyword found

Result string structure:



Quality of keyword found according to parameter "Tolerance": n ... 2, 1, 0, -1, -2, ... -9

Index of keyword found:
-1 = not found, 0 ... nnnn = index

Next position in Src$, directly behind the keyword found

Der Source-String: Src$

Src$ usually contains:

Keywords
Separators
Other characters

There is at least 1 separator required for separating between keywords.

Keywords can consist of 1...32 characters and must not contain separators.

The source string can have an arbitrary length.

Separator flag string

Sep$ defines those codes which are supposed to be used as separators between keywords in Src$. For practical purposes all characters which are not used are also defined as separators.

Sep$ is a flag string which always consists of exactly 256 characters. Every byte represents one flag for the according code: 00=this code is NOT a separator; every other flag value marks a separator.

If the separator flag string is empty (""), or only contains ‹0›-Bytes, no keyword will be found as the complete source string will be viewed as one large keyword.

Key_Word_List$

The keyword list contains all used keywords, in different scripts and languages, if necessary. The keywords on the list can be of different lengths and are separated by a separator. The first character on the keyword list is used as a separator. This character only serves for separating the keywords on the list, it has no other purpose.

Example - find exact match

A command string (source string) is to be checked for the next relevant keyword. There is a keyword list with all relevant keywords. The start position for the source string check is = 7. Keywords are supposed to occur as an exact match in the source string in order to be detected.

```
     Src$ = "please print on display and then wait 60 minutes"
  Key_WL$ = ".show.print.lptr.com.line.display.secondary.wait.delay."
Start_Pos = 7                   ' start at position 7
Tolerance = 0                   ' find exact match
R$ = Check_Keyword$ (Src$, Sep$, Key_WL$, Start_Pos, Tolerance)

   ' => R$ "0C 00 00 00 01 00 00 00 00 00 00 00"
   '        <=========> <=========> <=========>
   '          Next-Pos     Index      Tolerance
   '
   ' Next Position = 12   (in sorce-string)
   '         Index = 1    (second item in Key_WL$)
   '     Tolerance = 0    (exact match)
```

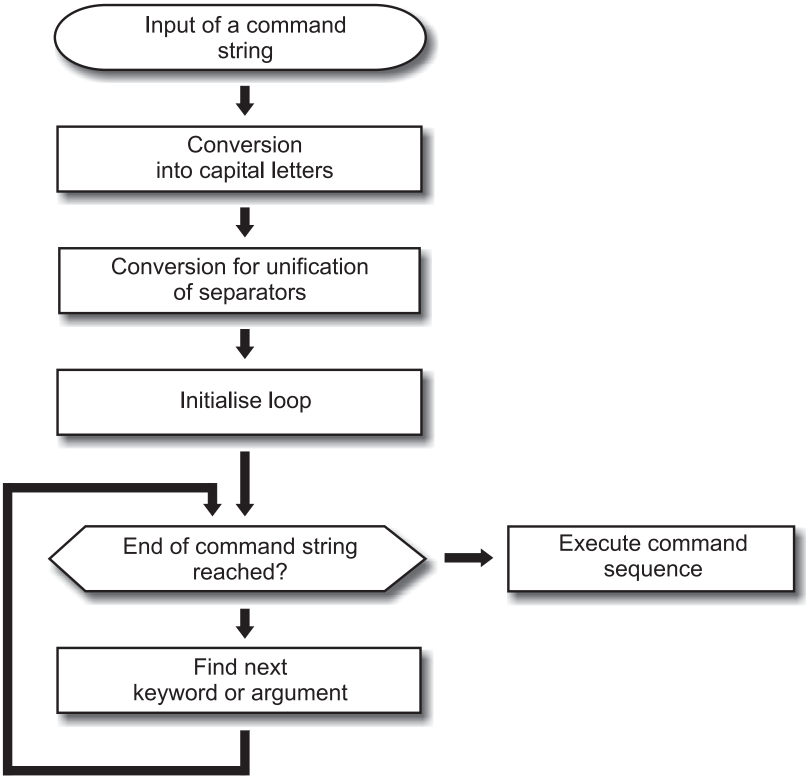
Here the keyword “print”, which is at position 2 on the list (Index=1), was detected. For further analysing the source string one can use the output of “next-Pos”, which points directly behind the found keyword “print”.

A start position before 7 would lead to a result of index = -1 (not found), as comparing starts at the start position, and no position before 7 will give a valid keyword: “pl...”, “le...”, “ea...”, “as...”, “se ...”, “e ...” are NOT in the keyword list.

Example 2

Tolerant command input, e.g. for remote controlling via SMS or email. It is the objective to create a simple human-machine interface which uses easily understandable plaintext commands to activate specific functions.

Program structure:



First of all every command/source string is brought to a standard form by conversion: Capital letters and uniform separators - including equal treatment of decimal point and decimal comma.

Then strings are checked for relevant keywords and numerical arguments.

In order to make operation most simple and tolerant, it is important to implement a flexible and tolerant usage of keywords. This applies to multiple aspects:

(a) It will be allowed to use different terms for the same functions,
(b) It will be allowed to abbreviate long terms,
(c) It will be allowed to make minor spelling mistakes without affecting the function.

The function „Check_Keyword$“ fulfils all these requirements:

Using different terms for the same circumstance (a)

For example:

valid keywords:	**Meaning:**
do switch tune set turn make execute	command for a **switching operation**
on bright open hot up high run	**ON**
off dark close cold down low stop	**OFF**

The implementation in BASIC:

```
.
.
Key_WL$ = "&
-do-switch-tune-set-turn-make-execute&                         ' Ndx =  0... 6
-on-bright-open-hot-up-high-run&                               ' Ndx =  7.. 13
-off-dark-close-cold-down-low-stop--"                          ' Ndx = 14...20

R$ = Check_Keyword$ (Src$, Sep$, Key_WL$, 0, 0)
Index = NFROMS (R$, 4,4)              ' get Index of Keyword
Command_NDX = NFROMS ("00 00 00 00 00 00 00&
 01 01 01 01 01 01 01 02 02 02 02 02 02 02"%, Index, 1)
```

If one of the keywords stated above occurs in the source string Src$, the following values for “Index” and “Command_NDX” are created:

“do”	→	Index = 0	Command_NDX = 0
“switch”	→	Index = 1	
“tune”	→	Index = 2	
“set”	→	Index = 3	
“turn”	→	Index = 4	
“make”	→	Index = 5	
“execute”	→	Index = 6	
“on”	→	Index = 7	Command_NDX = 1
“bright”	→	Index = 8	
“open”	→	Index = 9	
“hot”	→	Index = 10	
“up”	→	Index = 11	
“high”	→	Index = 12	
“run”	→	Index = 13	
“off”	→	Index = 14	Command_NDX = 2
“dark”	→	Index = 15	
“close”	→	Index = 16	
“cold”	→	Index = 17	
“down”	→	Index = 18	
“low”	→	Index = 19	
“stop”	→	Index = 20	

Due to the keyword table's structure the command task

"switch on"

can be given with several command string versions.
Some examples:

"please ***switch on*** the light now"
"***turn*** the light ***on***"
"***switch on***"
"***set*** heating ***high***"
"***tune*** heating ***up***"
"***switch*** to ***hot***"
"***turn*** the heating ***on***"
"***make*** it ***bright*** here"
etc...

You will notice that it is possible to implement a diversified command input into a human-machine interface by choosing alternative words for the same function.

Allow abbreviations (b)

The variety of command versions can be further increased by including a fault-tolerance for keyword identifying:

Tolerance = +1... +nn, it is allowed to ➡ **abbreviate terms**

In case of a tolerance of +1 ... +nn a keyword will be detected

(i) if it exists completely correct in the source string or
(ii) if the first 1 ... nn characters match

In the example given above with a tolerance of +3 also those terms would be detected correctly, which would not be detected with a tolerance of 0 (=exact match):

"please ***switch on*** the light now"
"please ***switc on*** the light now"
"please ***swit on*** the light now"
"please ***swi on*** the light now"
„***switch*** the light ***on***"
„***swit*** light ***on***"

For instance, the following commands would not work:

"please ***sw on*** the light now" ---> to short for tolerance = 3
"***switching on*** the light" ---> to long, regarded as different keyword
"***switches*** the light ***on***" ---> to long, regarded as different keyword
"***swing on***" ---> first 3 chars match, but regarded as different keyword

Allowing abbreviations has the following advantages:

- The number of valid variations increases without having to extend the keyword list. This creates a much more reliable interface even for occasional users.
- For frequent use long commands can be avoided by using convenient abbreviations.
- Records with commands become more compact, transmissions become shorter.

Allow spelling mistakes (c)

The variety of command variants can be increased even more by also allowing spelling mistakes to a certain extend using the tolerance parameter.

Incorrect spellings can have various reasons and it is annoying, if commands are not executed because of typos or uncertainty about the correct spelling.

Fault tolerance is usually desired, if

- the application has nothing to do with processes concerning security
- NOT executing commands entails disadvantages
- there is no feedback channel. You possibly do not realise that the command is not executed because of the incorrect spelling
- access is given to a number of people and spelling mistakes are therefore inevitable

Check_Keyword$ differentiate 3 kinds of spelling mistakes:

1.) Incorrect characters
2.) Redundant characters
3.) Missing characters

Up to 3 of those mistakes can be accepted in one keyword.

The valid tolerance parameters are:

Tolerance:	**0**	**-1**	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**	**-8**	**-9**
Accept no errors:	●	○	○	○	○	○	○	○	○	○
1 faulty character = OK:	○	●	○	○	○	○	○	○	○	○
2 faulty characters = OK:	○	○	○	○	●	○	○	○	○	○
3 faulty characters = OK:	○	○	○	○	○	○	○	●	○	○
1 character too much = OK:	○	○	●	○	○	○	○	○	○	○
2 characters too much = OK:	○	○	○	○	○	●	○	○	○	○
3 characters too much = OK:	○	○	○	○	○	○	○	○	●	○
1 character too little = OK:	○	○	○	●	○	○	○	○	○	○
2 characters too little = OK:	○	○	○	○	○	○	●	○	○	○
3 characters too little = OK:	○	○	○	○	○	○	○	○	○	●

Examples

1.)	Correct form	heating
	1 incorrect character	heeting
	2 incorrect characters	heering
	3 incorrect characters	heerong

2.)	Correct form	heating
	1 redundant character	heeating
	2 redundant characters	heeatting
	3 redundant characters	heeattingg
	3 redundant characters	heaattiing

3.)	Correct form	heating	
	1 missing character	heting	
	1 missing character	heatig	
	1 missing character	heatin	
	3 missing characters	hetg	
	3 missing characters	eati	
	3 missing characters	etng	
	3 missing characters	heat	(= abbreviation)

When using the tolerance parameters you get a result string containing the information, which tolerance the keyword found in the source text has. I.e. even if you allow a tolerance concerning the spelling, the actual spelling in the source string can be perfectly correct or less incorrect than allowed by the tolerance:

```
R$ = Check_Keyword$ (Src$, Sep$, Key_WL$, Start, -4)  ' allow 2 mistakes
```

If a valid keyword is found here, the tolerance can take 3 possible values in R$:

Tolerance = 0 ➡ keyword spelled correctly
Tolerance = -1 ➡ keyword with 1 incorrect character
Tolerance = -4 ➡ keyword with 2 incorrect characters

Spelling mistakes and abbreviations

In order to give the command interface maximum flexibility, you can check command strings repeatedly with different tolerances and analyse each result.

For instance it could make sense to allow:

1.) Abbreviations of minimum 3 characters
2.) 1 missing character = OK
3.) 1 redundant character = OK
4.) 1 incorrect character = OK

Under those conditions the keyword “switch“ is detected being spelled as follows:

switchs
switch
switc
swit
swi
switsh, swidch, swetch, svitch, ...
swtch, sitch, swith, swich, ...
switsch, swiitch, swittch, swwitch, ...

By using different keywords for one circumstance as well as by using different tolerance parameters you can create most flexible command interfaces by comparatively simple means.

Check_Keyword$ makes for short runtimes and simplifies programming.

Note:

When creating the keyword list please consider that searching for a matching keyword always begins at the top of the list. Basically the order of keywords can be random; however, it has the following effects:

- The order of keywords defines their according index: 0, 1, 2, 3...
- Keywords, which occur often, should be placed at the top of the list, in order to avoid wasting runtime unnecessarily by checking the bottom of the list
- What comes first should be found first:
  Therefore “water” should be positioned BEHIND “water fall” on the list.

Chk_Fnam

Found_Ptr = Chk_Fnam (Src$, Start_Pos, Stepwidth, FName$, Method)

Function: Checks a string for filenames - exact match or with wildcards.

Application: Search in file systems

Parameters:

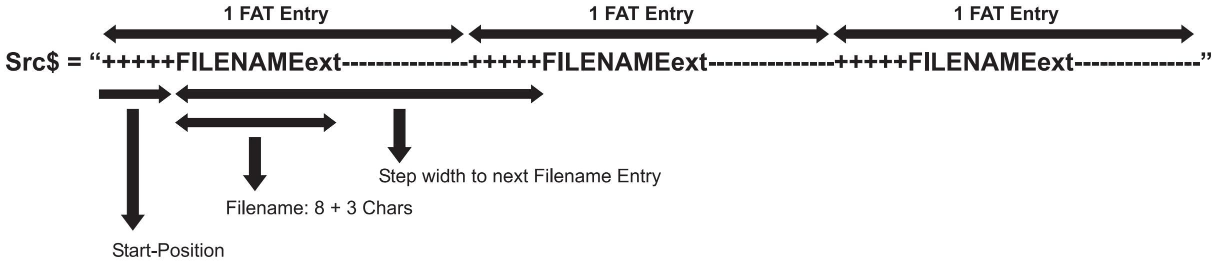
	B	W	L	S	F	
Src$	-	-	-	●	-	Source string with FAT and filenames
Start_Pos	●	●	●	-	-	Start position for searching the filename in Src$: 0 ... nnnn
Stepwidth	●	●	●	-	-	Step width of getting to the next filename entry in the byte: 0 ... nnnn
FName$	-	-	-	●	-	Filename for searching: "FILENAME" + "EXT"
Method	●	●	●	-	-	Filename type and searching method: 0: Filename + Ext like in DOS: Filename: 8 ASCII characters: "A"..."Z", "0...9", "-", possibly filled up with blanks Ext: 3 ASCII characters: "A"..."Z", 0...9", "-", possibly filled up with blanks ONLY capital letters as well as wildcards: "*" = 0...8 arbitrary characters in the filename "*" = 0...3 arbitrary characters in the EXT "?" = exactly 1 arbitrary character

Function value:

	B	W	L	S	F	
Found_Ptr	●	●	●	-	-	Pointer to the beginning of the filename found in Src$: 0 ... nnnn If not found: = -1

Chk_Fnam$ Search Filename

Typical source string structure:



In the source string the filename is formatted as follows:

8 + 3 ASCII- characters if necessary filled up with blanks e.g.:

```
.....FILENAMEEXT......
.....FILENAMEEX ......
.....FILENAMEE  ......
.....FILENAME   ......
.....FILENAM EXT......
.....FILENA  EXT......
.....FILEN   EXT......
.....FILE    EX ......
.....FIL     E  ......
.....FI         ......
.....F          ......
```

This is the notation for the next searched filename (Fnam$):

filename ‹dot› extension

(extension and dot can be left out, if applicable)

E.g.:
"C-001.DAT"
"GO.EXE"
"S-001.MP3"
"A"
"A."
"ABC.D"

Chk_Fnam$

Search Filename

Some examples for Fnam$ values and according filename entries in Src$:

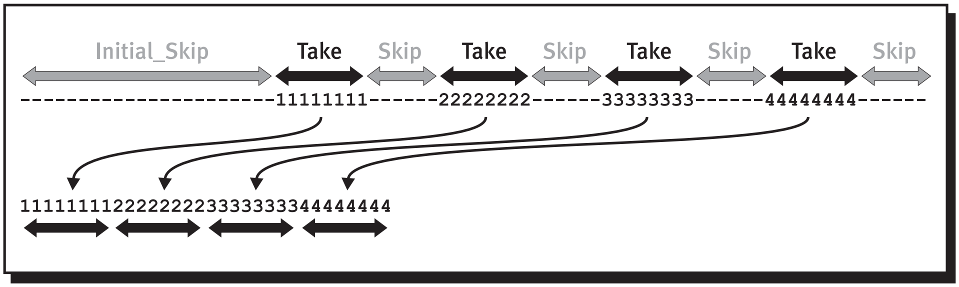
```
 Fnam$           <--Src$--->
                 12345678...
"*.TXT"          HALL    TXT   -->  yes found
"HA*.TXT"        HALL    TXT   -->  yes found
"HA*.TX*"        HALL    TXT   -->  yes found
"HALL*.TXT"      HALL    TXT   -->  yes found

"HA?*.T??"       HALL    TXT   -->  yes found
"HA??*.TXT"      HALL    TXT   -->  yes found
"HA???*.TXT"     HALL    TXT   -->  NOT found
"HA????.TXT"     HALL    TXT   -->  NOT found


"H*.*"           HALL    TXT   -->  yes found
"*.TXT"          HALL    TXT   -->  yes found
"*.*"            HALL    TXT   -->  yes found
"?A*.T*"         HALL    TXT   -->  yes found
```

Concentrate$

Concentrate$ (Source_Destin$, Initial_Skip, Take, Next_Skips)

Function: Concentrate$ concentrates data arrays which e.g. exist as records in strings.

Extracts and concentrates arrays from a string



Parameters:

	B	W	L	S	F	
Source_Destin$	-	-	-	●	-	Input and output string which is to be concentrated.
Initial_Skip	●	●	●	-	-	Skip this amount of bytes initially,
Take	●	●	●	-	-	then take this amount of bytes
Next_Skips	●	●	●	-	-	and repeat skipping this amount of bytes until bytes are taken again.

This sequence is executed from the string's beginning to the string's end.

No function value

Example

Section from a customer data base in ASCII format

```
S_D$ = "&                                                 ' Section
Marks     Bob    Boston      12345Rodeo Drive    00023813&' from a
Michelson Sven   Baltimore      77Huntington St. 00015222&' customer file
Miller    Elena  San Antonio   21114-th Street   00007817&
Modrow    Joan   New York     3231Broadway       00118265"
'234567890-234567-23456789012-2345-23456789012345-2345678  <-- Field-Length
'23456789.123456789.123456789.123456789.123456789.1234567  <-- Count

Concentrate$ (S_D$, 35, 15, 43)
```

S_D$ after activating the function:

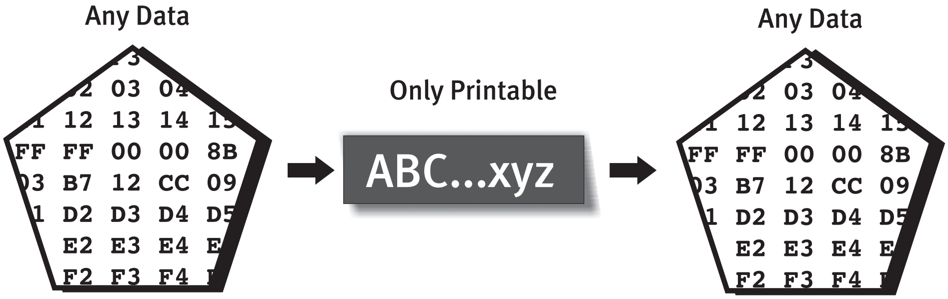
```
-23456789012345-23456789012345-23456789012345-23456789012345
Rodeo Drive    Huntington St. 14-th Street   Broadway
```

Concentrate$ cuts the array “street“ from the example string.

Conv_Base64$

ERG$ = Conv_Base64$ (Source$, method)

Function:	Converts arbitrary binary data to the base64 code for email attachments and decodes it again.



Base64 conversion / decoding

Parameters:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source with arbitrary binary data
Method	●	●	●	-	-	Conversion direction 0: binary ➡ Base64 1: binary ➡ Base64 -1: Base64 ➡ binary
						Function value:
Erg$	-	-	-	●	-	Result string converted according to “method”

Conv_Base64$ is used to represent arbitrary sets of data bytes (8 bit each) such as texts, binary data, graphics, sound, control codes etc. with a character set of 64 printable characters - and vice versa. The base64 coding is one-to-one and is used in the area of emails to code attachments of arbitrary content.

This conversion is also suitable for applications where transparent transmissions are required , although the transmission/memory channel works with code restrictions.

Base64 character table:

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The following section will deal with the base64 Code in detail.

Base64 Content-Transfer-Encoding

The base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable. The encoding and decoding algorithms are simple, but the encoded data are consistently only about 33 percent largerthan the unencoded data.

This encoding is virtually identical to the one used in Privacy Enhanced Mail (PEM) applications, as defined in RFC 1421. The base64 encoding is adapted from RFCred clear text.

A 65-character subset of US-ASCII is used, enabling 6 bits to be represented per printable character. (The extra 65th character, "=", is used to signify a special processing function.)

NOTE: This subset has the important feature that it is represented identically in all versions of ISO 646, including US ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. Other popular encodings, such as the encoding used by the uuencode utility and the base85 encoding specified as part of Level 2 PostScript, do not share these properties, and thus do not fulfill the portability requirements a binary transport encoding for mail must meet.

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single digit in the base64 alphabet. When encoding a bit stream via the base64 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first. That is, the first bit in the stream will be the high-order bit in the first byte, and the eighth bit will be the low-order bit in the first byte, and so on.

Each 6-bit group is used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string. These characters, identified in Table 1, below, are selected so as to be universally representable, and the set excludes characters with particular significance to SMTP (e.g., ".", CR, LF) and to the encapsulation boundaries defined in this document (e.g., "-").

Table 1: The base64 alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The output stream (encoded bytes) must be represented in lines of no more than 76 characters each. All line breaks or other characters not found in Table 1 must be ignored by the decoding software. In base64 data, characters other than those in Table 1, line breaks, and other white space probably indicate a transmission error, about which a warning message or even a message rejection might be appropriate under some circumstances.

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 24 input bits are available in an input group, zero bits are addeed (on the right) to form an integral number of 6-bit groups. Padding at the end of the data is performed using the '=' character. Since all base64 input is an integral number of octets, only the following cases can arise:

(1) the final quantum of encoding input is an integral multiple of 24 bits here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding,

(2) the final quantum of encoding input is exactly 8 bits here, the final unit of encoded output will be two characters followed by two "=" padding characters,

or

(3) the final quantum of encoding input is exactly 16 bits here, the final unit of encoded output will be three characters followed by one "=" padding character.

Because it is used only for padding at the end of the data, the occurrence of any '=' characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three.

Any characters outside of the base64 alphabet are to be ignored in base64-encoded data. The same applies to any illegal sequence of characters in the base64 encoding, such as "====="

Care must be taken to use the proper octets for line breaks if base64 encoding is applied directly to text material that has not been converted to canonical form. In particular, text line breaks must be converted into CRLF sequences prior to base64 encoding. The important thing to note is that this may be done directly by the encoder rather than in a prior canonicalization step in some implementations.

NOTE: There is no need to worry about quoting apparent encapsulation boundaries within base64-encoded parts of multipart entities because no hyphen characters are used in the base64 encoding.

Count_Patt$

```
CNT$   =   Count_Patt$ (  &  '
           Src$,          &  ' Source string is checked
           Patt$,         &  ' String with 1...nn patterns, length: Patt_ELen
           Patt_ELen,     &  ' Pattern entry length
           Pos,           &  ' Counting start position
           Cnt_Len)          ' Length of counting procedure in Src$
```

Function: Count_Patt$ checks a source string for the occurrence of given byte patterns and counts their frequency.

Parameters:

	B	W	L	S	F	
Src$	-	-	-	●	-	Source string with data which are to be checked for the occurrence of specific byte patterns. Start position and length are specified in parameters “Pos” and “Cnt_Len”.
Patt$	-	-	-	●	-	String with a list of patterns which are to be searched for and counted. Every pattern has a fixed length according to parameter “Patt_ELen”. ‹patt-0› ‹patt-1› ‹patt-2› ‹patt-3› ‹patt-4› ...
Patt_ELen	●	●	●	-	-	Length of pattern in string Patt$.
Pos	●	●	●	-	-	Start position in source string Src$.
Cnt_Len	●	●	●	-	-	Count length in Src$: The maximum number of bytes is analysed.
						Function value:
Cnt$	-	-	-	●	-	String with a list of LONG counter values. Every LONG value represents the frequency of the according pattern in the source string: ‹cnt-0› ‹cnt-1› ‹cnt-2› ‹cnt-3› ... ‹cnt-n-1›

Example:

```
Src$      = "00 00 0F 77 77 77 00 0F 0F 0F 0F"%
Patt$     = "00 0F 77 77"%
'            <=0=> <=1=>    <-- pattern index
Patt_ELen = 2
Pos       = 0
CLen      = Len (Src$)
CNT$  = Count_Patt$ ( Src$, Patt$, Patt_Elen, Pos, CLen)
'
' After executing "Count_Patt$":
'
'  CNT$ -->  "02 00 00 00 01 00 00 00"   '
'             <=== 0 ===> <=== 1 ===>    ' <- counter index
```

Another example:

```
Src$      = "abc de ed ef eg ee ee eee eee eee eee"
Patt$     = "e ee"
'             0 1    <-- pattern index
Patt_ELen = 2
Pos       = 0
CLen      = 11
CNT$  = Count_Patt$ ( Src$, Patt$, Patt_Elen, Pos, CLen)
'
' After executing "Count_Patt$":
'
'  CNT$ -->  "02 00 00 00 01 00 00 00"   '
'             <=== 0 ===> <=== 1 ===>    ' <- counter index
```

CRC

CRC (A$, Pos, Len, Start_Factor, Step, CRC)

Function: Calculates a CRC checksum from a string's data of a given length and starting from a start position.

```
Create 32-Bit CRC:  1. byte * (FACTOR)          ➡ sum up to CRC
                    2. byte * (FACTOR+1*STEP)   ➡ sum up to CRC
                    3. byte * (FACTOR+2*STEP)   ➡ sum up to CRC

                    4. byte * (FACTOR+3*STEP)   ➡ sum up to CRC
                        .           .
                        .           .
                  100. byte * (FACTOR+99*STEP)  ➡ sum up to CRC
                  101. byte * (FACTOR+100*STEP) ➡ sum up to CRC
                        .           .
                        .           .

        CRC sum = 32 bit = 8 HEX digits
                        (overflow: don't care)
```

CRC calculation

Parameters:

	B	W	L	S	F	
A$	-	-	-	●	-	Source data
Pos	●	●	●	-	-	Start position in source string: 0 ... (LEN(A$)-1)
Len	●	●	●	-	-	Number of bytes for CRC calculation 1...LEN(A$)
Start_Factor	●	●	●	-	-	Input: Start factor for CRC calculation Output: next start factor for continued CRC calculation
Step	●	●	●	-	-	Factor step width for CRC calculation
CRC	●	●	●	-	-	Input: Initial value for CRC calculation, usually = 0 Output: Result CRC

No function value

Continuous CRC Calculation with Adjustable Parameters

CRC is used for calculating simple checksums and CRCs in strings such as large and continuously flowing data streams. To implement this, the function contains:

Mere input parameters: A$, Pos, Len, Step
Input and output parameters: Start_Factor, CRC

Example for simply calculating a simple checksum:

example 1:

```
A$ = "Hello World"
START_FACTOR= 1
SUM = 0
CRC  (A$, 0, LEN(A$), Start_Factor, 0, SUM)    ' Step = 0
```

This example calculates a simple checksum from all bytes of string A$. The simple checksum's reliability is low, since basic errors such as confusion of bytes or 2 bit flips in the same position etc. will not be detected. The CRC produces reliable results by defining several factors:

example 2:

```
A$ = "Hello World"
START_FACTOR= 67
SUM = 0
CRC  (A$, 0, LEN(A$), Start_Factor, 2, SUM)    ' Step = 2
```

In case of vast amounts of data and in case of continuously transmitted data streams, the CRC can be calculated continuously - creating intermediate results.

The according example produces the same result as example 2:

example 3:

```
A$ = "Hello World"
START_FACTOR= 67
SUM = 0
CRC  (A$, 0, 5, Start_Factor, 2, SUM)  ' Step = 2
CRC  (A$, 5, 6, Start_Factor, 2, SUM)  ' Step = 2
```

When accessing CRC for the second time, the "Start_Factor" is already set to the next required factor (77) for continuing the CRC calculation; SUM already contains the CRC value from the first run. Both example 2 and example 3 produce the same final result.

Cut_and_Paste

```
A$  =  Cut_and_Paste$ ( SRC$, Offset, Len )     ' cut out String from String
R   =  Cut_and_PasteR ( SRC$, Offset )          ' cut out Real from String
N   =  Cut_and_PasteN ( SRC$, Offset, Len )     ' cut out Num from String
```

Function: Further “Cut_and_Paste” functions (cp. V5.0 „Cut_and_Paste“) for string, num and real variables.

Parameters:

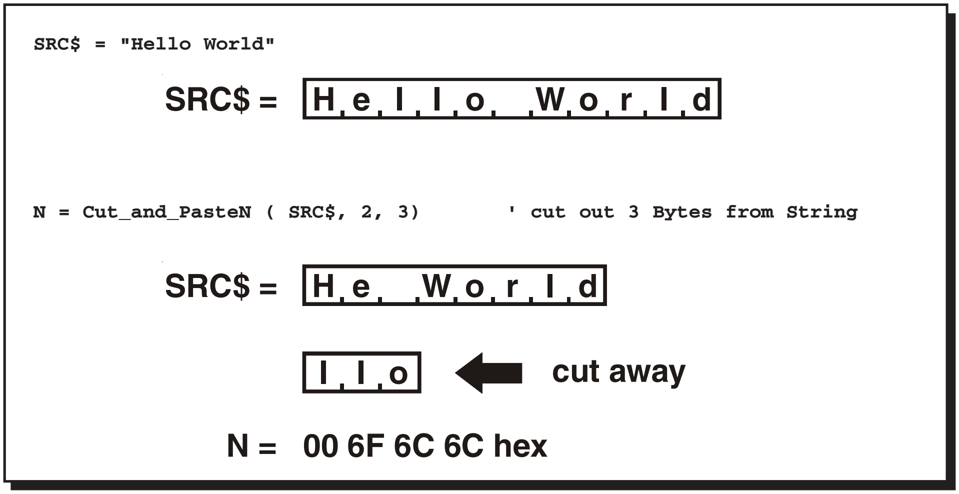
	B	W	L	S	F	
SRC$	-	-	-	●	-	Source string
Offset	●	●	●	-	-	Position in the source string from which the cutting begins.
Len	●	●	●	-	-	Number of bytes which are to be cut and transferred to the result variable.
						Function values:
A$	-	-	-	●	-	Cut string
R	-	-	-	-	●	Cut floating point value (real)
N	●	●	●	-	-	Cut integral value

These functions cut data bytes from a source string and transfer these bytes to the respective response variable.

Cut_and_Paste functions allow simply manipulating strings, adapting and cutting records, inter-tasks communicating, transferring characters/buffers etc.

An according example can be found on the next page.

Example:



```
SRC$ = "Hello World"
```

```
N = Cut_and_PasteN ( SRC$, 2, 3)       ' cut out 3 Bytes from String
```

Also see: INSERT$

Calc_ECC

ECC$ = Calc_ECC (Data$, Start_Pos, ECC_Method)

Function: Calculates an ECC (Error Correction Code) for a data block which can be used for detecting bit errors and for correcting 1 bit errors.

Application: Saving memory blocks in mass storages (SRAM, FLASH etc.). Also see function „Correct_ECC“.

Parameters:

	B	W	L	S	F	
DATA$	-	-	-	●	-	For this data string - or a part of it - the Error Correction Code is calculated. The ECC is calculated for exactly 256 bytes.
Start	●	●	●	-	-	0 ... nnnn start offset in DATA$
ECC_Method	●	●	●	-	-	ECC method selection: 00 method 0: ECC according to “SmartMedia™ Physical Format Specification Version 1.20” For further description of the format see: ”Correct_ECC” function
						Function value:
ECC$	-	-	-	●	-	24 result bits = 3 bytes in one string:

	7	6	5	4	3	2	1	0	Bit-No
Byte-1:	LP07	LP06	LP05	LP04	LP03	LP02	LP01	LP00	
Byte-2:	LP15	LP14	LP13	LP12	LP11	LP10	LP09	LP08	
Byte-3:	CP5	CP4	CP3	CP2	CP1	CP0	"1"	"1"	

Correct_ECC

Flag = Correct_ECC (Data$, Start, ECC_stored, ECC_calculated, ECC_Method)

Function:	Corrects a data block with possible parity error and corrects 1 error or signalises 2 errors. Application: Saving memory blocks in mass storages (SRAM, FLASH etc.). Also see function: „Calc_ECC“.

Parameters:

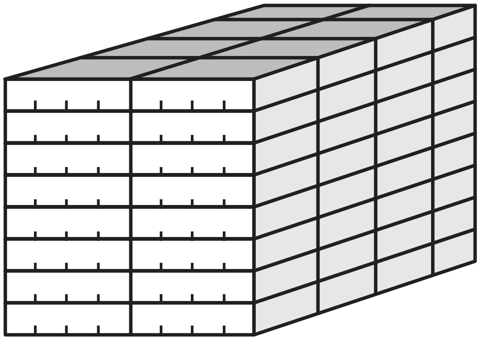
	B	W	L	S	F	
DATA$	-	-	-	●	-	For this data string - or a part of it - the Error Correction Code is calculated and correction is executed where applicable.
Start	●	●	●	-	-	0 ... nnnn start offset in DATA$
ECC_stored	●	●	●	-	-	The ECC as it is saved on the storage medium.
ECC_calculated	●	●	●	-	-	This is how the ECC was calculated from the real DATA in Data$.
ECC_Method	●	●	●	-	-	ECC method selection: 00 method 0: ECC according to “SmartMedia™ Physical Format Specification Version 1.20”
						Function value:
FLAG	●	●	●	-	-	Result flag: 00: OK - nothing left to correct: ECC_stored = ECC_calculated 01: 1 byte in DATA$ was corrected and therefore: ECC_stored = ECC_calculated 02: Correction was executed --> BUT: ECC_stored <> calculated

Correct_ECC

Check and Correct with Error Correction Code

Description of the ECC (Error Correction Code) creation according to “SmartMedia™ Physical Format Specification Version 1.20” from the SSFDC Forum:

A data block of 256 bytes is regarded as a bit stream of 2048 bits. Each of these 2048 bits contains a 11-bit address.



22 subsets of 1024 bits each are created from this. For each of these subsets of 1024 bits

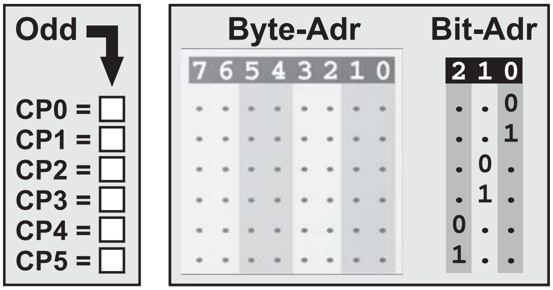
one ***ODD-Parity Bit***

is created.

The SmartMedia™ commission differentiates:

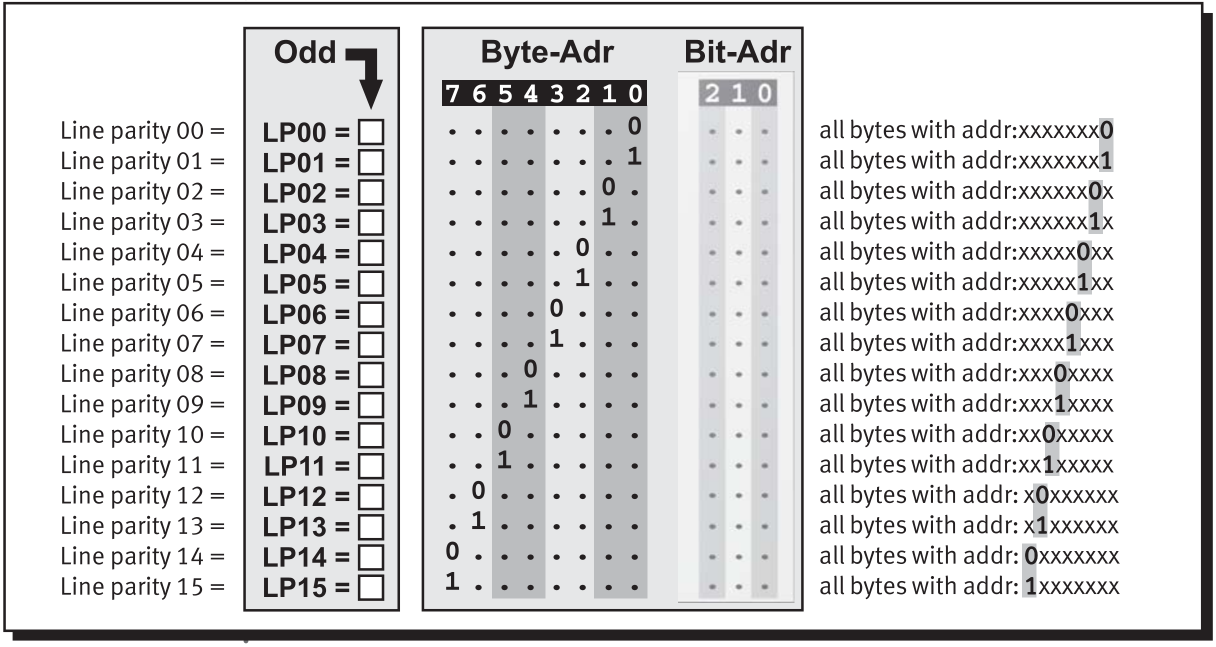
"6 column parities":	Bit addresses inside of bytes (3 addr bits), and
"16 line parities":	Which determine the single bytes (8 addr bits)

Calculation of the 6 (odd) column parity bits:



Column parity 0= **CP0 =** ☐ Bit addr: 000 + 010 + 100 + 110
Column parity 1= **CP1 =** ☐ Bit addr: 001 + 011 + 101 + 111
Column parity 2= **CP2 =** ☐ Bit addr: 000 + 001 + 100 + 101
Column parity 3= **CP3 =** ☐ Bit addr: 010 + 011 + 110 + 111
Column parity 4= **CP4 =** ☐ Bit addr: 000 + 001 + 010 + 011
Column parity 5= **CP5 =** ☐ Bit addr: 100 + 101 + 110 +111

Calculation of the 16 (odd) line parity bits:



	Odd	Byte-Adr 7 6 5 4 3 2 1 0	Bit-Adr 2 1 0	
Line parity 00 =	LP00 = ☐	 0	. . .	all bytes with addr:xxxxxxx0
Line parity 01 =	LP01 = ☐	 1	. . .	all bytes with addr:xxxxxxx1
Line parity 02 =	LP02 = ☐	. . . . . . 0 .	. . .	all bytes with addr:xxxxxx0x
Line parity 03 =	LP03 = ☐	. . . . . . 1 .	. . .	all bytes with addr:xxxxxx1x
Line parity 04 =	LP04 = ☐	. . . . . 0 . .	. . .	all bytes with addr:xxxxx0xx
Line parity 05 =	LP05 = ☐	. . . . . 1 . .	. . .	all bytes with addr:xxxxx1xx
Line parity 06 =	LP06 = ☐	. . . . 0 . . .	. . .	all bytes with addr:xxxx0xxx
Line parity 07 =	LP07 = ☐	. . . . 1 . . .	. . .	all bytes with addr:xxxx1xxx
Line parity 08 =	LP08 = ☐	. . . 0	. . .	all bytes with addr:xxx0xxxx
Line parity 09 =	LP09 = ☐	. . . 1	. . .	all bytes with addr:xxx1xxxx
Line parity 10 =	LP10 = ☐	. . 0	. . .	all bytes with addr:xx0xxxxx
Line parity 11 =	LP11 = ☐	. . 1	. . .	all bytes with addr:xx1xxxxx
Line parity 12 =	LP12 = ☐	. 0	. . .	all bytes with addr: x0xxxxxx
Line parity 13 =	LP13 = ☐	. 1	. . .	all bytes with addr: x1xxxxxx
Line parity 14 =	LP14 = ☐	0	. . .	all bytes with addr: 0xxxxxxx
Line parity 15 =	LP15 = ☐	1	. . .	all bytes with addr: 1xxxxxxx

22 ODD parity bits are combined in three bytes to the ECC (Error Correction Code) from 8 columns + 16 lines as follows:

	7	6	5	4	3	2	1	0	Bit-No
Byte-1:	LP07	LP06	LP05	LP04	LP03	LP02	LP01	LP00	
Byte-2:	LP15	LP14	LP13	LP12	LP11	LP10	LP09	LP08	
Byte-3:	CP5	CP4	CP3	CP2	CP1	CP0	"1"	"1"	

Correct_ECC is used to protect saved data in mass storages and long-term storages against unnotices modification and to detect and correct single bit errors.

Find_opt_Group$

```
Pos_Ndx$  =  Find_opt_Group$ (  &  '
             Nums$,             &  '  Quantities in cases
             Group_Ndx_List$,   &  ' Case combinations of groups
             Size_of_Group,     &  ' Number of cases in each group
             Size_of_Nums,      &  ' Number of bytes per case, always=2 (word)
             Target_Val,        &  ' Total target value
             Tolerance_Band,    &  ' Tolerance band
‹ opt ›      Start_Pos,         &  ' Start position in: “Group_Ndx_List$”
‹ opt ›      Target_Tol )          ' Target value tolerance for: “Target_Val”
```

Function: Find_opt_Group$ compiles an optimal group of cases, which contain different quantities, such as containers in filling machines.

There it is a specific task to find a combination of several containers, which has the quantity closest to the target value for filling, from a number of containers with different quantities of material. In such cases the deviation from a given target value should be minimal and a given tolerance band should be adhered to.

In these applications often a high decision speed is required, in order to be able to fully deploy the plant’s productivity. Find_opt_Group$ accelerates the decision process enormously by checking hundreds or thousands of combinations for their efficiency in one single function.

Find_opt_Group$ takes over the optimising process for compiling the optimal group of cases, the amount of which comes as close to the target quantity as specified by the parameters.

Parameters:

	B	W	L	S	F	
Nums$	-	-	-	●	-	String with the current quantities of all containers/ cases: 0 ... FFFFH unsigned. Number of cases according to parameter “Size_of_Group”, size of values according to parameter “Size_of_Nums” - currently always WORD.

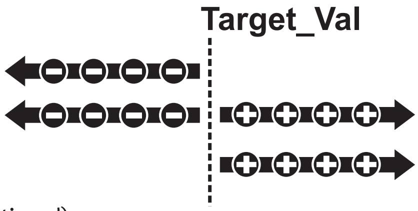
	B	W	L	S	F	
						Structure of Nums$: lo - hi lo - hi lo - hi lo - hi **Nums$ = “22 10 A1 12 E2 13 82 0B”%** #0 #1 #2 #3 Here: Cases 0 ... 3 Nums$ contains the current quantity of up to 32 cases (i.e. containers of the filling machine) as WORD values in the Big-Endian model (as common in Tiger-Basic). Unsigned value range: 0 ... 65535.
Group_Ndx_List$	-	-	-	●	-	String with index list of all case groups. This list contains all possible combinations of cases at given group size according to “Size_of_Group”. Indices are maintained as bytes, value range: 0 ... Size_of_Group-1

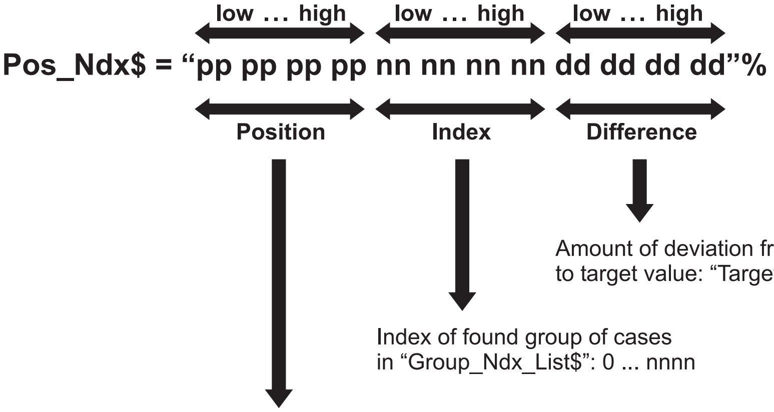
Group_Ndx_List$ = “00 01 02 00 01 03 00 02 03 01 ...”%

Group-0 Group-1 Group-2

Size_of_Group	●	●	●	-	-	Number of cases, which are to belong to one group: 1 ... number of existing cases
Size_of_Nums	●	●	●	-	-	Number of bytes required for the quantity of each case in Nums$. Compulsory = 2 (Word).
Target_Val	●	●	●	-	-	The desired total target value. This value should be reached exactly or as exactly as possible by combining the number of cases given in “Size_of_Group”.
Tolerance_Band	●	●	●	-	-	Defines the tolerance band for result calculation. “Find_opt_Group$” searches the group of cases, which has the lowest deviation from the desired target value (Target_Val). Specifying the tolerance band determines, if the target quantity should be

Find Optimal Group Compilation

	B	W	L	S	F	
						(-1) => less-than-or-equal (0) => less than, greater than (amount of difference) (+1) => greater-than-or-equal the target value. Tolerance bands: 
Start_Pos	●	●	●	-	-	(Optional) Start position in: “Group_Ndx_List$” 0 ... nnnn. Is used for e.g. being able to sequentially determine all exact or tolerance afflicted matches.
Target_Tol	●	●	●	-	-	(Optional) Target quantity tolerance for: “Target_Val” With this specification you can set a valid result deviation, to ensure that such a group is interpreted as OK.
						Function value:
Pos_Ndx$	-	-	-	●	-	3 LONG result values in string:



Amount of deviation from result
to target value: “Target_Val”

Index of found group of cases
in “Group_Ndx_List$”: 0 ... nnnn

Position of result group of cases
in “Group_Ndx_List$”

B W L S F

"Position":
Specifies the position of the result group of cases as existing in string"Group_Ndx_List$".
Values: 0 ... nnnn
Value: -1 => Error

"Index":
Specifies the index of the result group (as before).
Values: 0 ... nnnn
Value: -1 => Error

"Difference":
Signed LONG with the deviation of the found result to the set target value "Target_Val".

The function "Find_opt_Group$" provides numerous options for compiling a number of groups to a result quantity. "Find_opt_Group$" allows handling those tasks with just few program lines and at a short runtime.

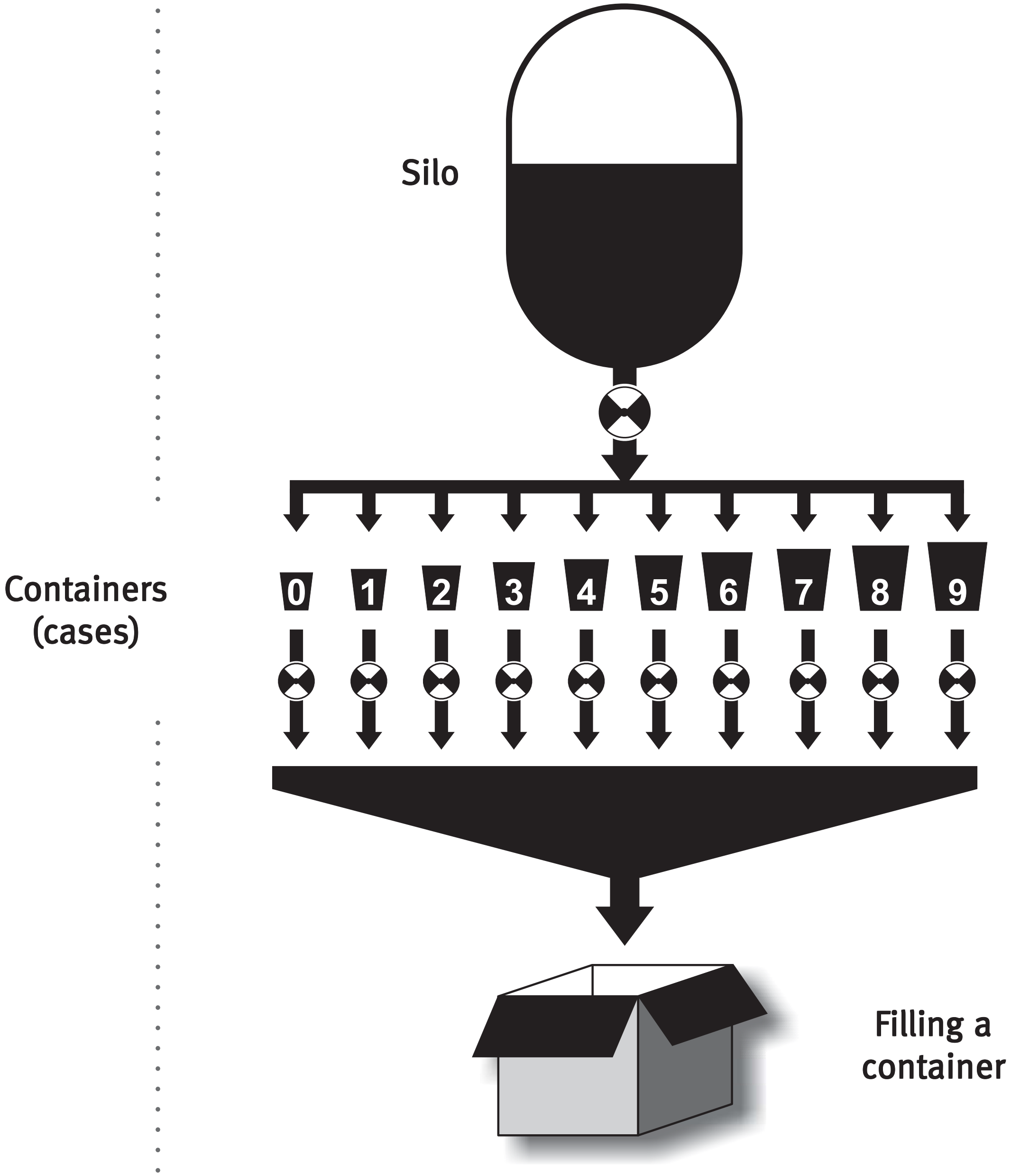
To put the functions across in detail, the following description is based on the model of a

filling machine

With this model we will demonstrate different functions and aspects.

The following figure depicts the schematic structure of such a filling machine. The plant in this example is used for filling bulk material, e.g. granulate material, into bags or barrels. The requirements of this task are

- high filling speed
- tight weight tolerances



Filling machine

For this purpose the plant’s structure is as follows: There are 10 containers in total, where the defined quantity of bulk material from the silo is filled into. The filling procedure should take as little time as possible. During operation there are, however, tolerances concerning the actual quantities filled so that each of the 10 containers is equipped with an electronic scale. Like this the real filling weight of each container can be determined fast and exactly during operation.

At the beginning of a filling cycle 10 weight values of 10 containers are provided. A weight as exact as possible has to be filled into a destination container by pouring together the content of several containers in the most efficient way.

A simple numerical example:

Case:	Amount of material in the cases:	Numerical value:
#0	************************************	36
#1	*******************************	31
#2	******************************	30
#3	*******************	19
#4	***********************	23
#5	******************	18
#6	*****************	17
#7	***************	15
#8	************	12
#9	********************	20

Assumed it is our task to achieve a target quantity of 100 by combining 2... 6 cases from the example presented above:

- If possible, the exact result is to be achieved.
- If this is not possible, the result should lie within a given tolerance band.
- If the latter is not possible either, this should be detected as an error and the according error processing should be initiated.

By trying you will find e.g. the following 3 combinations of 4 cases - with different results:

1. Group: #0 + #1 + #2 + #8 = 109 Tolerance = +9
2. Group: #0 + #1 + #4 + #8 = 102 Tolerance = +2
3. Group: #1 + #2 + #5 + #9 = 99 Tolerance = -1

Find_opt_Group$

Find Optimal Group Compilation

Soon it becomes clear that there might be an exact result, maybe even several. However, this is usually not that easy to detect. The number of possible combinations of 2 or more cases results in:

Group size	<-------------- Number of case combinations		---------->	
1:	10	/ 1	=	10
2:	10*9	/ 1*2	=	45
3:	10*9*8	/ 1*2*3	=	120
4:	10*9*8*7	/ 1*2*3*4	=	210
5:	10*9*8*7*6	/ 1*2*3*4*5	=	252
6:	10*9*8*7*6*5	/ 1*2*3*4*5*6	=	210
7:	10*9*8*7*6*5*4	/ 1*2*3*4*5*6*7	=	120
8:	10*9*8*7*6*5*4*3	/ 1*2*3*4*5*6*7*8	=	45
9:	10*9*8*7*6*5*4*3*2	/ 1*2*3*4*5*6*7*8*9	=	10
Total			=	1022

For calculating the result an algorithm, which systematically finds a sufficiently exact result by using combinatorics and evaluation, is used. This task is fulfilled by “Find_opt_Group$”.

It is another important task of “Find_opt_Group$”, to achieve a preferably fast runtime performance. This is done by dint of the prepared tables with all possible case combinations. This information is provided in the parameter “Group_Ndx_List$” for the according group size:

```
Size_of_Group    = 2
Group2_Ndx_List$ = "00 01 00 02 00 03 00 04 00 05 00 06 00 07 ..."%
'                    <===> <===> <===> <===> <===> <===> <===>
'
' Groups with 2 cases are kept as groups of 2 bytes = 2 indices in
' the list. In total all 45 possible combinations are saved in this
' string so that it has a length of 90 bytes.
```

The string for groups with 3 or more cases:

```
Size_of_Group    = 3
Group3_Ndx_List$ = "00 01 02 00 01 03 00 01 04 00 01 05 00 01 06 ..."%
'                    <======> <======> <======> <======> <======>
'
' Groups consisting of 3 cases are kept as groups of 3 bytes = 3 indices
' in the list. In total all 120 possible combinations are saved in this
' string so that is has a length of 360 bytes.
```

Find Optimal Group Compilation

“Find_opt_Group” can be used with 6, 7 or 8 parameters. It is advisable for many technical applications to also use the 8th parameter (target tolerance).

Like this the runtime performance can be improved considerably.

In technical applications it is usually not important, to calculate the absolutely optimal result of all - this would mean computing and comparing every single possible combination. Here it is sufficient to find an effectually good result. You set a result tolerance which is still acceptable and a tolerance range: So a valid result is found with the first combination, which meets those requirements.

Depending on the machine’s construction you can also make decisions about the case group size. In the next example bags are to be filled with 25 kg and the following 10 case fillings are measured:

Case:	Quantity of material in cases:	in kg:
#0	***********************************	4.76
#1	************************************	4.88
#2	*************************************	4.92
#3	*************************************	4.99
#4	**************************************	4.93
#5	**************************************	5.09
#6	***************************************	5.17
#7	**	5.28
#8	***	5.33

This machine fills the containers with a flat ramp, every case with ca. 5 kg, however, with under- and overfilling. In this case it obviously only makes sense to check the groups of 5 cases, since there would no better result with less or more cases. In other cases especially small cases with small tare quantities could make sense:

#0	***	23.66
#1	*****	1.88
#2	****	1.42
#3	****	1.24
#4	***	0.89
#5	**	0.56
#6	**	0.42
#7	*	0.29
#8	*	0.19

Find Optimal Group Compilation

Depending on the task only one these group sizes are worked with, which possible solutions could exist for. Likewise the most likely group size is started with and the valid tolerance range is utilised for speed optimisation.

The example program “Fillmachine_Find_opt_Group_Vxxx.TIG”demonstrates the basic functionality.

Without specifying the 8th parameter all possible combinations are individually calculated and compared. This results in the respectively best combinatory result for a certain group size. Only in special cases, when an exact match is found, the search procedure is aborted. This procedure is always to be used, if the best possible result has to be achieved.

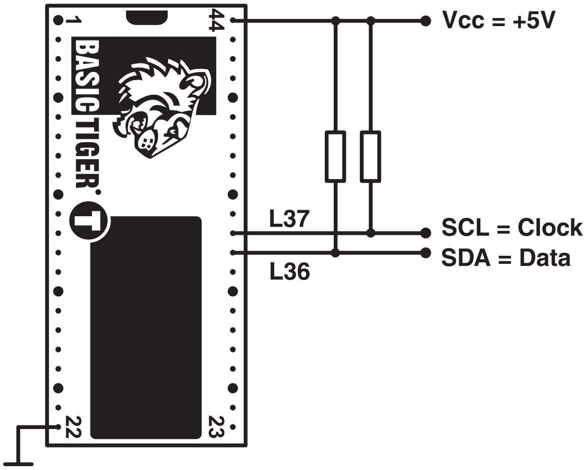
Finally the 7th parameter Start_Pos can be used to continue the search from an arbitrary position of Group_Ndx_List$, if a result is found before. You could, e.g., calculate all possible solutions like this (see: “Fillmachine_Find_opt_Group_Vxxx.TIG”).

I2CL_Setup

I2CL_Setup (Port, Clock_Pin, Data_Pin, Speed)

Set up I2C-Bus respectively ISO 7816

Function: Specifies Tiger pins used and a possible timing speed reduction for low level functions I2C-Bus / ISO 7816 communication. All I2CL... functions use the Tiger as bus master.



I2C bus / ISO 7816 bus (example)

Parameters:

	B	W	L	S	F	
Port	●	●	●	-	-	Internal port for signals: SDA and SCL
Clock_Pin	●	●	●	-	-	Clock output pin: (Bit-no.: 0...7) = clock generated by master
Data_Pin	●	●	●	-	-	DATA-I/O Pin: (Bit-no.: 0...7) = bidirectional
Speed	●	●	●	-	-	0 = no speed reduction 1...20 = speed reduction
						Function value:
-	-	-	-	-	-	No function value

Example:

Example

```
I2CL_Setup  ( 3, 7, 6, 0 )
```

This function line implements the following definition for the I2C-Bus low level and the ISO-7816 communication.

SCL = Clock	**SDA = Data**
port 3, bit 7	port 3, bit 6

0 ==› no speed reduction

Every further I2C-Bus / ISO 7816 low level function access uses the definitions implemented by I2CL_Setup.

Both bus lines SCL and SDA (CLOCK and DATA) are designed as “wired-and” signals. On every bus line there are:

- A pull-up resistor which pulls the line’s level to +5V as well as
- Open collector outputs of the bus sharing units and
- High-resistance inputs of the bus sharing units

If all bus sharing units are inactive, i.e. no connected bus sharing unit affects a bus line, for every unit a +5V level = “1” is applied.

If one or several units pull a bus line with the open collector output to GND level (0V), the bus signal is “0” for all units - “wired-and”. The level diagram for I2C bus / ISO 7816 mirrors this circumstance as follows:

- Only 1 clock signal is depicted - the clock is always given by the master
- The 2 data signals of the 2 bus sharing units concerned are depicted, so that the flow of information can be seen

Signal transmission on the I2C bus is not carried out at a fixed baud rate; however, it is limited for some chips and configurations. According to the setting of the “Speed”-parameter in I2CL_SETUP (...), I2CL_Read$ (...) accounts for the shortest bit time possible. Besides data transmission on the I2C bus is carried out completely statically - i.e. it can be stopped at any point and continued unaltered.

Note:	The ISO-7816 transmission uses an additional RESET line for resetting the address counter. In case of an ISO 7816 application this line is implemented with any available Tiger pin or extension pin and is controlled by the BASIC program. The “Answer-to-Reset” (ATR) procedure is standardised by ISO 7816-3.

There are several example programs with the prefix “I2CL_” and the extension “TIG”.

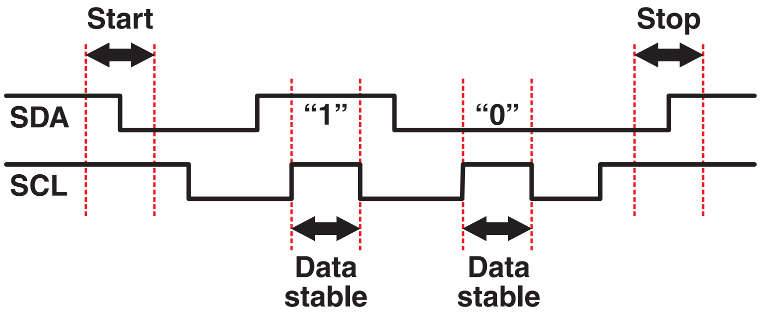
Also see:	I2CL_START, I2CL_STOP, I2CL_RELEASE, I2CL_READ$, I2CL_WRITE, I2CL_RESULT

I2CL_Start

I2CL_Start (speed)

Set start condition

Function: Creates start condition on the I2C-Bus / ISO-7816 channel.
This function is used as a bus master.



I2C-Bus / ISO 7816 signals

Parameters:

	B	W	L	S	F	
Speed	●	●	●	-	-	0 = no speed reduction 1...20 = speed reduction
						Function value:
-	-	-	-	-	-	No function value

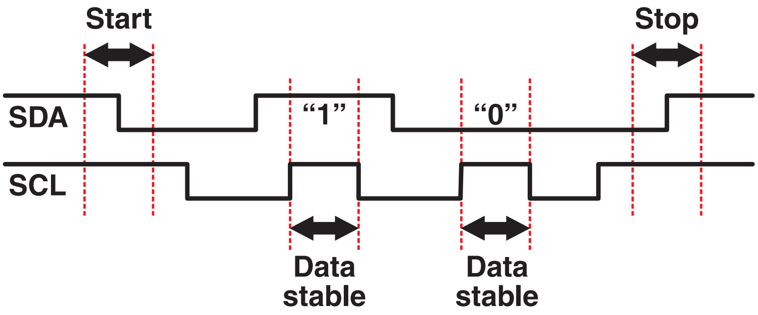
Also see: I2CL_SETUP, I2CL_STOP, I2CL_RELEASE, I2CL_READ$, I2CL_WRITE, I2CL_RESULT

I2CL_Stop

I2CL_Stop (speed)

Set stop condition

Function: Creates stop condition on the I2C-Bus / ISO-7816 channel.
This function is used as a bus master.



I2C-Bus / ISO 7816 signals

Parameters:

	B	W	L	S	F	
Speed	●	●	●	-	-	0 = no speed reduction 1...20 = speed reduction
						Function value:
-	-	-	-	-	-	No function value

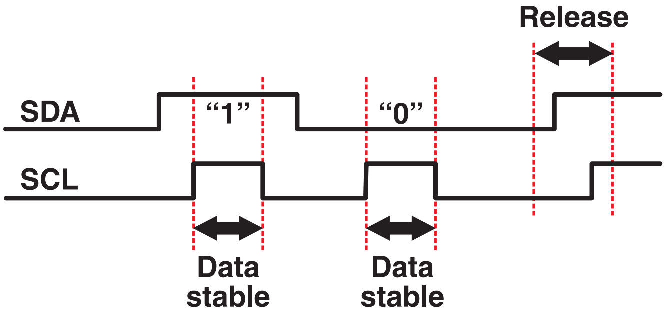
Also see: I2CL_SETUP, I2CL_START, I2CL_RELEASE, I2CL_READ$, I2CL_WRITE, I2CL_RESULT

I2CL_Release

I2CL_Release (Dummy)

Release SDA + SCL lines

Function: Switches both bus lines to high-impedance state, without creating a stop condition. This function is used as a bus master.



I2C-Bus / ISO 7816 signals

Parameters:

	B	W	L	S	F	
Dummy	●	●	●	-	-	Parameter having no effect
						Function value:
-	-	-	-	-	-	No function value

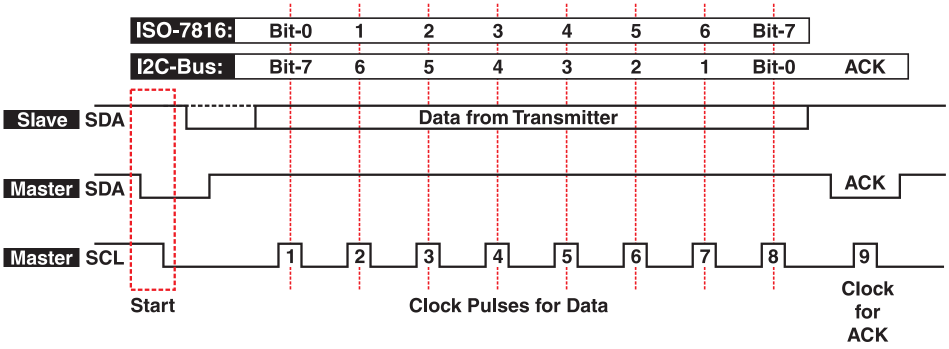
Also see: I2CL_SETUP, I2CL_START, I2CL_STOP, I2CL_READ$, I2CL_WRITE, I2CL_RESULT

I2CL_Read$

```
A$   =   I2CL_Read$ ( nob )           ' READ Byte(s) from I2C bus
A$   =   I2CL_Read$ ( nob, 7816 )     ' READ Byte(s) from ISO 7816 bus
```

Read from I2C-Bus respectively from ISO 7816

Function: Reads the specified number of bytes from the I2C-Bus / ISO 7816 bus. This function is used as a bus master.



I2C-Bus / ISO 7816 Read

Parameters:

	B	W	L	S	F	
nob	●	●	●	-	-	Number of bytes to be read: 0 ... 32
						Function value:
A$	-	-	-	●	-	Result string contains the received bytes

Both bus lines SCL and SDA (CLOCK and DATA) are designed as “wired-and” signals. On every bus line there are:

- A pull-up resistor which pulls the line’s level to +5V as well as
- Open collector outputs of the bus sharing units and
- High-resistance inputs of the bus sharing units

If all bus sharing units are inactive, i.e. no connected bus sharing unit affects a bus line, for every unit a +5V level = “1” is applied.

If one or several units pull a bus line with the open collector output to GND level (0V), the bus signal is “0” for all units - “wired-and”. The level diagram for I2C bus / ISO 7816 mirrors this circumstance as follows:

- Only 1 clock signal is depicted - the clock is always given by the master
- The 2 data signals of the 2 bus sharing units concerned are depicted, so that the flow of information can be seen

Signal transmission on the I2C bus is not carried out at a fixed baud rate; however, it is limited for some chips and configurations. According to the setting of the “Speed”-parameter in I2CL_SETUP (...), I2CL_Read$ (...) accounts for the shortest bit time possible. Besides data transmission on the I2C bus is carried out completely statically - i.e. it can be stopped at any point and continued unaltered.

Note: The ISO-7816 transmission uses an additional RESET line for resetting the address counter. In case of an ISO 7816 application this line is implemented with any available Tiger pin or extension pin and is controlled by the BASIC program.

The “Answer-to-Reset” (ATR) procedure is standardised by ISO 7816-3.

There are several example programs with the prefix “I2CL_” and the extension “TIG”.

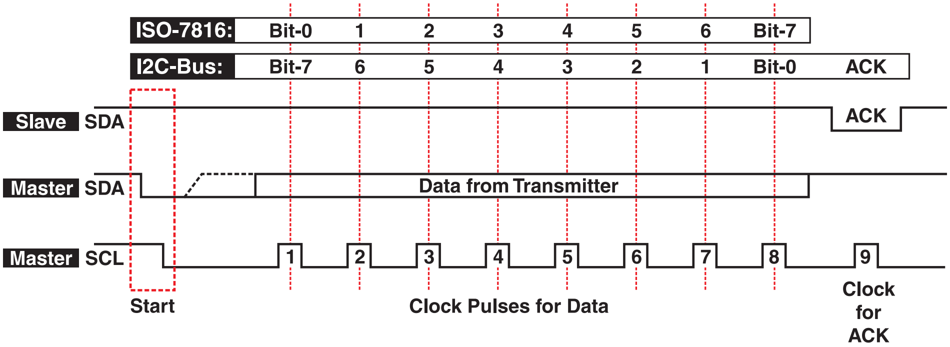
Also see: I2CL_SETUP, I2CL_START, I2CL_STOP, I2CL_RELEASE, I2CL_WRITE, I2CL_RESULT

I2CL_Write

```
NAK  =  I2CL_Write ( A$ )              ' Write A$ to I2C bus
NAK  =  I2CL_Write ( N1, N2 )          ' Write N2 Bytes of N1 to I2C bus

NAK  =  I2CL_Write ( A$, 7816 )        ' Write A$ to ISO 7816
NAK  =  I2CL_Write ( N1, N2, 7816 )    ' Write N2 Bytes of N1 to ISO 7816
```

Write to I2C-Bus respectively ISO 7816

Function: Writes the specified number of bytes to the I2C-Bus / ISO 7816 bus. This function is used as a bus master.



I2C-Bus / ISO 7816 Write

Parameters:

	B	W	L	S	F	
A$	-	-	-	●	-	Transmission string: 0 ... 32 characters
N1	●	●	●	-	-	Numerical value, from which 1, 2, 3 or 4 bytes are transmitted
N2	●	●	●	-	-	Number of bytes which are to be transmitted: 1...4 bytes
7816	-	●	●	-	-	Identifier for “ISO 7816” transmission format

Function value:

	B	W	L	S	F	
NAK	●	●	●	-	-	Number of received ‹NAK› conditions during I2C-Bus transmission Of no relevance during ISO 7816 transmission

Both bus lines SCL and SDA (CLOCK and DATA) are designed as “wired-and” signals. On every bus line there are:

- A pull-up resistor which pulls the line’s level to +5V as well as
- Open collector outputs of the bus sharing units and
- High-resistance inputs of the bus sharing units

If all bus sharing units are inactive, i.e. no connected bus sharing unit affects a bus line, for every unit a +5V level = “1” is applied.

If one or several units pull a bus line with the open collector output to GND level (0V), the bus signal is “0” for all units - “wired-and”. The level diagram for I2C bus / ISO 7816 mirrors this circumstance as follows:

- Only 1 clock signal is depicted - the clock is always given by the master
- The 2 data signals of the 2 bus sharing units concerned are depicted, so that the flow of information can be seen

Signal transmission on the I2C bus is not carried out at a fixed baud rate; however, it is limited for some chips and configurations. According to the setting of the “Speed”-parameter in I2CL_SETUP (...), I2CL_Read$ (...) accounts for the shortest bit time possible. Besides data transmission on the I2C bus is carried out completely statically - i.e. it can be stopped at any point and continued unaltered.

Note: The ISO-7816 transmission uses an additional RESET line for resetting the address counter. In case of an ISO 7816 application this line is implemented with any available Tiger pin or extension pin and is controlled by the BASIC program.

The “Answer-to-Reset” (ATR) procedure is standardised by ISO 7816-3.

There are several example programs with the prefix “I2CL_” and the extension “TIG”.

Also see: I2CL_SETUP, I2CL_START, I2CL_STOP, I2CL_RELEASE, I2CL_READ$, I2CL_RESULT

I2CL_Result

Flag = I2CL_Result ()

Function: Reads the return code of the I2CL function executed last.

Parameters:

	B	W	L	S	F	
-	-	-	-	-	-	No parameter
						Function value:
Flag	●	●	●	-	-	Return code: 81H = read OK 82H = write OK 83H = erase OK 84H = setup OK F1H = read error F2H = write error F3H = erase error F4H = setup error FAH = parameter error FFH = I2C / function error

Also see: I2CL_SETUP, I2CL_START, I2CL_STOP, I2CL_RELEASE, I2CL_READ$, I2CL_WRITE

Insert$

N$ = Insert$ (Source$, S_Pos, Ins$, I_Pos, I_Len)

Function: Inserts a string (or a part of it) into another string.

```
Before:       SOURCE$ = "Hello world"
                 INS$ = "to the "

              SOURCE$ = INSERT$ (SOURCE$, 6, INS$, 0,7)


After:        SOURCE$ = "Hello to the world"
```

Insert string

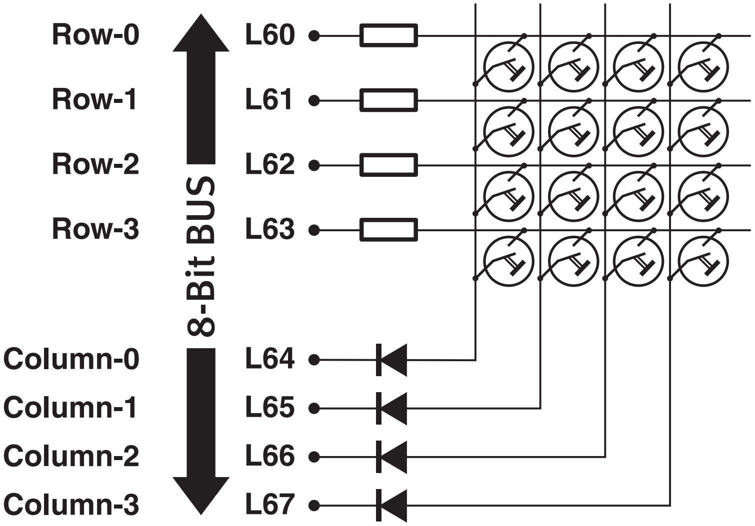
Parameters:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source data string
S_Pos	●	●	●	-	-	Insert position in source string: 0 ... nnnn
Ins$	-	-	-	●	-	String which is inserted
I_Pos	●	●	●	-	-	Position in Ins$ of first character to be inserted: 0 ... nn
I_Len	●	●	●	-	-	Number of characters from Ins$ which are to be inserted
						Function value:
N$	-	-	-	●	-	Result string with inserted string

Key_Direct

```
N   =   Key_Direct (Port, Column, Mode, Speed)   ' Read key column
N   =   Key_Direct (Port, Column, Mode)          ' Read key column
```

Function: Implements small keyboards economically. This is done by the Key_Direct function, which uses the data bus to read up to 16 keys or switches (e.g. DIPs). No extra Tiger connection is needed and all bus operations can still be executed at the same time (LCD outputs, xPort in-/outputs... dev-driver ... etc.).



Connect 16 keys to 4 x R and 4 x D

Parameters:

	B	W	L	S	F	
Port	●	●	●	-	-	Bus port, e.g. 6 or 8
Column	●	●	●	-	-	Column for reading
Mode	●	●	●	-	-	0 = Read number of the key pressed first: 0,1,2,3 No key pressed = -1 Any further key pressed: ignore

	B	W	L	S	F	
						1 = Reads 1NIBBLE with all 4 keys: Value “0” = key/switch open Value “1” = key/switch closed
Speed	●	●	●	-	-	0 = full speed 1...16 = reduce speed for longer line and/or high resistance
						Function value:
N	●	●	●	-	-	Key-no. 0...3, -1 or nibble value, depending on the mode

In order to read a 4 x 4 switch matrix, 4 nibbles are simply polled one after another:

Example

switch matrix

```
KN0  =   Key_Direct (6, 0, 1, 0)' Lies Tasten-Spalte-0 => Nibble
KN1  =   Key_Direct (6, 1, 1, 0)' Lies Tasten-Spalte-1 => Nibble
KN2  =   Key_Direct (6, 2, 1, 0)' Lies Tasten-Spalte-2 => Nibble
KN3  =   Key_Direct (6, 3, 1, 0)' Lies Tasten-Spalte-3 => Nibble
```

In case of an input keyboard mode= 0 is used. The following example shows a small section of the code which can be used as a task and fills the keyboard buffer (KEY_BUF$) with characters. If standard key elements with pressure point contacts are used, no further debouncing is needed - the 60 ms idle time of each scanning loop suffice. Please use available functions as “Debounce” for more complex cases (see example programs).

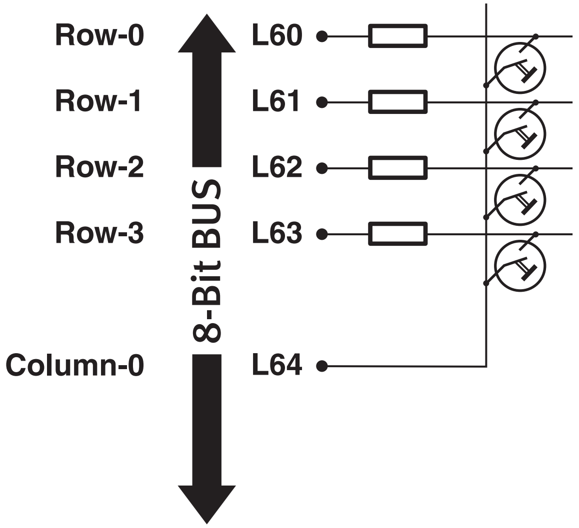
Example

4 x 4 keyboard

```
KEY_PRESSED = 0                 ' Key-State = initial
FOR EVER = 0 TO 0 STEP 0        ' <---------- endless loop ----------->
  FOR COL = 0 TO 3              ' Scan 4 Key Columns
    N = Key_Direct (6, COL, 0,0)' Read Key Column => Key-No
    IF N<>-1 AND KEY_PRESSED = 0 THEN                 '
      KEY_PRESSED = 1           ' Yes, Key pressed
      GOTO DONE                 '
    ENDIF                       '
  NEXT                          '
  KEY_PRESSED = 0               ' No Key pressed => Key-State = initial

DONE:                                  '
  IF KEY_PRESSED = 1 THEN       ' If Key pressed, then generate Key-Code
    CODE = N+4*COL              '
    CHAR$ = MID$ ("0123456789abcdef", CODE, 1)  ' convert to ASCII Char
    KEY_BUF$ = KEY_BUF$ + CHAR$ ' <-- Char into Keyboard-Buffer
    KEY_PRESSED = 2             ' -> Next State
  ENDIF                         '
  WAIT_DURATION 60              ' slow down a bit for keyboard scanning
NEXT                            ' <---------- endless loop ----------->
```

Economical Keyboards - Less Parts, no Extra Tiger Connection

Depending on a project's requirements every kind of partial circuit can be used, if only a few keys are to be used:



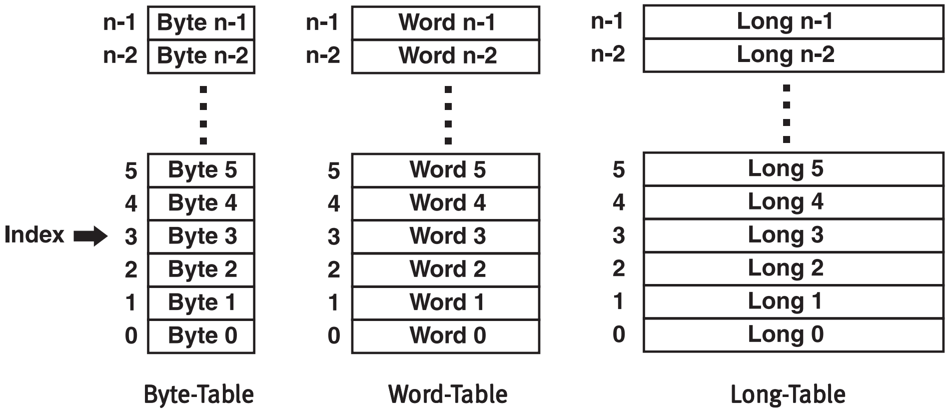
Key matrix in partial circuit

Likewise an arbitrary combination of keys (dynamic) and switches (static) is possible. On the hardware side there is no differentiation between keys and static switches, both kinds of contacts are treated accordingly by the scan routine.

BLookup#, WLookup#, LLookup#

Byte table: A = BLookup# (Index)
Word table: A = WLookup# (Index)
Long table: A = LLookup# (Index)

Function: Especially fast access to “Look Up” tables in DATA-FLASH or in strings. Lookup is used to make mathematical functions, characteristic curves, calibration functions etc. quickly available for real time applications via table access.



Parameters:

	B	W	L	S	F	
Index	●	●	●	-	-	Index: 0 ... (n-1) table access
						Function value:
A	●	●	●	-	-	Table value: Byte, word or long

There are 4 lookup tables of each type (byte, word and long) available. The function name defines the table’s type and number, e.g.:

BLookup2	=>	Byte table 2
LLookup4	=>	Long table 4

Lookup Tables

WLookup1 => Word table 1
WLookup4 => Word table 4

The following 12 tables are available in total:

Byte table-1	Word table-1	Long table-1
Byte table-2	Word table-2	Long table-2
Byte table-3	Word table-3	Long table-3
Byte table-4	Word table-4	Long table-4

Lookup is called with an index as a parameter:

Example

```
A   =  WLookup4  ( Index )          ' accesses word table-4
                                    ' Index: 0 ... n-1
```

Before using a lookup table, it is initialised with valid values. This takes place by allocation to a table string or a DATA-FLASH area.

Initialise string table

```
A$ = "10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 05 05 05 05 04 03 02 01"%
A = BLOOKUP1 (1, A$)                ' initialize lookup table in string
                                    ' 1 = dummy Index
```

This allocation creates a link between:

BLOOKUP1 <==> A$

Initialising functions accordingly, if the value table exists in the DATA-FLASH area:

Initialise FLASH table

```
DATALABEL  FLASH_TABLE
.
.
A = BLOOKUP1 (1, FLASH_TABLE)       ' initialize Lookup-Table in FLASH
.                                   ' 1 = dummy Index
.
FLASH_TABLE::    ... DATA-Flash Table Area ...
```

Example taken from a program:

Example

```
A$ = "10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 05 05 05 05 04 03 02 01"%
N = BLOOKUP1 (1,A$)     ' associate Lookup-Table to String A$
PRINT #1, "<1>";        ' clear Text LCD
FOR N=0 TO 10           '
  PRINT #1, "=>"; BLOOKUP1(N); " ";
  WAIT DURATION 500     ' --- wait a moment ---
NEXT                    '
```

The allocation of string A$ to the lookup table 1 / byte is carried out by specifying 2 parameters:

1. Index (dummy in this case)
2. Lookup table's location (string or DATA-FLASH area)

In this example both the index and the function value N are only used as dummies. For every further access to this lookup table the following function's short form with only one parameter = index is chosen:

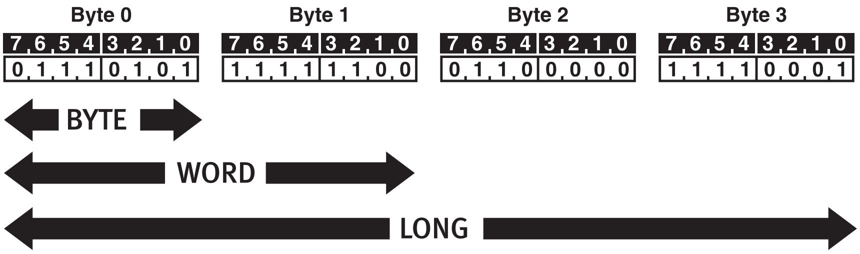
X = BLOOKUP1 (index)

This notation produces the fastest runtimes. If a large number of lookup tables have to be processed, also the first form with two parameters (index + table_specification) is possible. This notation needs a slightly longer runtime.

The table's structure:

WORDs and LONGs in the lookup tables are filed as a big endian format - as it is always done in Tiger-BASIC:

Low-order byte first - highest byte last:



?Lookup#

Lookup Tables

Using lookup tables can be of great advantage for the system’s performance. Access to lookup tables is carried out very fast, so that an enormous increase of speed is created when using lookup tables to replace time-consuming computations.

For instance for real control or graphic presentations in the area of engineering the 16-digit exactness of REAL numbers often is not required. 3 or 4 decimal places are usually sufficient. In those cases speed is the more important factor.

For example when controlling drives, transforming coordinates for maps on a graphic display, adjusting quickly etc.

Accelerating mathematic functions

Assume having to repeatedly calculate a mathematical connection with trigonometric functions to be able to run a control cycle. In order to increase the speed of this procedure, you choose a transformation to LONG values instead of using REAL values, e.g. the measured value “bar” is transformed to the LONG value “microbar”. The LONG number range is large enough to represent a value of 1,000 bar in microbar. All fast computing procedures are brought to table format, as possible, and are processed 50 times (or even more) faster by lookup.

Such computing tables can be:

1.) calculated again and written to strings on every program start (takes long).

2.) calculated and written to the DATA FLASH area at the first start of the program. For every further start there will already be a valid table.

3.) invoked into the DATA FLASH area of the program already at compilation time as data file.

An example program in directory “Example” demonstrates such an application.

Pin

A = Pin (ADR, Bit_Pos)   ' read bit of port

Function: Reads a single pin of a port.

Parameters:

	B	W	L	S	F	
ADR	●	●	●	-	-	Port address
Bit_Pos	●	●	●	-	-	Bit position: 0 ... 7
						Function value:
A	●	●	●	-	-	Read bit of port, value: 0, 1

Example program:

```
TASK MAIN                              ' Begin Task MAIN
  IF  PIN (8, 7) = 1  THEN             ' test Bit-7 in Port-8 = 1  ??
'     ...                              ' do this if Bit = "1"
  ELSE                                 '
'     ...                              ' do this alternatively, as Bit = "0"
  ENDIF                                '
END                                    ' Program End
```

Also see XPort system for extended I/Os:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

Push + Pop

```
FLG    =  PushN ( A$, NUM, Anz )          ' Push Anz Bytes of NUM to A$
FLG    =  PushR ( A$, REAL )              ' Push 1 REAL = 8 Bytes to A$
FLG    =  Push$ ( A$, Stri$, Pos, Anz )   ' Push Anz Bytes from Stri$
                                          ' starting at Pos to A$

NUM    =  PopN ( A$, Anz )                ' Pop Anz Bytes from A$ to NUM
REAL   =  PopR ( A$)                      ' Pop 8 Bytes from A$ to REAL
Stri$  =  Pop$ ( A$, Anz )                ' Pop Anz Bytes from A$ to Stri$
```

Function: PUSHs data (byte, word, long, real, string) to string and vice versa POPs from string => to byte, word, long, real, string.

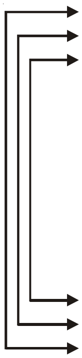
Parameters:

	B	W	L	S	F	
A$	-	-	-	●	-	Stack area
NUM	●	●	●	-	-	Source data (integer) to PUSH
REAL	-	-	-	-	●	Source data (real) = 8 bytes to PUSH
Stri$	-	-	-	●	-	Source data (string) = nn bytes to PUSH
Pos	●	●	●	-	-	Start position in Stri$
Anz	●	●	●	-	-	Number of bytes to PUSH / POP
						Function value:
FLG	●	●	●	-	-	Number of actually PUSHed bytes
N	●	●	●	-	-	POPped numerical result
R	-	-	-	-	●	POPped real result
X$	-	-	-	●	-	POPped string result

Stack Structure for Strings

The functions PUSH and POP for the data types byte, word, long, real and string allow setting up stack structures in strings. In the usual application of stacks (FILO = first in, last out) data are read back (POPped) from the stack in exactly reversed sequence as they were written (PUSHed) to the stack before. You can also create data conversions by a PUSH / POP sequence (as demonstrated below). Stacks allow setting up recursive structures, local variables and parameters.

Example program sequence:



```
FLG = PushN (A$, N1, 4)          ' Save N1 (long) to Stack on A$
FLG = PushR (A$, R1)             ' Save R1 (real) to Stack on A$
FLG = Push$ (A$, S$, 0, 12)      ' Save S$ (string of 12) to Stack on A$
.
.
.
N1 = 12345                       ' do other calculations with variables
R1 = 1.2345                      ' N1, R1 and S$
S$ = "-"                         '
.
.
.
S$ = Pop$ (A$, 12)               ' get back S$
R1 = PopR (A$)                   ' get back R1
N1 = PopN (A$, 4)                ' get back N1
```

Please note the nested PUSH - POP structure. Mixing up the order of data types when reading back with POP would immediately destroy the data consistency.

The PUSH function adds bytes to the string, i.e. it extends it. POP shortens the string accordingly. PUSH function and its effect on string A$:

```
A$ = "Hello"
```

A$ = H e l l o

```
B$ = " World"                    ' set B$ to:  „ World“
FLG = Push$ (A$, B$, 0, 6)       ' Push 6 Bytes of B$ to Stack (A$)
```

A$ = H e l l o   W o r l d

Stack structure for Strings

Integer and real values are also maintained as Big Endian presentation on the stack:

```
A$ = "Hello"
```

A$ = H e l l o

```
N1 = 12345678H                    ' set N1 to:    12 34 56 78 hex
FLG = PushN (A$, N1, 2)           ' Push 2 Bytes of N1 to Stack (A$)
```

A$ = H e l l o **+ <78H> <56H>**

Strings are maintained character by character without changing their sequence. PUSH adds strings to the stack; POP withdraws the according number of characters from the back and shortens the stack string:

```
A$ = "Hello World"
```

A$ = H e l l o   W o r l d

```
N = PopN (A$, 2)                                  ' Pop 2 BYTEs from A$
```

A$ = H e l l o   W o r

N = 00 00 64 6C hex

```
X$ = Pop$ (A$, 5)                                 ' Pop 5 Chars to String from A$
```

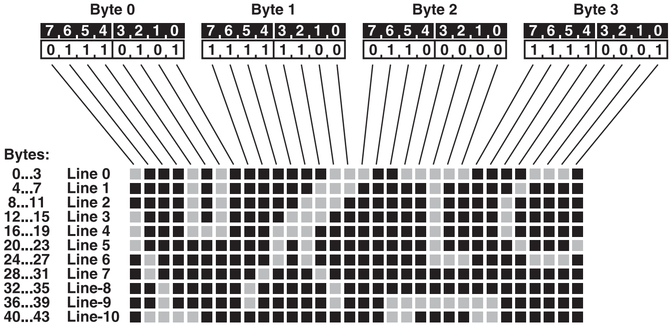
A$ = H e l l

X$ = o   W o r

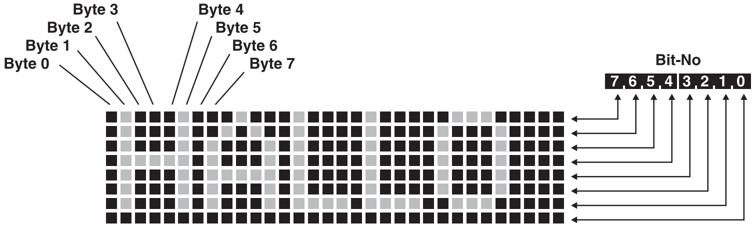
Graphic_Reformat

Graphic_Reformat (SRC$, DEST$, Width, Height, Re_Format)

Function: Reverses the pixel structure of a pixel graphic: horizontal => vertical.



Horizontal pixel columns



Vertical pixel columns

Parameters:

	B	W	L	S	F	
Src$	-	-	-	●	-	Source with horizontally oriented pixels
Destin$	-	-	-	●	-	Destination with vertically oriented pixels
Width	●	●	●	-	-	Format width in pixels: 1 ... nnnn
Height	●	●	●	-	-	Format height in pixels: 1 ... nnnn
Re_Format	●	●	●	-	-	Reformat flag: 0 ... 3
						0: Bit-7 = left ➡ Bit-7 = bottom
						1: Bit-7 = right ➡ Bit-7 = bottom
						2: Bit-7 = left ➡ Bit-7 = top
						3: Bit-7 = right ➡ Bit-7 = top

No function value

Graphic_Reformat

Graphic Pixel Reformatting

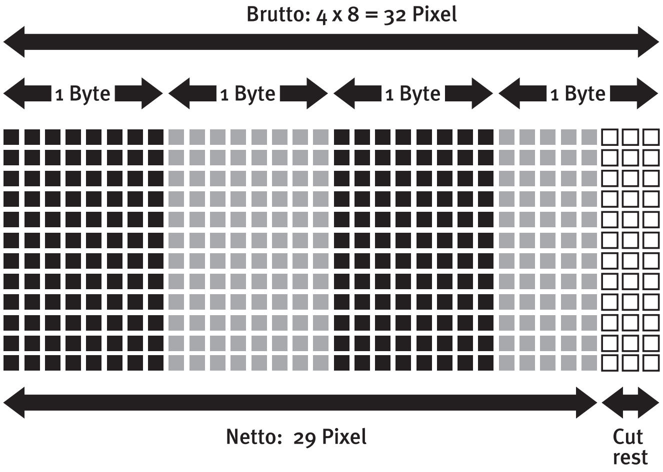
Graphic_Reformat converts the pixel data's structure.

In Tiger-BASIC a pixel graphic is administered in horizontal rows. 1 byte = 8 horizontal pixels, in which bit 7 represents the respective pixel on the left in a group of 8 pixels. This structure of pixels is also often used in LC displays and row-oriented monitors and printing units.

Another common pixel orientation is based on matrix printing units; all 8 bits of a byte represent the according 8 pixels arranged vertically. Graphic_Reformat allows reformatting to all 4 possible combinations with only one BASIC line and a short runtime:

horizontal: Bit-7 ⬌ left ➡ vertical: Bit-7 ⬌ top
horizontal: Bit-7 ⬌ left ➡ vertical: Bit-7 ⬌ bottom
horizontal: Bit-7 ⬌ right ➡ vertical: Bit-7 ⬌ top
horizontal: Bit-7 ⬌ right ➡ vertical: Bit-7 ⬌ bottom

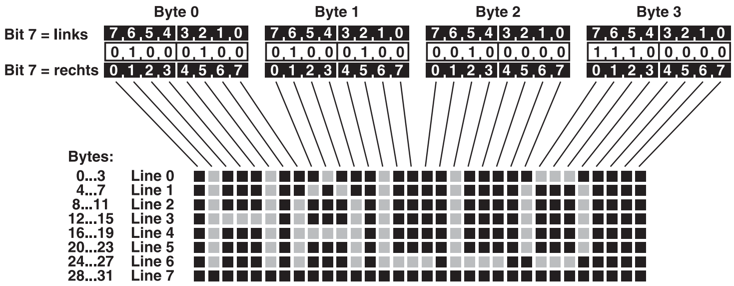
Graphic_Reformat processes all widths and heights, even if they are not a multiple of 8. In this case the according remaining pixels at the right or bottom end drop out.



Example: Format width = 29 pixels

Example program „GRAPHIC_REFORMAT_Demo_001.TIG“:

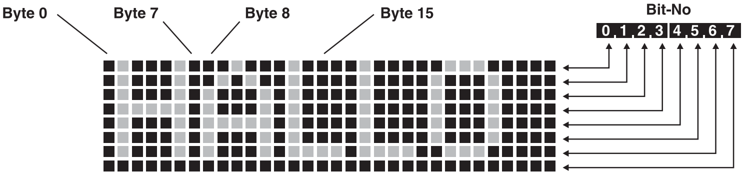
Let’s use as the starting pixel structure the “Hallo” graphic from the function’s description:



In the case of horizontal pixel rows (with bit 7 on the left, just as on a graphic LC-display) this results in the following bytes (in hex):

```
44  44  20  E0  44  A4  21  10  45  14  21  10  7C  14  21  10
45  F4  21  10  45  14  21  10  45  17  BC  E0  00  00  00  00
```

Now the pixels structure is converted so that the bytes are arranged in vertical pixel columns. At first this is done by Reformat-flag 0, with bit 7 being on the left of the starting structure and being at the bottom in the destination structure. The bytes are generated as shown in the following figure:



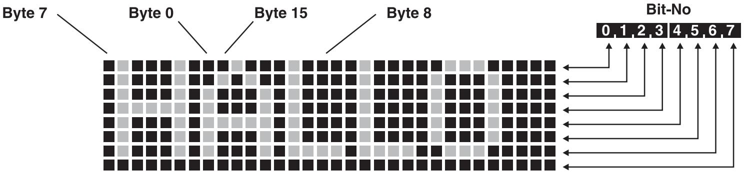
Graphic_Reformat (Src$, Dest$, 32, 8, 0)

The following bytes (in hex) result from the vertical pixel columns :

```
00  7F  08  08  08  7F  00  7C  12  11  12  7C  00  7F  40  40
40  00  7F  40  40  40  00  3E  41  41  41  3E  00  00  00  00
```

Now we transform the original pixel structure with Reformat-flag 1, this time bit 7 being on the right in the starting structure and again at the bottom of the destination structure. The bytes are generated as shown in the following figure:

Graphic Pixel Reformatting

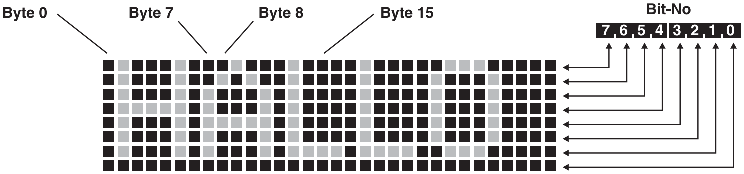


Graphic_Reformat (Src$, Dest$, 32, 8, 1)

The following bytes (in hex) result from the vertical pixel columns:

```
7C  00  7F  08  08  08  7F  00  40  40  7F  00  7C  12  11  12
3E  00  40  40  40  7F  00  40  00  00  00  00  3E  41  41  41
```

This time we use Reformat-flag 2, bit 7 being on the left in the starting structure and at the top in the destination structure. The bytes are created as shown below:

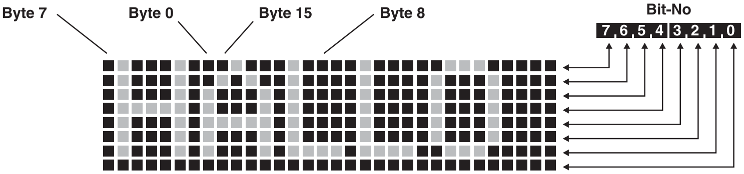


Graphic_Reformat (Src$, Dest$, 32, 8, 2)

The following bytes (in hex) result from the vertical pixel columns:

```
00  FE  10  10  10  FE  00  3E  48  88  48  3E  00  FE  02  02
02  00  FE  02  02  02  00  7C  82  82  82  7C  00  00  00  00
```

Finally we use Reformat-Flag 3, with bit 7 being on the right of the starting structure and at the top of the destination structure. The bytes are created as shown below:



Graphic_Reformat (Src$, Dest$, 32, 8, 3)

This following bytes (in hex) result from the vertical pixel columns:

```
3E  00  FE  10  10  10  FE  00  02  02  FE  00  3E  48  88  48
7C  00  02  02  02  FE  00  02  00  00  00  00  7C  82  82  82
```

Text_Reformat$

```
Dest$   =   Text_Reformat$ ( Source$, Dest_Width, &
                             Dest_Cut_Wrap,       &
                             Dest_Fill_Blank,     &
                             Dest_Line_End )
```

Function: Reformats ASCII text strings for output media such as terminals, LCDs, printers etc.

Parameters:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source text: ASCII, printables and ‹CR›‹LF› allowed
Dest_Width	●	●	●	-	-	Destination width: printable chars per row
Dest_Cut_Wrap	●	●	●	-	-	Destination lines: 0 = cut, 1 = wrap
Dest_Fill_Blank	●	●	●	-	-	Fill destination lines with blanks: 0 = no, 1 = yes
Dest_Line_End	●	●	●	-	-	Destination lines' end flag: 0: inactive 1: place‹CR› to the line's end 2: place ‹CR›‹LF›to the line's end

Function value:

	B	W	L	S	F	
Dest$	-	-	-	●	-	Reformatted result string

Text_Reformat$ is used to format text outputs for different output devices. ASCII text with printable characters and possibly ‹CR›‹LF› codes is expected as a source string. .

Text_Reformat allows the following operations:

- Reset length of lines
- Cut lines
- Wrap lines
- Fill lines with blanks
- remove and/or add ‹CR› ‹LF› from/to the line's end

Text_Reformat$ is also used for e.g. partially scrolling in different windows on a screen.

Text Reformatting

Reformatting example:

Source string:

```
123456789.123456789.123456789.
"the quick brown fox jumps over"<CR><LF>
"the quick brown fox jumps over"<CR><LF>
```

Destination strings:

❶
```
123456789.123456789.
"the quick brown fox "<CR><LF>
"jumps over"<CR><LF>
"the quick brown fox "<CR><LF>
"jumps over"<CR><LF>
```

❷
```
123456789.123456789.
"the quick brown fox "
"jumps over          "
"the quick brown fox "
"jumps over          "
```

❸
```
123456789.123456789.
"the quick brown fox "
"the quick brown fox "
```

❹
```
123456789.123456789.
"the quick brown fox "<CR>
"the quick brown fox "<CR>
```

❺
```
123456789.123456789.
"the quick brown fox "<CR><LF>
"the quick brown fox "<CR><LF>
```

1.) Dest$ = Text_Reformat$ (Source$, 20, 1, 0, 2)
2.) Dest$ = Text_Reformat$ (Source$, 20, 1, 1, 0)
3.) Dest$ = Text_Reformat$ (Source$, 20, 0, 0, 0)
4.) Dest$ = Text_Reformat$ (Source$, 20, 0, 0, 1)
5.) Dest$ = Text_Reformat$ (Source$, 20, 0, 0, 2)

Scan_or_Skip

New_Pos = Scan_or_Skip (SRC$, Charset$, Pos, Scan_Skip)

Function: Scans or skips given character collectives in strings. Scan_or_Skip is typically used to set up command interpreters and to analyse terms and data structures.

Parameters:

	B	W	L	S	F	
SRC$	-	-	-	●	-	Source string
Charset$	-	-	-	●	-	Charset flag string, length: exactly 256 bytes There is a flag for every character code in the string: Flag value 0: Code does ***not*** belong to charset Flag value X: Code belongs to charset
Pos	●	●	●	-	-	Start position in source string for scanning
Scan_Skip	●	●	●	-	-	Flag decides about way of scanning: Positive value -> SCAN Negative value -> SKIP
						Function value:
New_Pos	●	●	●	-	-	Found position according to scan criteria: 0.. nnn That is the position of the first character of this kind. -1: Not Found, reached end of string -2: Error, empty string or Pos outside of SRC$

Scan_or_Skip checks a source string for the presence of specific character sets. In the case of “Scan” the occurrence of characters of a certain character set is searched for, starting from a given position. As soon as such a character is found scanning is completed and the character’s position in the source string is transferred as a function value.

Example:

```
Blank_Flag$ = Fill$ ("00"%, 256)
Blank_Flag$ = NTOS$ (Blank_Flag$, 32, 1, 0FFH)
NPos = Scan_or_Skip ("The quick brown", Blank_Flag$, 0, 1) 'Scan for Blank
```

Searching for Specific Characters

Blank_Flag$ is a flag string consisting of 256 bytes, all bytes = 00, apart from the byte at position 32 (20H). This is the flag for the ASCII space character. Therefore this flag string defines a character set which only consists of a space character.

After function call NPos has the value 3, which is the position of the character found first of the character set according to Blank_Flag$.

Now you could extend the character set with "separators" in general, e.g. all characters apart from letters and numbers.

```
Separator$ = "&
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' 00...0F
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' 10...1F
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' 20...2F
00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF  &    ' 30...3F
FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  &    ' 40...4F
00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF  &    ' 50...5F
FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  &    ' 60...6F
00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF  &    ' 70...7F
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' 80...8F
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' 90...9F
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' A0...AF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' B0...BF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' C0...CF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' D0...DF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  &    ' E0...EF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF"%     ' F0...FF
```

In the case of "Skip" the occurrence of characters which are NOT part of a defined character set is searched for, starting from a given position. As soon as such a character is found, scanning is completed and the character's position in the source string is transferred as a function value.

Example:

```
Blank_Flag$ = Fill$ ("00"%, 256)
Blank_Flag$ = NTOS$ (Blank_Flag$, 32, 1, 0FFH)
NPos = Scan_or_skip ("  A   B  C    D", Blank_Flag$, 0, -1) ' Scan for
                                                            ' NON Blank
```

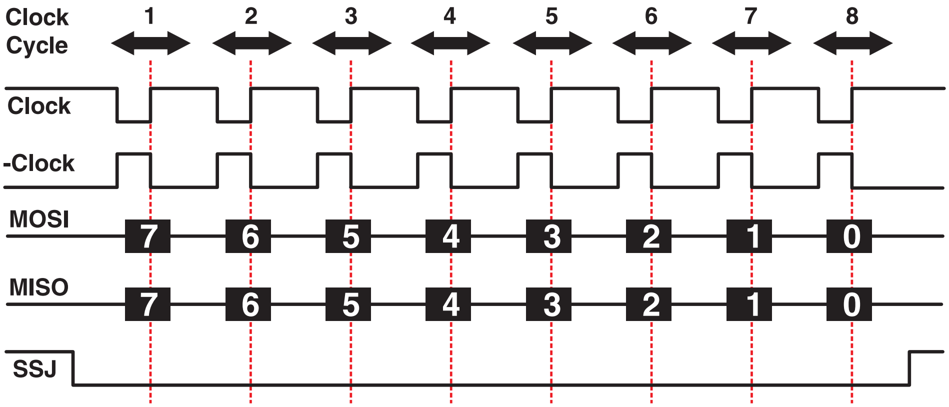
After function call NPos has the value 2, which is the position of the character found first, which is NOT part of the character set according to Blank_Flag$.

SPI_SETUP

```
FLG  =  SPI_SETUP ( CLK_MOSI_Port, CLK_Pin, MOSI_Pin,  &
                    SSJ_Port,  SSJ_Pin,                &
                    MISO_Port, MISO_Pin, MSB_First)
```

Specifying the pins for a SPI-Bus

Function: Specifies the SPI bus for the function: SPI_IO$ (...).



SPI Communication

Parameters:

	B	W	L	S	F	
CLK_MOSI_Port	●	●	●	-	-	Internal port for output signals: CLK + MOSI
CLK_Pin	●	●	●	-	-	Clock output pin: (Bit-No.: 0...7) = clock generated by master
MOSI_Pin	●	●	●	-	-	DATA output pin: (Bit-No.: 0...7) = master-OUT, slave-IN
SSJ_Port	●	●	●	-	-	Internal port for output signal: SSJ
SSJ_Pin	●	●	●	-	-	SSJ output pin: (Bit-No.: 0...7) = Low-active CE for SPI device
MISO_Port	●	●	●	-	-	Internal port for input signal: MISO
MISO_Pin	●	●	●	-	-	DATA input pin: (Bit-No.: 0...7)
MSB_First	●	●	●	-	-	Flag: 0=LSB first, X=MSB first
						Function value:
OK_FLAG	●	●	●	-	-	0 = everything OK!, X = 1...n: nth parameter incorrect

Serial Chip Interfacing

The SPI_SETUP is executed once during program sequence and defines the I/O configuration for all following SPI accesses.

Example:

Example

```
FLG  =      SPI_SETUP  ( 8,0,1,  8,2,  8,3,   -1)
```

This function line implements the follwowing definitions for SPI communication:

Clock	**MOSI**	**MISO**	**SSJ**
Port-8, Bit-0	Port-8, Bit-1	Port-8, Bit-2	Port-8, Bit-3

-1 ==> MSB first

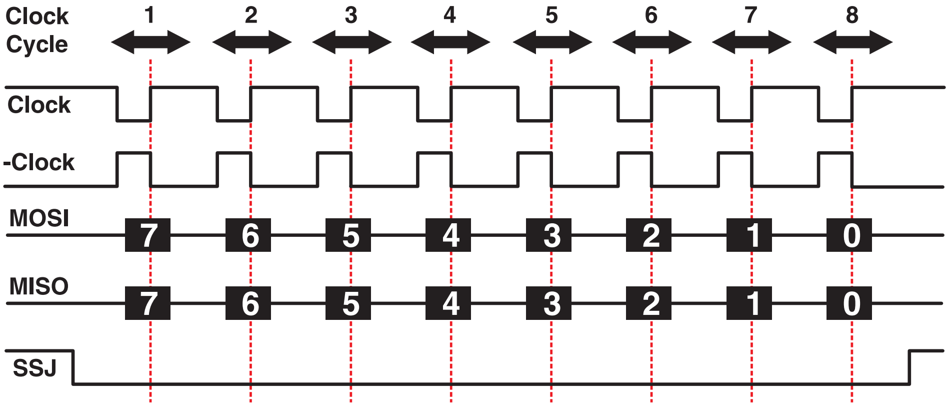
Also see: SPI_IO$

SPI_IO$

Rec$ = SPI_IO$ (Transm$)

Communication via SPI bus

Function: Transmits and receives data at the same time via the PSI Interface as specified in SPI_SETUP.



SPI Communication

Parameters:

	B	W	L	S	F	
Transm$	-	-	-	●	-	Transmission string, is transmitted to an external device via the SPI channel defined by "SPI_SETUP". The string's length defines how many bytes of data are both sent and received. String length: 0 ... max. string length
						Function value:
Rec$	-	-	-	●	-	String with received data bytes. Rec$ has the same length as Transm$ after this operation. The maximum length of the Rec$ string according to declaration defines the maximum length, which can be received with one function call.

Example:

Example

```
Rec$ = "hello"            ' just write something in receive string
Tra$ = "1234567890"       ' this what we are going to transmit
Rec$ = SPI_IO (Transm$)   ' send and receive through SPI channel
```

The device connected to the SPI channel received the following byte sequence:

“1234567890”

The device sent the following bytes:

“the quick”

This byte sequence was assigned to string Rec$.

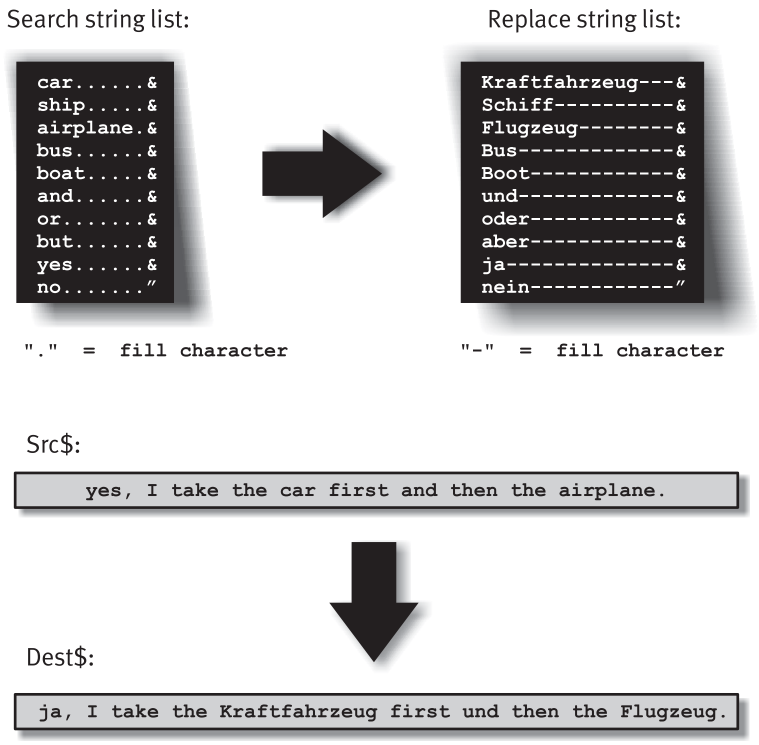
Also see: SPI_SETUP

Universal_Convert$

```
Dest$ = Universal_Convert$ ( Src$, Search_List$, Replace_List$ )
Dest$ = Universal_Convert$ ( Src$, Search_List$, Replace_List$, Start )
Dest$ = Universal_Convert$ ( Src$, Search_List$, Replace_List$, Start, Len )
```

Convert search words to replace words

Function: Universal string converter with search string list and replace string list.



Search string list:

```
car......&
ship.....&
airplane.&
bus......&
boat.....&
and......&
or.......&
but......&
yes......&
no......."
```

"." = fill character

Replace string list:

```
Kraftfahrzeug---&
Schiff----------&
Flugzeug--------&
Bus-------------&
Boot------------&
und-------------&
oder------------&
aber------------&
ja--------------&
nein------------"
```

"-" = fill character

Src$:

```
yes, I take the car first and then the airplane.
```

Dest$:

```
ja, I take the Kraftfahrzeug first und then the Flugzeug.
```

Conversion source => destination

Universal_Convert$ searches the whole list of search words at every source string position. If there is a match, the word is replaced immediately by the replace-word and the search is continued after the pasted replace word. If there is no match the next byte position in the source string is being called on and all search words are checked again until the source string's end is reached.

The sequence of search words in the search list is important in two ways:

a) Conversion speed.
Frequently occurring search words in the source string should be placed at the top of the list as possible. This reduces the search effort and accelerates conversion.

b) Conversion logic.
What is found first is converted first. This has essential effects on the result (see example 2).

Parameters:

	B	W	L	S	F	
Src$	-	-	-	●	-	Input string which is to be converted
Search_List$	-	-	-	●	-	String with a list of search words Format: 1. Byte = Field length 2. Byte = Fill character- ignored! These bytes are followed by fields of constant length with search words which are possibly filled with fill characters at the end.
Replace_List$	-	-	-	●	-	String with a list of replace words. Same format as the search list.
Start	●	●	●	-	-	Optional start position in Src$ for the conversion procedure.
Len	●	●	●	-	-	Optional length specification for search area in Src$
						Function value:
Dest$	-	-	-	●	-	Converted destination string

The program code for the conversion example given above is the following.

Example 1

```
Search_List$ = "&      '
<9>.&                  ' Field length = 9, fill character = "."
car......&             '  1. search word
ship.....&             '  2. search word
airplane.&             '  3. search word
bus......&             '  4. search word
boat.....&             '  5. search word
and......&             '  6. search word
or.......&             '  7. search word
but......&             '  8. search word
yes......&             '  9. search word
no......."             ' 10. search word

Replace_List$ = "&
<16>-&                 ' Field length = 16, fill character = "-"
Kraftfahrzeug---&      '  1. replace word
Schiff----------&      '  2. replace word
Flugzeug--------&      '  3. replace word
Bus-------------&      '  4. replace word
Boot------------&      '  5. replace word
und-------------&      '  6. replace word
oder------------&      '  7. replace word
aber------------&      '  8. replace word
ja--------------&      '  9. replace word
nein------------"      ' 10. replace word

Source$ = "yes, I take the car first and then the airplane."

Destin$ = Universal_Convert$ (Source$, Search_List$, Replace_List$)
```

Source$:

```
yes, I take the car first and then the airplane.
```

Destin$:

```
ja, I take the Kraftfahrzeug first und then the Flugzeug.
```

Example 2 (containing German forms of greeting, but you'll get the idea)

Attention!

```
Search_List$ = "&      '
<9>.&                  ' Field length = 9, fill character = "."
Sgr......&             '  1. search word
Sge......&             '  2. search word
SgH......&             '  3. search word
SgDuH....&             '  4. search word
Mfg......&             '  5. search word
Mfgs.....&             '  6. search word
Hvl......&             '  7. search word
HGs......&             '  8. search word
AlG......&             '  9. search word
Bb......."             ' 10. search word

Replace_List$ = "&
<31>-&                 ' Field length = 31, fill character = "-"
Sehr geehrter------------------&        '  1. replace word
Sehr geehrte-------------------&        '  2. replace word
Sehr geehrte Herren------------&        '  3. replace word
Sehr geehrte Damen und Herren--&        '  4. replace word
Mit freundlichem Gruss---------&        '  5. replace word
Mit freundlichen Grüssen-------&        '  6. replace word
Hochachtungsvoll---------------&        '  7. replace word
Herzliche Grüße----------------&        '  8. replace word
Alles Gute---------------------&        '  9. replace word
Bis bald-----------------------"        ' 10. replace word

Source$ = "SgDuH,<CR><LF><LF>Es bleibt wie besprochen!<CR><LF><LF>AlG"

Destin$ = Universal_Convert$ (Source$, Search_List$, Replace_List$)
```

Source$:

```
"SgDuH,<CR><LF><LF>Es bleibt wie besprochen!<CR><LF><LF>AlG"
```

Destin$:

```
"Sehr geehrte Damen und Herren,

Es bleibt wie besprochen!

Alles Gute"
```

Basically Universal_Convert$ can convert in both directions , e.g. from the short form to the long form (example 2) and back. This is done by exchanging the search and replace lists:

```
Src$ = "SgDuH,<CR><LF><LF>Es bleibt wie besprochen!<CR><LF><LF>AlG"
D1$  = Universal_Convert$ (Src$,Search_List$, Replace_List$) ' D1$ = long
D2$  = Universal_Convert$ (D1$, Replace_List$, Search_List$) ' D2$ = short
D3$  = Universal_Convert$ (D2$, Search_List$, Replace_List$) ' D3$ = long
D4$  = Universal_Convert$ (D3$, Replace_List$, Search_List$) ' D4$ = short
```

Please note that this definition of search and replace lists contains a logic error. You will notice it when the following conversion and the according reconversion are executed:

Conversion:

`"SgH!"` ➡ `"Sehr geehrte Herren!"`

Reconversion:

`"Sehr geehrte Herren!"` ➡ `"Sge Herren!"`

This conversion command is non-ambiguous, but it is not one to one.

In this case remedy is possible - simply by changing the sequence of search and replace words in the lists. Just place the longer words in front of the short words in order to avoid the premature conversion of „Sehr geehrte“:

Search_List$:

```
SgH......&
SgDuH....&
Sgr......&
Sge......&
Mfg......&
Mfgs.....&
Hvl......&
HGs......&
AlG......&
Bb.......”
```

`"." = fill character`

Replace_List$:

```
Sehr geehrte Herren------------&
Sehr geehrte Damen und Herren--&
Sehr geehrter------------------&
Sehr geehrte-------------------&
Mit freundlichem Gruss---------&
Mit freundlichen Grüssen-------&
Hochachtungsvoll---------------&
Herzliche Grüße----------------&
Alles Gute---------------------&
Bis bald-----------------------”
```

`"-" = fill character`

Typical applications:

Compressed data banks

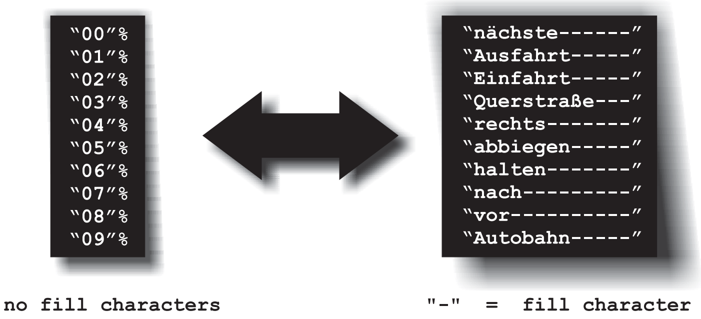
Security

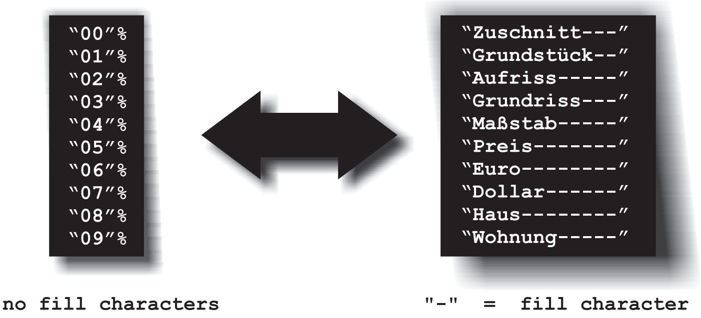
Code converter

Device and interface adjustments

Universal_Convert$ is typically used for:

- adaptive data compressions
- Application specific data compression, e.g. highly compressed data banks in embedded systems with clear text messages:





- Character set adjustments:

    character => character
    character => character sequence
    character sequence=> character
    character sequence=> character sequence

- Device adjustments
- Code suppression, filtering, replacements
- Security applications, encryption

UNPACK_DC$

DATA$ = UNPACK_DC$ (CTRL$, SRC$, POS, LEN, FLAG, METHOD)

Decompressing and splitting into 2 channels:

Function: Decompresses a source data stream and divides it into 2 channels: DATA channel and CTRL channel.

DATA channel + CTRL-Kanal

UNPACK_DC$

Parameters:

	B	W	L	S	F	
CTRL$	-	-	-	●	-	Destination string with decompressed CTRL information.
SRC$	-	-	-	●	-	Source string
POS	●	●	●	-	-	Start position in the source string: 0 ... nnnn
LEN	●	●	●	-	-	Maximum length from the source string, 0=till end of string. The conversion is always finished at the string’s end.
FLAG	●	●	●	-	-	Flag code, 1 Byte: 00H ... 0FFH
METHOD	●	●	●	-	-	Selection parameter PACK / UNPACK method: 0 = Pack method “0”, details below.
						Function value:
DATA$	-	-	-	●	-	Destination string with decompressed DATA information

The function is typically used to transmit 2 logic channels via a single channel and with it to make use of a simple compression procedure. This could be all kinds of storage media (internal memory, EEProms, SRams, SmartMedia, ...), as well as any kind of data transmission (RS-232/485, CAN-Bus, Ethernet etc.).

Compressing and multiplex methods

Method 0:

Packing of CTRL information in the data stream:

‹FLag›	BYTE	Flag byte signalises the beginning of a group of CTRL information (bytes).
‹LEN›	BYTE	Number of following CTRL bytes: LEN = 01...7F --› 1...127 CTRL bytes follow. If more CTRL bytes are needed, another ‹Flag›‹Len› ...byte...sequence follows.

Example:
“the quick ‹Flag›‹7›1234567brown fox”

=› DATA$ = “the quick brown fox”
=› CTRL$ = 1234567

Special case (1): “transparency”
LEN = 00 --› 1 x Flag code in the DATA stream or in the CTRL byte stream

Example: ‹Flag› = “A”
“hello world A‹0›“
=› DATA$ = “hello world A”
=› CTRL$ = “ “ ‘ empty

“the quick A‹3›A‹0›BCbrown fox”
=› DATA$ = “the quick brown fox”
=› CTRL$ = “ABC”

Special case (2): "Simple compression" in the DATA channel

3 byte sequence in the data stream, which stated the code of the DATA character to be expanded:

‹Flag› ‹LEN: 80...FF› ‹Code›

LEN = 80 --› 4 x CODE in the DATA stream
LEN = 81 --› 5 x CODE in the DATA stream

LEN = 82 --> 6 x CODE in the DATA stream
.
LEN = FF --> 131 x CODE in the DATA stream

At the beginning the UNPACK_DC$ function's state is:

DATA, no "packed" code

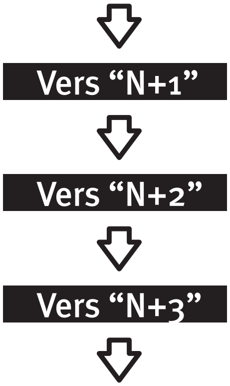
Compare functions: PACK$ and UPACK$

Update_Me

ECODE = UPDATE_ME (Data_Label, Option)

Function: Replaces an existing program with a succeeding version of the program (update) and runs this program.

The update procedure takes place under full control of the application program. The current version of the application program checks the update to the next program version. The next program version checks the update to the next but one version etc.



Parameters:

	B	W	L	S	F	
Data_Label	●	●	●	-	-	States the FLASH address, where a new version of the program exists in the Tiger DATA-FLASH area: 0 ... nnnn
Option	●	●	●	-	-	Always “0”.
						Function value:
ECODE	●	●	●	-	-	Integer value for error code:

Error code:	Meaning:
0	OK. Does not occur, because in this case the new version executes a warm start and does not return to the function value.
1	This Tiger module version does not support this function.
2	RAM size is not correct: The new program version is compiled for another RAM size.

3	FLASH size is not correct: The new program version is compiled for a different FLASH size.
4	The new program version is too big - it does not fit into this module.
5	The new program version does not exist completely in the FLASH or there is a data error in the file.
6	The new program version in the DATA area has been generated with too old compiler version.
7	No such log FLASH ADDR in this module (in this program).
8	RAM size not OK.
9	FLASH sector size is not OK.
10	The new program was not compiled for the project model "PM_FULL" (it has to be).
11 ... 20	File errors: Length or CRC error in the new program version in DATA FLASH.
103	Too high FLASH address found in the program; The program is either too long or it was saved at a too high ADDR in the DATA FLASH.
Other	All other errors are a sign of a defective or incomplete program code. In this case check for correct transmission and saving of the new program.

If none of these errors occurs, the update procedure is set off immediately. The program execution does not return to this function, but the new version is started directly (warm start). Depending on the module’s size this procedure takes some seconds to one minute.

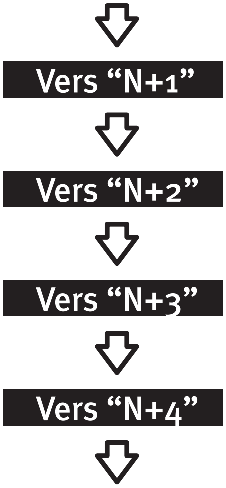
Remote Software Updating

The function Update_Me (...) allows for automatic updating applications under the entire control of the application program. The procedure runs in 3 phases:

1.) Transmitting the succeeding generation of the application program to the Tiger system.

2.) Filing the succeeding generation of the application program in the DATA flash area of the Tiger.

3.) Executing Update_Me (...), the succeeding program generation takes over control.



This remote update allows for flexible customisation to the requirements of the respective application. It is particularly possible to implement typical requirements such as:

- Different transmission channels for updates:
  Broadcast, packet radio, serial, Ethernet, Web, point-to-point, SmartMedia flash cards, IR connection etc.

- Security:
  Authorisation for updating under the full control of the application program. pass word protection, PIN and/or TAN (Transaction-no.) protection, error correcting codes / protocols of communication, ciphering, security codes etc.

- Update control:
  Updates can be provided by the Tiger system long before activating, time controlled updates, automatic updates by activate codes, extension of subscription etc.

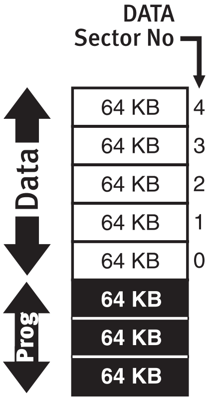
- Fast updates, no extra hardware
  When the next program version is available in the DATA-FLASH area, the actual updating procedure only takes a couple of seconds to ca. 1 minute. During this time the currently running application program is stopped, deleted and replaced by the succeeding program version. Then the new program version takes over executing with a warm start.

Remote Software Updating

Since the actual update procedure is not executed until the complete correct program version is available in the memory, there is only a minimal system down time, during which the application control is stopped. So interferences in communication have no effect on the systems' down time.

How does the update procedure work in the Tiger system?

The following figure shows a schematic illustration of the Tiger FLASH memory with its two areas for the program code and the freely available DATA flash:



Typical FLASH configuration with 512 kBytes FLASH size

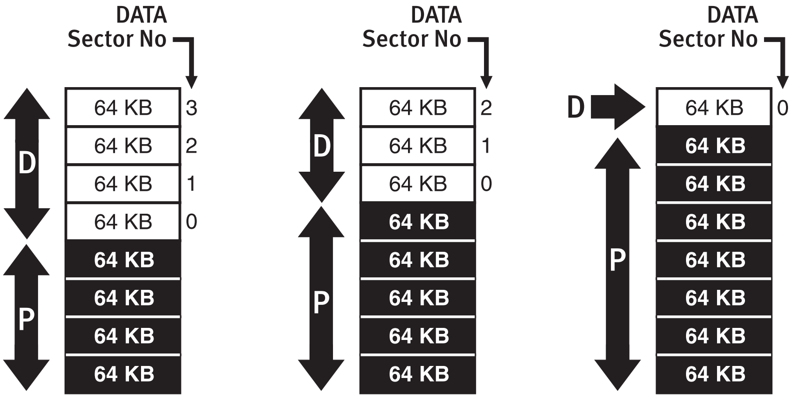
The Tiger program occupies a number of sectors according to its size. Free FLASH sectors are freely available as DATA-FLASH area for the program. This area data can be arbitrarily written and read; whole sectors can be deleted. The Tiger-BASIC program has a number of instructions/functions at its disposal for this (PEEK_FLASH, POKE_FLASH, POKEM_FLASH, ERASE_FLASH, ... see FLASH-functions).

The size of the DATA-FLASH area depends on the program length as well as in the memory size of the BASIC-Tiger computer used. A Tiger-BASIC program knows the start address, the size of the DATA-FLASH area and further values concerning the memory's state from the following system variables:

```
0                                  ' DATA-FLASH start ADDR, fixed, always = 0

SYVARN (SYSVN_FLASH_SSIZE,0)       ' 35, 0: FLASH sector size
SYVARN (SYSVN_FLASH_ASEC, 0)       ' 36, 0:Total number of  FLASH sectors
SYVARN (SYSVN_FLASH_GSIZE , 0)     ' 37, 0: Total size of FLASH memory
```

```
SYVARN (SYSVN_FLASH_DSEC, 0)    ' 38, 0:Number of available DATA sectors
SYVARN (SYSVN_FLASH_DSIZE, 0)   ' 39, 0:Size DATA-FLASH area
```

Please note that the created address space in the system's FLASH memory depends on the different program lengths and module sizes:



(a) (b) (c)

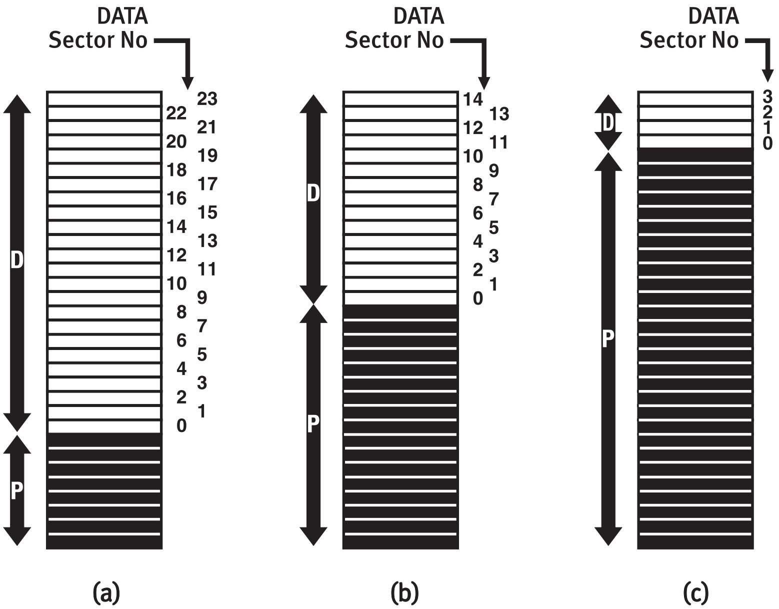
FLASH size:	512 KB	512 KB	512 KB
Sector size:	64 KB	64 KB	64 KB
Base ADDR data:	0	0	0
Last ADDR data:	4000H-1	3000H-1	1000H-1
DATA-FLASH size:	4000H	3000H	1000H

E.g. in (b) the program code occupies 5 sectors of 64 KByte each = 320 KByte. In the FLASH 3 sectors of 64 KByte remain as DATA area = 192 KByte.

Although in 3 cases (a) ... (c) the DATA-FLASH area exists at different places of the memory, the address space always begins with ADDR = 0 and only the available DATA-FLASH size is different.

Accordingly are the proportions with other FLASH sizes (see following page).

FLASH ADDR space for program code and DATA-FLASH area with 2 MByte FLASH memory with 64 KByte sectors:

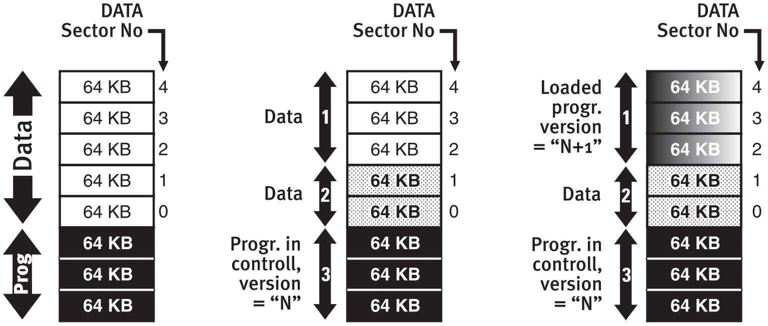


	(a)	(b)	(c)
FLASH size:	2 MB	2 MB	2 MB
Sector size:	64 KB	64 KB	64 KB
Base ADDR data:	0	0	0
Last ADDR data:	18000H-1	F000H-1	4000H-1
DATA-FLASH size:	18000H	F000H	4000H

If an application has sufficient free FLASH memory capacity it is possible to use the DATA-FLASH area as a memory for a new program version, in order to initialise a software update in the BASIC-Tiger afterwards.

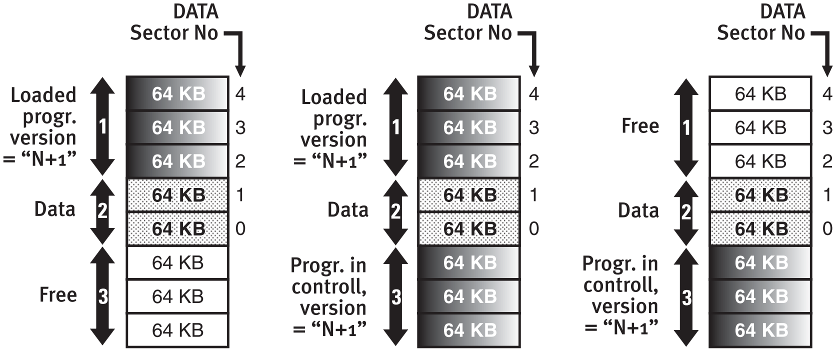
The following figure depicts the phases schematically:

- (a) Memory state at the beginning: Areas for program code and DATA-FLASH.
- (b) Before loading a new program version:
  DATA-FLASH is divided into 2 areas: Data-1 is reserved for the new program version; Data-2 is available for other arbitrary data.
- (c) After loading a new program version ("N+1") to the Data-1 area, data in the FLASH area Data-2 remain unaltered.



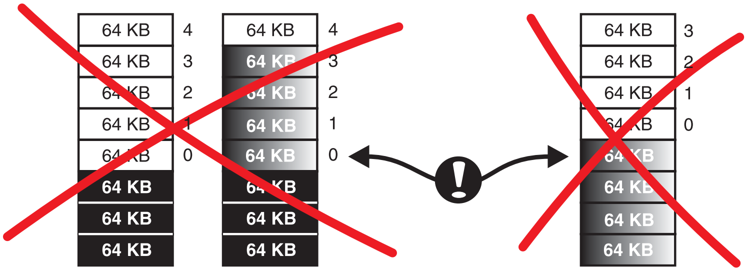
(a) (b) (c)

- (d) After initialising the function "Update_Me (...)" the current program (version "N") is deleted at first.
- (e) Then the Version "N+1" is copied to the FLASH's code area and started.
- (f) Under the control of the new program version "N+1" the upper area of the FLASH (Data-1) is deleted again (by the application program version "N+1") and is available for future updates or other tasks.

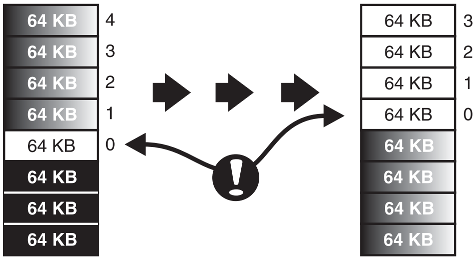


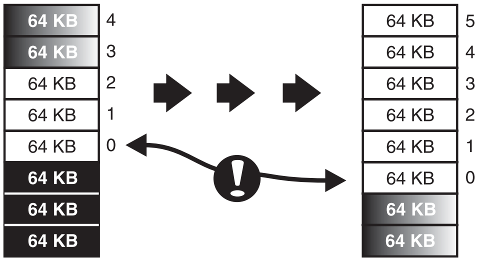
(d) (e) (f)

When determining the FLASH memory sectioning you must note that the memory areas for the new program version must *not* overlap:

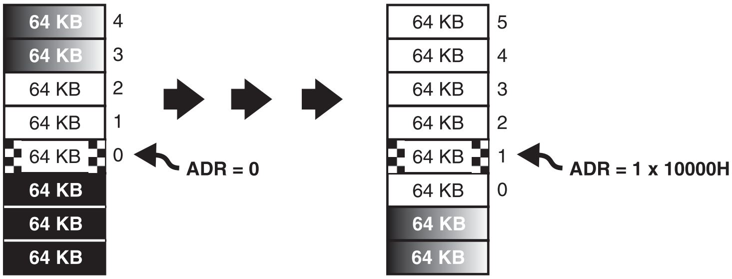


Furthermore size and position of the DATA-FLASH area always changes, if the new program version differs in size from the current program version so that it occupies a different number of FLASH sectors:

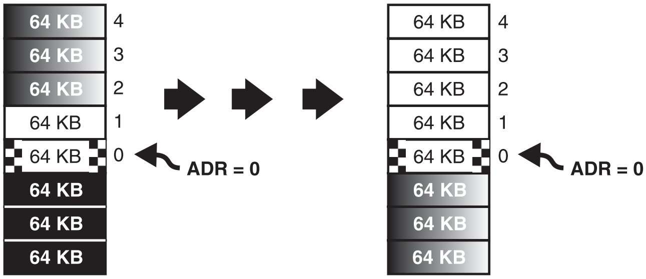




Different program sizes of previous and succeeding program version causes shifted data areas (a):

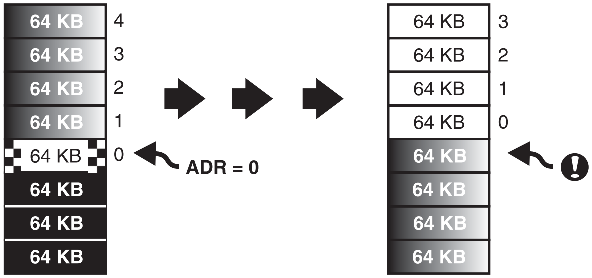


(a)



Different sizes can also lead to data areas being overwritten(b):

(b)



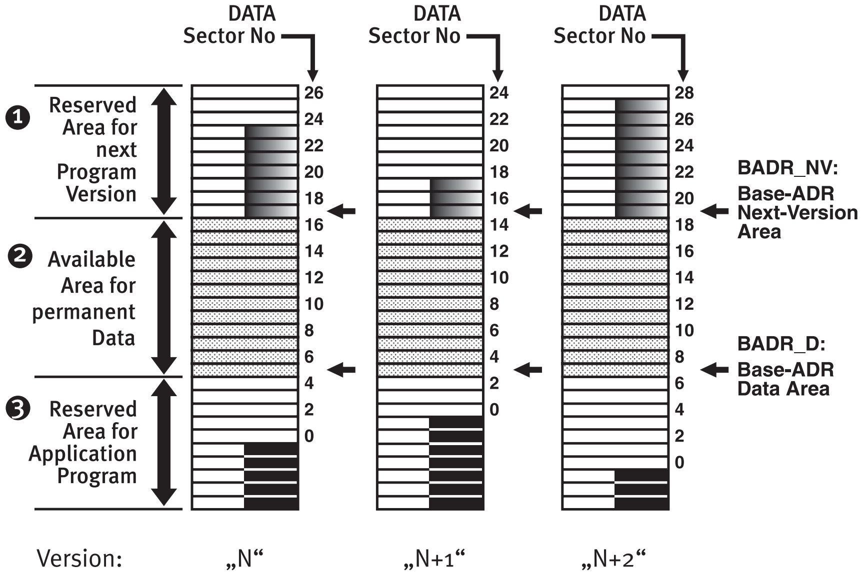
This circumstance can be controlled by the application program with the already mentioned system variables. The FLASH areas can be selectively used for

(1) the DATA-FLASH area for saving the succeeding version of the application program,

(2) the DATA-FLASH area which can be used for permanent data storing,

(3) the currently run program code.

Example for FLASH memory sectioning in 3 successive application program versions:



For this the application program calculates 2 base pointers for each DATA-FLASH area (1) and (2) already during startup phase:

BADR_D = DATA_FLASH_LEN - (FLASH_GESAMT_LEN - PROGRAM_AREA_LEN) ❸

BADR_NV = BADR_D + Available_DATA_Area_Len ❷

Remote Software Updating

The Update_Me function only sets off an update procedure, if a ***complete*** and ***correctly received*** program version of the next generation is available. This has the advantage that an update procedure can never be initialised as long as the new program was not completely transmitted.

An update procedure is not initialised unless a new program version is available in the Tiger memory.

One basic principle is essential for the successful usage of Update_Me:

! The application program's next version must always be able to again load the succeeding version of the application program and to execute an update with it. !

Before a remote application is equipped with a new program version, you should carry out thorough system tests with the new version - especially it has to be made sure that the update sequence is not interrupted.

! Concerning the product design you must consider that a power fail must not occur during update procedure, because this could also interrupt the update sequence. !

According precautions:

- Sufficient battery reserves for at least an Update_Me procedure
- Do not allow switching off during the update procedure.

In case of unattended, widespread applications you can leave out those precautions to simplify the design - e.g. for cost reducing reasons, if

(i) a power-down is very unlikely
(ii) updating frequency is low
(iii) costs for updating manually are maintainable.

Some basic considerations:

(i) The medium frequency of a blackout is assumed to occur e.g. 3 times p.a. for a specific application.

An update procedure is assumed to take e.g. 20 seconds.

Therefore every update which is not protected bears a risk of the blackout occurring during the update procedure of:

$$\text{RISC} = 3 \times 20 / (365 \times 24 \times 3600) = 0{,}000002$$

or in other words:

The probability of the blackout occurring during an update procedure is

1 : 500.000

which is quite unlikely.

If 1000 systems are involved in one application and a program update is to be loaded presumably every 6 months, the probability of 1out of all systems being affected by a blackout during an update procedure in a certain year is:

ca. 1 : 250

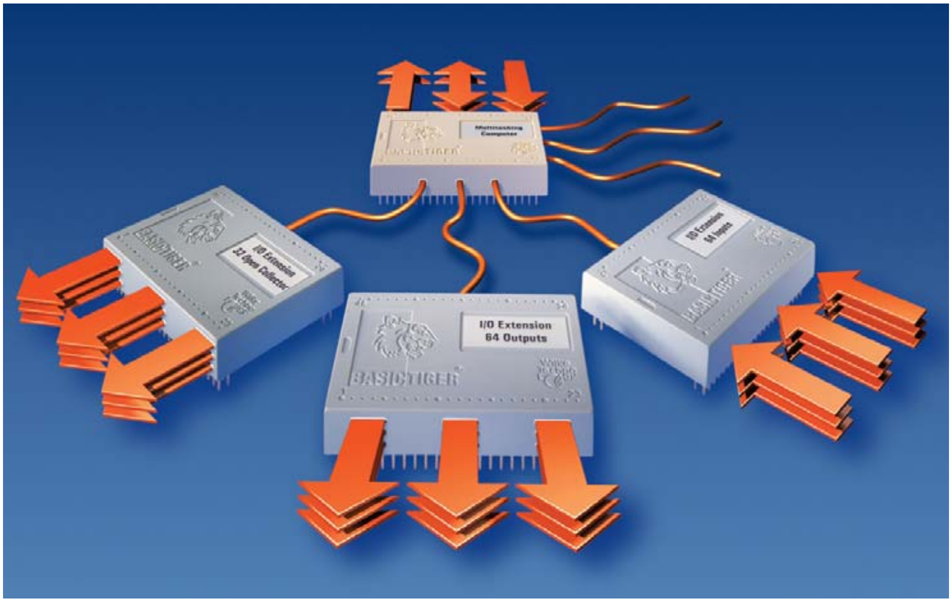
This means that about once in 250 years one out of 1000 systems is affected by such an interference, which requires a manually updating.

This risk would be very much lower than the risk of interrupting the update sequence because of incorrect programming.

However you should avoid an accumulation of risks in such a scenario. This means that the update procedure is carried out gradually for all systems, so that a common power-down can only lead to one malfunction - and does not affect the whole collective.

XPorts

I/O Expansions for BASIC-Tiger



The XPort system allows for expanding the I/O structure in the BASIC-Tiger system.The following Tiger-BASIC functions for the XPort port expansion system are described in this chapter:

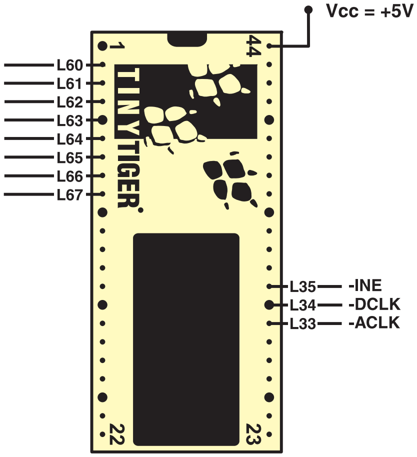
XSetup => defines XPort signals
Xin => reads XPort
Xin$ => reads XPort(s)
XOut => writes to XPort(s)
XSet => sets bit on XPort
XRes => reset bit on XPort
Xinv => invert bit on XPort
XPin => read bit from XPort

For this example circuitries with EP expansion modules and programming examples are provided. Up to 512 I/O ports can be added to a Tiger project with an XPort system, fully supported by software.

XSetup

```
Flag  =   XSetup ( Bus_Port,     &
                   CTRL_Port,    &
                   Bit_ACLK,     &
                   Bit_DCLK,     &
                   Bit_INE,      &
                   CTRL2_Port,   &
                   Bit_Bus_CE  )
```

Function: Defines bus and ctrl signal lines for the XPort system.



XPort signal lines - default setting

Parameters:

	B	W	L	S	F	
Bus_Port	●	●	●	-	-	Port, which is used as an 8-bit DATA-/ADDR bus: Port 6 or port 8
CTRL_Port	●	●	●	-	-	Port, which is used as a 3-Bit CTRL bus for signals: ACLK, DCLK, -INE
Bit_ACLK	●	●	●	-	-	Bit-no: 0...7 for ACLK: ADDR clock
Bit_DCLK	●	●	●	-	-	Bit-no: 0...7 for DCLK: DATA clock

Bit_INE	●	●	●	-	-	Bit-no: 0...7 for -INE: Input-enable (low active)
CTRL2_Port	●	●	●	-	-	Port for CE signal which is used by XBus_OutR / XBus_InR. This signal is ***only*** needed for the functions XBus_OutR and XBus_InR. For applications which do not use these functions the signal should be set to a dummy value, i.e. a non-existent I/O pin, e.g. port 4, bit 7 (BASIC-Tiger, TINY-Tiger or Econo-Tiger).
Bit_Bus_CE	●	●	●	-	-	Bit-No: 0...7 for Bus_CE: BUS access CE signal. This signal is set to ACTIVE by the Tiger (i.e. its former level is INVERTED) during XBUS access. So the other unit knows that a BUS transmission takes place. Further signals, such as DATA direction can be defined and operated by the user in the BASIC program where applicable. **Function value:**
Flag	●	●	●	-	-	0 = OK, parameters accepted 1...5 = No. of the incorrect parameters

Note:

XSetup assigns Tiger I/O pins to the signals of the XPort systems. The pins themselves are not altered, no direction assignment takes place and no value is set.

Therefore the common procedure is to:

1.)	XSetup	assign XPort pins
2.)	Dir_Pin	set Ctrl pins to outputs
3.)	Out	set pins to the defined level.

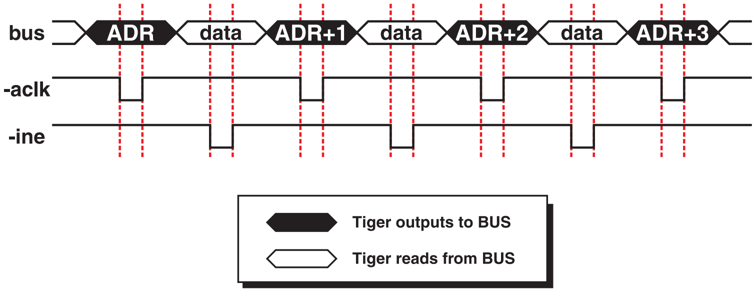
Note:

During an XBUS access, also other device drivers (LCD1, LCD2, parallel IN / OUT etc.) can use this bus. A running XBUS transmission is interrupted by such requests if necessary(XBUS_CE is set to “inactive” during this time).Then the XBUS transmission is continued again.

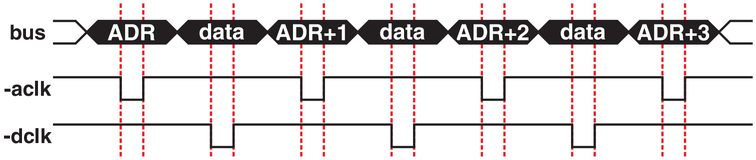
XSetup

Expanded I/O Ports - XPorts

The XPort system extends the Tiger I/O structure to up to 4096 I/O lines. For this an 8-bit bus is used for transmitting address and data bytes as well as 3 ctrl lines for controlling the data stream (-ine, aclk, dclk). The I/O bus accesses for inputs and outputs to and from the XPort system:



Read access to XPorts in ascending ADR order



Write access to XPorts in ascending ADR order

The XPort system interacts directly with the I/O expansion modules of the EP line. Access to a Tiger system's XPort is carried out via only 1 BASIC line. Examples of circuitries and belonging program codes are presented in the description of the functions Xin/Xin$ and XOut.

As long as one does not deviate from the default setting for the ePort system (Bus = L60...67, aclk=L33, dclk=L34, -ine=L35), an XSetup function is not required. XSetup is used if:

1.) another pin assignment is to be set

or

2.) a level system for "aclk" and "dclk" is to be explicitely set. Both signal diagrams presented above show Ctrl signals "aclk" and "dclk" in inverse logic: "-aclk" and "-dclk".

Expanded I/O Ports - XPorts

The following 2 code examples present the explicit determination of the respective ports and pins with the definition of their starting levels.

```
#INCLUDE DEFINE_A.INC              ' general definitions
USER_EPORT ACT, NOACTIVE           ' disable e-Port system

TASK MAIN                          ' Begin task MAIN
  DIR_PORT 3,11000111B             ' set port 3: L33...35 = outputs
  OUT 3, 00111000B, 0FFH           ' set L33 ... L35 to "1" = high

  FLAG = XSETUP (  6,      &       ' 8-bit bus port
                   3,      &       ' 2(3)-bit ctrl port
                   3,      &       ' Bit-no ACLK-signal
                   4,      &       ' Bit-no DCLK-signal
                   5,      &       ' Bit-no -INE signal
                   4,      &       ' Port for Ctrl2 signal "CE"
                   7       )       ' Bit-7 = "CE" => dummy

  A$ = Xin (0, 8)                  ' Read 8 bytes of 8 XPorts ex ADR 0
  XOUT (10H, A$)                   ' Output 8 bytes to XPort 10H...17H
END                                ' End task MAIN
```

This code sequence defines the standard ctrl pins and the bus port to port 6. The ctrl signals are set to “-dclk”, “-aclk” and “-ine”.

The same sequence with a accordingly changed code line “OUT 3...” defines the ctrl signal: “dclk”, “aclk” and “-ine”:

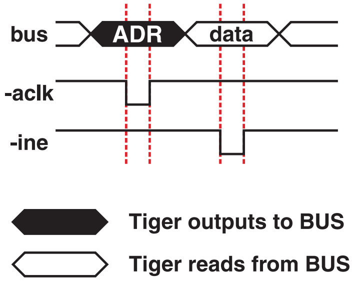
```
  OUT 3, 00111000B, 00100000B   ' set: L33=0, L34=0, L35=1 (-ine)
```

Xin

1 Byte

```
N = Xin ( ADDR )          ' <ADDR> <data>
```

Function: Reads 1 byte from I/O expansion port (XPort) “ADR”.



Xin (ADDR)

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O Expansion port (XPort) Addr: 0 ... 255
						Function value:
N	●	●	●	-	-	1 byte result in numerical variable

Xin works with the XPort system’s default setting respectively with the settings explicitly made with XSETUP (...).

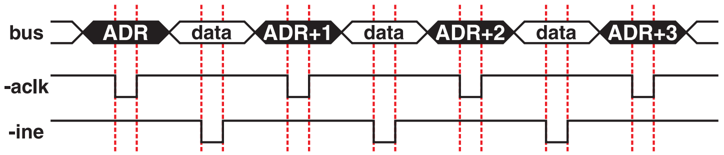
For further XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

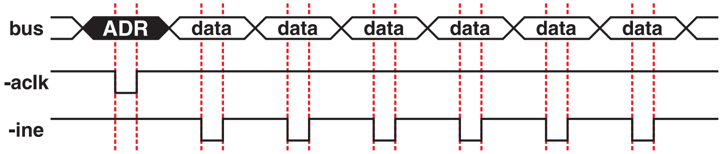
Xin$ n bytes

```
X$ = Xin$ ( ADDR, NO )     ' <ADDR> <data> <ADDR+1> <data> <ADDR+2> <data> ...
X$ = Xin$ ( -ADDR, NO )    ' <ADDR> <data> <data> <data> ...
X$ = Xin$ ( 256, NO )      ' <data> <data> <data> ...
```

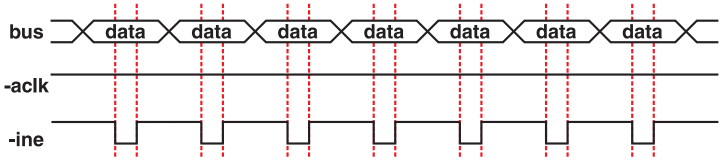
Function: Reads expansion port in different bus access modes.



Xin$ (ADDR, NO)



Xin$ (-ADDR, NO)



Xin$ (256, NO)

⬢ Tiger outputs to BUS ⬡ Tiger reads from BUS

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O Expansion Addr: 0 ... 255, or flag (256)
NO	●	●	●	-	-	Number of bytes to read
						Function value:
X$	-	-	-	●	-	Result string with read data bytes

Xin$ works with the XPort system's default setting respectively with the settings explicitly made with XSETUP (...).

Xin$ is used for reading data from I/O expansion modules and other peripheral devices (memory, logic...). In its original form data are read from the ascending port addresses. This allows promptly scanning a number of input lines with one single BASIC instruction.

For fast parallel data transfer from peripheral units to the Tiger the following forms were added in version 5.2:

```
X$ = Xin$ (-ADDR, NO)          ' <ADDR> <data> <data> <data> ...
X$ = Xin$ (256, NO)            ' <data> <data> <data> ...
```

As appropriate, further ctrl lines are controlled by the BASIC program, as is required for communicating with the peripheral unit (e.g. R/W or -CE signal).

For further XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

Next:

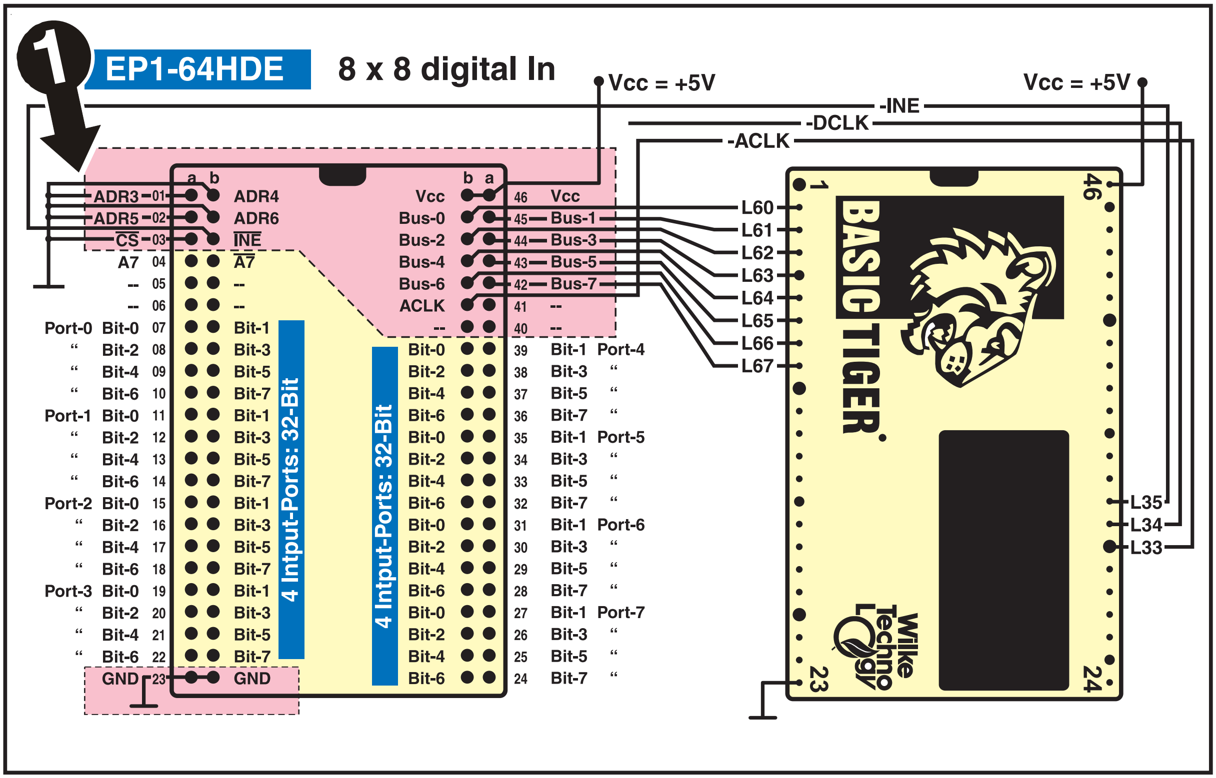
Short examples for the connecting I/O expansion modules and controlling them with a Tiger-BASIC program.

Example 1

An application needs a larger number of digital inputs. Therefore the I/O expansion module EP1-64HDE, which provides 64 digital inputs, is added to the Basic-Tiger.

Since this is the only expansion module in the system, base address 0 is chosen for the expansion module. This takes place by programming the pins:

1a = ADDR-3
1b = ADDR-4
2a = ADDR-5
2b = ADDR-6 See: ❶



Connections of EP1 expansion module and BASIC-Tiger
- Standard signal assignment -

ADDR-7 will not be decoded, so the addresses below 80h 1:1 are mirrored in the upper half of the address space from 80h to 0FFh.

The bottom 3 address bits are used for decoding the 8 ports inside the expansion module:

Addr = 00: Port-0
Addr = 01: Port-1
Addr = 02: Port-2
Addr = 03: Port-3
Addr = 04: Port-4
Addr = 05: Port-5
Addr = 06: Port-6
Addr = 07: Port-7

As mentioned before, those 8 ports are mirrored in the ADDR range 80h ... 87h. I.e. up to 128 ports = 1024 I/O lines can be added to the system without further decoding measures.

Furthermore please note that the control line called “dclk” (data-clock) is NOT used here, since only inputs and no outputs can be carried out with the EP1. So a port write cycle has no influence on the expansion module.

Since the standard signal assignment was used for the XPort system in this case, it is not necessary to execute the XSetup (...) function. Only one function call is required for reading in all 8 XPorts of the expansion module, so that an executable program (which can be compiled) is created with only 3 lines of code:

```
Task MAIN                  ' Begin main task
  A$ = Xin$ (0, 8)         ' Read 8 bytes of 8 XPorts starting from ADDR 0
End                        ' Program end
```

The standard signal assignment of the XPort system is:

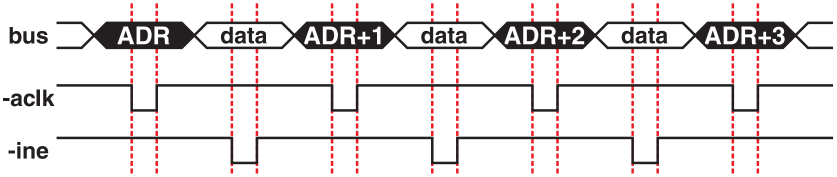
L60 ... L67	8-Bit parallel bus	Data and address transmission
L33	-ACLK	Address clock
L34	-DCLK	Data clock
L35	-INE	Input enable

Depending on the I/O conditions of the respective BASIC-Tiger module these assignments can be changed. Likewise the signals ACLK and INE can be replaced by -ACLK and -INE, if desired. Both types of signals (true and inverse) work with the I/O expansion module; for other expansion circuits another type of signal can be more advantageous.

The following code example shows the explicit determination of the respective ports and pins with their initial level definition.

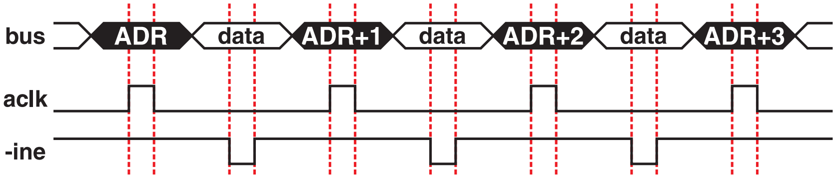
```
#INCLUDE DEFINE_A.INC             ' general definitions
USER_EPORT ACT, NOACTIVE          ' disable e-Port system
TASK MAIN                         ' Beginning task MAIN
  DIR_PORT 3,11000111B            ' set port 3: L33...35 = outputs
  OUT 3, 00111000B, 0FFH          ' set L33 ... L35 to “1” = high
  FLAG = XSETUP ( 6,      &       ' 8-bit bus port
                  3,      &       ' 2(3)-bit ctrl port
                  3,      &       ' Bit-no ACLK-signal
                  4,      &       ' Bit-no DCLK-signal
                  5,      &       ' Bit-no -INE signal
                  4,      &       ' Port forr Ctrl2 signal “CE”
                  7       )       ' Bit-7 = “CE” => dummy

  A$ = Xin$ (0, 8)                ' Read 8 bytes of 8 XPorts start at ADDR 0
END                               ' End task MAIN
```

In this example the same I/O lines were used for bus and control signal as set by default. However we set the three control lines ACLK, DCLK and INE to “high” by the second line in task “OUT 3,...”. So the following signal scheme results:



Control lines -aclk, -in

The control lines ACLK and (DCLK) can be preset to “low” level accordingly, which creates the following signal diagram.

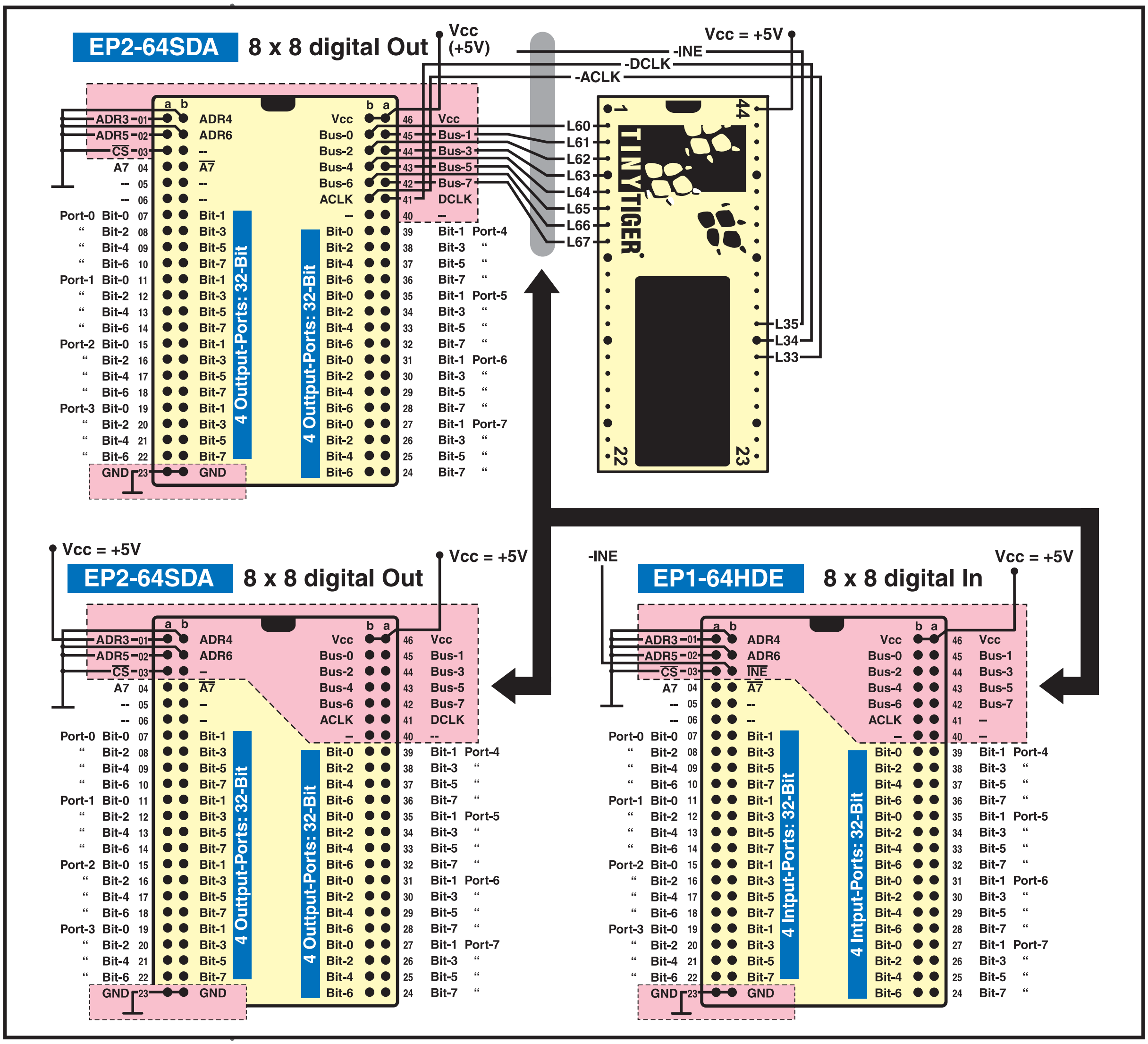


Control lines -aclk, -ine

So the code line belonging to “aclk” and “dclk” in the example given above is:

```
OUT 3, 00111000B, 00100000B ' set L33=0, L34=0, L35=1 (-INE)
```

As a second example the connection of several I/O expansion modules for large I/O structures - here 2 EP2 and 1 EP1 module with 64 digital out/inputs each.



Connection diagram 2 EP2 + 1 EP2 expansion module TINY-Tiger

- Standard signal assignment -

Expansion options:

(a) Like this up to 16 of such input I/O expansion modules can be connected, every module getting an individual base address: 0, 8H, 10H, 18H ... 78H (see (1)). No more additional decoding measures are necessary. In the BASIC program these inputs are addressed as presented above, however with the according larger address range:

```
A$ = Xin$ (0, 8)        ' Read 8 bytes from 8 XPorts, start ADDR 0
A$ = Xin$ (8, 8)        ' Read 8 bytes from 8 XPorts, start ADDR 8
A$ = Xin$ (10H, 8)      ' Read 8 bytes from 8 XPorts, start ADDR 10H
A$ = Xin$ (18H, 8)      ' Read 8 bytes from 8 XPorts, start ADDR 18H
```

Single ports can be addressed likewise

```
A$ = Xin$ (2AH, 1)      ' Read 1 byte from XPort ADDR 2AH
```

as all 128 ports = 1024 input lines (128 input bytes) in a single row:

```
A$ = Xin$ (0, 128)      ' Read 128 bytes from 128 XPorts, start ADDR 0
```

(b) Up to 32 input modules of 8 XPorts each (= 256 XPorts = 2048 inputs) can be inserted with an additional decoding of ADDR-7 into this structure. Suggestions for circuitries can be found in the TEC-Eurocard family on the Internet (www.wilke.de). Addressing the XPort inputs can be accordingly:

From 00 ... to ... FFH

(c) In addition to the XPort inputs also XPort outputs can be added. For XPort outputs the same applies as to inputs. 16 modules = 128 output ports = 1024 outputs with EP2-64SDA expansion modules can be implemented without additional decoding of ADDR-7. These output ports can assign the same address space as the possibly existing input ports. I.e. both the input module EP1 and the output module EP2 can exist at a certain base address. Accessing a mutual XPort address is executed accordingly:

```
XOUT (THIS_ADDR, ....)       ‘ writes starting from XPort address “THIS_ADDR”
A$ = XIN$ (THIS_ADDR, ...)   ‘ reads starting from XPort address “THIS_ADDR”
```

So the number of possibly expanded I/O channels doubles again:

(i) Without address decoding ADDR7:
16 output modules = 128 output ports = 1024 outputs
16 input modules = 128 input ports = 1024 inputs

Total = 2048 I/Os

(ii) WIth address decoding ADDR7:
32 output modules = 256 output ports = 2048 outputs
32 input modules = 256 input ports = 2048 inputs

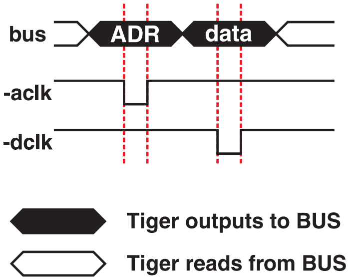
Total = 4096 I/Os

XOut

1 byte

```
XOut ( ADDR, N )          ' <ADDR> <data>  -  write 1 Byte
```

Function: Writes 1 byte to the I/O expansion port (XPort) “ADDR“.



XOut (ADDR, data)

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O expansion port (XPort) addr: 0 ... 255
N	●	●	●	-	-	Low byte is written to XPort

No function value

XOut works with the default settings of the XPort system respectively with the explicit settings in XSETUP (...).

For more XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

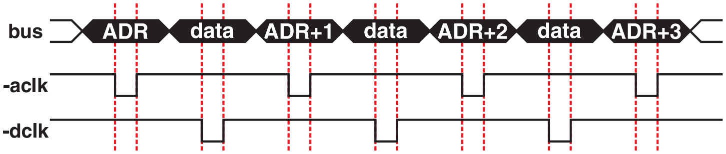
XOut

n bytes

```
XOut ( ADDR, A$ )                  ' write a number of bytes to subsequ. ADDRs
XOut ( ADDR, "" )                  ' write ONLY 1 ADDR cycle, NO data
XOut ( -ADDR, A$ )                 ' write many bytes, 1 ADDR cycle only
XOut ( 256, A$ )                   ' write many bytes, NO ADDR cycle

XOut ( ADDR, Flash, FLen )         ' write many bytes to subsequ. ADDRs
XOut ( ADDR, Flash, 0 )            ' write ONLY 1 ADDR cycle, NO data
XOut ( -ADDR, Flash, FLen )        ' write many bytes, 1 ADDR cycle only
XOut ( 256, Flash, FLen )          ' write many bytes, NO ADDR cycle
```

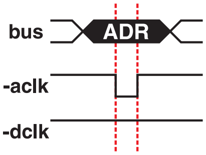
Function: Writes to I/O expansion ports (XPorts) in different modes.

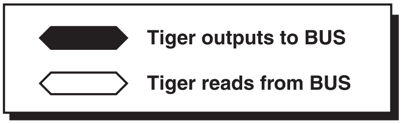
Write to successive XPorts



XOut (ADDR, A$) ● XOut (ADDR, Flash, FLen)

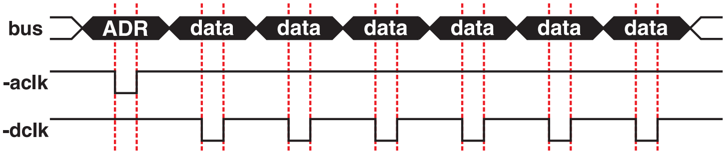
Only write ADDR to the bus once





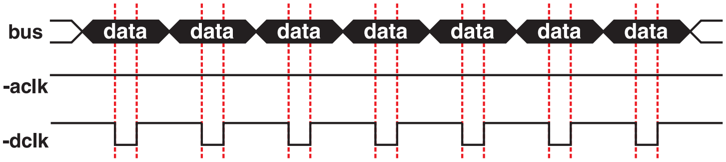
XOut (ADDR, "")

Write ADDR to bus once and then write many data bytes



XOut (-ADDR, A$) ● XOut (-ADDR, Flash, FLen)

Only write many data bytes



XOut (256, A$) ● XOut (256, Flash, FLen)

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O Expansion Addr: 0 ... 255, or flag (256)
A$	-	-	-	●	-	Data string for sending to XPort(s)
Flash	●	●	●	-	-	Flash-ADDR which data bytes lie at for being sent
FLen	●	●	●	-	-	Number of bytes in the flash for being sent to XPort(s)

No function value

XOut works with the XPort system's default setting respectively with the settings explicitly made with XSETUP (...).

For more XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

Next:

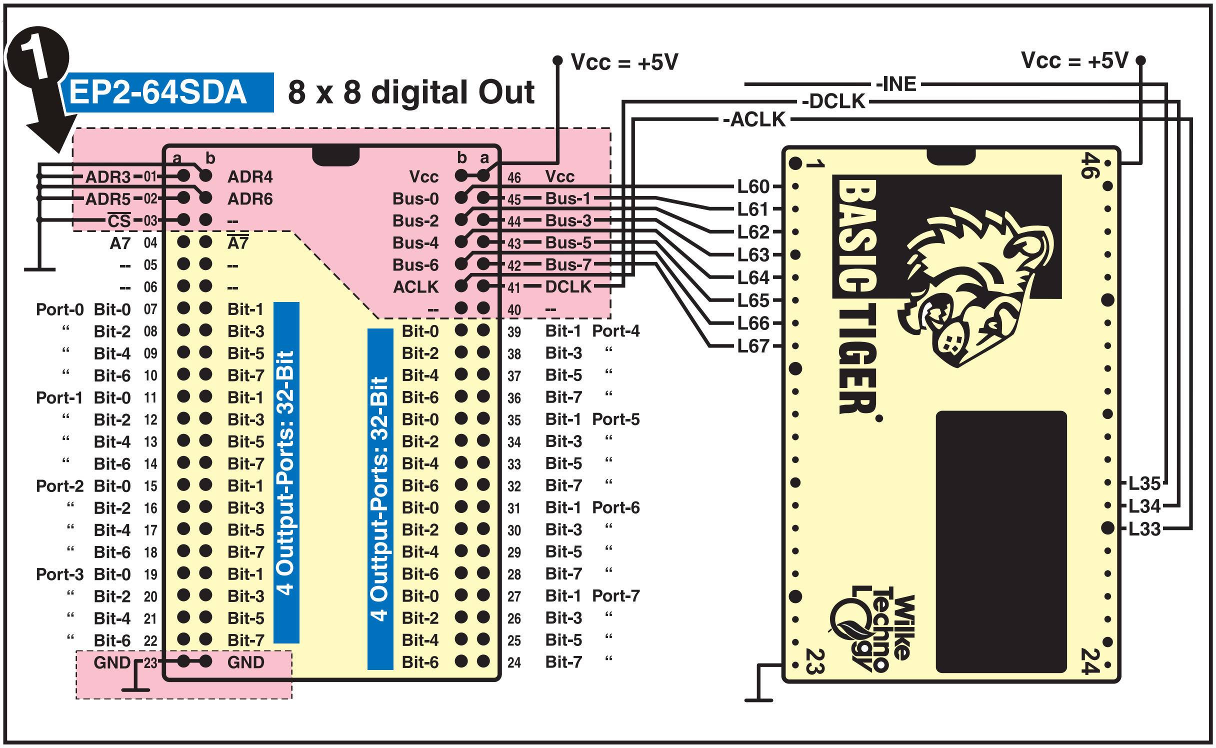
Short examples for connecting I/O expansion modules and controlling them with an application program.

Example 1

An application needs a higher number of digital outputs. So an EP2-64SDA I/O expansion module, which provides 64 digital outputs, is inserted in addition to the BASIC-Tiger.

Since this is the only expansion module in the system, the basic address for the expansion ports = 0 is chosen. This is carried out by pin programming the pins:

1a = ADDR-3
1b = ADDR-4
2a = ADDR-5
2b = ADDR-6 See: ❶



Connection diagram EP2 Expansion module and BASIC-Tiger
- Standard signal assignment -

ADDR-7 is not being decoded, therefore the addresses below 80h 1:1 are mirrored in the upper half of the address space above 80h to 0FFh.

The bottom 3 address bits are inserted into the expansion module for decoding the 8 ports:

Addr = 00: Port-0
Addr = 01: Port-1
Addr = 02: Port-2
Addr = 03: Port-3
Addr = 04: Port-4
Addr = 05: Port-5
Addr = 06: Port-6
Addr = 07: Port-7

As mentioned before, those 8 ports are mirrored in the ADDR range 80h ... 87h. I.e. up to 128 ports = 1024 I/O lines can be added to the system without further decoding measures.

Furthermore please note that the control line called “-INE” (Input enable) is NOT used here, since only outputs and no inputs can be carried out with the EP2. So a port read cycle has no influence on the expansion module.

Since the standard signal assignment was used for the XPort system in this case, it is not necessary to execute the XSetup (...) function. Only one function call is required for setting all 8 XPorts of the expansion module to the wanted signal values, so that an executable program (which can be compiled) is created with only 3 lines of code:

```
Task MAIN                                 ' Beginning main task
  Xout (0, "00 01 03 07 0F 1F 3F 7F"%)   ' write 8 Bytes to 8 XPorts
End                                       ' Program end
```

The standard signal assignment of the XPort system is by default:

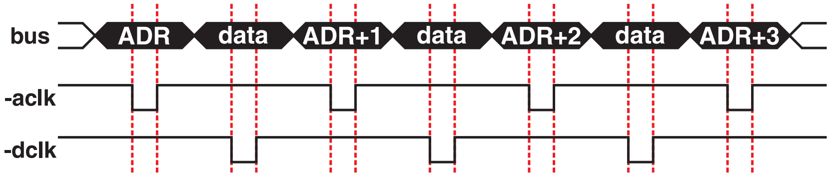
L60 ... L67	8-Bit parallel bus	Data and address transmission
L33	-ACLK	Address clock
L34	-DCLK	Data clock
L35	-INE	Input enable

Depending on the I/O conditions of the respective BASIC-Tiger module these assignments can be changed. Likewise the signals ACLK and DCLK can be replaced by -ACLK and -DCLK, if desired. Both types of signals (true und inverse) work with the I/O expansion modules; for other expansion circuits another type of signal can be more advantageous.

The two following code examples show the explicit determination of the respective ports and pins with the definition of their initial level.

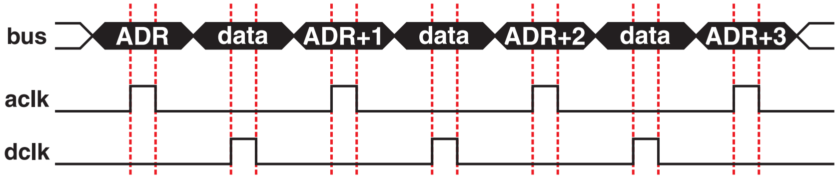
```
#INCLUDE DEFINE_A.INC           ' general definitions
USER_EPORT ACT, NOACTIVE        ' disable e-Port system
TASK MAIN                       ' Beginning Task MAIN
  DIR_PORT 3,11000111B          ' set Port 3: L33...35 = outputs
  OUT 3, 00111000B, 0FFH        ' set L33 ... L35 to "1" = high
  FLAG = XSETUP ( 6,      &     ' 8-Bit bus port
                  3,      &     ' 2(3)-bit ctrl port
                  3,      &     ' Bit-no ACLK-signal
                  4,      &     ' Bit-no DCLK-signal
                  5,      &     ' Bit-no -INE signal
                  4,      &     ' Port for Ctrl2 signal "CE"
                  7       )     ' Bit-7 = "CE" => dummy

  XOUT (0, "FF 00 7F 03 0F F0 55 AA"%) ' Output Bytes to XPort 0...7
END                             ' End task MAIN
```

In this example the same I/O lines, which are defined by default, were used for the bus and the control signals. However, we set the 3 control lines ACLK, DCLK and -INE to “high“ by the second line in task “OUT 3, ...”. So the following signal scheme results:



control lines aclk, -dclk

Accordingly the control lines ACLK and DCLK can be set to ”low”, which generates the following signal scheme:

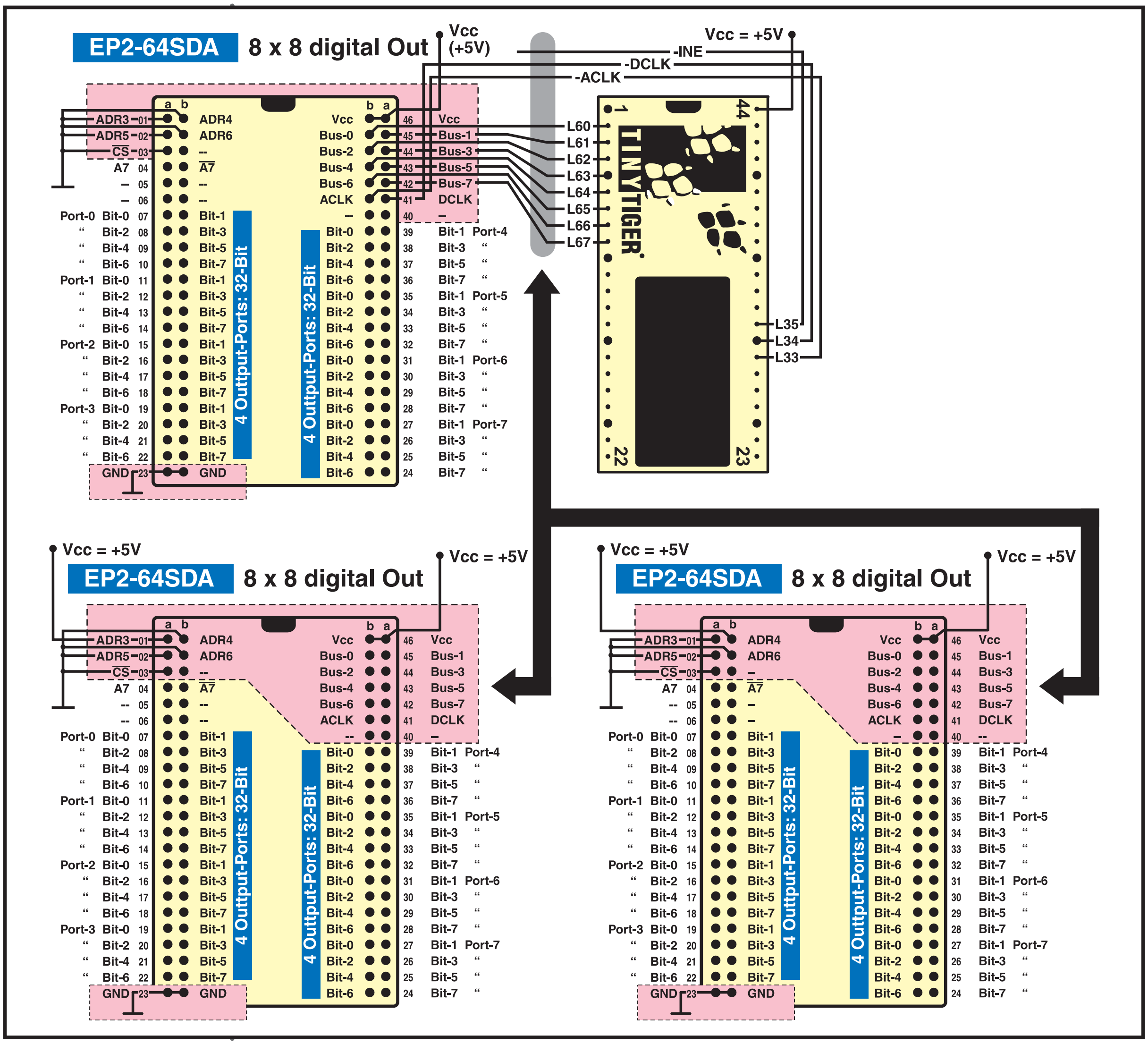


control lines aclk, dclk

The code line belonging to “aclk” and “dclk” (in this example) would be:

```
  OUT 3, 00111000B, 00100000B   ' set L33=0, L34=0, L35=1 (-INE)
```

The connection of several I/O expansion modules for large I/O structures, here 3 EP2 modules with 64 digital outputs each, is shown as a second example:



Connection diagram 3 EP2 expansion modules and TINY-Tiger

- Standard signal assignment -

Expansion options:

(a) Like this up to 16 of such output I/O expansion modules can be connected, every module getting an individual base address: 0, 8H, 10H, 18H ... 78H (see (1)). No more additional decoding measures are necessary. In the BASIC program these inputs are addressed as presented above, however with the according larger address range:

```
XOUT (00H, "00 01 02 03 04 05 FF 07"%) ' Output Bytes to XPort 00...07
XOUT (08H, "08 09 0A 0B EE EE 0E 0F"%) ' Output Bytes to XPort 08...0F
XOUT (10H, "10 11 12 AA AA 15 16 17"%) ' Output Bytes to XPort 10...17
XOUT (18H, "18 BB CC DD EE FF 1E 1F"%) ' Output Bytes to XPort 18...1F
```

Of couse single ports can be addressed likewise

```
XOUT ( 3BH, "AA"%)       ' Output 1 Byte to XPort 3BH
```

as all 128 ports = 1024 output lines (128 output bytes) in a single row:

```
XOUT ( 0, FILL$("A7"%, 128) )  ' Set all 128 XPorts to A7H
```

(b) Up to 32 output modules of 8 XPorts each (= 256 XPorts = 2048 outputs) can be inserted with an additional decoding of ADDR-7 into this structure. Suggestions for circuitries can be found in the TEC-Eurocard family on the Internet (www.wilke.de). Addressing the XPort inputs can be accordingly:

From 00 ... to ... FFH

(c) In addition to the XPort outputs also XPort inputs can be added. Basically for XPort inputs the same applies as to XPort outputs. 16 modules = 128 input ports = 1024 inputs can be implemented with EP1-64HDE expansion modules without additional decoding of ADDR-7. These input ports can assign the same address space as the possibly existing output ports. I.e. both the input module EP1 and the output module EP2 can exist at a certain base address. Accessing a mutual XPort address is executed accordingly:

```
XOUT (THIS_ADDR, ....)       ‘ writes starting from XPort address “THIS_ADDR”
A$ = XIN$ (THIS_ADDR, ...)   ‘ reads starting from XPort address “THIS_ADDR”
```

So the number of possibly expanded I/O channels doubles again:

(i) Without address decoding ADDR7:
16 output modules = 128 output ports = 1024 outputs
16 input modules = 128 input ports = 1024 inputs

Total = 2048 I/Os

(ii) WIth address decoding ADDR7:
32 output modules = 256 output ports = 2048 outputs
32 input modules = 256 input ports = 2048 inputs

Total = 4096 I/Os

XSet, XRes, Xinv

```
XSet ( ADDR, Bit_Pos )      ' set 1 bit in XPort ADDR: 00 ... FF
XRes ( ADDR, Bit_Pos )      ' RESET 1 bit in XPort ADDR: 00 ... FF
Xinv ( ADDR, Bit_Pos )      ' Toggle 1 bit in XPort ADDR: 00 ... FF
```

Function: Manipulates a bit of an XPort output:

- Set bit
- Reset bit
- Invert bit

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O expansion port (XPort) Addr: 0 ... 255
Bit_Pos	●	●	●	-	-	Bit-position: 0 ... 7

No function value

Program example:

```
TASK MAIN                  ' Beginning task MAIN
  XSET (2BH, 2)            ' Set Bit-2 in XPort 2BH
  XRES (2BH, 2)            ' REST Bit-2 in XPort 2BH
  XINV (2BH, 2)            ' Invert Bit-2 in XPort 2BH
END                        ' Program end
```

In this example the default settings of the e-port system are used: Bus = L6, ackl= L33, dclk = L34, -ine = L35. Other settings -> see function “XSetup”.

For more XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

XPin

A = XPin (ADDR, Bit_Pos) **' lies 1 Bit in XPort ADDR: 00 ... FF**

Function: Reads a single port of an XPort input.

Parameters:

	B	W	L	S	F	
ADDR	●	●	●	-	-	I/O Expansion port (XPort) Addr: 0 ... 255
Bit_Pos	●	●	●	-	-	Bit position: 0 ... 7
						Function value:
A	●	●	●	-	-	Bit read from XPort, value: 0, 1

Program example:

```
TASK MAIN                                ' Beginning task MAIN
  IF  XPIN (6AH, 7) = 1  THEN            ' Teste Bit-7 in XPort 6AH = 1  ??
'    ...                                 ' do this if Bit = “1”
  ELSE                                   '
'    ...                                 ' do this alternatively, as Bit = “0”
  ENDIF                                  '
END                                      ' Program end
```

In this example the default settings of the e-port system are used: Bus = L6, ackl= L33, dclk = L34, -ine = L35. Other settings -> see function “XSetup“.

For more XPort I/O functions also see:

XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

Device Drivers

empty page

PS2 (PS2 Input device emulation)

File name: PS2_Pp.TDD

```
INSTALL_DEVICE #D, “PS2_Pp.TDD“
```

PS2 input device emulation

Function: This device driver allows for emulating a PS2 input device, such as a PC keyboard, with a BASIC-Tiger.

Parameters:

	B	W	L	S	F	
D	●	●	●	-	-	The driver device number
Pp	-	-	-	-	-	in the file name stands for: P: internal clock line port p: clock pin

The driver uses two output lines: Clock and data. The following pin combinations are feasible by choosing the suitable driver file:

Driver name	Clock pin	Data pin
PS2_33.TDD	L33	L34
PS2_34.TDD	L34	L35
PS2_35.TDD	L35	L70
PS2_40.TDD	L40	L42
PS2_42.TDD	L42	L33
PS2_60.TDD	L60	L61
PS2_61.TDD	L61	L62
PS2_62.TDD	L62	L63
PS2_63.TDD	L63	L64
PS2_64.TDD	L64	L65
PS2_65.TDD	L65	L66
PS2_66.TDD	L66	L67
PS2_67.TDD	L67	L40
PS2_70.TDD	L70	L71
PS2_71.TDD	L71	L72
PS2_72.TDD	L72	L73
PS2_73.TDD	L73	L60
PS2_80.TDD	L80	L81
PS2_81.TDD	L81	L82
PS2_82.TDD	L82	L83
PS2_83.TDD	L83	L84
PS2_84.TDD	L84	L85
PS2_85.TDD	L85	L86
PS2_86.TDD	L86	L87
PS2_87.TDD	L87	L70

PS2 (PS2 Input Device Emulation)

The driver requires the time base driver “TimerA.tdd“ for providing the clock. The time base driver should be set to a frequency of ca. 10 to 15 kHz e.g. for emulating a PC keyboard.

The PS2 driver works similar to a serial interface for transmitting data to the destination/PC and back. The application resembles the “ Ser1b.tdd“ driver, but has the following special features:

- 2 x 256 bytes buffer for sending and receiving data.
- Secondary address =0 (1 channel, bidirectional).
- User function codes for testing and deleting buffers.

PS2_Pp.TDD user function codes

User function code for input (instruction GET):

No	Symbol	Description
1	UFCI_IBU_FILL	Input buffer fill level (bytes)
2	UFCI_IBU_FREE	Free space in input buffer (bytes)
3	UFCI_IBU_VOL	Input buffer size (bytes)
33	UFCI_OBU_FILL	Output buffer fill level (bytes)
34	UFCI_OBU_FREE	Output buffer free space (bytes)
35	UFCI_OBU_VOL	Output buffer size (bytes)
65	UFCI_LAST_ERRC	Last error code
99	UFCI_DEV_VERS	Driver version

User function codes for output (instruction PUT):

Nr	Symbol	Description
1	UFCO_IBU_ERASE	Delete input buffer
33	UFCO_OBU_ERASE	Delete output buffer

Outputting characters

Output is preferably carried out with PUT, which outputs characters as they are, i.e. without converting them to ASCII (in case of numbers) and without adding CR and LF.

If there is not enough space in the output buffer and a output is done nevertheless, the instruction PUT or PRINT (and therefore the whole task) waits until there is space in the buffer again. This waiting can be avoided by querying the free buffer space before outputting. Example for the output of 4 bytes as hex code, only if there is sufficient output buffer space:

```
GET #PS2, #0, #UFCI_OBU_FREE, 0, OUTFREE    ' ask for buffer space
IF OUTFREE > 3 THEN                         ' if at least 4 bytes are free
  PUT #PS2, "12 1C 9C AA"%                  ' output characters
ENDIF
```

Reading characters

Received characters are filed in the input buffer by the device driver. The characters are preferably read with GET.

GET does not wait and only reads, if there is something in the buffer. If a numerical variable is read, it contains the value 0 (zero), if

- nothing was in the buffer
- a 0 (zero) was in the buffer

In order to differentiate those cases, it should be inquired, if there are characters residing in the input buffer before trying to read.

Example for checking, whether there is at least one character in the input buffer:

```
GET #PS2, #0, #UFCI_IBU_FILL, 0, INFIL      ' read buffer fill level
IF INFIL > 0 THEN                           ' if something is in the buffer
  GET #PS2, 1, B                            ' read one byte
ENDIF
```

There is a number of PS/2 input devices, such as keyboards, mouses, barcode scanners etc. These devices communicate bidirectional with the host, i.e. they receive commands and answer respectively receipt them and send data to the host. These command records and data can be different from device to device and from host to host. So if you would like to simulate a keyboard, please catch up on which codes have to be generated and which commands can be expected from the host.

"PS2_Keyboard_Emulation_Demo_001.TIG" is an example program commented in detail for emulating a PC keyboard with a BASIC-Tiger. The program can be found in the directory **DeviceDriver_Examples** of your Tiger-BASIC installation.

empty page

PWMX (Pulse Width Modulation)

File name: PWMX_Pp.TDD

INSTALL_DEVICE #D, “PWMX_Pp.TDD“, range, factor

Pulse width modulation at an adjustable carrier frequency

Function: This device driver allows the output of pulse width modulated signals via an arbitrary I/O pin at a constant adjustable carrier frequency.

Parameters:

	B	W	L	S	F	
D	●	●	●	-	-	The driver device number
Pp	-	-	-	-	-	in the file name stands for: P: PWM signal line internal port p: PWM signal line pin number
Range	●	●	●	-	-	Definition of the range or basic clock pulse
Factor	●	●	●	-	-	Factor by which the basic clock pulse is to be devided

The fundamental frequency to be set is calculated as follows:
desired carrier frequency * 256

Example: A PWM signal carrier frequency of 250Hz requires a fundamental frequency (basic clock pulse) of 250 * 256 = 64.000 Hz. So the factor for clock pulse range 1 (basic clock pulse 2.5 MHz) has to be set to 39.0625 (rounded off to 39).

Notes:

- The smallest possibly adjustable fundamental frequency is 610 Hz, which corresponds to a carrier frequency for the PWM signal of 2.383 Hz.
- All settings between 1 and 610 Hz are interpreted as 610 Hz.
- All settings being 0 or negative values are interpreted as 0 Hz (no output)
- **Important:** Both TIMERA.TDD and PWMX.TDD use very high internal frequencies. Therefore onlyone of these drivers can be used inside an application.

PWM output

An output is initialised by the following instruction:

PUT #D, Duty

	B	W	L	S	F	
D	●	●	●	-	-	Driver device number
Duty	●	●	●	-	-	Duty has 257 possible values, from 0 to 256. 0 = no duty (signal permanently low) 256 = 100% duty (signal permanently high) 1...255 = adjustable duty / cycle-ratio, e.g. 128 = 50% duty or 64 = 25% duty

The PWM signal set last is outputted until a new setting is carried out.

The fundamental frequency can be changed at any point during program flow:

PUT #D, #1, fundamental frequency

	B	W	L	S	F	
D	●	●	●	-	-	Driver device number
Fundamental frequency	●	●	●	-	-	The restrictions stated under driver installation apply for setting the fundamental frequency. If the desired fundamental frequency cannot be set exactly, the best possible approximation is used.

Example: The desired carrier frequency is 20 Hz, so the required fundamental frequency is 5120 Hz. The best possible approximation is range 1 with factor 122 - this corresponds to 5122 Hz. This is the fundamental frequency used.

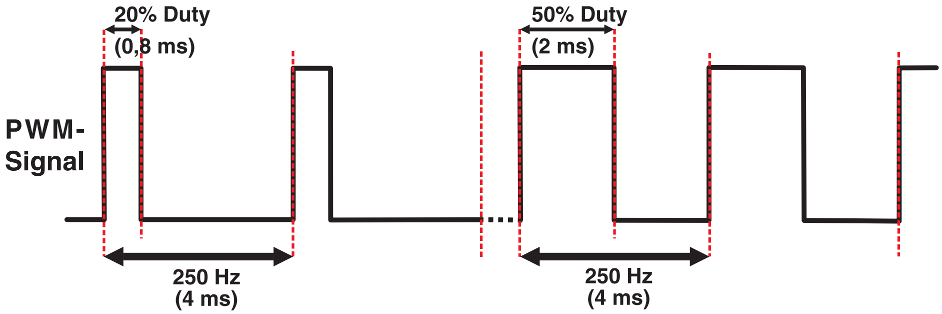
PWMX (Pulse Width Modulation)

Example:

Outputting a PMW signal to pin L80 at a 250 Hz carrier frequency and a duty of 20% at first, 50% after 5 seconds, stop after another 5 seconds.

```
USER_VAR_STRICT                          ' check variable declaration

TASK MAIN                                ' Start Task MAIN

LONG N                                   ' declare variables

INSTALL_DEVICE #2, "PWMX_80.TDD",1,39    ' Fundamental frequency 64.1 kHz -->
                                         ' carrier frequency. 250.4 Hz

N = 51                                   ' duty ca. 20%
PUT #2, N                                ' output PWM signal
WAIT_DURATION 5000                       ' wait for 5 seconds
N = 128                                  ' duty 50%
PUT #2, N                                ' output PWM signal
WAIT_DURATION 5000                       ' wait for 5 seconds
N = 0                                    ' duty 0% (constantly low)
PUT #2, N                                ' output PWM signal

END                                      ' End of task MAIN
```

This program generates approximately the following output (for simpler visualisation the values are rounded):



empty page

SER_XUART (Serial I/O with external UART)

File name: SER_XUART.TDD

INSTALL_DEVICE #D, “SER_XUART.TDD“, address, type, number

Serial communication via external UARTcomponent

Function: This device driver allows serial input and output via external UART components, e.g. the expansion module EP33-4SER with 4 additional serial interfaces.

Parameters:

	B	W	L	S	F	
D	●	●	●	-	-	Driver device number
Address	●	●	●	-	-	Base address of the first UART component
Type	●	●	●	-	-	Type of the modules connected (0 = EP33)
Number	●	●	●	-	-	Number of modules connected

Every EP-33 module assigns two expanded ports, e.g. 10h and 11h or 28h and 29h. In case of an address space of 256 possible addresses maximum 128 modules respectively 512 serial interfaces can be controlled.

If several modules are used, the modules’ addresses should be aligned to each other. In case of 3 modules with 12 interfaces in total this would be e.g.:

1. Module:	Addresses 10h + 11h
2. Module:	Addresses 12h + 13h
3. Module:	Addresses 14h + 15h

So the driver installation would look as follows:

```
INSTALL_DEVICE #33, "SER_XUART.TDD", 10h, 0, 3
```

The numbers of the single channels are simply counted up, starting from 0. In the example presented above the single channels are thus addressed via the secondary addresses 0 to 11.

The symbols for the different user function codes and their parameters are summarised in files DEFINE_XUART.INC and UFUNC_XUART.INC. Those should be integrated into the source code.

SER_XUART.TDD user function codes

User function code for queries (instruction PUT):

No.	Symbol	Description
1	UFCO_TRANSMIT_DATA_DACC	Data transmission “send”
2	UFCO_RECEIVE_DATA_DACC	Data transmission “receive”
3	UFCO_START_STOP_TRANSMIT	“Send” data 0 = stop <>0 = start
4	UFCO_START_STOP_RECEIVE	“Receive” data 0 = stop <>0 = start
5	UFCO_START_STOP_REC_TRA	“Send” and “receive” data 0 = stop <>0 = start
6	UFCO_ABORT_TRANSMIT	Abort ”sending” immediately
7	UFCO_ABORT_RECEIVE	Abort “receiving” immediately
8	UFCO_ABORT_REC_TRA	Abort “sending” and “receiving” immediately
9	UFCO_RESET_MODULE	Reset module (4 channels)
10	UFCO_RESET_ALL_MODULES	Reset all modules
11	UFCO_SET_BAUD_PARAM	Set interface parameters (compatible with comSER1B)
12	UFCO_SET_BAUD_PARAMX	Set interface parameters (XUART)

Setting interface parameters

Both parameters for a serial channel (baud rate and bit/parity) can be set with two user function codes. In the first case the same parameter values as in the case of driver SER1B are used, in the second case special values for the driver SER_XUART are used. An example for setting channels 2 and 3 to 38,400 bauds, 8 data bits, 1 stop bit and even parity is given below; once with values compatible with SER1B (channel 2) and once with XUART values (channel 3):

```
PUT #33, #2, #UFCO_SET_BAUD_PARAM, BD_38_400, DP_8E  ' with SER1B-parameters
PUT #33, #3, #UFCO_SET_BAUD_PARAMX, BR_38400,&       ' with XUART-parameters
  WORD_LENGTH_8 + STOP_BIT_1 + EVEN_PARITY
```

Available settings with XUART parameter values for baud rate:

No (dec.)	No (hex)	Symbol	Meaning
18432	4800h	BR_50	50 bauds
12288	3000h	BR_75	75 bauds
9216	2400h	BR_100	100 bauds
8378	20BAh	BR_110	110 bauds
6144	1800h	BR_150	150 bauds
4608	1200h	BR_200	200 bauds
3072	0C00h	BR_300	300 bauds
2304	0900h	BR_400	400 bauds
1536	0600h	BR_600	600 bauds
1024	0400h	BR_900	900 bauds
768	0300h	BR_1200	1.200 bauds
512	0200h	BR_1800	1.800 bauds
384	0180h	BR_2400	2.400 bauds
256	0100h	BR_3600	3.600 bauds
192	00C0h	BR_4800	4.800 bauds
128	0080h	BR_7200	7.200 bauds
96	0060h	BR_9600	9.600 bauds
64	0040h	BR_14400	14.400 bauds
48	0030h	BR_19200	19.200 bauds
32	0020h	BR_28800	28.800 bauds
29	001Dh	BR_31250	31.250 bauds
24	0018h	BR_38400	38.400 bauds
16	0010h	BR_57600	57.600 bauds (default)
15	000Fh	BR_62500	62.500 bauds
12	000Ch	BR_76800	76.800 bauds
8	0008h	BR_115200	115.200 bauds
6	0006h	BR_153600	153.600 bauds
4	0004h	BR_230400	230.400 bauds
3	0003h	BR_307200	307.200 bauds
2	0002h	BR_460800	460.800 bauds
1	0001h	BR_921600	921.600 bauds

Available settings with XUART parameter values for bits & parity:

No (dec.)	No (hex)	Symbol	Meaning
0	00h	WORD_LENGTH_5	5 data bits
1	01h	WORD_LENGTH_6	6 data bits
2	02h	WORD_LENGTH_7	7 data bits
3	03h	WORD_LENGTH_8	8 data bits
0	00h	STOP_BIT_1	1 stop bit
4	04h	STOP_BIT_1_2	1½ stop bits
4	04h	STOP_BIT_2	2 stop bits
0	00h	NO_PARITY	no parity
8	08h	ODD_PARITY	odd parity
24	18h	EVEN_PARITY	even parity
40	28h	MARK_PARITY	mark
56	38h	SPACE_PARITY	space

For a valid setting a number of data bits, a number of stop bits and one parity are combined, the according values are added. So for 7 data bits, 1 stop bit and even parity you can either take the value 26 as a parameter or add the single symbols (WORD_LENGTH_7 + STOP_BIT_1 + EVEN_PARITY).

Outputting characters

Output takes place by transferring the string to be transmitted to the driver by the user function code UFCO_TRANSMIT_DATA_DACC.

PUT #D, #C, #UFCO_TRANSMIT_DATA_DACC, Text$ [, Offset, number]

The optional parameters of this function allow outputting starting from a certain position in the string (**Offset**) as well as a certain **number** of characters (0 = until the end of the string).

Example for transmitting a string via channel 0:

```
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT$        ' Output via channel 0
```

In addition the driver provides a reload function, i.e. during transmission of the current string, the next string can already be transferred to the driver. This makes for continuous transmission.

Example for the uninterruptible transmission of two strings via channel 0 + reload:

```
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$       ' Output via channel 0
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$       ' Reload via channel 0
```

By querying the string length with the function LEN it can be found out, how many characters of one string still have to be sent. This value is constantly updated. Like this the next output of a string can be held back until the previous output is completed.

Example for ‘waiting’ for the completion of string transmission:

```
FOR EVER = 0 TO 0 STEP 0                         ' infinite loop
  TEXT$ = STR$(TICKS())                          ' create transm. string
  PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT$   ' output via channel 0
  WHILE LEN(TEXT$) > 0                           ' Wait here until string
  ENDWHILE                                       ' is sent completely
NEXT                                             '
```

Of course “waiting” can be combined with the reload function. While one string is being sent, the second one is free (again); it can be assigned once more and transferred to the driver.

Example: Two strings are written alternately, as soon as one string is free again:

```
TEXT1$ = "1:" + STR$(TICKS())                    ' Create 1. string
TEXT2$ = "2:" + STR$(TICKS())                    ' Create 2. string
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$    ' Output 1. string
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$    ' Output 2. string
FOR EVER = 0 TO 0 STEP 0                         ' Infinite loop
  IF LEN(TEXT1$) = 0 THEN                        ' If 1. string is sent
    TEXT1$ = "1:" + STR$(TICKS())                ' new 1. string
    PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$' new output 1. string
  ENDIF                                          '
  IF LEN(TEXT2$) = 0 THEN                        ' If 2.string is sent
    TEXT2$ = "2:" + STR$(TICKS())                ' new 2. string
    PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$' new output 2.string
  ENDIF                                          '
NEXT                                             '
```

Reading characters

Received characters are saved by the device driver in the string transferred by the user function code UFCO_RECEIVE_DATA_DACC.

PUT #D, #C, #UFCO_TRANSMIT_DATA_DACC, Text$ [, offset, number]

The optional function parameters allow saving the received characters from a specific string position (**Offset**) and only saving a certain **number** of characters (0 = to the string's end). If no parameter is given, the complete string is filled.

Example for reading into a string from channel 0:

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text$        'reading from channel 0
```

If a new string is transferred to the device driver by this function, all received characters are immediately saved in this new string, even if the previous string is not filled completely.

Besides the driver provides a reload function for reception, i.e. while receiving in the current string, the next string can already by passed to the driver. For using the reload function the parameters **offset** und **number** have to be given. In this case the first string is filled from position **offset** with **number** of characters, before the second string is filled. So the receive strings are not switched at once.

Example: Two strings with the length 100 are used for reception. Not before the string text1$ is full, the received characters are saved in text2$.

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text1$, 0, 100'reading from channel 0
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text2$, 0, 100'reading from channel 0
```

Also in this case the current length of the string can be determined by the function LEN. Like this you can find out, how many characters are already saved in the string and if therefore a new receive string has to be passed to the device driver, in order to ensure an uninterruptible reception.

Example: Two strings of the length 500 are used for receive. Shortly before one string is full, the other one is used for receiving and the first one is e.g. attached to a global buffer string.

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text1$     ' 1.reception string
FOR EVER = 0 TO 0 STEP 0                         ' infinite loop
  IF LEN(Text1$) > 450 THEN                      ' If string is almost full
    PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text2$ ' 2.reception string
    Buffer$ = Buffer$ + Text1$                   ' 1.string in buffer
  ENDIF                                          '
  IF LEN(Text2$) > 450 THEN                      ' If string is almost full
    PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, TEXT1$ ' 1.reception string
    Buffer$ = Buffer$ + Text2$                   ' 2.string in buffer
  ENDIF                                          '
NEXT                                             '
```

Reset module

There are two user function codes for resetting modules. Either one module or all modules existing in the system can be reset. The secondary address determines which module is reset.

Examples:

```
PUT #33, #0, #UFCO_RESET_MODULE, 0      ' Channel0: Reset 1.module(Ch 0-3)
PUT #33, #3, #UFCO_RESET_MODULE, 0      ' Channel3: Reset 1.module(Ch 0-3)
PUT #33, #4, #UFCO_RESET_MODULE, 0      ' Channel4: Reset 2.module(Ch 4-7)
PUT #33, #0, #UFCO_RESET_ALL_MODULES, 0  ' any channel: Reset al modules
```

empty page

SHI (clocked, serial input)

Filename: SHI_PpPp.TDD

```
INSTALL_DEVICE #D, "SHI_PpPp.TDD", Timeout, Rec_Edge,          &
                        Bit_Seq, Byte_Seq, Bits_per_Byte
```

clocked, serial input

Function: This device driver allows for a clocked, serial input from an external chip. The first two numbers of the driver's name specify port and pin of the clock line; the last two numbers specify port and pin of the data line.

Parameters:

	B	W	L	S	F	
D	●	●	●	-	-	Driver device number
Timeout	●	●	●	-	-	Timeout in ms (2...255) for detecting a data packet
Rec_Edge	●	●	●	-	-	Determines, at which clock edge the data bits are read: 0=rising edge, X=falling edge
Bit_Seq	●	●	●	-	-	Sequence of the received bits: 0=low-bit-first, X=high-bit-first
Byte_Seq	●	●	●	-	-	Sequence of the received bytes: 0=low-bit-first, X=high-bit-first
Bits_per_Byte	●	●	●	-	-	Number of received bits for each byte (1...8)

The device driver set “SHI_PpPp.TDD“ is used for reading from clocked, serial data streams to the BASIC-Tiger. In contrast to byte-oriented serial data, in which case bytes are read from a serial data stream, this driver functions like a serial shift register input, which reads data sequential bit-by-bit.

So no explicit packet delimiters such as start and stop bits are expected. However, they can be received and deleted in the program later.

The driver requires two input lines: Clock and data.

The clock line can use both inverted and not inverted pulses for the clock. Besides, the driver can be set that data lines are read either at falling or rising edges.

With every clock one bit is read from the data line and written to the internal buffer according to the following settings:

SHI (Clocked, Serial Input)

- The first bit is a MSB or a LSB.
- The group of bits received first (e.g. one byte) is the first or the last one in the buffer.
- Received bits can be transferred to the buffer at a rate of 1 bit per byte to 8 bits per byte.
  Received formats of more than 8 bits in each group of bits (e.g. 16 data bits + 1 start bit + 1 stop bit = 18 bits) are processed in an 8-bits-per-byte scheme by the device driver and later reconverted by a converting function in BASIC.

Furthermore no “strobe“ and “start“ signals are required to initialise data stream reception. The driver detects the beginning and the end of a data stream by using the timeout state.

If the clock signal pauses at least for the length of the timeout period, the driver treats all data bits received up to this point as a complete set of received bits. Then the BASIC program can read them with a “GET“ instruction. After “GET“ the data buffer is cleared.

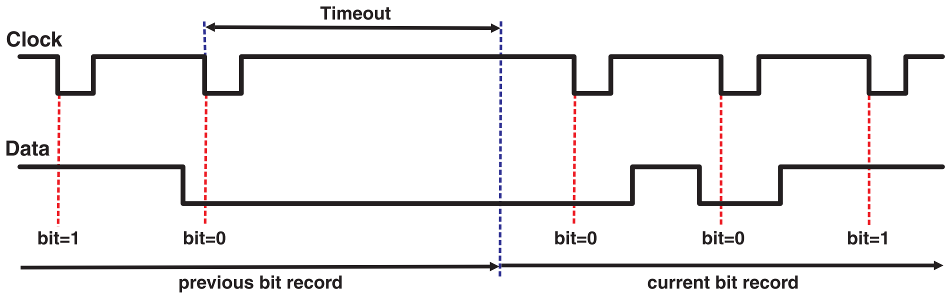
If data are not read by the BASIC program with a “GET“, the previous bit record is overwritten with new data; however, a “double-buffer“ mechanism however ensures that data records are always completed and that it is not a combination of two incomplete records.

Further driver specifications:

- Max. clock frequency: 20.000 Hz
- Min. clock frequency: 4 Hz (due to the 255 ms timeout upper limit)
- Timeouts: 2...255 ms

  Note: Timeout setting to “2“ creates a timeout of 1...2 ms
  Timeout setting to “x“ creates a timeout of (x-1)...x ms

Logic diagram:



SHI (Clocked, Serial Input)

Reading a serial data stream from the device driver's buffer is carried out by a GET instruction:

GET #D, 0, A$

	B	W	L	S	F	
D	●	●	●	-	-	Driver device number
A$	●	●	●	-	-	String which takes all databytes contained in the input buffer

Example:

Outputting a clocked, serial data stream with the SHIFT_OUT function via pins L84 and L85, receiving those data with the SHI driver at pins L80 and L81.

```
USER_VAR_STRICT                 ' check variable declaration

TASK MAIN                       ' start task MAIN

STRING Send$, Rec$              ' Strings for sending and receiving
LONG C                          ' Counter variable

INSTALL_DEVICE #1, "LCD1.TDD"                     ' Install LCD
INSTALL_DEVICE #2, "SHI_8081.TDD", 2, 1, 0, 1, 8 ' 2ms timeout, falling
                                                  ' edge, LSB, HBF, 8 bpb

DIR_PIN 8,4,0                   ' L84 is output (data pin for SHIFT_OUT)
DIR_PIN 8,5,0                   ' L85 is output (clock pin for SHIFT_OUT)
OUT 8,00110000b,00000000b       ' set L84 and L85 to idle state

FOR C = 100 TO 149              ' some cycles...
  Send$ = "Test" + STR$(C)      ' set up transmission string

  SHIFT_OUT 8, 4, 5, Send$, -64 ' 64 bits are shifted,
                                ' most significant bit (MSB) first
  WAIT_DURATION 10              ' wait briefly until transmission and
                                ' reception are completed
  GET #2, 0, Rec$               ' Read driver buffer content

  PRINT #1, "<1>"; Send$        ' output transmission string on LCD
  PRINT #1, Rec$                ' output reception string on LCD
  WAIT_DURATION 500             ' wait 500ms
NEXT                            ' next cycle

END                             ' End of task MAIN
```

empty page

Applications

empty page

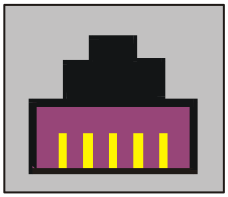
Ethernet Application

Function libraries: ts_???.inc

Connecting a Tiger to a network

Function: This libraries provide funtions for establishing a connection to a a network via an Ethernet/Web adapter.

These functions include the Ethernet/Web adapter configuration as well as dial-up and data exchange.



Thereby diverse protocols such as POP/SMTP (for email), HTTP (for Internet), ARP, TCP/IP, DHCP or DNS can be used for communication.

In the following descriptions first of all every function contained in these libraries and their application are explained. This is done with several sample programs. Their functionality and the functions used are explained.

If you have the compiler version 5.2 (or higher) installed, you find these programs in the directory “Examples_Web_Ethernet” of your Tiger installation. Otherwise you can also download them from our website “www.wilke-technology.com”.

Ethernet/Web Adapter

Programming Guide

Version 1.01a

Table of contents

Content

1. Introduction

1.1 About this application

Information about this application

- This application deals with the **Ethernet/Web Adapter** versions **EM01-ETH-S, EM02-SER-S, EM03-ETH-P, EM04-SER-P**.
- The information in this application is relevant for the **Ethernet/Web Adapters version V1.3 or higher** (see imprint on the front side of the Ethernet/Web modules) and for the **Tiger Basic Sockets (TBSockets) implementation version 1.01** (see the bGetAdapterProgVers subroutine in the "Get Version of the Adapter Software" paragraph of this manual).
- If the notation Ethernet/Web Adapter is used in this application, the corresponding text refers to all mentioned module types; the notation Ethernet Adapter is used for EM01-ETH-S and EM03-ETH-P modules only; the term Web Adapter only describes EM02-SER-S and EM04-SER-P modules.
- This application consists of two main parts. The **"Demo Programs"** chapter introduces client/server programming with Ethernet/Web Adapter and Tiger Basic and describes sample programs in detail:
    - *Client_Simple_Ethe.Tig*, *Client_Simple_Ppp.Tig* - implement simple tcp client that actively opens connection and communicates with an echo server.
    - *Server_Ethe.Tig* - implements simple tcp server that waits passively for someone to contact the Ethernet Adapter and sends simple messages to the remote peer in loop.
    - *Client_Dhcp_Ethe.Tig* - demonstrates how to get dynamic IP address (subnet mask and gateway) from DHCP server and starts a simple tcp client that actively opens connection and communicates with an echo server.
    - *Client_Dns_Ethe.Tig* - demonstrates how to get an IP address corresponding to a host name from DNS server and starts a simple tcp client that actively opens connection using the obtained IP address and communicates with an echo server.
    - *Smtp_Client_Ethe.Tig*, *Smtp_Client_Ppp.Tig* - demonstrate how to send an email using the SMTP (RFC 821, RFC 1651) protocol. The protocol is implemented in Tiger Basic language, it is delivered as a source code and can be changed by the user to comply with the requirements of the particular SMTP server.

  The chapter **"Programming with Tiger Basic Sockets (TBSockets)"** explains how to use particular Tiger Basic subroutines for implementing the client/server applications by means of Ethernet/Web Adapter.
- To understand how to use the Ethernet/Web Adapter without Tiger Basic controller, please read the "Ethernet_Adapter__Protocol_[Vers].pdf" document as well.

1.2 Protocols and Services Provided

1.2.1 Ethernet Adapter

Protocols provided

- Address Resolution Protocol (**ARP**; RFC 0826, RFC 1122)
- Internet Protocol (**IP**; RFC 0791)
- Internet Control Message Protocol (**ICMP**; RFC 0792)
- Domain Name Services (**DNS**; RFC 1034, RFC 1035) Client
- Transmission Control Protocol (**TCP**; RFC 0793, RFC 1122)
- Dynamic Host Configuration Protocol (**DHCP**; RFC 2131) Client

1.2.2 Web Adapter

- Point-to-Point Protocol (**PPP, LCP, IPCP, AHDLC**; RFC 1172, 1570, 1332)
- Internet Protocol (**IP**; RFC 0791)
- Internet Control Message Protocol (**ICMP**; RFC 0792)
- Domain Name Services (**DNS**; RFC 1034, RFC 1035) Client
- Transmission Control Protocol (**TCP**; RFC 0793)
- Dynamic Host Configuration Protocol (**DHCP**; RFC 2131) Client

1.2.3 Protocols implemented as software examples

- Simple Mail Transfer Protocol (**SMTP**, RFC 2821)
- Post Office Protocol - Version 3 (**POP3**, RFC 1939)

1.3 Software Terms and Requirements

Explanations concerning software

- The Ethernet/Web Adapter is pre-programmed to be able to exchange the commands and data with a Basic Tiger controller (or another controller) and to execute the commands. The user has no direct access to the program in the Ethernet/Web Adapter.
- The Basic Tiger controller communicates with the Ethernet/Web Adapter by using a selection of tasks and subroutines named Tiger Basic Sockets (TBSockets). TBSockets are written in Tiger Basic programming language and delivered as a source code. To compile TBSockets and appropriate demos **Tiger Basic IDE version 5.01 or higher** is required.
- The include (.inc) files with names having the “ts_” prefix contain the implementation of TBSockets. All TBSockets include files must be copied to the place that is accessible for the Tiger Basic Compiler, by default the location of the include files is the “Include” directory of the Tiger Basic installation and this location is immediately visible for the Compiler.

2. Demo Programs

2.1 Terminology

Definition

IP address: Identifier for a computer or device on a TCP/IP network. Networks using the TCP/IP protocol route messages based on the IP address of the destination. The format of an IP address is a 32-bit numeric address written as four numbers separated by periods. Each number can be zero to 255. For example, 1.140.16.220 could be an IP address. Within an isolated network, you can randomly assign IP addresses as long as each one is unique. However, connecting a private network to the Internet requires using registered IP addresses (called Internet addresses) to avoid duplicates.

Port: Endpoint to a logical connection in TCP/IP and UDP networks. The port number identifies what type of port it is. For example, port 80 is used for HTTP traffic.

DNS: Internet service that translates domain names into IP addresses. Because domain names are alphabetic, they're easier to remember. The Internet, however, is in fact based on IP addresses. Therefore every time you use a domain name a DNS service must translate the name into the corresponding IP address. For example, the domain name *www.example.com* might translate to *198.121.204.2*. The DNS system is, in fact, its own network. If one DNS server doesn't know how to translate a particular domain name, it asks another one, and so on, until the correct IP address is returned.

DHCP: Protocol for assigning dynamic IP addresses to devices on a network. With dynamic addressing, a device can have a different IP address every time it connects to the network. In some systems, the device's IP address can even change while it is still connected. DHCP also supports a mix of static and dynamic IP addresses. Dynamic addressing simplifies network administration because the software keeps track of IP addresses rather than requiring an administrator to manage the task. This means that a new computer can be added to a network without the hassle of manually assigning it to a unique IP address. Many ISPs use dynamic IP addressing for dial-up users.

PAP: The most basic form of authentication, in which a user's name and password are transmitted over a network and compared to a table of name-password pairs. Typically, the passwords stored in the table are encrypted. The main weakness of PAP is that both username and password are transmitted "in the clear" — that is, in an unencrypted form.

Client: An application that initiates the connection and relies on a server to perform some operations.

Server: An application that passively waits to respond to clients requests.

Socket: A software object that provides interface to a network protocol (i.e. TCP/IP). It is identified by protocol and local/remote address/port.

Tiger Basic Sockets (TBSockets): A collection of subroutines and tasks written in Tiger Basic programming language. It provides a software interface between Basic Tiger and Ethernet/Web Adapter that actually implements the TCP/IP protocol. To use TBSockets an application must include the “ts_coinc.inc” file.

2.2 Network Configuration

Network configuration

- The Ethernet/Web Adapter is not pre-configured, so any free address on the net may be used either for the Ethernet/Web Adapter itself or for the peer host (e.g. PC).
- If a crossover cable connected directly to the peer host is used, static IP addresses and subnet masks are required.
- If a router is used in the net which the PC (and the Ethernet/Web Adapter) will be attached to, the default gateway should be set up additionally to the IP address and the subnet mask.
- For a DHCP client demo a DHCP server must be installed on the network (e.g. on the PC).
- Please ask your **network administrator** for valid specifications. Setting improper data may cause the applications to not work (correctly) or even interfere with other network participants.

!

2.3 Ethernet/Web Adapter Settings

Adapter settings, configuration data

- The **EM01/EM02** Adapters communicate with the controlling device through the serial channel. Each Tiger Basic application intending to work with the EM01/EM02 Adapters must install the device driver for the particular **serial channel** with correct settings. The baud rate for the serial channel may be modified by using of the external selection mechanism (see “Datasheet_EM01_Eth_S_[Vers]“ or „Datasheet_EM02_Ser_S_[Vers]” documents). The default setting for baud rate is **38400 bauds** for **EM01**, and **19200 bauds** for **EM02**. Other parameters are unchangeable: **8N1**. The Basic-Tiger controller uses the serial channel 0 (**SER0**, with hardware handshake) to communicate with the EM01/EM02 Adapters.
- Most of the configuration constants useful for demo programs and applications can be found in the „ts_conf.inc“ file. The file is subdivided into logical sections. E.g. all constants concerning DNS are put together and a comment line containing the word „DNS“ marks the beginning of the section. The settings for the particular demo programs will be explained below, in the corresponding paragraphs.

- The “ts_conf.inc” file is included into the “ts_coinc.inc” file. So if an application includes “ts_coinc.inc”, “ts_conf.inc” is also included automatically.
- To work with many copies of the “ts_conf.inc” file at the same time, please comment out the corresponding ‘#include’ line in the “ts_coinc.inc” file and include the proper copy of the “ts_conf.inc” into the particular application file (tig).
- The constants defined in the section “MODULE TYPE” select the software configuration for the particular module type. It is very important to choose the type constant properly. Only one constant (TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04) must be activated at the moment.

2.4 Dialling Procedure for Web Adapters

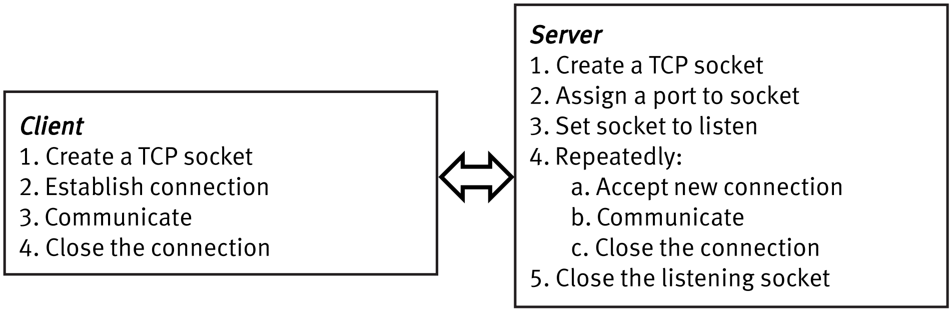
Dialling procedure for Web Adapters

Before the particular subroutines can be used to implement the client/socket functionality, the connection to an ISP (Internet Service Provider) should be established in the case of Web Adapter (EM02, EM04). Some of the demo programs below use the dialling procedure with the subsequent PAP authentication. The actual dialling is done with the *bDialIsp* subroutine call (or with the *bDialIspWithLogin* if the login and the password must be entered explicitly, not by using of PAP), but some settings must be performed before. The *bSetupPapSecret* subroutine sets the user name and the user password for PAP. The *bSetupIsp* subroutine defines the dialling number of the ISP (other parameters of this subroutine are used only if the *bDialIspWithLogin* mechanism must be activated).

2.5 Client/Server Model

2.5.1 TCP Client/Server Interaction

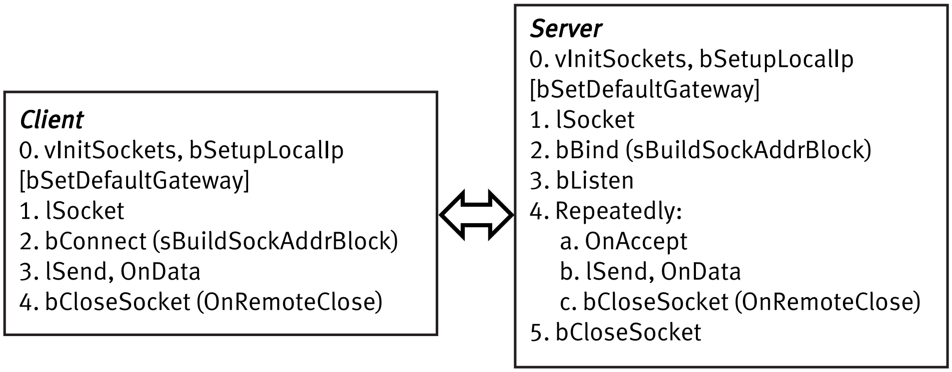
Client/Server interaction



Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection
5. Close the listening socket

2.5.2 TCP Client/Server Interaction using TBSockets



Client
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. bConnect (sBuildSockAddrBlock)
3. lSend, OnData
4. bCloseSocket (OnRemoteClose)

Server
0. vInitSockets, bSetupLocalIp
[bSetDefaultGateway]
1. lSocket
2. bBind (sBuildSockAddrBlock)
3. bListen
4. Repeatedly:
    a. OnAccept
    b. lSend, OnData
    c. bCloseSocket (OnRemoteClose)
5. bCloseSocket

2.5.2.1 How to set up a very simple TCP Client using TBSockets

Setting up a simple client

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Connect the remote peer (*bConnect*; one of the parameters of the subroutine is a timeout value defining how long to wait till the connection was successful),
- Send the data (*lSend*),
- Receive the data serving a callback task (*OnData*) launched each time the data are transmitted,
- Close the socket (*bCloseSocket*).

2.5.2.2 How to build a very simple TCP Server using TBSockets

Setting up a simple server

- Initialize the internal variables (*vInitSockets*), set the IP address (*bSetupLocalIp*) and, possibly, the default gateway (*bSetDefaultGateway*) for the local host,
- Create a new socket (*lSocket*),
- Bind the socket to a particular port (*bBind*),
- Start listening to the bound port for incoming connections (*bListen*)
- Accept the connection from a remote peer serving a callback task (*OnAccept*) launched each time when a new socket for the accepted connection is created,
- Send the data using the new socket (*lSend*),
- Receive the data serving a callback task (*OnData*) launched each time the data are transmitted,
- Close all open sockets (*bCloseSocket*).

All subroutines and tasks which may be of importance for implementing client/server applications with an Ethernet/Web Adapter are described in detail below in this “Programming Guide” manual.

2.6 Simple Client Demo

Example for a client application

- Name :
  client_simple_ethe.tig (EM01, EM03),
  client_simple_ppp.tig (EM02, EM04)
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only wild cards for user-defined reactions on some asynchronous events like transfer of data from remote peer etc.

- Purpose: This demo program is a simple client that actively opens connection and communicates with an **echo** server.

- Explanation:
  1. Dials the ISP (Web Adapters only) :
     - See “2.3 Dialling Procedure for Web Adapters”,
  2. Creates a new socket (*lSocket*),
  3. Establishes the connection to a remote host (*bConnect*),
  4. If the remote host was connected the message loop is started; the message loop involves:
     - Sending (*lSend*) the TEXT_TO_SEND character sequence,
     - Waiting for an echo or for quit signal (“Q” or “QUIT”; *OnData* task in the “client_clbks.inc”) or for closing of socket on the remote host (*OnRemoteClose* task in the “client_clbks.inc”); the timeout is not checked;
     - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
  5. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by the user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - Module type in the “MODULE TYPE” section:
    - TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
    - TS_LOCAL_IP_ADDRESS
    - TS_LOCAL_IP_SUBNET_MASK

- ➢ IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
  - ❖ TS_CONNECT_TO_PEER_IP
  - ❖ TS_CONNECT_TO_PEER_PORT
- ➢ Default gateway address in the “GATEWAY” section (if a router is a part of the network)
  - ❖ TS_DEFAULT_GATEWAY

- How to test it:
  - ✓ Use the “SServer.exe“ program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.7 Simple Server Demo

Example for a server application

- Name :

  server_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *server_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only wild cards for user-defined reactions on some asynchronous events like coming of data from remote peer etc.

- Purpose: This demo program is a simple server that waits passively for someone to contact the Ethernet Adapter and sends simple messages to the remote peer in loop .

- Explanation:
  1. Creates a new socket (*lSocket*),
  2. Binds the socket to a particular port (*bBind*),
  3. Listens to the bound port for incoming connections (*bListen*),
  4. Accepts the connection from a remote peer and stores a new socket for the accepted connection (*OnAccept* task in the “server_clbks.inc”),
  5. If a new connection was established the message loop is started; the message loop involves:
     - ○ Sending (*lSend*) the TEXT_TO_SEND character sequence through the new socket,
     - ○ Waiting for quit signal (“Q” or “QUIT“; *OnData* task in the “client_clbks.inc”) or for closing of socket on the remote host (*OnRemoteClose* task in the “client_clbks.inc”); the timeout is not checked;
     - ○ Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),

6. Closes all open sockets (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants which must be correctly set by the user to compile and run this demo are saved in the *ts_conf.inc* file in the following section:
  - ➢ Module type in the "MODULE TYPE" section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the "LOCAL HOST" section:
    - ❖ TS_LOCAL_IP_ADDRESS
    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ Port and IP address to listen (INADDR_ANY to accept every address) in the "SERVER SETTINGS" section:
    - ❖ TS_SERVER_LISTEN_TO_PORT
    - ❖ TS_SERVER_ACCEPT_IP

- How to test it:
  - ✓ Use **ping** program to check whether a device with the TS_LOCAL_IP_ADDRESS ip address is attached to the network.
    ping ‹ TS_LOCAL_IP_ADDRESS ›

  - ✓ Use **telnet** program to establish a connection to the server, to receive messages from it and to force the server to quit the session by entering „q".
    telnet ‹ TS_LOCAL_IP_ADDRESS › ‹ TS_SERVER_LISTEN_TO_PORT ›

2.8 DHCP Client Demo

Example for a DHCP client

- Name :
  client_dhcp_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only wild cards for user-defined reactions on some asynchronous events like the success of the DHCP request, coming of data from remote peer etc.

- Purpose: This program demonstrates how to get dynamic IP address (subnet mask and gateway) from **DHCP** server and starts a simple client that actively opens connection and communicates with an **echo** server.

- Explanation:
    1. Enables DHCP request (*bSetupDhcp*),
    2. Creating a new (first) socket forces (*lSocket)* the Ethernet Adapter to send a request to the DHCP server and obtain dynamic IP address, subnet mask and gateway,
    3. Creates a new socket (*lSocket*),
    4. Waits for dynamic parameters sent by the DHCP server (*OnDhcpUp* task in the „client_clbks.inc“) and checks the timeout value,
    5. Establishes the connection to a remote host (*bConnect*),
    6. If the remote host was connected the message loop is started; the message loop involves:
        - Sending (*lSend*) the TEXT_TO_SEND character sequence,
        - Waiting for an echo or for a quit signal (“Q” or “QUIT”; *OnData* task in the “client_clbks.inc”) or for closing of socket on the remote host (*OnRemoteClose* task in the “client_clbks.inc”); the timeout is not checked;
        - Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
    7. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
    - Module type in the “MODULE TYPE” section:
        - TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
    - Flag to enable the compiling of source code parts dealing with DHCP in the section “DHCP”
        - TS_DHCP_ENABLED
    - How long to wait for a response from the DHCP server in the section “DHCP”
        - TS_DHCP_REQUEST_TIMEOUT
    - Static IP address and subnet mask of the Ethernet Adapter (for the case if the DHCP server is unreachable) in the “LOCAL HOST” section:
        - TS_LOCAL_IP_ADDRESS
        - TS_LOCAL_IP_SUBNET_MASK
    - IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
        - TS_CONNECT_TO_PEER_IP
        - TS_CONNECT_TO_PEER_PORT
    - Default gateway address in the “GATEWAY” section (if a router is a part of the network)
        - TS_DEFAULT_GATEWAY

- How to test it:
  - ✓ Install and configure a DHCP server on the computer to which the Ethernet Adapter can connect.
    - ○ Some links to DHCP servers for Windows:
      - http://www.magikinfo.com/dhcp.htm (MagikDHCP)
      - http://www.billiter.com/ (ipLease)
    - ○ Some links to Internet sites describing how to configure a DHCP server on Linux:
      - http://www.tldp.org/HOWTO/Net-HOWTO/
  - ✓ Use the “SServer.exe” program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.9 DNS Client Demo

Example for a DNS client

- Name :

  client_dns_ethe.tig
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *client_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only wild cards for user-defined reactions on some asynchronous events like transfer of data from remote peer etc.

- Purpose: This program demonstrates how to get an IP address corresponding to a host name from **DNS** server and starts a simple client that actively opens connection using the obtained IP address and communicates with an **echo** server.

- Explanation:
  1. Creates a new socket (*lSocket*),
  2. Enables DNS requests and sets the DNS server IP address (*bSetupDns*),
  3. Sets a default gateway if necessary (*bSetDefaultGateway*),
  4. Tries to obtain an IP address for a host name from the DNS Server (*lDnsGetIpByName*),
  5. Establishes the connection to a remote host using the obtained IP address (*bConnect*),
  6. If the remote host was connected the message loop is started; the message loop involves:
     - ○ Sending (*lSend*) the TEXT_TO_SEND character sequence,
     - ○ Waiting for an echo or for a quit signal (“Q” or “QUIT”; *OnData* task in the “client_clbks.inc”) or for closing the socket on the

      remote host (*OnRemoteClose* task in the “client_clbks.inc”); the timeout is not checked;
    - ○ Repeating the loop either SEND_LOOPS times or quitting by quit or close-socket signal or running endless (timeout error),
  7. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  All constants that must be correctly set by user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - ➢ Module type in the “MODULE TYPE” section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04
  - ➢ Flag to enable the compiling of source code parts dealing with DNS in the section “DNS”
    - ❖ TS_ DNS_ENABLED
  - ➢ IP address of a DNS server in the section “DNS”
    - ❖ TS_DNS_SERVER_IP
  - ➢ How long to wait for a response from the DNS server in the section “DNS”
    - ❖ TS_DNS_REQUEST_TIMEOUT
  - ➢ Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
    - ❖ TS_LOCAL_IP_ADDRESS
    - ❖ TS_LOCAL_IP_SUBNET_MASK
  - ➢ IP address and port of the remote computer to which the connection must be established in the “CONNECTION TARGET” section:
    - ❖ TS_CONNECT_TO_PEER_NAME
    - ❖ TS_CONNECT_TO_PEER_PORT
    - ❖ TS_CONNECT_TO_PEER_IP – for the case if the DNS server is unreachable
  - ➢ Default gateway address in the “GATEWAY” section (if a router is a part of the network)
    - ❖ TS_DEFAULT_GATEWAY

- How to test it:
  - ✓ Use DNS server of an ISP or DNS server on LAN.
  - ✓ Use the “SServer.exe” program installed in the “..\Tools\SimpleServer” directory or another tcp server that is enabled to send an echo to the client.

2.10 SMTP Client Demo

Example for a SMTP client

- Name :
  smtp_client_ethe.tig (EM01, EM03),
  smtp_client_ppp.tig (EM02, EM04)
- Include Files:
  1. *ts_coinc.inc* – includes all TBSockets files,
  2. *smtp_pop_clbks.inc* – includes the user callback tasks for the demo; originally, these tasks are only placeholders for user-defined reactions on some asynchronous events like coming of data from remote peer etc.,
  3. *smtp_pop_subs.inc* – includes the subroutines implementing SMTP and POP client protocols.

- Purpose: This program demonstrates how to send an email using **SMTP** (RFC 821, RFC 1651) protocol. The protocol is implemented in Tiger Basic language, it is delivered as source code and can be changed by the user to comply with the requirements of the particular SMTP server,

- Explanation:
  1. Dials the ISP (Web Adapters only) :
     - ○ See “2.3 Dialling Procedure for Web Adapters”,
  2. Creates a new socket (*lSocket*),
  3. Establishes the connection to a SMTP server (*bConnect*), identified by IP address and port number (normally: 25)
  4. If the server is connected, an email will be prepared and sent; sending an email involves:
     - ○ Setting all values relevant to the authentication of the email account’s owner and of the sender (see below: *Configuration Constants*),
     - ○ Setting all values relevant to the header information and the contents of the particular email (see below: *Configuration Constants*),
     - ○ Sending the email (*mail_send*) by means of exchange of the SMTP requests and responses between this SMTP client program and the connected SMTP server;
  5. Closes the socket (*bCloseSocket*) after leaving the message loop.

- Configuration Constants:
  Most of the constants which must be correctly set by the user to compile and run this demo are saved in the *ts_conf.inc* file in the following sections:
  - ➢ Module type in the “MODULE TYPE” section:
    - ❖ TS_EM_01 or TS_EM_02 or TS_EM_03 or TS_EM_04

- ➢ Static IP address and subnet mask of the Ethernet Adapter in the “LOCAL HOST” section:
  - ❖ TS_LOCAL_IP_ADDRESS
  - ❖ TS_LOCAL_IP_SUBNET_MASK
- ➢ IP address or name (if DNS is enabled), and port of the SMTP server to which the connection must be established in the “SMTP” section:
  - ❖ TS_SMTP_SERVER_IP or TS_SMTP_SERVER_NAME
  - ❖ TS_SMTP_PORT
- ➢ Default gateway address in the “GATEWAY” section (if a router is part of the network)
  - ❖ TS_DEFAULT_GATEWAY
- ➢ Parameters of the user of the SMTP service, i.e. account name, optional login and password (if authentication is activated) in the “SMTP” section:
  - ❖ TS_SMTP_ACCOUNT_NAME
  - ❖ TS_SMTP_AUTH_ENABLED, TS_SMTP_LOGIN, TS_SMTP_PASSWORD
- ➢ Parameters of the sender of an email, i.e. sender name and sender domain (in the form “domainname.com”) in the “SMTP” section:
  - ❖ TS_SMTP_SENDER_NAME
  - ❖ TS_SMTP_SENDER_DOMAIN

The parameters and the contents of the particular email are directly placed in the *smtp_client_ethe.tig* file :

- ➢ The name of the email sender, the email address of the receiver, the subject of the email and the message itself:
  - ❖ EMAIL_DATA_FROM
  - ❖ EMAIL_DATA_TO
  - ❖ EMAIL_DATA_SUBJECT
  - ❖ EMAIL_DATA_INIT_TEXT

- How to test it:
  - ✓ Use an extern SMTP server or a local one (i.e. from “QKsoft” http://www.qksoft.com/).

3. Programming with Tiger Basic Sockets (TBSockets)

3.1 Terms

3.1.1 What is TBSockets

Tiger Basic Sockets description

TBSockets is a selection of tasks and subroutines written in the Tiger Basic programming language, which implements an interface to the network world for the Basic Tiger based applications using specific hardware (Ethernet/Web Adapter designed by Wilke Technology GmbH). TBSockets consists of user interface subroutines (plus user defined callback tasks) and of chain of the supporting tasks and subroutines. This document describes the user interface.

3.1.2 How to use TBSockets

To use the TBSockets features:

- Include the 'ts_coinc.inc' file into the projects main 'tig' file,
- For the EM01/EM02 Adapters: install the device driver for serial channel in the "Main" task. The default settings for serial channel 0 (**SER0**): **38400, 8N1** for **EM01**, and **19200, 8N1** for **EM02**. The serial channel 0 is preferred because of implemented hardware handshake,
- Write the callback tasks *OnData*, *OnAccept, OnRemoteClose, OnDhcpUp* etc (find the appropriate placeholder tasks in the "def_clbks.inc" file in the "Ethernet_Web_Examples" directory),
- Call the *vInitSockets* (sub vInitSockets()) subroutine before using any other TBSockets subroutine. The *vInitSockets* intializes some variables, and starts supporting tasks,
- Write a client/server application based on the subroutines described below.

3.2 General Setup Subroutines

3.2.1 Set Local Ip Address and Local Ip Subnet Mask

Set local IP address and subnet mask

Purpose:
The *bSetupLocalIp* subroutine is used to set the local ip address to *lpLocalIpAddress* and the local ip subnet mask to *lpLocalIpSubnetMask*.

Signature:
sub bSetupLocalIp(long lpLocalIpAddress; long lpLocalIpSubnetMask; var byte bpvSuccess)

3.2.2 Set Default Gateway

Set default gateway

Purpose:
The *bSetDefaultGateway* subroutine sets the default gateway (the IP address of the intranet interface for the incoming connection) to the *lpDefaultGateway* value.

Signature:
sub bSetDefaultGateway(long lpDefaultGateway; var byte bpvSuccess)

Comments:

- If this subroutine is not called, the default gateway for a Web Adapter will be initially set to 192.168.1.253 (hex: C0A801FD).
- If this subroutine is not called, the default gateway for a Ethernet Adapter will be not initialised at all.
- Once it was explicitly or implicitly set, the default gateway can not be deleted.

3.3 'Get Param' Subroutines

3.3.1 Get Local Ip Address

Get local IP-Address

Purpose:
The *lGetLocalIp* subroutine gets the local ip address for the *lpSocket* socket.

Signature:
sub lGetLocalIp(long lpSocket; var long lpvLocalIp)

Return:
On error: the *lpvLocalIp* is set to DUMMY_IP (hex: FFFFFFFF) and the *lLastErrorCode* contains the error code.
On success: the *lpvLocalIp* contains the requested ip address.

3.3.2 Get Local Port Number

Get local port

Purpose:
The *wGetLocalPort* subroutine gets the local port number for the *lpSocket* socket.

Signature:
sub wGetLocalPort(long lpSocket; var word wpvLocalPort)

Return:
On error: the *wpvLocalPort* is set to DUMMY_PORT (hex: FFFF) and the *lLastErrorCode* contains the error code.
On success: the *wpvLocalPort* contains the requested port number.

3.3.3 Get Remote Ip Address

Get server IP address

Purpose:
The *lGetRemoteIp* subroutine gets the ip address of the remote host for the *lpSocket* socket.

Signature:
sub lGetRemoteIp(long lpSocket; var long lpvRemoteIp)

Return:
On error: the *lpvRemoteIp* is set to DUMMY_IP (hex: FFFFFFFF) and the *lLastErrorCode* contains the error code.
On success: the *lpvRemoteIp* contains the requested ip address.

3.3.4 Get Remote Port Number

Get server port

Purpose:
The *wGetRemotePort* subroutine gets the port number of the remote host for the *lpSocket* socket.

Signature:
sub wGetRemotePort(long lpSocket; var word wpvRemotePort)

Return:
On error: the *wpvRemotePort* is set to DUMMY_PORT (hex: FFFF) and the *lLastErrorCode* contains the error code.
On success: the *wpvRemotePort* contains the requested port number.

3.3.5 Get Version of the Adapter Software

Get firmware version

Purpose:
The *bGetAdapterProgVers* subroutine returns the version of the adapter program.

Signature:
sub bGetAdapterProgVers(var long lpvProgVers)

Return:
On error: the *lpvProgVers* is set to (-1) and the *lLastErrorCode* contains the error code.
On success: the *lpvProgVers* contains the requested version.

Comments:

- The program version is returned in a long variable and can be converted to the readable string form by using of the *sConvProgVersToString* subroutine.

  Signature:
  sub sConvProgVersToString(long lpProgVers; var string spvProgVers$)
- The ADAPTER_PROG_VERS_STR_SIZE constant tells how long the spvProgVers$ string must minimally be.

3.4 Ethernet MAC Address Subroutines (Ethernet Adapter)

The Ethernet MAC (Media Access Control) address is a hardware address which uniquely identifies each node of an Ethernet network. The Ethernet MAC addresses are 48 bits, usually expressed as 12 hexadecimal digits. All Ethernet adapters (both EM01 and EM03 series) are delivered with an unique MAC address stored in the non-volatile memory. So, normally, using this subroutine must be avoided, except an area of unique MAC addresses was additionally reserved for produced series, otherwise changing the MAC address can be dangerous.

3.4.1 Set MAC Address

Set new Ethernet MAC address

Purpose:
The *bSetMacAddress* subroutine forces an Ethernet adapter to use the new *spMacAddress$* MAC address and to store this in the non-volatile memory.

Signature:
sub bSetMacAddress(string spMacAddress$; var byte bpvSuccess)

Comments:

- There are two methods to assign the 12 hexadecimal digits of an Ethernet MAC address to the *spMacAddress$* string in the Tiger Basic program:
  spMacAddress$ = „FF FF FF FF FF FF“%
  or
  spMacAddress$ = „<0FFh> <0FFh> <0FFh> <0FFh> <0FFh> <0FFh>“
- The MAC_ADDR_SIZE constant tells how long the *spMacAddress$* string must minimally be.

3.4.2 Get MAC Address

Get Ethernet MAC address of the adapter

Purpose:
The *bGetMacAddress* subroutine gets the actually used MAC address of the Ethernet adapter and saves it in the spvMacAddress$ variable.

Signature:
sub bGetMacAddress(var string spvMacAddress$; var byte bpvSuccess)

Comments:
The MAC_ADDR_SIZE constant tells how long the *spMacAddress$* string should be.

3.5 Tcp Settings Subroutines

3.5.1 Set Tcp Window Size

Set TCP window size

The tcp window size determines how many characters can be sent or received in one tcp package.

Purpose:
The *bSetTcpWinSize* subroutine sets the tcp window size to the *lpTcpWinSize* value. This value cannot exceed 128.

Signature:
sub bSetTcpWinSize(long lpSocket; long lpTcpWinSize; var byte bpvSuccess)

Comments:

- The default tcp window size for any adapter is 128 bytes.

3.5.2 Set Tcp Keep-Alive Segments

Cyclic transmission of dummy data for testing and keeping alive an active connection

Purpose:
The *bSetTcpKeepAlive* subroutine activates the keep-alive mechanism and sets its parameters to send a keep-alive probe every *lpTcpKATicks* milliseconds *bpTcpKAProbes* times.

Signature:
sub bSetTcpKeepAlive(long lpSocket; long lpTcpKATicks; byte bpTcpKAProbes; var byte bpvSuccess)

Comments:

- The *lpTcpKATicks* variable specifies the number of milliseconds between consecutive keep-alive probes.
- The *bpTcpKAProbes* variable specifies the number of probes that can be sent without response before declaring a socket to be dead.
- To deactivate the keep-alive mechanism, call the *bSetTcpKeepAlive* subroutine with the *lpTcpKATicks* and *bpTcpKAProbes* parameters set to 0 (zero).
- By default, the sending of the keep-alive segments is not activated.

3.5.3 Get Tcp Settings

Get TCP settings

Purpose:
The *lGetTcpSettings* subroutine gets the actually used tcp window size and tcp keep-alive parameters.

Signature:
sub lGetTcpSettings(long lpSocket; var long lpvTcpWinSize, lpvTcpKATicks;
var byte bpvTcpKAProbes; var byte bpvSuccess)

Comments:
If the *lpTcpKATicks* and the *bpvTcpKAProbes* variables are set to 0 (zero), the keep-alive mechanism is deactivated.

3.6 Modem Subroutines (Web Adapter)

3.6.1 Communicate directly via Modem

Functions for communicating via modem

The subroutines of this subsection immediately enable data exchange with a modem, without involving any function of a Web Adapter which might influence the transferred data.

Purpose:
The *bModemSend* subroutine sends the *spDataToSend$* string to the attached modem.

Signature:
sub bModemSend (string spDataToSend$; long lpTimeOut; var byte bpvDataSent)

Purpose:
The *bModemReceive* subroutine receives the data (*lpRecvBufSize* maximum size) from the modem in the *spvReceivedData$* string.

Signature:
sub bModemReceive(var string spvReceivedData$; long lpRecvBufSize; long lpTimeOut; var byte bpvIsReceived)

Purpose:
The *wModemGetSendReady* subroutine returns the number of bytes of data a modem can accept for sending to the *wpvSendSize* parameter .

Signature:
sub wModemGetSendReady(long lpTimeOut; var word wpvSendSize)

Purpose:
The *wModemGetRecvReady* subroutine returns the number of bytes of unprocessed data the modem has in its receive buffer to the *wpvRecvSize* parameter.

Signature:
sub wModemGetRecvReady(long lpTimeOut; var word wpvRecvSize)

Return:
All subroutines presented above try to get the particular work done during the *lpTimeOut* period of time. If the *lpTimeOut* expires and the function is not executed the return value is FALSE for *bModemSend* and *bModemReceive* subroutines or 0 (zero) for *wModemGetSendReady* and *wModemGetRecvReady* subroutines; the corresponding error code is saved in the *lLastErrorCode* variable.

3.6.2 Send AT Commands

Send AT commands

Purpose:
The *bSendATCommand* subroutine sends the *spATCommandToSend$* AT command to the modem and receives a reply in the *spvATReply$*.

Signature:
sub bSendATCommand(string spATCommandToSend$; long lpATTimeOut; var string spvATReply$; var byte bpvSuccess)

Comments:

- The *bSendATCommand* subroutine appends the ending Carriage Return (0d hex) character to the *spATCommandToSend$* string.
- The maximal size of the reply which is set to 128 now (see: “AT_REPLY_MAX_SIZE”) must not be changed by the user. The *lpATTimeOut* is a timeout value measured in seconds to wait for the modem to reply.

3.6.3 Listen to Modem Data

“overhear” modem

The alternative mechanism to communicate with the modem is listening to the modem until a certain number of data has been transferred and a callback task is launched.

Purpose:
The *bModemStartListen* subroutine starts the listening procedure. The *lpMinRecvDataSize* parameter defines how many characters must be received before the callback task can be launched.

Signature:
sub bModemStartListen(long lpMinRecvDataSize; var byte bpvSuccess)

Purpose:
The bModemStopListen subroutine stops the listening procedure which was started by the *bModemStartListen* call.

Signature:
sub bModemStopListen(var byte bpvSuccess)

Purpose:
The callback task which is launched when data is transferred.

Signature:
task OnModemData

Comments:

- Two global variables are set by the TBSockets Launcher at launching this task: *lActModemDataSize* and *sActModemDataBuffer*$.
- The *lActModemDataSize* variable stores the size of the transferred data.
- The *sActModemDataBuffer*$ variable stores the transferred data itself.
- The *OnModemData* task is normally written by the user. In the given examples the implementation of this task can be found in the include files named according to the following pattern: 'application_name_clbks.inc'.

3.6.4 Dialling Procedure

Dialling procedure

The dialling procedure for the modem consists of dialling an Internet Service Provider number (phone number set by *bSetupIsp*) and authenticating of the user (user name and password set by *bSetupIsp* or *bSetPapSecrets* according to the chosen diallingscheme).

3.6.4.1 Set Internet Service Provider Data

ISP settings

Purpose:
The *bSetupIsp* subroutine sets the data used while dialling an Internet Service Provider to the *spModemDialString*$ phone number (without ATDT-prefix and without ‹Carriage Return›-suffix), the *spUserName*$ user name and the *spUserPassword*$ user password.

Signature:
sub bSetupIsp(string spModemDialString$; string spUserName$;
string spUserPassword$; var byte bpvSuccess)

Comments:

- In case of applying the *bDialIsp* (not *bDialIspWithLogin*) subroutine for dialling an ISP the *spUserName*$ and the *spUserPassword*$ parameters will be ignored and the proper user name and password should be set by means of the *bSetPapSecrets* subroutine.
- The *bSetupIsp* subroutine must be called before dialling the ISP by *one* of the dialling subroutines.

3.6.4.2 Set PAP Secrets (Authentication Parameters)

Set authentication parameter s

Purpose:
The *bSetPapSecrets* subroutine is used to define the authentication parameters transferred to ISP as a part of the PPP protocol. The so called PAP secrets will be set to *spUserName*$ user login name and to *spUserPassword*$ user password. The parameter

bpIndex declares to which set of authentication parameters the login name and password belong.

Signature:
sub bSetPapSecrets (string spUserName$; string spUserPassword$; byte bpIndex; var byte bpvSuccess)

Comments:

- The parameters of each set will be tested one after another until the connection to an ISP is established. The maximal number of authentication parameter sets is 3.
- The call of the *bDialIsp* (not *bDialIspWithLogin*) subroutine causes the transferring of PAP secrets during PPP phase.
- The *bSetPapSecrets* subroutine must be called before dialling the ISP by the *bDialIsp* subroutine.

3.6.4.3 Dial with Login

Dialling in ISP with login data

Purpose:
According to this dialling scheme the login procedure is started immediately after dialling an ISP, and PPP is actively launched on the successful logging (user name and password are correct)

Signature:
sub bDialIspWithLogin(long lpDialTimeout; var long lpvAssignedIpAddr; var byte bpvSuccess)

Comments:
The phone number, user name and password must be set by means of the *bSetupIsp* subroutine.

3.6.4.4 Dial without Login

Dialling in ISP without login data

Purpose:
Some ISPs (for example ISPs for GPRS) require the authentication made by PAP during the PPP phase. The dialling procedure for such an ISP doesn’t invoke sending user name and password, because the ISP itself establishes the PPP connection and asks the client for authentication.

Signature:
sub bDialIsp(long lpDialTimeout; var long lpvAssignedIpAddr; var byte bpvSuccess)

Comments:
The appropriate authentication parameters (login and password) must be set for this dialling scheme by means of the *bSetPapSecrets* subroutine.

Comments:

- Both dialling subroutines return an IP address assigned by the ISP to our node to the *lpvAssignedIpAddr* parameter.
- The *lpDialTimeout* defines the timeout to wait for connection to the ISP.

3.6.5 Hanging Up Procedure

Hang up/cut off connection to ISP

Purpose:
The *bHangUp* subroutine forces the adapter to finish the modem session by sending the commands “+++” and “ath” (both commands wait for “OK”s) to the modem.

Signature:
sub bHangUp(var byte bpvSuccess)

3.6.6 Set the Modem Baud Rate

Define modem baud rate

Purpose:
The *bSetModemBaudrate* subroutine sets the speed of communication between the adapter and modem to *lpModemBaudRate*.

Signature:
sub bSetModemBaudrate (long lpModemBaudRate; var byte bpvSuccess)

3.6.7 Get the CTS Pin State

Get the CTS pin state

Purpose:
The *bGetCtsPinState* subroutine returns in *bpvCtsPinState* the state of the CTS pin of the adapter connected to modem. The pin state is one of the following constants: PIN_STATE_HIGH (1), PIN_STATE_LOW (0), PIN_STATE_UNDEF (hex: FF).

Signature:
sub bGetCtsPinState(var byte bpvCtsPinState)

Return:
On error: the *bpvCtsPinState* is set to PIN_STATE_UNDEF (hex: FF) and the *lLastErrorCode* contains the error code.
On success: the *bpvCtsPinState* is either PIN_STATE_HIGH (1) or PIN_STATE_LOW (0).

3.7 DHCP Subroutines

Routinen for the DHCP protocol

An application using DHCP should enable dhcp (*bSetupDhcp* call) before creating the first socket (*lSocket* call); the very first call of the *lSocket* causes sending of dhcp request, and in this phase the application should check whether the dhcp request was successful while the *bDhcpIsUp* global variable is tested on “TRUE” (*bDhcpIsUp* must be previously set to “FALSE“); if the request was successful, the callback task *OnDhcpUp* is launched, the *bDhcpIsUp* variable is set to “TRUE” and the new ip address assigned by dhcp server is stored in the *lActDhcpUpIpAddr* global variable.

3.7.1 Enable DHCP

(de)activate DHCP protocol

Purpose:
The *bSetupDhcp* subroutine activates or deactivates the request made by the client to the remote DHCP server to get the ip address and other dynamically assigned parameters of the connection.

Signature:
sub bSetupDhcp(byte bpDhcpFlag; var byte bpvSuccess)

Comments:

- The value of *bpDhcpFlag* is one of the following constants defined in ‘ts_com_d.inc’:
  USE_DHCP (1) – activates DHCP request,
  NO_USE_DHCP (2) – deactivates DHCP request,
  USE_STANDARD - NO_USE_DHCP
- The *bSetupDhcp* subroutine must be called before opening the first socket by the *lSocket* subroutine.

3.7.2 Get the IP address assigned by DHCP Server

Get IP addresses assigned by the DHCP server

Purpose:
If the request succeeds, the *OnDhcpUp* callback task is launched and the new ip address assigned by the dhcp server is stored in the *lActDhcpUpIpAddr* global variable.

Signature:
task OnDhcpUp

Comments:

- Three global variables are set by the TBSockets Launcher at launching this task:
  long lActDhcpUpIpAddr,
  long lActDhcpUpNetMask,
  long lActDhcpUpGateway.
- The *lActDhcpUpIpAddr* variable stores the ip address, the *lActDhcpUpNetMask* variable – the subnet mask, and the *lActDhcpUpGateway* variable – the default gateway assigned to the client by the remote DHCP server.
- The *OnDhcpUp* task is normally written by the user. In the delivered examples the implementation of this task can be found in the include files named corresponding to the following pattern: ‘application_name_clbks.inc’.

3.8 DNS Subroutines

Routines for DNS protocol

An application using DNS should send the first dns request only after the first new socket was created (*lSocket* call). Then the actual dns request is done by means of the following:

- the dns is enabled and the dns server ip address is set by *bSetupDns* call,
- the default gateway ip address is set by *bSetDefaultGateway* call,
- the dns request for a remote host name is done by *lDnsGetIpByName* call;

if the dns request was successful, the corresponding ip address is given back to the application as a parameter of the *lDnsGetIpByName* subroutine.

3.8.1 Enable DNS and set DNS Server IP address

(de)activate DNS protocol

Purpose:
The *bSetupDns* subroutine enables the DNS request made by the client to the remote DNS server to get the ip address corresponding to a particular host name. The *bSetupDns* subroutine also sets the IP address of the DNS server to the *lpDnsServerIpAddress* value.

Signature:
sub bSetupDns(byte bpDnsFlag; long lpDnsServerIpAddress; var byte bpvSuccess)

Comments:

- The value of *bpDnsFlag* is one of the following constants defined in ‘ts_com_d.inc’:
  USE_DNS (1) – enables DNS request,
  NO_USE_DNS (2) – disables DNS request,
  USE_STANDARD - NO_USE_DNS
- The *bSetupDns* subroutine must be called after opening the first socket by the *lSocket* subroutine.

3.8.2 Get IP address for a Host Name from DNS Server

Get IP address for a host from the DNS server

Purpose:
The *lDnsGetIpByName* subroutine requests the IP address for the spHostName$ host name from the DNS Server.

Signature:
sub lDnsGetIpByName(string spHostName$; long lpTimeOut; var long lpvIpAddress)

Return:
On error: the *lpvIpAddress* is set to zero (0) and the *lLastErrorCode* contains the error code.
On success: the *lpvIpAddress* contains the requested ip address.

Comments:

- Before the DNS request can be accomplished, the DNS must be enabled and the IP address of the DNS Server must be configured (*bSetupDns* call), and the Default Gateway must be set (*bSetDefaultGateway* call).

3.9 Client-Server Subroutines

Routines for client/ server communikation

Typically, one of the ends of a socket-based data communication is a *server*, the other is a *client*.

3.9.1 The Common Elements

3.9.1.1 Open Socket

Open socket / connection

Purpose:
The subroutine used by both, client and server, is *lSocket*. The *lSocket* subroutine allocates a new socket with the required characteristics. The maximal number of the sockets running concurrently is limited to 6 (six).

Signature:
sub lSocket(byte bpAddrFormat, bpType; var long lpvSocket)

Return:
On error: the *lpvSocket* is set to (-1) and the *lLastErrorCode* contains the error code.
On success: the *lpvSocket* contains the numerical identifier of the allocated socket.

Comments:

- The *bpAddrFormat* is an address format specification. The only format currently supported is PF_INET, which is the ARPA Internet address format. The constant PF_INET is defined in ‘ts_com_d.inc’.
- Two values are defined for the *bpType* argument, again, in ‘ts_com_d.inc’. Both start with ‘SOCK_’. The most common one is SOCK_STREAM, which tells the system you are asking for a *reliable stream delivery service* (which is TCP in this case).
- If you asked for SOCK_DGRAM, you would be requesting a *connectionless datagram delivery service* (in our case, UDP). The UDP service is not supported in this release. Please don’t use it.
- The Unconnected Socket: Nowhere in the *lSocket* subroutine have we specified to what other system we should be connected. Our newly created socket remains *unconnected*.

3.9.1.2 Close Socket

Close socket/ connection

Purpose:
Each opened socket must be closed if it is no longer in use.

Signature:
sub bCloseSocket(long lpSocket; var byte bpvSuccess)

Comments:

- The *lpSocket* argument is the socket, i.e., the value returned by the *lSocket* subroutine or the value saved in the *lActAcceptSocket* variable when accepting a new client.

3.9.1.3 Remote Socket Closed Notification

Notification of disconnection from the remote socket

Purpose:
If the remote peer closes the connection, the '*OnRemoteClose'* task is started.

Signature:
Task OnRemoteClose

Comments:

- The *lActRemoteCloseSocket* global variable contains the socket that was immediately closed by the remote peer.
- The additional call of the bCloseSocket subroutine is not necessary. If called, it will return an error.

3.9.1.4 Socket Address Block

Socket address block structure

Various subroutines of the sockets family expect the string generally referred to as 'Socket Address Block' as one of the arguments. The data of different types and sizes are stored in the Socket Address Block string. The particular fields of this block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about the network addresses:

Offset	Size	Description
SABLK_LEN_OFFS	SABLK_LEN_SIZE	Size of SA Block
SABLK_FAMILY_OFFS	SABLK_FAMILY_SIZE	Address Family
SABLK_PORT_OFFS	SABLK_PORT_SIZE	Port
SABLK_ADDR_OFFS	SABLK_ADDR_SIZE	IP Address

The size of the Socket Address Block is fix now (use the constants SABLK_DEFAULT_LEN, SABLK_INET_LEN defined in 'ea_conf.inc' and 'ts_com_d.inc'), but may be modified in the future.

The only *address family* currently supported is AF_INET (defined in 'ts_com_d.inc').

The *port* is a value of type 'word' (16-bit unsigned integer) standing for the connection port number.

The *ip address* is of type 'long' (32-bit signed integer). Because the value of a numeric type can not directly represent an ip address in the more convenient 'dotted' notation, one should convert it into a 32-bit integer. For example the value 0C0A80102H (or 3232235778) is used to express the 192.168.1.2 ip address.

The exact meaning of the *port* and *ip address* fields depends on the subroutine of the sockets family using the Socket Address Block.

Two subroutines considerably facilitate accessing the particular values of the Socket Address Block:

Signature:
sub sBuildSockAddrBlock(var string spvSockAddrBlock$; byte bpSaLength, bpSaFamily; word wpSaPort; long lpSaAddress)
and

Signature:
sub vParseSockAddrBlock(string spSockAddrBlock$; var byte bpvSaLength, bpvSaFamily; var word wpvSaPort; var long lpvSaAddress)

The *sBuildSockAddrBlock* subroutine copies *bpSaLength*, *bpSaFamily, wpSaPort* and *lpSaAddress* to the Socket Address Block string *spvSockAddrBlock$*.
On the contrary the *vParseSockAddrBlock* subroutine extracts the particular values of the Socket Address Block *spSockAddrBlock$* and saves them in *bpvSaLength*, *bpvSaFamily*, *wpvSaPort*, *lpvSaAddress*.

3.9.2 Client

Functions for a client

Typically, the client initiates the connection to the server. The client knows which server it is about to call: it knows its IP address, and it knows the *port* the server resides at.

3.9.2.1 Connect

Connect to host via a specific port

Purpose:
Once a client has created a socket, it needs to connect it to a specific port on a remote system.

Signature:
sub bConnect(long lpSocket; long lpConnEstTimeOut; string spSockAddr$; word wpSockAddrLen; var byte bpvSuccess)

Return:
If *bConnect* is successful, it returns TRUE in the *bpvSuccess*. Otherwise it returns FALSE and stores the error code in the *lLastErrorCode* variable (global).

Comments:

- The *lpSocket* argument is the socket, i.e., the value returned by the *lSocket* subroutine.
- The *spSockAddr$* string is a Socket Address Block, the structure of which we presented extensively. In this case, the *port* and *ip address* identify the server which has to be connected.
- Finally, *wpSockAddrLen* informs the system how many bytes are in our Socket Address Block string.
- There are many reasons why *bConnect* may fail. For example, with an attempt to an Internet connection, the IP address may not exist, or it may be down, or just too busy, or it may not have a server listening at the specified port. Or it may outrightly *refuse* any request for a specific code.

3.9.3 Server

Functions for a server

The typical server does not initiate the connection. Instead, it waits for a client to call it and request services. It does know neither when the client will call, nor how many clients will call. It may be just sitting there, waiting patiently, one moment, the next moment, it can find itself swamped with requests from a number of clients, all calling in at the same time.
The sockets interface offers two basic subroutines and one callback task to handle this.

3.9.3.1 Bind

Bind a socket to a specific port

Purpose:
There are 65535 IP ports, but a server usually processes requests that come in on only one of them. The *bBind* subroutine is used to tell sockets which port is to serve.

Signature:
sub bBind(long lpSocket; string spSockAddr$; word wpSockAddrLen; var byte bpvSuccess)

Return:
If *bBind* succeeds, it returns TRUE in the *bpvSuccess*. Otherwise it returns FALSE and stores the error code in the l*LastErrorCode* variable (global).

Comments:

- The *lpSocket* argument is the socket, i.e., the value returned by the *lSocket* subroutine.
- The *spSockAddr$* string is a Socket Address Block of the *wpSockAddrLen* length. Besides specifying the *port* in *spSockAddr$*, the server may include its *ip address*. However, it can just use the symbolic constant INADDR_ANY to

indicate it will serve all requests to the specified port regardless of what its IP address is. This symbol, is defined in ‘ts_com_d.inc’.

3.9.3.2 Listen

Wait for incoming connection

Purpose:
The server waits for an incoming connection with the *bListen* subroutine.

Signature:
sub bListen(long lpSocket, lpBackLog; var byte bpvSuccess)

Return:
The *bListen* subroutine returns in the *bpvSuccess* TRUE on success, otherwise the *bpvSuccess* is FALSE and the l*LastErrorCode* variable contains the error code.

Comments:

- The *lpSocket* argument is the socket, i.e., the value returned by the *lSocket* subroutine.
- The *lpBackLog* variable tells sockets how many incoming requests to accept while you are busy processing the last request. In other words, it determines the maximum size of the queue of pending connections.
- The number of back logs (*lpBackLog*) is limited to 5 (five).

3.9.3.3 Connection Accepted Notification

Connection accepted by server

Purpose:
The server accepts the connection by starting the ‘*OnAccept*’ task.

Signature:
task OnAccept

Comments:

- Three global variables are set by the server and can be analysed in the *OnAccept* task, that is started by the TBSockets Launcher if the server accepts a new connection:
  long lActAcceptSocket,
  word wActAcceptSABlockLen,
  string sActAcceptSABlock$
- The *lActAcceptSocket* value differs from the socket used in the *bBind* and *bListen* subroutines. Indeed, a new socket is created on accept. You will use this new socket to communicate with the client.
- What happens to the old socket? It continues to listen for more requests (remember the *lpBackLog* variable we passed to *bListen*?) until we close it.
- Now, the new socket is only meant for communications. It is fully connected. We cannot pass it to *bListen* again, trying to accept additional connections.

- The *sActAcceptSABlock$* string contains the port number and the ip address of the client.
- An established connection with a client remains active until either server or client hang up.
- The *OnAccept* task is normally written by the user. In the delivered examples the implementation of this task can be found in the include files named correspondingly to the following pattern: ‘application_name_clbks.inc’.

3.9.4 Send and Receive Data

Once the connection is established the data can be exchanged in both directions.

Routines for data transfer

3.9.4.1 Send

Purpose:
The *lSend* subroutine must be used to send the data to the remote host.

Sending data

Signature:
sub lSend(long lpSocket; string spData$; long lpDataLen; word wpFlags; var long lpvSentBytesNum)

Return:
The *lpvSentBytesNum* argument returns the total number of bytes sent to the remote host with the *lSend* subroutine. The l*LastErrorCode* variable contains the error code if sending fails.

Comments:

- The *lpSocket* argument is the socket, i.e., the value returned by the *lSocket* subroutine.
- The *lSend* subroutine sends the data of *lpDataLen* from the *spData$* string. If *lpDataLen* is longer than the size of a packet, the data will be split into many packets by *lSend*. A time interval between two packets can be defined by using of the TS_PARTIAL_SEND_PAUSE constant. By default, the TS_PARTIAL_SEND_PAUSE constant is set to 0 (zero).
- The *wpFlags* argument is reserved for future extensions and is not used now.

3.9.4.2 Data Received Notification

Purpose:

Data received norification

If the data are received, the TBSockets launches the *OnData* task:

Signature:
task OnData

Comments:

- Three variables are set by the TBSockets Launcher and must be used to access the received data:
  long lActDataSocket,
  long lActDataSize,
  string sActDataBuffer$
- The *lActDataSocket* variable identifies the socket to which the received data belong.
- The *sActDataBuffer$* string contains the proper data of the *lActDataSize* length.
- The *OnData* task is normally written by the user. In the examples provided the implementation of this task can be found in the include files named corresponding to the following pattern: 'application_name_clbks.inc'.

3.10 Error Handling and Error Codes

Overview error handling and error codes

- If the return value is not explicitly described for the particular subroutine, this simple rule must be applied: if the subroutine succeeds the *bpvSuccess* variable is set to TRUE (1), if it fails the *bpvSuccess* is FALSE (0).

- In case of error nearly all TBSockets subroutines specify the reason of failure in the *lLastErrorCode* variable in case of error. The following constants are used as error codes (the constants are defined in the 'ts_com_d.inc' file):

  - *all subroutines*
    CME_TIMEOUT(254) - timeout error for any command

  - *all SetUp subroutines*
    CME_SETUP_BAD_SUBCOMMAND (1) - not implemented subcommand

  - *bSetupDhcp*
    CME_SETUP_INV_DHCP_FLAG (3) - invalid DHCP flag value

  - *bSetupDns*
    CME_SETUP_INV_DNS_FLAG (4) - invalid DNS flag value

  - *bSetupIsp*
    CME_SETUP_MDM_DIAL_NO_MEM (6) - no memory to copy the modem dial string
    CME_SETUP_USER_NAME_NO_MEM (7) - no memory to copy the isp user name
    CME_SETUP_USER_PASSWORD_NO_MEM (8) - no memory to copy the isp user password

- *bSetPapSecrets*
  CME_SETUP_PAP_ID_TOO_LONG (9) - pap id too long
  CME_SETUP_PAP_PASSW_TOO_LONG (10) - pap password too long
  CME_SETUP_PAP_INV_INDEX (11) - invalid pap secrets index

- *bSetMacAddress*
  CME_SETUP_MAC_INV_SIZE (12) - size of mac address is not 6 bytes

- *bSetTcpWinSize*
  CME_SETUP_TCP_WIN_SIZE_INV_SOCK (13) - invalid socket for tcp window size
  CME_SETUP_TCP_WIN_SIZE_INV (14) - invalid size for tcp window

- *bSetTcpKeepAlive*
  CME_SETUP_TCP_KA_INV_SOCK (15) - invalid socket for tcp keep-alive

- *lGetLocalIp, wGetLocalPort, lGetRemoteIp, wGetRemotePort, lGetTcpSettings*
  CME_GET_PARAM_INV_SOCK (1) - invalid socket

- *bGetCtsPinState*
  CME_GET_PARAM_CTS_NOT_IN_USE (2) - cts pin is not used (handshake is off)

- *bGetMacAddress*
  CME_GET_PARAM_MAC_INV_BUF_SIZE (3) - not enough memory for mac string (not sent by Ethernet/Web Adapter)

- *bSendATCommand*
  CME_SEND_AT_NOT_AVAIL (1) - command not available
  CME_SEND_AT_MDM_NOT_INIT (2) - the modem is not initialised yet
  CME_SEND_AT_MDM_SEND_NOT_READY (3) - the modem send buffer is full, and the timeout is expired
  CME_SEND_AT_MDM_RECV_NOT_READY (4) - the modem receive buffer is empty, and the timeout is expired

- *bModemSend, bModemReceive, wModemGetSendReady, wModemGetRecvReady*
  CME_MC_NOT_AVAIL (1) - the command not available
  CME_MC_BAD_SUBCOMMAND (2) - the subcommand not available
  CME_MC_MDM_NOT_INIT (3) - the modem is not initialised yet
  CME_MC_SEND_NOT_READY (4) - the modem send buffer is full, and the timeout is expired
  CME_MC_RECV_NOT_READY (5) - the modem receive buffer is empty, and the timeout is expired

- *bModemStartListen*
  CME_MC_NOT_ALL_SOCKS_FREE (6) - not all sockets are free, ergo cannot start listening to modem
  CME_MC_MIN_SIZE_TOO_BIG (7) - min size is bigger than max packet size
  CME_MC_ALREADY_LISTENING (8) - listening to modem already started

- *bModemStopListen*
  CME_MC_NO_LISTENING (9) - no procedure listening to modem

- *bDialIsp, bDialIspWithLogin*
  CME_DIAL_NOT_AVAIL (1) - command not available
  CME_DIAL_BAD_SUBCOMMAND (2) - subcommand not available
  CME_DIAL_INV_TIMEOUT (3) - invalid timeout value
  CME_DIAL_ALREADY_DIALED (4) - already connected
  CME_DIAL_TIME_EXPIRED (5) - not connected: time expired
  CME_DIAL_NO_DIAL_STRING (6) - no dial string entered
  CME_DIAL_NO_USER_NAME (7) - user name not initialised
  CME_DIAL_NO_USER_PASSWD (8) - user password not initialised

- *bHangUp*
  CME_HANGUP_NOT_AVAIL (1) - the command is not available

- *lDnsGetIpByName*
  CME_DNS_EFORMAT (1) - the server believes the request was improperly formated
  CME_DNS_ESERVER (2) - the DNS server encountered an internal failure
  CME_DNS_ENAME (3) - the requested name does not exist
  CME_DNS_ENOTIMP (4) - the name server does not support the requested kind of query
  CME_DNS_EREFUSED (5) - the name server refused the request
  CME_DNS_EYXDOMAIN (6) - some name that ought not to exist, does exist
  CME_DNS_EYXRRSET (7) - some RRset that ought not to exist, does exist
  CME_DNS_ENXRRSET (8) - some RRset that ought to exist, does not exist
  CME_DNS_ENOTAUTH (9) - the server is not authorative for the zone named in the Zone section
  CME_DNS_ENOTZONE (10) - a name used in the prerequisites or Update Section is not within the zone denoted by the Zone section
  CME_DNS_ETIMEOUT (33) - the request timed out
  CME_DNS_EBADANSWER (34) - the DNS subsystem was unable to

understand the returned request. The returned request failed one of several internal validations
CME_DNS_NOT_AVAIL (40) – dns command not available
CME_DNS_SERV_NOT_KNOWN (41) - ip of dns server not set
CME_DNS_NO_MEM (42) - no memory to execute the command
CME_DNS_NOT_READY (43) - timeout error
CME_DNS_NAME_TOO_LONG (50) – requested name too long

- *lSocket*
  CME_SOCKET_INV_AF (1) - invalid address format
  CME_SOCKET_INV_TYPE (2) - invalid type
  CME_SOCKET_NO_MEM (3) - no memory for new socket
  CME_SOCKET_INV_SOCK (4) - invalid socket returned by pair (not sent by Ethernet/Web Adapter)
  CME_SOCKET_SOCK_OCCUPIED (5) - socket is already occupied (not sent by Ethernet/Web Adapter)

- *bCloseSocket*
  CME_CLOSE_INV_SOCK (1) - invalid socket

- *bConnect*
  CME_CONNECT_INV_SOCK (1) - invalid socket
  CME_CONNECT_SOCK_BOUND (2) - socket is already bound
  CME_CONNECT_SIZE_FAULT (3)- incorrect size of SABlk
  CME_CONNECT_TIMED_OUT (9)- timed out
  CME_CONNECT_SOCK_BUSY (10)- socket is busy
  CME_CONNECT_NO_ROUTE (11)- source address is invalid

- *bBind*
  CME_BIND_INV_SOCK (1) - invalid socket
  CME_BIND_SOCK_BOUND (2) - socket is already bound
  CME_BIND_SIZE_FAULT (3) - incorrect size of SABlk
  CME_BIND_ADDR_IN_USE (4) - ip/port pair is in use
  CME_BIND_NET_DOWN (6) - net is down (udp only)

- *bListen*
  CME_LISTEN_INV_SOCK (1) - invalid socket
  CME_LISTEN_BACKLOG_OVER (2) - too many backlogs
  CME_LISTEN_SOCK_BUSY (4) - socket is busy

- lSend
  CME_SEND_INV_SOCK (1) - invalid socket
  CME_SEND_NOT_SENT (2) - send error
  CME_SEND_SOCK_NOT_CONNECTED (3) - socket is not connected
  CME_SEND_NO_MEM (4) - no memory to send data

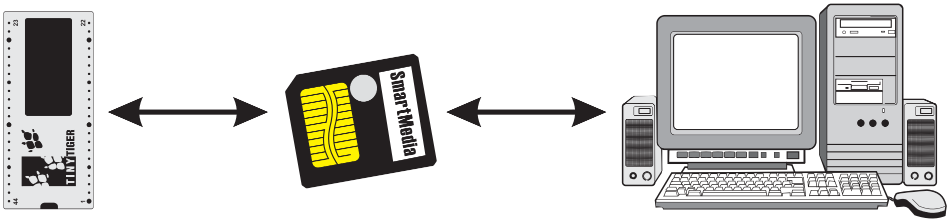
SmartMedia Application

Function libraries: fs_???.inc

Write to and read from SmartMedia cards using FAT-16 file system

Function: These libraries provide functions which allow working on a SmartMediaCard connected to a BASIC-Tiger using a FAT-16 file system .

So these functions enable the user to read a SmartMedia card written by a BASIC-Tiger with the PC and vice versa. This simplifies data transfer betwenn Tiger and PC enormously.



This does not only work with a PC, but also with all other devices which use the FAT-16 file system.

For using SmartMedia cards also the according device drivers are required (Smedia_xxMB.tdd). The Tiger communicates with the card via those drivers.

In the following section we will give you a description of all functions contained in this library and how to use them. Also a number of smaple programs is provided.

If you have the compiler version 5.2 (or higher) installed, you find these programs in the directory “Examples_SmartMedia” of your Tiger installation. Otherwise you can also download them from our website “www.wilke-technology.com”.

Basic Tiger File System for SmartMedia

Version 1.04

Table of contents

Contents

empty page

1. Introduction

General information

Basic Tiger File System (BTFS) is a collection of subroutines written in the Tiger Basic programming language and implementing the general functionality of the FAT file system for permanent storage devices. BTFS consists of three hierarchical layers: File System API, FAT implementation, special hardware support. A device driver for the particular hardware underlies the BTFS.

1.1 BTFS for SmartMedia Card

BTFS for SmartMedia Card has the following structure:
FS API ← FAT ← SmartMedia Routines ← SmartMedia Device Driver.

FS API is a set of subroutines working with files and directories. FS API layer is considered to be the most interesting layer for an application programmer and exactly this layer is described more detailed in this chapter.

FAT is an implementation of the FAT12/FAT16 file system (with long names support).

SmartMedia Routines is a set of subroutines written with regard to the specifications of SmartMedia card devices.

SmartMedia Device Driver is a Basic Tiger device driver implementing an elementary interface between SmartMedia hardware and a Tiger Basic application.

1.2 BTFS for SmartMedia File List

1.2.1 FS Include Files (directory “File_System” or “Include”)

Related files

fs_conf.inc	definitions that can be changed by user
fs_coinc.inc	definitions relevant for all FS layers; co-including all FS components. *Only this file must be explicitly included in the Tiger Basic application using BTFS*.
fs_inx_i.inc	implementation of FS API and some maintaining subroutines.
fs_inx_d.inc	definitions relevant for FS API.
fs_fat_i.inc	implementation of FAT12/FAT16 with long names support.
fs_fat_d.inc	definitions useful for FAT12/FAT16 implementation.
fs_fmt_i.inc	implementation of formatting process.
fs_dat_i.inc	implementation of date and time conversions.
fs_dat_d.inc	definitions relevant for date and time conversions.
fs_hal_d.inc	definition of Hardware Abstraction Layer; HAL is used to simplify the adaptation of the file system subroutines to the working with other storage devices.

fs_smc_i.inc	implementation of subroutines working with SmartMedia and conforming to the SmartMedia specifications and to the special features of the SmartMedia device driver.
fs_smc_d.inc	definitions relevant for SmartMedia subroutines.
fs_ecc_i.inc	implementation of ECC calculation for SmartMedia.

1.2.2 FS Examples (directory "File_System")

dir_create_del.tig, file_open.tig, file_size.tig, file_pointer.tig, file_attributes.tig, file_time.tig, file_format.tig, file_sync.tig, file_copy.tig, file_find.tig, get_hd_info.tig, get_fs_info.tig

1.2.3 SmartMedia Device Drivers (directory "TB_Drivers" or "Bin")

smedia_16mb.tdd, smedia_32mb.tdd, smedia_64mb.tdd, smedia_128mb.tdd
Any device driver fits for all SmartMedia cards of the exact or smaller size.

1.2.4 SmartMedia Functions (directory "TB_System_Files" or "Bin")

Some new built-in functions are extensively used by the BTFS subroutines. The functions are located in the following enclosed system files:
tac0000.tac, tac0000_.tac, tac0100.tac, tac0100_.tac
The enclosed system files require the Tiger Basic compiler version 5.01 or higher.

1.2.5 SmartMedia Low Level Examples (directory "Random_Access")

smedia_test_era_wr_rd_ser0_v03.tig, smedia_hex_dump_to_ser_02.tig
Note: This test may destroy very important SmartMedia header information and make the SmartMedia card unusable.

1.3 Supported SmartMedia Card Types and Other Limitations

Supported media and restrictions

The following SmartMedia card types are supported by BTFS at present: 1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb, 64Mb, 128Mb.

Most formatting programs use FAT12/FAT16 format for the various types of SmartMedia cards, but can be set to use other formats. You should avoid this as only FAT12/FAT16 is supported by BTFS.

Although long file names are supported, it's not possible to differentiate files with identical first 6 characters.

The BTFS subroutines are not re-entrant. Be careful using the BTFS subroutines in the different tasks.

1.4 BTFS System Requirements

BTFS requires the Tiger Basic compiler version 5.01 or higher. *The enclosed system files (extension: TAC) must be copied to the "..\Bin" directory of the Tiger Basic software.*

2. File System API (application program interface)

2.1 File System Setup

2.1.1 Initialising the File System Hardware

Setting up the File System

Subroutine:
sub bFileSystemHardwareInit(var byte bpvHdInitOk)

The *bFileSystemHardwareInit* subroutine calls special subroutines initializing a particular storage medium (e.g.: SmartMedia) that is to be used by the file system. This subroutine retrieves also the parameters of the storage medium.

This subroutine returns in *bpvHdInitOk* TRUE on successful initializing, and FALSE on error.

Be prepared: This subroutine can take long when run with SmartMedia.

Example: all

2.1.2 Setting Up the File System

Subroutine:
sub bSetupFileSystem(var byte bpvIsFSSetupOk)

The *bSetupFileSystem* subroutine initializes internal file system data, reads the boot sector and retrieves current file system settings.

This subroutine returns in *bpvIsFSSetupOk* TRUE on success, and FALSE on error.

Example: nearly all

2.2 Opening and Closing Files

2.2.1 Opening Files

Opening and Closing Files

Subroutine:
sub lOpenFile(string spFileName$; long lpFlags; var long lpvHandle)

The *lOpenFile* subroutine creates and returns a new file descriptor for the file named by *spFileName$*. Initially, the file position indicator for the file exists at the beginning of the file.

The *lpFlags* argument controls how the file is to be opened. This is a bit mask; you create the value by using bitwise OR on the appropriate parameters (using the 'bitor' operator in TB). File status flags *lpFlags* fall into three following categories.

Access modes

File Access Modes:
The file access modes allow a file descriptor to be used for reading, writing, or both. The access modes are chosen when the file is opened, and never change.

O_RDONLY
Open the file for read access.

O_WRONLY
Open the file for write access.

O_RDWR
Open the file for both reading and writing.

O_RDONLY and O_WRONLY are independent bits that can be bitwise-ORed together, and it is valid for either bit to be set or clear. This means that O_RDWR is the same as O_RDONLY|O_WRONLY. A file access mode of zero is equal in meaning to O_RDWR.

Open-time Flags:
The open-time flags specify options affecting how open will behave. These options are not preserved once the file is open.

O_CREAT
The file will be created if it doesn't already exist.

O_EXIST
Check, whether the file exists, don't open the file. In the case of a success the return value is zero, which does not mean that a file descriptor was assigned to an opened file.

I/O Operating Modes:
The operating modes affect how input and output operations using a file descriptor work.

O_APPEND
The bit that enables append mode for the file. If set, then all 'write' operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file.

The normal return value *lpvHandle* from *lOpenFile* is a non-negative long integer file descriptor. In the case of an error, a value of {-1} is returned instead.

Example

Example: "file_open.tig"

2.2.2 Closing File

Subroutine:
sub bCloseFile(long lpHandle; var byte bpvIsFileClosed)

The *bCloseFile* subroutine closes the file descriptor *lpHandle*.

The normal return value *bpvIsFileClosed* from *bCloseFile* is TRUE. If the file descriptor *lpHandle* is invalid, the value *bpvIsFileClosed* is assigned to FALSE.

Example: “file_open.tig”

2.3 File Input and File Output

2.3.1 Reading File

Reading and writing files

Subroutine:
sub lReadFile(long lpHandle; var string spvBuffer$; long lpSize; var long lpvNumBytesRead)

The *lReadFile* subroutine reads up to *lpSize* bytes from the file with descriptor *lpHandle*, storing the results in the *spvBuffer$*. (This is not necessarily a character string, and no terminating null character is added.)

The return value *lpvNumBytesRead* is the number of bytes actually read. This might be less than *lpSize*; for example, if there aren’t that many bytes left in the file. Note that reading less than *lpSize* bytes is not an error.

A value of zero indicates end-of-file (except if the value of the *lpSize* argument is also zero). This is not considered an error. If you keep calling *lReadFile* during end-of-file, it will keep returning zero and do nothing else.
If *lReadFile* returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero.
In case of an error, *lReadFile* returns {-1}.

Example

Example: “file_open.tig”

2.3.2 Writing File

Subroutine:
sub lWriteFile(long lpHandle; string spBuffer$; long lpSize; var long lpvNumBytesWritten)

The *lWriteFile* subroutine writes up to *lpSize* bytes from *spBuffer$* to the file with descriptor *lpHandle*. The data in *spBuffer$* is not necessarily a character string and a null character is output like any other character.

The return value is the number of bytes actually written. This may be *lpSize*, but can be smaller. Your program should call *lWriteFile* in a loop, iterating until all data are written. In the case of an error, *lWriteFile* returns {-1}.

Example

Example: “file_open.tig”

2.4 Setting and Getting the File Position of a Descriptor

Getting and Setting the position in a file

The File Position of a Descriptor specifies the position in the file for the next read or write operation.

2.4.1 Getting the File Position

Subroutine:
sub lGetFilePointer(long lpHandle; var long lpvCurFilePtr)

The *lGetFilePointer* subroutine is used to read the file position of the file with descriptor *lpHandle*.

The return value *lpvCurFilePtr* from lGetFilePointer is normally the current file position, measured in bytes from the beginning of the file. If the value of the file descriptor is invalid, *lGetFilePointer* returns a value of {-1}.

Example

Example: „file_pointer.tig“

2.4.2 Setting the File Position

Subroutine:
sub lSetFilePointer(long lpHandle; long lpOffset; byte bpWhence; var long lpvNewFilePtr)

The *lSetFilePointer* subroutine is used to change the file position of the file with the descriptor *lpHandle*.

The *bpWhence* argument specifies how the *lpOffset* should be interpreted, and it must be one of the symbolic constants FILE_BEGIN, FILE_CURRENT, or FILE_END.
FILE_BEGIN
Specifies that bpWhence is a count of characters from the beginning of the file. This count must be positive.

FILE_CURRENT
Specifies that bpWhence is a count of characters from the current file position. This count may be positive or negative.
FILE_END
Specifies that bpWhence is a count of characters from the end of the file. This count must be positive.

The return value *lpvNewFilePtr* from *lSetFilePointer* normally is the resulting file position, measured in bytes from the beginning of the file. You can use this feature together with FILE_CURRENT to read the current file position, although the using of *lGetFilePointer* is more efficient.
If the file position cannot be changed, or the operation is in some way invalid, *lSetFilePointer* returns a value of {-1}.
The position past the current end can not be set, and the file can not be extended by using *lSetFilePointer*.

Example

Example: “file_pointer.tig”

2.5 Getting the File Size

Getting file size

Subroutine:
sub lGetFileSize(long lpHandle; var long lpvFileSize)

The *lGetFileSize* subroutine is used to read the file size of the file with descriptor *lpHandle*.

The return value *lpvFileSize* from *lGetFileSize* is normally the file size, measured in bytes. The subroutine *lGetFileSize* returns a value of {-1} on error.

Example

Example: “file_size.tig”

2.6 Creating Directories

Erzeugen von Verzeichnissen

Subroutine:
sub bCreateDirectory(string spFileName$; var byte bpvIsCreated)

The *bCreateDirectory* subroutine creates a new, empty directory with name *spFileName$*.

A return value *bpvIsCreated* of TRUE indicates successful completion, and FALSE indicates failure.

Example

Example: “dir_create_del.tig”

2.7 Deleting Files and Directories

Deleting files and directories

Subroutine:
sub bDeleteFile(string spFileName$; var byte bpvIsDeleted)

The *bDeleteFile* subroutine deletes the file or the directory *spFileName$*.
A read-only file (i.e. a file with the set „DIR_ATTR_READONLY“ attribute) cannot be removed.
A directory must be empty before it can be removed; in other words, it can only contain entries for '.' and '..'.

This subroutine returns in *bpvIsDeleted* TRUE on successful completion, and FALSE on error.

Example

Example: "dir_create_del.tig"

2.8 Setting Current Directory

Setting current directory

Current Directory is a directory to which every not absolute path is related. A root directory name consists of one character "\" ("/" is also accepted). An absolute path always begins with the root directory name. A relative path must never have the root directory name as a very first part of the whole path.

Subroutine:
sub bSetCurrentDir(string spNewCurrentDir$;var byte bpvIsDirSet)

The *bSetCurrentDir* subroutine sets Current Directory to the *spNewCurrentDir$*.

This subroutine returns in *bpvIsDirSet* TRUE on successful setting, and FALSE on error.

Example

Example: "dir_create_del.tig"

2.9 File Attributes

Reading and setting file attributes

File Attribute is a byte value describing the most common properties of any particular file system entry (file or directory). A File Attribute is a combination of following constants:

DIR_ATTR_FILE
The entry is a file.

DIR_ATTR_READONLY
The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.

DIR_ATTR_SYSTEM
The file or directory is part of or is used exclusively by the operating system.

DIR_ATTR_HIDDEN
The file or directory is hidden. It is not included in an ordinary directory listing.

DIR_ATTR_VOLUME
Volume label attribute means that this entry contains the disk label in the filename and extension fields. Volume label is valid only in the root directory. Common sense says, there should only be one volume label per disk. For the entry to really contain the volume label, the attribute should be exactly DIR_ATTR_VOLUME.

DIR_ATTR_DIRECTORY
The entry is a directory.

DIR_ATTR_ARCHIVE
The file or directory is an archive file or directory. Applications use this flag to mark files for backup or removal.

2.9.1 Getting the File Attributes

Getting file attributes

Subroutine:
sub bGetFileAttributes(string spFileName$; var byte bpvFileAttr; var byte bpvAttrReadOk)

The *bGetFileAttributes* subroutine reads a file attribute value of the file *spFileName$*, storing the result in the *bpvFileAttr*.

This subroutine returns in *bpvAttrReadOk* TRUE on successful reading, and FALSE on error.

Example

Example: “file_attributes.tig”

2.9.2 Setting the File Attributes

Setting file attributes

Subroutine:
sub bSetFileAttributes(string spFileName$; byte bpNewFileAttr; var byte bpvAttrSetOk)

The *bSetFileAttributes* subroutine writes a new File Attribute value *bpNewFileAttr* of the file *spFileName$*.

This subroutine returns in *bpvAttrSetOk* TRUE on successful writing, and FALSE on error.

Example

Example: “file_attributes.tig”

2.10 File Time

2.10.1 Time and Date Format

Date and time for getting and settimg files

The file time fields have the following format:

Bits	Range	Translated Range	Valid Range	Description
0..4	0..31	0..62	0..59	Seconds/2
5..10	0..63	0..63	0..59	Minutes
11..15	0..31	0..31	0..23	Hours

The file date fields have the following format:

Bits	Range	Translated Range	Valid Range	Description
0..4	0..31	0..31	1..28 up to 1..31	Day
5..8	0..15	0..15	1..12	Month
9..15	0..127	1980..2107	1980..2107	Year, add 1980 to convert

2.10.2 Getting the File Time

Getting file date and time

Subroutine:
sub bGetFileTime(string spFileName$; var word wpvCreateDate, wpvCreateTime, wpvAccessDate, wpvWriteDate, wpvWriteTime; var byte bpvIsTimeRead)

The *bGetFileTime* subroutine retrieves the date and time that a file *spFileName$* was created, last accessed, and last modified.

wpvCreateDate
Date the file was created.
wpvCreateTime
Time the file was created.
wpvAccessDate
Date the file was last accessed.
wpvWriteDate
Date the file was last modified.
wpvWriteTime
Time the file was last modified.

All the time and date fields are represented in the format described in the “Time and Date Format”.

Example

Example: “file_time.tig”

2.10.3 Setting the File Time

Setting file date and time

Subroutine:
sub bSetFileTime(string spFileName$; word wpCreateDate, wpCreateTime, wpAccessDate, wpWriteDate, wpWriteTime; var byte bpvIsTimeWritten)

The *bSetFileTime* subroutine sets the date and time a file *spFileName$* was created, last accessed, and last modified.

wpCreateDate	Date the file was created.
wpCreateTime	Time the file was created.
wpAccessDate	Date the file was last accessed.
wpWriteDate	Date the file was last modified.
wpWriteTime	Time the file was last modified.

All the time and date fields are represented in the format described in the “Time and Date Format”.

Example

Example: “file_time.tig”

2.11 Find File

Find files

Two subroutines described below return the search result in a string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about a found file:

Information about file

Offset	Size	Description
FFD_ATTR_OFFS	FFD_ATTR_SIZE	file attribute
FFD_CREATE_TIME_MS_OFFS	FFD_CREATE_TIME_MS_SIZE	ms part of file creating time
FFD_CREATE_TIME_OFFS	FFD_CREATE_TIME_SIZE	file creating time
FFD_CREATE_DATE_OFFS	FFD_CREATE_DATE_SIZE	file creating date
FFD_ACCESS_DATE_OFFS	FFD_ACCESS_DATE_SIZE	date of the last file access
FFD_SIZE_OFFS	FFD_SIZE_SIZE	file size
FFD_NAME_OFFS	FFD_NAME_SIZE	file name (max. 8 symbols)
FFD_EXT_OFFS	FFD_EXT_SIZE	file extension (max. 3 symbols)
FFD_LONG_NAME_OFFS	FFD_LONG_NAME_SIZE	long file name
FFD_ABRIDGED_NAME_OFFS	FFD_ABRIDGED_NAME_SIZE	abridged file name

Note:

1. The following subroutines searches only for short file names (names in the format 8.3). So two long names with 6 or more equal first characters can not be differentiated.
2. If the file name was found and there is an entry for the long name, this long name will be saved in the memory block at the FFD_LONG_NAME_OFFS offset or at the FFD_ABRIDGED_NAME_OFFS offset (if this form of presentation is preferred).
3. The file name at the FFD_NAME_OFFS offset is extended with blanks up to FFD_NAME_SIZE (8) size; the file extension at the FFD_EXT_OFFS offset – up to FFD_EXT_SIZE (3) size.
4. The abridged form of presentation makes sense if one knows that the file name is in the format 8.3 and one would like to use the found name (placed at the FFD_ABRIDGED_NAME_OFFS offset in the format 8.3 with dot and without extending blanks) directly in the next file operation.
5. The size of the memory block can be equal or greater than FFD_STRUCT_SHORT_SIZE.
6. The following size constants are predefined:
   - FFD_STRUCT_SHORT_SIZE: without fields for the long or abridged file name
   - FFD_STRUCT_ABRIDGED_SIZE: FFD_STRUCT_SHORT_SIZE + the maximal length of the file name in the abridged form (FFD_NAME_SIZE + FFD_EXT_SIZE + 1[for “dot”])
   - FFD_STRUCT_FULL_SIZE: FFD_STRUCT_SHORT_SIZE + the maximal length of the long file name
   - FFD_STRUCT_DEFAULT_SIZE: FFD_STRUCT_ABRIDGED_SIZE

2.11.1 Searching file name

Searching file name

Subroutine:
sub bFindFirstFile(string spSearchedFileName$; var string spvFfdStruct$; var byte bpvFound)

The *bFindFirstFile* subroutine searches a directory for a file whose name matches the specified *spSearchedFileName*$ file name
and fills the *spvFfdStruct$* string with the information about the found file on success.
The *spSearchedFileName*$ filename can contain wildcard characters (* and ?).

This subroutine returns in *bpvFound* TRUE on success, and FALSE on error.

Subroutine:
sub bFindNextFile(var string spvFfdStruct$; var byte bpvFound)

The *bFindNextFile* subroutine continues searching a directory for a file whose name matches the filename that was specified in the previous call of the *bFindFirstFile* subroutine in the parameter *spSearchedFileName*$ and fills on success the *spvFfdStruct$* string with the information about the found file. The process begins at the position next to the position where the previous search was successfully completed by the *bFindFirstFile* or *bFindNextFile* subroutine.

This subroutine returns in *bpvFound* TRUE on success, and FALSE on error.

Example

Example: “file_find.tig”

2.12 Getting the information about the storage media

Information about storeage media

Subroutine:
sub bGetHardwareInfo(var string spvInfoSet$; var byte bpvIsRead)

The *bGetHardwareInfo* subroutine reads the information about the currently used storage media into the *spvInfoSet$* string.

The *bGetHardwareInfo* subroutine returns TRUE in the *bpvIsRead* on successful reading, and FALSE on error.

The *bGetHardwareInfo* subroutine saves the result in the *spvInfoSet$* string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about a the storage media:

Offset	Size	Description
HDI_MAKER_CODE_POS	HDI_MAKER_CODE_SIZE	manufacturer code
HDI_ID_CODE_POS	HDI_ID_CODE_SIZE	card identifier
HDI_BYTES_IN_SPARE_POS	HDI_BYTES_IN_SPARE_SIZE	number of bytes in spare field
HDI_BYTES_IN_PAGE_POS	HDI_BYTES_IN_PAGE_SIZE	number of DATA bytes in a page
HDI_PAGES_IN_BLOCK_POS	HDI_PAGES_IN_BLOCK_SIZE	number of pages in a block
HDI_BYTES_IN_BLOCK_POS	HDI_BYTES_IN_BLOCK_SIZE	number of DATA bytes in a block
HDI_NO_OF_BLOCKS_POS	HDI_NO_OF_BLOCKS_SIZE	total number of blocks
HDI_ADR_HIGH_BLOCK_POS	HDI_ADR_HIGH_BLOCK_SIZE	base address of the highest block
HDI_ADR_END_POS	HDI_ADR_END_SIZE	end address = first address after the last byte

Note:
The size of the *spvInfoSet$* string must be equal or higher than HDI_BLOCK_SIZE.

Example

Example: “get_hd_info.tig”

2.13 Getting information about the file system

Information about file system

Subroutine:
sub bGetFileSystemInfo(var string spvBootRecord$; var byte bpvIsBootRecRead)

The *bGetFileSystemInfo* subroutine reads information about the file system into the *spvBootRecord*$ string. Information is extracted from the boot record of a FAT-formatted storage media.

The *bGetFileSystemInfo* subroutine returns TRUE in the *bpvIsBootRecRead* on successful reading, and FALSE on error.

The *bGetFileSystemInfo* subroutine saves the result in the *spvBootRecord*$ string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing information about a the storage media:

Offset	Size	Description
BS_OEM_NAME_POS	BS_OEM_NAME_SIZE	the system that formatted the disk
BPB_BYTES_PER_SECT_POS	BPB_BYTES_PER_SECT_SIZE	the length in bytes of one physical sector
BPB_SECT_PER_CLUSTER_POS	BPB_SECT_PER_CLUSTER_SIZE	the number of sectors in one logical cluster
BPB_RESERVED_SECT_POS	BPB_RESERVED_SECT_SIZE	the number of reserved sectors
BPB_NUMBER_OF_FATS_POS	BPB_NUMBER_OF_FATS_SIZE	the number of file allocation tables
BPB_ROOT_ENTRIES_POS	BPB_ROOT_ENTRIES_SIZE	the number of entries in the root directory
BPB_TOTAL_SECT_POS	BPB_TOTAL_SECT_SIZE	total number of sectors on the disk
BPB_MEDIA_POS	BPB_MEDIA_SIZE	media descriptor
BPB_SECT_PER_FAT_POS	BPB_SECT_PER_FAT_SIZE	the number of sectors in one FAT
BPB_HIDDEN_SECT_POS	BPB_HIDDEN_SECT_SIZE	the number of hidden sectors
BPB_TOTAL_SECT_BIG_POS	BPB_TOTAL_SECT_BIG_SIZE	the a number of sectors if greater 65535
BS_VOLUME_LABEL_POS	BS_VOLUME_LABEL_SIZE	the disk label
BS_FILE_SYSTEM_POS	BS_FILE_SYSTEM_SIZE	the file system name (FAT12/16)

Note:

The size of the *spvBootRecord*$ string must be equal or greater than BOOT_RECORD_SIZE.

Example

Example: “get_fs_info.tig”

2.14 Formatting the Storage Media

Formatting media

Subroutine:
sub bFormatMediaLogicalWin(var byte bpvSuccess)

The *bFormatMediaLogicalWin* subroutine formats a storage media (f.e. SmartMedia) using the settings preferred by the Windows own formatting routines.

This subroutine returns in *bpvSuccess* TRUE on success, and FALSE on error.

Subroutine:
sub bFormatMediaLogical(var byte bpvSuccess)

The *bFormatMediaLogical* subroutine formats a storage media (f.e. SmartMedia) using the settings recommended by the SSFDC Forum.

This subroutine returns in *bpvSuccess* TRUE on success, and FALSE on error.

Example

Example: “file_format.tig”

2.15 Synchronizing the File System

Synchronise file system with RAM

For efficiency reasons, some intensively used data structures of the FAT file system are temporarily stored in the RAM memory while the file system operations are performed. Before the permanent storage media (f. e. SmartMedia) is unplugged, all the data structures must be copied from the RAM to the permanent storage media. The process of copying the data is named “synchronization”. The synchronization may be performed either by calling the *vSynchronizeFS* subroutine explicitly or by implementing a task, that sets a value of the synchronization timeout using the *lSetSyncTimeout* subroutine and calls the *bSynchronizeFSRegularly* subroutine in the endless loop.
The synchronization timeout values are measured in seconds.

Subroutine:
sub vSynchronizeFS()

The *vSynchronizeFS* subroutine writes all data structures that were temporarily saved in the RAM to the media .

Subroutine:
sub lGetSyncTimeout(var long lpvSyncTimeout; var long lpvCurSyncTimeoutCounter)

The *lGetSyncTimeout* subroutine returns the recently set synchronization timeout value in the *lpvSyncTimeout* and the current value of the timeout counter in the *lpvCurSyncTimeoutCounter.*

If the timeout values have not been yet initialised, the *lGetSyncTimeout* subroutine returns –1 in both *lpvSyncTimeout* and *lpvCurSyncTimeoutCounter.*

Subroutine:
<u>sub lSetSyncTimeout(long lNewSyncTimeout; var long lpvPrevSyncTimeout)</u>

The *lSetSyncTimeout* subroutine sets the new synchronization timeout value to the *lNewSyncTimeout* value.

The *lSetSyncTimeout* subroutine returns the previously set synchronization timeout value in the *lpvPrevSyncTimeout* or –1 if it has not been initialised yet.

Subroutine:
<u>sub bSynchronizeFSRegularly(var byte bpvTimeoutReached)</u>

The *bSynchronizeFSRegularly* subroutine calls the *vSynchronizeFS* subroutine when the synchronization timeout is over.

This subroutine returns in the *bpvTimeoutReached* TRUE if the synchronisation was performed, else FALSE is returned.

Example

Example: “file_sync.tig”

3. What Must Be Done

Planned improvements

1. Some subroutines are too slow. The execution speed must be increased by means of improved algorithms or built-in functions written directly in the processor language.
2. ECC correction process for SmartMedia is not implemented at the moment.
3. Although long file names are supported, it’s not possible to differentiate files with identical first 6 characters.
4. The information about errors is very scanty. The error messages must be extended. Probably, something like the GetLastError subroutine will be implemented.
5. The subroutines were tested with 8Mb, 32Mb, 64Mb SmartMedia cards. Additional tests would be useful.
6. It is conceivable to use the BTFS with other kinds of storage media, not only with SmartMedia card. For example, one can implement the hardware support layer for the Basic Tiger internal user flash.
7. The BTFS subroutines are not re-entrant. It can be important to find a way to make the BTFS subroutines re-entrant without compromising on the efficiency.
8. More comments in the programs and better documentation is everyone’s most fervent wish.J

4. Useful References

Refereces for more information

1. SmartMedia Card Specifications:

   http://www.ssfdc.or.jp/english/index.htm

2. About FAT:

   http://averstak.tripod.com/fatdox/00dindex.htm

   http://msdn.microsoft.com/

SMS Routines

Function library: sms_Vxxx.inc

Sending and receiving text messages

Function: Provides functions which allow for sending a text message respectively reading saved text messages via a mobile phone or a modem.

SMS basics

Almost everybody owns a mobile phone nowadays. Besides making phone calls the probably even more popular way to use a mobile phone is to write text messages. We send a message via the mobile phone network and the receiver is able to read the message within seconds. So there is a “transmitter” and a “receiver”. Usually both are human beings which are able to communicate this way.

What is SMS?

But what exactly is SMS (Short Message Service)? For the user a text message sent via SMS is a short message which he can read on his mobile phone. And that’s it in fact, because most information can be displayed by the mobile phone. At the top of the message the sender and the time the message was sent are displayed. To put it simply, a text message is a data packet, which is not sent via a serial interface or a network, but via the mobile phone network.

It is the task of the “transmitter” to format the transmitted data correctly and to send them to a kind of centre. This centre evaluates the message and forwards it to the actual receiver.

It is the “receiver’s” task to receive these data and to evaluate them . The arrival of a new text message has to be signalised.

The tasks of both “transmitter” and “receiver” can be adopted by the Tiger with little effort. This kind of communication has the advantage that it is simple and it can be used almost around the world.

Applications and requirements

Applications for SMS control could be e.g. a kind of alarm system which sends a text message to a given number when a certain state occurs. You could also think of remote controls using SMS. Controlling over an arbitrary distance with certain commands would be possible. There are many options in this area.

Requirements for such an application are a data capable phone which is able to send text messages or a SMS/GRPS modem. If you use a modem, you can connect it directly to the Tiger serial interface with a zero modem cable. Of course, the mobile phone is the cheaper alternative. If you are interested in further information on coupling Tiger and mobile phone, please see the application note no. 56 – “BASIC-Tiger and SMS”. It provides detailed descriptions. The application note can be downloaded on “www.wilke.de”.

Functions of sms_Vxxx.inc:

Encoding text messages for transmitting

- sendSMS: This function is used for simply sending text messages to a given number. The function has 2 parameters:
  - ○ N$: This first parameter is a number, to which a text message is to be sent. This has to be transferred in a string "+491631234567". So the international access code first, here: "+49" for Germany. This is followed by the phone number without the first "0".
  - ○ T$: This string contains the text to be send. The text is displayed the same in the message. The only restriction is that the text must not be longer than 160 characters, since a text message is limited to 160 characters by default.

  After sending a text message it also makes sense to check the reply of the mobile phone/modem, whether the text message was successfully sent. You have to wait until the buffer of the serial interface is full, since the Tiger and the mobile phone communicate via this interface. The buffer is read and the content is compared to the different options. The following events can occur:
  - ○ +CMGS:<mr>[,ackpdu>] OK --> successfully sent
  - ○ +CMS ERROR:<err> --> error

  Parameters:
  - ○ <mr>: GSM 03.40 TP-message-reference in integer format
  - ○ <ackpdu>: User data element of the PDU string
  - ○ <err>: The error occurring

Decoding read text messages

- computeRcvdPdu: This function is used for decoding a text message in PDU format. Information is filtered and saved in different variables. So if a text message was received, this function should be called to filter the right data.

  Parameters:
  - ○ string$: Simply transfer the received PDU string unaltered to this parameter.
  - ○ return$: This variable is the return value of this function. After completing the function the text message is saved in text modus here, i.e. the text which you can read on your mobile phone. This string is, of course, also limited to 160 characters.

  The remaining data are saved in the global variables:
  - ○ rcvdSMSC$: Number received from service centre
  - ○ rcvdSenderNum$: Sender
  - ○ rcvdPduTxt$: Message received in text mode

  In addition there are also functions which can be used for reading these variables:
  - ○ getRcvdPduSMSC: Get number from service centre
  - ○ getRcvdSenderNumber: Get sender
  - ○ getRcvdePduTxt: Get message text

In this function only the most important information is read - not all information. If more data are needed, the function can be arbitrarily expanded. The places, in which a code can be entered, are already applied. You just have to search the position in the code, on which the information is deleted. The comment states exactly, which data are deleted there.

How do SMS functions work?

The function's functionality

The functionality of both functions is relatively complex, since the encoding in the PDU mode is not simple. If you are still interested and want to understand these functions, you may continue reading. First of all you have to deal with encoding in detail. What exactly is the PDU mode? Actually it is only a code similar to e.g. the ASCII code.

A PDU message has always the same structure. It could be as follows:

0791947122723033250000C91947112325476000000BD4F29C4E2FE3E9BA4D19

We will deal with the single elements in detail, in order to explain how to handle such a string.

07:
This is the length of the provided address data (i.e. the length of the SMSC address) in bytes.

91947122723033:
Address data. This part has to be divided again:
1. byte: 91
There are 2 options:
91: for numbers in the international format
81: for numbers in the national format with area code or numbers without area code.
From 2. byte: 947122723033
The remaining bytes contain the number of the SMSC, which is to be used. The number is BCD-encoded, i.e. 4 bits make for one cypher of the number. The number begins in the second half byte of the first byte. The possibly unused half byte at the last cypher always has to contain a 0xF.

Bits 7 6 5 4	Bits 3 2 1 0
second cypher	first cypher
fourth cypher	third cypher
sixth cypher	fifth cypher
...	...
1 1 1 1	last cypher

+49172123456 would be: 07919471123254F6. 1212 would be: 03812121

Important: If there is a SMSC number configured in the phone, you can omit it when sending the message. In this case you enter 00 at the beginning, i.e. the number has the length 0. The mobile phone interprets it correctly and replaces it with the standard number.

25:

This byte contains 6 parameters:
Bit 0 + 1: Message Type Indication: 01 for 'SMS-SUBMIT MS to SMSC'
Bit 2: Reject Duplicates: 0 for 'off', 1 for 'on'
Bit 3 + 4: Validity Period: 00 if there is no VP field
Bit 5: Status Report Request: 0 for 'off', 1 for 'off'
Bit 6: User Data Header Ind.: 0 for 'no UDH'
Bit 7: Reply Path: 0 for 'off', 1 for 'on'

00:

Reference number of message. If no reference number is given, i.e. 00, the mobile phone/modem itself assigns one. Notices of receipt or error messages contain this reference number and therefore can be assigned to the original message.

0C:

This is the length of the destination phone number, i.e. the number to which the message is sent (uncoded, in cyphers!). However, sound is not included in the length yet.

91947112325476:

This is the phone number which is encoded exactly as the SMSC number, i.e. the figures are swapped in pairs.

00:

Protocol Identifier: The number identifies the protocol. Here it is always possible to enter 00, which is the default value.

00:

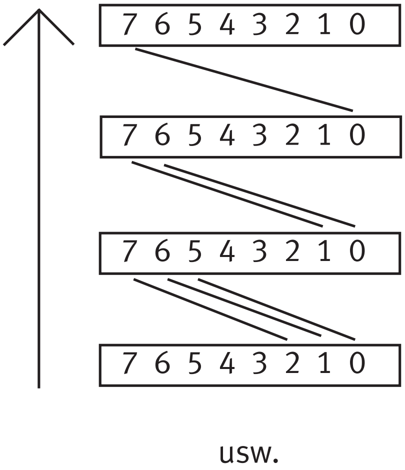
Data Coding Scheme: The byte tells the phone, how to decode the data. Here, however, it is also possible to enter 00 in each case.

0B:

This is the length byte of the user data, i.e. the length of the message, or, to be exact, the length of the message text. This length refers to the number of characters in the text after decoding.

D4F29C4E2FE3E9BA4D19:

This is the actual text which encodes the SMS text to the PDU mode. The text actually encoded in 7 bits is now changed to an 8 bit format, which can save some bytes in longer texts. In case of 160 characters 20 bytes are saved.
The bits are placed according to the scheme below. The least significant bits of a character are placed at/shifted to the highest free positions of the previous byte. Like this positions remain free in this byte, but on the first positions. So the more significant bits are moved to the right, to clear a space for the next bits on the higher position.



Once you understand this system, you should also be able to comprehend the algorithm.

When decoding the PDU message, you go through all information and analyse it. The function “decodeTelNum” can decode phone numbers and the SMSC by swapping entries in pairs and saving the numbers which result from this.

SMS Routines

Now the function “smsPduToTxt” alters the text to real text mode by placing bits on the correct position respectively re-establishing the 7 bit characters. First of all 2 bytes are converted to one. The Tiger receives e.g. CD = 1 x C & 1 x D, but we need a byte of the value 0xCD. So we create a string, which is structured exactly like this. Every byte has to be mirrored, in order to bring the bits in the right order. The different bits are placed respectively read. The 7 bit character values are saved in an array of bytes. This array is converted to a string by assigning the according character to every byte later.

The function “sendSMS” encodes the text to be sent exactly as presented above. The 7 bit ASCII characters are saved in a character string, in which the bits are positioned according to this scheme. To implement this, the original text is saved binary in a string at first. We only use a large string, one byte of which adapts the value of one bit, which is either 1 or 0. This allows accessing quickly via the index. The text is saved in this string, however in a 7 bit format, bit 8 is left out (it is always 0 anyway). After converting the text to PDU format the phone number is processed: The numbers are swapped and a “F” is added at the and, if the the last half byte stands alone. If a “+” is part of the phone number , it is replaced with a 19, otherwise an 18. The 19 becomes 91 after decoding and means that international numbers are used.

Examples

There are two example programs, which use these functions to transmit a text message via mobile phone:

“SendSMS.tig” sends a text to a destination phone number, both the text and the phone number of the receiver being defined as constants in the program header.

“ReceiveSMS.tig” reads the first message saved on your mobile phone. If no message was saved, the program could e.g. output an error message. The according sector remains empty by default, i.e. nothing happens at all.

empty page

Index

empty page

Index

Alphabetical index

empty page

Notes

Notes