Update
Tiger Basic™ 5.2

Neue Funktionen in Version 5.2

Produktion: Wilke Technology GmbH
Heider-Hof-Weg 23D
D-52080 Aachen, Germany
www.wilke-technology.de
info@wilke-technology.de

Umschlag: Der Design Pool, www.derdesignpool.de
Druck: Print Production, www.printproduction.de
Ausgabe: 1. Auflage Juli 2004
Art-Code: BTI-MANUG5.2-1

Inhaltsverzeichnis

Zu diesem Buch

leere Seite

Vorbereitungen

Um mit der vorliegenden neuen Version der Tiger-Basic IDE arbeiten zu können benötigen Sie einen PC als Entwicklungsrechner sowie ein daran angeschlossenes BASIC-Tiger Zielsystem.

Systemvoraussetzungen PC

Für den Einsatz der Tiger-Basic IDE 5.2 wird ein PC mit folgender Mindestausstattung empfohlen:

- Pentium Prozessor 500 Mhz oder schneller
- Festplatte mit mindestens 40 MByte freiem Speicher
- SVGA Grafik (800 x 600) oder höhere Auflösung
- Maus
- Windows 2000 / Me oder neuere Version
- 1 freier COM-Port (COM1: ... COM4:)
- CD-ROM-Laufwerk

BASIC-Tiger Zielsystem

Als Zielsystem kann jedes Tiger Proto-Board oder jedes Gerät mit einem BASIC-Tiger Computer eingesetzt werden, das über eine RS-232 Schnittstelle (Ch-1) verfügt und das in den PC-Mode versetzt werden kann.

Es wird empfohlen mindestens folgende Speicherausstattung zu verwenden:

SRAM	128 KByte
FLASH	512 KByte

Sicherheitshinweis

BASIC-Tiger Computer Module werden in unterschiedlichsten Anwendungen, Geräten und Maschinen als „Embedded Computer“ eingesetzt. Hier übernehmen sie typischerweise Aufgaben wie Kommunikation, Bediener-Interface, Überwachungsfunktionen und Geräte- und Anlagen-Steuerungen.

In jedem Fall muss sichergestellt sein, dass solche Installationen sich in einem gesicherten Zustand befinden wenn an ihnen Programmtests durchgeführt werden, die Programm-Ausführung gestoppt und/oder gestartet wird. Der Anwender hat in jedem solchen Fall geeignete Vorkehrungen zur Vermeidung von Sach- und/oder Personenschäden zu treffen - beispielsweise angeschlossene Anlagen in energielosen Zustand zu bringen oder nicht mit echten, ev. vertraulichen Datenbeständen zu arbeiten.

Ferner gibt es zu dieser IDE zahlreiche Entwicklungs-Tools wie Prototyping Boards, Entwicklungs-Kits, Adapter, Kabel und Zubehörteile. Solche Komponenten dienen nur als Arbeitshilfe bei der Entwicklung und Erprobung von Computer-Software und Hardware-Schaltungen. Ihr Zweck ist, ausschließlich dem sachkundigen Entwickler bei der Anfertigung von Laboraufbauten Zeit zu ersparen und Anregungen für eigene Designs zu geben.

In keinem Fall dürfen diese Komponenten von Nicht-Fachleuten installiert oder betrieben werden. Ebenso ist es nicht vorgesehen, daß derartige Laboraufbauten zur Steuerung von Anlagen und Geräten eingesetzt werden, insbesondere, wenn deren Fehlfunktion zu Schäden oder Gefahren führen können. In keinem Fall dürfen mit diesen Laboraufbauten hohe Spannungen oder Ströme gehandhabt werden.

Bei allen Änderungen an Schaltungen sowie beim Öffnen von Gehäusen, sind die Geräte bzw. Versuchsaufbauten vollständig von der Stromversorgung zu trennen. Empfindliche Bauteile wie CMOS Schaltkreise sind zur Vermeidung von Beschädigung nur in einer antistatisch ausgestatteten Laborumgebung zu handhaben.

Typografische Konventionen

Die folgenden Schriften und Symbole werden verwendet, damit Sie wichtige Informationen schnell erkennen können:

Element	Bedeutung
`Programmlisting`	Tiger-BASIC Programmlisting
Instruktion / Funktion	Tiger-BASIC Instruktion / Funktion
B W L S F Parameter 1 ● ● ● - - Parameter 2 - - - ● - Parameter 3 - - - - ●	Parameter-Beschreibung: ● Datentyp erlaubt - Datentyp nicht erlaubt B Datentyp BYTE W Datentyp WORD L Datentyp LONG S Datentyp STRING F Datentyp REAL (Floating Point)
Anmerkung	Anmerkungen / Hervorhebungen

Instruktionen & Funktionen

leere Seite

Kommentierte Funktionsübersicht

Übersicht über die in Version 5.2 hinzugekommenen Funktionen:

```
Res$ = BFlag_Tab$  ( These_Chars$, Is_Char_Flg, Is_NOT_Char_Flg )
```
Erzeugt Flag-Tabellen in einem String von konstant 256 Byte Länge

```
ADR = Bit_Map_Adjust  ( Bitmap$, Record_Size, ADR, Value )
```
Korrigiert eine Data-Block-Adresse entsprechend einer (In)valid-Block Bitmap

```
Num = Bit_Map_Cnt  ( Bitmap$, Value )
```
Zählt die „0“-Bits oder „1“-Bits einer 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
```
Liest ein Bit aus einer Bitmap / schreibt Bit in Bitmap

```
R$ = Check_Keyword$  ( Src$, Key_Word_List$, Sep$, Start_Pos, Tolerance )
```
Untersucht Strings nach Keywords - z.B. nach Kommando-Worten und Argumenten

```
Found_Ptr = Chk_Fnam  ( Src$, Start_Pos, Stepwidth, FName$, Method )
```
Untersucht einen String auf das Vorhandensein eines Filenamens

```
Concentrate$  ( Source_Destin$, Initial_Skip, Take, Next_Skips )
```
Fasst Daten-Felder zusammen die z.B. in Record-Form in Strings vorliegen

```
ERG$ = Conv_Base64$  ( Source$, Methode )
```
Konvertiert Daten von / nach Base64-Format

```
Count_Patt$  ( Src$, Patt$, Patt_ELen, Pos, Cnt_Len )
```
Untersucht String auf das Vorhandensein von vorgegebenen Byte-Mustern

```
CRC  ( A$, Pos, Len, Start_Factor, Step, CRC )
```
Bildet eine CRC Prüfsumme aus den Daten eines String

```
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
```
Zusätzliche „Cut_and_Paste“ Funktionen für String, Num und Real Variablen

```
ECC$ = Calc_ECC  ( Data$, Start_Pos, ECC_Method )
```
Berechnet einen ECC (Error-Correction-Code) für einen Daten-Block

```
Flag = Correct_ECC  ( Data$, Start, ECC_stored, ECC_calculated, ECC_Method )
```
Korrigiert einen Data-Block mit eventuellem Parity-Fehler

```
Pos_Ndx$ = Find_opt_Group$  ( Num$, Group_Ndx_List$, Size_of_Group,
    Size_of_Nums, Target_Val, Tolenrance_Band, Start_Pos, Target_Tol )
```
Stellt optimale Gruppen von Fächern zusammen, die unterschiedliche Inhaltsmengen besitzen

I2CL_Setup (Port, Clock_Pin, Data_Pin, Speed)
Spezifiziert die für den I²C-Bus eingesetzten Tiger Pins

I2CL_Start (Speed)
Erzeugt eine Start-Condition auf dem I2C-Bus / ISO-7816 Kanal

I2CL_Stop (Speed)
Erzeugt eine Stop-Condition auf dem I2C-Bus / ISO-7816 Kanal

I2CL_Release (Dummy)
Schaltet beide Bus-Leitungen in Hi-Impedance Zustand (ohne Stop-Condition)

```
A$ = I2CL_Read$ ( nob )                 ' Read Byte(s) from I2C-Bus
A$ = I2CL_Read$ ( nob, 7816 )           ' Read Byte(s) from ISO-7816 Bus
```
Liest von I2C-Bus / ISO-7816 Bus die spezifizierte Anzahl von Bytes ein

```
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
```
Schreibt die spezifizierte Anzahl Bytes auf I2C-Bus / ISO-7816 Bus

Flag = I2CL_Result ( )
Liest den Ergebnis Code der zuletzt ausgeführten I2CL-Funktion

N$ = Insert$ (Source$, S_Pos, Ins$, I_Pos, I_Len)
Fügt einen String (oder Teil davon) in einen String ein

```
N = Key_Direct ( Port, Column, Mode, Speed )    ' Lies Tasten-Spalte
N = Key_Direct ( Port, Column, Mode )           ' Lies Tasten-Spalte
```
Liest bis zu 16 Tasten oder Schalter direkt über den Daten-Bus

A = BLookup# (Index)
A = WLookup# (Index)
A = LLookup# (Index)
Schneller Zugriff auf Daten in „Look Up“ Tabellen in DATA-FLASH Area oder Strings

A = Pin (ADR, Bit_Pos)
Liest einen einzelnen Pin eines Ports ein

```
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$, Len )                   ' Pop a String from A$
```
Daten (Byte, Word, Long, Real, String) auf String PUSH-en und ebenso wieder vom String POP-en => zu Byte, Word, Long, Real, String

Graphic_Reformat (Src$, Dest$, Width, High, Re_Format)
Kehrt die Pixel-Struktur einer Pixel-Grafik um: waagerecht => senkrecht

Dest$ = Text_Reformat (Source$, Dest_Width, Dest_Cut_Wrap, Dest_Fill_Blank, Dest_Line_End)
Reformatiert ASCII-Text Strings für Ausgabemedien wie Terminals, LCDs, Drucker etc.

New_Pos = Scan_or_Skip (Src$, Charset$, Pos, Scan_Skip)
Sucht oder überspringt in Strings vorgegebene Zeichen-Kollektive

FLG = SPI_Setup (Clk_MOSI_Port, Clk_Pin, MOSI_Pin, SSI_Port, SSI_Pin, MISO_Port, MISO_Pin, MSB_First)
Spezifiziert den SPI-Bus für die Funktion: SPI_IO (....)

Rec$ = SPI_IO$ (Transm$)
Sendet und empfängt gleichzeitig Daten über SPI-Bus wie in SPI_SETUP spezifiziert

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 Konverter mit Such-String Liste und Replace-String Liste

Data$ = Unpack_DC$ (Ctrl$, Src$, Pos, Len, Flag, Method)
Dekomprimiert einen Source-Datenstrom und trennt ihn in 2 Kanäle auf: DATA-Kanal und CTRL-Kanal

Err_Code = Update_Me (Data_Label, Option)
Ersetzt ein bestehendes Programm durch eine nachfolgende Programm-Version (Update) und startet dieses Programm

Flag = XSetup (Bus_Port, Ctrl_Port, Bit_ACLK, Bit_DLCK, Bit_INE, Ctrl2_Port, Bit_Bus_CE)
Definiert Bus- und Ctrl-Signal-Leitungen für das XPort System

N = Xin (ADR) ' ‹ADR› ‹data›
Liest 1 Byte von I/O Erweiterungs-Port (XPort) „ADR“ ein

X$ = Xin$ (ADR, Anz) ' ‹ADR› ‹data› ‹ADR+1› ‹data› ...
X$ = Xin$ (-ADR, Anz) ' ‹ADR› ‹data› ‹data› ‹data› ...
X$ = Xin$ (256+, Anz) ' ‹data› ‹data› ‹data› ...
Liest Erweiterungs-Port ein mit verschiedenen Bus-Zugriffsarten

Xout (ADR, N) ' ‹ADR› ‹data› - write 1 Byte
Schreibt 1 Byte auf den I/O Erweiterungs-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
```
Schreibt in verschiedenen Modes auf I/O Erweiterungs-Ports (XPort)

```
XSet  ( ADR, Bit_Pos )               ' setze 1 Bit in XPort ADR: 00 ... FF
```
Setzt ein Bit eines XPort-Ausgangs „high“ = 1

```
XRes  ( ADR, Bit_Pos )               ' reset 1 Bit in XPort ADR: 00 ... FF
```
Setzt ein Bit eines XPort-Ausgangs „low“ = 0

```
XInv  ( ADR, Bit_Pos )               ' toggle 1 Bit in XPort ADR: 00 ... FF
```
Invertiert ein Bit eines XPort-Ausgangs

```
A = XPin  ( ADR, Bit_Pos )           ' lies 1 Bit in XPort ADR: 00 ... FF
```
Liest einen einzelnen Pin eines XPort-Eingangs ein

BFlag_Tab$ (i)

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

Funktion: BFlag_Tab$ erzeugt Flag Tabellen in einem String von konstant 256 Byte Länge. Jedes Byte des Flag-Strings repräsentiert einen Zeichen-Code entsprechend seiner Position im Flag-String.

BFlag_Tab$ arbeitet auf 3 verschiedene Weisen, die erste (i) ist:

Jedes Byte im String kann einen Flag-Wert von 2 möglichen enthalten, „Is_Char_Flg“ oder „Is_NOT_Char_Flg“ deren Position im String durch die Zeichen-Codes im Parameter „These_Chars$“ festgelegt wird

Beispiel:

Hex Tabelle des entstehenden Flag-Strings Res$

```
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 = Flagwert für Codes die zum Zeichensatz gehören,
„77“ = Is_NOT_Char_Flg = Flag für Codes, die NICHT zum Zeichensatz gehören.

Ein Flag-String wird immer dann gebraucht, wenn es um schnelle Konversionen, Filter, Maskierungen, Search, Scan, Skip und ähnliche Vorgänge geht. Ein Beispiel findet sich im Zusammenspiel mit der Scan_or_Skip Funktion dort.

BFlag_Tab$ **Byte Flag Tabelle aufbauen**

Parameter:

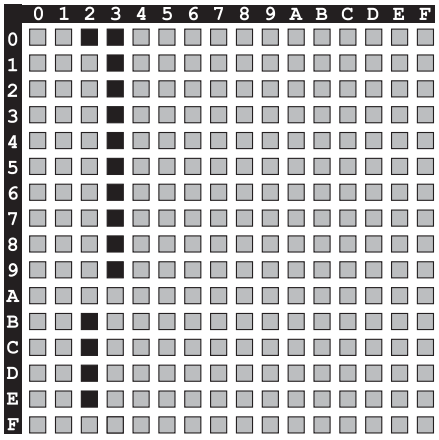
	B	W	L	S	F	
These_Chars$	-	-	-	●	-	String mit allen Zeichen, die zum Charcater-Set gehören sollen
Is_Char_Flg	●	●	●	-	-	Flag-Wert zur Kennzeichnung von Codes in „Res$“, die zum Zeichensatz lt. Definition in „These_Chars$“ gehören (i) oder 256 (ii) oder 256 + Offset (iii)
Is_NOT_Char_Flg	●	●	●	-	-	Flag-Wert zur Kennzeichnung von Codes in „Res$“, die *NICHT* zum Zeichensatz lt. Definition in „These_Chars$“ gehören
						Funktionswert:
Res$	-	-	-	●	-	Flag-String, genau 256 Byte lang (wenn ausreichend deklariert). Jedes Byte des String ist ein Flag mit einem der beiden möglichen Werte: „Is_Char_Flg“ oder „Is_NOT_Char_Flg“.

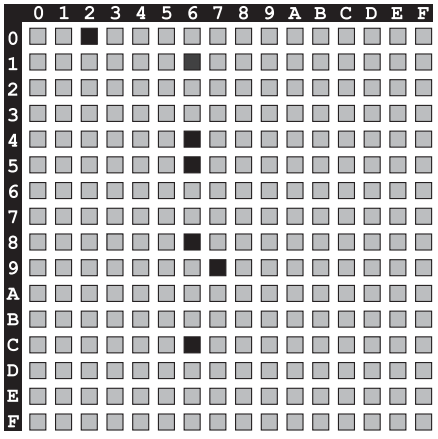
2 Beispiele:

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

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

Flag Verteilung in R$ nach Ausführung der BFlag_Tab$ Funktion





BFlag_Tab$ (ii)

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

Funktion: BFlag_Tab$ erzeugt hier (ii) einen Flag String bei dem jedes Flag-Byte, das ein Zeichen des Zeichensatzes markiert durch seinen eigenen Code gebildet wird. Die Zeichen die **nicht** zum Zeichensatz gehören, werden wie bei (i) durch den von „Is_NOT_Char_Flg“ vorgegebenen Code gebildet.

Beispiel:

```
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
```

Hex Tabelle des entstehenden Flag-Strings Res$

‹Code› = Flagwert für Codes die zum Zeichensatz gehören,

"." = ‹2EH› = Is_NOT_Char_Flg = Flag für Codes, die NICHT zum Zeichensatz gehören.

BFlag_Tab$ (iii)

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

Funktion: BFlag_Tab$ erzeugt hier (iii) einen Flag String bei dem jedes Flag-Byte, das ein Zeichen des Zeichensatzes markiert durch seinen eigenen Code + dem Wert von „Offset“ gebildet wird. Die Zeichen die **nicht** zum Zeichensatz gehören, werden wie bei (i) und (ii) durch den von „Is_NOT_Char_Flg“ vorgegebenen Code gebildet.

Beispiel:

```
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
```

Hex Tabelle des entstehenden Flag-Strings Res$

‹Code› = Flagwert für Codes die zum Zeichensatz gehören,

"." = ‹2EH› = Is_NOT_Char_Flg = Flag für Codes, die NICHT zum Zeichensatz gehören.

BFlag_Tab$ — Byte Flag Tabelle aufbauen

BFlag_Tab$ kann z.B. dazu eingesetzt werden die in einem Text vorkommenden Zeichen zu erfassen und durch Flags in einem Flag-String zu kennzeichnen.

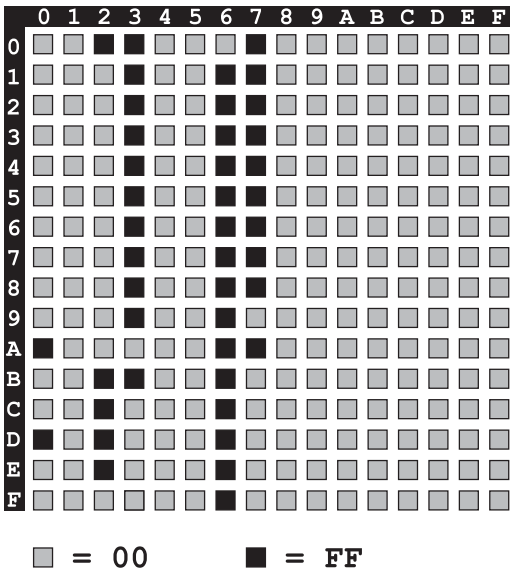
Dazu ein Beispiel:

```
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                                      '
```

... was zu folgendem Flag-String Aussehen führen könnte:

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

Beispielhafte Flag Verteilung in Flag_Total$ nach dem Ausstieg aus dem Endlos-Loop



Diese Tabelle sagt aus, dass im Text nur Kleinbuchstaben (kein "y"), einige Satzzeichen und ‹CR› und ‹LF› Codes vorkommen.

Bit_Map_Adjust

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

Funktion: Korrigiere eine Data-Block Adresse entsprechend einer Invalid-Block bzw. Valid-Block Bitmap.

Anwendung: Verwaltung von Speicherblöcken, FAT-Systeme, RAM-Disks, FLash-Disks, ...

Parameter:

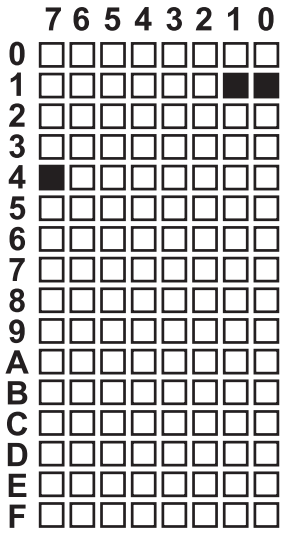
	B	W	L	S	F	
Bitmap$	-	-	-	●	-	Bitmap, je 1 Bit ist ein Flag für einen Speicherblock von der Größe wie in Parameter „Record_Size“ angegeben. In Bitmap sind gültige bzw. ungültige Speicherblocks markiert (siehe „Value“).
Record_Size	●	●	●	-	-	Block-Größe: 1 ... 7FFFH Bytes
ADR	●	●	●	-	-	Speicher-Adresse: 0 ... 7FFF FFFFH
Value	●	●	●	-	-	0: „0“-Bit kennzeichnet „valid Block“ 1: „1“-Bit kennzeichnet „invalid Block“
						Funktionswert:
ADR	●	●	●	-	-	ggf. korrigierte Adresse auf die nächste gültige Block-Adresse: 0 ... nnnnnnnn ADR mit oder ohne Korrektur -1 Fehler: Bitmap$ = leer -2 Fehler: Record-Size unzulässig -3 Fehler: ADR liegt (ev. nach Korrektur) hinter ADR-Ende

Bit_Map_Adjust **Adjust ADR entsprechend Bitmap**

Beispiel:

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

Bitmap$ zeigt 3 „Invalid Blocks“:



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

Der Eingangs-Wert von ADR = 8000H liegt lt. Bitmap in einem „invalid Block“. Durch Bit_Map_Adjust wird die Adresse ADR auf den nächsten gültigen Block verschoben: A000H.

Bit_Map_Adjust wird eingesetzt um Speicherblocks in Massenspeichern zu verwalten. In einer Bitmap werden typischerweise ungültige / gültige Blocks verwalten oder freie / belegte Blocks.

Ein Einsatz dieser Funktion findet sich in den Applikationen des FAT-File Systems mit Smart-Media Speichermedien.

Bit_Map_Cnt

```
Num = Bit_Map_Cnt (Bitmap$, Value)
```

Funktion: Zählt die „0“-Bits oder „1“-Bits einer Bitmap

Anwendung: Verwaltung von Speicherblöcken, FAT-Systeme, RAM-Disks, FLash-Disks, ...

Parameter:

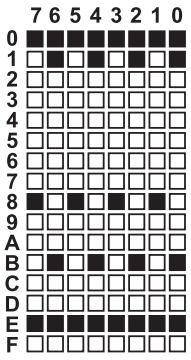
	B	W	L	S	F	
Bitmap$	-	-	-	●	-	Bitmap, je 1 Bit ist ein Flag für einen Speicherblock von der Größe wie in Parameter „Record_Size“ angegeben. In Bitmap sind gültige bzw. ungültige Speicherblocks markiert (siehe „Value“).
Value	●	●	●	-	-	0: „0“-Bits zählen X: „1“-Bits zählen
						Funktionswert:
Num	●	●	●	-	-	0 ... nnnnnnnn Anzahl der gezählten Bits -1 Fehler: Bitmap$ ist leer

Beispiel:

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

Nach Ausführung o.g. Sequenz ist Num der Wert 28 zugewiesen.

Dies entspricht der Anzahl von „1“ Bits in Bitmap$.



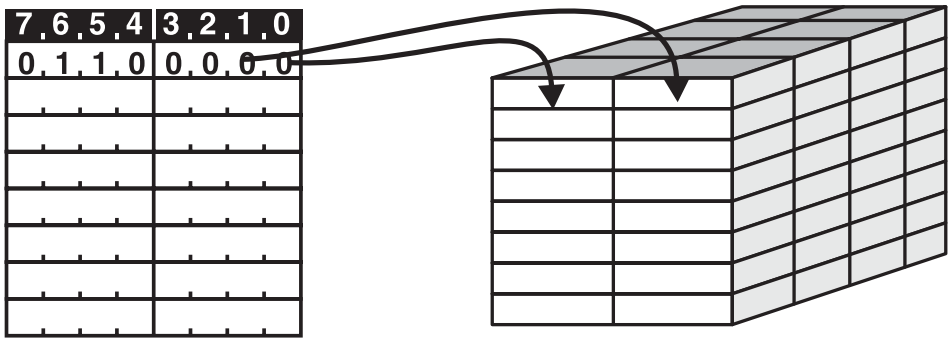
Bit_Map_RD
Bit_Map_WR

```
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
```

Lese und schreibe Bits in Bitmap

Funktion: Lies ein Bit aus einer Bitmap / schreibe Bit in Bitmap.

Bitmaps eignen sich z.B. zur Verwaltung von Speichermedien wie Disks, und RAM oder FLASH Massenspeicher. Je 1 Bit entspricht im Massenspeicher ein Block, Sektor oder Cluster der entweder belegt oder frei, gültig oder defekt sein kann.



Bitmap
1 Bit

=>

Massen-Speicher Organisation
1 Block, Sektor, Cluster, ...

Bit_Map_RD
Bit_Map_WR

Read + Write Bitmap

Parameter:

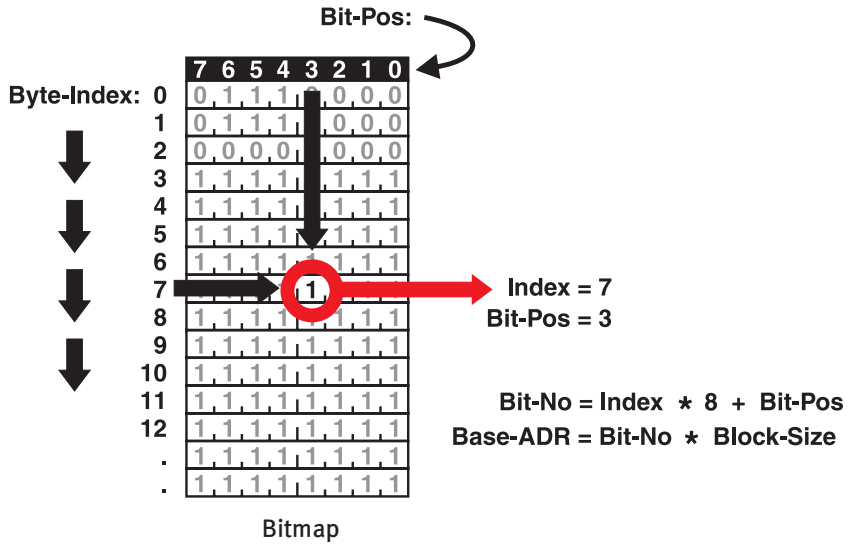
	B	W	L	S	F	
BMap$	-	-	-	●	-	Bitmap String, Ein- und Ausgang. 1 Bit entspricht 1 Speicherblock von der Größe wie mit „Blk_Size“ definiert.
Blk_Size	●	●	●	-	-	Größe eines Speicher Blocks (Sektors, Clusters, ...) in Bytes, Werte: 1 ... 7FFFH
ADR	●	●	●	-	-	Adresse in den Massenspeicher: 0 ... nnnn nnnn
Val	●	●	●	-	-	Bit-Wert, der in die Bitmap eingetragen werden soll: 0 = „0“ Bit schreiben X = „1“ Bit schreiben
						Funktionswert:
N	●	●	●	-	-	Bisheriger Wert des entsprechenden Bits aus Bitmap: 0 = Bit ist/war =0 1 = Bit ist/war =1 -1 = Fehler: BMap$ = leer -2 = Fehler: Block-Size unzulässig -3 = Fehler: ADR nicht in dieser Bitmap enthalten

Der Funktionswert steht in jedem Fall zur Verfügung, beim Lesen (Bit_Map_RD) als auch beim Schreiben (Bit_Map_WR). Beim Schreiben gibt N den Bit-Wert an, der vor dem Schreibvorgang an dieser Stelle vorgelegen hat.

Bit_Map_RD
Bit_Map_WR

Read + Write Bitmap

Der Zusammenhang zwischen Bit-Position in Bitmap (Bit-No), Blockgröße (Block-Size) und Speicher-Adresse (Base-ADR):



Beispiel:

```
Block-Size = 16 kByte =  4000H

Byte-Index:    Bit-Pos:     Bit-No:     Block-No:         Block Start-ADR:

     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

Disks und andere Massenspeicher-Systeme benutzen Bitmap-Tabellen um belegte und freie Blöcke zu kennzeichen. Statt im entsprechenden Speicherblock eine Kennzeichnung vorzunehmen, wird nur das zugehörige Bit in einer Bitmap (z.B. Record-Allocation-Table) entsprechend gesetzt.

Bit_Map_RD und Bit_Map_WR vereinfachen und beschleunigen diese Zugriffe. Beispiele für die Nutzung dieser Funktionen finden sich u.a. in der Applikation: FAT-System für Smart-Media FLASH Cards.

Hier zwei kurze Beispiele über die Wirkungsweise der Funktionen:

Beispiel-1: Lies ein Bit aus Bitmap entsprechend Block_Size und ADR:

```
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")
```

Beispiel-2: Schreibe ein Bit in Bitmap und lies den Wert dieses Bits vor der Schreiboperation aus:

```
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)

Funktion: Untersucht Strings nach Keywords - z.B. nach Kommando-Worten und Argumenten.

Anwendungen: Textanalysen, Kommando-Interpreter, Fernsteuerungen per SMS, e-mail, Sicherheits-Anwendungen, ...

Parameter:

	B	W	L	S	F	
Src$	-	-	-	●	-	Eingangs-String in dem das nächste gültige Schlüssel-Wortes gesucht werden soll.
Sep$	-	-	-	●	-	Separator-Flag String: genau 256 Byte langer String mit Flags: 00 = dieser Code ist KEIN Separator / Trenner im Source-String XX = dieser Code ist ein Separator / Trenner im Source-String Dieser Flag-String markiert jene Zeichen-Codes in Src$, die als Trenner zwischen Keywords verwendet werden. Bei der Suche nach Keywords, werden diese Zeichen übergangen, diese Zeichen sind niemals Bestandteil eines Keywords. Falls Sep$ = "" (leer) => keine Trenner-Zeichen in Src$ suchen.
Key_Word_List$	-	-	-	●	-	String mit einer Liste von Schlüssel-Worten. Format: "‹sep›Keyword_1‹sep›Keyword_2‹sep›..." z.B.: ".schalte.schalt.mache.mach.aus.off.zu.weg."
Start_Pos	●	●	●	-	-	Start-Position in Src$ ab der nach Schlüssel-Worten gesucht wird: 0 ... nnnn.

Tolerance ● ● ● - - Kennzeichnet die Toleranz beim Erkennen eines Key-Words aus der Key_Word_List:

Toleranz:	0	1	2	3	4	5	6	7	8	9
Vollständige Übereinstimmung nötig:	●	○	○	○	○	○	○	○	○	○
1 Zeichen Übereinstimmung nötig:	○	●	○	○	○	○	○	○	○	○
2 Zeichen Übereinstimmung nötig:	○	○	●	○	○	○	○	○	○	○
3 Zeichen Übereinstimmung nötig:	○	○	○	●	○	○	○	○	○	○
4 Zeichen Übereinstimmung nötig:	○	○	○	○	●	○	○	○	○	○
5 Zeichen Übereinstimmung nötig:	○	○	○	○	○	●	○	○	○	○
6 Zeichen Übereinstimmung nötig:	○	○	○	○	○	○	●	○	○	○
7 Zeichen Übereinstimmung nötig:	○	○	○	○	○	○	○	●	○	○
8 Zeichen Übereinstimmung nötig:	○	○	○	○	○	○	○	○	●	○
9 Zeichen Übereinstimmung nötig:	○	○	○	○	○	○	○	○	○	●

Toleranz:	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
Keinen Fehler dulden:	●	○	○	○	○	○	○	○	○	○
1 fehlerhaftes Zeichen = OK:	○	●	○	○	○	○	○	○	○	○
2 fehlerhafte Zeichen = OK:	○	○	○	○	●	○	○	○	○	○
3 fehlerhafte Zeichen = OK:	○	○	○	○	○	○	○	●	○	○
+1 Zeichen zuviel = OK:	○	○	●	○	○	○	○	○	○	○
+2 Zeichen zuviel = OK:	○	○	○	○	○	●	○	○	○	○
+3 Zeichen zuviel = OK:	○	○	○	○	○	○	○	○	●	○
-1 Zeichen zu wenig = OK:	○	○	○	●	○	○	○	○	○	○
-2 Zeichen zu wenig = OK:	○	○	○	○	○	○	●	○	○	○
-3 Zeichen zu wenig = OK:	○	○	○	○	○	○	○	○	○	●

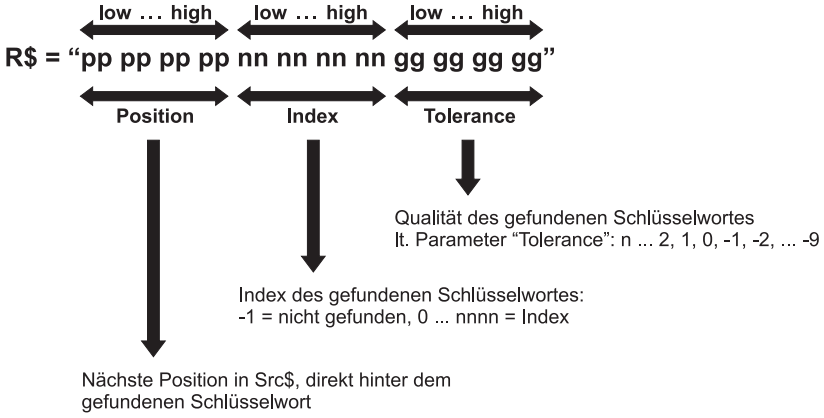
Funktionswert:

R$ - - - ● - Ergebnis-String mit Informationen über das ev. gefundene nächste Schlüsselwort.

Länge = 12 Bytes = 3 LONG Zahlenwerte:

1. LONG = Position in Src$
2. LONG = Index des gefundenen Schlüsselwortes
3. LONG = Tolerance des gefundenen Schlüsselwortes

Aufbau des Ergebnis-Strings:



Der Source-String: Src$

Src$ enthält üblicherweise:

Schlüsselworte
Separatoren
sonstige Zeichen

Zwischen Schlüsselworten wird mindestens 1 Separatorzeichen zur Trennung benötigt.

Schlüsselworte können aus 1...32 Zeichen bestehen und dürfen keine Separator-Zeichen enthalten.

Der Source-String kann beliebige Länge haben.

Key_Word_List$

Die Keyword-Liste enthält alle verwendeten Schlüsselwörter, ggf. in verschiedenen Schreibweisen und Sprachen. Die Schlüsselwörter einer Liste können unterschiedliche Länge haben und werden durch ein Trenner-Zeichen voneinander getrennt. Das 1. Zeichen des Keyword-Listen Strings wird als das Trennerzeichen verwendet. Dieses

Zeichen dient nur zur Trennung der Schlüsselwörter in der Liste, es hat keinerlei Bedeutung sonst.

Separator Flag-String

Sep$ definiert jene Codes, die in Src$ als Separator-Zeichen zwischen Schlüsselwörtern dienen sollen. Sinnvollerweise werden auch alle nicht verwendeten Zeichen als Separatorzeichen definiert.

Sep$ ist ein Flag String der immer aus genau 256 Zeichen besteht. Jedes Byte repräsentiert ein Flag für den zugehörigen Code: 00=dieser Code ist KEIN Separatorzeichen, jeder andere Flag-Wert markiert ein Separatorzeichen.

Falls der Separator Flag-String leer ist (""), bedeutet dies, dass hier ohne Separatorzeichen gearbeitet wird. Schlüsselworte können dann bündig aneinandergereiht sein (müssen es aber nicht).

Beispiel - genaue Übereinstimmung suchen

Ein Kommando String (Source-String) soll untersucht werden auf das nächste relevante Schlüsselwort. Dazu existiert eine Key-Word-Liste mit den relevanten Schlüsselwörtern. Die Start-Position für die Source-String Untersuchung ist = 0. Schlüsselworte sollen exakt im Source-String auftreten um erkannt zu werden:

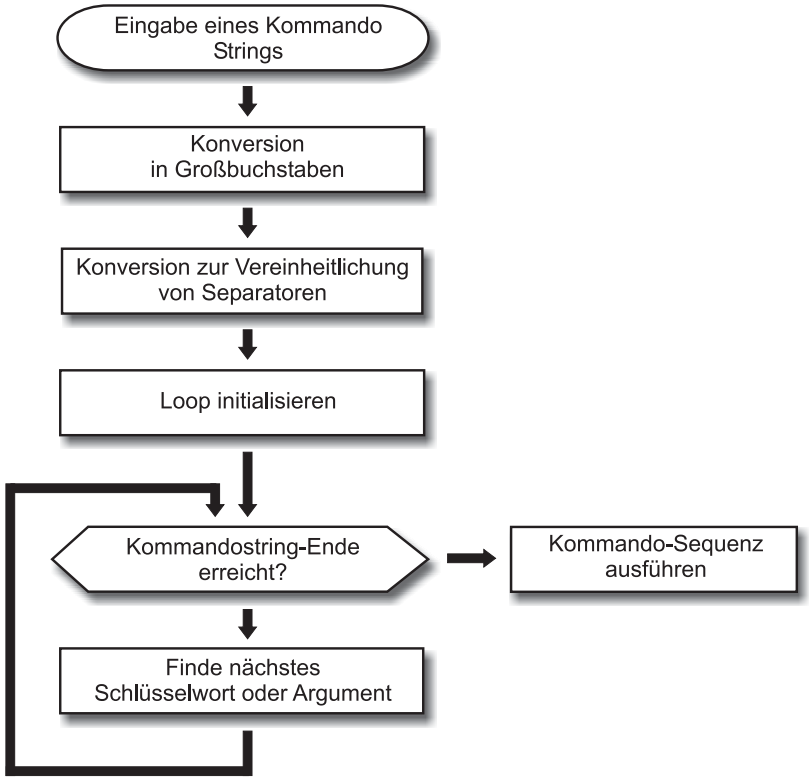
```
      Src$ = "please print on display and then wait 60 minutes"
   Key_WL$  = ".show.print.lptr.com.line.display.secondary.wait.delay."
 Start_Pos = 0                    ' start at position 0
 Tolerance = 0                    ' find exact match
 R$ = Check_Keyword$ (Src$, Key_WL$, Sep$, Start_Pos,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

Es wurde das Schlüsselwort „print“ erkannt, welches als 2. in der Liste steht (Index =1). Für die weitere Analyse des Source Strings, steht die Angabe von „Next-Pos“ zur Verfügung, die direkt hinter das gefundene Schlüsselwort „print“ zeigt.

Beispiel-2

Tolerante Kommando-Eingabe z.B. zur Fernsteuerung per SMS oder Email. Es ist das Ziel ein einfaches Mensch-Maschine Interface zu schaffen, das leicht verständliche Klartext-Kommandos verwendet um bestimmte Funktionen auszulösen.

Grundsätzliche Programmstruktur:



Zunächst wird jeder Kommandostring durch Konversionen in eine Standard-Form gebracht, hier Großbuchstaben und einheitliche Trenner - einschließlich Gleichbehandlung von Dezimalpunkt und Dezimalkomma.

Anschließend werden die Strings nach relevanten Schlüsselwörten und numerischen Argumenten durchsucht.

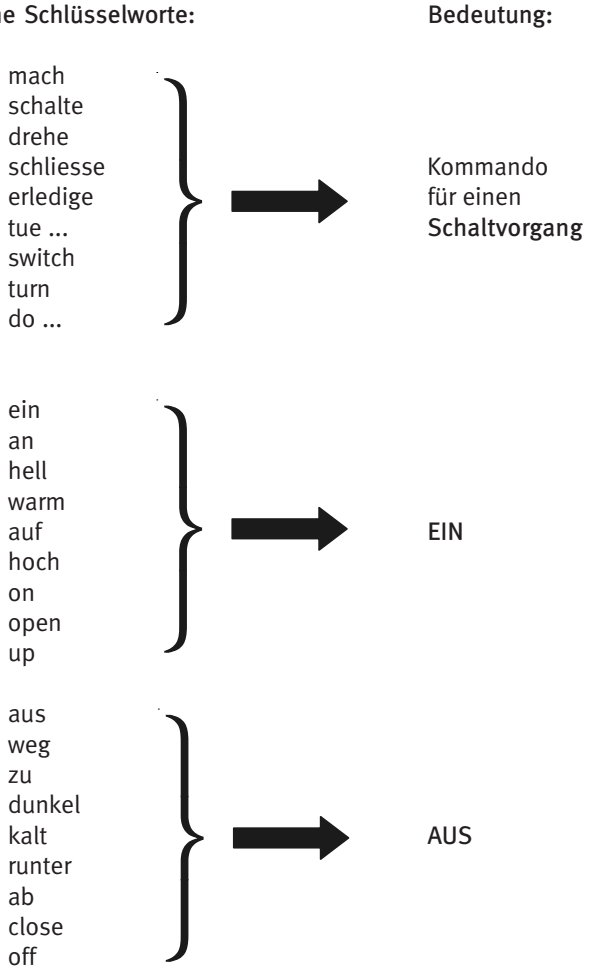
Um die Bedienung möglichst einfach und tolerant zu gestalten ist es wichtig, eine flexible und tolerante Verwendung von Schlüsselwörten zu realisieren. Dies trifft zu in

mehrfacher Hinsicht:

(a) es soll erlaubt sein unterschiedliche Begriffe für die gleiche Funktion zu verwenden,
(b) es soll erlaubt sein, lange Begriffe abzukürzen,
(c) es soll erlaubt sein, kleinere Schreibfehler machen zu dürfen, ohne die Funktion zu beeinträchtigen.

All diese Anforderungen erfüllt die Funktion: „Check_Keyword$“:

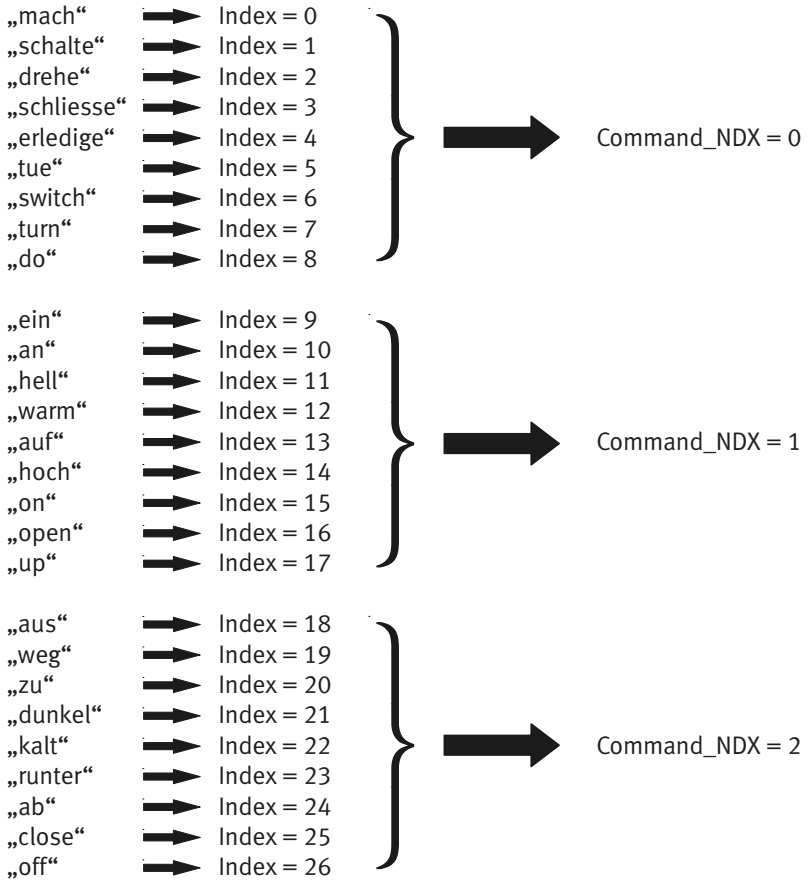
(a) Unterschiedliche Begriffe verwenden für gleiche Sachverhalte, zum Beispiel:



und die Umsetzung in BASIC-Code:

```
.
.
Key_WL$ = "&
-mach-schalte-drehe-schliesse-erledige-tue-switch-turn-do&  ' Ndx =  0...8
-ein-an-hell-warm-auf-hoch-on-open-up&                     ' Ndx =  9...17
-aus-weg-zu-dunkel-kalt-runter-ab-close-off-...."          ' Ndx = 18...26

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

Wenn im Source-String Src$ eines der o.g. Schlüsselworte auftritt, kommen jeweils folgende Werte für „Index“ und „Command_NDX“ zustande:



Mit diesem Aufbau der Keyword Tabelle kann der Sachverhalt:

„schalte ein“

bereits mit einer Vielzahl von Kommando String Varianten vorgeben werden.
Einige Beispiele:

„bitte *schalte* jetzt das licht *ein*“
„*drehe* das licht *an*“
„*turn* the light *on*“
„*switch on*“
„*drehe* auf *warm*“
„mach mal die heizung *an*“
„*mach* es *hell* hier“
„*schliesse* die heizung *auf*“

Man erkennt, dass durch Wahl alternativer Worte für den gleichen Sachverhalt, eine abwechselungsreiche Kommando-Eingabe in einem Mensch-Maschine Interface möglich wird.

Abkürzungen erlauben (b)

Entsprechend kann man die Vielfalt der Kommando-Varianten noch weiter erhöhen, indem man eine Toleranz bei der Keyword-Identifizierung hereinbringt:

Toleranz = +1... +nn, es ist erlaubt ➡ **Begriffe abzukürzen.**

Mit der Toleranz-Angabe von +1 ... +nn wird ein Keyword erkannt, wenn es:

(i) komplett richtig im Source-String vorliegt, oder
(ii) die ersten 1 ... nn Zeichen übereinstimmen.

In obigem Beispiel würden mit einer Toleranzangabe von +3 auch diese Fälle richtig erkannt, die mit Toleranz = 0 (genaue Übereinstimmung) nicht erkannt würden:

„bitte *schalte* jetzt das licht *ein*“
„bitte *schalt* jetzt das licht *ein*“
„bitte *schal* jetzt das licht *ein*“
„bitte *scha* jetzt das licht *ein*“
„bitte *sch* jetzt das licht *ein*“
„bitte *schalten* sie jetzt das licht *ein*“

„*schaltens* mal das licht *ein*“
„bitte *sch ein*“
„*sch ein*“

Nicht funktionieren würde z.B:
„bitte *sch* jetzt das licht *ei*“

Erlaubte Abkürzungen haben oft Vorteile:

- die Vielzahl der zulässigen Varianten steigt, ohne, dass man die Liste der Schlüsselwörter verlängern muss. Das schafft ein noch zuverlässigeres Interface auch für gelegenliche Benutzer.

- Vielbenutzer können sich lange Kommando-Worte sparen und bequeme Kürzel verwenden.

- Datensätze mit Kommandos werden kompakter, kürzere Übermittelungen.

Schreibfehler erlauben (c)

Die Vielfalt der Kommando-Varianten kann noch weiter erhöht werden, indem man durch den Toleranz Parameter auch fehlerhafte Schreibweisen bis zu einem vorbestimmten Grad zuläßt.

Fehlerhafte Schreibweisen haben viele Ursachen und oft ist es ärgerlich, wenn Kommandos wegen Tippfehlern oder Unsicherheit über korrekte Schreibweise nicht ausgeführt werden.

Fehlertoleranz ist häufig dann erwünscht, wenn

- die Anwendung nicht mit sicherheitsrelevanten Vorgängen zu tun hat
- die NICHT-Ausführung des Kommandos Nachteile bringt
- es keinen Rück-Kanal gibt. Man merkt u.U. gar nicht, dass das Kommando gar nicht ausgeführt wird wegen fehlerhafter Schreibweise.
- Wenn man einem großen Personenkreis den Zugriff erlauben will und fehlerhafte Schreibweisen daher fast unvermeidlich sind

Check_Keyword$ unterscheidet 3 Typen von Schreibfehlern:

1.) fehlerhafte Zeichen
2.) überzählige Zeichen
3.) fehlende Zeichen

Es können jeweils bis zu 3 solcher Fehler in einem Schlüsselwort akzeptiert werden:

Mögliche Toleranzparameter hierzu:

Toleranz:	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
Keinen Fehler dulden:	●	○	○	○	○	○	○	○	○	○
1 fehlerhaftes Zeichen = OK:	○	●	○	○	○	○	○	○	○	○
2 fehlerhafte Zeichen = OK:	○	○	○	○	●	○	○	○	○	○
3 fehlerhafte Zeichen = OK:	○	○	○	○	○	○	○	●	○	○
+1 Zeichen zuviel = OK:	○	○	●	○	○	○	○	○	○	○
+2 Zeichen zuviel = OK:	○	○	○	○	○	●	○	○	○	○
+3 Zeichen zuviel = OK:	○	○	○	○	○	○	○	○	●	○
-1 Zeichen zu wenig = OK:	○	○	○	●	○	○	○	○	○	○
-2 Zeichen zu wenig = OK:	○	○	○	○	○	○	●	○	○	○
-3 Zeichen zu wenig = OK:	○	○	○	○	○	○	○	○	○	●

Beispiele

1.)	Korrekte Form	Heizung	
	1 fehlerhaftes Zeichen	Heisung	
		Haizung	
	2 fehlerhafte Zeichen	Haizunk	
	3 fehlerhafte Zeichen	Haizunk	
2.)	Korrekte Form	Heizung	
	1 überzähliges Zeichen:	Heizzung	
	2 überzählige Zeichen:	Heizzungg	
	3 überzählige Zeichen:	Hjeizzungg	
	3 überzählige Zeichen:	Haeiszungg	
3.)	Korrekte Form	Heizung	
	1 fehlendes Zeichen:	Hizung	
	1 fehlendes Zeichen:	Heiung	
	1 fehlendes Zeichen:	Heizng	
	3 fehlende Zeichen:	Hezg	
	3 fehlende Zeichen:	eing	
	3 fehlende Zeichen:	Hein	
	3 fehlende Zeichen:	Heiz	(= Abkürzung)

Beim Einsatz des Toleranz Parameters erhält man im Ergebnis-String die Information zurück mit welcher Toleranz das gefundenen Schlüsselwort im Source Text vorgefunden wurde. D.h. auch wenn man eine tolerante Schreibweise für Schlüsselwörter zuläßt kann die wirkliche Schreibweise im Source String vollkommen korrekt sein oder weniger falsch als man toleriert:

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

Wird ein gültiges Schlüsselwort hier gefunden, kann in R$ die Toleranz-Angabe 3 mögliche Werte annehmen:

Toleranz = 0 ➡ Schlüsselwort mit vollkommen richtiger Schreibweise
Toleranz = -1 ➡ Schlüsselwort mit 1 falschem Zeichen gefunden
Toleranz = -4 ➡ Schlüsselwort mit 2 falschen Zeichen gefunden

Schreibfehler und Abkürzungen

Um einem Kommando-Interface die maximale Flexibilität zu geben, kann man Kommando-Strings wiederholt mit unterschiedlichen Toleranz-Vorgaben untersuchen und jeweils das Resultat überprüfen.

Zum Beispiel kann es sinnvoll sein zu dulden:

1.) Abkürzungen bis auf mind. 3 Zeichen
2.) 1 fehlendes Zeichen = OK
3.) 1 zusätzliches Zeichen = OK
4.) 1 fehlerhaftes Zeichen = OK

Das Schlüsselwort: „schalte“ wird mit diesen Vorgaben z.B. so erkannt:

schalten
schalte
schalt
schal
scha
sch
schalde, schulte, scjalte, zchalte, ...
schlte, shalte, scalte, schale, ...
schahlte, schaalte, schallte, schaltae, ...

Durch den Einsatz verschiedener Schlüsselbegriffe für einen Sachverhalt, sowie den Einsatz verschiedener Toleranz-Parameter kann man mit relativ einfachen Mitteln äusserst flexible Kommando-Interfaces schaffen.

Check_Keyword$ sorgt für kurze Laufzeiten und vereinfacht die Programmierung.

Hinweis:

Beim Aufbau der Schlüsselwort Liste ist zu bedenken, dass die Suche nach einem passenden Schlüsselwort immer vom Beginn der Liste an erfolgt. Prinzipiell ist die Reihenfolge der Schlüsselwörter in der Liste beliebig, sie hat jedoch folgende Auswirkungen:

- die Reihenfolge der Schlüsselwörter legt ihren jeweiligen Index fest: 0, 1, 2, 3 ...
- Schlüsselwörter, die sehr häufig auftreten sollten möglichst weit oben in der Liste geführt werden, damit nicht unnötig viel Laufzeit bei der Suche weiter unten in der Liste verbraucht wird.
- Was zuerst kommt kann auch zuerst gefunden werden:
  „Wasser“ sollte daher NACH „Wasserhahn“ in der Liste stehen.

Chk_Fnam

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

Funktion: Untersucht einen String auf das Vorhandensein eines Filenamens, exaktes Match oder mit Wildcards.

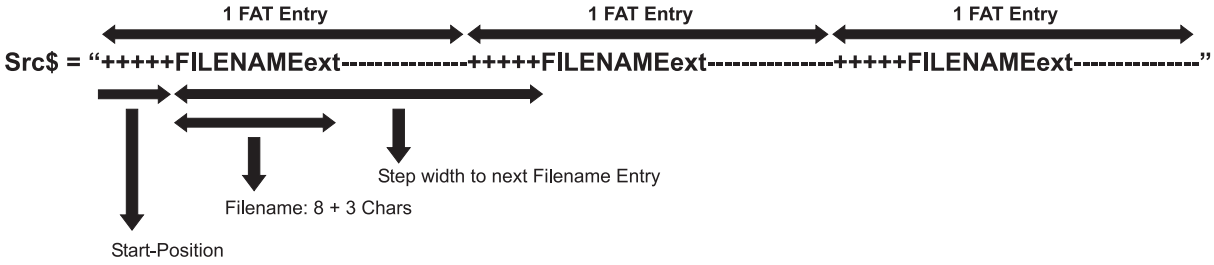
Anwendung: Suchen in Filesystemen

Parameter:

	B	W	L	S	F	
Src$	-	-	-	●	-	Source-String mit FAT und Filenamen
Start_Pos	●	●	●	-	-	Start-Position zum Suchen des Filenamens in Src$: 0 ... nnnn
Stepwidth	●	●	●	-	-	Schrittweite in Byte um zum nächsten Filenamen-Eintrag zu kommen: 0 ... nnnn
FName$	-	-	-	●	-	Filename zum Suchen: "FILENAME" + "EXT"
Method	●	●	●	-	-	Filename Typ und Suchmethode: 0: Filename + Ext wie in DOS: Filename: 8 ASCII-Zeichen: "A"..."Z", "0...9", "-" ggf. aufgefüllt mit Blanks Ext: 3 ASCII-Zeichen: "A"..."Z", "0...9", "-" ggf. aufgefüllt mit Blanks Nur GROSS-Buchstaben, sowie Wildcards: "*" = 0...8 bel. Zeichen im Filename "*" = 0...3 bel. Zeichen im EXT "?" = genau 1 bel. Zeichen
						Funktionswert:
Found_Ptr	●	●	●	-	-	Pointer auf den Anfang des gefundenen Filenamen in Src$: 0 ... nnnn Falls nicht gefunden: = -1

Chk_Fnam$ Suche Filename

Typischer Source-String Aufbau:



Im Source-String ist der Filename wie folgt formatiert:

8 + 3 ASCII-Zeichen ggf. mit Blanks ausgefüllt, zum Beispiel:

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

Die Notation für den gesuchten nächsten Filenamen (Fnam$) entspricht der üblichen Schreibweise:

Filename ‹Punkt› Erweiterung

wobei Erweiterung und Punkt ggf. entfallen können.

Also z.B:
"C-001.DAT"
"GO.EXE"
"S-001.MP3"
"A"
"A."
"ABC.D"

Suche Filename

Einige Beispiele für Fnam$ Werte und entsprechende Filename-Einträge in Src$:

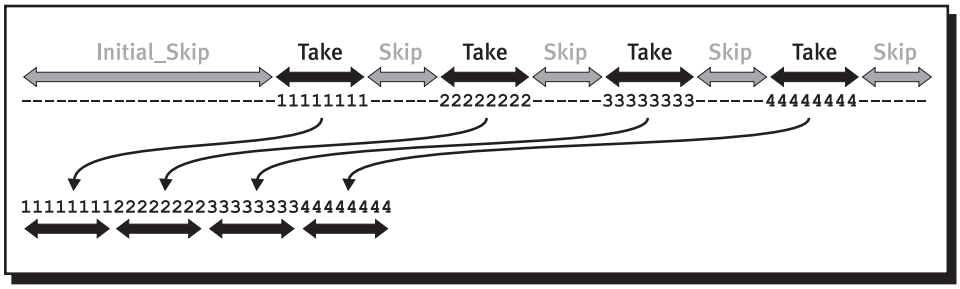
```
 Fnam$           <--Src$--->
                 12345678...
"*.TXT"          HALL    TXT   -->  ja gefunden
"HA*.TXT"        HALL    TXT   -->  ja gefunden
"HA*.TX*"        HALL    TXT   -->  ja gefunden
"HALL*.TXT"      HALL    TXT   -->  ja gefunden

"HA?*.T??"       HALL    TXT   -->  ja gefunden
"HA??*.TXT"      HALL    TXT   -->  ja gefunden
"HA???*.TXT"     HALL    TXT   -->  NICHT gefunden
"HA????.TXT"     HALL    TXT   -->  NICHT gefunden

"H*.*"           HALL    TXT   -->  ja gefunden
"*.TXT"          HALL    TXT   -->  ja gefunden
"*.*"            HALL    TXT   -->  ja gefunden
"?A*.T*"         HALL    TXT   -->  ja gefunden
```

Concentrate$

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

Funktion: Concentrate$ fasst Daten-Felder zusammen die z.B. in Record-Form in Strings vorliegen.

Extrahiert und konzentriert Daten-Bereiche aus String



Parameter:

	B	W	L	S	F	
Source_Destin$	-	-	-	●	-	Ein- und Ausgangs-String der konzentriert werden soll.
Initial_Skip	●	●	●	-	-	Überspringe diese Anzahl von Bytes zu Beginn,
Take	●	●	●	-	-	dann nimm diese Anzahl von Bytes
Next_Skips	●	●	●	-	-	und überspringe dann immer wieder diese Anzahl von Bytes, bevor wieder ‹Take› Bytes genommen werden.

Diese Sequenz wird von Anfang bis Ende des Strings ausgeführt.

Kein Funktionswert

Beispiel

Ausschnitt aus Kundenstamm Datei im ASCII Format

```
S_D$ = "&                                                    ' Ausschnitt
Marks      Bob     Boston       12345Rodeo Drive     00023813&' aus einer
Michelson Sven     Baltimore       77Huntington St.  00015222&' Kunden-Datei
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$ nach dem Funktionsaufruf:

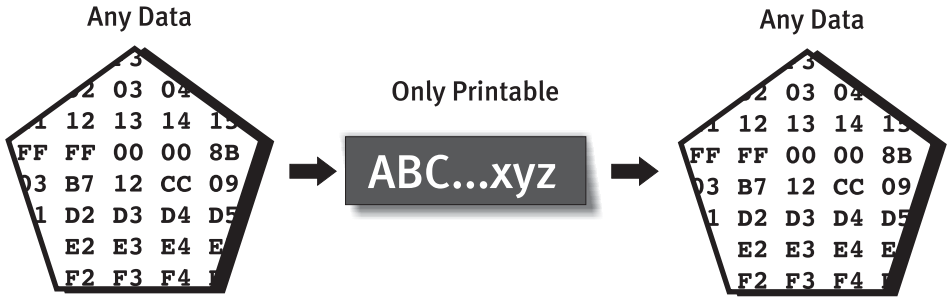
```
-23456789012345-23456789012345-23456789012345-23456789012345
Rodeo Drive     Huntington St. 14-th Street    Broadway
```

Concentrate$ schneidet hier die Informations-Felder „Strasse“ aus dem Beispiel-String heraus.

Conv_Base64$

```
ERG$ =  Conv_Base64$ ( Source$, Methode )
```

Funktion: Konvertiert beliebige Binär-Daten in den BASE64 Code für email Attachments und zurück.



Base64 Konvertierung / Dekodierung

Parameter:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source mit waagerecht orientierten Pixeln
Methode	●	●	●	-	-	Konversions-Richtung
						0: Binär ➡ Base64
						1: Binär ➡ Base64
						-1: Base64 ➡ Binär
						Funktionswert:
Erg$	-	-	-	●	-	Ergebnis String konvertiert lt. „Methode“

Conv_Base64$ wird eingesetzt um beliebige Bestände von Daten-Bytes (a 8 Bit) wie z.B. Texte, Binärdaten, Graphic, Sound, Steuercodes, ... auf einen Zeichensatz von 64 druckbaren Zeichen abzubilden - und wieder zurück. Die Base64 Abbildung ist ein-eindeutig und wird im Bereich e-Mail verwendet um Attachments beliebigen Inhaltes zu kodieren.

Ebenso eignet sich diese Konvertierung für Anwendungen, wo transparente Übertragungen gewünscht werden, obwohl der Übertragungs-/Speicherkanal mit Code-Beschränkungen arbeitet.

Die Base64 Zeichentabelle:

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		

Für eine weitergehende Beschäftigung mit dem Base64 Code nachfolgend eine detailierte Beschreibung (in englisch).

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 property 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 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$,          &  ' Source-String, wird untersucht
         Patt$,         &  ' String mit 1...nn Patterns, Länge je: Patt_ELen
         Patt_ELen,     &  ' Entry-Länge von Pattern
         Pos,           &  ' Start Position zum Zählen
         Cnt_Len)          ' Länge in Src$ für Zählvorgang
```

Funktion: Count_Patt$ untersucht einen Source-String auf das Vorhandensein von vorgegebenen Byte-Pattern und zählt deren Häufigkeit.

Parameter:

	B	W	L	S	F	
Src$	-	-	-	●	-	Source-String mit Daten, die auf das Auftreten bestimmter Byte-Muster hin untersucht werden sollen. Start-Position und Länge wie in Parametern „Pos“ und „Cnt_Len“ angegeben.
Patt$	-	-	-	●	-	String mit einer Liste von Patterns die gesucht und gezählt werden sollen. Jedes Pattern hat feste Länge lt. Parameter „Patt_ELen“. ‹patt-0› ‹patt-1› ‹patt-2› ‹patt-3› ‹patt-4› ...
Patt_ELen	●	●	●	-	-	Pattern-Länge im String Patt$.
Pos	●	●	●	-	-	Start-Position in Source-String Src$.
Cnt_Len	●	●	●	-	-	Count Länge in Src$: diese maximale Anzahl von Bytes wird untersucht.
						Funktionswert:
Cnt$	-	-	-	●	-	String mit einer Liste von LONG-Counterwerten. Jeder LONG Wert gibt die Häufigkeit des entsprechenden Patterns im Source-String wieder: ‹cnt-0› ‹cnt-1› ‹cnt-2› ‹cnt-3› ... ‹cnt-n-1›

Beispiel:

```
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)
'
' Nach Ausführung von "Count_Patt$":
'
'  CNT$ -->  "02 00 00 00 01 00 00 00"    '
'              <=== 0 ===> <=== 1 ===>     ' <- Counter-Index
```

Noch ein Beispiel:

```
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)
'
' Nach Ausführung von "Count_Patt$":
'
'  CNT$ -->  "02 00 00 00 01 00 00 00"    '
'              <=== 0 ===> <=== 1 ===>     ' <- Counter-Index
```

CRC

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

Funktion: Bildet eine CRC Prüfsumme aus den Daten eines String ab einer Anfangs-Position und mit vorgegebener Länge.

```
Bilde 32-Bit CRC:   1. Byte * (FAKTOR)          ➡ aufsummieren auf CRC
                    2. Byte * (FACTOR+1*STEP)   ➡ aufsummieren auf CRC
                    3. Byte * (FACTOR+2*STEP)   ➡ aufsummieren auf CRC

                    4. Byte * (FACTOR+3*STEP)   ➡ aufsummieren auf CRC
                       .          .
                       .          .
                  100. Byte * (FACTOR+99*STEP)  ➡ aufsummieren auf CRC
                  101. Byte * (FACTOR+100*STEP) ➡ aufsummieren auf CRC
                       .          .
                       .          .

          CRC Summe = 32 Bit = 8 HEX-Digits
                       (bei Ueberlauf: don't care)
```

CRC Berechnung

Parameter:

	B	W	L	S	F	
A$	-	-	-	●	-	Source Daten
Pos	●	●	●	-	-	Start-Position in Source-String: 0 ... (LEN(A$)-1)
Len	●	●	●	-	-	Anzahl Bytes zur CRC-Berechnung: 1 ... LEN(A$)
Start_Factor	●	●	●	-	-	Eingang: Start-Faktor zur CRC-Berechnung Ausgang: nächster Start-Faktor für fortgeführte CRC Berechnung
Step	●	●	●	-	-	Schrittweite für Faktor zur CRC-Berechnung
CRC	●	●	●	-	-	Eingang: Anfangswert zur CRC-Berechnung, zu Beginn üblicherweise = 0 Ausgang: Ergebnis CRC

Kein Funktionswert

CRC

Fortlaufende CRC Berechnung mit einstellbaren Parametern

CRC wird eingesetzt zur Bildung von einfachen Prüfsummen und CRCs in Strings wie auch in umfangreichen, auch kontinuierlich fließenden Datenströmen. Dazu verfügt die Funktion über:

reine Eingangs-Parameter: A$, Pos, Len, Step

sowie über Ein- und Ausgangs-Parameter: Start_Factor, CRC

Beispiel zur einfachen Berechnung einer einfachen Prüfsumme:

Beispiel-1:

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

Dieses Beispiel erzeugt eine einfache Prüfsumme über alle Bytes des Strings A$. Die Sicherheit der einfachen Prüfsumme ist gering, da einfache Fehler wie Byte-Vertauschung oder 2 Bit-Flips in gleicher Position ... etc. nicht erkannt werden. Sicherere Ergebnisse produziert der CRC mit Festlegung von verschiedenen Faktoren:

Beispiel-2:

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

Bei besonders großen Datenmengen und auch bei fortlaufend übertragenen Datenströmen, kann der CRC fortlaufend abschnittweise - mit Erzeugung von Zwischenergebnissen - berechnet werden. Hier das entsprechende Beispiel, welches das gleiche Resultat liefert wie Beispiel 2.

Beispiel-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
```

Beim 2. Aufruf von CRC ist „Start_Factor“ bereits richtig auf den nächsten erforderlichen Faktor (77) für die Fortsetzung der CRC Berechnung gesetzt, SUM enthält bereits den CRC-Wert aus dem 1. Funktionsaufruf. Im Endergebnis liefern Beispiel-2 und Beispiel-3 das gleiche Ergebnis.

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
```

Funktion: Zusätzliche „Cut_and_Paste“ Funktionen (vergl. V5.0 „Cut_and_Paste“) für String, Num und Real Variablen.

Parameter:

	B	W	L	S	F	
SRC$	-	-	-	●	-	Source-String
Offset	●	●	●	-	-	Position im Source-String, an der das Ausschneiden beginnt.
Len	●	●	●	-	-	Anzahl Bytes, die herausgeschnitten und in die Ergebnis-Variable transferiert werden sollen.
						Funktionswerte:
A$	-	-	-	●	-	Herausgeschnittener String
R	-	-	-	-	●	Herausgeschnittener Fließkomma-Wert (Real)
N	●	●	●	-	-	Herausgeschnittener ganzzahliger Wert

Diese Funktionen schneiden Daten-Bytes aus einem Source-String heraus und transferieren diese Bytes in die jeweilige Zielvariable.

Cut_and_Paste Funktionen erlauben einfache Manipulationen an Strings, Anpassung und Zuschneiden von Daten-Sätzen, Inter-Task Kommunikation, Zeichen- / Bufferübergaben, ... etc.

Beispiel hierzu auf der nächsten Seite.

Weitere Cut and Paste Funktionen

Beispiel:

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

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

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

SRC$ = H e W o r l d

l l o ← cutted away

N = 00 6F 6C 6C hex

siehe auch: INSERT$

Calc_ECC

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

Funktion: Berechnet einen ECC (Error-Correction-Code) für einen Daten-Block, mit dem Bitfehler erkannt und 1 Bit-Fehler korrigiert werden können.

Anwendung: Sicherung von Speicherblocks in Massenspeichern (SRAM, FLASH ... etc.). Siehe auch Funktion: „Correct_ECC“.

Parameter:

	B	W	L	S	F	
DATA$	-	-	-	●	-	Für diesen DATA-String - oder einen Teil davon - wird der Error-Correction-Code ermittelt. Der ECC wird über genau 256 Bytes berechnet.
Start	●	●	●	-	-	0 ... nnnn Start-Offset in DATA$
ECC_Method	●	●	●	-	-	ECC Methoden-Wahl: 00 Methode-0: ECC gemäß „SmartMedia(tm) Physical Format Specification Version 1.20“ Nähere Format-Beschreibung siehe: „Correct_ECC“ Funktion
						Funktionswert:
ECC$	-	-	-	●	-	24 Ergebnis-Bits = 3 Bytes in einem 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)

Funktion: Korrigiert einen Data-Block mit eventuellem Parity-Fehler und korrigiert 1 Fehler oder signalisiert 2 Fehler.

Anwendung: Sicherung von Speicherblocks in Massenspeichern (SRAM, FLASH ... etc.). Siehe auch Funktion: „Calc_ECC“.

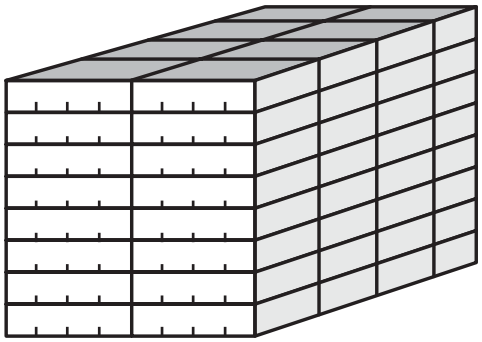
Parameter:

	B	W	L	S	F	
DATA$	-	-	-	●	-	Für diesen DATA-String - oder einen Teil davon - wird der Error-Correction-Code ermittelt und ggf. eine Korrektur vorgenommen.
Start	●	●	●	-	-	0 ... nnnn Start-Offset in DATA$
ECC_stored	●	●	●	-	-	Das ist der ECC wie er auf dem Speicher-Medium gespeichert ist.
ECC_calculated	●	●	●	-	-	So wurde der ECC aus den tatsächlichen DATEN in Data$ berechnet.
ECC_Method	●	●	●	-	-	ECC Methoden-Wahl: 00 Methode-0: ECC gemäß „SmartMedia(tm) Physical Format Specification Version 1.20“
						Funktionswert:
FLAG	●	●	●	-	-	Ergebnis-Flag: 00: Alles OK, es gab nichts zu korrigieren: ECC_stored = ECC_calculated 01: Es wurde 1 Byte in DATA$ korrigiert und danach ist jetzt: ECC_stored = ECC_calculated 02: Es wurde Korrektur gemacht --› ABER: ECC_stored ‹› calculated

Correct_ECC **Check and Correct with Error-Correction-Code**

Beschreibung der ECC (Error-Correction-Code) Bildung nach „SmartMedia™ Physical Format Specification Version 1.20“ aus dem SSFDC Forum:

Ein Datenblock von 256 Byte wird als ein Bitstrom von 2048 Bits betrachtet. Jedes dieser 2048 Bits besitzt eine 11-Bit Adresse.



Daraus werden 22 Teilmengen von jeweils 1024 Bits gebildet. Für jede dieser Teilmengen (a 1024 bit) wird:

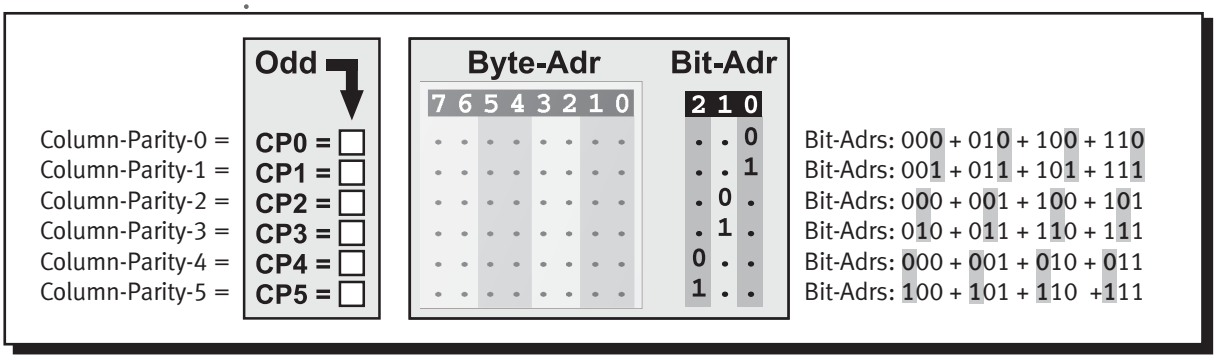
ein *ODD-Parity Bit*

gebildet.

Die SmartMedia™ Commision unterscheidet:

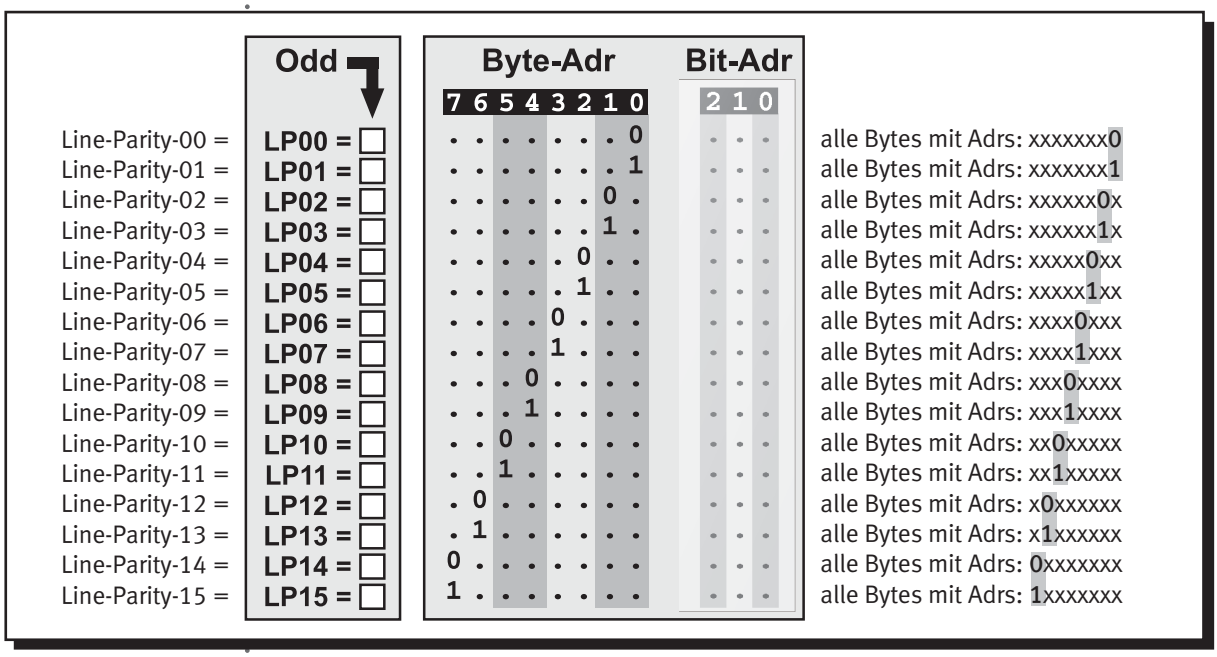
"6 Column-Parities": Bit-Adressen innerhalb von Bytes (3 Adr-Bits), sowie
"16 Line-Parities": welche die einzelnen Bytes festlegt (8 Adr-Bits)

Berechnung der 6 (Odd) Column Parity Bits:



Check and Correct with Error-Correction-Code

Berechnung der 16 (Odd) Line-Parity Bits:



Aus 8 Columns + 16 Lines werden 22 ODD-Parity Bits wie folgt zum ECC (Error-Correction-Code) in 3 Bytes zusammengestellt:

	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 wird eingesetzt um gespeicherte Daten in Massenspeichern und Langzeitspeichern gegen unbemerkte Veränderung zu schützen und einzelne Bitfehler erkennen und korrigieren zu können.

Find_opt_Group$

```
Pos_Ndx$ = Find_opt_Group$ ( &  '
           Nums$,            &  ' Mengen in den Fächern
           Group_Ndx_List$,  &  ' Fächerkombinationen der Gruppen
           Size_of_Group,    &  ' Anzahl Fächer je Gruppe
           Size_of_Nums,     &  ' Anzahl Bytes je Fach, immer = 2 (Word)
           Target_Val,       &  ' Ziel Menge gesamt
           Tolerance_Band,   &  ' Toleranz-Band
‹ opt ›    Start_Pos,        &  ' Start-Position in: „Group_Ndx_List$“
‹ opt ›    Target_Tol )          ' Ziel-Mengen Toleranz für: „Target_Val“
```

Funktion: Find_opt_Group$ stellt optimale Gruppen von Fächern zusammen, die unterschiedliche Inhaltsmengen besitzen - z.B. die Sammelbehälter in Abfüllanlagen.

Die besondere Aufgabenstellung dort ist es, aus einer Reihe von Sammelbehältern bzw. Fächern mit unterschiedlicher Befüllung an Materialmengen, die Kombination von mehreren Behältern herauszufinden, die der gewünschten Zielmenge zum Abfüllen möglichst nahe kommt. Auch soll in solchen Fällen die Abweichung von der genauen Zielmenge möglichst klein bleiben und ein vorgegebenes Toleranzfeld soll eingehalten werden.

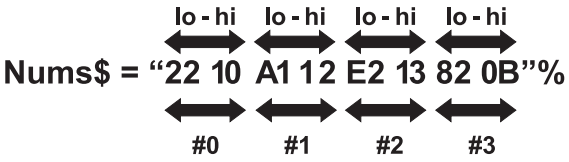
Bei diesen Anwendungen ist häufig eine hohe Entscheidungsgeschwindigkeit gefordert um die Produktivität der Anlage voll ausfahren zu können. Find_opt_Group$ beschleunigt den Entscheidungsvorgang maßgeblich indem mit einer einzigen Funktion hunderte oder tausende von Kombinationen auf ihre Tauglichkeit hin überprüft werden.

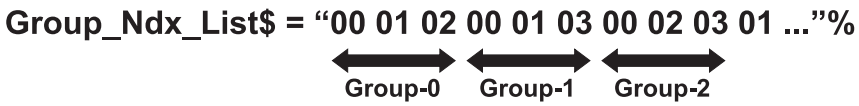
Find_opt_Group$ übernimmt den Optimierungsprozeß um die beste Gruppe von Fächern zusammenzustellen, deren Summe der gewünschten Zielmenge so nah kommt wie durch die Parameter spezifiziert.

Parameter:

	B	W	L	S	F	
Nums$	-	-	-	●	-	String mit den aktuellen Inhalten aller Sammelbehälter / Fächer: 0 ... FFFFH vorzeichenlos. Anzahl der Fächer lt. Parameter „Size_of_Group“, Wertgröße jeweils lt. Parameter „Size_of_Nums“ - z.Zt. immer WORD.

Find_opt_Group$ Finde optimale Gruppen Zusammenstellung

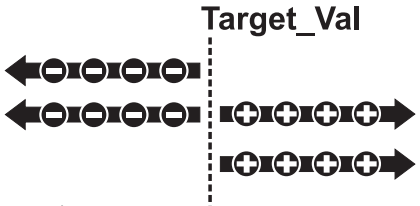
	B	W	L	S	F	
						Aufbau von Nums$:  Hier: Fächer 0 ... 3 Nums$ enthält die aktuellen Inhaltsmengen von den bis zu 32 Fächern der Abfüllmaschine als Word-Werte im Big-Endian Modell (wie generell in Tiger-Basic). Vorzeichenloser Wertebereich: 0 ... 65535.
Group_Ndx_List$	-	-	-	●	-	String mit Index-Liste aller Fächer-Gruppen. Diese Liste enthält alle möglichen Kombinationen von Fächern bei vorgegebener Gruppen-Größe lt. „Size_of_Group". Indizes werden als Byte geführt, Wertebereich: 0 ... Size_of_Group-1

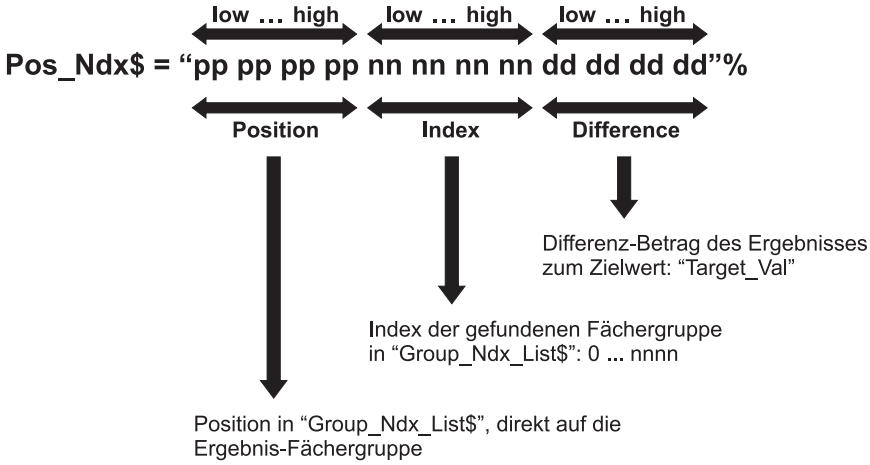


Size_of_Group	●	●	●	-	-	Anzahl von Fächern, die zu einer Gruppe gehören sollen: 1 ... Anzahl vorhandener Fächer
Size_of_Nums	●	●	●	-	-	Anzahl Bytes für die Mengenangaben je Fach in Nums$. Obligatorisch = 2 (Word).
Target_Val	●	●	●	-	-	Der gewünschte Gesamtmengen-Zielwert. Dieser Wert soll genau oder möglichst genau erreicht werden durch Kombination der unter „Size_of_Group" angegeben Anzahl von Fächern.
Tolerance_Band	●	●	●	-	-	Definiert das Toleranz-Band für die Ergebnis-Findung. „Find_opt_Group$" sucht die Fächer-Gruppe, die die kleinste Abweichung vom gewünschten Ergebniswert (Target_Val) besitzt. Die Angabe des Toleranz-Bandes legt fest, ob die

Find_opt_Group$ · Finde optimale Gruppen Zusammenstellung

	B	W	L	S	F	
						Zielmenge (-1) => kleiner-gleich (0) => kleiner-größer (Betrag der Differenz) (+1) => größer-gleich sein soll. Toleranz-Bänder:
Start_Pos	●	●	●	-	-	(Optional) Start-Position in: „Group_Ndx_List$“ 0 ... nnnn. Wird eingesetzt um z.B. alle exakten oder toleranzbehafteten Matches der Reihe nach ermitteln zu können.
Target_Tol	●	●	●	-	-	(Optional) Ziel-Mengen Toleranz für: „Target_Val“ Mit dieser Angabe kann man eine zulässige Ergebnis-Abweichung vorgeben, so dass eine solche Gruppe als OK gewertet wird.
						Funktionswert:
Pos_Ndx$	-	-	-	●	-	3 LONG Ergebnis-Werte im String:





Find_opt_Group$ **Finde optimale Gruppen Zusammenstellung**

B W L S F

„Position“:
Gibt die Position der Ergebnis-Fächergruppe an wie sie im String „Group_Ndx_List$“ vorliegt.
Werte: 0 ... nnnn
Wert: -1 => Fehler

„Index“:
Gibt den Index der Ergebnis-Gruppe (wie vor) an.
Werte: 0 ... nnnn
Wert: -1 => Fehler

„Difference“:
Signed LONG mit der Abweichung des gefundenen Ergebnisses vom eingestellten „Target_Val“ Wert.

Die Funktion „Find_opt_Group$“ bietet zahlreiche Möglichkeiten wenn es darum geht, eine Anzahl von Gruppen zu einer Ergebnis-Menge zusammenzufassen. „Find_opt_Group$“ erlaubt es mit wenigen Programmzeilen und bei kurzer Laufzeit solche Aufgaben zu bewältigen.

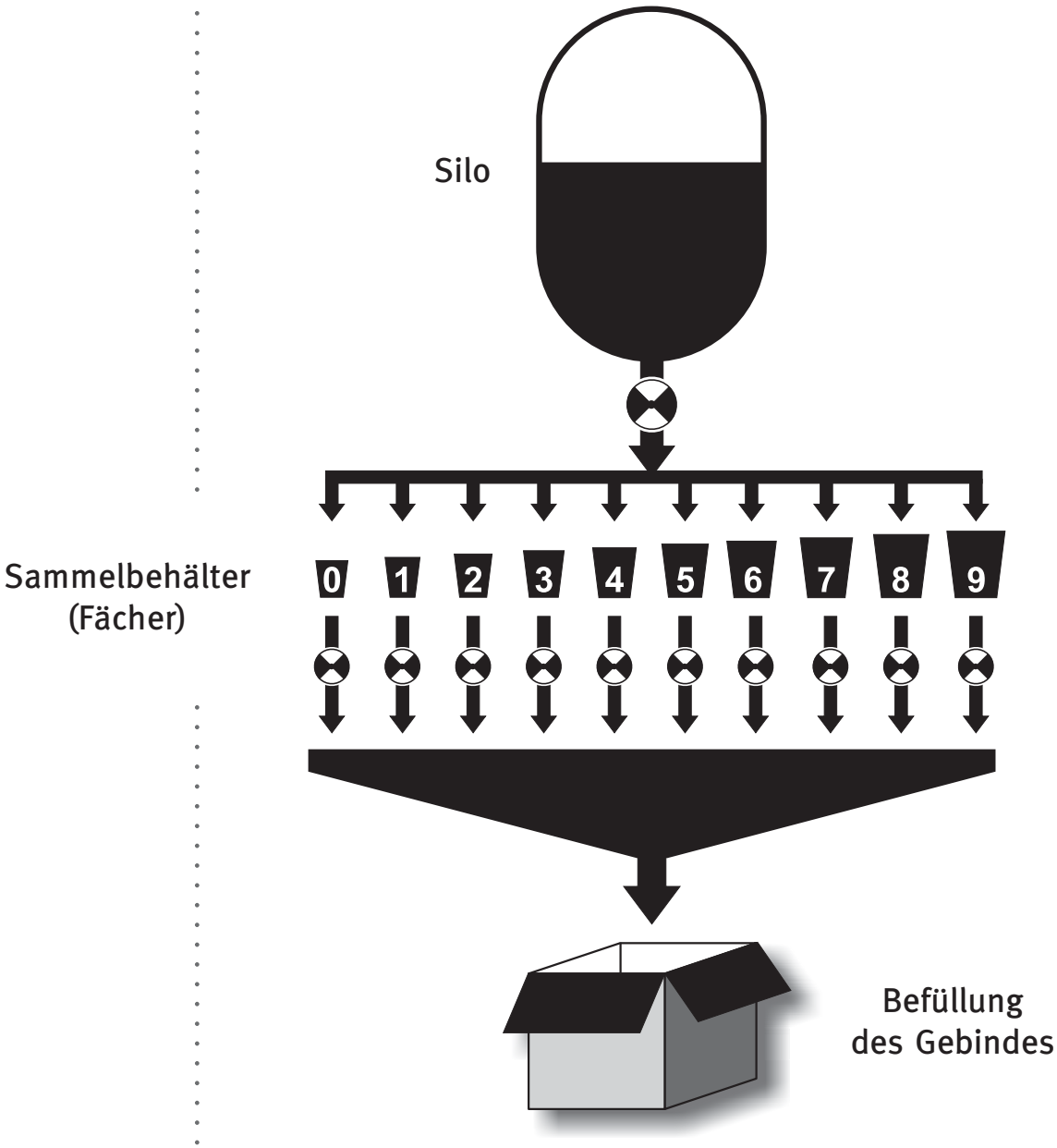
Um die Funktionen im Einzelnen verständlich zu machen, gehen wir hier vom Modell einer:

Abfüll-Anlage

aus. An Hand dieses Modells werden verschiedene Funktionen und Aspekte veranschaulicht.

Das folgende Bild gibt den schematischen Aufbau einer solchen Abfüllanlage wieder. Die Abfüllanlage in diesem Beispiel dient dazu Schüttware, z.B. Granulat, in Säcke oder Fässer abzufüllen. Die Anforderungen dabei liegen auf

- hoher Abfüll-Geschwindigkeit
- sowie engen Gewichtstoleranzen



Abfüll-Anlage

Find_opt_Group$ **Finde optimale Gruppen Zusammenstellung**

Zu diesem Zweck ist die Anlage so aufgebaut, dass es insgesamt 10 Fächer gibt, in die vorbestimmte Mengen des Schüttgutes aus dem Silo gefüllt werden. Die Befüllung dieser Fächer erfolgt möglichst schnell. Die dabei befüllten tatsächlichen Mengen weisen im Betrieb jedoch Toleranzen auf, so dass jedes der 10 Fächer mit einer elektronischen Waage ausgerüstet ist. So kann man das wirkliche Füllgewicht jedes Faches im laufenden Betrieb schnell und genau feststellen.

Zu Beginn eines Befüllungs-Zyklus stehen 10 Gewichtswerte von 10 Fächern zur Verfügung. Durch möglichst günstiges Zusammenschütten des Inhaltes mehrerer Fächer möchte man ein möglichst genaues Gewicht in das Ziel-Gebinde einfüllen.

Ein einfaches Zahlen-Beispiel:

Fach:	Material-Menge in den Fächern:	Zahlenwert:
#0	**********	36
#1	**********	31
#2	**********	30
#3	**********	19
#4	**********	23
#5	**********	18
#6	**********	17
#7	**********	15
#8	**********	12
#9	**********	20

Es sei die Aufgabe eine Ziel-Menge von 100 aus der Kombination von 2 ... 6 Fächern in obigem Beipiel zu erzielen:

- Wenn möglich soll ein exaktes Ergebnis erzielt werden,
- wenn das nicht möglich ist, soll das Ergebnis in einem vorgegebenen Toleranzfeld liegen.
- Falls auch das nicht möglich ist, soll dies als Fehler erkannt werden und eine entsprechende Fehlerbehandlung eingeleitet werden können.

Durch Probieren findet man z.B. folgende 3 Kombinationen von 4 Fächern - mit unterschiedlichen Ergebnissen:

1. Gruppe: #0 + #1 + #2 + #8 = 109 Tolerance = +9
2. Gruppe: #0 + #1 + #4 + #8 = 102 Tolerance = +2

3. Gruppe: #1 + #2 + #5 + #9 = 99 Tolerance = -1

Es wird schnell klar, dass möglicherweise auch ein exaktes Ergebnis existiert, u.U. sogar mehrere, dies im Regelfall jedoch nicht leicht erkennbar ist. Die Anzahl möglicher Kombinationen von 2 oder mehr Fächern aus 10 Fächern insgesamt ergibt sich zu:

Gruppen-Größe	‹---------- Anzahl an Fächer-Kombinationen ----------›		
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
Gesamt			= 1022

Zur Ergebnisfindung ist ein Algorithmus einzusetzen, der durch Kombinatorik und Bewertung systematisch ein Ergebnis ausreichender Genauigkeit findet. Diese Aufgabe erfüllt die „Find_opt_Group$“.

Eine wichtige Aufgabe von „Find_opt_Group$“ ist es auch, ein möglichst schnelles Laufzeit-Verhalten zu erreichen. Hierzu wird von vorbereiteten Tabellen mit den möglichen Fächerkombinationen Gebrauch gemacht. Diese Information wird im Parameter „Group_Ndx_List$“ für die jeweilige Gruppengröße zur Verfügung gestellt:

```
Size_of_Group     = 2                    '
Group2_Ndx_List$ = "00 01 00 02 00 03 00 04 00 05 00 06 00 07 ..."%
'                     <===> <===> <===> <===> <===> <===> <===>
'
' Gruppen von 2 Fächern werden durch Gruppen von 2 Bytes = 2 Indizes in
' der Liste geführt. Insgesamt werden alle möglichen 45 Kombinationen in
' diesem String gespeichert, so dass eine Länge von 90 Byte entsteht.
```

Entsprechend aufgebaut ist der String für Gruppen mit 3 oder mehr Fächern:

```
Size_of_Group     = 3                    '
Group3_Ndx_List$ = "00 01 02 00 01 03 00 01 04 00 01 05 00 01 06 ..."%
'                     <=======> <=======> <=======> <=======> <=======>
'
' Gruppen von 3 Fächern werden durch Gruppen von 3 Bytes = 3 Indizes in
' der Liste geführt. Insgesamt werden alle möglichen 120 Kombinationen in
' diesem String gespeichert, so dass eine Länge von 360 Byte entsteht.
```

Find_opt_Group$ Finde optimale Gruppen Zusammenstellung

„Find_opt_Group“ kann mit 6, 7 oder 8 Parametern eingesetzt werden. Für viele technische Anwendungungen empfiehlt es sich auch den 8. Parameter, Target-Tolerance, zu verwenden.

Hierdurch kann das Laufzeitverhalten erheblich beschleunigt werden.

In technischen Anwendungen kommt es in der Regel nicht darauf an das absolut beste Ergebnis von allen zu ermitteln - dazu müste man sämtliche Kombinationsmöglichkeiten berechnen und vergleichen - hier reicht es aus, ein genügend gutes Ergebnis zu finden. Man gibt eine noch akzeptierte Ergebnis-Toleranz und ein Toleranzband vor und hat ein gültiges Ergebnis bei der ersten gefundenen Kombination von Fächern, die diesen Anforderungen genügt.

Je nach Konstruktion der Abfüllanlage kann man ferner Einschränkungen treffen, was die Gruppengröße der Fächer angeht. Man betrachte dieses Beispiel, wo jeweils Säcke mit 25 kg befüllt werden sollen und folgende 10 Fächerbefüllungen gemessen werden:

Fach:	Material-Menge in den Fächern:	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

Diese Anlage befüllt die Fächer mit einer flachen Rampe, jedes Fach mit ca. 5 kg, jedoch in einer Unter- und Überbefüllung. Hier ist es offensichtlich nur sinnvoll die 5-er Gruppen zu untersuchen, da mit weniger oder mehr Fächern kein gutes Ergbenis zu erreichen wäre. In anderen Fällen sind auch besonders kleine Fächer denkbar mit kleinen Tarier-Mengen:

#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_opt_Group$ **Finde optimale Gruppen Zusammenstellung**

Je nach Aufgabenstellung wird nur mit den Gruppengrößen gearbeitet, für die Lösungsmöglichkeiten existieren können. Ebenso wird ggf. zur Geschwindigkeitsoptimierung mit der wahrscheinlichsten Gruppengröße begonnen und das zulässige Toleranzfeld ausgeschöpft.

Das Programm-Beispiel „Fillmachine_Find_opt_Group_Vxxx.TIG“ zeigt die grundsätzliche Funktionsweise.

Ohne Angabe des 8. Parameters werden alle möglichen Kombinatinen jeweils einzeln durchgerechnet und verglichen. Man erhält dann das jeweils beste kombinatorische Ergebnis mit einer bestimmten Gruppengröße. Nur im Sonderfall wenn ein exaktes Match gefunden wird, bricht der Suchvorgang vorzeitig ab. Diese Vorgehensweise ist immer dann einzusetzen wenn es um die jeweils bestmöglichen Resultate geht.

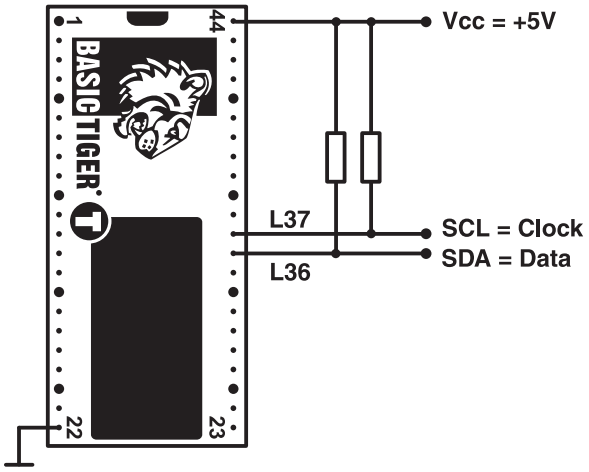
Schließlich kann der 7. Parameter, Start_Pos dazu eingesetzt werden an beliebiger Stelle von Group_Ndx_List$ die Suche fortzusetzen, wenn zuvor bereits ein Ergebnis gefunden worden ist. Man könnte z.B. alle möglichen Lösungen so ermitteln (siehe: „Fillmachine_Find_opt_Group_Vxxx.TIG“).

I2CL_Setup

I2CL_Setup (Port, Clock_Pin, Data_Pin, Speed)

I2C-Bus bzw. ISO 7816 einrichten

Funktion: Spezifiziert die eingesetzten Tiger Pins und eine ev. Timing-Verlangsamung für Low Level Funktionen I2C-Bus / ISO 7816 Kommunikation. Alle I2CL... Funktionen arbeiten als Bus-Master.



I2C-Bus / ISO 7816 Bus (Beispiel)

Parameter:

	B	W	L	S	F	
Port	●	●	●	-	-	Interner Port für Signale: SDA und SCL
Clock_Pin	●	●	●	-	-	Clock-Output Pin: (Bit-Nr: 0...7) = Clock generated by Master
Data_Pin	●	●	●	-	-	DATA-I/O Pin: (Bit-Nr: 0...7) = bidirektional
Speed	●	●	●	-	-	0 = keine Verlangsamung 1...20 = Speed Reduction
						Funktionswert:
-	-	-	-	-	-	Kein Funktionswert

Beispiel:

Beispiel

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

Diese Funktionszeile setzt folgende Festlegungen für I2C-Bus Low Level und ISO-7816 Kommunikation in Kraft:

SCL = Clock	SDA = Data
Port-3, Bit-7	Port-3, Bit-6

0 ==› keine Geschwindigkeits-Reduktion

Alle weiteren I2C-Bus / ISO 7816 Low Level Funktionsaufrufe verwenden die durch I2CL_Setup hier getroffenen Festlegungen.

Die beiden Bus-Leitungen SCL und SDA (CLOCK und DATA) sind als „wired-and“ Signale ausgeführt, es gibt auf jeder Busleitung:

- einen Pull-Up Widerstand der den Leitungspegel nach +5V zieht, sowie
- Open-Collector Ausgänge der Busteilnehmer und
- hochohmige Eingänge der Busteilnehmer

Sind alle Busteilnehmer inaktiv, d.h. alle angeschlossenen Busteilnehmer beeinflussen eine Busleitung nicht, so liegt für alle Teilnehmer ein +5V Pegel = „1“ an.

Wenn ein oder mehrere Teilnehmer eine Busleitung mit dem Open-Collector Ausgang zum GND Pegel (0V) ziehen, so sehen alle Teilnehmer das Bussignal „0“ - „wired-and“. Das Pegel-Diagramm für I2C-Bus / ISO 7816 gibt diesen Umstand in der Form wieder, dass:

- nur 1 Clock-Signal dargestellt ist - Clock wird immer vom Master vorgegeben
- und 2 Data-Signale gezeigt werden, von jedem der beiden betroffenen Busteilnehmer, woraus man den Fluß der Infomation erkennen kann.

Die Signalübertragung auf dem I2C-Bus folgt keiner fixen Bit/s Rate, ist jedoch für manche Chips und Konfigurationen begrenzt. Entsprechend der Einstellung des „Speed“-

Parameters in I2CL_SETUP (...) wird die kürzest mögliche Bitzeit beim I2CL_Read$ (...) berücksichtigt. Ansonsten erfolgt die Datenübertragung auf dem I2C-Bus rein statisch - kann also zu jedem Zeitpunkt angehalten und danach unverändert fortgesetzt werden.

Hinweis: Die ISO-7816 Übertragung verwendet noch eine RESET-Leitung zum Rücksetzen des Adr-Counters. Diese Leitung wird bei einer ISO 7816 Anwendung mit einem bel. Tiger-Pin oder Erweiterungspin realisiert und vom BASIC-Programm aus gesteuert.

Die „Answer-to-Reset“ (ATR) Prozedur ist standardisiert durch ISO 7816-3.

Verschiedene Programm-Beispiele sind verfügbar mit Namens-Präfix: „I2CL_“ und Extension „TIG“.

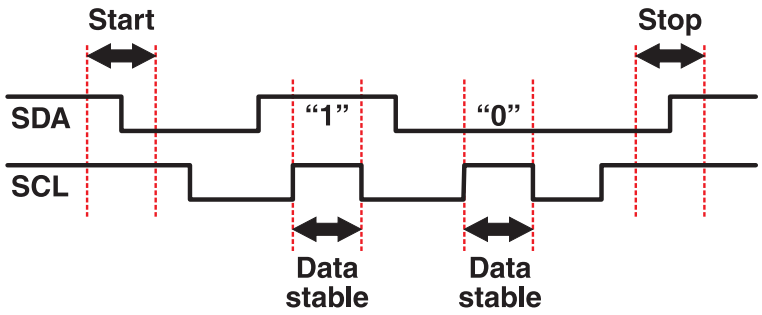
Siehe auch: I2CL_STOP, I2CL_START, I2CL_RELEASE, I2CL_READ, I2CL_WRITE, I2CL_RESULT.

I2CL_Start

I2CL_Start (speed)

Set Start Condition

Funktion: Erzeugt eine Start-Condition auf dem I2C-Bus / ISO-7816 Kanal.
Diese Funktion arbeitet als Bus-Master.



I2C-Bus / ISO 7816 Signale

Parameter:

	B	W	L	S	F	
Speed	●	●	●	-	-	0 = keine Verlangsamung, 1...20 = Geschwindigkeitsreduktion
						Funktionswert:
-	-	-	-	-	-	kein Funktionswert

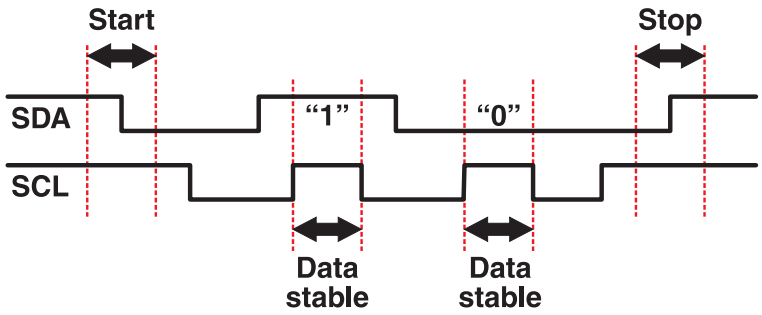
Siehe auch: I2CL_STOP, I2CL_RELEASE, I2CL_READ, I2CL_WRITE, I2CL_RESULT

I2CL_Stop

I2CL_Stop (speed)

Set Stop Condition

Funktion: Erzeugt eine Stop-Condition auf dem I2C-Bus / ISO-7816 Kanal.
Diese Funktion arbeitet als Bus-Master.



I2C-Bus / ISO 7816 Signale

Parameter:

	B	W	L	S	F	
Speed	●	●	●	-	-	0 = keine Verlangsamung, 1...20 = Geschwindigkeitsreduktion
						Funktionswert:
-	-	-	-	-	-	kein Funktionswert

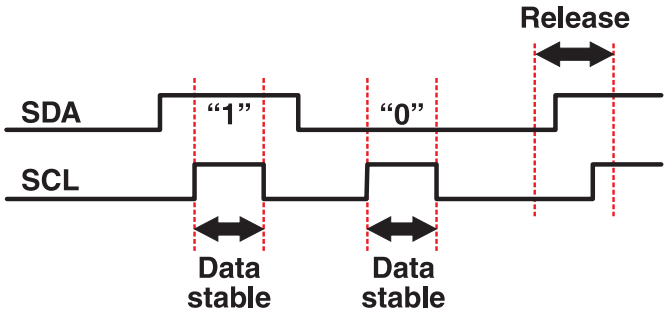
Siehe auch: I2CL_START, I2CL_RELEASE, I2CL_READ, I2CL_WRITE, I2CL_RESULT

I2CL_Release

I2CL_Release (Dummy)

Release SDA + SCL Lines

Funktion: Schaltet beide Bus-Leitungen in den Hi-Imedance Zustand, ohne dabei eine Stop-Condition zu erzeugen. Diese Funktion arbeitet als Bus-Master.



I2C-Bus / ISO 7816 Signale

Parameter:

	B	W	L	S	F	
Dummy	●	●	●	-	-	Parameter ohne Wirkung
						Funktionswert:
-	-	-	-	-	-	kein Funktionswert

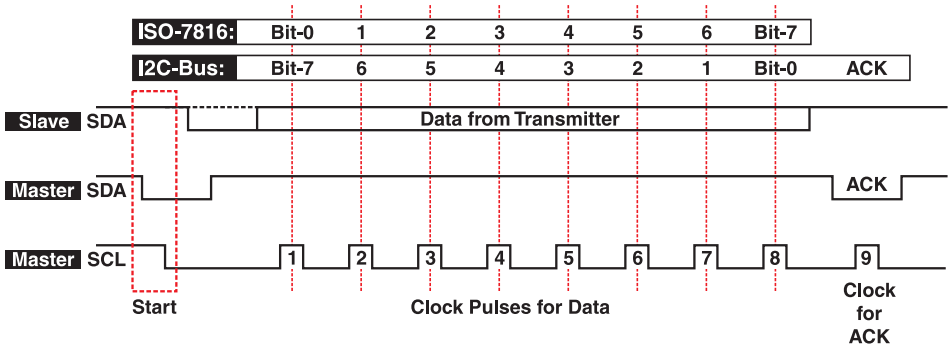
Siehe auch: I2CL_START, I2CL_STOP, I2CL_READ, I2CL_WRITE, I2CL_RESULT

I2CL_Read$

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

von I2C-Bus bzw. ISO 7816 lesen

Funktion: Liest von I2C-Bus / ISO 7816 Bus die spezifizierte Anzahl von Bytes ein. Diese Funktion arbeitet als Bus-Master.



I2C-Bus / ISO 7816 Read

Parameter:

	B	W	L	S	F	
nob	●	●	●	-	-	Anzahl Bytes die gelesen wereden sollen: 0 ... 32
						Funktionswert:
A$	-	-	-	●	-	Ergebnis String enthält die empfangenen Bytes

Die beiden Bus-Leitungen SCL und SDA (CLOCK und DATA) sind als „wired-and“ Signale ausgeführt, es gibt auf jeder Busleitung:

- einen Pull-Up Widerstand der den Leitungspegel nach +5V zieht, sowie
- Open-Collector Ausgänge der Busteilnehmer und
- hochohmige Eingänge der Busteilnehmer

Sind alle Busteilnehmer inaktiv, d.h. alle angeschlossenen Busteilnehmer beeinflussen eine Busleitung nicht, so liegt für alle Teilnehmer ein +5V Pegel = „1“ an.

Wenn ein oder mehrere Teilnehmer eine Busleitung mit dem Open-Collector Ausgang zum GND Pegel (0V) ziehen, so sehen alle Teilnehmer das Bussignal „0“ - „wired-and“. Das Pegel-Diagramm für I2C-Bus / ISO 7816 gibt diesen Umstand in der Form wieder, dass:

- nur 1 Clock-Signal dargestellt ist - Clock wird immer vom Master vorgegeben
- und 2 Data-Signale gezeigt werden, von jedem der beiden betroffenen Busteilnehmer, woraus man den Fluß der Infomation erkennen kann.

Die Signalübertragung auf dem I2C-Bus folgt keiner fixen Bit/s Rate, ist jedoch für manche Chips und Konfigurationen begrenzt. Entsprechend der Einstellung des „Speed“-Parameters in I2CL_SETUP (...) wird die kürzest mögliche Bitzeit beim I2CL_Read$ (...) berücksichtigt. Ansonsten erfolgt die Datenübertragung auf dem I2C-Bus rein statisch - kann also zu jedem Zeitpunkt angehalten und danach unverändert fortgesetzt werden.

Hinweis: Die ISO 7816 Übertragung verwendet noch eine RESET-Leitung zum Rücksetzen des Adr-Counters. Diese Leitung wird bei einer ISO 7816 Anwendung mit einem bel. Tiger-Pin oder Erweiterungspin realisiert und vom BASIC-Programm aus gesteuert.

Die „Answer-to-Reset“ (ATR) Prozedur ist standardisiert durch ISO 7816-3.

Verschiedene Programm-Beispiele sind verfügbar mit Namens-Präfix: „I2CL_“ und Extension „TIG“.

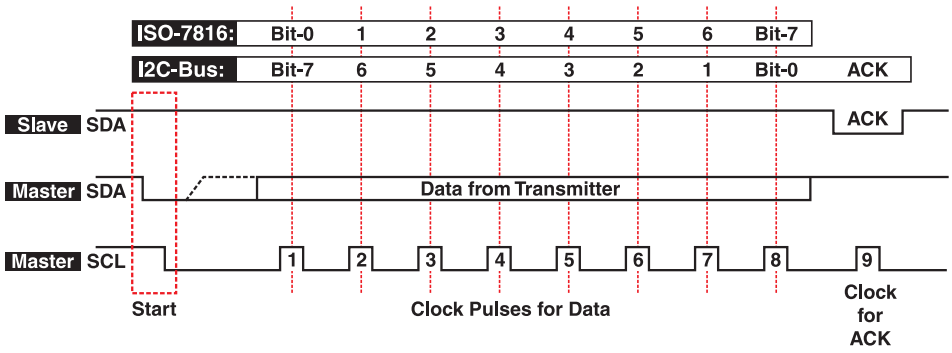
Siehe auch: I2CL_START, I2CL_STOP, I2CL_READ, I2CL_WRITE, I2CL_RESULT

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
```

auf I2C-Bus bzw. ISO 7816 schreiben

Funktion: Schreibt die spezifizierte Anzahl Bytes auf I2C-Bus / ISO 7816 Bus. Diese Funktion arbeitet als Bus-Master.



I2C-Bus / ISO 7816 Write

Parameter:

	B	W	L	S	F	
A$	-	-	-	●	-	Sendestring: 0 ... 32 Zeichen
N1	●	●	●	-	-	Numerischer Wert von dem 1, 2, 3 oder 4 Bytes gesendet werden
N2	●	●	●	-	-	Anzahl der Bytes, die gesendet werden sollen: 1...4 Bytes
7816	-	●	●	-	-	Kennung „ISO 7816“ Übertragungs-Format
						Funktionswert:
NAK	●	●	●	-	-	Anzahl der empfangenen ‹NAK› Conditions bei I2C-Bus Übertragung. Ohne Bedeutung bei ISO 7816 Übertragung.

Die beiden Bus-Leitungen SCL und SDA (CLOCK und DATA) sind als „wired-and“ Signale ausgeführt, es gibt auf jeder Busleitung:

- einen Pull-Up Widerstand der den Leitungspegel nach +5V zieht, sowie
- Open-Collector Ausgänge der Busteilnehmer und
- hochohmige Eingänge der Busteilnehmer

Sind alle Busteilnehmer inaktiv, d.h. alle angeschlossenen Busteilnehmer beeinflussen eine Busleitung nicht, so liegt für alle Teilnehmer ein +5V Pegel = „1“ an.

Wenn ein oder mehrere Teilnehmer eine Busleitung mit dem Open-Collector Ausgang zum GND Pegel (0V) zieht, so sehen alle Teilnehmer das Bussignal „0“ - „wired-and“. Das Pegel-Diagramm für I2C-Bus / ISO 7816 gibt diesen Umstand in der Form wieder, dass:

- nur 1 Clock-Signal dargestellt ist - Clock wird immer vom Master vorgegeben
- und 2 Data-Signale gezeigt werden, von jedem der beiden betroffenen Busteilnehmer, woraus man den Fluß der Infomation erkennen kann.

Die Signalübertragung auf dem I2C-Bus folgt keiner fixen Bit/s Rate, ist jedoch für manche Chips und Konfigurationen begrenzt. Entsprechend der Einstellung des „Speed“-Parameters in I2CL_SETUP (...) wird die kürzest mögliche Bitzeit beim I2CL_Read$ (...) berücksichtigt. Ansonsten erfolgt die Datenübertragung auf dem I2C-Bus rein statisch - kann also zu jedem Zeitpunkt angehalten und danach unverändert fortgesetzt werden.

Hinweis: Die ISO 7816 Übertragung verwendet noch eine RESET-Leitung zum Rücksetzen des Adr-Counters. Diese Leitung wird bei einer ISO 7816 Anwendung mit einem bel. Tiger-Pin oder Erweiterungspin realisiert und vom BASIC-Programm aus gesteuert.

Die „Answer-to-Reset“ (ATR) Prozedur ist standardisiert durch ISO 7816-3.

Verschiedene Programm-Beispiele sind verfügbar mit Namens-Präfix: „I2CL_“ und Extension „TIG“.

Siehe auch: I2CL_START, I2CL_STOP, I2CL_READ, I2CL_WRITE, I2CL_RESULT

I2CL_Result

```
Flag  =  I2CL_Result ( )
```

Funktion: Liest den Ergebnis Code der zuletzt ausgeführten I2CL-Funktion.

Parameter:

	B	W	L	S	F	
-	-	-	-	-	-	Kein Parameter
						Funktionswert:
Flag	●	●	●	-	-	Ergebnis-Code: 81H = Read OK 82H = Write OK 83H = Erase OK 84H = Setup OK F1H = Read Fehler F2H = Write Fehler F3H = Erase Fehler F4H = Setup Fehler FAH = Parameter Fehler FFH = I2C / Funktions Fehler

Insert$

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

Funktion: Fügt einen String (oder Teil davon) in einen String ein.

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

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


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

String-Insert

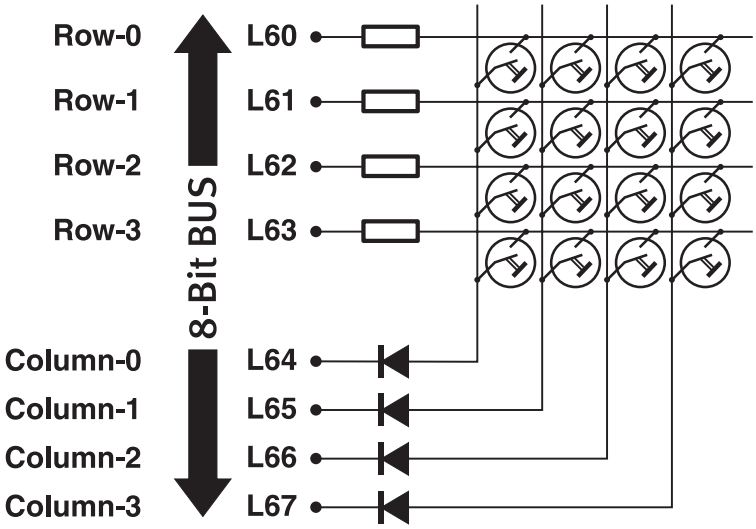
Parameter:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source Datenstring
S_Pos	●	●	●	-	-	Insert-Position in Source-String: 0 ... nnnn
Ins$	-	-	-	●	-	Insert-String, der eingefügt wird
I_Pos	●	●	●	-	-	Position in Ins$, ab der eingefügt wird: 0 ... nn
I_Len	●	●	●	-	-	Anzahl Zeichen aus Ins$, die eingefügt werden sollen.
						Funktionswert:
N$	-	-	-	●	-	Ergebnis-String mit eingefügtem String

Key_Direct

```
N  =  Key_Direct (Port, Column, Mode, Speed)  ' Lies Tasten-Spalte
N  =  Key_Direct (Port, Column, Mode)         ' Lies Tasten-Spalte
```

Funktion: Kleine Tastaturen ökonomisch realisieren. Hierzu dient die Key_Direct Funktion, die den Daten-Bus nutzt um bis zu 16 Tasten oder Schalter (z.B. DIPs) einzulesen. Es wird kein eigener Tiger Anschluß verwendet und alle Bus-Operationen können weiterhin gleichzeitig ausgeführt werden (LCD Ausgaben, xPort Ein-/Ausgaben, ... Dev-Driver ... etc.).



16 Tasten mit 4 x R und 4 x D anschließen

Parameter:

	B	W	L	S	F	
Port	●	●	●	-	-	Bus Port, z.B. 6 oder 8
Column	●	●	●	-	-	Spalte zum Einlesen
Mode	●	●	●	-	-	0 = Nr. der 1. gedrückten Taste einlesen: 0,1,2,3 Keine Taste gedrückt = -1 Weitere gedrückte Tasten: ignorieren

Key_Direct

Sparsame Keyboards - wenig Teile, keine extra Tiger-Leitungen

	B	W	L	S	F	
						1 = Lies 1 NIBBLE ein mit allen 4 Tasten: Wert „0“ = Taste/Schalter offen Wert „1“ = Taste/Schalter geschlossen.
Speed	●	●	●	-	-	0 = full speed 1...16 = verlangsamen für längere Leitung und/oder hohe Widerstände
						Funktionswert:
N	●	●	●	-	-	Key-Nr 0...3, -1 oder Nibble-Wert, je nach Mode

Um eine 4 x 4 Switch Matrix einzulesen werden einfach die 4 Nibbles der Reihe nach abgefragt:

Beispiel

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
```

Für eine Eingabe-Tastatur wird im Mode=0 gearbeitet. Folgendes Beispiel zeigt ein kleines Codestück, das als eigene Task eingesetzt werden kann und einen Tastatur-Puffer (KEY_BUF$) mit Zeichen füllt. Bei Verwendung üblicher Tasten-Elemente mit Druckpunkt-Kontakt, ist keine weitere Entprellung mehr erforderlich - die 60 ms Pause je Scan-Loop reichen bereits zur Entprellung aus. Bei schwierigeren Fällen sei auf existierende Funktionen wie „Debounce“ und die entsprechenden Beispiel-Programme hingewiesen.

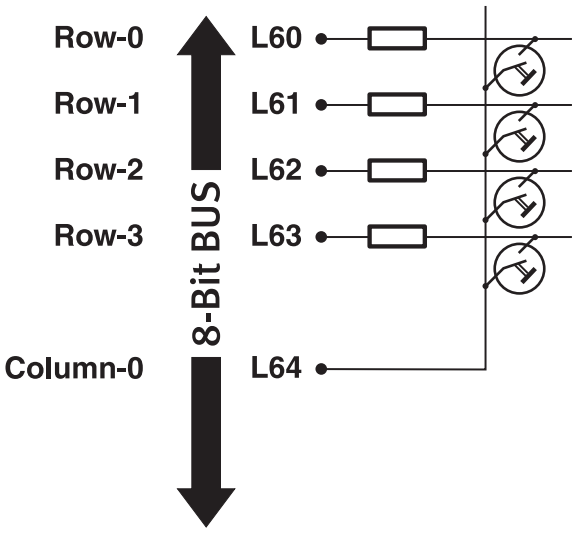
Beispiel

4 x 4 Tastatur

```
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 ---------->
```

Sparsame Keyboards - wenig Teile, keine extra Tiger-Leitungen

Je nach Anforderungen des Projektes kann jede Art der Teilbeschaltung eingesetzt werden wenn nur wenige Tasten zum Einsatz kommen:



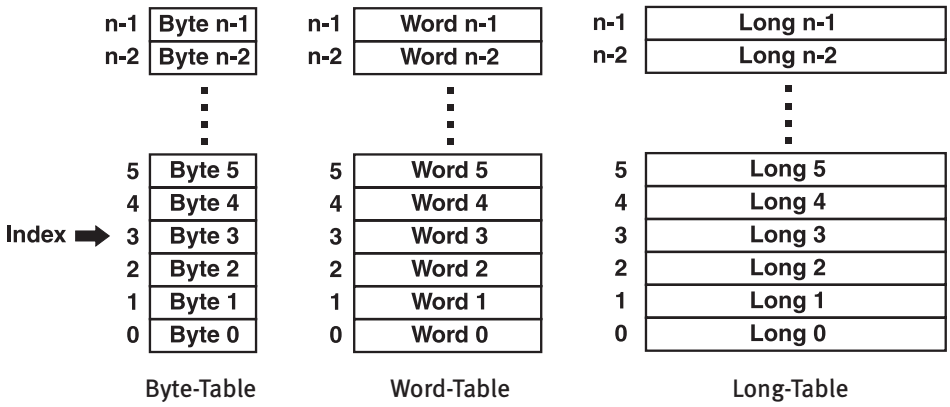
Tasten-Matrix in Teil-Bestückung

Ebenso ist eine beliebige Kombination von Tasten (dynamisch) und Switches (statisch) möglich. Hardwareseitig gibt es keine Unterscheidung zwischen Tasten und statischen Schaltern, die Scan-Routine behandelt beide Kontakt-Typen entsprechend.

BLookup#, WLookup#, LLookup#

Byte-Table: `A = BLookup# ( Index )`
Word-Table: `A = WLookup# ( Index )`
Long-Table: `A = LLookup# ( Index )`

Funktion: besonders schneller Zugriff auf „Look Up“ Tabellen im DATA-FLASH oder in Strings. Lookup wird eingesetzt um mathematische Funktionen, Kennlinien, Kalibrierungsfunktionen etc. über Tabellenzugriffe sehr schnell in Echtzeitanwendungen verfügbar zu machen.



Parameter:

	B	W	L	S	F	
Index	●	●	●	-	-	Index: 0 ... (n-1) für Tabellen-Zugriff
						Funktionswert:
A	●	●	●	-	-	Tabellenwert: Byte, Word oder Long

Es stehen jeweils 4 Lookup-Tabellen der Typen Byte, Word und Long zur Verfügung. Über den Funktionsnamen werden Typ und Tabellen-Nummer festgelegt, z.B:

BLookup2 => Byte-Table 2
LLookup4 => Long-Table 4

Lookup-Tables

WLookup1 => Word-Table 1
WLookup4 => Word-Table 4

Folgende 12 Tabellen stehen insgesamt zur Verfügung:

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 wird mit einem Index als Parameter aufgerufen:

Beispiel

```
A  =  WLookup4 ( Index )          ' greift auf Word-Table-4 zu
                                  ' Index: 0 ... n-1
```

Vor der Benutzung einer Lookup Table wird diese mit gültigen Werten initialisiert. Dies geschieht durch Zuweisung zu einem Table-String oder einem DATA-Flash Bereich.

String-Table initialisieren

```
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
```

Diese Zuweisung stellt eine Verknüpfung her zwischen:

BLOOKUP1 ‹==› A$

Entsprechend sieht die Initialisierung aus, wenn die Werte-Tabelle im DATA-Flash Bereich liegt:

FLASH-Table initialisieren

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

Ein Programm-Beispiel:

Beispiel

```
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                   '
```

Die Zuweisung des Strings A$ zur Lookup-Table-1 / Byte erfolgt durch Angabe von 2 Parametern:

1. Index (hier nur Dummy)
2. Ort der Lookup-Tabelle (String oder DATA-FLASH Bereich)

In diesem Beispiel dienen der Index wie auch der Funktionswert N nur als Platzhalter / Dummy. Für jeden nachfolgenden Zufriff auf diese Lookup-Tabelle wird die Kurzform dieser Funktion mit nur einem Parameter = Index gewählt:

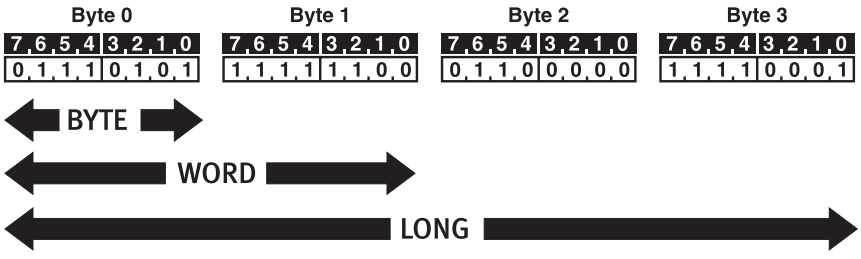
X = BLOOKUP1 (index)

Diese Notation bringt die schnellsten Laufzeit-Ergebnisse. Wenn mit einer großen Zahl von Lookup-Tabellen gearbeitet werden muß, so ist auch die 1. Form mit 2 Parametern: Index + Table_Spezifikation möglich. Diese Notation benötigt eine etwas längere Laufzeit.

Der Tabellen-Aufbau:

WORDs und LONGs in den Lookup-Tabellen werden - wie stets in Tiger-BASIC - als Big-Endian Format abgelegt:

low-order-byte first - highest byte last:



Lookup Tables

Die Verwendung von Lookup-Tabellen kann von großem Vorteil für die Systemleistung sein. Lookup-Tabellen Zugriffe laufen sehr schnell ab und können so für beträchtlichen Geschwindigkeitszuwachs sorgen wenn sie länger dauernde Berechnungen ersetzen.

Zum Beispiel im Bereich der Technik, in realen Steuerung und Grafik-Präsentationen ist häufig nicht gefragt mit 16-stelliger Genauigkeit zu arbeiten wie bei REAL Zahlen. Schon 3 oder 4 Dezimalstellen sind oft vollkommen ausreichend. Eher gefragt ist dort Geschwindigkeit.

Zum Beispiel bei der Steuerung von Antrieben, Koordinaten-Transformationen für Karten auf einem Graphic-Display, für schnelle Regelungen, ... etc.

Mathematische Funktionen beschleunigen

Man habe die Aufgabe in einer Regelung einen mathematischen Zusammenhang mit trigonometrischen Funktionen immer wieder zu berechnen um den Regelkreis betreiben zu können. Um die Geschwindigkeit dieses Vorgangs zu erhöhen, wählt man statt REAL Werten eine Abbildung auf LONG Werte, z.B. werde die Mess-Größe „bar“ auf eine LONG Zahl abgebildet als „micro-bar“. Der LONG Zahlenumfang ist groß genug um auch noch einen Wert von 1000 bar in micro-bar Einteilung darstellen zu können. Alle schnellen Rechenvorgänge werden dann soweit möglich in Tabellenform gebracht, die per Lookup 50(oder-mehr)-mal schneller ablaufen.

Solche Berechnungs-Tabellen kann man:

1.) Jedesmal bei Programmstart neu ausrechnen und in Strings schreiben (dauert).

2.) Beim 1.Start des Programms ausrechnen und in den DATA-Flash Bereich schreiben. Bei jedem späteren Programmstart liegt dann bereits eine gültige Tabelle vor.

3.) Berechnungstabelle wird das Daten-File zur Compilationszeit in den DATA-Flash Bereich des Programms eingebunden.

Ein Beispiel-Programm im Verzeichnis „Example“ / „Beispiele“ zeigt eine solche Anwendung.

Pin

```
A = Pin ( ADR, Bit_Pos )          ' lies 1 Bit in Port
```

Funktion: Liest einen einzelnen Pin eines Ports ein.

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	Port-Adresse
Bit_Pos	●	●	●	-	-	Bit-Position: 0 ... 7
						Funktionswert:
A	●	●	●	-	-	Eingelesenes Bit von Port, Wert: 0, 1

Programm-Beispiel:

```
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
```

Siehe auch XPort-System für erweiterte 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$
```

Funktion: Daten (Byte, Word, Long, Real, String) auf String PUSH-en und ebenso wieder vom String POP-en => zu Byte, Word, Long, Real, String.

Parameter:

	B	W	L	S	F	
A$	-	-	-	●	-	Stack-Bereich
NUM	●	●	●	-	-	Quell-Daten (Integer) zum PUSH-en
REAL	-	-	-	-	●	Quell-Daten (Real) = 8 Byte zum PUSH-en
Stri$	-	-	-	●	-	Quell-Daten (String) = nn Bytes zum PUSH-en
Pos	●	●	●	-	-	Start-Position in Stri$
Anz	●	●	●	-	-	Anzahl Bytes zum PUSH-en / POP-en
						Funktionswerte:
FLG	●	●	●	-	-	Anzahl tatsächlich ge-PUSH-te Bytes
N	●	●	●	-	-	ge-POP-tes numerisches Ergebnis
R	-	-	-	-	●	ge-POP-tes Real-Ergebnis
X$	-	-	-	●	-	ge-POP-tes String-Ergebnis

Stack-Struktur für Strings

Die Funktionen PUSH und POP für Datentypen Byte, Word, Long, Real und String gestatten den Aufbau von Stackstrukturen auf Strings. Im normalen Gebrauch von Stacks (FiLo = First-In, Last-Out) werden Daten in genau der umgekehrten Sequenz vom Stack heruntergeholt (POP) als sie früher auf den Stack geschoben worden sind (PUSH). Es können jedoch auch Datenkonvertierungen durch eine PUSH / POP Sequenz erzeugt werden (s.u.). Stacks gestatten den Aufbau rekursiver Strukturen, lokaler Variablen und Parameter.

Beispiel-Programmsequenz:

```
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
```

Man beachte die eingeschachtelte PUSH - POP Struktur. Ein Vertauscher würde die Konsistenz der Daten sofort zerstören.

Die PUSH-Funktion fügt Bytes an den String an - verlängert ihn also, POP verkürzt den String entsprechend. PUSH-Funktion und Wirkung auf den 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-Struktur für Strings

Integer und Real-Werte werden auch auf dem Stack als Big-Endian Darstellungen geführt:

```
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 werden Zeichen für Zeichen geführt ohne ihre Zeichenfolge dabei zu ändern. PUSH-en fügt Strings an den Stack an, POP-en entnimmt die entsprechende Anzahl von Zeichen von hinten und verkürzt den 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$, 5)               ' Pop 5 Chars to String from A$
```

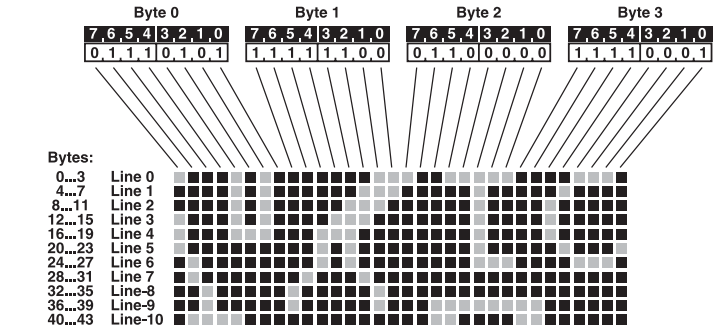
A$ = H e l l

X$ = o W o r

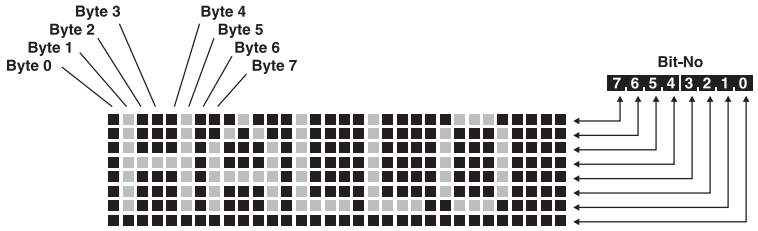
Graphic_Reformat

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

Funktion: kehrt die Pixel Strukur einer Pixel-Grafik um: waagerecht => senkrecht.



waagerechte Pixel-Spalten



senkrechte Pixel-Spalten

Parameter:

	B	W	L	S	F	
Src$	-	-	-	●	-	Source mit waagerecht orientierten Pixeln
Destin$	-	-	-	●	-	Destination mit senkrecht orientierten Pixeln
Width	●	●	●	-	-	Format-Breite in Pixeln: 1 ... nnnn
Height	●	●	●	-	-	Format-Höhe in Pixeln: 1 ... nnnn
Re_Format	●	●	●	-	-	Reformat-Flag: 0 ... 3

0: Bit-7 = links ➡ Bit-7 = unten
1: Bit-7 = rechts ➡ Bit-7 = unten
2: Bit-7 = links ➡ Bit-7 = oben
3: Bit-7 = rechts ➡ Bit-7 = oben

kein Funktionswert

Grafik-Pixel Reformatierung

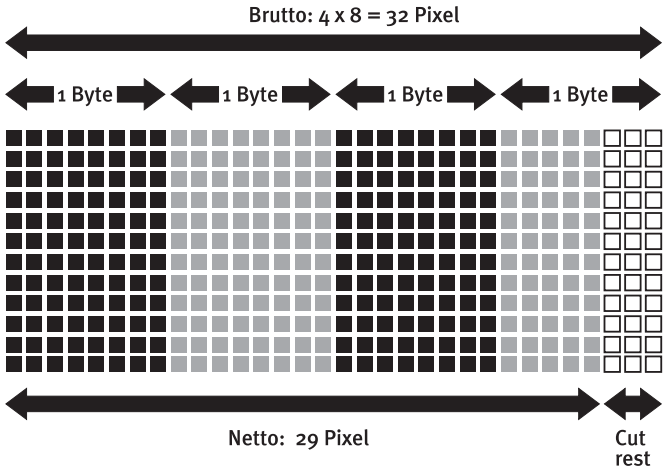
Graphic_Reformat wandelt Pixel-Daten in ihrer Struktur um.

Hier in Tiger-BASIC wird Pixel-Grafik in waagerechten Zeilen verwaltet. 1 Byte = 8 waagerechte Pixel, wobei Bit-7 das jeweils linke Pixel einer 8-er Gruppe repräsentiert. Diese Anordnung von Pixeln wird auch häufig in LCD-Displays und zeilenorientierten Monitoren und Druckwerken eingesetzt.

Eine weitere gängige Pixelorientierung orientiert sich an Matrix-Druckwerken, die 8 Bits eines Bytes repräsentieren jeweils 8 senkrecht übereinander liegende Pixel. Graphic_Reformat gestattet die jeweilige Umformatierung mit nur 1 BASIC-Zeile und kurzer Laufzeit in allen 4 möglichen Kombinationen:

waagerecht:	Bit-7	⟷	links	➡	senkrecht:	Bit-7	⟷ oben
waagerecht:	Bit-7	⟷	links	➡	senkrecht:	Bit-7	⟷ unten
waagerecht:	Bit-7	⟷	rechts	➡	senkrecht:	Bit-7	⟷ oben
waagerecht:	Bit-7	⟷	rechts	➡	senkrecht:	Bit-7	⟷ unten

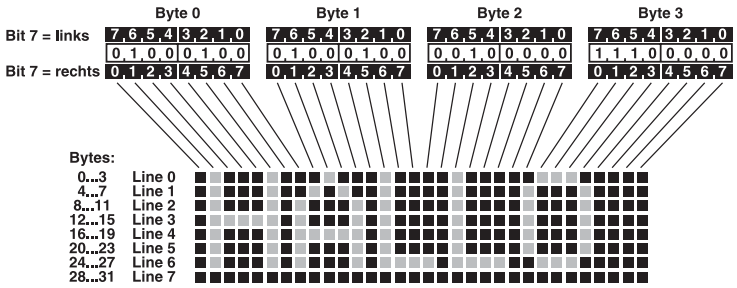
Graphic_Reformat bearbeitet alle Breiten und Höhen, auch, wenn diese nicht ganze Vielfache von 8 sind. Die entsprechenden Rest-Pixel am rechten oder unteren Rand fallen dann fort.



Beispiel: Format-Breite = 29 Pixel

Beispielprogramm „GRAPHIC_REFORMAT_Demo_001.TIG“:

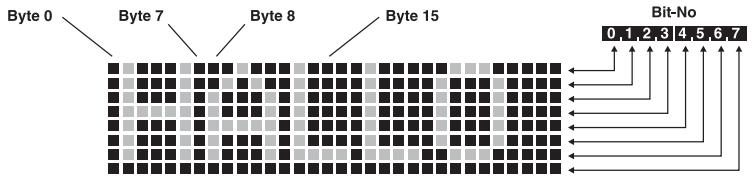
Nehmen wir als Ausgangs-Pixelstruktur die „Hallo“-Grafik aus der Funtionsbeschreibung:



Daraus ergeben sich bei waagerechten Pixelzeilen (mit Bit 7 links, wie bei einem grafischen LC-Display) folgende 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
```

Nun wandeln wir diese Pixelstruktur so um, daß die Bytes in senkrechten Pixelspalten angeordnet sind. Zunächst mit Reformat-Flag 0, wobei Bit 7 in der Ausgangsstruktur links ist und in der Zielstruktur unten. Dabei werden die Bytes wie in folgender Abbildung gebildet:

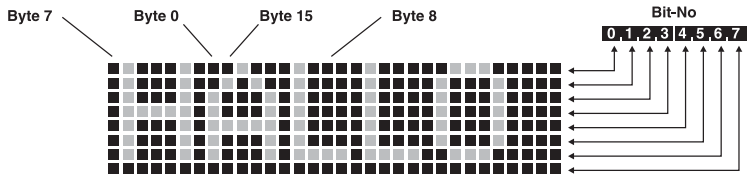


Graphic_Reformat (Src$, Dest$, 32, 8, 0)

Dabei ergeben sich aus den senkrechten Pixelspalten folgende Bytes (in Hex):

```
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
```

Jetzt wandeln wir die Original-Pixelstruktur mit Reformat-Flag 1 um, wobei Bit 7 in der Ausgangsstruktur diesmal rechts ist und in der Zielstruktur wieder unten. Dabei werden die Bytes wie in folgender Abbildung gebildet:

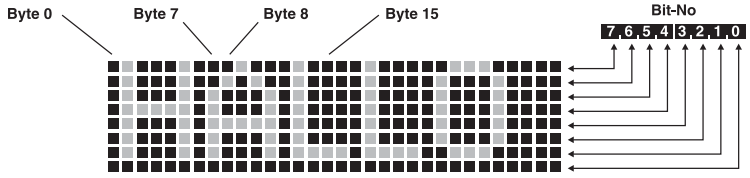


Graphic_Reformat (Src$, Dest$, 32, 8, 1)

Dabei ergeben sich aus den senkrechten Pixelspalten folgende Bytes (in Hex):

```
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
```

Diesmal verwenden wir Reformat-Flag 2, wobei Bit 7 in der Ausgangsstruktur links ist und in der Zielstruktur oben. Dabei werden die Bytes wie in folgender Abbildung gebildet:

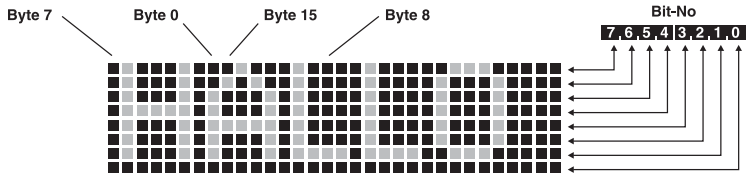


Graphic_Reformat (Src$, Dest$, 32, 8, 2)

Dabei ergeben sich aus den senkrechten Pixelspalten folgende Bytes (in Hex):

```
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
```

Zuletzt das Ganze mit Reformat-Flag 3, mit Bit 7 in der Ausgangsstruktur rechts und in der Zielstruktur oben. Dabei werden die Bytes wie in folgender Abbildung gebildet:



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

Dabei ergeben sich aus den senkrechten Pixelspalten folgende Bytes (in Hex):

```
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 )
```

Funktion: reformatiert ASCII-Text Strings für Ausgabemedien wie Terminals, LCDs, Drucker ... etc.

Parameter:

	B	W	L	S	F	
Source$	-	-	-	●	-	Source Text: ASCII, printables und ‹CR›‹LF› erlaubt
Dest_Width	●	●	●	-	-	Ergebnis-Zeilenlänge: printable Chars / Zeile
Dest_Cut_Wrap	●	●	●	-	-	Ergebnis-Zeilen: 0 = abschneiden, 1 = umbrechen
Dest_Fill_Blank	●	●	●	-	-	Ergebnis-Zeilen mit Blanks auffüllen: 0 = nein, 1 = ja
Dest_Line_End	●	●	●	-	-	Ergebnis Zeilen-Ende Flag: 0: inaktiv 1: setze ‹CR› ans Zeilenende 2: setze ‹CR›‹LF› ans Zeilenende
						Funktionswert:
Dest$	-	-	-	●	-	Reformatierter Ergebnis-String

Text_Reformat$ wird eingesetzt um Textausgaben für verschiedene Ausgabegeräte zu formatieren. Als Source-String wird ein ASCII-Text mit druckbaren Zeichen und ggf. ‹CR›‹LF› Codes erwartet.

Text_Reformat ermöglicht folgende Operationen:
- Zeilen-Länge neu festlegen
- Zeilen abscheiden
- Zeilen umbrechen
- Zeilen mit Blanks auffüllen
- ‹CR› ‹LF› am Zeilenende entfernen und/oder anfügen

Text_Reformat$ wird z.B. auch eingesetzt um partielles Scrollen in verschiedenen Fenstern auf einem Bildschirn zu ermöglichen.

Text_Reformat$ Text Reformatierung

Re-Formatierungs Beispiele:

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 )
```

Funktion: Sucht oder überspringt in Strings vorgegebene Zeichen-Kollektive. Scan_or_Skip wird typischerweise eingesetzt zum Aufbau von Kommando-Interpretern und zur Analyse von Ausdrücken und Datenstrukturen.

Parameter:

	B	W	L	S	F	
SRC$	-	-	-	●	-	Source-String
Charset$	-	-	-	●	-	Charset Flag-String, genau 256 Bytes Länge. Für jeden Zeichen-Code gibt es ein Flag im String: Flagwert 0: Code gehört *nicht* zum Zeichensatz. Flagwert X: Code gehört zum Zeichensatz.
Pos	●	●	●	-	-	Startposition im Source-String für Suchvorgang
Scan_Skip	●	●	●	-	-	Flag entscheidet über Suchart: positiver Wert -› SCAN negativer Wert -› SKIP
						Funktionswerte:
New_Pos	●	●	●	-	-	Gefundene Position lt. Suchkriterium: 0 ... nnn Das ist die Position auf das 1. derartige Zeichen. -1: Not Found, String Ende erreicht -2: Error, Leer-String oder Pos außerhalb SRC$

Scan_or_Skip untersucht einen Source-String auf das Vorhandensein bestimmter Zeichengruppen. Im Falle von „Scan“ wird ab einer vorgegebenen Startposition das Auftreten von Zeichen eines vorgegebenen Zeichensatzes gesucht. Sobald ein solches Zeichen gefunden wird ist der Suchvorgang abgeschlossen, die Position dieses Zeichens im Source-String wird als Funktionswert übergeben.

Beispiel:

```
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
```

Blank_Flag$ ist ein Flag-String von 256 Byte Länge, alle Bytes = 00, ausser das Byte an Position 32 (20H). Das ist das Flag für das ASCII „Space“ Zeichen. Dieser Flag-String definiert mithin einen Zeichensatz der nur aus dem Space-Zeichen besteht.

NPos hat nach dem Funktionsaufruf den Wert 3, das ist die Position des 1. gefundenen Zeichens des Zeichensatzes lt. Blank_Flag$.

Entsprechend könnte man nun den Zeichensatz erweitern auf „Trenner-Zeichen“ ganz allgemein, also z.B. alle Zeichen außer Buchstaben und Zahlen:

```
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 FF 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
```

Im Falle von „Skip“ wird ab einer vorgegebenen Startposition das Auftreten von Zeichen die NICHT Teil des vorgegebenen Zeichensatzes sind gesucht. Sobald ein solches Zeichen gefunden wird ist der Suchvorgang abgeschlossen, die Position dieses Zeichens im Source-String wird als Funktionswert übergeben.

Beispiel:

```
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
```

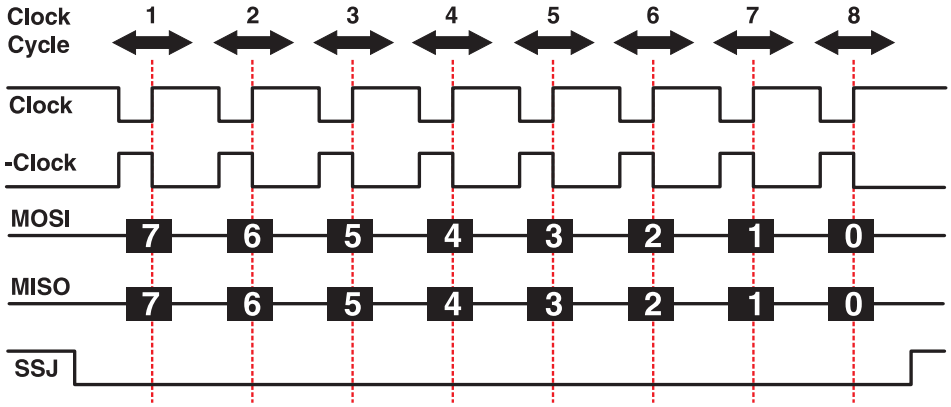
NPos hat nach dem Funktionsaufruf den Wert 2, das ist die Position des 1. gefundenen Zeichens das NICHT zum Zeichensatzes lt. Blank_Flag$ gehört.

SPI_SETUP

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

Spezifizieren der Pins für einen SPI-Bus

Funktion: Spezifiziert den SPI-Bus für die Funktion: SPI_IO$ (...).



SPI Kommunikation

Parameter:

	B	W	L	S	F	
CLK_MOSI_Port	●	●	●	-	-	Interner Port für Output-Signale: CLK + MOSI
CLK_Pin	●	●	●	-	-	Clock-Output Pin: (Bit-Nr: 0...7) = Clock generated by Master
MOSI_Pin	●	●	●	-	-	DATA-Output Pin: (Bit-Nr: 0...7) = Master-OUT, Slave-IN
SSJ_Port	●	●	●	-	-	Interner Port für Output-Signal: SSJ
SSJ_Pin	●	●	●	-	-	SSJ-Output Pin: (Bit-Nr: 0...7) = Low-Active CE für SPI-Device
MISO_Port	●	●	●	-	-	Interner Port für Input-Signal: MISO
MISO_Pin	●	●	●	-	-	DATA-Input Pin: (Bit-Nr: 0...7)
MSB_First	●	●	●	-	-	Flag: 0=LSB-first, X=MSB-first
						Funktionswert:
OK_FLAG	●	●	●	-	-	0 = alles OK!, X = 1...n: n-ter Parameter fehlerhaft

SPI_SETUP

Serial Chip Interfacing

Das SPI_SETUP wird im Programmablauf einmal ausgeführt und legt die I/O Konfiguration für alle anschliessenden SPI-Zugriffe fest.

Beispiel:

Beispiel

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

Diese Funktionszeile setzt folgende Festlegungen für SPI-Kommunikation in Kraft:

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

-1 ==› MSB-first

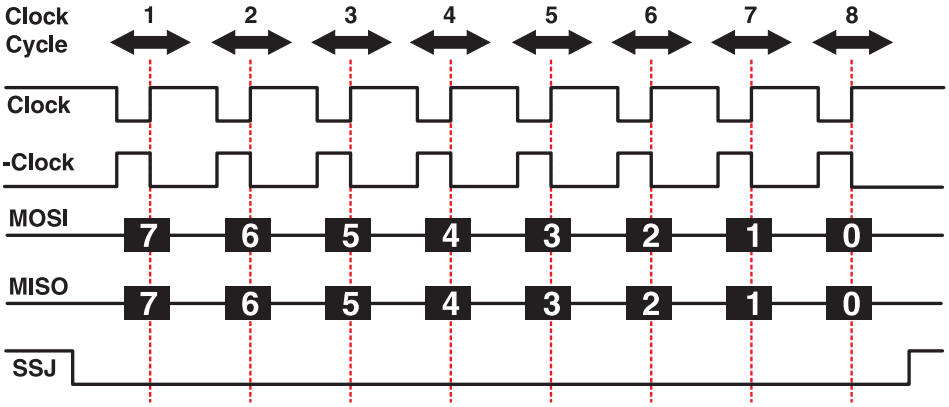
Siehe auch: SPI_IO$

SPI_IO$

```
Rec$ =  SPI_IO$ ( Transm$ )
```

Kommunikation über SPI-Bus

Funktion: Sendet und empfängt gleichzeitig Daten über PSI Interface wie in SPI_SETUP spezifiziert.



SPI Kommunikation

Parameter:

	B	W	L	S	F	
Transm$	-	-	-	●	-	Sende-String, wird über den mit „SPI_SETUP“ definierten SPI-Kanal an ein externes Device übertragen. Die Länge dieses Strings legt fest wieviele Bytes an Daten sowohl gesendet, als auch empfangen werden. String-Längen: 0 ... max. Stringlänge
						Funktionswert:
Rec$	-	-	-	●	-	String mit empfangenen Datenbytes. Rec$ hat im Anschluß an diese Operation die gleiche Länge wie Transm$. Die maximale Länge des Rec$ Strings laut Deklaration legt die maximale Länge fest, die mit einem Funktionsaufruf empfangen werden kann.

Ein Beispiel:

Beispiel

```
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
```

Das am SPI-Kanal angeschlossene Device hat folgende Bytefolge erhalten:

„1234567890“

Das Device habe folgende Bytes gesendet:

„the quick “

Diese Bytefolge ist dem String Rec$ zugewiesen worden.

Siehe auch: 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 )
```

Setze um:

Funktion: Universal String Konverter mit Such-String Liste und Replace-String Liste.

Such-Worte in Replace-Worte

Such-String Liste:

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

Replace-String Liste:

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

```
"."  =  Füllzeichen
```

```
"-"  =  Füllzeichen
```

Src$:

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

Dest$:

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

Konversion Source => Destination

Universal_Convert$ durchsucht an jeder Source-String Position die ganze Liste an Such-Ausdrücken. Gibt es Übereinstimmung mit einem Such-Ausdruck, findet sofort die Ersetzung durch den Replace-Ausdruck statt und der Suchvorgang wird hinter dem eingefügten Replace-Ausdruck fortgesetzt. Gibt es keine Übereinstimmung, wird die

nächste Byte-Position im Source-String aufgesucht und es werden wieder alle Such-Ausdrücke durchsucht. Bis das Source-String Ende erreicht wird.

Die Reihenfolge der Such-Ausdrücke in der Such-Liste hat Bedeutung in zweierlei Hinsicht:

a) Konversions-Geschwindigkeit.
Häufig auftretende Such-Ausdrücke im Source-String sollten möglichst weit oben in der Liste auftauchen. Das reduziert den Such-Aufwand, beschleunigt die Konversion.

b) Konversions-Logik.
Was zuerst gefunden wird, wird zuerst konvertiert. Das hat wesentliche Auswirkungen auf das Ergebnis (siehe Beispiel-2).

Parameter:

	B	W	L	S	F	
Src$	-	-	-	●	-	Eingangs-String der konvertiert werden soll
Search_List$	-	-	-	●	-	String mit einer Liste von Such-Ausdrücken. Format: 1. Byte = Feld-Länge 2. Byte = Füllzeichen - wird ignoriert ! Es folgen Felder konstanter Länge mit Such-Ausdrücken, die ggf. mit Füllzeichen hinten aufgefüllt sind.
Replace_List$	-	-	-	●	-	String mit einer Liste von Replace-Ausdrücken. Gleiches Format wie Such-Liste
Start	●	●	●	-	-	Optionale Start-Position in Src$ für den Konversions-Vorgang.
Len	●	●	●	-	-	Optionale Längenangabe für den Such-Bereich in Src$.
						Funktionswert:
Dest$	-	-	-	●	-	Konvertierter Ergebnis-String

Es folgt der Programm-Code für das Konversions-Beispiel oben.

Beispiel-1

```
Search_List$ = "&       '
<9>.&                   ' Feld-Länge = 9, Füllzeichen = "."
car......&              '  1. Suchwort
ship.....&              '  2. Suchwort
airplane.&              '  3. Suchwort
bus......&              '  4. Suchwort
boat.....&              '  5. Suchwort
and......&              '  6. Suchwort
or.......&              '  7. Suchwort
but......&              '  8. Suchwort
yes......&              '  9. Suchwort
no......."              ' 10. Suchwort

Replace_List$ = "&
<16>-&                  ' Feld-Länge = 16, Füllzeichen = "-"
Kraftfahrzeug---&       '  1. Replace Wort
Schiff----------&       '  2. Replace Wort
Flugzeug--------&       '  3. Replace Wort
Bus-------------&       '  4. Replace Wort
Boot------------&       '  5. Replace Wort
und-------------&       '  6. Replace Wort
oder------------&       '  7. Replace Wort
aber------------&       '  8. Replace Wort
ja--------------&       '  9. Replace Wort
nein------------"       ' 10. Replace Wort

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.
```

Beispiel-2

Vorsicht!

```
Search_List$ = "&       '
<9>.&                   ' Feld-Länge = 9, Füllzeichen = "."
Sgr......&              '  1. Suchwort
Sge......&              '  2. Suchwort
SgH......&              '  3. Suchwort
SgDuH....&              '  4. Suchwort
Mfg......&              '  5. Suchwort
Mfgs.....&              '  6. Suchwort
Hvl......&              '  7. Suchwort
HGs......&              '  8. Suchwort
AlG......&              '  9. Suchwort
Bb......."              ' 10. Suchwort

Replace_List$ = "&
<31>-&                  ' Feld-Länge = 31, Füllzeichen = "-"
Sehr geehrter-----------------&       '  1. Replace Wort
Sehr geehrte------------------&       '  2. Replace Wort
Sehr geehrte Herren-----------&       '  3. Replace Wort
Sehr geehrte Damen und Herren--&      '  4. Replace Wort
Mit freundlichem Gruss--------&       '  5. Replace Wort
Mit freundlichen Grüssen------&       '  6. Replace Wort
Hochachtungsvoll--------------&       '  7. Replace Wort
Herzliche Grüße---------------&       '  8. Replace Wort
Alles Gute--------------------&       '  9. Replace Wort
Bis bald----------------------"       ' 10. Replace Wort

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"
```

Grundsätzlich kann man mit Universal_Convert$ in beide Richtungen konvertieren,

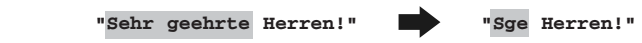
also im Beispiel-2 von der Kurzform in die Langform und wieder zurück. Dazu vertauscht man einfach die Such- und Ersetzungs-Listen:

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

Man beachte allerdings, dass diese Definition der Such- und Replace-Listen einen logischen Fehler enthält. Man erkennt ihn wenn folgende Konversion und die entsprechende Rück-Konversion ausgeführt wird:

Die Konversion:

Die ***Rück***-Konversion:



Diese Konversions-Anweisung ist zwar eindeutig, aber nicht ein-eindeutig.

In diesem Fall ist Abhilfe möglich, einfach durch Ändern der Reihenfolge der Such- und Replace-Ausdrücke in den Listen. Man stellt die längeren Ausdrücke vor die kürzeren um voreilige Konversion von „Sehr geehrte“ zu verhindern:

Search_List$:

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

"." = Füllzeichen

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----------------------”
```

"-" = Füllzeichen

Universal_Convert$ wird typischerweise eingesetzt für:

- adaptive Daten-Kompressionen
- Applikations-spezifische Daten-Kompressionen, z.B. hochkomprimierte Datenbanken in Embedded Systemen mit Klartext-Meldungen:

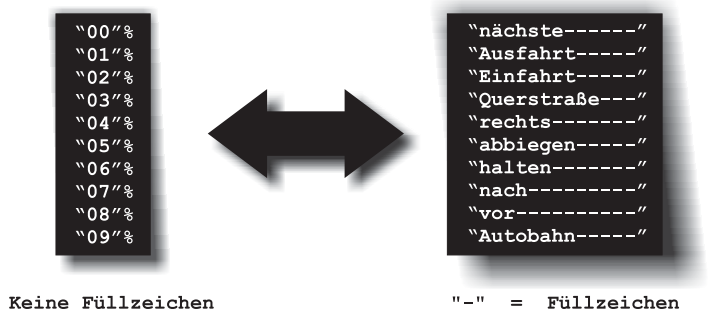
Typische Anwendungen:

komprimierte Datenbanken

Sicherheit

Code-Converter

Geräte- und Interface-Anpassungen



- Zeichensatz-Anpassungen:

  Zeichen => Zeichen
  Zeichen => Zeichen-Sequenz
  Zeichen-Sequenz => Zeichen
  Zeichen-Sequenz => Zeichen-Sequenz

- Geräte-Anpassungen
- Code Unterdrückung, Filterung, Ersetzungen
- Sicherheits-Anwendungen, Chiffrierungen

UNPACK_DC$

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

Dekomprimieren und in 2 Kanäle aufspalten:

Funktion: Dekomprimiert einen Source-Datenstrom und trennt ihn in 2 Kanäle auf: DATA-Kanal und CTRL-Kanal.

DATA-Kanal + CTRL-Kanal

UNPACK_DC$

Parameter:

	B	W	L	S	F	
CTRL$	-	-	-	●	-	Destination-String mit dekomprimierten CTRL-Informationen.
SRC$	-	-	-	●	-	Source-String
POS	●	●	●	-	-	Start-Position im Source-String: 0 ... nnnn
LEN	●	●	●	-	-	Maximale Länge aus dem Source-String, 0=bis String-Ende. Die Umsetzung ist in jedem Fall beendet bei String-Ende.
FLAG	●	●	●	-	-	Flag-Code, 1 Byte: 00H ... 0FFH
METHOD	●	●	●	-	-	Auswahl-Parameter PACK / UNPACK-Methode: 0 = Pack-Methode „0“, Details: s.u.
						Funktionswert:
DATA$	-	-	-	●	-	Destination-String mit dekomprimierten DATA-Informationen

Die Funktion wird typischerweise eingesetzt um 2 logische Kanäle über einen einzigen Kanal zu übertragen und hierbei auch von einem einfachen Kompressionsverfahren Gebrauch zumachen. Dies können alle Arten von Speichermedien sein (interner Speicher, EEProms, SRams, SmartMedia, ...), wie auch jede Form der Datenübertragung (RS-232 / 485, CAN-Bus, Ethernet ... etc.).

Komprimierungs- und Multiplex-Methoden

Methode-0:

Verpackung von CTRL-Informationen im Datenstrom:

‹FLag› BYTE Flag-Byte signalisiert den Beginn einer Gruppe von CTRL-Informationen (Bytes).

‹LEN› BYTE Anzahl der nachfolgenden CTRL-Bytes:
LEN = 01...7F --› es folgen 1...127 CTRL-Bytes. Werden mehr CTRL-Bytes gebraucht, schliesst sich eine weitere ‹Flag›‹Len› ...Byte... Sequenz an.

Beispiel:
„the quick ‹Flag›‹7›1234567brown fox“

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

Sonderfall (1): "Transparenz"
LEN = 00 --› 1 x Flag-Code im DATEN-Strom oder im CTRL-Byte-Strom

Beispiele: ‹Flag› = „A“
„hello world A‹0›“
=› DATA$ = „hello world A“
=› CTRL$ = „“ ‘ leer

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

Sonderfall (2): "Einfache Kompression" in DATA-Kanal

3 Byte Sequenz im Datenstrom, das den Code des zu expandierenden DATA-Zeichens angibt:

‹Flag› ‹LEN: 80...FF› ‹Code›

LEN = 80 --› 4 x CODE im DATEN-Strom
LEN = 81 --› 5 x CODE im DATEN-Strom

UNPACK_DC$ **Source-Datenströme dekomprimieren**

LEN = 82 --› 6 x CODE im DATEN-Strom
.
LEN = FF --› 131 x CODE im DATEN-Strom

Der Status zu Beginn der UNPACK_DC$ Funktion ist:

DATA, kein "eingepackter" Code

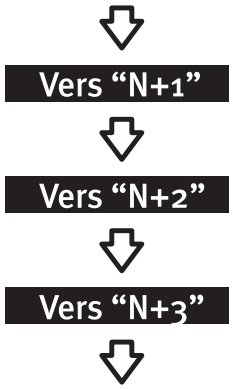
Vergleiche Funktionen: PACK$ und UPACK$

Update_Me

ECODE = UPDATE_ME (Data_Label, Option)

Funktion: ersetzt ein bestehendes Programm durch eine nachfolgende Programm-Version (Update) und startet dieses Programm.

Der Update-Vorgang geschieht unter voller Kontrolle des Anwendungs-Programmes. Die aktuelle Anwendungs-Programm Version kontrolliert das Update auf die nächste Programm-Version. Die nächste Programm-Version kontrolliert dann später das Update auf die übernächste Version ... usf..



Parameter:

	B	W	L	S	F	
Data_Label	●	●	●	-	-	Gibt die FLASH-Adresse an, ab der im DATA-FLASH Bereich des Tigers eine neue Programm-Version liegt: 0 ... nnnn
Option	●	●	●	-	-	Immer „0“.
						Funktionswert:
ECODE	●	●	●	-	-	Ganzzahliger Funktionswert für Fehler-Code:

Fehler-Code:	Bedeutung:
0	OK. Kommt jedoch nicht vor, da in diesem Fall die neue Programm-Version sofort einen Warmstart ausführt und nicht wieder zum Funktionswert zurückkehrt.
1	Diese Tiger-Modul Version unterstützt diese Funktion nicht.
2	RAM-Größe nicht richtig: die neue Programm-Version ist für andere RAM-Größe compiliert.

3	FLASH-Größe nicht richtig: die neue Programm-Version ist für andere FLASH-Größe compiliert.
4	Neue Programm-Version ist zu groß - paßt nicht in dieses Modul.
5	Die neue Programm-Version liegt im FLASH nicht komplett vor, oder es gibt Datenfehler im File.
6	Neue Programm-Version in DATA Area ist mit zu altem Compiler generiert worden.
7	Keine solche log-FLASH-ADR in diesem Modul (bei diesem Prog).
8	RAM Größe nicht OK.
9	FLASH Sektor Größe nicht OK.
10	Das neue Programm ist nicht für Project-Model "PM_FULL" compiliert worden (muss es aber).
11 ... 20	File-Error: Längen oder CRC Fehler in neuer Programm-Version im DATA-FLASH.
103	zu große FLASH-Adresse im Programm gefunden, entweder ist Programm zu lang oder es wurde bei zu hoher ADR im Data-Flash gespeichert.
andere	Alle anderen Fehler weisen auf fehlerhaften oder unvollständigen Programm-Code hin. Überprüfen Sie in diesem Fall auf korrekte Übertragung und Abspeicherung des neuen Programms.

Wenn keiner der genannten Fehler vorliegt wird der Update-Vorgang sofort ausgelöst. Die Programmausführung kehrt dann nicht wieder zu dieser Funktion zurück, sondern die neue Programm-Version wird direkt neu gestartet (Warmstart). Je nach Modulgröße dauert dieser Vorgang einige Sekunden bis ca. 1 Minute.

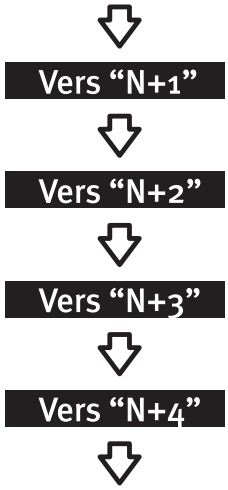
Remote Software Updating

Die Funktion Update_Me (...) ermöglicht das selbsttätige updaten von Anwendungen unter voller Kontrolle des Anwendungsprogrammes. Der Vorgang läuft in 3 Phasen ab:

1.) Nachfolgende Generation des Anwendungsprogrammes an Tiger System übertragen.

2.) Nachfolgende Generation des Anwendungsprogrammes im DATA Flash Bereich des Tigers ablegen.

3.) Update_Me (...) ausführen, die nachfolgende Programm-Generation übernimmt die Kontrolle.



Diese Methode des Remote Updates ermöglicht eine flexible Anpassung an die Erfordernisse der jeweiligen Anwendung. Insbesondere können typische Anforderungen realisiert werden wie:

- Verschiedene Übertragungs-Kanäle für Updates:
  Rundfunk, Packet Radio, Serial, Ethernet, Web, Punkt-zu-Punkt, SmartMedia FlashCards, IR-Verbindung, ... u.v.m.

- Sicherheit:
  Berechtigung zum Updaten voll unter Kontrolle des Anwendungsprogrammes. Paßword-Schutz, PIN- und/oder TAN (Transaction-Nr) Schutz, fehlerkorrigierende Codes / Protokolle auf dem Übertragungsweg, Chiffrierung, Sicherheitscodes, ... etc.

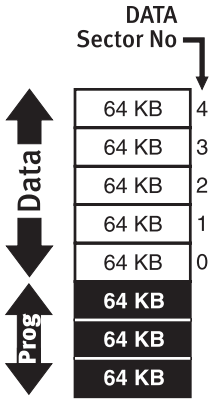
- Update Steuerung:
  Updates können im Tiger-System bereits lange vor ihrer Aktivierung vorgehalten werden, zeitgesteuerte Updates, automatische Updates durch Freischaltcodes, Abonement-Verlängerung, ... etc.

- Schnelles Updating, keine Extra-Hardware
  Wenn die nächste Programmversion im DATA-Flash Bereich vorliegt, dauert der eigentliche Update-Vorgang nur einige Sekunden bis ca. 1 Min. In dieser Zeit wird das aktuelle, gerade ablaufende Anwendungsprogramm gestoppt, gelöscht und durch die nachfolgende Programmversion ersetzt. Dann übernimmt die neue Programmversion die Ausführung mit einem Warmstart.

Da der eigentliche Update Vorgang erst ausgeführt wird, wenn eine vollständige, korrekte Programmversion im Speicher vorliegt, entsteht nur eine minimale Totzeit des Systems in der die Steuerung der Anwendung ruht. Störungen auf dem Übertragungswege haben daher keinen Einfluß auf die Totzeit des Systems.

Wie funktioniert der Update Vorgang im Tiger-System?

Hierzu die schematische Darstellung des FLASH-Speichers im Tiger mit seinen beiden Bereichen für Programmcode und frei verfügbarer DATA-Flash:



Typische FLASH Konfiguration bei 512 kByte FLASH-Größe

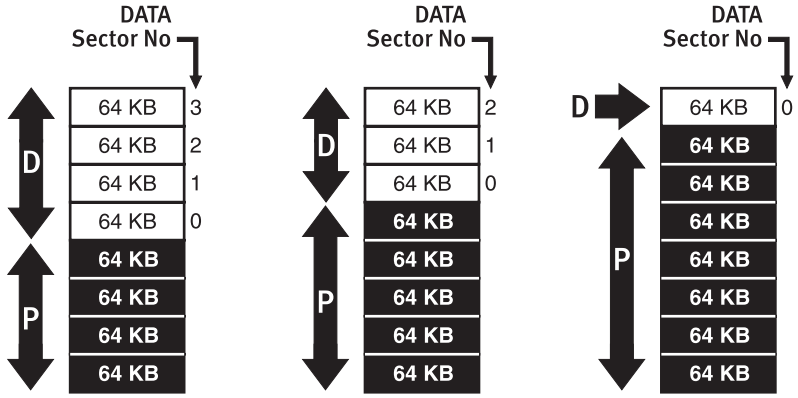
Ein Tiger-Programm belegt entsprechend seiner Größe eine Anzahl von FLASH-Sektoren. Nicht belegte FLASH-Sektoren, können vom Programm frei verwendet werden als DATA-FLASH Bereich. In diesem Bereich können Daten beliebig geschrieben, gelesen und ganze Sektoren gelöscht werden. Dem Tiger-BASIC Programm stehen hierzu eine Reihe von Instruktionen/Funktionen zur Verfügung (PEEK_FLASH, POKE_FLASH, POKEM_FLASH, ERASE_FLASH, ... siehe FLASH-Funktionen).

Die Größe des DATA-FLASH Bereiches hängt ab von der Programm-Länge wie auch von der Speichergröße des eingesetzten BASIC-Tiger Computers. Ein Tiger-BASIC Programm kennt Startadresse, Größe des DATA-FLASH Bereich sowie weitere Werte zur Speicher-Situation durch diese System-Variablen:

```
0                                   ' DATA-FLASH Start ADR, fix, immer = 0

SYVARN (SYSVN_FLASH_SSIZE,0)        ' 35, 0: FLASH Sektor-Größe
SYVARN (SYSVN_FLASH_ASEC, 0)        ' 36, 0: Anzahl aller FLASH-Sektoren
SYVARN (SYSVN_FLASH_GSIZE , 0)      ' 37, 0: Gesamt-Größe FLASH-Speicher
```

```
SYVARN (SYSVN_FLASH_DSEC, 0)      ' 38, 0: Anzahl verfügbare DATA-Sektoren
SYVARN (SYSVN_FLASH_DSIZE, 0)     ' 39, 0: Größe DATA-FLASH-Area
```

Man beachte den jeweils entstehenden Adress-Raum der DATA Area im FLASH Speicher des Systems abhängig von unterschiedlichen Programm-Längen und Modul-Größen:



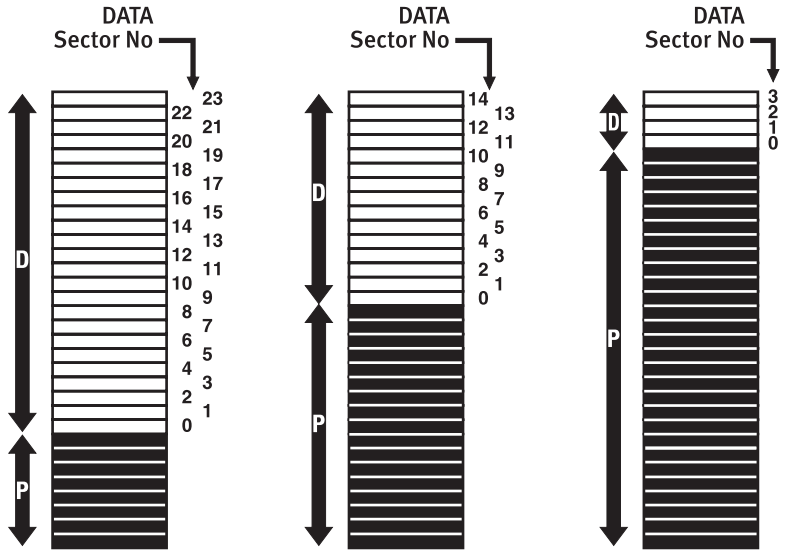
	(a)	(b)	(c)
FLASH-Size:	512 KB	512 KB	512 KB
Sector-Size:	64 KB	64 KB	64 KB
Base ADR Data:	0	0	0
Last ADR Data:	4000H-1	3000H-1	1000H-1
DATA-FLASH Size:	4000H	3000H	1000H

Im Fall (b) z.B. belegt der Programm Code 5 Sektoren a 64 kByte = 320 kByte.
An DATA Area im FLASH verbleiben 3 Sektoren a 64 kByte = 192 kByte.

Obwohl in den 3 Fällen (a) ... (c) der DATA-FLASH Bereich jeweils an verschiedenen Stellen des Speichers residiert, beginnt der Adress-Raum stets mit der ADR = 0 und lediglich die verfügbare DATA-FLASH Größe unterscheidet sich.

Entsprechend liegen die Verhältnisse bei anderen FLASH-Größen (siehe folgende Seite).

FLASH Adr-Raum für Programm-Code und DATA-FLASH Bereich bei 2 MByte FLASH Speicher mit 64 kByte Sektoren:



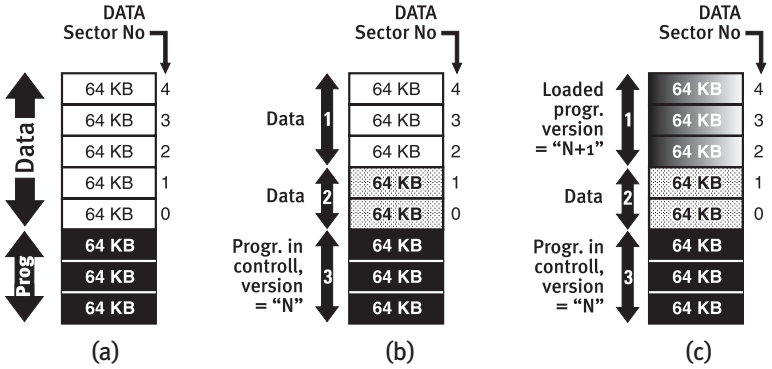
(a) (b) (c)

	(a)	(b)	(c)
FLASH-Size:	2 MB	2 MB	2 MB
Sector-Size:	64 KB	64 KB	64 KB
Base ADR Data:	0	0	0
Last ADR Data:	18000H-1	F000H-1	4000H-1
DATA-FLASH Size:	18000H	F000H	4000H

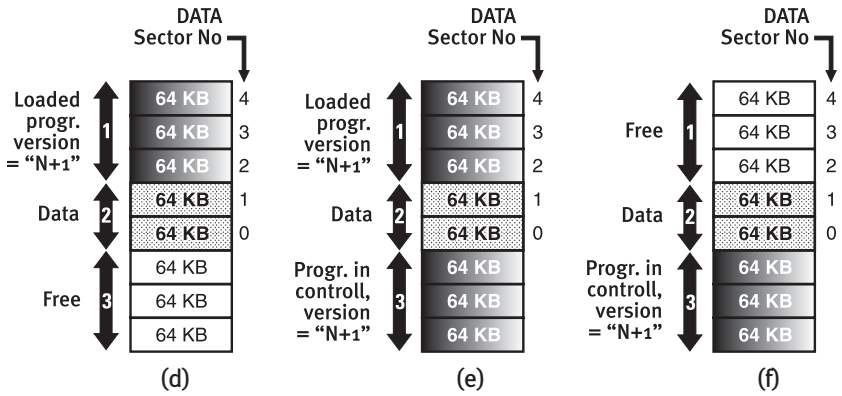
Sofern eine Anwendung nun über ausreichend freien FLASH-Speicherplatz verfügt ist es möglich den DATA-FLASH Bereich als Speicher für eine neue Programm-Version zu nutzen um anschließend ein Software-Update im BASIC-Tiger auszulösen.

Die folgende Darstellung, zeigt schematisch die Phasen:

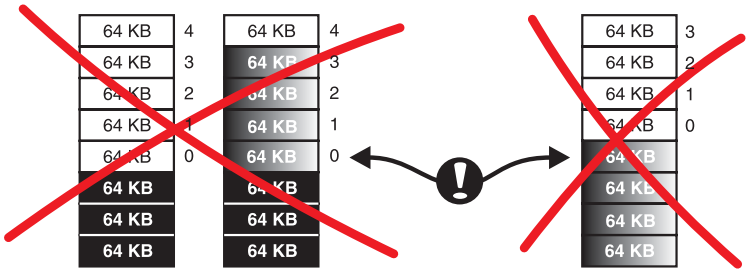
(a) Speichersituation zu Beginn: Bereiche für Programm-Code und DATA-FLASH.
(b) Vor dem Laden einer neuen Programm-Version:
DATA-FLASH in 2 Bereiche unterteilt: Data-1 reserviert für neue Programm-Version, Data-2 verfügbar für beliebige andere Daten.
(c) Nach dem Laden einer neuen Programm-Version („N+1“) in Data-1 Bereich, Daten in FLASH-Bereich Data-2 bleiben unverändert.



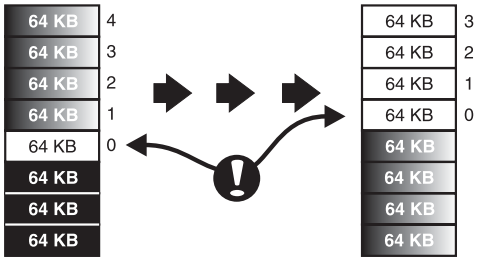
(d) Nach dem Start der Funktion „Update_Me (...)“ wird zunächst das aktuelle Programm (Version „N“) gelöscht
(e) und dann die Version „N+1“ in den Code Bereich des FLASHs kopiert und gestartet.
(f) Unter Kontrolle der neuen Programm-Version „N+1“ wird der obere Bereich des FLASHs (Data-1) wieder gelöscht (vom Anwender-Programm der Version „N+1“) und steht für zukünftige Updates oder andere Aufgaben bereit.

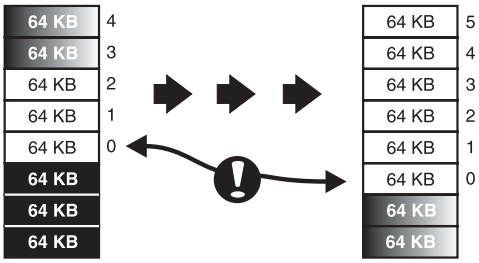


Bei der Festlegung der FLASH-Speicher Aufteilung ist zu beachten, dass sich Speicherbereiche für die neue Programm-Version *nicht* überlappen dürfen:

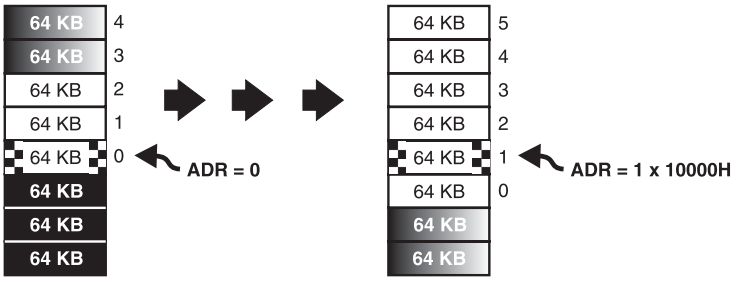


Ferner verändert sich die Größe und Positionierung des DATA-FLASH Bereiches immer dann, wenn die neue Programm-Version eine andere Größe hat als die aktuelle Programm-Version:

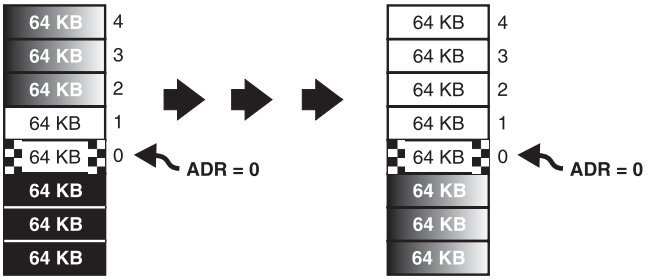




Unterschiedliche Programm-Längen von Vorgänger-Version zu Nachfolger-Generation führt zu verschobenen Datenbereichen (a):

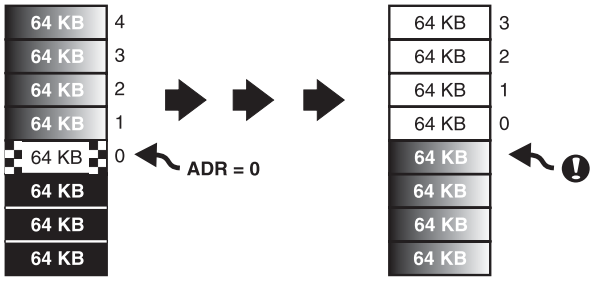


(a)



und das kann dabei auch zum Überschreiben von Datenbereichen kommen (b):

(b)



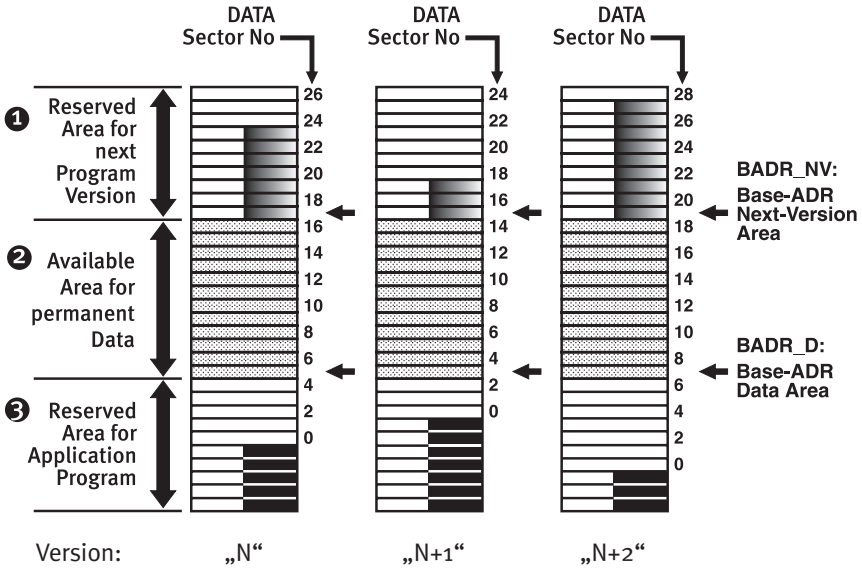
Diesen Sachverhalt kann ein Anwendungs-Programm mit den bereits genannten System-Variablen kontrollieren und Bereiche des FLASH gezielt einsetzen für:

(1) DATA-FLASH Bereich zur Aufnahme der Nachfolge-Version eines Anwendungs-Programmes,

(2) DATA-FLASH Bereich, der zur dauerhaften Daten-Speicherung eingesetzt werden kann,

(3) aktuell ausgeführter Programm-Code.

Beispiel FLASH-Speicheraufteilung in 3 aufeinanderfolgfenden Anwendungs-Programm Versionen:



Hierzu berechnet das Anwendungs-Programm gleich in der Startup-Phase jeweils 2 Basis-Pointer auf die DATA-FLASH Bereiche (1) und (2):

```
BADR_D  =  DATA_FLASH_LEN  -  ( FLASH_GESAMT_LEN  -  PROGRAM_AREA_LEN )
                                                              ❸

BADR_NV =  BADR_D  +  Available_DATA_Area_Len
                      ❷
```

Remote Software Updating

Mit der Update_Me Funktion wird ein Update Vorgang nur dann ausgelöst, wenn eine *vollständige* und *korrekt empfangene* Programm-Version der nächsten Generation vorliegt. Dies hat den Vorteil, daß niemals ein Update-Vorgang gestartet werden kann, so lange das neue Programm nicht vollständig übertragen worden ist.

Bevor eine neue Programm-Version nicht vollständig im Speicher eines Tigers vorliegt, wird ein Update-Vorgang nicht ausgelöst.

Für den erfolgreichen Einsatz der Update_Me Funktion ist ein Grundsatz von entscheidender Bedeutung:

Stets muss die nächste Version des Anwendungs-Programms in der Lage sein

wiederum eine nächste Version des Anwendungs-Programms zu laden und ein Update hiermit auszuführen.

Bevor eine entfernte Anwendung mit einer neuen Programm-Version ausgestattet wird, sollte man sorgfältige Tests mit der neuen Version durchführen - insbesondere muß immer gewährleistet sein, dass die Update-Kette niemals unterbrochen wird.

Weiterhin ist beim Produkt-Design darauf zu achten, dass es während des Update-Vorganges nicht zu einem Powerfail kommen kann - dies könnte ebenfalls die Update-Kette unterbrechen.

Entsprechende Vorsorgemaßnahmen:

- Batterie-Reserve mindestens ausreichend für den Update_Me Prozeß
- Ausschalten nicht zulassen während eines Update-Vorganges.

Für unbediente, weit verstreute Anwendungen kann zur Vereinfachung des Designs - z.B. aus Kostengründen - auf solche Maßnahmen immer dann verzichtet werden, wenn:

(i) die Wahrscheinlichkeit eines Power-Downs sehr niedrig ist
(ii) die Häufigkeit eines Updates gering ist
(iii) die Kosten für ein manuelles Update vertretbar sind.

Hierzu einige überschlägige Betrachtungen:

(i) Die mittlere Häufigkeit eines Stromausfalles werde in einer Anwendung z.B. auf 3 x p.a. eingeschätzt.

Die Dauer eines Update-Prozesses dauere z.B. 20 Sekunden.

Somit verursacht jedes ungeschütze Update ein Risko, dass der Stromausfall genau während des Update-Prozesses auftreten kann von:

RISC = 3 x 20 / (365 x 24 x 3600) = 0,000002

oder anders ausgedrückt:

Die Wahrscheinlichkeit, daß genau dann ein Stromausfall eintritt, während ein Update-Vorgang ausgeführt wird liegt bei

1 : 500.000

Ziemlich unwahrscheinlich.

Wenn in einer Anwendung 1000 Systeme im Spiel sind und voraussichtlich alle 6 Monate ein Programm-Update eingespielt werden soll, so ergibt sich eine Wahrscheinlichkeit dafür, dass 1 System von allen während eines Updates in einem bestimmten Jahr von einem Stromausfall betroffen wird zu:

ca. 1 : 250

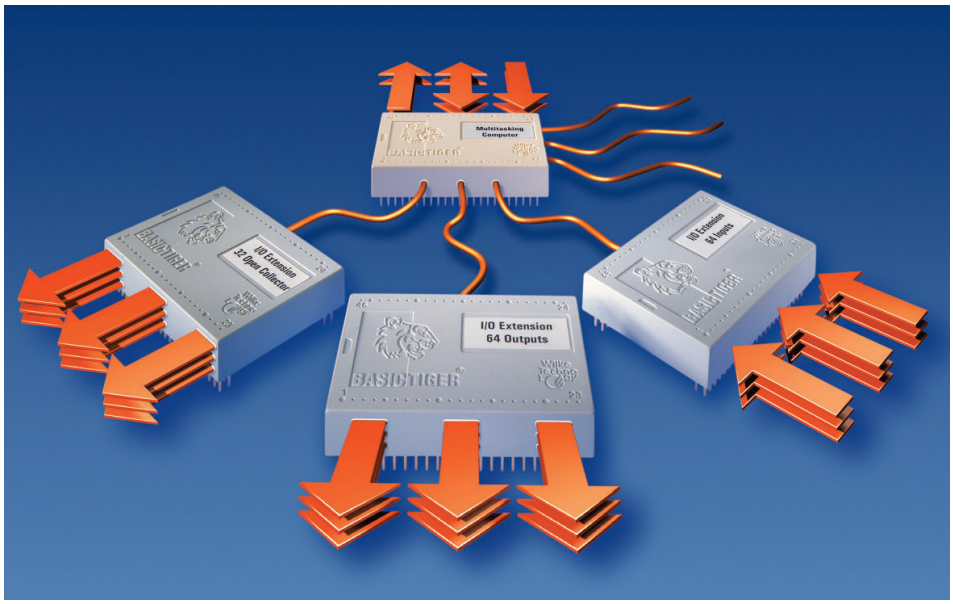
Das würde bedeuten, daß etwa einmal in 250 Jahren, eines der 1000 Systeme von einer solchen Störung betroffen wäre, die ein manuelles Updating erforderlich macht.

Dieses Risiko dürfte weit, weit geringer liegen, als das Risko durch fehlerhafte Programmierung die Update-Kette zu unterbrechen.

Allerdings sollte man klugerweise eine Risko-Bündelung in solch einem Szenario vermeiden. Hier bedeutet das z.B., dass man den Update-Vorgang bei allen Systemen zeitlich gestaffelt ausführt, so dass ein gemeinsamer Stromausfall nur zu maximal 1 Störung führen kann - und nicht das ganze Kollektiv betroffen sein kann.

XPorts

I/O Erweiterungen für BASIC-Tiger



Für die Erweiterung der I/O Struktur im BASIC-Tiger System steht das XPort System zur Verfügung. Nachfolgend werden folgende Tiger-BASIC Funktionen für das Port Erweiterungs-System XPort beschrieben:

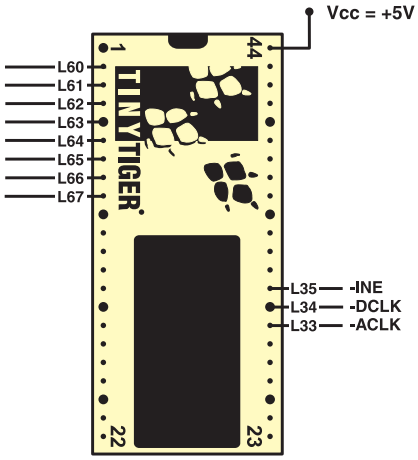
XSetup	=> definiert XPort-Signale
Xin	=> lies XPort ein
Xin$	=> lies XPort(s) ein
XOut	=> scheibe auf XPort(s)
XSet	=> setze Bit in XPort
XRes	=> lösche Bit in XPort
Xinv	=> invertiere Bit in XPort
XPin	=> lies Bit von XPort

Hierzu stehen Beispiel-Schaltungen mit EP-Erweiterungsmodulen und Programmierbeispiele zur Verfügung. Es lassen sich mit dem XPort-System bis zu 512 I/O-Ports in ein Tiger-Projekt hinzufügen mit voller Software-Unterstützung.

XSetup

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

Funktion: Definiert Bus- und Ctrl-Signal-Leitungen für das XPort System.



XPort Signalleitungen - Default Setting

Parameter:

	B	W	L	S	F	
Bus_Port	●	●	●	-	-	Port, der als 8-Bit DATA-/ADR-Bus verwendet wird: Port 6 oder Port 8
CTRL_Port	●	●	●	-	-	Port, der als 3-Bit CTRL-Bus verwendet wird für Signale: ACLK, DCLK, -INE
Bit_ACLK	●	●	●	-	-	Bit-Nr: 0...7 für ACLK: ADR-Clock
Bit_DCLK	●	●	●	-	-	Bit-Nr: 0...7 für DCLK: DATA-Clock

Bit_INE	●	●	●	-	-	Bit-Nr: 0...7 für -INE: Input-Enable (low active)
CTRL2_Port	●	●	●	-	-	Port für CE-Signal das bei XBus_OutR / XBus_InR verwendet wird. Dieses Signal wird *nur* für die Funktionen XBus_OutR und XBus_InR benötigt. Für Anwendungen die diese Funktionen nicht einsetzen sollte dieses Signal auf einen Dummy-Wert gesetzt werden, also einen nicht existierenden I/O-Pin. Z.B: Port-4, Bit-7 (BASIC-Tiger, TINY-Tiger oder Econo-Tiger).
Bit_Bus_CE	●	●	●	-	-	Bit-Nr: 0...7 für Bus_CE: BUS-Access CE-Signal. Dieses Signal wird vom Tiger AKTIV gesetzt (d.h. INVERTIERT gegenüber dem ursprünglichen Pegel) während ein XBUS-Zugriff erfolgt. Damit weiss die andere Einheit, dass gerade eine BUS-Übertragung statt findet. Weitere Signale, wie z.B. die DATA-Direction kann der Anwender ggf. selbst im BASIC-Prog. definieren und bedienen. **Funktionswert:**
Flag	●	●	●	-	-	0 = OK, Parameter akzeptiert 1...5 = Nr. des fehlerhaften Parameters

Beachte:

XSetup ordnet Tiger I/O-Pins den Signalen des XPort Systems zu. Die Pins selbst werden nicht verändert, es wird keine Direction-Zuordnung vorgenommen oder ein Wert gesetzt.

Die übliche Vorgehensweise ist daher:

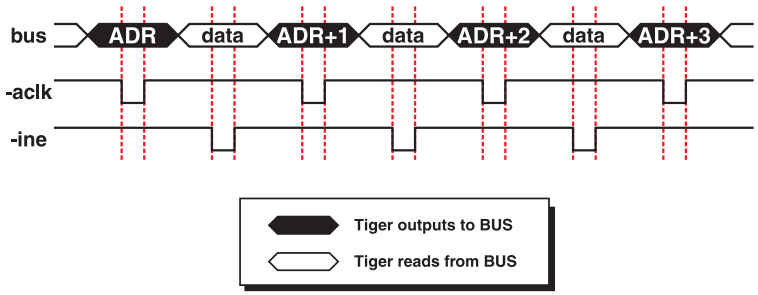
1.)	XSetup	XPort Pins zuweisen
2.)	Dir_Pin	Ctrl-Pins als Ausgang setzen
3.)	Out	Pins auf definierten Pegel setzen.

Beachte:

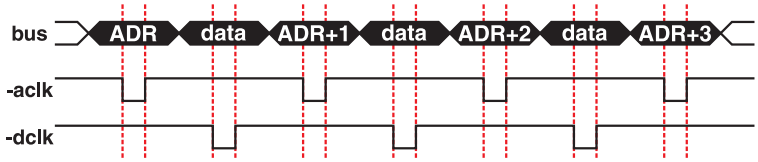
Während eines XBUS-Zugriffs, können auch andere Device-Treiber (LCD1, LCD2, parallel IN / OUT, ... etc.) diesen Bus benutzen. Eine laufende XBUS-

Übertragung wird ggf. durch solche Requests unterbrochen (XBUS_CE wird während dieser Zeit "inactive" gesetzt). Anschließend wird die XBUS-Übertragung wieder fortgesetzt.

Das XPort System erweitert die Tiger I/O Struktur um bis zu 4096 I/O Leitungen. Hierzu wird ein 8-Bit Bus für die Übertragung von Adress- und Daten-Bytes verwendet sowie 3 Ctrl-Leitungen für die Steuerung des Datenflusses (-ine, aclk, dclk). Die I/O-Buszugriffe für Eingaben und Ausgaben auf das XPort-System:



Lese-Zugriff auf XPorts in aufsteigender ADR-Lage



Schreib-Zugriff auf XPorts in aufsteigender ADR-Lage

Das XPort System arbeitet direkt mit den I/O-Erweiterungsmodulen der EP-Reihe zusammen. Zugriffe auf XPorts eines Tiger-Systems erfolgen jeweils in nur 1 BASIC-Zeile. Schaltungsbeispiele und zugehöriger Programmcode ist in den Beschreibungen der Funktionen Xin/Xin$ und XOut enthalten.

Sofern man nicht vom Default-Setting für das ePort System abweicht (Bus = L60...67, aclk=L33, dclk=L34, -ine=L35), ist eine XSetup Funktion nicht erforderlich. XSetup wird eingesetzt wenn:

1.) eine andere Pin-Zuordnung getroffen werden soll

oder

2.) ein Pegel-System für „aclk“ und „dclk“ explizit festgelegt werden soll.
Die beiden oben gezeigten Signal-Diagramme zeigen die Ctrl-Signale „aclk“ und „dclk“ in inverser Logik: „-aclk“ und „-dclk“.

Nachfolgende 2 Code-Beispiele zeigen das explizite Festlegen der jeweiligen Ports und Pins mit Definition ihrer Anfangs-Pegel.

```
#INCLUDE  DEFINE_A.INC            ' allgemeine Definitionen
USER_EPORT ACT, NOACTIVE          ' disable e-Port System

TASK MAIN                         ' Beginn Task MAIN
  DIR_PORT 3,11000111B            ' setze Port 3: L33...35 = Ausgaenge
  OUT 3, 00111000B, 0FFH          ' setze L33 ... L35 auf "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 fuer Ctrl2 Signal "CE"
                   7      )       ' Bit-7 = "CE" => dummy

  A$ = Xin (0, 8)                 ' Lies 8 Bytes von 8 XPorts ab ADR 0
  XOUT (10H, A$)                  ' Output 8 Bytes to XPort 10H...17H
END                               ' Ende Task MAIN
```

Diese Code-Sequenz definiert die Standard Ctrl-Pins und den Bus auf Port 6. Die Ctrl-Signale werden zu: „-dclk“, „-aclk“ und „-ine“ festgelegt.

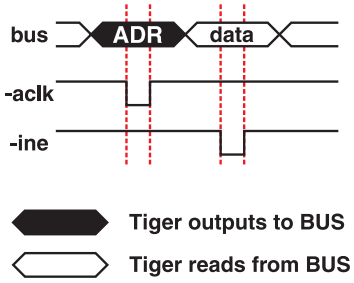
Die gleiche Sequenz mit einer entsprechend veränderten Codezeile „OUT 3...“ definiert die Ctrl-Signale: „dclk“, „aclk“ und „-ine“:

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

Xin 1 Byte

```
N = Xin ( ADR )              ' ‹ADR› ‹data›
```

Funktion: Liest 1 Byte von I/O Erweiterungs-Port (XPort) „ADR“ ein.



Xin (ADR)

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion-Port (XPort) Adr: 0 ... 255
						Funktionswert:
N	●	●	●	-	-	1 Byte Ergebnis in numerischer Variable

Xin arbeitet mit den Default Einstellungen des XPort-Systems bzw. mit den explizit getroffenen Einstellungen durch XSETUP (...).

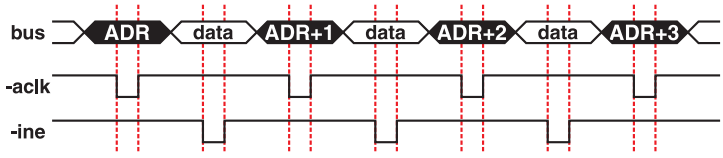
Weitere XPort I/O-Funktionen siehe auch:

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

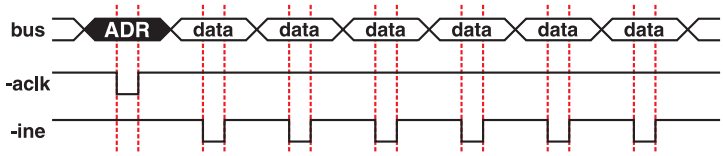
Xin$ n Bytes

```
X$ = Xin$ ( ADR, ANZ )     ' ‹ADR› ‹data› ‹ADR+1› ‹data› ‹ADR+2› ‹data› ...
X$ = Xin$ ( -ADR, ANZ )    ' ‹ADR› ‹data› ‹data› ‹data› ...
X$ = Xin$ ( 256+, ANZ )    ' ‹data› ‹data› ‹data› ...
```

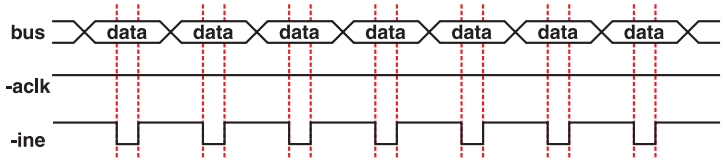
Funktion: Liest Erweiterungs-Port ein mit verschiedenen Bus-Zugriffsarten.



Xin$ (ADR, ANZ)



Xin$ (-ADR, ANZ)



Xin$ (256+, ANZ)

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion Adr: 0 ... 255, bzw. Flag (256+)
ANZ	●	●	●	-	-	Anzahl Bytes zum Einlesen
						Funktionswert:
X$	-	-	-	●	-	Ergebnis-String mit eingelesenen Daten-Bytes

Erweiterte I/O Ports - XPorts

Xin$ arbeitet mit den Default Einstellungen des XPort-Systems bzw. mit den explizit getroffenen Einstellungen durch XSETUP (...).

Xin$ wird eingesetzt zum Einlesen von Daten aus I/O-Erweiterungsmodulen und anderen peripheren Devices (Speicher, Logik, ...). In seiner urspünglichen Form werden Daten von aufsteigenden Port-Adressen eingelesen. Dies erlaubt die zeitnahe Abtastung einer Vielzahl von Eingangsleitungen mit einer einzigen BASIC-Instruktion.

Für den schnellen parallelen Datentransfer von peripheren Einheiten zum Tiger sind mit Version 5.2 auch die Formen

```
X$ = Xin$ (-ADR, ANZ)          ' ‹ADR› ‹data› ‹data› ‹data› ...
X$ = Xin$ (256+, ANZ)          ' ‹data› ‹data› ‹data› ...
```

hinzugekommen. Soweit erforderlich werden vom BASIC-Programm aus noch weitere Ctrl-Leitungen kontrolliert wie es die Kommunikation mit der peripheren Einheit erfordert (z.B R/W oder -CE Signale).

Weitere XPort I/O-Funktionen siehe auch:

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

Es folgen:

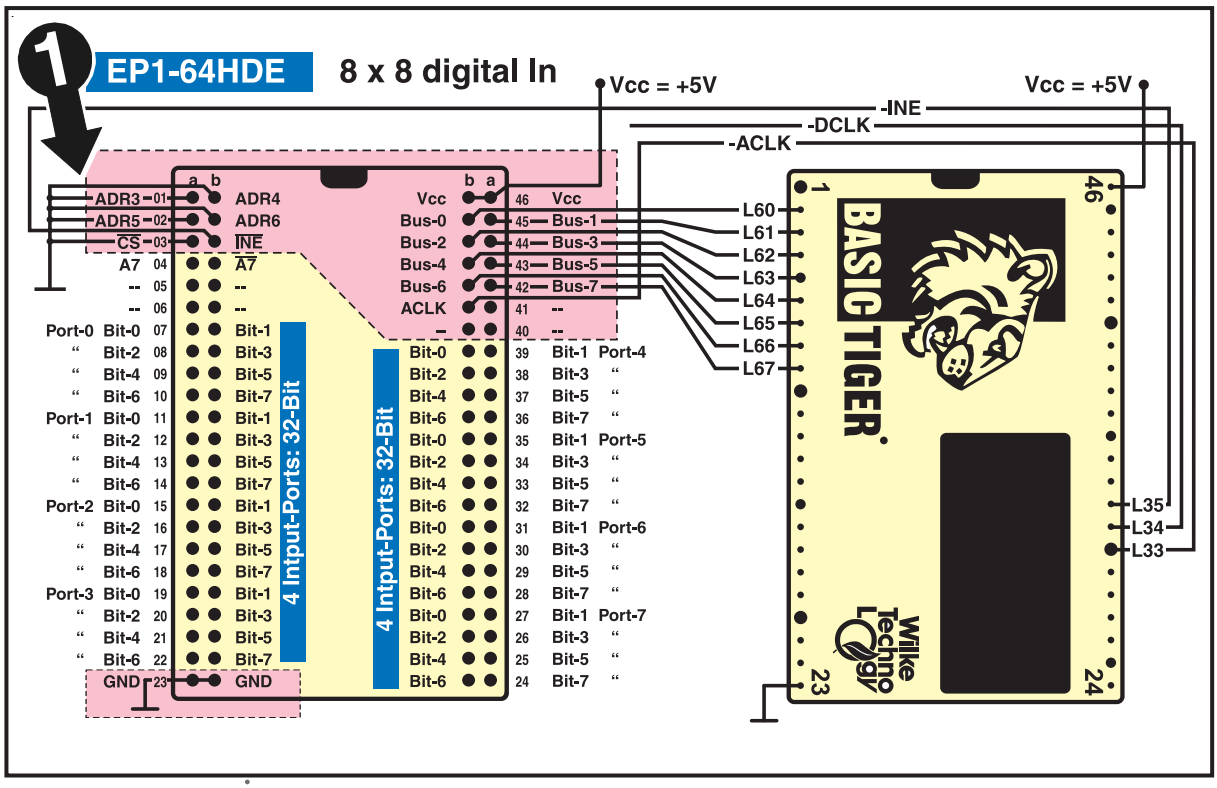
Kurze Beispiele für den Anschluß von I/O-Erweiterungsmodulen und ihre Ansteuerung mit einem Tiger-BASIC Programm.

Beispiel-1

Eine Anwendung brauche eine größere Anzahl von digitalen Eingängen. Man setzt daher zusätzlich zum Basic-Tiger ein I/O Erweiterungsmodul EP1-64HDE ein, das 64 digitale Eingänge liefert.

Da dies das einzige Erweiterungsmodul im System ist, wählt man die Basis-Adresse für die Erweiterungs-Ports = 0. Dies geschieht durch Pin-Programmierung der Pins:

1a = ADR-3
1b = ADR-4
2a = ADR-5
2b = ADR-6 Siehe: 1



Anschlußbild EP1 Erweiterungsmodul und BASIC-Tiger
- Standard Signalbelegung -

ADR-7 wird nicht decodiert, daher spiegeln sich die Adressen unterhalb 80h 1:1 in der oberen Hälfte des Adressraumes ab 80h bis 0FFh.

Die unteren 3 Adress-Bits werden zur Decodierung der 8 Ports im Erweiterungsmodul eingesetzt:

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

Wie bereits erwähnt, spiegeln sich diese 8 Ports im ADR-Bereich: 80h ... 87h wieder. D.h. ohne weitere Decodierungmaßnahmen können bis zu 128 Ports = 1024 I/O Leitungen dem System hinzugefügt werden.

Man beachte ferner, dass die mit „dclk“ (data-clock) bezeichnete Steuerleitung hier NICHT verwendet wird, da mit dem EP1 Modul nur Eingaben, jedoch keine Ausgaben gemacht werden können. Ein Port-Schreibzyklus hat mithin keine Wirkung auf das Erweiterungsmodul.

Da hier von der Standard-Signalbelegung für das XPort-System Gebrauch gemacht worden ist, ist es nicht erforderlich eine XSetup (...) Funktion auszuführen. Um alle 8 XPorts des Erweiterungsmoduls auf die gewünschten Signal-Werte zu setzen bedarf es lediglich eines einzigen Funktionsaufrufes, so daß ein kompilierbares, lauffähiges Programm mit nur 3 Zeilen entsteht:

```
Task MAIN                ' Anfang Main Task
  A$ = Xin$ (0, 8)       ' Lies 8 Bytes von 8 XPorts ab ADR 0
End                      ' Programm-Ende
```

Die Standard Signal-Belegung des XPort-Systems ist per Default:

L60 ... L67	8-Bit Parallel-Bus	Daten und Adressen Übertragung
L33	-ACLK	Address-Clock
L34	-DCLK	Data-Clock
L35	-INE	Input-Enable

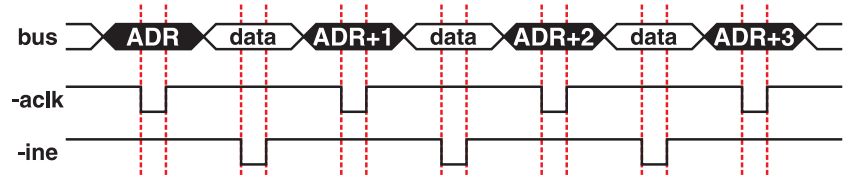
Je nach den I/O Gegebenheiten des jeweiligen BASIC-Tiger Moduls können diese Zuweisungen geändert werden. Ebenso kann man die Signale ACLK und DCLK auf Wunsch

auch durch -ACLK und -DCLK ersetzen. Beide Signaltypen (true und invers) arbeiten mit den I/O Erweiterungsmodulen, für andere Erweiterungs-Beschaltungen kann der eine oder andere Signaltyp vorteilhafter sein.

Nachfolgendes Code-Beispiel zeigt das explizite Festlegen der jeweiligen Ports und Pins mit Definition ihrer Anfangs-Pegel.

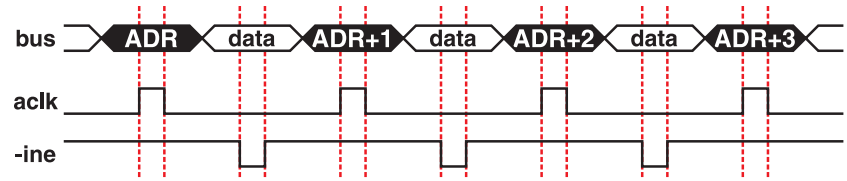
```
#INCLUDE  DEFINE_A.INC              ' allgemeine Definitionen
USER_EPORT ACT, NOACTIVE            ' disable e-Port System
TASK MAIN                           ' Beginn Task MAIN
  DIR_PORT 3,11000111B              ' setze Port 3: L33...35 = Ausgaenge
  OUT 3, 00111000B, 0FFH            ' setze L33 ... L35 auf „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 fuer Ctrl2 Signal „CE"
                  7       )         ' Bit-7 = „CE" => dummy

  A$ = Xin$ (0, 8)                  ' Lies 8 Bytes von 8 XPorts ab ADR 0
END                                 ' Ende Task MAIN
```

In diesem Beispiel wurden die gleichen I/O-Leitungen für den Bus und die Steuer-Signale verwendet wie sie per Default festliegen. Jedoch haben wir durch die 2. Zeile in der Task „OUT 3, ..." die 3 Steuerleitungen ACLK, DCLK und -INE auf „high" Pegel gesetzt, so dass wir hier folgendes Signalschema erhalten:



Steuerleitungen -aclk, -ine

Entsprechend können die Steuerleitungen ACLK und (DCLK) auch auf „low" Pegel vorbesetzt werden was folgendes Signalbild erzeugt:

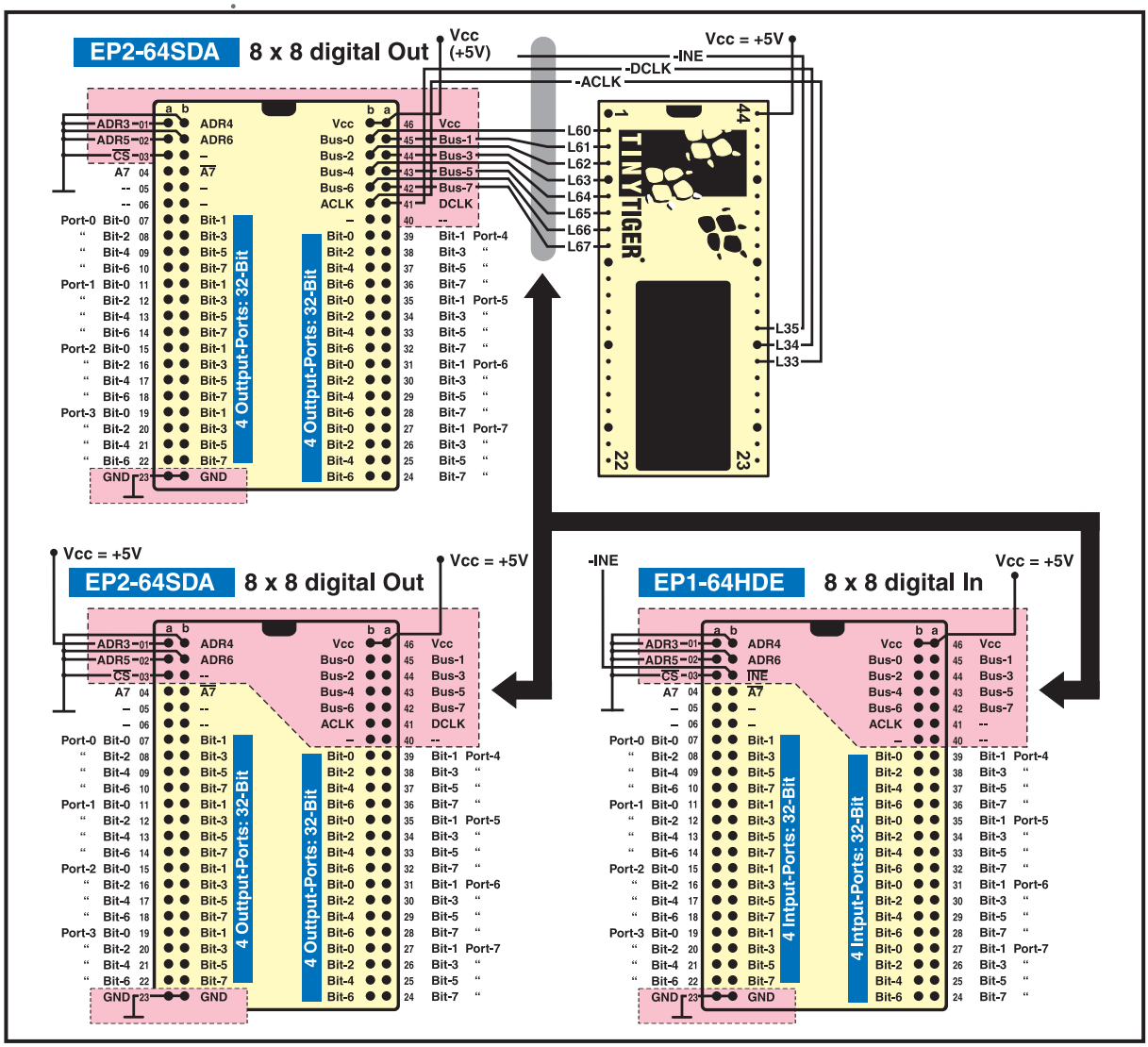


Steuerleitungen aclk, -ine

Die zugehörige Codezeile zu „aclk" und „dclk" lautet im obigen Beispiel dann:

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

Als 2. Beispiel der Anschluß mehrerer I/O-Erweiterungsmodule für große I/O-Strukturen - hier 2 St. EP2 und 1 St. EP1 Module mit je 64 digitalen Aus-/Eingängen.



Anschlußbild 2 St. EP2 + 1 St. EP2 Erweiterungsmodule, TINY-Tiger
- Standard Signalbelegung -

Erweiterungsmöglichkeiten:

(a) Nach diesem Schema können bis zu 16 solcher Input I/O-Erweiterungsmodule angeschlossen werden, wobei jedes Modul eine individuelle Basis-Adresse erhält: 0, 8H, 10H, 18H ... 78H (siehe: (1)). Es bedarf hierzu keinerlei zusätzlicher Decodierungsmaßnahmen. Im BASIC-Programm spricht man diese Ausgänge an wie bereits gezeigt, jedoch mit entsprechend größerem Adress-Umfang:

```
A$ = Xin$ (0, 8)          ' Lies 8 Bytes von 8 XPorts ab ADR 0
A$ = Xin$ (8, 8)          ' Lies 8 Bytes von 8 XPorts ab ADR 8
A$ = Xin$ (10H, 8)        ' Lies 8 Bytes von 8 XPorts ab ADR 10H
A$ = Xin$ (18H, 8)        ' Lies 8 Bytes von 8 XPorts ab ADR 18H
```

Ebenso kann man natürlich auch einzelne Ports ansprechen:

```
A$ = Xin$ (2AH, 1)        ' Lies 1 Byte von XPort 2AH
```

wie auch alle 128 Ports = 1024 Ausgangsleitungen in einer einzigen Zeile:

```
A$ = Xin$ (0, 128)        ' Lies 128 Bytes von 128 XPorts ab ADR 0
```

(b) Bis zu 32 Eingangs-Module a 8 XPorts (= 256 XPorts = 2048 Ausgänge) lassen sich in diese Struktur einfügen mit zusätzlicher Decodierung von ADR-7 - Schaltungsvorschläge hierzu finden sich in der TEC-Eurocard Familie im Web (www.wilke.de). Die Adressierung der XPort Eingänge reicht dann entsprechend:

von 00 ... bis ... FFH

(c) Zusätzlich zu XPort Eingängen lassen sich auch XPort Ausgänge hinzufügen. Für XPort-Ausgänge gilt grundsätzlich das gleiche wie für Eingänge. Ohne zusätzliche Decodierung von ADR-7 können 16 Module = 128 Eingangs-Ports = 1024 Eingänge mit den Erweiterungs-Modulen des Typs EP1-64HDE realisiert werden. Diese Eingangs-Ports können den gleichen Adressraum belegen wie auch eventuell vorhandene Ausgangs-Ports. D.h. unter einer bestimmten Basis-Adresse kann sowohl ein Eingangs-Modul EP1 wie auch ein Ausgangs-Modul EP2 vorhanden sein. Der Zugriff auf eine gemeinsame XPort Adresse erfolgt entsprechend:

```
XOUT (THIS_ADR, ....)        ' schreibt ab XPort-Adresse „THIS_ADR“
A$ = XIN$ (THIS_ADR, ...)    ' liest ab XPort-Adresse „THIS_ADR“
```

Somit verdoppelt sich nochmals die Zahl der möglichen erweiterten I/O Kanäle zu:

(i) Ohne Adress-Decodierung ADR7:

16 Output-Module	=	128 Output-Ports	=	1024 Ausgänge
16 Input-Module	=	128 Input-Ports	=	1024 Eingänge
		Gesamt	=	2048 I/Os

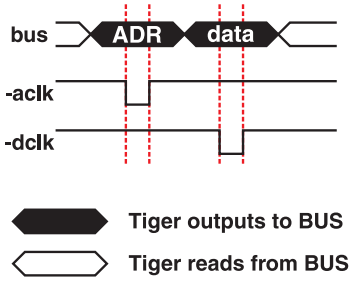
(ii) Mit Adress-Decodierung ADR7:

32 Output-Module	=	256 Output-Ports	=	2048 Ausgänge
32 Input-Module	=	256 Input-Ports	=	2048 Eingänge
		Gesamt	=	4096 I/Os

XOut 1 Byte

```
XOut ( ADR, N )          ' ‹ADR› ‹data›  -  write 1 Byte
```

Funktion: Schreibt 1 Byte auf den I/O Erweiterungs-Port (XPort) „ADR“.



XOut (ADR, data)

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion-Port (XPort) Adr: 0 ... 255
N	●	●	●	-	-	Low-Byte wird auf XPort geschrieben

Kein Funktionswert

XOut arbeitet mit den Default Einstellungen des XPort-Systems bzw. mit den explizit getroffenen Einstellungen durch XSETUP (...).

Weitere XPort I/O-Funktionen siehe auch:

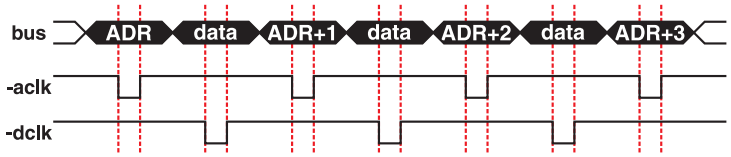
XSETUP
XBUS_OUTR, XBUS_INR
XIN, XIN$
XOUT
XSET, XINV, XRES
XPIN

XOut n Bytes

```
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
```

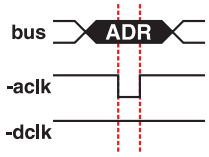
Funktion: Schreibt in verschiedenen Modes auf I/O Erweiterungs-Ports (XPort).

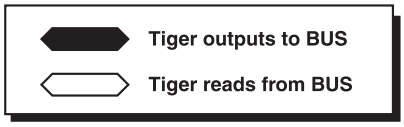
Schreibe auf aufeinander folgende XPorts



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

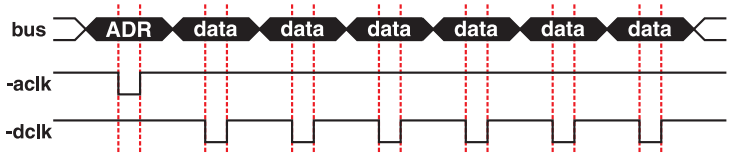
Schreibe nur 1-mal ADR auf den Bus





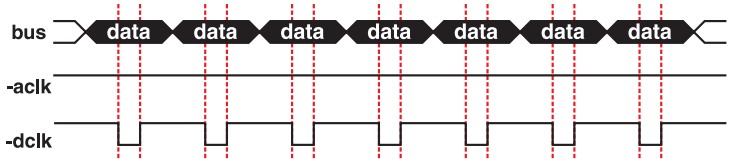
XOut (ADR, "")

Schreibe 1-mal ADR auf Bus und dann viele Data-Bytes



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

Schreibe nur viele Data-Bytes auf Bus



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

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion-Port (XPort) Adr: 0 ... 255
A$	-	-	-	●	-	Data-String zum Senden an XPort(s)
Flash	●	●	●	-	-	Flash-ADR ab der Data-Bytes zum Senden liegen
FLen	●	●	●	-	-	Anzahl Bytes im Flash zum Senden an XPort(s)

Kein Funktionswert

XOut arbeitet mit den Default Einstellungen des XPort-Systems bzw. mit den explizit getroffenen Einstellungen durch XSETUP (...).

Weitere XPort I/O-Funktionen siehe auch:

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

Es folgen:

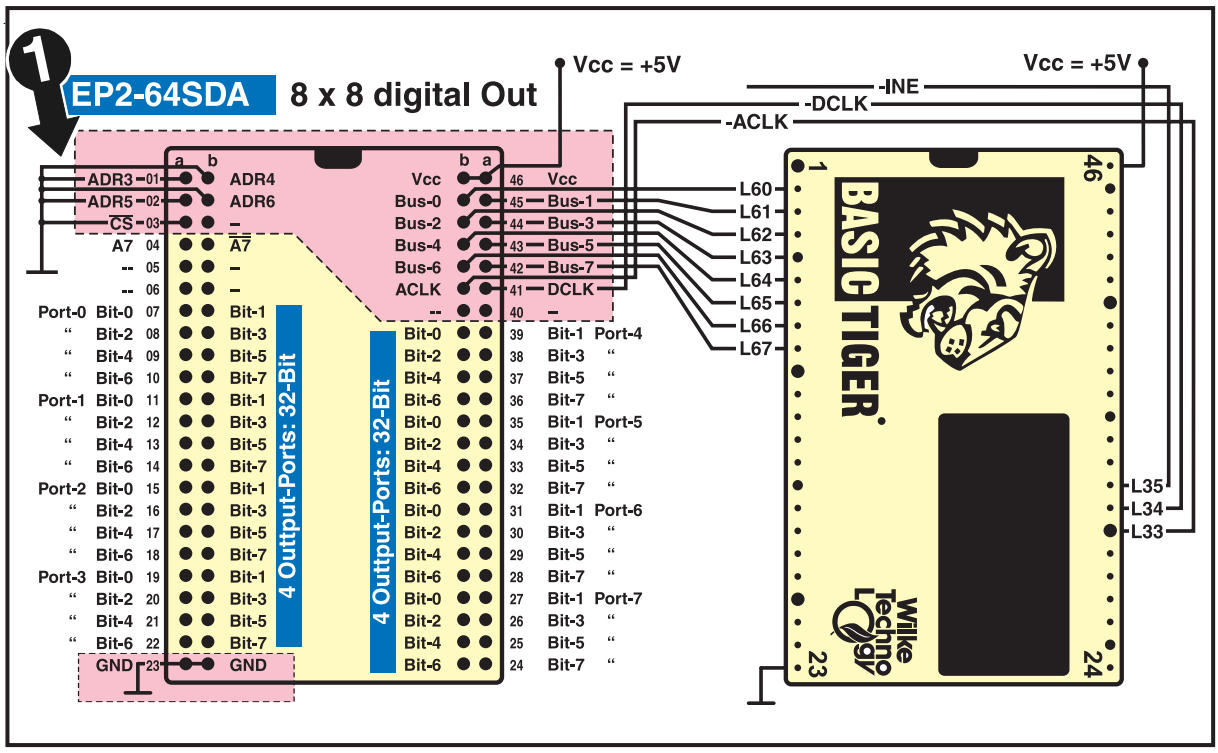
Kurze Beispiele für den Anschluß von I/O-Erweiterungsmodulen und und ihre Ansteuerung mit einem Anwendungs-Programm.

Beispiel-1

Eine Anwendung brauche eine größere Anzahl von digitalen Ausgängen. Man setzt daher zusätzlich zum Basic-Tiger ein I/O Erweiterungsmodul EP2-64SDA ein, das 64 digitale Ausgänge liefert.

Da dies das einzige Erweiterungsmodul im System ist, wählt man die Basis-Adresse für die Erweiterungs-Ports = 0. Dies geschieht durch Pin-Programmierung der Pins:

1a = ADR-3
1b = ADR-4
2a = ADR-5
2b = ADR-6 Siehe: 1



Anschlußbild EP2 Erweiterungsmodul und BASIC-Tiger
- Standard Signalbelegung -

ADR-7 wird nicht decodiert, daher spiegeln sich die Adressen unterhalb 80h 1:1 in der oberen Hälfte des Adressraumes ab 80h bis 0FFh.

Die unteren 3 Adress-Bits werden zur Dekodierung der 8 Ports im Erweiterungsmodul eingesetzt:

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

Wie bereits erwähnt, spiegeln sich diese 8 Ports im ADR-Bereich: 80H ... 87H wieder. D.h. ohne weitere Decodierungmaßnahmen können bis zu 128 Ports = 1024 I/O Leitungen dem System hinzugefügt werden.

Man beachte ferner, dass die mit „-INE“ (input-enable) bezeichnete Steuerleitung hier NICHT verwendet wird - da mit dem EP2 Modul nur Ausgaben, jedoch keine Eingaben gemacht werden können. Ein Port-Lesezyklus hat mithin keine Wirkung auf das Erweiterungsmodul.

Da hier von der Standard-Signalbelegung für das XPort-System Gebrauch gemacht worden ist, ist es nicht erforderlich eine XSetup (...) Funktion auszuführen. Um alle 8 XPorts des Erweiterungsmoduls auf die gewünschten Signal-Werte zu setzen bedarf es lediglich eines einzigen Funktionsaufrufes, so daß ein kompilierbares, lauffähiges Programm mit nur 3 Zeilen entsteht:

```
Task MAIN                                ' Anfang Main Task
  Xout (0, "00 01 03 07 0F 1F 3F 7F"%) ' schreibe 8 Bytes auf 8 XPorts
End                                      ' Programm-Ende
```

Die Standard Signal-Belegung des XPort-Systems ist per Default:

L60 ... L67	8-Bit Parallel-Bus	Daten und Adressen Übertragung
L33	-ACLK	Address-Clock
L34	-DCLK	Data-Clock
L35	-INE	Input-Enable

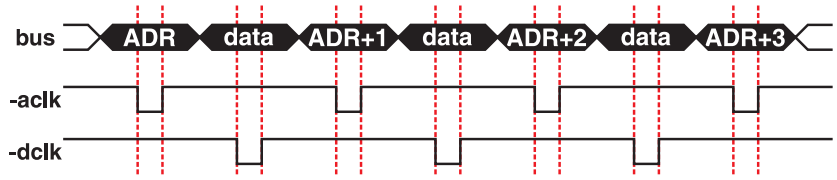
Je nach den I/O Gegebenheiten des jeweiligen BASIC-Tiger Moduls können diese Zuweisungen geändert werden. Ebenso kann man die Signale ACLK und DCLK auf Wunsch

auch durch -ACLK und -DCLK ersetzen. Beide Signaltypen (true und invers) arbeiten mit den I/O Erweiterungsmodulen, für andere Erweiterungs-Beschaltungen kann der eine oder andere Signaltyp vorteilhafter sein.

Nachfolgende 2 Code-Beispiele zeigen das explizite Festlegen der jeweiligen Ports und Pins mit Definition ihrer Anfangs-Pegel.

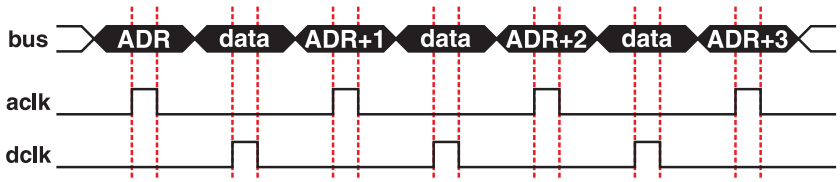
```
#INCLUDE  DEFINE_A.INC              ' allgemeine Definitionen
USER_EPORT ACT, NOACTIVE            ' disable e-Port System
TASK MAIN                           ' Beginn Task MAIN
  DIR_PORT 3,11000111B              ' setze Port 3: L33...35 = Ausgaenge
  OUT 3, 00111000B, 0FFH            ' setze L33 ... L35 auf „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 fuer 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                                 ' Ende Task MAIN
```

In diesem Beispiel wurden die gleichen I/O-Leitungen für den Bus und die Steuer-Signale verwendet wie sie per Default festliegen. Jedoch haben wir durch die 2. Zeile in der Task „OUT 3, ...“ die 3 Steuerleitungen ACLK, DCLK und -INE auf „high“ Pegel gesetzt, so dass wir hier folgendes Signalschema erhalten:



Steuerleitungen -aclk, -dclk

Entsprechend können die Steuerleitungen ACLK und DCLK auch auf „low“ Pegel vorbesetzt werden was folgendes Signalbild erzeugt:

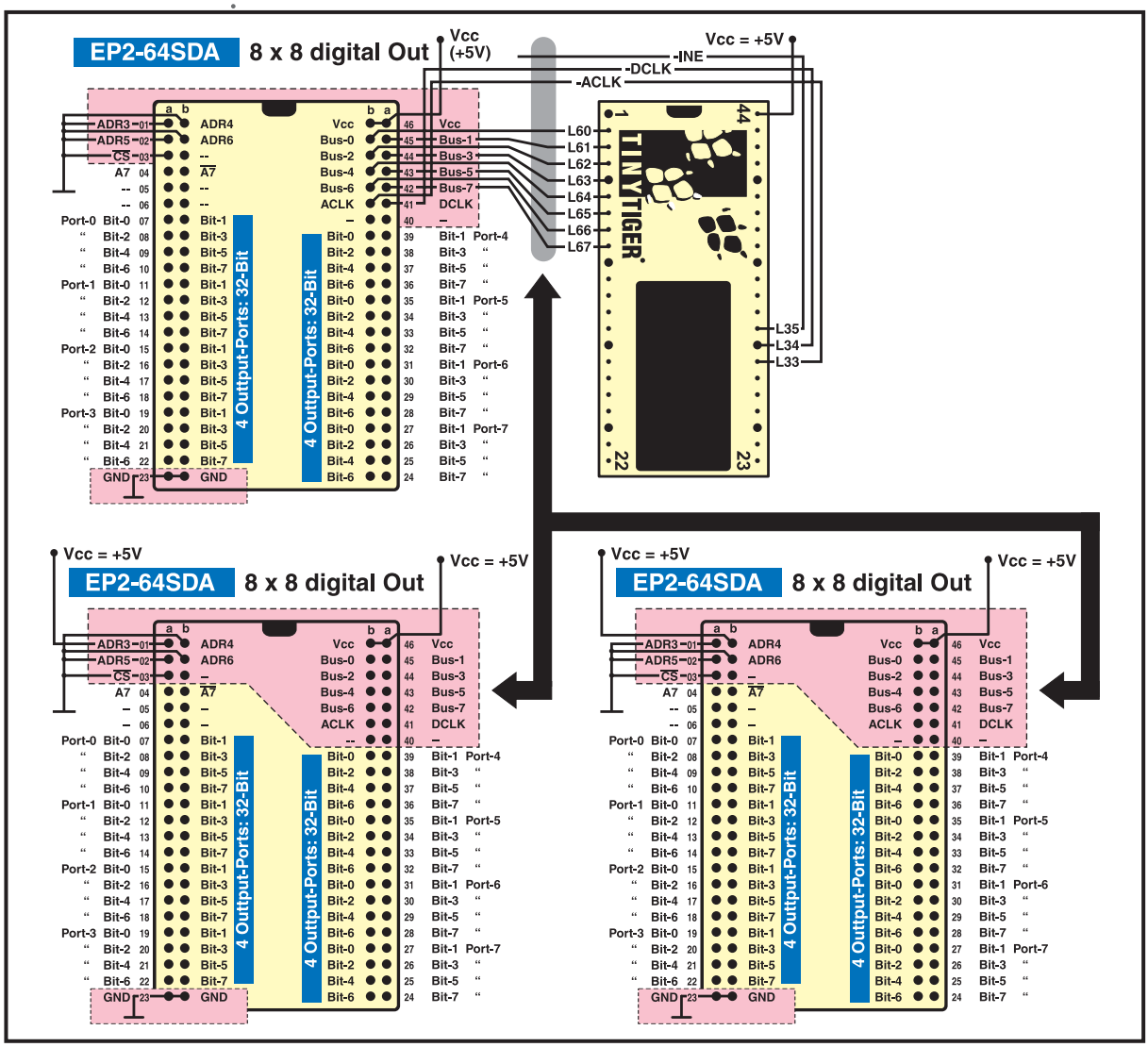


Steuerleitungen aclk, dclk

Die zugehörige Codezeile zu „aclk“ und „dclk“ lautet im obigen Beispiel dann:

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

Als 2. Beispiel der Anschluß mehrerer I/O-Erweiterungsmodule für große I/O-Strukturen - hier 3 St. EP2 Module mit je 64 digitalen Ausgängen.



Anschlußbild 3 St. EP2 Erweiterungsmodule und TINY-Tiger
- Standard Signalbelegung -

Erweiterungsmöglichkeiten:

(a) Nach diesem Schema können bis zu 16 solcher Output I/O-Erweiterungsmodule angeschlossen werden, wobei jedes Modul eine individuelle Basis-Adresse erhält: 0, 8H, 10H, 18H ... 78H (siehe: (1)). Es bedarf hierzu keinerlei zusätzlicher Decodierungsmaßnahmen. Im BASIC-Programm spricht man diese Ausgänge an wie bereits gezeigt, jedoch mit entsprechend größerem Adress-Umfang:

```
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
```

Ebenso kann man natürlich auch einzelne Ports ansprechen:

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

wie auch alle 128 Ports = 1024 Ausgangsleitungen in einer einzigen Zeile:

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

(b) Bis zu 32 Ausgangs-Module a 8 XPorts (= 256 XPorts = 2048 Ausgänge) lassen sich in diese Struktur einfügen mit zusätzlicher Decodierung von ADR-7 - Schaltungsvorschläge hierzu finden sich in der TEC-Eurocard Familie im Web (www.wilke.de). Die Adressierung der XPort Ausgänge reicht dann entsprechend:

von 00 ... bis ... FFH

(c) Zusätzlich zu XPort Ausgängen lassen sich auch XPort Eingänge hinzufügen. Für XPort-Eingänge gilt grundsätzlich das gleiche wie für Ausgänge. Ohne zusätzliche Decodierung von ADR-7 können 16 Module = 128 Eingangs-Ports = 1024 Eingänge mit den Erweiterungs-Modulen des Typs EP1-64HDE realisiert werden. Diese Eingangs-Ports können den gleichen Adressraum belegen wie auch eventuell vorhandene Ausgangs-Ports. D.h. unter eine bestimmten Basis-Adresse kann sowohl ein Eingangs-Modul EP1 wie auch ein Ausgangs-Modul EP2 vorhanden sein. Der Zugriff auf eine gemeinsame XPort Adresse erfolgt entsprechend:

```
XOUT (THIS_ADR, ....)        ' schreibt ab XPort-Adresse „THIS_ADR“
A$ = XIN$ (THIS_ADR, ...)    ' liest ab XPort-Adresse „THIS_ADR“
```

Somit verdoppelt sich nochmals die Zahl der möglichen erweiterten I/O Kanäle zu:

(i) Ohne Adress-Decodierung ADR7:

16 Output-Module	=	128 Output-Ports	=	1024 Ausgänge
16 Input-Module	=	128 Input-Ports	=	1024 Eingänge
		Gesamt	=	2048 I/Os

(ii) Mit Adress-Decodierung ADR7:

32 Output-Module	=	256 Output-Ports	=	2048 Ausgänge
32 Input-Module	=	256 Input-Ports	=	2048 Eingänge
		Gesamt	=	4096 I/Os

XSet, XRes, Xinv

```
XSet ( ADR, Bit_Pos )      ' setze 1 Bit in XPort ADR: 00 ... FF
XRes ( ADR, Bit_Pos )      ' RESET 1 Bit in XPort ADR: 00 ... FF
Xinv ( ADR, Bit_Pos )      ' Toggle 1 Bit in XPort ADR: 00 ... FF
```

Funktion: Manipuliere ein Bit eines XPort-Ausgangs:

- Set Bit
- Reset Bit
- Invert Bit

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion-Port (XPort) Adr: 0 ... 255
Bit_Pos	●	●	●	-	-	Bit-Position: 0 ... 7

Kein Funktionswert

Programm-Beispiel:

```
TASK MAIN                  ' Beginn 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
```

Das Beispiel verwendet die Default Settings des e-Port Systems: Bus = L6, ackl = L33, dclk = L34, -ine = L35. Andere Einstellungen -> siehe Funktion „XSetup“.

Weitere XPort I/O-Funktionen siehe auch:

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

XPin

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

Funktion: Liest einen einzelnen Pin eines XPort-Eingangs ein.

Parameter:

	B	W	L	S	F	
ADR	●	●	●	-	-	I/O Expansion-Port (XPort) Adr: 0 ... 255
Bit_Pos	●	●	●	-	-	Bit-Position: 0 ... 7
						Funktionswert:
A	●	●	●	-	-	Eingelesenes Bit von XPort, Wert: 0, 1

Programm-Beispiel:

```
TASK MAIN                          ' Beginn 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
```

Das Beispiel verwendet die Default Settings des e-Port Systems: Bus = L6, ackl = L33, dclk = L34, -ine = L35. Andere Einstellungen -> siehe Funktion „XSetup".

Weitere XPort I/O-Funktionen siehe auch:

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

Device-Treiber

leere Seite

PS2 (Emulation eines PS2-Eingabegerätes)

Dateiname: PS2_Pp.TDD

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

Emulation eines PS2-Eingabe gerätes

Funktion: Dieser Device-Treiber ermöglicht die Emulation eines PS2-Eingabegerätes mit dem BASIC-Tiger, wie z.B. eine PC-Tastatur.

Parameter:

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Pp	-	-	-	-	-	im Dateinamen steht für: P: interner Port der Taktleitung p: Taktpin

Der Treiber verwendet zwei Ausgangsleitungen: Takt (Clock) und Data. Folgende Pin-Kombinationen sind durch die Wahl der passenden Treiberdatei möglich:

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

PS2 (Emulation eines PS2-Eingabegerätes)

Für die Bereitstellung des Taktes benötigt der Treiber den Zeitbasis-Treiber "TimerA.tdd", dieser sollte z.B. für die Emulation einer PC-Tastatur auf eine Frequenz von ca. 10 bis 15 kHz eingestellt werden.

Der PS2-Treiber arbeitet ähnlich einfach wie eine serielle Schnittstelle für die Übertragung von Daten zum Ziel/PC und zurück. Die Anwendung ist ähnlich der des "Ser1b.tdd" Treibers mit diesen Besonderheiten:

- 2 x 256 Bytes Puffer für das Senden sowie den Empfang von Daten.
- Sekundär-Adresse = 0 (1 Kanal, bidirektional).
- User-Function-Codes zum Prüfen und Löschen von Puffern.

User-Function-Codes des PS2_Pp.TDD

User-Function-Codes für Abfragen (Instruktion GET):

Nr	Symbol	Beschreibung
1	UFCI_IBU_FILL	Füllstand des Eingangspuffers (Byte)
2	UFCI_IBU_FREE	freier Platz im Eingangspuffer (Byte)
3	UFCI_IBU_VOL	Größe des Eingangspuffers (Byte)
33	UFCI_OBU_FILL	Füllstand des Ausgangspuffers (Byte)
34	UFCI_OBU_FREE	freier Platz im Ausgangspuffer (Byte)
35	UFCI_OBU_VOL	Größe des Ausgangspuffers (Byte)
65	UFCI_LAST_ERRC	letzter Error-Code
99	UFCI_DEV_VERS	Version des Treibers

User-Function-Codes für Abfragen (Instruktion PUT):

Nr	Symbol	Beschreibung
1	UFCO_IBU_ERASE	Eingangspuffer löschen
33	UFCO_OBU_ERASE	Ausgangspuffer löschen

Zeichen ausgeben

Die Ausgabe erfolgt vorzugsweise mit PUT, das die Zeichen so ausgibt, wie sie sind, daß heißt ohne Umwandlung in ASCII (bei Zahlen) und ohne Anfügen von CR und LF.

Wenn nicht genügend Platz im Ausgabepuffer ist und trotzdem ausgegeben wird, dann wartet die Instruktion PUT oder PRINT (und damit die ganze Task) solange, bis

wieder Platz im Puffer ist. Dieses Warten kann verhindert werden, indem vor der Ausgabe der freie Platz im Puffer abgefragt wird.

Beispiel für die Ausgabe von 4Bytes als Hex-Codes nur dann, wenn noch genügend freier Platz im Ausgangspuffer ist:

```
GET #PS2, #0, #UFCI_OBU_FREE, 0, OUTFREE     ' erfrage Platz in Puffer
IF OUTFREE > 3 THEN                          ' wenn mindestens 4 Bytes frei
  PUT #PS2, "12 1C 9C AA"%                   ' Ausgabe der Zeichen
ENDIF
```

Zeichen einlesen

Empfangene Zeichen werden von Device-Treiber im Eingangspuffer abgelegt. Die Zeichen werden vorzugsweise mit GET eingelesen.

GET wartet nicht und liest nur etwas, wenn auch etwas im Puffer ist. Wird in eine numerische Variable gelesen, dann enthält diese den Wert 0 (Null), wenn

- nichts im Puffer war
- wenn eine 0 (Null) im Puffer war

Um die beiden Fälle zu unterscheiden, sollte man vor einem Leseversuch abfragen, ob Zeichen im Eingangspuffer liegen.

Beispiel für Überprüfung, ob mindestens ein Zeichen im Eingangspuffer ist:

```
GET #PS2, #0, #UFCI_IBU_FILL, 0, INFIL       ' lese Pufferfüllstand
IF INFIL > 0 THEN                            ' wenn etwas im Puffer ist
  GET #PS2, 1, B                             ' dann lies ein Byte
ENDIF
```

Es gibt eine Reihe von PS/2 Eingabegeräten, wie z.B. Tastaturen, Mäuse, Barcode-Scanner etc. Diese Geräte kommunizieren bidirektional mit dem Host, d.h. sie empfangen Kommandos und beantworten bzw. quittieren diese und senden Daten an den Host. Diese Befehlssätze und Daten können sich von Gerät zu Gerät und von Host zu Host ggf. unterscheiden. Wollen sie also eine Tastatur simulieren, informieren sie sich welche Codes generiert werden müssen und welche Kommandos vom Host zu erwarten sind.

Ein ausführlich kommentiertes Beispielprogramm für die Emulation einer PC-Tastatur mit dem BASIC-Tiger ist „PS2_Keyboard_Emulation_Demo_001.TIG“. Das Programm befindet sich im Verzeichnis **DeviceDriver_Examples** ihrer Tiger-BASIC Installation.

PS2 (Emulation eines PS2-Eingabegerätes)

leere Seite

PWMX (Pulsweitenmodulation)

Dateiname: PWMX_Pp.TDD

INSTALL_DEVICE #D, “PWMX_Pp.TDD“, Bereich, Teilerfaktor

Pulsweiten-modulation mit einstellbarer Trägerfrequenz

Funktion: Dieser Device-Treiber ermöglicht die Ausgabe von pulsweitenmodulierten Signalen mit einem beliebigen I/O-Pin mit konstanter, einstellbarer Trägerfrequenz.

Parameter:

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Pp	-	-	-	-	-	im Dateinamen steht für: P: interner Port der PWM-Signalleitung p: Pinnummer der PWM-Signalleitung
Bereich	●	●	●	-	-	Festlegung des Bereichstaktes der Basisfrequenz
Teilerfaktor	●	●	●	-	-	Faktor, mit dem der Bereichstakt heruntergeteilt wird

Die einzustellende Basisfrequenz errechnet sich aus:
gewünschte Trägerfrequenz * 256

Beispiel: für eine Trägerfrequenz des PWM-Signals von 250 Hz wird eine Basisfrequenz von 250 * 256 = 64.000 Hz benötigt. Also muß im Bereichstakt 1 (Bereichsfrequenz 2,5 MHz) der Teilerfaktor auf 39,0625 (gerundet auf 39) gesetzt werden.

Hinweise:

- Die kleinstmögliche einstellbare Basisfrequenz ist 610 Hz, das entspricht einer Trägerfrequenz für das PWM-Signal von 2,383 Hz.
- Alle Einstellungen zwischen 1 und 610 Hz werden als 610 Hz interpretiert.
- Alle Einstellungen von 0 oder negativen Werten werden als 0 Hz interpretiert (keine Ausgabe)
- **Wichtig:** Sowohl TIMERA.TDD als auch PWMX.TDD verwenden sehr hohe interne Frequenzen. Daher kann nur einer dieser beiden Treiber innerhalb einer Applikation eingesetzt werden.

PWM-Ausgabe

Eine Ausgabe wird durch folgende Instruktion gestartet:

PUT #D, Duty

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Duty	●	●	●	-	-	Duty hat 257 mögliche Werte, von 0 bis 256. 0 = No Duty (Signal dauerhaft low) 256 = 100% Duty (Signal dauerhaft high) 1...255 = einstellbares Duty / Cycle-Verhältnis, z.B. 128 = 50% Duty oder 64 = 25% Duty

Das zuletzt eingestellte PWM-Signal wird solange ausgegeben, bis eine neue Einstellung vorgenommen wird.

Die Basisfrequenz kann während des Programmlaufs jederzeit geändert werden mit:

PUT #D, #1, Basisfrequenz

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Basisfrequenz	●	●	●	-	-	Für die Basisfrequenz gelten die unter Treiberinstallation beschriebnen Einschränkungen. Kann die gewünschte Basisfrequenz nicht exakt eingestellt werden, wird die bestmögliche Annäherung verwendet.

Beispiel: Die gewünschte Trägerfrequenz ist 20 Hz, die erforderliche Basisfrequenz ist also 5120 Hz. Die bestmögliche Annäherung ist Bereich 1 mit Teilerfaktor 122 - dies entspricht 5122 Hz. Diese Basisfrequenz wird dann verwendet.

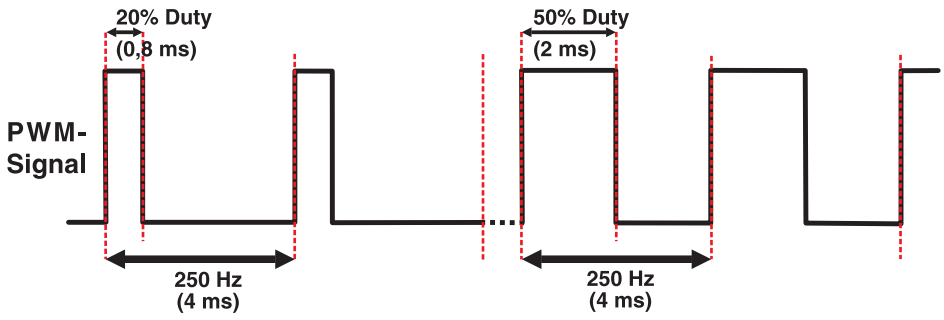
PWMX (Pulsweitenmodulation)

Beispiel:

Ausgabe von eines PMW-Signals auf Pin L80 mit 250 Hz Trägerfrequenz und einer Duty von zunächst 20%, nach 5 Sekunden dann 50%, nach weiteren 5 Sekunden stoppen.

```
USER_VAR_STRICT                          ' prüfe Variablen-Deklaration

TASK MAIN                                ' Start Task MAIN

LONG N                                   ' deklariere Variablen

INSTALL_DEVICE #2, "PWMX_80.TDD",1,39    ' Basisfreq. 64.1 kHz -->
                                         ' Trägerfreq. 250.4 Hz

N = 51                                   ' Duty ca. 20%
PUT #2, N                                ' Ausgabe des PWM-Signals
WAIT_DURATION 5000                       ' warte 5 Sekunden
N = 128                                  ' Duty 50%
PUT #2, N                                ' Ausgabe des PWM-Signals
WAIT_DURATION 5000                       ' warte 5 Sekunden
N = 0                                    ' Duty 0% (konstant low)
PUT #2, N                                ' Ausgabe des PWM-Signals

END                                      ' Ende der Task MAIN
```

Dieses Programm erzeugt in etwa folgende Ausgabe (zur einfacheren Veranschaulichung sind die Werte gerundet):



PWMX (Pulsweitenmodulation)

leere Seite

SER_XUART (Serieller I/O mit externem UART)

Dateiname: SER_XUART.TDD

INSTALL_DEVICE #D, “SER_XUART.TDD“, Adresse, Typ, Anzahl

Serielle Kommunikation über externe UART-Bausteine

Funktion: Dieser Device-Treiber ermöglicht die serielle Ein- und Ausgabe auf externen UART-Bausteinen, wie z.B. mit dem Erweiterungsmodul EP33-4SER mit 4 zusätzlichen seriellen Schnittstellen.

Parameter:

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Adresse	●	●	●	-	-	Basis-Adresse des ersten UART-Bausteins
Typ	●	●	●	-	-	Typ der angeschlossenen Module (0 = EP33)
Anzahl	●	●	●	-	-	Anzahl der angeschlossenen Module

Jedes EP-33 Modul belegt zwei erweiterte Ports, z.B. 10h und 11h oder 28h und 29h. Bei einem Adressraum von256 möglichen Adressen lassen sich also maximal 128 Module bzw. 512 serielle Schnittstellen ansteuern.

Werden mehrere Module verwendet, sollten die Adressen der Module bündig hintereinander liegen. Bei 3 Modulen mit zusammen 12 Schnittstellen also z.B.:

1. Modul: Adressen 10h + 11h
2. Modul: Adressen 12h + 13h
3. Modul: Adressen 14h + 15h

Die Treiber-Installation würde dann wie folgt aussehen:

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

Die Nummern für die einzelnen Kanäle werden, beginnend bei 0, einfach hochgezählt. In obigem Beispiel werden die einzelnen Kanäle demnach über die Sekundär-Adressen 0 bis 11 angesprochen.

Die Symbole für die verschiedenen User-Function-Codes und deren Parameter sind in den Dateien DEFINE_XUART.INC und UFUNC_XUART.INC zusammengefaßt. Diese sollten in den Quellcode eingebunden werden.

User-Function-Codes des SER_XUART.TDD

User-Function-Codes für Abfragen (Instruktion PUT):

Nr	Symbol	Beschreibung
1	UFCO_TRANSMIT_DATA_DACC	Datenübertragung „senden“
2	UFCO_RECEIVE_DATA_DACC	Datenübertragung „empfangen“
3	UFCO_START_STOP_TRANSMIT	„Senden“ von Daten 0 = stoppen ‹›0 = starten
4	UFCO_START_STOP_RECEIVE	„Empfang“ von Daten 0 = stoppen ‹›0 = starten
5	UFCO_START_STOP_REC_TRA	„Senden“ und „Empfang“ von Daten 0 = stoppen ‹›0 = starten
6	UFCO_ABORT_TRANSMIT	„Senden“ von Daten sofort abbrechen
7	UFCO_ABORT_RECEIVE	„Empfang“ von Daten sofort abbrechen
8	UFCO_ABORT_REC_TRA	„Senden“ und „Empfang“ von Daten sofort abbrechen
9	UFCO_RESET_MODULE	Ein Modul resetten (4 Kanäle)
10	UFCO_RESET_ALL_MODULES	Alle Module resetten
11	UFCO_SET_BAUD_PARAM	Schnittstellenparameter setzen (SER1B-kompatible Werte)
12	UFCO_SET_BAUD_PARAMX	Schnittstellenparameter setzen (XUART-Werte)

Einstellen der Schnittstellenparameter

Die beiden Parameter für einen seriellen Kanal (Baudrate und Bit/Parität) können über zwei User-Function-Codes eingestellt werden. Im ersten Fall werden dieselben Parameterwerte wie beim Treiber SER1B verwendet, im zweiten Fall werden spezielle Werte für den Treiber SER_XUART verwendet. Nachfolgend ein Beispiel für das Einstellen der Kanäle 2 und 3 auf 38.400 Baud, 8 Datenbit, 1 Stopbit und Even Parity, einmal mit SER1B-kompatiblen Werten (Kanal 2) und einmal mit XUART-Werten (Kanal 3):

```
PUT #33, #2, #UFCO_SET_BAUD_PARAM, BD_38_400, DP_8E  ' mit SER1B-Parametern
PUT #33, #3, #UFCO_SET_BAUD_PARAMX, BR_38400,&       ' mit XUART-Parametern
  WORD_LENGTH_8 + STOP_BIT_1 + EVEN_PARITY
```

SER_XUART (Serieller I/O mit externem UART)

Verfügbare Einstellungen für die Baudrate mit den XUART-Parameterwerten:

Nr (dez.)	Nr. (hex)	Symbol	Bedeutung
18432	4800h	BR_50	50 Baud
12288	3000h	BR_75	75 Baud
9216	2400h	BR_100	100 Baud
8378	20BAh	BR_110	110 Baud
6144	1800h	BR_150	150 Baud
4608	1200h	BR_200	200 Baud
3072	0C00h	BR_300	300 Baud
2304	0900h	BR_400	400 Baud
1536	0600h	BR_600	600 Baud
1024	0400h	BR_900	900 Baud
768	0300h	BR_1200	1.200 Baud
512	0200h	BR_1800	1.800 Baud
384	0180h	BR_2400	2.400 Baud
256	0100h	BR_3600	3.600 Baud
192	00C0h	BR_4800	4.800 Baud
128	0080h	BR_7200	7.200 Baud
96	0060h	BR_9600	9.600 Baud
64	0040h	BR_14400	14.400 Baud
48	0030h	BR_19200	19.200 Baud
32	0020h	BR_28800	28.800 Baud
29	001Dh	BR_31250	31.250 Baud
24	0018h	BR_38400	38.400 Baud
16	0010h	BR_57600	57.600 Baud (Default)
15	000Fh	BR_62500	62.500 Baud
12	000Ch	BR_76800	76.800 Baud
8	0008h	BR_115200	115.200 Baud
6	0006h	BR_153600	153.600 Baud
4	0004h	BR_230400	230.400 Baud
3	0003h	BR_307200	307.200 Baud
2	0002h	BR_460800	460.800 Baud
1	0001h	BR_921600	921.600 Baud

Verfügbare Einstellungen für Bits & Parität mit den XUART-Parameterwerten:

Nr (dez.)	Nr. (hex)	Symbol	Bedeutung
0	00h	WORD_LENGTH_5	5 Datenbits
1	01h	WORD_LENGTH_6	6 Datenbits
2	02h	WORD_LENGTH_7	7 Datenbits
3	03h	WORD_LENGTH_8	8 Datenbits
0	00h	STOP_BIT_1	1 Stopbit
4	04h	STOP_BIT_1_2	1½ Stopbits
4	04h	STOP_BIT_2	2 Stopbits
0	00h	NO_PARITY	Keine Parität
8	08h	ODD_PARITY	Ungerade Parität
24	18h	EVEN_PARITY	Gerade Parität
40	28h	MARK_PARITY	Markierung
56	38h	SPACE_PARITY	Leerzeichen

Für eine gültige Einstellung wird je eine Anzahl von Datenbits, eine Anzahl von Stopbits und eine Parität kombiniert, die entsprechenden Werte addiert. Für 7 Datenbit, 1 Stopbit, gerade Parität kann man als Parameter also entweder den Wert 26 schreiben oder die einzelnen Symbole addieren (WORD_LENGTH_7 + STOP_BIT_1 + EVEN_PARITY).

Zeichen ausgeben

Die Ausgabe erfolgt durch Übergabe des zu sendenden Strings an den Treiber über den User-Function-Code UFCO_TRANSMIT_DATA_DACC.

PUT #D, #C, #UFCO_TRANSMIT_DATA_DACC, Text$ [, Offset, Anzahl]

Die optionalen Parameter dieser Funktion ermöglichen die Ausgabe ab einer bestimmten Position im String (Offset) sowie einer bestimmten Anzahl von Zeichen (0 = bis zum Ende des Strings).

Beispiel für das Senden eines Strings auf Kanal 0:

```
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT$        ' Ausgabe auf Kanal 0
```

Darüber hinaus verfügt der Treiber über eine Reload-Funktion, d.h. noch während der aktuelle String gesendet wird, kann bereits der nächste String an den Treiber übergeben werden. So ist kontinuierliches Senden möglich.

Beispiel für das unterbrechungsfreie Senden zweier Strings auf Kanal 0 mit Reload:

```
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$        ' Ausgabe auf Kanal 0
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$        ' Reload auf Kanal 0
```

Durch die Abfrage der Stringlänge mit der Funktion LEN kann ermittelt werden, wieviele Zeichen eines Strings noch zu sendensind. Dieser Wert wird ständig aktualisiert. So kann mit der nächsten Ausgabe eines Strings gewartet werden, bis die vorherige Ausgabe beendet ist.

Beispiel für das 'Warten' auf vollständiges Versenden eines Strings:

```
FOR EVER = 0 TO 0 STEP 0                              ' Endlosschleife
  TEXT$ = STR$(TICKS())                               ' Sendestring erstellen
  PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT$        ' auf Kanal 0 ausgeben
  WHILE LEN(TEXT$) > 0                                ' Hier warten bis String
  ENDWHILE                                            ' komplett gesendet ist
NEXT                                                  '
```

Natürlich kann dieses 'Warten' auch mit der Reload-Funktion kombiniert werden. Während ein String noch versendet wird, ist der zweite (wieder) frei und kann bereits neu besetzt und an den Treiber übergeben werden.

Beispiel: Zwei Strings werden wechselweise geschrieben, sobald sie wieder frei sind:

```
TEXT1$ = "1:" + STR$(TICKS())                         ' 1.Sendestring erzeugen
TEXT2$ = "2:" + STR$(TICKS())                         ' 2.Sendestring erzeugen
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$         ' 1.Sendestring ausgeben
PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$         ' 2.Sendestring ausgeben
FOR EVER = 0 TO 0 STEP 0                              ' Endlosschleife
  IF LEN(TEXT1$) = 0 THEN                             ' Falls 1.String gesendet
    TEXT1$ = "1:" + STR$(TICKS())                     ' neuer 1.Sendestring
    PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT1$     ' neue Ausgabe 1.String
  ENDIF                                               '
  IF LEN(TEXT2$) = 0 THEN                             ' Falls 2.String gesendet
    TEXT2$ = "2:" + STR$(TICKS())                     ' neuer 2.Sendestring
    PUT #33, #0, #UFCO_TRANSMIT_DATA_DACC, TEXT2$     ' neue Ausgabe 2.String
  ENDIF                                               '
NEXT                                                  '
```

Zeichen einlesen

Empfangene Zeichen werden vom Device-Treiber in den über den User-Function-Code UFCO_RECEIVE_DATA_DACC übergebenen String gespeichert.

SER_XUART (Serieller I/O mit externem UART)

PUT #D, #C, #UFCO_TRANSMIT_DATA_DACC, Text$ [, Offset, Anzahl]

Die optionalen Parameter dieser Funktion ermöglichen, daß die empfangenen Zeichen ab einer bestimmten Position im String (Offset) gespeichert werden sowie daß nur eine bestimmte Anzahl von Zeichen gespeichert wird (0 = bis zum Ende des Strings). Wird kein Parameter angegeben, wird der komplette String gefüllt.

Beispiel für das Einlesen in einen Strings von Kanal 0:

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text$          ' Einlesen von Kanal 0
```

Wird ein neuer String mit dieser Funktion an den Device-Treiber übergeben, werden sofort alle empfangenen Zeichen in diesem neuen String gespeichert, selbst wenn der alte String noch nicht komplett gefüllt ist.

Darüber hinaus verfügt der Treiber auch für den Empfang über eine Reload-Funktion, d.h. noch während im aktuellen String empfangen wird, kann bereits der nächste String an den Treiber übergeben werden. Für die Verwendung der Reload-Funktion müssen die Parameter Offset und Anzahl angegeben werden. In diesem Fall wird wird der erste String beginnend ab Position Offset mit Anzahl Zeichen befüllt, bevor der zweite String befüllt wird. Es wird also nicht sofort umgeschaltet.

Beispiel: Zwei Strings mit der Länge 100 werden für den Empfang genutzt. Erst wenn der String Text1$ voll ist, werden empfangene Zeichen in Text2$ gespeichert.

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text1$, 0, 100 ' Einlesen von Kanal 0
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text2$, 0, 100 ' Einlesen von Kanal 0
```

Auch hier kann die aktuelle Länge des Strings mit der Funktion LEN ermittelt werden. So kann man feststellen, wieviele Zeichen bereits im String abgespeichert sind und ob demzufolge ein neuer Empfangsstring an den Device-Treiber übergeben werden sollte, um einen unterbrechungsfreien Empfang zu gewährleisten.

Beispiel: Zwei Strings mit der Länge 500 werden für den Empfang benutzt. Kurz bevor einer voll ist, wird der andere für den Empfang genutzt und der erste z.B. an einen globalen Puffer-String angehängt.

```
PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text1$      ' 1.Empfangsstring
FOR EVER = 0 TO 0 STEP 0                          ' Endlosschleife
  IF LEN(Text1$) > 450 THEN                       ' Falls String fast voll
    PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, Text2$  ' 2.Empfangsstring
    Puffer$ = Puffer$ + Text1$                    ' 1.String in Puffer
  ENDIF                                           '
```

```
    IF LEN(Text2$) > 450 THEN                          ' Falls String fast voll
      PUT #33, #0, #UFCO_RECEIVE_DATA_DACC, TEXT1$ '  1.Empfangsstring
      Puffer$ = Puffer$ + Text2$                       ' 2.String in Puffer
    ENDIF                                              '
  NEXT                                                 '
```

Module zurücksetzen (Reset)

Es gibt zwei User-Function-Codes für das Resetten von Modulen. Es können entweder ein Modul oder alle im System befindlichen Module zurückgesetzt werden. Die Sekundäradresse bestimmt dabei, welches Modul zurückgesetzt wird.

Beispiele:

```
PUT #33, #0, #UFCO_RESET_MODULE, 0        ' Kanal 0: Reset 1.Modul (Ch 0-3)
PUT #33, #3, #UFCO_RESET_MODULE, 0        ' Kanal 3: Reset 1.Modul (Ch 0-3)
PUT #33, #4, #UFCO_RESET_MODULE, 0        ' Kanal 4: Reset 2.Modul (Ch 4-7)
PUT #33, #0, #UFCO_RESET_ALL_MODULES, 0   ' Kanal egal: Reset alle Module
```

Hinweis zur Adressierung der Module

Wie bereits erwähnt hat jedes EP33-4SER Modul 2 physikalische Adressen. Während beim Device-Treiber immer die untere der beiden Adressen angegeben wird (z.B. 10h), muß am Modul selbst immer die höhere Adresse angegeben werden (z.B. 11h). Dies muß unbedingt beachtet werden um eine ordnungsgemäße Funktion zu gewährleisten.

leere Seite

SHI (Getaktete, serielle Eingabe)

Dateiname: SHI_PpPp.TDD

```
INSTALL_DEVICE #D, "SHI_PpPp.TDD", Timeout, Rec_Edge,                    &
                                   Bit_Seq, Byte_Seq, Bits_per_Byte
```

getaktete, serielle Eingabe

Funktion: Dieser Device-Treiber ermöglicht eine getaktete, serielle Eingabe von einem externen Chip. Die ersten beiden Ziffern im Namen des Treiber geben Port und Pin der Taktleitung an, die letzten beiden Ziffern Port und Pin der Datenleitung.

Parameter:

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
Timeout	●	●	●	-	-	Timeout in ms (2...255) zur Erkennung eines Datenpakets
Rec_Edge	●	●	●	-	-	Legt fest, bei welcher Takt-Flanke die Datenbits eingelesen werden: 0=steigende Flanke, X=fallende Flanke
Bit_Seq	●	●	●	-	-	Reihenfolge der empfangenen Bits: 0=Low-Bit-first, X=High-Bit-first
Byte_Seq	●	●	●	-	-	Reihenfolge der empfangenen Bytes: 0=Low-Bit-first, X=High-Bit-first
Bits_per_Byte	●	●	●	-	-	Anzahl der je Byte empfangenen Bits (1...8)

Der Gerätetreiber-Satz "SHI_PpPp.TDD" dient zum Einlesen von getakteten, seriellen Datenströmen in den BASIC-Tiger. Anders als bei Byte-orientierten seriellen Daten, wo Bytes aus einem seriellen Datenstrom abgegriffen werden, arbeitet dieser Treiber wie ein serieller Shift-Register Eingang, der die Daten Bit für Bit sequentiell einliest.

Es werden also keine expliziten Paket-Begrenzer wie Start- oder Stopbits erwartet, diese können allerdings empfangen und später im Programm entfernt werden.

Der Treiber benötigt zwei Eingangsleitungen: Takt (Clock) und Data.

Die Taktleitung darf sowohl inverierte wie auch nicht invertierte Pulse für den Takt verwenden, der Treiber kann außerdem so eingestellt werden, daß die Datenleitung entweder bei fallender oder bei steigender Flanke ausgelesen wird.

SHI (Getaktete, serielle Eingabe)

Bei jedem Takt wird ein Bit von der Datenleitung gelesen und entsprechend den folgenden Einstellugnen in den internen Puffer geschrieben:

- Das erste Bit ist MSB oder LSB.
- Die erste empfangene Bitgruppe (z.B. ein Byte) ist die erste oder die letzte im Puffer.
- Empfangene Bits können mit einer Rate von 1 Bit je Byte bis zu 8 Bits je Byte in den Puffer übertragen werden.
  Empfangene Formate mit mehr als 8 Bits je Bitgruppe (z.B. 16 Datenbits + 1 Startbit + 1 Stopbit = 18 Bits) werden vom Treiber in einem 8 Bit je Byte Schema verarbeitet, und später mit einer Konvertierungsfunktion in BASIC zurückgewandelt.

Weiterhin werden keine “Strobe“ oder “Start“ Signale benötigt, um den Empfang eines Datenstroms zu initiieren. Der Treiber erfaßt den Anfang und das Ende eines Datenstroms durch die Verwendung des Timeout-Zustands.

Wenn das Taktsignal zumindest für die Länge der Timeout-Periode pausiert, behandelt der Treiber alle bis dahin empfangenen Datenbits als einen kompletten Satz empfangener Bits. Diese können vom BASIC-Programm ab diesem Zeitpunkt mit einer “GET“ Instruktion ausgelesen werden. Nach dem “GET“ ist der Datenpuffer gelöscht.

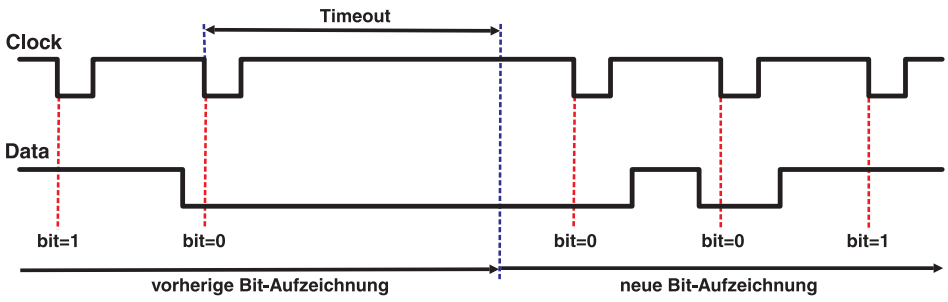
Werden die Daten nicht durch das BASIC-Programm mit einem “GET“ ausgelesen, wird die vorherige Bitaufzeichnung durch neue Daten überschrieben, durch einen “double-buffer“ Mechanismus wird jedoch sichergestellt, daß eine Datenaufzeichnung immer komplett ist und nicht die Kombination zweier unvollständiger Aufzeichnungen.

Weitere Spezifikationen des Treibers:

- Max. Taktfrequenz: 20.000 Hz
- Min. Taktfrequenz: 4 Hz (aufgrund der 255 ms Timeout Obergrenze)
- Timeouts: 2...255 ms

  Hinweis: Eine Timeout Einstellung von “2“ erzeugt ein Timeout von 1...2 ms
  Eine Timeout Einstellung von “x“ erzeugt ein Timeout von (x-1)...x ms

Logik-Diagramm:



SHI (Getaktete, serielle Eingabe)

Das Auslesen eines seriellen Datenstroms aus dem Puffer des Treibers erfolgt mit einer GET-Instruktion:

GET #D, 0, A$

	B	W	L	S	F	
D	●	●	●	-	-	Gerätenummer des Treibers
A$	●	●	●	-	-	String, der alle im Eingangspuffer enthaltenen Datenbytes aufnimmt

Beispiel:

Ausgabe eines getakteten, seriellen Datenstroms mit der SHIFT_OUT Funktion über die Pins L84 und L85, Empfang dieser Daten mit dem SHI-Treiber auf den Pins L80 und L81.

```
USER_VAR_STRICT                   ' prüfe Variablen-Deklaration

TASK MAIN                         ' Start der Task MAIN

STRING Send$, Rec$                ' Strings für Senden und Empfangen
LONG C                            ' Zählvariable

INSTALL_DEVICE #1, "LCD1.TDD"                      ' LCD installieren
INSTALL_DEVICE #2, "SHI_8081.TDD", 2, 1, 0, 1, 8 ' 2ms Timeout, fallende
                                                   ' Flanke, LSB, HBF, 8 bpb

DIR_PIN 8,4,0                     ' L84 ist Ausgang (Datenpin für SHIFT_OUT)
DIR_PIN 8,5,0                     ' L85 ist Ausgang (Taktpin für SHIFT_OUT)
OUT  8,00110000b,00000000b        ' setze L84 und L85 auf Ruhezustand

FOR C = 100 TO 149                ' einige Durchläufe...
  Send$ = "Test" + STR$(C)        ' Sendestring aufbauen

  SHIFT_OUT 8, 4, 5, Send$, -64 ' 64 bits werden geshifted,
                                  ' höchstwertiges (MSB) zuerst
  WAIT_DURATION 10                ' warte kurz bis Senden und Empfang
                                  ' beendet sind
  GET #2, 0, Rec$                 ' Inhalt des Treiber-Puffers einlesen

  PRINT #1, "<1>"; Send$          ' Sendestring auf LCD ausgeben
  PRINT #1, Rec$                  ' Empfangsstring auf LCD ausgeben
  WAIT_DURATION 500               ' 500ms warten
NEXT                              ' nächster Durchlauf

END                               ' Ende der Task MAIN
```

SHI (Getaktete, serielle Eingabe)

leere Seite

Applikationen

leere Seite

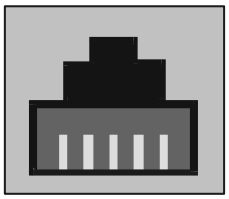
Ethernet Application

Funktionsbibliotheken: ts_???.inc

Anschluß des Tigers an ein Netzwerk

Funktion: Diese Bibliotheken stellen Funktionen zur Verfügung, mit denen über einen Ethernet/Web Adapter eine Verbindung zu einem Netzwerk hergestellt werden kann.

Die Funktionen beinhalten sowohl die Konfiguration des Ethernet/Web Adapters, als auch den Verbindungsaufbau und den Datenaustausch.



Dabei können für die Kommunikation verschiedene Protokolle verwendet werden, wie z.B. POP/SMTP (für Email), HTTP (für Internet), ARP, TCP/IP, DHCP oder DNS.

Die nachfolgende Beschreibung (in englischer Sprache) erläutert zunächst alle in diesen Bibliotheken enthaltenen Funktionen, sowie wie diese eingesetzt werden. Dies geschieht anhand einer Reihe von Beispielprogrammen, deren Funktionsweise und verwendete Funktionen erläutert werden.

Sollten Sie eine Compiler-Version ab 5.2 installiert haben, finden Sie diese Programme im Verzeichnis „Examples_Web_Ethernet“ Ihrer Tiger-Installation. Andernfalls können Sie diese auch von unserer Webseite „www.wilke.de“ laden.

Ethernet/Web Adapter

Programming Guide

Version 1.01a

Table of contents

Inhalt

1. Introduction

1.1 About this document

Informationen zu dieser Beschreibung

- This document refers to the **Ethernet/Web Adapter** types **EM01-ETH-S, EM02-SER-S, EM03-ETH-P, EM04-SER-P**.
- The information in this document 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).
- In this document if the notation Ethernet/Web Adapter is used, then 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 describes only EM02-SER-S and EM04-SER-P modules.
- This document consists of two main chapters. The **„Demo Programs“** chapter introduces the client/server programming with Ethernet/Web Adapter and Tiger Basic and describes in details the sample programs:
  - *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 in loop simple messages to the remote peer.
  - *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 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 user to comply with the requirements of the particular SMTP server.

  The **„Programming with Tiger Basic Sockets (TBSockets)“** chapter 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, read the „Ethernet_Adapter__Protocol_[Vers].pdf“ document additionally.

1.2 Protocols and Services Provided

1.2.1 Ethernet Adapter

Unterstützte Protokolle

- 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

Erläuterungen zur Software

- The Ethernet/Web Adapter is pre-programmed to be able to exchange the commands and data with 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 of a collection of tasks and subroutines named Tiger Basic Sockets (TBSockets). TBSockets are written in Tiger Basic programming language and delivered as source code. To compile TBSockets and appropriate demos the **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

Begriffserklärung

IP address: An 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 assign IP addresses at random 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: In TCP/IP and UDP networks, an endpoint to a logical connection. The port number identifies what type of port it is. For example, port 80 is used for HTTP traffic.

DNS: An Internet service that translates domain names into IP addresses. Because domain names are alphabetic, they're easier to remember. The Internet however, is really based on IP addresses. Every time you use a domain name, therefore, 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: A 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 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 the 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 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

Netzwerk-Konfiguration

- 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 peer host (e.g. PC).
- If a crossover cable connected directly to peer host is used, static IP addresses and subnet masks are required.
- If a router is used in the net to which the PC (and the Ethernet/Web Adapter) will be attached, 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 disturb other network participants.

2.3 Ethernet/Web Adapter Settings

Adapter-Einstellungen, Konfigurations-daten

- 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 Baud** for **EM01**, and **19200 Baud** 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 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 the „ts_coinc.inc“, the „ts_conf.inc“ is 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

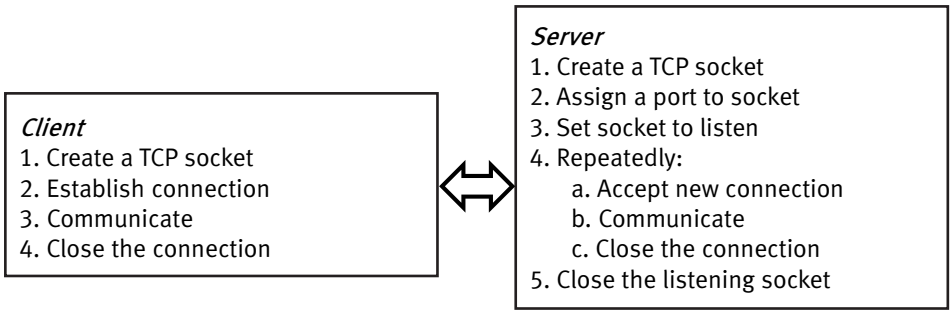
Einwahlprozedur bei Web-Adapter

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 *bDiallsp* subroutine call (or with the *bDiallspWithLogin* 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 *bSetuplsp* subroutine defines the dialling number of the ISP (other parameters of this subroutine are used only if the *bDiallspWithLogin* mechanism must be activated).

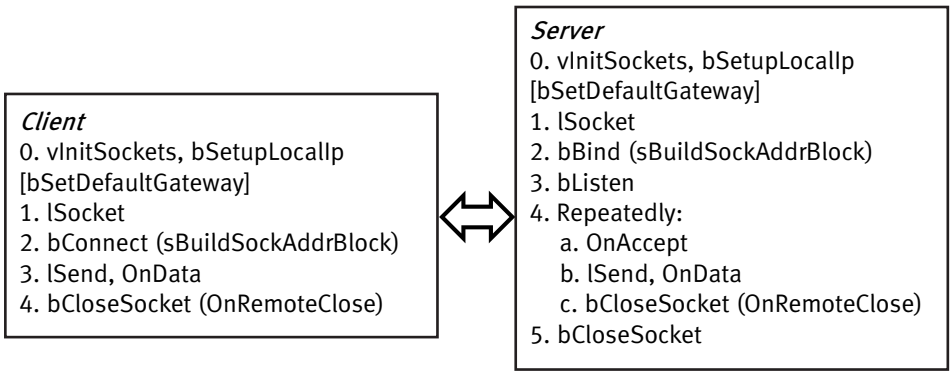
2.5 Client/Server Modell

2.5.1 TCP Client/Server Interaction

Interaktion von Client und Server



2.5.2 TCP Client/Server Interaction using TBSockets



2.5.2.1 How to build a very simple TCP Client using TBSockets

Einrichten eines einfachen 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 success of connecting),
- Send the data (*lSend*),
- Receive the data serving a callback task (*OnData*) launched each time when the data are coming,
- Close the socket (*bCloseSocket*).

2.5.2.2 How to build a very simple TCP Server using TBSockets

Einrichten eines einfachen 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 when the data are coming,
- Close all open sockets (*bCloseSocket*).

All subroutines and tasks that may be of importance for implementing of client/server applications with Ethernet/Web Adapter are described in detail below in this „Programming Guide“ manual.

2.6 Simple Client Demo

Beispiel für eine Client-Applikation

- 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 placeholders for user-defined reactions on some asynchronous events like coming 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 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 getaway 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

Beispiel für eine Server-Applikation

- 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 placeholders 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 in loop simple messages to the remote peer.

- 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 that must be correctly set by 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 on (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 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

Beispiel für einen 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 placeholders for user-defined reactions on some asynchronous events like success of 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 of a new (first) socket forces (*lSocket)* the Ethernet Adapter to send 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 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 getaway 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

Beispiel für einen 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 placeholders for user-defined reactions on some asynchronous events like coming 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 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 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 getaway 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

Beispiel für einen 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 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 was connected, an email will be prepared and sent; the sending of an email involves:
     - Setting all values relevant to the authentication of the owner of the email account 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:
  The most of 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

- 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 getaway address in the „GATEWAY“ section (if a router is a 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 placed in the *smtp_client_ethe.tig* file directly:

- 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

Beschreibung der Tiger-Basic Sockets

TBSockets is a collection of tasks and subroutines written in the Tiger Basic programming language and implementing 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

Setzen der lokalen IP-Adresse und 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

Setzen des 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

Auslesen der lokalen IP-Adresse

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

Auslesen des lokalen Ports

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

IP-Adresse des Server auslesen

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

Port des Server auslesen

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

Firmware-Version auslesen

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)

Ethernet MAC (Media Access Control) address is a hardware address that 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, the using of this subroutine must be avoided, except an area of unique MAC addresses was additionally reserved for produced series, otherwise the change of the MAC address can be dangerous.

3.4.1 Set MAC Address

Neue Ethernet MAC Adresse setzen

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

Ethernet MAC Adresse des Adapters lesen

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

Setzen der TCP Fenstergröße

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 be greater than 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

Zyklisches Senden von Dummy-Daten zum Testen und Erhalten einer aktiven Verbindung

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

Auslesen der TCP Einstellungen

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 with Modem directly

Funktionen für Kommunikation mit Modem

The subroutines of this subsection enable the data exchange immediately with a modem, without involving any function of a Web Adapter that 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 in the *spvReceivedData$* string the data of the *lpRecvBufSize* maximal size from the modem.

Signature:
sub bModemReceive(var string spvReceivedData$; long lpRecvBufSize; long lpTimeOut; var byte bpvIsReceived)

Purpose:
The *wModemGetSendReady* subroutine returns in the *wpvSendSize* parameter the number of bytes of data that a modem can accept for sending.

Signature:
sub wModemGetSendReady(long lpTimeOut; var word wpvSendSize)

Purpose:
The *wModemGetRecvReady* subroutine returns in the *wpvRecvSize* parameter the number of bytes of unprocessed data the modem has in its receive buffer.

Signature:
sub wModemGetRecvReady(long lpTimeOut; var word wpvRecvSize)

Return:
All above subroutines try to get the particular work done during the *lpTimeOut* period of time. If the *lpTimeOut* was expired and the function was 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

Senden von AT-Kommandos

Purpose:
The *bSendATCommand* subroutine sends the *spATCommandToSend$* AT command to the modem and receives reply in the *spvATReply$*.

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 that is set now to 128 (see: „AT_REPLY_MAX_SIZE“) must not be changed by user. The *lpATTimeOut* is a timeout value measured in seconds to wait for reply from the modem.

3.6.3 Listen to Modem Data

Modem „belauschen“

The alternative mechanism to communicate with the modem is listening to the modem until a certain number of data has come 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 that was started by the *bModemStartListen* call.

Signature:
sub bModemStopListen(var byte bpvSuccess)

Purpose:
The callback task that is launched when the data is come.

Signature:
task OnModemData

Comments:
- Two global variables are set by the TBSockets Launcher at launching this task: *lActModemDataSize* and *sActModemDataBuffer*$.
- The *lActModemDataSize* variable stores the size of the coming data.
- The *sActModemDataBuffer*$ variable stores the coming data itself.
- The *OnModemData* task is normally written by 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'.

3.6.4 Dialling Procedure

Wahl-Prozedur

The dialling procedure for modem consists of dialling an Internet Service Provider (phone number set by *bSetupIsp*) and authenticating of the user (user name and password set by *bSetupIsp* or *bSetPapSecrets* correspondingly to the chosen scheme of dialling).

3.6.4.1 Set Internet Service Provider Data

Einstellungen für den ISP

Purpose:
The *bSetupIsp* subroutine sets the data used while dialling 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)

Einstellen der Authentizierungs-parameter

Purpose:
The *bSetPapSecrets* subroutine is used to define the authentication parameters transferred to ISP as a part of PPP protocol. The so named 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. Maximal number of the 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

Einwahl beim ISP mit Login-Daten

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

Einwahl beim ISP ohne Login-Daten

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 the sending of user name and password, because the ISP itself establishes the PPP connection and asks the client about 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 give back in the *lpvAssignedIpAddr* parameter an IP address assigned by the ISP to our node.
- The *lpDialTimeout* defines the timeout to wait for connection to the ISP.

3.6.5 Hanging Up Procedure

Auflegen/ Verbindung zum ISP trennen

Purpose:
The *bHangUp* subroutine forces the adapter to finish the modem session 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 Baudrate

Modem-Baudrate festlegen

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

Status des CTS-Pins feststellen

Purpose:
The *bGetCtsPinState* subroutine returns in the *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 für das DHCP Protokoll

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

DHCP Protokoll (de)aktivieren

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

Vom DHCP Server zugewiesene IP-Adresse holen

Purpose:
If the request succeeds, the *OnDhcpUp* callback task is launched and the new ip address assigned by 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 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

Routinen für das DNS Protokoll

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

DNS Protokoll (de)aktivieren

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 sets also 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

IP-Adresse für einen Host vom DNS Server holen

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

Routinen für Client/ Server Kommunikation

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

Socket öffnen (Verbindung öffnen)

Purpose:
The subroutine used by both, clients and servers, 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

Socket/Verbindung schließen

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 on accepting a new client.

3.9.1.3 Remote Socket Closed Notification

Nachricht daß Verbindung von Gegenstelle getrennt wurde

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

Aufbau eines Socket Adress Blocks

Various subroutines of the sockets family expect the string generally referenced 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 facilitate considerably the accessing of 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 the *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 to the *bpvSaLength*, *bpvSaFamily*, *wpvSaPort*, *lpvSaAddress*.

3.9.2 Client

Funktionen für einen 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

Verbinde über einen bestimmten Port mit Host

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 we have talked about extensively. In this case, the *port* and *ip address* identify the server that 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 outright *refuse* any request for specific code.

3.9.3 Server

Funktionen für einen Server

The typical server does not initiate the connection. Instead, it waits for a client to call it and request services. It does not know 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

Weise einem Socket einen bestimmten Port zu

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$* string is a Socket Address Block of the *wpSockAddrLen* length. Beside specifying the *port* in *spSockAddr$*, 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

Warte auf eingehende Verbindung

Purpose:
The server waits for 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

Verbindung vom Server akzeptiert

Purpose:
The server accepts the connection by starting the '*OnAccept*' task.

Signature:
task OnAccept

Comments:

- Three global variables are set by 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 meant only 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 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

Routinen für Datentransfer

Once the connection is established the data can be exchanged in both directions.

3.9.4.1 Send

Senden von Daten

Purpose:
The *lSend* subroutine must be used to send the data to the remote host.

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 by the *lSend* subroutine. The l*LastErrorCode* variable contains the error code if the 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 the future extensions and is not used now.

3.9.4.2 Data Received Notification

Nachricht, daß Daten ankommen

Purpose:
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 to 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 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'.

3.10 Error Handling and Error Codes

Übersicht über Fehlerbehandlung und Fehlermeldungen

- If the return value is not explicitly described for the particular subroutine, the 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).

- 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) - the command not available
  CME_SEND_AT_MDM_NOT_INIT (2) - the modem is not yet initialised
  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 yet initialised
  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) - the command not available
  CME_DIAL_BAD_SUBCOMMAND (2) - the 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 withing 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 return request. The return 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 is 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) – the 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

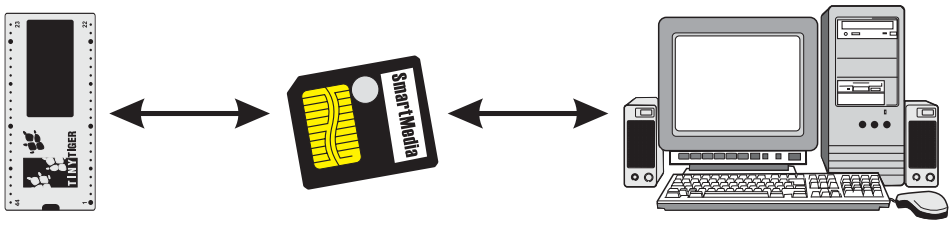
SmartMedia Application

Funktionsbibliotheken: fs_???.inc

Schreiben und Lesen von SmartMedia-Karten mit FAT-16 File-System

Funktion: Im diesen Bibliotheken werden Funktionen zur Verfügung gestellt, die das Arbeiten mit einem FAT-16 File-System auf einer an den BASIC-Tiger angeschlossenen SmartMedia Card ermöglichen.

Mit diesen Funktionen ist es also möglich, eine mit dem BASIC-Tiger beschriebene SmartMedia-Karte am PC auszulesen und umgekehrt. Dies vereinfacht erheblich den Datentransfer zwischen Tiger und PC.



Dies funktioniert nicht nur mit dem PC, sondern gleichermaßen mit allen anderen Geräten, die das FAT-16 File-System verwenden.

Zur Verwendung von SmartMedia-Karten werden ebenfalls die entsprechenden Device-Treiber benötigt (Smedia_xxMB.tdd), über die der Tiger mit der Karte kommuniziert.

Nachfolgend finden Sie eine Beschreibung (in englischer Sprache) zu allen in diesen Bibliotheken enthaltenen Funktionen, sowie diese eingesetzt werden. Analog gibt es dazu eine Reihe von Beispielprogrammen, auf die im folgenden verwiesen wird.

Sollten Sie eine Compiler-Version ab 5.2 installiert haben, finden Sie diese Programme im Verzeichnis „Examples_SmartMedia“ Ihrer Tiger-Installation. Andernfalls können Sie diese auch von unserer Webseite „www.wilke.de“ laden.

Basic Tiger File System for SmartMedia

Version 1.04

Table of contents

Inhalt

leere Seite

1. Introduction

Allgemeine Informationen

Basic Tiger File System (BTFS) is a collection of subroutines written in the Tiger Basic programming language and implementing general functionality of 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 the present document.

FAT is an implementation of 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 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“)

zugehörige Dateien

fs_conf.inc	definitions that can be changed by user
fs_coinc.inc	definitions relevant for all FS layers; co-including of 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

unterstützte Medien und Einschränkungen

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

Einrichten des File-Systems

Subroutine:
sub bFileSystemHardwareInit(var byte bpvHdInitOk)

The *bFileSystemHardwareInit* subroutine calls special subroutines initializing a particular storage medium (f.e.: 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 may take a long time 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 the File

Öffnen und Schließen von Dateien

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 is 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.

Zugriffsarten

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.

Beispiel

Example: „file_open.tig“

2.2.2 Closing the 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 the File

Lesen und Schreiben von Dateien

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* while at end-of-file, it will keep returning zero and doing 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}.

Beispiel

Example: „file_open.tig“

2.3.2 Writing the 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 the data is written.
In the case of an error, *lWriteFile* returns {-1}.

Beispiel

Example: „file_open.tig“

2.4 Setting and Getting the File Position of a Descriptor

Position innerhalb einer Datei lesen/ setzen

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 file descriptor is invalid, *lGetFilePointer* returns a value of {-1}.

Beispiel

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 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* is normally 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, though 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 of *lSetFilePointer*.

Beispiel Example: „file_pointer.tig“

2.5 Getting the File Size

Lesen der Dateigröße 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.

Beispiel 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.

Beispiel Example: „dir_create_del.tig“

2.7 Deleting Files and Directories

Löschen von Dateien und Verzeichnissen

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.

Beispiel

Example: „dir_create_del.tig“

2.8 Setting Current Directory

Wechsel des aktuellen Verzeichnisses

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 begins always 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.

Beispiel

Example: „dir_create_del.tig“

2.9 File Attributes

Dateiattribute auslesen und setzen

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 be only 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

Dateiattribute auslesen

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.

Beispiel

Example: „file_attributes.tig“

2.9.2 Setting the File Attributes

Dateiattribute setzen

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.

Beispiel

Example: „file_attributes.tig“

2.10 File Time

2.10.1 Time and Date Format

Datum und Zeitangabe für Dateien lesen und setzen

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

Dateidatum und -uhrzeit auslesen

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
: The date the file was created.

wpvCreateTime
: The time the file was created.

wpvAccessDate
: The date the file was last accessed.

wpvWriteDate
: The date the file was last modified.

wpvWriteTime
: The time the file was last modified.

All the time and date fields are represented in the format described in the „Time and Date Format“.

Beispiel

Example: „file_time.tig“

2.10.3 Setting the File Time

Dateidatum und -uhrzeit setzen

Subroutine:
sub bSetFileTime(string spFileName$; word wpCreateDate, wpCreateTime, wpAccessDate, wpWriteDate, wpWriteTime; var byte bpvIsTimeWritten)

The *bSetFileTime* subroutine sets the date and time that a file *spFileName$* was created, last accessed, and last modified.

wpCreateDate
The date the file was created.
wpCreateTime
The time the file was created.
wpAccessDate
The date the file was last accessed.
wpWriteDate
The date the file was last modified.
wpWriteTime
The time the file was last modified.

All the time and date fields are represented in the format described in the „Time and Date Format“.

Beispiel

Example: „file_time.tig“

2.11 Find File

Auffinden von Dateien

Two subroutines described below return the result of the searching 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:

Informationen über eine Datei

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 (the 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 was 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 for the file name

Suchen nach Dateiname

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*$ filename
and fills on success the *spvFfdStruct$* string with the information about the found file. The *spSearchedFileName*$ 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 the 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.

Beispiel

Example: „file_find.tig“

2.12 Getting the information about the storage media

Informationen über das Medium

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 greater than HDI_BLOCK_SIZE.

Beispiel

Example: „get_hd_info.tig“

2.13 Getting the information about the file system

Informationen über das File-System

Subroutine:
sub bGetFileSystemInfo(var string spvBootRecord$; var byte bpvIsBootRecRead)

The *bGetFileSystemInfo* subroutine reads the information about the file system into the *spvBootRecord*$ string. The 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 the 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.

Beispiel

Example: „get_fs_info.tig“

2.14 Formatting the Storage Media

Medium formatieren

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.

Beispiel

Example: „file_format.tig“

2.15 Synchronizing the File System

File-System mit RAM synchronisieren

For reasons of efficiency, some intensively used data structures of the FAT file system are temporary 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 of 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 in the endless loop the *bSynchronizeFSRegularly* subroutine. The synchronization timeout values are measured in seconds.

Subroutine:
sub vSynchronizeFS()

The *vSynchronizeFS* subroutine writes to the media all data structures that were temporary saved in the RAM.

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 the both *lpvSyncTimeout* and *lpvCurSyncTimeoutCounter.*

Subroutine:
sub lSetSyncTimeout(long lNewSyncTimeout; var long lpvPrevSyncTimeout)

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 yet initialised.

Subroutine:
sub bSynchronizeFSRegularly(var byte bpvTimeoutReached)

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.

Beispiel

Example: „file_sync.tig“

3. What Must Be Done

geplante Verbesserungen

1. Some subroutines are too slow. The execution speed must be increased by means of improved algorithms or built-in functions written in the processor language directly.

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 the efficiency.

8. More comments in the programs and better documentation is everyone's most fervent wish.J

4. Useful References

Referenzen für weitere Informationen

1. SmartMedia Card Specifications:

   http://www.ssfdc.or.jp/english/index.htm

2. About FAT:

   http://averstak.tripod.com/fatdox/00dindex.htm

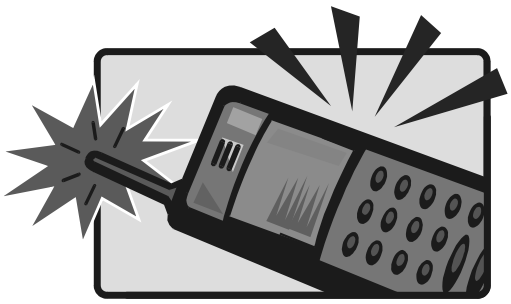
   http://msdn.microsoft.com/

SMS Routinen

Funktionsbibliothek: sms_Vxxx.inc

Senden und Empfangen von SMS Nachrichten

Funktion: Stellt Funktionen zur Verfügung, die das Senden einer SMS bzw. das Lesen einer gespeicherten SMS über ein Handy bzw. Modem ermöglichen.



Allgemeines zu SMS

In der heutigen Zeit hat fast jeder ein Handy. Neben dem Telefonieren die vielleicht noch beliebtere Methode das Handy zu benutzen, ist das Schreiben einer SMS. Wir verschicken eine Nachricht über das Mobilfunknetz und der Empfänger kann diese Nachricht wenige Sekunden später lesen. Wir haben also einen „Sender“ und einen „Empfänger“. In der Regel sind beides Menschen, die mittels einem Handy auf diese Weise kommunizieren können.

Was ist SMS?

Aber was genau ist eine SMS (Short Message Service)? Für den Benutzer ist diese SMS genau das oben erwähnte, eine Kurzmitteilung, die er in seinem Handy lesen kann. Und viel mehr ist es auch gar nicht, denn die meisten Informationen kann man dem Handy entnehmen. Man sieht als Anhang noch den Absender der Kurzmitteilung und den Zeitpunkt, an dem sie abgeschickt worden ist. Also ist eine SMS eine Ansammlung von Daten, die halt nicht über eine serielle Schnittstelle oder ein Netzwerk versendet werden, sondern über ein Mobilfunknetz.

Die Aufgabe des „Senders“ ist es, die zu übertragenden Daten richtig zu formatieren und an eine Art Zentrale zu senden. Diese Zentrale wertet die SMS aus und leitet diese an den eigentlichen Empfänger weiter.

Die Aufgabe des „Empfängers“ ist es diese Daten zu empfangen und auszuwerten. Sobald eine neue SMS eingetroffen ist, muss dieses signalisiert werden.

Die beiden Aufgaben des „Senders“ und des „Empfängers“ lassen sich ohne viel Aufwand vom Tiger übernehmen. Der Vorteil dieser Art der Kommunikation ist zweifelsfrei die Einfachheit an fast jeder Stelle der Welt problemlos Daten zu versenden.

Anwendungen & Voraussetzungen

Einsatzgebiete für eine SMS Steuerung liegen z.B. in einer Art Alarmanlage, die bei bestimmten Zuständen eine SMS an bestimmte Nummern versendet. Aber auch an eine Fernsteuerung via SMS ist zu denken. Mit bestimmten Kommandos in einer SMS könnte man über eine beliebige Distanz steuern. Die Möglichkeiten auf diesem Gebiet sind sehr weiträumig.

Die Voraussetzung für solch eine Applikation ist ein datenfähiges Handy, welches SMS versenden kann oder ein SMS/GRPS Modem. Sollten Sie ein Modem benutzen, dann können Sie dieses mit einem Nullmodemkabel direkt an der seriellen Schnittstelle des Tigers anschließen. Die billigere Alternative ist natürlich das Handy. Wenn sie weitere Informationen über die Kopplung von Tiger und Handy erhalten wollen, dann lesen sie bitte den Applikationsbericht Nummer 56 – „BASIC-Tiger sendet SMS“. Dort wird dies ausführlich erläutert. Dieser kann unter „www.wilke.de“ downgeladen werden.

Die Funktionen von sms_Vxxx.inc:

SMS für das Senden codieren

- sendSMS: Diese Funktion ist zum einfachen senden einer SMS an eine bestimmte Nummer. Die Funktion hat 2 Parameter:
  - N$: Dieser erste Parameter ist die Nummer, an welche die SMS geschickt werden soll. Diese muss in einem String übergeben werden in der Form: „+491631234567“. Also zuerst die Länderkennung, in unserem Fall die „+49“ für Deutschland. Danach folgt die Handynummer ohne die führende „0“.
  - T$: Dieser String enthält den zu sendenden Text. Der Text wird genau so in der sms angezeigt. Die einzige Einschränkung ist, dass dieser Text nicht länger als 160 Zeichen sein darf, da eine eine SMS standardgemäß auf 160 Zeichen beschränkt ist.

  Nach dem Senden einer SMS ist es zudem sinnvoll, die Antwort des Handys/ Modems zu überprüfen, ob die SMS auch erfolgreich abgeschickt worden ist. Dabei muss man warten, bis sich der Puffer der seriellen Schnittstelle gefüllt hat, da der Tiger und das Handy über diese kommunizieren. Man liest den Puffer aus und vergleicht den Inhalt mit den verschiedenen Möglichkeiten. Dabei kann es zu folgenden Ereignissen kommen:
  - +CMGS:‹mr›[,ackpdu›] OK --› Erfolgreich gesendet
  - +CMS ERROR:‹err› --› Ein Fehler ist aufgetreten

  Parameter:
  - ‹mr›: GSM 03.40 TP-Message-Reference in Integer Format
  - ‹ackpdu›: User Data Element vom PDU String
  - ‹err›: Der aufgetretene Fehler

eingelesene SMS decodieren

- computeRcvdPdu: Diese Funktion ist zum Entschlüsseln einer SMS in PDU Format. Die verschiedenen Informationen werden heraus gefiltert und in verschiedenen Variablen gespeichert. Wenn also eine SMS empfangen wurde, sollte man diese Funktion aufrufen, um die richtigen Daten zu filtern.

  Parameter:
  - string$: Übergeben Sie für diesen Parameter einfach den empfangenen PDU String ohne irgenwelche Änderungen.
  - return$: Diese Variable ist der Rückgabewert der Funktion. Nach dem Beenden der Funktion wird hier die SMS im Textmodus gespeichert, also genau der Text, den sie auch im Handy lesen können. Dieser String ist selbstverständlich wieder auf 160 Zeichen beschränkt.

  Die restlichen Daten werden in globalen Variablen gespeichert:
  - rcvdSMSC$: Empfangene Nummer vom Service Center
  - rcvdSenderNum$: Absender
  - rcvdPduTxt$: Erhaltene SMS im Textmodus

Außerdem gibt es noch Funktionen, mit denen sie diese Variablen auslesen können:

- getRcvdPduSMSC: Lese die Nummer vom Service Center aus
- getRcvdSenderNumber: Lese den Absender aus
- getRcvdePduTxt: Lese den enthaltenen Text der SMS aus

In dieser Funktion werden die wichtigsten Informationen ausgelesen, doch nicht alle. Falls die weiteren Daten benötigt werden, kann die Funktion beliebig erweitert werden. Die Stellen, an der Code eingefügt werden kann, sind bereits angelegt. Sie müssen lediglich die Stelle im Code suchen, an der die Informationen gelöscht werden. Im Kommentar wird genau gesagt, welche Daten dort gelöscht werden.

Wie arbeiten die SMS-Funktionen:

Funktionsweise der Funktionen

Die Funkionsweise der beiden Funktionen ist relativ kompliziert, da die Kodierung im PDU Modus nicht einfach ist. Wer sich trotzdem damit beschäftigen will und die Funktionen verstehen will, kann an dieser Stelle weiterlesen. Um einen Anfang zu machen, muss man sich zunächst genaustens mit der Codierung beschäftigen. Was genau ist der PDU Modus? Eigentlich ist er nur eine Codierung wie z.B. der ASCII Code auch.

Eine PDU Nachricht hat immer den gleichen Aufbau. Sie könnte z.B. so ausehen:
07919471227230332500 0C 919471123254760000 0B D4F29C4E2FE3E9BA4D19

Im folgenden werden wir auf die einzelnen Elemente eingehen, um ihnen zu zeigen, wie sie mit solch einem String umzugehen haben.

07:
Das ist die Länge der gelieferten Adressdaten (also die Länge der SMSC-Adresse) in Bytes.

91947122723033:
Die Adressdaten. Diesen Teil müssen wir nun nochmals aufteilen:
1. Byte: 91
Hier gibt es nur 2 Möglichkeiten:
91: für Nummern im internationalen Format
81: für Nummern im nationalen Format mit Ortsvorwahl oder Nummern ohne Vorwahl.
ab 2. Byte: 947122723033
Die restlichen Bytes enthalten die Nummer des SMSC, das verwendet werden soll, und zwar BCD-kodiert, also je 4 Bit ergeben eine Ziffer der Nummer. Die Nummer beginnt im zweiten Halbbyte des ersten Bytes, ein eventuell nicht benutztes Halbbyte an der letzten Stelle muß unter allen Umständen 0xF enthalten.

Bits 7 6 5 4	Bits 3 2 1 0
zweite Ziffer	erste Ziffer
vierte Ziffer	dritte Ziffer
sechste Ziffer	fünfte Ziffer
...	...
1 1 1 1	letzte Ziffer

Die +49172123456 würde also so dargestellt:07919471123254F6, die 1212 so: 03812121

Wichtig: Sollte im Telefon schon eine SMSC Nummer konfiguriert sein, so kann man diese beim Senden der SMS weglassen. In diesem Falle gibt man einfach zu Beginn eine 00 an, die Nummer hat dann also die Länge 0. Das Handy interpretiert das dann richtig und ersetzt diese durch die Standardnummer.

25:

In diesem Byte sind gleich 6 Parameter untergebracht:
Bit 0 + 1: Message Type Indication: 01 für 'SMS-SUBMIT MS to SMSC'
Bit 2: Reject Duplicates: 0 für 'aus', 1 für 'ein'
Bit 3 + 4: Validity Period: 00 für kein VP-Feld vorhanden
Bit 5: Status Report Request: 0 für 'aus', 1 für 'ein'
Bit 6: User Data Header Ind.: 0 für 'no UDH'
Bit 7: Reply Path: 0 für 'aus', 1 für 'ein'

00:

Referenznummer der Nachricht. Wenn keine angegeben wird, also 00, dann vergibt das Handy/Modem die Nummer selber. Zustellbestätigungen oder Fehlermeldungen enthalten diese Referenznummer und können damit der ursächlichen Nachricht zugeordnet werden.

0C:

Dies ist die Länge der Zieltelefonnummer, also die Nummer, an die die SMS geschickt wird (uncodiert, in Ziffern!). In der Länge ist allerdings noch nicht der Ton enthalten.

91947112325476:

Das ist die Telefonnummer, welche wiederum genau so kodiert ist wie auch schon die SMSC Nummer, die Nummern werden also wieder paarweise vertauscht.

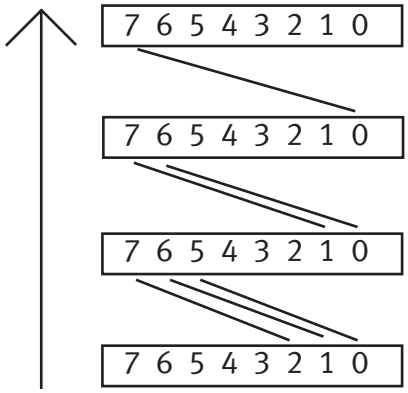
00:

Protocol Identifier: Die Nummer identifiziert das Protokoll. Hier kann man getrost immer 00 angeben, welches der Standardwert ist.

00:
Data Coding Scheme: Das Byte sagt dem Telefon, wie er die Daten zu decodieren hat, wobei man hier auch wieder immer 00 angeben kann.

0B:
Das ist das Längenbyte der Benutzerdaten, also die Länge der SMS, oder um genau zu sein, die Länge des Textes der SMS. Diese Länge bezieht sich auf die Anzahl der Zeichen des Textes nach der Decodierung.

D4F29C4E2FE3E9BA4D19:
Dies ist der eigentliche Text der SMS im PDU Modus codiert. Der eigentlich in 7 Bit codierte Text wird nun in ein 8 Bit Format geändert, wodurch man bei längeren Texten einige Byte einsparen kann. Bei 160 Zeichen werden 20 Byte gespart.
Die Bits werden nach dem unten stehenden Schema angeordnet. Die niedrigsten Bits eines Zeichens werden an die höchsten freien Stellen des vorigen Bytes gestellt/verschoben. Dadurch werden natürlich bei diesem Byte wieder Stellen frei, aber an den ersten Positionen. Also werden die höherwertigen Bits nach rechts verschoben, um an den oberen Positionen Platz für die nächsten Bits zu machen.



usw.

Wenn man das System einmal verstanden hat, sollte man auch den Algorithmus nachvollziehen können.

Beim Decodieren der PDU Nachricht geht man jede einzelne Information durch und analysiert diese. Die Funktion „decodeTelNum“ kann somit die Telefonnummer und die SMSC entschlüsseln, indem sie immer die Einträge paarweise vertauscht und die Nummern die sich dabei ergeben speichert.

In der Funktion „smsPduToTxt“ wird nun der Text in wirklichen Textmodus geändert, indem die Bits an die richtige Position gestellt werden, bzw. die 7 Bit Zeichen wieder hergestellt werden. Zunächst werden 2 Bytes zu einem konvertiert. Der Tiger empfängt z.B. CD = 1 x C & 1 x D, wir brauchen aber ein Byte mit dem Wert 0xCD. Wir erstellen also einen String, der genau so aufgebaut wird. Jedes Byte muss gespiegelt werden, um die Bits in die richtige Reihenfolge zu bringen. Die verschiedenen Bits werden jetzt angeordnet, bzw. ausgelesen. Die Werte von den eigentlichen 7 Bit Zeichen werden in einem Array von Bytes gespeichert. Dieser Array wird dann später in einen String umgewandelt, indem jedes Byte das zugehörige Zeichen zugordnet bekommt.

In der Funktion „sendSMS“ wird der zu sendende Text genau so codiert, wie es oben erwähnt wurde. Die 7 Bit ASCII Zeichen werden in einer Zeichenkette gespeichert, wo die Bits nach diesem Schema angeordnet sind. Dazu wird der ursprüngliche Text zunächst binär in einem String gespeichert. Wir benutzen nun einen großen String, wo ein Byte den Wert eines Bits annimmt, also entweder 1 oder 0 ist. Dadurch kann man durch den Index schnell darauf zugreifen. Man speichert in diesem String den Text, allerdings im 7 Bit Format, Bit 8 lassen wir aus, wobei dies sowieso immer 0 ist. Nach dem Umwandeln des Textes in PDU Format wird die Telefonnummer bearbeitet, wobei die Zahlen getauscht werden und am Ende evt. ein „F“ aufgefüllt wird, falls das letzte Halbbyte alleine steht. Wenn ein „+“ in der Telefonnummer vorkommt, dann setzen wir eine 19 an die entsprechende Stelle, ansonsten eine 18. Die 19 wird zur 91 nach Decodierung und heißt, dass internationale Nummern gebraucht werden, andernfalls nicht.

Beispiele

Es gibt zwei Beispielprogramme, die diese Funktionen nutzen um eine SMS per Handy zu übertragen:

„SendSMS.tig“ sendet einen Text an eine Zielrufnummer, wobei sowohl der Text wie auch die Rufnummer des Empfängers als Konstanten im Header des Programms definert sind.

„ReceiveSMS.tig“ liest die erste auf Ihrem Handy gespeicherte SMS. Sollte keine SMS gespeichert sein, könnte das Programm z.B. eine Fehlermeldung ausgeben. Der entsprechende Zweig ist per Default leer, d.h. es passiert einfach gar nichts.

leere Seite

Stichwortregister

leere Seite

Stichwortverzeichnis

Alphabetischer Index

N

P

S

T

U

W

X

Stichwortregister

leere Seite

Notizen

Notizen