

# **Tiger Graphic Library V1.14**

Blank Page

# Index

Tiger Graphic Library V1.14	1
Index	3
Introduction	9
Tiger Graphic Library Step by Step	12
Create Buttons on the Touch Panel	14
Using Graphic Fonts on the LCD	18
Realize a Graphical User Interface	22
Multitasking Programming and Displaying Values	28
Changing of Windows	35
Application Example	42
Components and its Handling	59
Elements	59
Windows	60
Creating and Placing Elements	61
Docking	62
Integrated Applications	63
Graphic Fonts	64
User Graphics	65
Graphical Functions	66
Tips and Tricks	68
Configuration	69
General Settings	70
Master Defines	71
Memory for Elements	72
Memory for Windows	73
Memory for Graphic Fonts	74
Choice of Graphic Fonts	76
Choice of Keyboards	77
Choice of RTC Applications	78
Hardware	79
Versions	82
General Subroutines	83
vTglInit	85
bTglLink	86
bTglDeleteElement	88

## Index

bTglDeleteElementFromWindow	90
bTglShowWindow	92
bTglHideWindow	93
bTglShow	94
bTglHide	95
bTglShowText	98
bTglShowLong, bTglShowWord, bTglShowByte	99
bTglShowReal	103
bTglShowGraph	105
vTglUpdate	109
vTglUpdateParams	109
bTglSetSize	110
wTglGetSize	110
bTglSetAddress	111
lTglGetAddress	111
bTglSetText	112
bTglSetFont	114
bTglSetFrame	115
bTglSetMargins	116
wTglGetMargins	116
bTglSetCoordinates	117
wTglGetCoordinates	117
bTglSetAttribute	119
lTglGetAttribute	123
vTglSetStandbyTime	125
lTglGetStandbyTime	126
bTglSetStandbyTermination	128
bTglSwitchStandby	131
vTglWaitTouchTp	133
vTglWaitReleaseTp	133
bTglGetTouch	133
wTglGetTouchedElement	134
wTglGetNumTouchedElements	135
bTglGetTouchedElementsFlag	136
vTglBeep	137
<b>Graphic</b>	<b>138</b>
bTglCreateGraphic	139
bTglPlaceGraphicInWindow	140
bTglDockGraphicInWindow	141
bTglCreateGraphicWnd	142
bTglCreateGraphicDockWnd	143

<b>Label</b>	<b>145</b>
bTglCreateLabel	146
bTglPlaceLabelInWindow	148
bTglDockLabelInWindow	149
bTglCreateLabelWnd	150
bTglCreateLabelDockWnd	152
<b>Buttons</b>	<b>155</b>
bTglCreateButton	157
bTglPlaceButtonInWindow	162
bTglDockButtonInWindow	165
bTglCreateButtonWnd	168
bTglCreateButtonDockWnd	172
bTglCreateButtonDockWnd	175
bTglGetPushButtonState	177
bTglGetButtonState	178
bTglSetButtonState	181
bTglGetKeycode	184
bTglWaitKeycode	185
<b>Text Button</b>	<b>186</b>
bTglCreateTextButton	188
bTglPlaceTextButtonInWindow	190
bTglDockTextButtonInWindow	191
bTglCreateTextButtonWnd	192
bTglCreateTextButtonDockWnd	194
<b>Slider</b>	<b>198</b>
bTglCreateSlider	202
bTglPlaceSliderInWindow	204
bTglDockSliderInWindow	207
bTglCreateSliderWnd	211
bTglCreateSliderDockWnd	212
bTglSetSliderValue	216
lTglGetSliderValue	217
bTglSetSliderLimits	220
<b>Listbox</b>	<b>221</b>
bTglCreateListbox	222
bTglPlaceListboxInWindow	223
bTglDockListboxInWindow	224
bTglCreateListboxWnd	225

## Index

bTglCreateListboxDockWnd	226
sTglGetListboxItem	229
wTglGetListboxIndex	230
bTglSetListboxIndex	230
Gauge	233
bTglCreateGauge	234
bTglPlaceGaugeInWindow	235
bTglDockGaugeInWindow	236
bTglCreateGaugeWnd	237
bTglCreateGaugeDockWnd	239
bTglShowGaugeValue	242
Keyboard	245
bTglInitKeyboard	246
bTglInitKeyboardSelfmade	251
sTglGetKeyboardInput	257
sTglGetKeybInputTimeout	259
RTC Applications	260
bTglInitRtc	261
bTglSetRtc	263
Text Graphics	265
Choosing the Graphic Fonts	267
Specific Parameters of Graphic Fonts	268
Designing Graphic Texts	274
Codes for Normal and Special Chars	276
Codes for Control Chars	279
Graphic Fonts Solo	281
Including Graphic Fonts	282
Hardware Configuration for the LCD	283
bTglCreateFont	285
bTglCreateFontParams	287
bTglSetFontParams	289
bTglGetFontParams	291
sTglBuildTextGraphic	292
lTglCalcTextToWindow	294
lTglGetLineHeight	295
lTglCalcTextGraphicWidth	296
User Graphic	297

## Index

vTglShowUserGraphic	302
vTglShowUserGraphicParams	303
vTglHideUserGraphic	304
sTglGetWindowGraphic	305
vTglPutWindowGraphic	306
vTglClearWindowGraphic	307
vTglPutStringToLcd	308
vTglPutStringToLcdParams	309
vTglPutFlashToLcd	310
Graphical Functions	311
sTglDrawGraph	312
Touch Panel	314
Read out Touch Panel Keyboard Buffer	315
Auto Repeat	316
Templates	317
Demo Menu	318
Error Codes	323
Overview of Example Programs	328
Overview of applications	334
Overview of Include Files	335
Documentation History	337

.....

**Index**

.....

Blank Page

.....



# Introduction

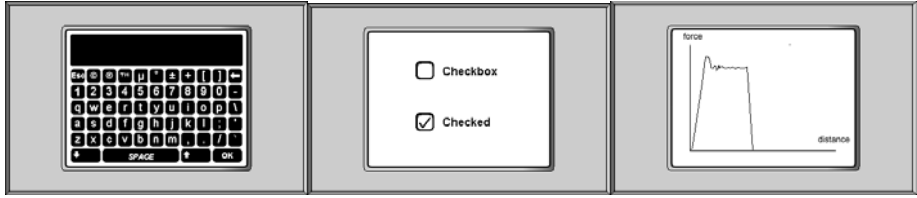
The Tiger Graphic Library simplifies programming the touch panel. The controlling of the devices is already easy to program with the comfortable use of the device drivers for each device with many features. With the Tiger Graphic Library there is a tool to realize a graphical user interface by calling just a few subroutines. In earlier times the part of programming the GUI mostly would take the longest time of the software development.



Now with the Tiger Graphic Library you are able to realize your project in a minimum of time and a maximum of user friendliness. Its great features will make it easy for you to give your program an individual nice looking without needs for a longer development time. You will find lots of example programs for the use of its subroutines and many applications which can be used as templates for your own programs.



## Introduction



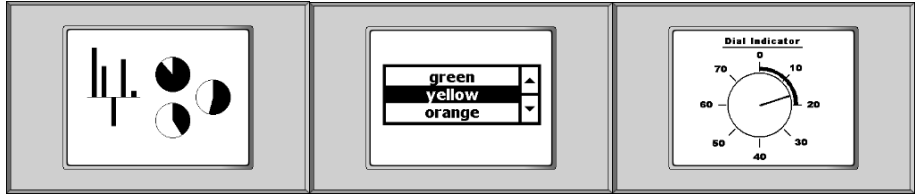
The Tiger Graphic Library provides subroutines for:

- creating buttons, switches and sliders for the touch panel
- listboxes for easy item selections
- graphic fonts with many special chars
- administrating whole pages containing touch panel functionalities and graphical elements for the LCD
- touch panel keyboards in different styles
- setting and displaying the real time clock in different styles
- displaying current measurands as
- creating gauges, pie charts and bar charts
- creating graphs
- showing dynamically created graphics of your own
- stand-by functions for saving energie
- marking elements by inversion, alternative graphic or blinking



Additionally the Tiger Graphic Library

- provides many templates for own projects
  - fully multitasking ability
  - will be updated continuously
- ➔ PLEASE LET US KNOW YOUR SUGGESTIONS!  
support@wilke.de



To get a first impression of the features of the Tiger Graphic Library please download the one of the demo programs on your TP1000 e.g.:

*C:\Programme\Wilke Technology\Tiger Basic 5.3\Applications\TP1000\_Demo\TP1000\_Demo.tig*

For your first program using the Tiger Graphic Library and the graphic fonts we would suppose to start with the chapter *Tiger Graphic Library Step by Step*.

For more information about the components of the Tiger Graphic Library and its handling see chapter *Components and its Handling*.

You will find all the printed sample programs and many more examples in your installation directory

*C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples.*

# Tiger Graphic Library Step by Step

The Tiger Graphic Library is an object orientated assembly of subroutines. The objects in the Tiger Graphic Library are called elements which can be shown on LCD occasionally including the touch panel functionality. The simplest element is a graphic. This is a bitmap of a certain size placed on the LCD. More complex elements are e.g. sliders or buttons. These elements additionally contain touch panel functions. You have the choice of using completely designed elements of the Tiger Graphic Library or of making your own design by creating new bitmaps for your elements or by creating graphic texts by using the graphic fonts.

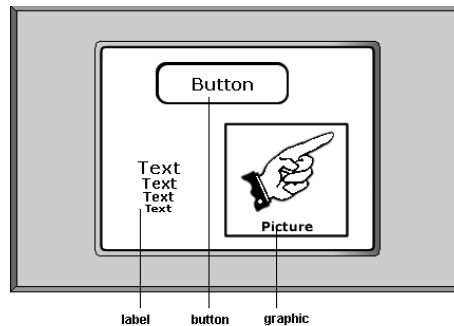


figure 1: Elements of the Tiger Graphic Library

For easy handling of that what is shown on the LCD, the elements are assembled in windows. In projects, mostly there is not just one window displayed constantly on LCD. You have to switch windows while you are navigating in a menu or update your LCD for getting user inputs or displaying changing information.

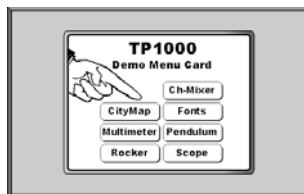


figure 2 Menu



figure 3: Graphical user interface

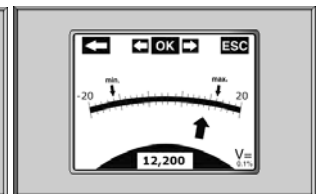


figure 4 Changing information

The Tiger Graphic Library provides both switching whole windows and switching single elements.

With the graphic fonts you can choose from many bitmap fonts of various sizes and types. The Tiger Graphic Library provides a tool for using these fonts as easy as every other font. Using graphic fonts instead of LCD fonts would give your application a good looking and an individual touch.



figure 5: Graphic fonts

If you want to work with the Tiger Graphic Library for the first time we suggest reading this chapter carefully. You will see how easy it will be to create a button or other elements and display them on the LCD. We suggest copying the code fragments directly into your future projects.

When using the Tiger Graphic Library please follow these basic steps:

1. Choose your LCD type
2. Include the files for the Tiger Graphic Library
3. Determine the numbers of your identifiers of windows and fonts
4. Declare variables especially for those elements to work with later in the program
5. Install the device drivers for the touch panel and the LCD
6. Initialize the Tiger Graphic Library
7. Create the elements and place them in windows
8. Display the window with its elements on the LCD
9. Save the bitmaps and texts for the elements in the flash memory

All code examples are written for the TP1000 hardware.

### Create Buttons on the Touch Panel

For getting started with the Tiger Graphic Library we will explain step by step how to create a button for the touch panel on the LCD. You will find this example in *C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_1\_button.TIG*

In the first step you choose your type of LCD. If you have a "white" LCD you need normal LCD output. Please set the value of the define LCD\_INVERSION\_MODE to 0. A "blue" LCD needs an inverted LCD output. Please set the value to 1. Do this always BEFORE the next step.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion  for "blue"  LCD
```

The second step is letting know the program that it should work with the Tiger Graphic Library. This is done by including the Tiger Graphic Library, right at the beginning of your program's source code.

```
#include TigerGraphicLibrary.INC
```

In the third step please determine the numbers for the identifiers of the windows in which the elements should be shown. Start with number 0 for the first window. You will need these identifiers when you place elements in or show it on the LCD.

```
' ' windows
#define WINDOW_ID              0
```

The bitmap for the graphic has to reside in flash memory. We do this in step 4 by declaring and setting the data label and naming the file with the bitmap.

```
datalabel dlButton ' flash pointer on bitmap of button
dlButton::
data filter "button.bmp", "GRAPHFLT", 0 ' WxH=76x33 BmpWidth=80
```

Normally the bitmap file must be saved in the directory where the tig file is saved. Otherwise there must be passed the path to the file with the data instruction. An alternative way for letting the compiler know the path to files is the preprocessor instruction *#project\_path*. Please see the programming manual for further information. The Tiger Graphic Library will find all files in the directories named *Bitmaps*, *Flash* or *Include*.

## Tiger Graphic Library Step by Step

In step 5 we declare the variables. Our standard variable *blReturn* for each tgl program is for the return value of the subroutines of the Tiger Graphic Library. With monitoring this return value you can control if the subroutine has been executed successfully or not. For the elements we suggest to create the identifiers dynamically by using an incremented variable *wlElementId*. An incremental creation of identifiers will protect you from giving the same identifier for more than one element.

```
byte blReturn          ' return value for TGL subroutines
word wlElementId       ' incremental Identifier
```

In step 6 we set the BASIC-Tiger™ ports and install the device drivers for the touch panel and the LCD. The included file *TGL\_DEVICE\_DRIVERS\_TP1000.INC* is for installing the device drivers of a TP1000. For details see code of the files *TGL\_DEVICE\_DRIVERS\_TP1000.INC* and *TigerGraphicLibraryConf.INC* in the directory *C:\Programme\Wilke Technology\Tiger Basic 5.3\TigerGraphicLibrary* and the data sheet for the TP1000.

```
#include TGL_DEVICE_DRIVERS_TP1000.INC
```

In step 7 we initialize the Tiger Graphic Library. Do this before calling any subroutine of the Tiger Graphic Library. Additionally we initialize the incremental identifier for the elements.

```
call vTglInit()
wlElementId = 0
```

After these steps you can call all the subroutines of the Tiger Graphic Library. Just mind the order of creating, placing and working with the elements.

CREATING elements means defining a type, giving a size and – depending from the type – further attributes. These attributes will be always the same wherever you will place the element.

PLACING means to determin the coordinates of the element on the LCD. As you will have more than one page on the LCD you assemble some elements for one page in a WINDOW for further handling. With the placing you determine further attributes for the element which can be different in each window you place the element in.

It is possible to create and place elements in a window in one go as we do it in the seventh step.

## Tiger Graphic Library Step by Step

```
call bTglCreateButtonWnd( &
76, 33, &           ' width, height of element
dlButton, 80, &      ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, & ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, & ' identifier of element, window
122, 100, &         ' x, y coordinate on LCD
0h, &              ' keycode
blReturn )          ' return code (0: OK exit >0: error exit)
```

In step 9 we display the button on the LCD. The touch panel functionality will be activated automatically. For changing a whole page use the following code by using the identifier of the window. For showing and hiding single elements in one window see the subroutines *bTglShow* and *bTglHide*.

```
call bTglShowWindow( WINDOW_1_ID, blReturn )
```

Now you are prepared to compose your "hello world" program for the Tiger Graphic Library.

Sample program:

```
'-----
' TGL_Lesson_1_button.TIG
'-----
'
' required software:   Tiger Graphic Library V1.01
' required hardware:  TP-1000
'-----
'*****
'      Step 1: Choose your LCD type
'*****
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion  for "blue"  LCD

'*****
'      Step 2: Include the files for the Tiger Graphic Library
'*****
#include TigerGraphicLibrary.INC

'*****
'      Step 3: Determine the numbers of your identifiers
'*****
' windows
#define WINDOW_ID              0

task main
'*****
'      Step 4: Declare your variables
'*****
byte blReturn                  ' return value for TGL subroutines
word wElementId                ' incremental Identifier for elements
```



```

*****
'      Step 5: Save the bitmap for the button in the flash memory
*****
datalabel dlButton          ' flash pointer on saved bitmap of button
dlButton::
data filter "button.bmp", "GRAPHFLT", 0          ' WxH=76x33 BmpWidth=80

*****
'      Step 6: Install the device drivers for the touch panel and the LCD
*****
#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
'      Step 7: Initialize the Tiger Graphic Library
*****
call vTglInit()
wlElementId = 0

*****
'      Step 8: Create the button and place it on the LCD
*****
call bTglCreateButtonWnd( &
76, 33, &          ' width, height of element
dlButton, 80, &    ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, & ' key attributes auto repeat, beep, type
wlElementId, WINDOW_ID, & ' identifier of element, window
122, 100, &        ' x, y coordinate on LCD
0h, &              ' keycode
blReturn )          ' return code (0: OK exit >0: error exit)

*****
'      Step 9: Display the button on the LCD
*****
call bTglShowWindow( WINDOW_ID, blReturn )
end

```

### Using Graphic Fonts on the LCD

In this chapter we will demonstrate step by step how to get an easy start working with the graphic fonts. The explanations in this chapter assume the knowledge of the chapter before. We suggest copying the code fragments directly into your future projects. You will find this example in

*C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_2\_label.TIG*

If you use the graphic fonts you need to make your own configuration of the Tiger Graphic Library. You do this by saving a COPY of the file

*C:\Programme\Wilke Technology\Tiger Basic 5.3\TigerGraphicLibrary\TigerGraphicLibraryConf.INC* in the same directory as for the file of the *.TIG*-file of your project. Mind the activated code lines for the font we will use in this example.

```
#define VALENCIA_21_BOLD
#include TGL_GRAFO_VALENCIA.INC
```

If you would activate fonts you do not use in your program, you would waste a lot flash memory. So mind which fonts you really need.

! If you configure the Tiger Graphic Library for your project, NEVER change the original configuration file in the directory of the Tiger Graphic Library. This file with its standard configuration runs with all the examples of the Tiger Graphic Library. Normally the standard configuration would work with little programs you will write, too.

Now we are prepared to follow the steps completely analog to the steps of the example for the button in the chapter before.

The steps 1 and 2 are equals the steps of the chapter before. Choose the LCD type and include the Tiger Graphic Library.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC
```

In the step 3 you define in addition to the identifiers of the windows the identifiers for the used graphic fonts. The numbers for the fonts have their own enumeration and start with 0.

```
' ' windows
#define WINDOW_ID      0
' ' fonts
#define FONT_ID        0
```

In step 4 we reside the text and its length for the label in the flash memory. We do this by using the instruction *data string*.

```
datalabel dlLabelText ' flash pointer on saved label text and its length
dlLabelText::
data string "Hello label!"
```

If you have longer texts you can reside the text and its length by the instruction *data file* as we did it with the bitmap for the button in the chapter before. Going this way please mind that the file has to start with a 4 byte value for the length of the text (low byte first). In case of using Tiger 1 the file has to start with a 2 byte value for the text length.

Residing the text in the flash is the most efficient way of passing texts to labels. An alternative way of creating labels is passing the text by a variable or a constant. For details see the description of the subroutine *bTglCreateLabel*.

Step 5 is for the declaration of the variables. We use the same variables as in the chapter before.

```
byte blReturn          ' return value for TGL subroutines
word wlElementId       ' incremental identifier
```

In the steps 6 and 7 the device drivers for the TP1000 will be installed and the Tiger Graphic Library will be initialized. The incremental Identifier starts with 0.

```
#include TGL_DEVICE_DRIVERS_TP1000.INC
call vTglInit()
wlElementId = 0
```

In the step 8 we need to create a font and a label. We need the font for the creation of the label by passing its identifier. The font does never need to be placed. The label is created and placed in one go.

```
call bTglCreateFontParams( &
FONT_ID, &                ' identifier of font
"Valencia", 21, "bold", &  ' name, size, type of font
"center", "center", &     ' alignment horizontal, vertical
"prop", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
```

## Tiger Graphic Library Step by Step

```
"imm", "char", &          ' overlay, wrap mode
bIReturn )                ' return code (0: OK exit  >0: error exit)

call bTglCreateLabelFWnd( &
200, 60, &                ' width, height of element
dILabelText, &            ' flash address of text of element
FONT_ID, 8, &             ' font identifier, frame thickness
wIElementId, WINDOW_ID, & ' identifier of element, window
60, 90, &                 ' x, y coordinate on LCD
bIReturn )                ' return code (0: OK exit  >0: error exit)
```

In step 9 we display the label with the graphic font on the LCD.

```
call bTglShowWindow( WINDOW_ID, bIReturn )
```

Sample program:

```
-----
' TGL_Lesson_2_label.INC
-----

' required software:   Tiger Graphic Library
' required hardware:  TP-1000
-----

*****
'      Step 1: Choose your LCD type
*****
#define LCD_INVERSION_MODE    0      ' 0 = normal    for "white" LCD
                                  ' 1 = inversion  for "blue"  LCD

*****
'      Step 2: Include the files for the Tiger Graphic Library
*****
#include TigerGraphicLibrary.INC

*****
'      Step 3: Determine the numbers of your identifiers
*****
' windows
#define WINDOW_ID            0
' fonts
#define FONT_ID              0
task main

' *****
'      Step 4: Save the label text and its length in the flash memory
' *****
datalabel dILabelText ' flash pointer on saved label text and its length
dILabelText::
data string "Hello label!"

' *****
'      Step 5: Declare your variables
' *****
```

```

byte blReturn          ' return value for TGL subroutines
word wlElementId       ' incremental identifier

'*****
'      Step 6: Install the device drivers for the touch panel and the LCD
'*****
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'      Step 7: Initialize the Tiger Graphic Library
'*****
call vTglInit()
wlElementId = 0

'*****
'      Step 8: Create the font and the label and place the label on LCD
'*****
call bTglCreateFontParams( &
FONT_ID, &                ' identifier of font
"Valencia", 21, "bold", &  ' name, size, type of font
"center", "center", &     ' alignment horizontal, vertical
"prop", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
"imm", "char", &         ' overlay, wrap mode
blReturn )                ' return code (0: OK exit >0: error exit)

call bTglCreateLabelFWnd( &
200, 60, &                ' width, height of element
dlLabelText, &            ' flash address of text of element
FONT_ID, 8, &             ' font identifier, frame thickness
wlElementId, WINDOW_ID, & ' identifier of element, window
60, 90, &                 ' x, y coordinate on LCD
blReturn )                ' return code (0: OK exit >0: error exit)

'*****
'      Step 9: Display the label on the LCD
'*****
call bTglShowWindow( WINDOW_ID, blReturn )
end

```

# Realize a Graphical User Interface

In this chapter we will demonstrate step by step how realize a graphical user interface (gui). We suggest copying the code fragments directly into your future projects. You will find this example in  
*C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_2\_label.TIG*

In Step 1 and 2 the inversion mode of the LCD will be set and the Tiger Graphic Library will be included.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC
```

In Step 3 the Identifiers for fonts and windows will be defined. Additionally we declare 2 variables for identifiers of those elements we will need to pass to subroutines in the further program code.

```
' windows
#define WINDOW_ID_name          0
#define WINDOW_ID_keyboard      1
' fonts
#define FONT_ID                  0
' elements
word wgNameId
word wgKeyboardId
#include TigerGraphicLibrary.INC
```

In the step 4 we define keycodes as return codes for the touch panel buttons. With this parameter passed with the placement of buttons we can decide which of the buttons in a window has been pressed. In this example only one button will be placed.

```
' keycodes
#define KEY_name                0
```

Steps 5 to 7 are for the declaration of the variables, the installation of the device drivers and the initialization of variables and the Tiger Graphic Library. Mind the setting of a higher task priority while the initialization. For a fast task switching, the priority should be reset to 1 when the program enter its main loop.

```
byte blReturn      ' return value for TGL subroutines
word wlElementId    ' incremental identifier for elements
word wlWindowId     ' identifier for windows
byte ever          ' endless loop
```

```

byte blKeycode      ' return code of button
string slName$(40h) ' user input

#include TGL_DEVICE_DRIVERS_TP1000.INC

set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wElementId = 0
set_len$( slName$, 0 )

...

set_task_prio main, 1 ' reset task priority for usual running

```

In step 8 the windows for the input demand will be built. Mind the saving of the current value of the incremental identifier for the element in *wgNameId* for later use before its creation and the following incrementation.

```

call bTglCreateFontParams( &
FONT_ID, &                ' identifier of font
"Valencia", 21, "bold", & ' name, size, type of font
"left", "top", &          ' alignment horizontal, vertical
"prop", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
"imm", "char", &          ' overlay, wrap mode
blReturn )                ' return code (0: OK exit >0: error exit)

call bTglCreateLabelWnd( &
190, 60, &                ' width, height of element
"name:", &                ' text of element
FONT_ID, 0, &              ' font identifier, frame thickness
wElementId, WINDOW_ID_name,& ' identifier of element, window
65, 60, &                 ' x, y coordinate on LCD
blReturn )                ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wElementId = wElementId + 1

' save this identifier for later use
wgNameId = wElementId
call bTglCreateTextButtonVarWnd( &
200, 60, &                ' width, height of element
FONT_ID, 4, &             ' font identifier, frame thickness
TGL_KEY_ATTR_AUTOREPEAT_OFF,& ' key attribute
wElementId, WINDOW_ID_name,& ' identifier of element, window
60, 120, &               ' x, y coordinate on LCD
KEY_name, &               ' keycode
blReturn )                ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wElementId = wElementId + 1

```

## Tiger Graphic Library Step by Step

In the step 9 a whole keyboard will be created with a single calling. Initialization subroutines in the Tiger Graphic Library create and place an assembly of elements in one or more windows. The identifiers will be incremented by the subroutine itself. The returned values will be the next free identifiers. That is the reason why these initialization subroutines will not accept constants as parameters for the identifiers for the windows and elements. The constant *WINDOW\_ID\_keyboard* must be parsed into a variable first before it can be passed to the initialization subroutine.

```
wlWindowId = WINDOW_ID_keyboard
call bTglInitKeyboard( &
TGL_KEYBl_STYLE_ENG, FONT_ID, & ' keyboard style, font
wElementId, wlWindowId, &      ' identifier of element, window
wgKeyboardId, &                ' identifier for keyboard view
blReturn )                    ' return code (0: OK exit >0: error exit)
```

If you debug the return value *wlWindowId* you will see, that the keyboard will need 2 identifiers for windows: one window for the unshifted and another window for the shifted keys.

In step 10 the window for the input demand will be shown.

```
call bTglShowWindow( WINDOW_ID, blReturn )
```

In step 11 the program waits until a button is pressed and arbitrates depending on the returned keycode which let you know which of the placed buttons has been pressed. In this example only one button has been placed.

```
call bTglWaitKeycode( blKeycode )
switchi blKeycode
case KEY_name:
'...'
endswitch
```

In step 12 a keyboard will be shown on LCD for the user input. The input will be returned with *s/Name\$*. All the functionality of getting the keyboard input is done by a single calling.

```
call sTglGetKeyboardInput( &
WINDOW_ID_keyboard, wgKeyboardId, & ' identifiers for keyboard
040h, slName$, &                    ' maximal user input length, user input
blReturn )                          ' return code (0: OK exit >0: error exit)
```

In step 13 the returned user input will be saved with the element. This way the user input will be shown with the element in the next loop after calling *bTglShowWindow*.



```
call bTglSetText( wgNameId, slName$, blReturn )
```

Sample program:

```
'-----
' TGL_Lesson_3_user_interface.TIG
'-----
' required software:   Tiger Graphic Library
' required hardware:  TP-1000
'-----

'*****
'      Step 1: Choose your LCD type
'*****
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue" LCD

'*****
'      Step 2: Include the files for the Tiger Graphic Library
'*****
#include TigerGraphicLibrary.INC

#define FONT_ID                0
' elements
word wgNameId
word wgKeyboardId

'*****
'      Step 4: Determine keycodes
'*****
' keycodes
#define KEY_name                0

task main

'*****
'      Step 5: Declare your variables
'*****
byte blReturn      ' return value for TGL subroutines
word wlElementId   ' incremental identifier for elements
word wlWindowId    ' identifier for windows
byte ever          ' endless loop
byte blKeycode     ' return code of button
string slName$(40h) ' user input

'*****
'      Step 6: Install the device drivers for the touch panel and the LCD
'*****
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'      Step 7: Initialize the Tiger Graphic Library
'*****
set_task_prio main, 16 ' speed up task for tgl initialization
call vTglInit()
wlElementId = 0
```

```

set_len$( slName$, 0 )

'*****
'      Step 8: Build a window with two elements
'*****
call bTglCreateFontParams( &
FONT_ID, &                                ' identifier of font
"Valencia", 21, "bold", &                ' name, size, type of font
"left", "top", &                          ' alignment horizontal, vertical
"prop", 0, &                              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, &                ' spacing char, vertical
"imm", "char", &                          ' overlay, wrap mode
blReturn )                                ' return code (0: OK exit >0: error exit)

call bTglCreateLabelWnd( &
190, 60, &                                ' width, height of element
"name:", &                                ' text of element
FONT_ID, 0, &                              ' font identifier, frame thickness
wlElementId, WINDOW_ID_name,&              ' identifier of element, window
65, 60, &                                  ' x, y coordinate on LCD
blReturn )                                ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wlElementId = wlElementId + 1

' save this identifier for later use
wgNameId = wlElementId
call bTglCreateTextButtonVarWnd( &
200, 60, &                                ' width, height of element
FONT_ID, 4, &                              ' font identifier, frame thickness
TGL_KEY_ATTR_AUTOREPEAT_OFF,&              ' key attribute
wlElementId, WINDOW_ID_name,&              ' identifier of element, window
60, 120, &                                ' x, y coordinate on LCD
KEY_name, &                                ' keycode
blReturn )                                ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wlElementId = wlElementId + 1

'*****
'      Step 9: Initialize a keyboard
'*****
wlWindowId = WINDOW_ID_keyboard
call bTglInitKeyboard( &
TGL_KEYBL_STYLE_ENG, FONT_ID, &          ' keyboard style, font
wlElementId, wlWindowId, &                ' identifier of element, window
wgKeyboardId, &                            ' identifier for keyboard view
blReturn )                                ' return code (0: OK exit >0: error exit)

set_task_prio main, 1 ' reset task priority for usual running
for ever = 0 to 0 step 0

'*****
'      Step 10: Display the window on the LCD
'*****
call bTglShowWindow( WINDOW_ID_name, blReturn )

'*****
'      Step 11: Wait for a pressed button

```

```
*****
call bTglWaitKeycode( blKeycode )
switchi blKeycode
case KEY_name:
*****
'      Step 3: Determine the numbers of your identifiers
*****
'' windows
#define WINDOW_ID_name          0
#define WINDOW_ID_keyboard      1
'' fonts

      *****
      '      Step 12: Get user input
      *****
      call sTglGetKeyboardInput( &
WINDOW_ID_keyboard, wgKeyboardId, &' identifiers for keyboard
040h, slName$, &          ' maximal user input length, user input
blReturn )          ' return code (0: OK exit >0: error exit)
endswitch

*****
'      Step 13: Save user input with element for later displaying
*****
call bTglSetText( wgNameId, slName$, blReturn )
next ' endless loop
end
```

# Multitasking Programming and Displaying Values

In this chapter we will demonstrate how to program in a multitasking mode. We will start a measuring task and display the measurands numerically and graphically.

The explanations in this chapter assume the knowledge of the chapter before. We suggest copying the code fragments directly into your future projects. You will find this example in

*C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_4\_tasks.TIG*

The steps 1 to 3 are equals the steps of the chapter before. Choose the LCD type, include the Tiger Graphic Library and determine the numbers for the identifiers.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC
' ' windows
#define WINDOW_ID_Measurand      0
' ' fonts
#define FONT_ID_text      0
#define FONT_ID_number    1
' ' elements
word wgValueId, wgBarId
```

The usual way to communicate between tasks in Tiger-BASIC™ is the use of global variables. Global variables must be declared outside of tasks or subroutines (step 4).

```
word wgMeasurand
```

In the steps 5 to 8 the lokal variables for the task main will be declared, the device drivers installed , the Tiger Graphic Library and the variables initialized and the windows be built. For acceleration we do his with a temporary higher task priority. Mind the parameter values “const” and *SPACING\_CHAR\_DEFAULT\_DIGIT* of the font for the numeric representation of the measurands. Choosing a constant spacing type here will avoid a flickering which would be seen on LCD with each change of the value.

```
byte ever      ' endless loop
byte blReturn  ' return value for TGL subroutines
word wlElementId ' incremental identifier for elements
word wlWindowId ' identifier for windows
string slName$(40h) ' user input
word wlMeasurand ' task input
long llValue     ' convert word to long
```

```
#include TGL_DEVICE_DRIVERS_TP1000.INC

set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wElementId = 0
set_len$( sName$, 0 )

call bTglCreateFontParams( &
FONT_ID_text, &           ' identifier of font
"Valencia", 21, "bold", & ' name, size, type of font
"center", "center", &    ' alignment horizontal, vertical
"prop", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
"imm", "char", &         ' overlay, wrap mode
b1Return )                ' return code (0: OK exit >0: error exit)

call bTglCreateFontParams( &
FONT_ID_number, &         ' identifier of font
"Valencia", 21, "bold", & ' name, size, type of font
"right", "center", &     ' alignment horizontal, vertical
"const", 0, &             ' spacing type, blank
SPACING_CHAR_DEFAULT_DIGIT,0,& ' spacing char, vertical
"imm", "char", &         ' overlay, wrap mode
b1Return )                ' return code (0: OK exit >0: error exit)

call bTglCreateLabelWnd( &
160, 60, &                ' width, height of element
"INPUT", &                ' flash address of text of element
FONT_ID_text, 0, &        ' font identifier, frame thickness
wElementId, WINDOW_ID_Measurand,& ' identifier of element, window
0, 0, &                  ' x, y coordinate on LCD
b1Return )                ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wElementId = wElementId + 1

call bTglCreateLabelVarWnd( &
140, 60, &                ' width, height of element
FONT_ID_number, 4, &      ' font identifier, frame thickness
wElementId, WINDOW_ID_Measurand,& ' identifier of element, window
10, 90, &                ' x, y coordinate on LCD
b1Return )                ' return code (0: OK exit >0: error exit)
' save this identifier
wgValueId = wElementId

' increment identifier for next element
wElementId = wElementId + 1

call bTglCreateGaugeWnd( &
40, 220, &                ' width, height of element
0,1023, &                ' limits of values
TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM,& ' type, location of base
4, &                    ' frame thickness
wElementId, WINDOW_ID_Measurand, & ' identifier of the element, window
220, 10, &              ' x, y coordinate on LCD
512, &                  ' start value for element
b1Return )                ' return code (0=Ok, >0=Error)
```

```
' save this identifier
wgBarId = wElementId

' increment identifier for next element
wElementId = wElementId + 1

set_task_prio main, 1 ' reset task priority for usual running
```

In step 9 we start the measuring task. For this example we generate random values secondly.

```
run_task tMeasure

task tMeasure
  byte ever
  randomize
  wait_next 1000
  for ever=0 to 0 step 0
    ' simulate 10-bit measurand
    wgMeasurand = (rnd(0)*1024) shr 16
    wait_next
  next
end
```

In step 10 we display the window on the LCD.

```
call bTglShowWindow( WINDOW_ID, blReturn )
```

In step 11 we update the LCD with each changing of the measurand. As the subroutines requires only the exact type of variable the measurand must be assigned to a long variable before passing (word to long conversion). To optimize the LCD update we pass the parameter value *TGL\_FALSE* to the show subroutines. This suppresses the LCD update and the element will be refreshed internally only. After refreshing all elements the whole LCD will be updated in one go.

```
' update LCD in case of changed measurand
if wMeasurand <> wgMeasurand then
  ' save new measurand
  wMeasurand = wgMeasurand
  ' convert word to long
  llValue = wgMeasurand
  ' update internally values first without updating the LCD
  call bTglShowLong( wgValueId, llValue, TGL_FALSE, blReturn )
  call bTglShowGaugeValue( wgBarId, llValue, TGL_FALSE, blReturn )
  ' update the whole LCD now in one go
  call vTglUpdate()
endif
```

To avoid later errors please mind this:

Never call a tgl subroutine when the task switching is disabled! Otherwise the program will hang in an endless loop. Internal tgl tasks would be disabled which garant a correct multitasking functionality.

Sample program:

```
'-----
' TGL_Lesson 4_tasks.TIG
'-----
' required software:    Tiger Graphic Library
' required hardware:    TP-1000
'-----

*****
'      Step 1: Choose your LCD type
*****
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue" LCD

*****
'      Step 2: Include the files for the Tiger Graphic Library
*****
#include TigerGraphicLibrary.INC

*****
'      Step 3: Determine the numbers of your identifiers
*****
' windows
#define WINDOW_ID_Measurand      0
' fonts
#define FONT_ID_text      0
#define FONT_ID_number    1
' elements
word wgValueId, wgBarId

*****
'      Step 4: Global variables
*****
word wgMeasurand

task main

' *****
'      Step 5: Declare your variables
*****
byte ever      ' endless loop
byte blReturn  ' return value for TGL subroutines
word wlElementId  ' incremental identifier for elements
word wlWindowId  ' identifier for windows
string slName$(40h) ' user input
word wlMeasurand  ' task input
long llValue      ' convert word to long

*****
'      Step 6: Install the device drivers for the touch panel and the LCD
*****
```

```
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'      Step 7: Initialize the Tiger Graphic Library
'*****
set_task_prio main, 16 ' speed up task for tgl initialization
call vTglInit()
wElementId = 0
set_len$( slName$, 0 )

'*****
'      Step 8: Build a window
'*****
call bTglCreateFontParams( &
FONT_ID_text, &          ' identifier of font
"Valencia", 21, "bold", & ' name, size, type of font
"center", "center", &    ' alignment horizontal, vertical
"prop", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
"imm", "char", &         ' overlay, wrap mode
bIReturn )               ' return code (0: OK exit >0: error exit)

call bTglCreateFontParams( &
FONT_ID_number, &        ' identifier of font
"Valencia", 21, "bold", & ' name, size, type of font
"right", "center", &     ' alignment horizontal, vertical
"const", 0, &             ' spacing type, blank
SPACING_CHAR_DEFAULT_DIGIT,0,& ' spacing char, vertical
"imm", "char", &         ' overlay, wrap mode
bIReturn )               ' return code (0: OK exit >0: error exit)

call bTglCreateLabelWnd( &
160, 60, &               ' width, height of element
"INPUT", &               ' flash address of text of element
FONT_ID_text, 0, &        ' font identifier, frame thickness
wElementId, WINDOW_ID_Measurand,& ' identifier of element, window
0, 0, &                  ' x, y coordinate on LCD
bIReturn )               ' return code (0: OK exit >0: error exit)

' increment identifier for next element
wElementId = wElementId + 1

call bTglCreateLabelVarWnd( &
140, 60, &               ' width, height of element
FONT_ID_number, 4, &      ' font identifier, frame thickness
wElementId, WINDOW_ID_Measurand,& ' identifier of element, window
10, 90, &                ' x, y coordinate on LCD
bIReturn )               ' return code (0: OK exit >0: error exit)
' save this identifier
wgValueId = wElementId

' increment identifier for next element
wElementId = wElementId + 1

call bTglCreateGaugeWnd( &
40, 220, &               ' width, height of element
0,1023, &                ' limits of values
TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM,& ' type, location of base
4, &                    ' frame thickness
```



```

wElementId, WINDOW_ID_Measurand, & ' identifier of the element, window
220, 10, & ' x, y coordinate on LCD
512, & ' start value for element
blReturn ) ' return code (0=Ok, >0=Error)
' save this identifier
wgBarId = wElementId

' increment identifier for next element
wElementId = wElementId + 1

set_task_prio main, 1 ' reset task priority for usual running

'*****
' Step 9: Start tasks
'*****
run_task tMeasure

'*****
' Step 10: Display the window on the LCD
'*****
call bTglShowWindow( WINDOW_ID_Measurand, blReturn )
' show measurand in the first loop
wlMeasurand = wgMeasurand + 1
for ever = 0 to 0 step 0

    '*****
    ' Step 11: Update LCD output
    '*****
    ' update LCD in case of changed measurand
    if wlMeasurand <> wgMeasurand then
        ' save new measurand
        wlMeasurand = wgMeasurand
        ' convert word to long
        llValue = wgMeasurand
        ' update internally values first without updating the LCD
        call bTglShowLong( wgValueId, llValue, TGL_FALSE, blReturn )
        call bTglShowGaugeValue( wgBarId, llValue, TGL_FALSE, blReturn )
        ' update the whole LCD now in one go
        call vTglUpdate()
    endif
next ' endless loop
end

'-----
' tMeasure:
'-----
' Simulate a measurand secondly
' which is normally get from device driver and following calculations
'-----

task tMeasure
byte ever
randomize
wait_next 1000
for ever=0 to 0 step 0
    ' simulate 10-bit measurand
    wgMeasurand = (rnd(0)*1024) shr 16
    wait_next
next

```

end

### Changing of Windows

In this chapter we will demonstrate step by step how change between two windows. In addition we will propose you how to organize your code in subroutines for a clearly arranged programming. You will find this example in *C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_5\_windows.TIG*

For this program we will use the same configuration for Tiger Graphic Library as in lesson 2.

The steps 1 and 2 are the standard starting steps for the programming with the Tiger Graphic Library. Choose the LCD type and include the Tiger Graphic Library.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC
```

Step 3 defines the static identifiers for the windows and the used graphic fonts.

```
' windows
#define WINDOW_ID_1      0
#define WINDOW_ID_2      1

' fonts
#define FONT_ID           0
```

Step 4 defines keycodes as return values from the touched buttons. You will need these constants later when you read out the touched buttons.

```
' keycodes
#define KEY_textbutton1 0
#define KEY_textbutton2 1
```

Steps 5 and 6 are the first two Steps in the task main which do always the same in our example programs: Declaring the variables and including the file for the installation of the device drivers.

```
byte blReturn      ' return value for TGL subroutines
word wElementId    ' incremental identifier for elements

#include TGL_DEVICE_DRIVERS_TP1000.INC
```

Step 7 to 10 are for initialization. You have to initialize the internal variables of The Tiger Graphic Library, the variables for this task, and the elements and windows you want to use in your application. As the creation of the elements and windows will

## Tiger Graphic Library Step by Step

take a number of code lines, we will separate these initializations in subroutines. The subroutine *wlInitGeneral* will create fonts and general elements which should be used and placed in several windows. The other subroutines *wlInitWindow1* and *wlInitWindow2* will create and place the elements in the specific windows. Like the initialization subroutines of the Tiger Graphic Library we pass the incremental identifier for the elements. Its input value is a free identifier and its return value is the next free identifier for elements.

```
set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wlElementId = 0
call wInitGeneral( wlElementId )
call wInitWindow1( wlElementId )
call wInitWindow2( wlElementId )
```

In the initialization subroutines we create a font and place a title and a text button in each window.

In step 11 we call the subroutine for the administration of the displaying of the windows on the LCD.

```
call bAdministrateWindows()
```

This subroutine runs in an endless loop. In step 11a we declare the variables for this elemental subroutine. Important is the variable *wlWindowId* which saves the identifier of the currently shown window.

```
byte ever      ' endless loop
byte blReturn  ' return value of the tgl subroutines
word wlWindowId ' identifier of currently shown window
```

In the following step 11b we set a start window and enter the endless loop.

```
wlWindowId = WINDOW_ID_1
for ever = 0 to 0 step 0
```

In each loop we show the current window on LCD and wait in the window subroutine for a pressed button. (steps 11c and 11d). With the pressed button the currently shown window will be changed and returned as parameter.

```
switchi wlWindowId
case WINDOW_ID_1:
    call wWindow1( wlWindowId )
    wlWindowId = WINDOW_ID_2
case WINDOW_ID_2:
    call wWindow2( wlWindowId )
    wlWindowId = WINDOW_ID_1
endswitch ' wlWindowId
```

Sample program:

```
'-----
' TGL_Lesson_5_windows.INC
'-----
' required software:   Tiger Graphic Library
' required hardware:  TP-1000
'-----
' *****
'          Step 1: Choose your LCD type
' *****
#define LCD_INVERSION_MODE      0          ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue" LCD

' *****
'          Step 2: Include the files for the Tiger Graphic Library
' *****
#include TigerGraphicLibrary.INC

' *****
'          Step 3: Determine the numbers of your identifiers
' *****
' windows
#define WINDOW_ID_1      0
#define WINDOW_ID_2      1

' fonts
#define FONT_ID          0

' *****
'          Step 4: Define keycodes for the buttons
' *****
' keycodes
#define KEY_textbutton1 0
#define KEY_textbutton2 1

task main

' *****
'          Step 5: Declare your variables
' *****
byte blReturn          ' return value for TGL subroutines
word wlElementId       ' incremental identifier for elements

' *****
'          Step 6: Install the device drivers for the touch panel and the LCD
' *****
```

```
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
' Step 7: Initialize the Tiger Graphic Library
'*****
set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wElementId = 0

'*****
' Step 8: Create general elements and fonts
'*****
call vInitGeneral( wElementId )

'*****
' Step 9: Build windows
'*****
call wInitWindow1( wElementId )
call wInitWindow2( wElementId )

'*****
' Step 10: Show windows
'*****
set_task_prio main, 1 ' reset task priority for usual running
call vAdministrateWindows()
end

'-----
' vInitGeneral:
'-----
' Create general elements and fonts
'-----
' RETURN VALUES:
'   wpvElementId      IN: free identifier for the creation of elements
'                     OUT: next free identifier
'-----
sub vInitGeneral( var word wpvElementId )
    byte blReturn

    call bTglCreateFontParams( &
    FONT_ID, &                ' identifier of font
    "Valencia", 21, "bold", &  ' name, size, type of font
    "center", "center", &    ' alignment horizontal, vertical
    "prop", 0, &              ' spacing type, blank
    SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
    "imm", "word", &         ' overlay, wrap mode
    blReturn )                ' return code (0: OK exit >0: error exit)
end

'-----
' wInitWindow1:
'-----
' Create and place elements in window 1.
'-----
' RETURN VALUES:
'   wpvElementId      IN: free identifier for the creation of elements
'                     OUT: next free identifier
'-----
```

```

sub wInitWindow1( var word wpvElementId )
    byte blReturn

    '' title
    call bTglCreateLabelWnd( &
LCD_WIDTH, 60, &          ' width, height of element
"Window 1", &            ' text of element
FONT_ID, 0, &            ' font identifier, frame thickness
wpvElementId, WINDOW_ID_1, & ' identifier of element, window
0, 0, &                  ' x, y coordinate on LCD
blReturn )                ' return code (0: OK exit  >0: error exit)
    '' increment identifier for next element
    wpvElementId = wpvElementId + 1

    '' button
    call bTglCreateTextButtonWnd( &
160, 80, &                ' width, height of element
"change", &                ' ftext of element
FONT_ID, 4, &              ' font identifier, frame thickness
TGL_KEY_ATTR_AUTOREPEAT_OFF,& ' key attribute
wpvElementId, WINDOW_ID_1, & ' identifier of element
40, 120, &                 ' x,y coordinate on LCD
KEY_textbutton1, &         ' keycode
blReturn )                 ' return code (0: OK exit  >0: error exit)
    '' increment identifier for next element
    wpvElementId = wpvElementId + 1
end

```

```

'-----
' wInitWindow2:
'-----
' Create and place elements in window 2.
' Abort Subroutine in case of failure.
'-----
' PARAMETERS:
'   wpWindowId          identifier for this window
'
' RETURN VALUES:
'   wpvElementId        IN: free identifier for the creation of elements
'                       OUT: next free identifier
'   bpvReturn           tgl return value (0=OK, >0=error)
'-----

```

```

sub wInitWindow2( var word wpvElementId )
    byte blReturn

    '' title
    call bTglCreateLabelWnd( &
LCD_WIDTH, 60, &          ' width, height of element
"Window 2", &            ' text of element
FONT_ID, 0, &            ' font identifier, frame thickness
wpvElementId, WINDOW_ID_2, & ' identifier of element, window
0, 0, &                  ' x, y coordinate on LCD
blReturn )                ' return code (0: OK exit  >0: error exit)
    '' increment identifier for next element
    wpvElementId = wpvElementId + 1

    '' button
    call bTglCreateTextButtonWnd( &

```

```

160, 80, &                                ' width, height of element
"change", &                                ' ftext of element
FONT_ID, 4, &                              ' font identifier, frame thickness
TGL_KEY_ATTR_AUTOREPEAT_OFF,&              ' key attribute
wpvElementId, WINDOW_ID_2, &              ' identifier of element
120, 120, &                                ' x,y coordinate on LCD
KEY_textbutton2, &                          ' keycode
blReturn )                                ' return code (0: OK exit >0: error exit)
' increment identifier for next element
wpvElementId = wpvElementId + 1
end

'-----
' vAdministrateWindows:
'-----
' Show start window and administrate the changing of windows
'-----
sub vAdministrateWindows()
'*****
' Step 11a: Declare lokal variables
'*****
byte ever ' endless loop
byte blReturn ' return value of the tgl subroutines
word wlWindowId ' identifier of currently shown window

'*****
' Step 11b: Set start window
'*****
wlWindowId = WINDOW_ID_1
for ever = 0 to 0 step 0

'*****
' Step 11c: Show current window
'*****
call bTglShowWindow( wlWindowId, blReturn )

'*****
' Step 11d: Get window input
'*****
switchi wlWindowId
case WINDOW_ID_1:
call wWindow1( wlWindowId )
wlWindowId = WINDOW_ID_2
case WINDOW_ID_2:
call wWindow2( wlWindowId )
wlWindowId = WINDOW_ID_1
endswitch ' wlWindowId
next ' endless loop
end

'-----
' wWindow1:
'-----
' Administrate user input in window 1
'-----
' RETURN VALUES:
' wpvWindowId IN: identifier of this window

```



```

'                                OUT: identifier of next window to be shown
'-----
sub wWindow1( var word wpvWindowId )
  ' endless loop
  byte ever
  ' button return code
  byte blKeycode

  for ever = 0 to 0 step 0
    call bTglWaitKeycode( blKeycode )
    switchi blKeycode
    case KEY_textbutton1:
      '*****
      ' Step 11di: Change window with return
      '*****
      wpvWindowId = WINDOW_ID_2
      return
    endswitch ' blKeycode
  next ' endless loop
end

'-----
' wWindow2:
'-----
' Administrate user input in window 2
'-----
' RETURN VALUES:
'   wpvWindowId           IN: identifier of this window
'                           OUT: identifier of next window to be shown
'-----
sub wWindow2( var word wpvWindowId )
  ' endless loop
  byte ever
  ' button return code
  byte blKeycode

  for ever = 0 to 0 step 0
    call bTglWaitKeycode( blKeycode )
    switchi blKeycode
    case KEY_textbutton2:
      '*****
      ' Step 11di: Change window with return
      '*****
      wpvWindowId = WINDOW_ID_1
      return
    endswitch ' blKeycode
  next ' endless loop
end

```

### Application Example

In this chapter you will find a template for a tgl application we would propose to copy for your own project. You will find this example in

*C:\Programme\Wilke Technology\Tiger Basic 5.3\Examples\TGL\_Examples\TGL\_STEP\_BY\_STEP\TGL\_Lesson\_6\_application.TIG*

For this program we will use the same configuration for Tiger Graphic Library as in lesson 2. The files *TGL\_Lesson\_6\_gui.INC* and *TGL\_Lesson\_6\_measure.INC* are part of this example, too.

In this application the functionality of the the examples in the lessons 1 to 5 will be united in a well structured program code as we would propose to do. This structure will it make easy for you to hold an overview over all the functionalities even in complex projects with a very long code.

The steps 1 and 2 are the standard starting steps for the programming with the Tiger Graphic Library. Choose the LCD type and include the Tiger Graphic Library.

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC
```

Step 3 we assembles all the program specific data which has to be poked in the user flash memory. We do this in a subroutine which will never be called. We suggest placing this subroutine behind the inclusion of the Tiger Graphic Library. This will allow you to identify the first free address in the user flash memory. Placing this subroutine in front of your own subroutines, tasks or inclusions will ensure the knowledge of the datalabels in all your subroutines.

If you you will poke data in the flash memory, the datalabel `dlTglFirstFreeAddr` could be helpful for you to find the free memory.

```
sub Flash()

    datalabel dlTitle1
    dlTitle1::
    data string "Title 1"

    datalabel dlTitle2
    dlTitle2::
    data string "Title 2"

    datalabel dlTextbutton
    dlTextbutton::
    data string "change window"

    ' first free address for poking the flash
    ' if there will NOT be any other data poked
```

```
' in following subroutines, tasks or inclusions
datalabel dlTglFirstFreeAddr
dlTglFirstFreeAddr::
end
```

The steps 4 and 5 are for the definitions of Identifiers, keycodes and the declaration of global variables.

```
' windows
#define WINDOW_ID_input      0
#define WINDOW_ID_measure   1
#define WINDOW_ID_keyboard   2

' fonts
#define FONT_ID_text         0
#define FONT_ID_number       1

' elements
word wgNameId
word wgKeyboardId
word wgValueId, wgBarId
word wgChangeButId

' keycodes
#define KEY_name             0
#define KEY_change           1

word wgMeasurand
```

Step 6 is for avoiding too long files and a better overview about the program modules. Instead of writing all the subroutines in the tig file we write inc files for each module. In this example we have one module for the graphical user interface with the window handling and one for the measuring. The compiler will insert the code lines of the include files to where the files are included.

```
#include TGL_Lesson_6_gui.INC
#include TGL_Lesson_6_measure.INC
```

Steps 7 and 8 are for declaration of variables and the installation of the device drivers.

```
byte blReturn      ' return value for TGL subroutines
word wlElementId    ' incremental identifier for elements
word wlWindowId     ' identifier for windows

#include TGL_DEVICE_DRIVERS_TP1000.INC
```

Step 9 is for initialization. New are the additional parameters for the initialization subroutines for the fonts, elements and windows.

## Tiger Graphic Library Step by Step

```
set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wElementId = 0
call wInitGeneral( wElementId, blReturn )
call wInitInput( WINDOW_ID_input, wElementId, blReturn )
call wInitMeasure( WINDOW_ID_measure, wElementId, blReturn )
wWindowId = WINDOW_ID_keyboard
call bTglInitKeyboard( &
TGL_KEYBL_STYLE_ENG, FONT_ID_text, & ' keyboard style, font
wElementId, wWindowId, & ' identifier of element, window
wgKeyboardId, & ' identifier for keyboard view
blReturn ) ' return code (0: OK exit >0: error exit)
```

We pass the identifier of the window to the initialization subroutines. This way we need not to modify the parameter for the window identifier in the called tgl subroutines anymore. Now we can pass the parameter *wpWindowId* to each tgl subroutine in each initialization subroutine. This will help avoiding copy paste errors.

As second new parameter we pass the variable for tgl return value. It will be very helpfull to check the tgl return code after each called tgl subroutine as done in step 9a.

```
if bpvReturn <> TGL_MSG_OK then
    return
endif
```

In case of failure the subroutine will be aborted by returning an error code. If the program does not do what you have been intended to do, you just need to debug the few return codes of the subroutines in the task main instead of the many tgl subroutines.

Sample program:

```
'-----
' TGL_STEP_6_application.TIG
'-----
' required software:    Tiger Graphic Library
' required hardware:    TP-1000
'-----
' *****
'          Step 1: Choose your LCD type
' *****
#define LCD_INVERSION_MODE    0          ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue"  LCD
' *****
'          Step 2: Include the files for the Tiger Graphic Library
' *****
#include TigerGraphicLibrary.INC
' *****
```

```

'      Step 3: Poke data in user flash
'*****
'-----
' Never called subroutine.
' Assembles the specific user flash data poked by download.
' Should be placed BEHIND the inclusion of the TigerGraphicLibrary
' to be able to identify the first address of the free user flash memory.
' Should be placed IN FRONT OF your own subroutines, tasks or inclusions
' to ensure the knowledge of the datalabels in all your subroutines.
' Datalabels are always global!
'-----
sub vFlash()

    datalabel dlName
    dlName::
    data string "name:"

    datalabel dlButton
    dlButton::
    data filter "button.bmp", "GRAPHFLT", 0      ' WxH=76x33 BmpWidth=80

    ' first free address for poking the flash
    ' if there will NOT be any other data poked
    ' in following subroutines, tasks or inclusions
    datalabel dlTglFirstFreeAddr
    dlTglFirstFreeAddr::
end

'*****
'      Step 4: Definitions and global variables for the gui
'*****
' windows
#define WINDOW_ID_input      0
#define WINDOW_ID_measure    1
#define WINDOW_ID_keyboard    2

' fonts
#define FONT_ID_text          0
#define FONT_ID_number        1

' elements
word wgNameId
word wgKeyboardId
word wgValueId, wgBarId
word wgChangeButId

' keycodes
#define KEY_name              0
#define KEY_change            1

'*****
'      Step 5: Application specific definitions and global variables
'*****
word wgMeasurand

'*****
'      Step 6: Application specific include files
'*****
#include TGL_Lesson_6_gui.INC

```

```
#include TGL_Lesson_6_measure.INC

'-----
' main:
'-----

task main

'*****
'    Step 7: Declare your variables
'*****
byte blReturn      ' return value for TGL subroutines
word wlElementId   ' incremental identifier for elements
word wlWindowId    ' identifier for windows

'*****
'    Step 8: Install the device drivers for the touch panel and the LCD
'*****
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'    Step 9: Initialize
'            i) Tiger Graphic Library
'            ii) variables
'            iii) general elements and fonts
'            iv) user application windows
'            v) tgl application windows
'*****
set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call vTglInit()
wlElementId = 0
call wInitGeneral( wlElementId, blReturn )
call wInitInput( WINDOW_ID_input, wlElementId, blReturn )
call wInitMeasure( WINDOW_ID_measure, wlElementId, blReturn )
wlWindowId = WINDOW_ID_keyboard
call bTglInitKeyboard( &
TGL_KEYB1_STYLE_ENG, FONT_ID_text, & ' keyboard style, font
wlElementId, wlWindowId, &          ' identifier of element, window
wgKeyboardId, &                    ' identifier for keyboard view
blReturn )                          ' return code (0: OK exit >0: error exit)

'*****
'    Step 10: Run application specific tasks
'*****
run_task tMeasure

'*****
'    Step 11: Display the start window on the LCD
'            and administrate the changing of the windows
'*****
set_task_prio main, 1 ' reset task priority for usual running
call bAdministrateWindows( blReturn )
end
```

```
'-----
' TGL_Lesson_6_gui.INC
'-----
```

```

-----
' wInitGeneral:
-----
' Create fonts and general elements.
' Abort Subroutine in case of failure.
-----
' RETURN VALUES:
'   wpvElementId      IN: free identifier for the creation of elements
'                     OUT: next free identifier
'   bpvReturn         tgl return value (0=OK, >0=error)
-----
sub wInitGeneral( var word wpvElementId; var byte bpvReturn )

    ' fonts
    call bTglCreateFontParams( &
    FONT_ID_text, &                                ' identifier of font
    "Valencia", 21, "bold", &                        ' name, size, type of font
    "center", "center", &                            ' alignment horizontal, vertical
    "prop", 0, &                                       ' spacing type, blank
    SPACING_CHAR_DEFAULT, 0, &                        ' spacing char, vertical
    "imm", "word", &                                  ' overlay, wrap mode
    bpvReturn )                                       ' return code (0: OK exit >0: error exit)

    *****
    ' Step 9a: Check return code
    *****
    if bpvReturn <> TGL_MSG_OK then
        return
    endif

    call bTglCreateFontParams( &
    FONT_ID_number, &                                ' identifier of font
    "Valencia", 21, "bold", &                        ' name, size, type of font
    "right", "center", &                            ' alignment horizontal, vertical
    "const", 0, &                                       ' spacing type, blank
    SPACING_CHAR_DEFAULT_DIGIT, 0, &                  ' spacing char, vertical
    "imm", "char", &                                  ' overlay, wrap mode
    bpvReturn )                                       ' return code (0: OK exit >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif

    ' general elements
    *****
    ' Step 9b: Save identifier of element for later placing
    *****
    wgChangeButId = wpvElementId

    call bTglCreateButton( &
    76, 33, &                                         ' width, height of element
    dlButton, 80, &                                  ' address, format width of bitmap
    TGL_KEY_ATTR_AUTOREPEAT_OFF, &                  ' key attributes auto repeat, beep, type
    wpvElementId, &                                  ' identifier of element
    bpvReturn )                                       ' return code (0: OK exit >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
endif

```

```

*****
'      Step 9c: Increment variable for next free element identifier
'              after each creation
*****
wpvElementId = wpvElementId + 1
end

-----
' wInitInput:
-----
' Create and place elements in window for input.
' Abort Subroutine in case of failure.
-----
' PARAMETERS:
'   wpWindowId          identifier for this window
'
' RETURN VALUES:
'   wpvElementId        IN: free identifier for the creation of elements
'                       OUT: next free identifier
'   bpvReturn           tgl return value (0=OK, >0=error)
-----

sub wInitInput( word wpWindowId; var word wpvElementId; &
var byte bpvReturn )

    call bTglCreateLabelFWnd( &
    200, 60, &                ' width, height of element
    dlName, &                 ' flash address of text of element
    FONT_ID_text, 0, &        ' font identifier, frame thickness
    wpvElementId, wpWindowId, & ' identifier of element, window
    60, 60, &                 ' x, y coordinate on LCD
    bpvReturn )               ' return code (0: OK exit >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
    wpvElementId = wpvElementId + 1

    call bTglCreateTextButtonVarWnd( &
    200, 60, &                ' width, height of element
    FONT_ID_text, 4, &        ' font identifier, frame thickness
    TGL_KEY_ATTR_AUTOREPEAT_OFF, & ' key attribute
    wpvElementId, wpWindowId, & ' identifier of element, window
    60, 120, &                ' x, y coordinate on LCD
    KEY_name, &               ' keycode
    bpvReturn )               ' return code (0: OK exit >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
    wgNameId = wpvElementId
    wpvElementId = wpvElementId + 1

    call bTglPlaceButtonInWindow( &
    wgChangeButId, wpWindowId, & ' identifier of element
    122, 205, &                ' x,y coordinate on LCD
    KEY_change, &              ' keycode
    bpvReturn )               ' return code (0: OK exit >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif

```



```

end

'-----
' wInitMeasure:
'-----
' Create and place elements in window for displaying a measurand.
' Abort Subroutine in case of failure.
'-----
' PARAMETERS:
'   wpWindowId          identifier for this window
'
' RETURN VALUES:
'   wpvElementId        IN: free identifier for the creation of elements
'                       OUT: next free identifier
'   bpvReturn           tgl return value (0=OK, >0=error)
'-----
sub wInitMeasure( word wpWindowId; var word wpvElementId; &
var byte bpvReturn )

call bTglCreateLabelWnd( &
160, 60, &                ' width, height of element
"DIGITS", &                ' flash address of text of element
FONT_ID_text, 0, &        ' font identifier, frame thickness
wpvElementId, wpWindowId,& ' identifier of element, window
0, 0, &                    ' x, y coordinate on LCD
bpvReturn )                ' return code (0: OK exit >0: error exit)
if bpvReturn <> TGL_MSG_OK then
return
endif
wpvElementId = wpvElementId + 1

call bTglCreateLabelVarWnd( &
140, 60, &                ' width, height of element
FONT_ID_number, 4, &      ' font identifier, frame thickness
wpvElementId, wpWindowId,& ' identifier of element, window
10, 90, &                 ' x, y coordinate on LCD
bpvReturn )                ' return code (0: OK exit >0: error exit)
if bpvReturn <> TGL_MSG_OK then
return
endif
wgValueId = wpvElementId
wpvElementId = wpvElementId + 1

call bTglCreateGaugeWnd( &
40, 220, &                ' width, height of element
0,1023, &                 ' limits of values
TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM,& ' type, location of base
4, &                      ' frame thickness
wpvElementId, wpWindowId, & ' identifier of the element, window
230, 10, &                 ' x, y coordinate on LCD
512, &                    ' start value for element
bpvReturn )                ' return code (0=Ok, >0=Error)
if bpvReturn <> TGL_MSG_OK then
return
endif
wgBarId = wpvElementId
wpvElementId = wpvElementId + 1

```

```

call bTglPlaceButtonInWindow( &
wgChangeButId, wpWindowId, & ' identifier of element
122, 205, & ' x,y coordinate on LCD
KEY_change, & ' keycode
bpvReturn ) ' return code (0: OK exit >0: error exit)
if bpvReturn <> TGL_MSG_OK then
return
endif
end
end

```

```

'-----
' bAdministrateWindows:
'-----
' Show start window and administrate the changing of windows
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
' bpvReturn          tgl return value (0=OK, >0=error)
'-----
sub bAdministrateWindows( var byte bpvReturn )
' ' endless loop
byte ever
' ' identifier of currently shown window
word wlWindowId

wlWindowId = WINDOW_ID_input
for ever = 0 to 0 step 0

call bTglShowWindow( wlWindowId, bpvReturn )
if bpvReturn <> TGL_MSG_OK then
return
endif

switchi wlWindowId
case WINDOW_ID_input:
call wInput( wlWindowId, bpvReturn )
case WINDOW_ID_measure:
call wMeasure( wlWindowId, bpvReturn )
default:
' ' invalid window
bpvReturn = 0FFh
return
endswitch ' wlWindowId
next ' endless loop
end
end

```

```

'-----
' wInput:
'-----
' Get name and show it.
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
' wpvWindowId      IN: identifier of this window
'                  OUT: identifier of next window to be shown
' bpvReturn         tgl return value (0=OK, >0=error)
'-----

```

```

sub wInput( var word wpvWindowId; var byte bpvReturn )
    byte blKeycode      ' button return code
    string slName$(40h) ' user input

    set_len$( slName$, 0 )
    call bTglWaitKeycode( blKeycode )
    switchi blKeycode
    case KEY_name:
        call sTglGetKeyboardInput( &
            WINDOW_ID_keyboard, wgKeyboardId, &' identifiers for keyboard
            040h, slName$, &          ' maximal user input length, user input
            bpvReturn )              ' return code (0: OK exit >0: error exit)
        if bpvReturn <> TGL_MSG_OK then
            return
        endif
        call bTglSetText( wgNameId, slName$, bpvReturn )
        if bpvReturn <> TGL_MSG_OK then
            return
        endif
        '' WINDOW_ID_input will be recalled!
        return
    case KEY_change:
        wpvWindowId = WINDOW_ID_measure
        return
    default:
        '' invalid window
        bpvReturn = 0FFh
        return
    endswitch ' blKeycode
end

```

```

'-----
' wMeasure:
'-----
' Display measurands on LCD.
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
'   wpvWindowId      IN: identifier of this window
'                   OUT: identifier of next window to be shown
'   bpvReturn        tgl return value (0=OK, >0=error)
'-----

```

```

sub wMeasure( var word wpvWindowId; var byte bpvReturn )
    byte ever          ' endless loop
    byte blKeycode      ' button return code
    word wLIbuFill      ' filling of button buffer
    word wLMeasurand    ' task input
    long lIValue        ' convert word to long

    for ever = 0 to 0 step 0

        '' check buttons
        call bTglGetKeycode( blKeycode, wLIbuFill )
        if 0 < wLIbuFill then
            switchi blKeycode
            case KEY_change:
                wpvWindowId = WINDOW_ID_input
                return

```

```

default:
  ' invalid window
  bpvReturn = 0FFh
  return
endswitch ' blKeycode
endif

' update LCD
if wlMeasurand <> wgMeasurand then
  ' save new measurand
  wlMeasurand = wgMeasurand
  ' convert word to long
  llValue = wgMeasurand
  ' update internally first without updating LCD
  call bTglShowLong(      wgValueId, llValue, TGL_FALSE, bpvReturn )
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  call bTglShowGaugeValue( wgBarId, llValue, TGL_FALSE, bpvReturn )
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  ' update the whole LCD now in one go
  call vTglUpdate()
endif

release_task
next ' endless loop
end call bAdministrateWindows( blReturn )
end

```

```

'-----
' TGL_Lesson_6_gui.INC
'-----

' wInitGeneral:
'-----
' Create fonts and general elements.
' Abort Subroutine in case of failure.
'-----

' RETURN VALUES:
'   wpvElementId      IN: free identifier for the creation of elements
'                     OUT: next free identifier
'   bpvReturn          tgl return value (0=OK, >0=error)
'-----

sub wInitGeneral( var word wpvElementId; var byte bpvReturn )

  ' fonts
  call bTglCreateFontParams( &
    FONT_ID_text, &                                ' identifier of font
    "Valencia", 21, "bold", &                        ' name, size, type of font
    "center", "center", &                            ' alignment horizontal, vertical
    "prop", 0, &                                       ' spacing type, blank
    SPACING_CHAR_DEFAULT, 0, &                        ' spacing char, vertical
    "imm", "word", &                                  ' overlay, wrap mode
    bpvReturn )                                       ' return code (0: OK exit  >0: error exit)

```

```

*****
'      Step 9a: Check return code
*****
if bpvReturn <> TGL_MSG_OK then
    return
endif

call bTglCreateFontParams( &
FONT_ID_number, &          ' identifier of font
"Valencia", 21, "bold", &   ' name, size, type of font
"right", "center", &       ' alignment horizontal, vertical
"const", 0, &              ' spacing type, blank
SPACING_CHAR_DEFAULT_DIGIT,0,&' spacing char, vertical
"imm", "char", &          ' overlay, wrap mode
bpvReturn )                ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
    return
endif

' general elements
*****
'      Step 9b: Save identifier of element for later placing
*****
wgChangeButId = wpvElementId

call bTglCreateButton( &
76, 33, &                  ' width, height of element
dlButton, 80, &            ' address, format width of bitmap
TGL_KEY_ATTR_AUTOREPEAT_OFF, &' key attributes auto repeat, beep, type
wpvElementId, &           ' identifier of element
bpvReturn )                ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
    return
endif

*****
'      Step 9c: Increment variable for next free element identifier
'              after each creation
*****
wpvElementId = wpvElementId + 1
end

-----
' wInitInput:
-----
' Create and place elements in window for input.
' Abort Subroutine in case of failure.
-----
' PARAMETERS:
'   wpWindowId          identifier for this window
'
' RETURN VALUES:
'   wpvElementId        IN: free identifier for the creation of elements
'                       OUT: next free identifier
'   bpvReturn           tgl return value (0=OK, >0=error)
-----

```

```

sub wInitInput( word wpWindowId; var word wpvElementId; &
var byte bpvReturn )

    call bTglCreateLabelFWnd( &
    200, 60, &                                ' width, height of element
    dlName, &                                ' flash address of text of element
    FONT_ID_text, 0, &                        ' font identifier, frame thickness
    wpvElementId, wpWindowId,&                ' identifier of element, window
    60, 60, &                                ' x, y coordinate on LCD
    bpvReturn )                              ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
    wpvElementId = wpvElementId + 1

    call bTglCreateTextButtonVarWnd( &
    200, 60, &                                ' width, height of element
    FONT_ID_text, 4, &                        ' font identifier, frame thickness
    TGL_KEY_ATTR_AUTOREPEAT_OFF,&            ' key attribute
    wpvElementId, wpWindowId,&                ' identifier of element, window
    60, 120, &                                ' x, y coordinate on LCD
    KEY_name, &                                ' keycode
    bpvReturn )                              ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
    wgNameId = wpvElementId
    wpvElementId = wpvElementId + 1

    call bTglPlaceButtonInWindow( &
    wgChangeButId, wpWindowId, &            ' identifier of element
    122, 205, &                                ' x,y coordinate on LCD
    KEY_change, &                                ' keycode
    bpvReturn )                              ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
        return
    endif
end

'-----
' wInitMeasure:
'-----
' Create and place elements in window for displaying a measurand.
' Abort Subroutine in case of failure.
'-----
' PARAMETERS:
'   wpWindowId            identifier for this window
'
' RETURN VALUES:
'   wpvElementId          IN: free identifier for the creation of elements
'                           OUT: next free identifier
'   bpvReturn              tgl return value (0=OK, >0=error)
'-----
sub wInitMeasure( word wpWindowId; var word wpvElementId; &
var byte bpvReturn )

    call bTglCreateLabelWnd( &
    160, 60, &                                ' width, height of element

```

```

"DIGITS", &                                ' flash address of text of element
FONT_ID_text, 0, &                          ' font identifier, frame thickness
wpvElementId, wpWindowId, &                ' identifier of element, window
0, 0, &                                    ' x, y coordinate on LCD
bpvReturn )                                ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
    return
endif
wpvElementId = wpvElementId + 1

call bTglCreateLabelVarWnd( &
140, 60, &                                ' width, height of element
FONT_ID_number, 4, &                      ' font identifier, frame thickness
wpvElementId, wpWindowId, &                ' identifier of element, window
10, 90, &                                ' x, y coordinate on LCD
bpvReturn )                                ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
    return
endif
wgValueId = wpvElementId
wpvElementId = wpvElementId + 1

call bTglCreateGaugeWnd( &
40, 220, &                                ' width, height of element
0,1023, &                                ' limits of values
TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM, &    ' type, location of base
4, &                                    ' frame thickness
wpvElementId, wpWindowId, &                ' identifier of the element, window
230, 10, &                                ' x, y coordinate on LCD
512, &                                    ' start value for element
bpvReturn )                                ' return code (0=Ok, >0=Error)
if bpvReturn <> TGL_MSG_OK then
    return
endif
wgBarId = wpvElementId
wpvElementId = wpvElementId + 1

call bTglPlaceButtonInWindow( &
wgChangeButId, wpWindowId, &                ' identifier of element
122, 205, &                                ' x,y coordinate on LCD
KEY_change, &                              ' keycode
bpvReturn )                                ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
    return
endif
end

'-----
' bAdministrateWindows:
'-----
' Show start window and administrate the changing of windows
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
'     bpvReturn          tgl return value (0=OK, >0=error)
'-----
sub bAdministrateWindows( var byte bpvReturn )
    '' endless loop

```

```

byte ever
' identifier of currently shown window
word wlWindowId

wlWindowId = WINDOW_ID_input
for ever = 0 to 0 step 0

    call bTglShowWindow( wlWindowId, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
        return
    endif

    switchi wlWindowId
    case WINDOW_ID_input:
        call wInput( wlWindowId, bpvReturn )
    case WINDOW_ID_measure:
        call wMeasure( wlWindowId, bpvReturn )
    default:
        ' invalid window
        bpvReturn = 0FFh
        return
    endswitch ' wlWindowId
next ' endless loop
end

'-----
' wInput:
'-----
' Get name and show it.
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
'   wpvWindowId      IN:  identifier of this window
'                   OUT: identifier of next window to be shown
'   bpvReturn        tgl return value (0=OK, >0=error)
'-----

sub wInput( var word wpvWindowId; var byte bpvReturn )
    byte blKeycode      ' button return code
    string slName$(40h) ' user input

    set_len$( slName$, 0 )
    call bTglWaitKeycode( blKeycode )
    switchi blKeycode
    case KEY_name:
        call sTglGetKeyboardInput( &
            WINDOW_ID_keyboard, wgKeyboardId, &' identifiers for keyboard
            040h, slName$, &' maximal user input length, user input
            bpvReturn ) ' return code (0: OK exit >0: error exit)
        if bpvReturn <> TGL_MSG_OK then
            return
        endif
        call bTglSetText( wgNameId, slName$, bpvReturn )
        if bpvReturn <> TGL_MSG_OK then
            return
        endif
        ' WINDOW_ID_input will be recalled!
        return
    case KEY_change:

```



```

        wpvWindowId = WINDOW_ID_measure
        return
    default:
        ' invalid window
        bpvReturn = 0FFh
        return
    endswitch ' blKeycode
end

'-----
' wMeasure:
'-----
' Display measurands on LCD.
' Abort Subroutine in case of failure.
'-----
' RETURN VALUES:
'   wpvWindowId      IN: identifier of this window
'                   OUT: identifier of next window to be shown
'   bpvReturn        tgl return value (0=OK, >0=error)
'-----

sub wMeasure( var word wpvWindowId; var byte bpvReturn )
    byte ever          ' endless loop
    byte blKeycode      ' button return code
    word wIbuFill       ' filling of button buffer
    word wlMeasurand    ' task input
    long llValue        ' convert word to long

    for ever = 0 to 0 step 0

        ' check buttons
        call bTglGetKeycode( blKeycode, wIbuFill )
        if 0 < wIbuFill then
            switchi blKeycode
            case KEY_change:
                wpvWindowId = WINDOW_ID_input
                return
            default:
                ' invalid window
                bpvReturn = 0FFh
                return
            endswitch ' blKeycode
        endif

        ' update LCD
        if wlMeasurand <> wgMeasurand then
            ' save new measurand
            wlMeasurand = wgMeasurand
            ' convert word to long
            llValue = wgMeasurand
            ' update internally first without updating LCD
            call bTglShowLong(   wgValueId, llValue, TGL_FALSE, bpvReturn )
            if bpvReturn <> TGL_MSG_OK then
                return
            endif
            call bTglShowGaugeValue( wgBarId, llValue, TGL_FALSE, bpvReturn )
            if bpvReturn <> TGL_MSG_OK then
                return
            endif
        endif
    endfor
end

```

```
        '' update the whole LCD now in one go
        call vTglUpdate()
    endif

    release_task
next ' endless loop
end
```

```
'-----
' TGL_Lesson_6_Measure.INC
'-----

' tMeasure:
'-----
' Simulate a measurand secondly
' which is normally get from device driver and following calculations
'-----

task tMeasure
    byte ever
    randomize
    wait_next 1000
    for ever=0 to 0 step 0
        '' simulate 10-bit measurand
        wgMeasurand = (rnd(0)*1024) shr 16
        wait_next
    next
end
```

# Components and its Handling

## Elements

The Tiger Graphic Library is based on elements. Everything you can see on the LCD or you can use in your program are elements. The simplest element is a graphic. It is nothing else but a bitmap of a certain size placed on the LCD. More complex elements are e.g. sliders or buttons. These elements additionally contain touch panel functions. In order to display an element on your LCD you have to create it first.

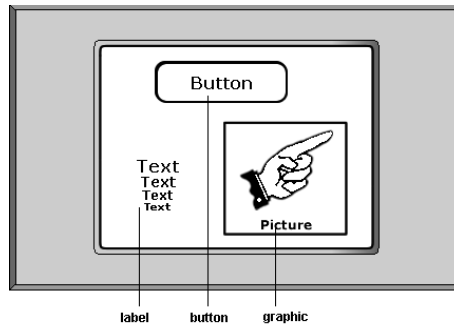


figure 6: Elements of the Tiger Graphic Library

Every element has a unique identifier. This identifier is given by the user when he creates the element. You can use or change this element by its identifier. Ensure that you use every identifier for an element only once, even if the elements differ in type. Two graphic elements need two different identifiers. Possible values for the identifiers are 0...65534. In the default configuration the maximum value for an element is **1999**. You can change this limit by creating your own copy of the configuration file *TigerGraphicLibraryConf.INC*. Please read chapter *Configuration* before doing this.

Available elements in the Tiger Graphic Library:

- Graphic
- Label
- Button
- Text button
- Slider
- Listbox
- Gauge

### Windows

A window is a container of many elements which can be shown on LCD together. A created element is saved in the memory but it is not used yet. To show an element on the LCD, it has to be placed into a window. It is possible to create many windows, which can be switched at any time. E.g. you could generate a keyboard and wait for input. After completing the input, a second window with status information can be shown.



figure 7: Input number



figure 8: Display Input

Each window has its own identifier. You can use or change a specific window by its identifier. Possible values for the identifiers are 0...65534. In the default configuration the maximum value for a window is **99**. You can change this limit by creating your own copy of the configuration file *TigerGraphicLibraryConf.INC*. Please read chapter *Configuration* before doing this.

The identifiers of the windows differ from the identifiers of the elements. You can use elements and windows with identical identifiers, e.g. an element with identifier 5 and a window with identifier 5 at the same time are allowed.

! There is no problem to use one element in different windows, but never use one element several times in one window.

After placing an element in a window you can show or hide each element as needed in your project.

### Creating and Placing Elements

You create an element by defining its width, height and occasionally its specific attributes. After that you can place your element in one or more windows at the position with the coordinates x, y. For the same element this position can be different in each window. Some elements have specific attributes e.g. keycodes for buttons which can be different in each window, too. For details see the special chapters for each element.

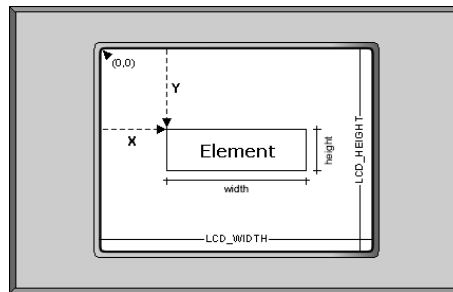
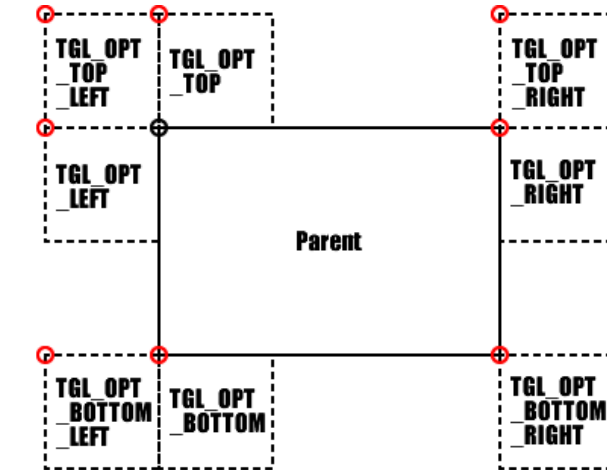


figure 9: Creating and placing elements

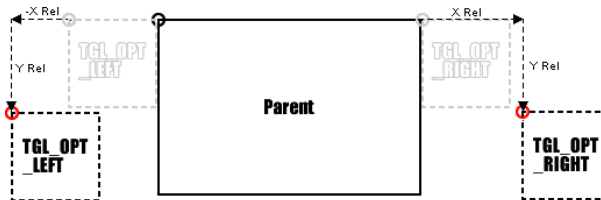
With the placing of elements in a window you will define an assembly of elements which will be shown together on LCD with all its touch panel functionalities. The Tiger Graphic Library will administrate these windows for you. After placing your elements in windows you just need to call a single subroutine for switching from one window to another.

### Docking

The standard way to place an element into a window is to set the absolute x-/y-coordinates. The docking function is an easy way to place elements relative to other elements. You can move the docking point with the x-/y-offset.



- Source
- Docking Point



- Source
- Docking Point

figure 10: Docking elements

### Integrated Applications

Integrated applications are no elements, but consist of one or more predefined complete windows, which can be easily shown in your projects. Normally they are ready for use after calling an initialization subroutine and calling one single subroutine.

Available integrated applications in the Tiger Graphic Library:

- Keyboards
- RTC Applications

For details see the chapters in this manual.



figure 11: RTC application as integrated application



figure 12: Keyboard as integrated application -

! Besides the integrated applications you will find templates in this manual. Have a look on these applications. Surely you will find some useful solutions for your own applications.

### Graphic Fonts

The Tiger Graphic Library works with individual graphic fonts. You do not need the LCD fonts anymore. Now you have the possibility to design the fonts by your own style as simple as the LCD fonts do, without any need of more program code.



figure 13: Various graphic fonts

Usually the text graphics are placed on the LCD using labels. See the chapter *Labels* for details.

If the labels should have touch panel functionalities, please see chapter *Text Buttons* for details.

It is also possible to create text graphics without creating any element. For details see the subroutines in chapter *Text Graphics*

Available special subroutines for Text Graphics in the Tiger Graphic Library:

- *bTglCreateFontParams*
- *bTglCreateFont*
- *bTglSetFontParams*
- *bTglGetFontParams*
- *sTglBuildTextGraphic*
- *lTglCalcTextToWindow*
- *lTglGetLineHeight*
- *lTglCalcTextGraphicWidth*
- *sTglCalcLineWidths*



### User Graphics

It is certainly possible to draw and show graphics by your own with the graphic functions of the Tiger-BASIC™ programming language at the same time as showing a window assembled by the subroutines of the Tiger Graphic Library.

For easy displaying your own graphic the Tiger Graphic Library provides you some special subroutines for LCD output. To be sure not to be disturbed by the outputs of the Tiger Graphic Library please use these subroutines.

Available special subroutines for LCD output in the Tiger Graphic Library:

- *vTglShowUserGraphic*
- *vTglShowUserGraphicParams*
- *vTglHideUserGraphic*
- *sTglGetWindowGraphic*
- *sTglPutWindowGraphic*
- *sTglClearWindowGraphic*
- *vTglPutStringToLcd*
- *vTglPutStringToLcdParam*
- *vTglUpdateLcd*
- *vTglPutFlashToLcd*
- *sTglDrawGraph*

For details see the chapter *User Graphic*.

! For LCD outputs mind using the one of the two LCD layers the Tiger Graphic Library does not use. The LCD device driver will "or" these two layers. In the default configuration the Tiger Graphic Library uses the layer 1. You can determine this layer by the define *SHOW\_WINDOW\_LAYER* in the file *TigerGraphicLibraryDefs.INC*. If you use the special subroutines for LCD output you need not care about this!

### Graphical Functions

The Tiger Graphic Library provides standard graphical functions. Especially for a dynamical creation of graphics in a running program you can use these functions. You can use these functions for visualizing values.

Available special subroutines for graphical functions in the Tiger Graphic Library:

- *sTglDrawRectangle*
- *sTglDrawFrame*
- *sTglDrawBar*
- *sTglDrawCircle*
- *sTglDrawPie*
- *sTglUpdatePie*
- *sTglDrawHand*
- *wTglCalcCircleParams*
- *sTglDrawGraph*

For details see the chapter *Graphical Functions*.

### Editor

For displaying correctly the program codes of the example programs and the source code ensure the Tiger BASIC™ editor setting "Options->Editor...: Tabstop = 8.

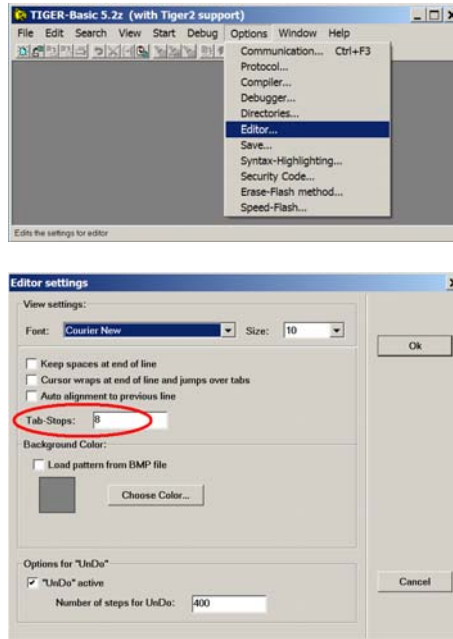


figure 14: Editor settings

### Tips and Tricks

Each subroutine of the Tiger Graphic Library returns a result about its operation. It helps you debugging your program. The return value informs you an invalid handling of subroutines. If an element in your program does not work as you expect, please check this return value. For a directed search start debugging in the following order:

- creation of elements or initialization of the integrated application
- placing of elements in windows
- subroutines for changing attributes of elements
- subroutines for showing elements and windows
- subroutines for working with elements

For optimizing the memory you can copy the file *TigerGraphicLibraryConf.INC* in the directory of your project and modify the parameters which are defined in this file for your own usage. For more information see chapter *Configuration*.

! NEVER build a text graphic by passing an uninitialized string. Before filling a string with a text graphic ensure that the string has the right length for the given graphic area.

! NEVER call a tgl subroutine when the task switching is disabled! Otherwise the program will could hang in an endless loop. Internal tgl tasks would be disabled which garant a correct multitasking functionality.

# Configuration

You can configure the Tiger Graphic Library for your own use by setting parameters in the file *TigerGraphicLibraryConf.INC*. For small projects you can use the given standard configuration. For bigger projects or projects with little memory you have to modify the configuration for your own use. Configuring the Tiger Graphic Library mainly means giving memory for the elements you use in your project and saving memory for elements you do not use. The default configuration takes about 270k bytes of RAM and 410kBytes of ROM and allows the use of 2000 elements in 100 windows.

If you configure the Tiger Graphic Library for your project, do NOT change the original configuration file in the directory of the Tiger Graphic Library. This file with its standard configuration runs with all the examples of the Tiger Graphic Library and all the small programs you will write. If you want to make more projects with the Tiger Graphic Library with different configurations, please copy the file *C:\Programme\Wilke Technology\Tiger Basic 5.3\TigerGraphicLibrary\TigerGraphicLibraryConf.INC* into your directory for your project containing your \*.TIG-file. This way you will be able to create a specific configuration file for each project.

## Why should I modify TigerGraphicLibraryConf.INC?

In smaller projects you can save RAM memory, if you reduce the number of elements. In very large projects you can increase the number of used elements, if you need more of them. A very individual configuration is possible. You could need many graphics but no slider for example. Perhaps you do not need any graphic font or you do not want to use any of the applications the Tiger Graphic Library provides. Just try the standard configuration. Normally you won't need any changes, but defining your fonts.

### General Settings

As in each other Tiger-BASIC™ application you can make some global settings for your project. For details see the programming manual of the Tiger-BASIC™ language.

The sizes for *user\_string\_size*, *user\_tempstr\_size* and *user\_stack\_size* are default values.

```
' ' initializes all variables for program start
' ' ==> should be deactivated for development!!
user_var_init
' ' enforces declarations of variables
' ' ==> helps avoiding errors caused by wrong types of variable!!
user_var_strict
' ' default string size for declarations
' ' can be 0 if string length is given for each string declaration
' ' defining each string length by declaration saves stack memory
user_string_size    64
' ' size of temporary strings in some string operations
' ' combined operations or logical terms could cause errors,
' ' if this value is smaller than the used string
user_tempstr_size   9600
' ' memory size for all variables and subs in one task
' ' a program require this size of ram for each task
' ' the tgl runs with additionally 2 tasks
' ' graphic fonts    require a minimum stack size of 300 bytes
' ' bitmap elements require a minimum stack size of 400 bytes
' ' text elements    require a minimum stack size of 700 bytes
user_stack_size     2k
```

### Master Defines

The master defines are controlling a whole group of functions of the Tiger Graphic Library. If you don't need them, you can save a lot of RAM and FLASH memory.

Define	Functionality
TGL_ELEMENTS	Activates elements and windows
TGL_TOUCHPANEL	Activates all elements, which use the touch panel. You could turn them off, if you only need graphical functions.
TGL_GRAPHIC_FONTS	Activates all elements, which make use of the Graphic Fonts. Graphic Fonts automatically creates graphical text on the LCD from standard strings or constants. You can turn them off, if you only need graphical elements and no elements with texts.
TGL_KEYBOARDS	Activates keyboards for user input in different styles.
TGL_RTC_APPLICATIONS	Activates all graphical RTC applications

For a more detailed choice of the font styles see section *Choice of Graphic Fonts*.  
For a more detailed choice of the keyboard styles see section *Choice of Keyboards*.  
For a more detailed choice of the RTC application styles see section *Choice of RTC applications*.

To activate a group of functions, the according define must be active. This means that there is no starting inverted comma in the first position of the line. If you want to deactivate a define, just comment out the line with the define with an inverted comma in the first position of the line.

In the following example elements and windows, touch panel functions and graphic fonts are activated but not the keyboard and RTC applications.

```
#define TGL_ELEMENTS           ' activates elements and windows
#define TGL_TOUCHPANEL        ' activate all touch panel functions
#define TGL_GRAPHIC_FONTS      ' activate all graphic font functions
'#define TGL_KEYBOARDS         ' activate keyboard applications
'#define TGL_RTC_APPLICATIONS  ' activate RTC applications
```

### Memory for Elements

There are several defines to determine its the maximum numbers. For each element RAM memory is reserved, which can not be used for other data. An individual configuration can save RAM. For the number of elements mind summarizing the numbers of elements of all types.

Define	Bytes per Element
TGL_MAX_NUM_ELEMENTS	4
TGL_MAX_NUM_GRAPHICS	14
TGL_MAX_NUM_LABELS	23
TGL_MAX_NUM_BUTTONS	15
TGL_MAX_NUM_TEXT_BUTTONS	24
TGL_MAX_NUM_SLIDERS	43
TGL_MAX_NUM_GAUGES	20
TGL_MAX_NUM_LISTBOXES	23

Example: You need 100 Buttons in your application. This will reserve 1500 Bytes of RAM. Each Button needs 15 Bytes =>  $100 \times 15 = 1500$  Bytes. Just write:

```
#define TGL_MAX_NUM_BUTTONS 100
```



### Memory for Windows

You can specify the maximum number of windows with `TGL_MAX_NUM_WINDOWS`. There is an internal memory pool for all attributes of every window. The default size of this memory pool is 20k. If you need more space, you will get an error `TGL_ERR_WINDOW_STR_LEN`. In this case increase the value of `TGL_WINDOW_ATTRIBUTES_LEN`. Do not set a too small value for this define. For orientations you need about 10 to 14 bytes per element which is placed in a window plus some bytes for administration of the windows and element types.

The define `TGL_MAX_NUM_BLINK` limits the maximum number of blinking elements in the same window.

The Tiger Graphic Library controls the `TOUCHPANEL.TDD` device driver. The device driver needs some direct access strings, which contain information about active elements like buttons or sliders. These driver strings are administrated from the Tiger Graphic Library itself. You can set the maximal length of these strings. By limiting the maximal touch elements which are active in the same window.

Define	Description
<code>TGL_MAX_NUM_WINDOWS</code>	maximal numbers of windows
<code>TGL_WINDOW_ATTRIBUTES_LEN</code>	size of memory pool for specific attributes of elements in windows
<code>TGL_MAX_NUM_BLINK</code>	maximum number of blinking elements in the same window
<code>TGL_MAX_NUM_BUTTONS_IN_WINDOW</code>	maximal number of buttons in one window
<code>TGL_MAX_NUM_SLIDERS_IN_WINDOW</code>	maximal number of sliders in one window

Default setting:

```
#define TGL_MAX_NUM_WINDOWS      100
#define TGL_WINDOW_ATTRIBUTES_LEN 20k
#define TGL_MAX_NUM_BLINK       50
#define TGL_MAX_NUM_BUTTONS_IN_WINDOW 100
#define TGL_MAX_NUM_SLIDERS_IN_WINDOW 20
```

### Memory for Graphic Fonts

The Graphic Fonts provide comfortable subroutines for an easy use. Doing this the Tiger Graphic Library needs RAM. For each task you are calling one of the subroutines of the Tiger Graphic Library handling with Graphic Fonts you have to reserve RAM. You do this by the define `TGL_MAX_NUM_TXT_GRAPHIC_STR`. If you use subroutines of the Graphic Fonts or the Tiger Graphic Library in more different tasks, you could get an error `TGL_ERR_FONT_TASKS_OVERFLOW`. In this case increase the value of `TGL_MAX_NUM_TXT_GRAPHIC_STR`.

You can also specify the values of `TGL_GRA_WIDTH` and `TGL_GRA_HEIGHT`. It defines the maximum size for one of the text graphic strings. Normally this size is equal the size of the LCD. The return value `TGL_ERR_FONT_GRAPHIC_OVERFLOW` indicates an error if you have tried to build a too large text graphic. In this case please increase the values of `TGL_GRA_WIDTH` and `TGL_GRA_HEIGHT`. The define `TGL_TXT_GRAPHIC_TXT_LEN` determines the maximum number of characters of which the text graphic has to be built.

The define `TGL_MAX_NUM_TXT_GRAPHIC_STR` determines the maximum number of tasks that can simultaneously make use of the Graphic Fonts functions. When using these functions **only** in combination with elements and windows, the maximum number of tasks will be 1. Setting the define to the smallest possible value will save RAM for your project.

The define `TGL_MAX_NUM_FONTS` determines the maximum number of different fonts you are going to create.

All elements, which contain text data, need memory for their texts. You have two possibilities to save the text. The most economical way for constant texts is saving the texts in the flash. In this case you have to set a datalabel, write the 4 byte length of the text followed by the text itself into the flash.

Variable texts must be saved in RAM. The size of the memory pool for these texts is specified by `TGL_MAX_MEM_TEXTS_LEN`. For Tiger 1 this value is limited on 7FFCh (~31k). For Tiger 2 there is no limit.

Define	Description
<code>TGL_GRA_WIDTH</code>	maximum width of text graphics
<code>TGL_GRA_HEIGHT</code>	maximum height of text graphics
<code>TGL_TXT_GRAPHIC_TXT_LEN</code>	maximum text length for building text graphics in one go
<code>TGL_MAX_NUM_FONTS</code>	maximum number of created fonts

## Configuration

Define	Description
TGL_MAX_MEM_TEXTS_LEN	size of memory pool for all texts in elements which are not saved in the flash memory
TGL_MAX_NUM_TXT_LINES	maximal number of lines for text graphics

Default setting:

```
' ' maximal size for text graphics for building text graphic in one go
' ' required RAM = (TGL_GRA_WIDTH/8)*TGL_GRA_HEIGHT
#define TGL_GRA_WIDTH          320 ' must be a multiple of 8!
#define TGL_GRA_HEIGHT         240
' ' maximal text length building text graphic in one go
#define TGL_TXT_GRAPHIC_TXT_LEN 512
' ' maximal number of created fonts
' ' required RAM = 25 * TGL_MAX_NUM_FONTS
#define TGL_MAX_NUM_FONTS      100
' ' memory pool for texts in elements which are not saved in the flash memory
#define TGL_MAX_MEM_TEXTS_LEN  10k
' ' determine maximal number of lines for text graphics
' ' required RAM = 8*TGL_MAX_NUM_TXT_LINES
#define TGL_MAX_NUM_TXT_LINES  20
```

### Choice of Graphic Fonts

In your project you can choose from many different fonts. For each font you use in your project you have to apply it by setting a define for this font. You have to compose this define yourself. You may define as many fonts as you require. The number of fonts you define is not limited by the value of TGL\_MAX\_NUM\_FONTS, but take care to define not too many fonts, to save flash memory. All font bitmaps will be placed in flash. The amount of used flash mainly depends on the font size. For example Helsinki\_7\_n.bmp uses 6kbytes of flash and VALENCIA\_48\_NORMAL uses 155kbytes of flash. For used flash memory please see the sizes of the bitmaps in your explorer *TigerGraphicLibrary\Graphic\_Fonts\Bitmaps*

Fonts are defined by its name, type and size.

Name	Type	Size
Amsterdam	bold	8,11,16,21
Atlanta	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Helsinki	normal bold	7,8,9,10,11,12,14,18,22,26 10,12,14,18,22,26,28,32,52,56,60
Istanbul	normal	8,10,11,12,14,18,21
Stockholm	bold	8,11,16,21
Tokio	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Valencia	normal, bold italic, bold italic	8,10,11,12,14,18,21,24,36,48 8,10,11,12,14,18,21,24,36,48

Finally the font families of the fonts you are going to use must be included. Font families are Valencia, Tokio, ... .

E.g if you want to use VALENCIA\_12\_BOLD and TOKIO\_18\_NORMAL please decomment the following code lines:

```
#define VALENCIA_12_BOLD
#define TOKIO_18_NORMAL
#include TGL_GRAFO_VALENCIA.INC
#include TGL_GRAFO_TOKIO.INC
```

### Choice of Keyboards

The Tiger Graphic Library provides touch panel keyboards for user input. If the master define TGL\_KEYBOARDS is activated you have the choice between keyboards in different styles and for different use.

E.g if you need a keyboard in Hungarian style with all the Hungarian special chars the code must be as follows:

```
'#define TGL_KEYB_DIG_1      ' digit block
'#define TGL_KEYB_1_ENG     ' english style
'#define TGL_KEYB_1_GER     ' german style
'#define TGL_KEYB_1_TRK     ' turkey style
#define TGL_KEYB_1_HUN      ' hungarian style
```



figure 15: Keyboard Hungarian style unshifted



figure 16: Keyboard Hungarian style shifted

### Choice of RTC Applications

The Tiger Graphic Library provides applications for setting date and time using the touch panel and displaying the current time and date on LCD. If the master define TGL\_RTC\_APPLICATIONS is activated you have the choice between RTC applications of different styles and for different use.

E.g if you want to show and set the time and the date in a digital style, the code must be as follows:

```
#define TGL_DEF_RTC_STYLE_1      ' show and set a digital clock
```



figure 17: displaying time and date in digital style 1

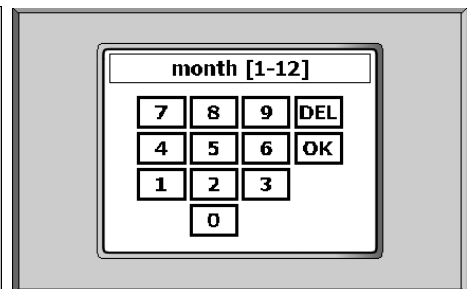


figure 18: setting time and date in digital style 1

### Hardware

The default settings in the configuration file for the Tiger Graphic Library are for the TP 1000. The settings concern the beeper, the LCD and the touch panel. The include file TGL\_DEVICE\_DRIVERS\_TP1000.INC uses these defines.

For the BEEPER you have to look up the ports and pins in the data sheets of your hardware.

```
# ' BEEPER
#define PORT_BEEPER      4
#define PIN_BEEPER       2
#define BEEPER_ON        0FFh
#define BEEPER_OFF       0
```

Each type of LCD has its own size. If your LCD differs from quarter VGA size (320x240) change the settings for the size of the LCD.

```
#define LCD_WIDTH        320    ' pixel width
#define LCD_HEIGHT       240    ' pixel height
```

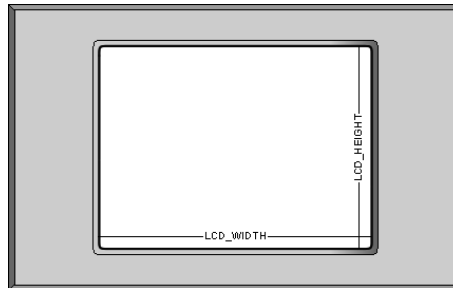


figure 19: Size LCD

The other defines are for the ports and pins for the reset and the backlight.

```
#define PORT_LCD          8
#define PIN_LCD_RESET     5
#define LCD_SHUT_DOWN     0
#define LCD_START_UP      0FFh
#define PIN_LCD_BACKLIGHT 2
#define LCD_BACKLIGHT_ON  0
#define LCD_BACKLIGHT_OFF  0FFh
#define BUSY_TIME_LCD     100    ' time for LCD for starting up
```

## Configuration

Alternatively to the example programs of the Tiger Graphic Library you can determine in the configuration file if the type of your LCD needs a normal or an inverted output. If you do it here you need not do this in the beginning of your program.

```
#ifndef LCD_INVERSION_MODE
#define LCD_INVERSION_MODE 0      ' 0 = normal    for "white" LCD
#endif                          ' 1 = inversion for "blue"  LCD
```

For the types of the touch panel see the table below. For details see the manual of the touch panel device driver.

```
#define TP_TYPE TP_TYP_1
```

No	Typ	Description
0	TP_TYP_0	TP-1000 (use this mode, if you use ANALOG1 or ANALOG2 parallel to the TOUCHPANEL)
1	TP_TYP_1	TP-1000 (stand alone, without other analog device drivers)
2	TP_TYP_2	Graphic Toolkit (use this mode, if you use ANALOG1 or ANALOG2 parallel to the TOUCHPANEL)
3	TP_TYP_3	Graphic Toolkit (stand alone, without other analog device drivers)
4	TP_TYP_4	TEC-1000 (use this mode, if you use ANALOG1 or ANALOG2 parallel to the TOUCHPANEL)
5	TP_TYP_5	TEC-1000 (stand alone, without other analog device drivers)

These are the settings for the eeprom on TP1000.

```
#define TP_CALIB_EEPROM_ADDR 0
#define PORT_EEPROM          7
#define CLK_EEPROM           0
#define DATA_EEPROM         1
#define SPEED_REDUCTION_EEPROM 0
```



## Configuration

From the eeprom address TP\_CALIB\_EEPROM\_ADDR on are 26 bytes reserved for touch panel calibration. Mind this when using the eeprom. For explicit calibration please call vCalibrateTouchpanelFlashForce() which is part of the include file C:\Programme\Wilke Technology\Tiger Basic 5.3\Include using the parameters as follows:

```
call vCalibrateTouchpanelFlashForce( sgTglScreen$, TP, LCD, &  
PORT_EEPROM, CLK_EEPROM, DATA_EEPROM, TP_CALIB_EEPROM_ADDR )  
put #LCD, #0, #UFCO_SET_INV, LCD_INVERSION_MODE ' occasionally changed in  
' vCalibrateTouchpanelFlashForce
```

## Versions

Your applications which are written for older versions of the Tiger Graphic Library can be downgraded without need of reinstalling the old version of the Tiger Graphic Library. If you do not need the actual features of the Tiger Graphic Library just deactivate the defines for later versions in this file and occasionally substitute the configuration file of your project with the actual configuration file for garanting success in compiling.

E.g if you need to compile a program, written with the version V1.00 of the Tiger Graphic Library and you get compiler errors when you compile with the newest version, just deactivate all version defines which are newer than the version the application has been written for:

```
'#define TGL_V1_01  
'#define TGL_V1_13  
'#define TGL_V1_14
```

If you want to ensure to profit of the actual features all defines for the versions must be activated. There could be the need of adjusting the calls of some subroutines.

## General Subroutines

The subroutines described here are for all types of elements.

Subroutine for the initialization of the Tiger Graphic Library

- *vTglInit*

Subroutines for showing and hiding elements

- *bTglShowWindow*
- *bTglHideWindow*
- *bTglShow*
- *bTglHide*
- *bTglShowText*
- *bTglShowLong*
- *bTglShowReal*
- *bTglShowGraph*
- *vTglUpdate*
- *vTglUpdateParams*

Subroutines for modifying and deleting elements

- *bTglLink*
- *bTglDeleteElement*
- *bTglDeleteElementFromWindow*
- *bTglSetSize*
- *bTglGetSize*
- *bTglSetAddress*
- *bTglGetAddress*
- *bTglSetText*
- *bTglSetFont*
- *bTglSetFrame*
- *bTglGetMargins*
- *bTglSetMargins*
- *bTglSetCoordinates*
- *wTglGetCoordinates*
- *bTglSetAttribute*
- *bTglGetAttribute*

Subroutines for controlling LCD and touch panel

- *vTglSetStandbyTime*
- *lTglGetStandbyTime*
- *bTglSetStandbyTermination*
- *bTglSwitchStandby*
- *vTglWaitTouchTp*
- *vTglWaitReleaseTp*
- *bTglGetTouch*
- *wTglGetTouchedElement*
- *wTglGetNumTouchedElements*
- *bTglGetTouchedElementsFlag*
- *vTglBeep*

### vTglInit

call vTglInit()

Function:      Initializes all internal variables of the Tiger Graphic Library and runs internal tasks.

!      Before calling any subroutine of the Tiger Graphic Library call *vTglInit* for initialization.

bTglLink

```
call bTglLink( ElementId1, ElementId2, Result )
```

Function: Links one element to another in order to make a combined element. The new functionality depends on the types of the linked elements (see table).

Parameters:

	B	W	L	S	F	
ElementId1	-	●	-	-	-	unique identifier of first element
ElementId2	-	●	-	-	-	unique identifier of second element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

Element-1	Element-2	Functionality
Slider	Graphic	The graphic is used as the slider button for the slide bar (Slider).
Label	Graphic	The graphic is used as background for the label. The graphic is XORed to the label. The elements 1 and 2 must have the same sizes.
Button	Graphic	The graphic is saved as alternative graphic for the button. If the button is pressed by the user, the alternative graphic is shown, until the button is released again.  If the button is a switch button the alternative graphic is shown as long as the button state is active. The elements 1 and 2 must have the same sizes.
Text button	Label	If the text button is a switch button, the label is shown as long as the button state is active. The label can be designed completely independent from the text buttons text, font and frame. The elements 1 and 2 must have the same sizes.

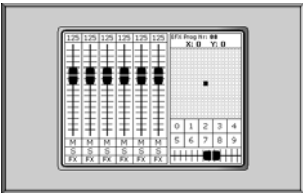


figure 20: Linked elements: slider buttons on sliders



figure 21: Linked elements dark background for user input



figure 22: Linked elements empty and checked boxes

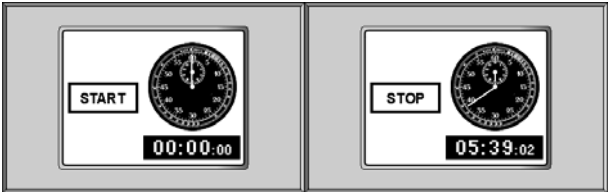


figure 23: Linked element showing alternative texts "start" "stop"

## bTglDeleteElement

call `bTglDeleteElement( ElementId, Result )`

Function: Deletes an element completely after removing it from all windows.

### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element

	B	W	L	S	F	
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For deleting an element from single windows please call *bTglDeleteElementfomWindow*.

For deleting an element temporarily from LCD without deleting the element from the window, please call *bTglHide*.

Sample program:

```

-----
' TGL_bTglDeleteElement.TIG
-----
#define LCD_INVERSION_MODE      0          ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
    datalabel dlButton          ' flash pointer on saved bitmap of button
    byte blReturn               ' return value for TGL subroutines
    word wlElementId

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
    
```



```

call vTglInit()
wElementId = 0

*****
' TGL ELEMENTS AND WINDOWS
*****
call bTglCreateButtonWnd( &
56, 40, &                                ' width, height of element
dlButton, 56, &                          ' bitmap address, width
TGL_KEY_ATTR_DEFAULT, &                  ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, &                 ' identifier of element, window
30, 30, &                                ' x,y coordinate on LCD
041h, &                                  ' key code
blReturn )                               ' return value

*****
' show window
*****
call bTglShowWindow( WINDOW_ID, blReturn )

loop 7FFFFFFFh

'' delete after 2s element from LCD and tgl memorys
wait_duration 2000
call bTglDeleteElement( wElementId, blReturn )

'' recreate after 2 seconds element and show it on LCD
wait_duration 2000
call bTglCreateButtonWnd( &
56, 40, &                                ' width, height of element
dlButton, 56, &                          ' bitmap address, width
TGL_KEY_ATTR_DEFAULT, &                  ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, &                 ' identifier of element, window
30, 30, &                                ' x,y coordinate on LCD
041h, &                                  ' key code
blReturn )                               ' return value
call bTglShow( wElementId, WINDOW_ID, blReturn )
endloop

dlButton::
data filter "num_ok.bmp", "GRAPHFLT", 0      ' OK-button 56x40, bitmap
width 56
end

```

### bTglDeleteElementFromWindow

call `bTglDeleteElementFromWindow( ElementId, WindowId, Result )`

Function: Deletes an element from a window. The element itself will not be deleted.

#### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
Result	●	-	-	-	-	

#### Return Values:

error code, for details see table of error codes  
 0 ok  
 >0 error

For deleting an element completely please call *bTglDeleteElement*.

For deleting an element temporarily from LCD without deleting the element from the window, please call *bTglHide*.

Sample program:

```

-----
' TGL_bTglDeleteElementFromWindow.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  datalabel dlButton          ' flash pointer on saved bitmap of button
  byte blReturn               ' return value for TGL subroutines
  word wElementId             ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

*****

```

```

' INITIALIZATION
*****
call vTglInit()
wElementId = 0

*****
' TGL ELEMENTS AND WINDOWS
*****
call bTglCreateButtonWnd( &
56, 40, &                ' width, height of element
dlButton, 56, &          ' bitmap address, width
TGL_KEY_ATTR_DEFAULT, &  ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, & ' identifier of element, window
30, 30, &                ' x,y coordinate on LCD
041h, blReturn )          ' key code, return value

*****
' show created and deleted elements
*****
call bTglShowWindow( WINDOW_ID, blReturn )

loop 7FFFFFFFh

'' after 2s delete button from LCD and from window
'' but not from tgl memory
wait_duration 2000
call bTglDeleteElementFromWindow( wElementId, WINDOW_ID, blReturn )

'' after 2s replace button in window and show it on LCD
wait_duration 2000
call bTglPlaceButtonInWindow( &
wElementId, WINDOW_ID, & ' identifier of element, window
30, 30, &                ' x,y coordinate on LCD
041h, &                  ' key code
blReturn )                ' return value
call bTglShow( wElementId, WINDOW_ID, blReturn )
endloop

*****
' FLASH
*****
dlButton::
data filter "num_ok.bmp", "GRAPHFLT", 0 ' WxH=56x40, BmpWidth=56
end

```

bTglShowWindow

call bTglShowWindow( WindowId, Result )

Function: Activates all visible elements which are placed in one window, shows them on LCD and activates its functionalities. The displayed window is actual window now.

Parameters:

	B	W	L	S	F	
WindowId	-	●	-	-	-	identifier of the window
Result	●	-	-	-	-	

Return Values:

error code, for details see table of error codes  
0 ok  
>0 error

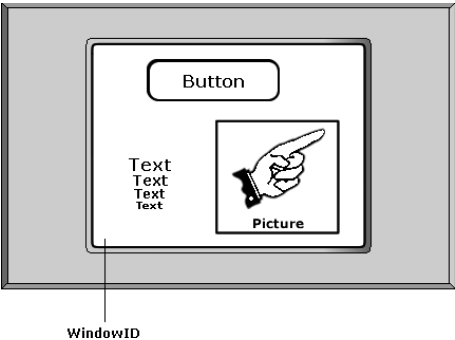


figure 24: Show window

Mind that calling of *bTglShowWindow* will lengthen the stand-by timeout. Never call this subroutine while you are waiting for stand-by.

### bTglHideWindow

call bTglHideWindow()

This function deactivates all elements of the actual window and clears the LCD screen. You can use the LCD for any other tasks. It is possible to create new windows or place new elements in the window. All touch panel functions are disabled.

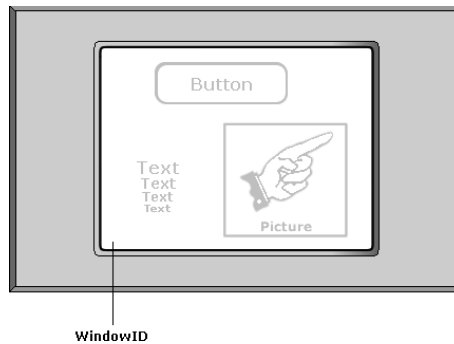


figure 25: Hide window

Mind that the use of the stand-by functions will require an active window.

bTglShow

call bTglShow( ElementId, WindowId, Result )

Function: Makes an element visible in its window and occasionally activates its functionalities.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of the window
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

This subroutine has the effect that a hidden element would be displayed on LCD again if the subroutine *bTglShowWindow* would have been called. If the window is the actual displayed window the LCD will be automatically updated. For hiding an element call the subroutine *bTglHide*.

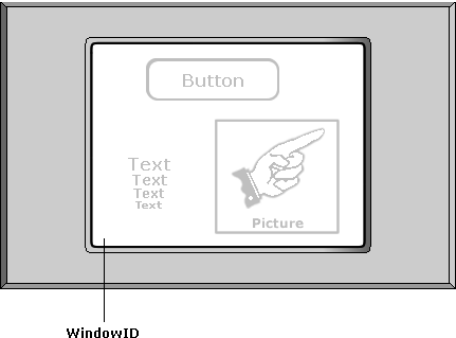


figure 26: All elements hidden

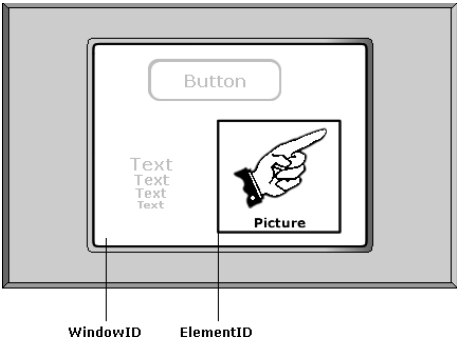


figure 27: One element shown

bTglHide

call bTglHide( ElementId, WindowId, Result )

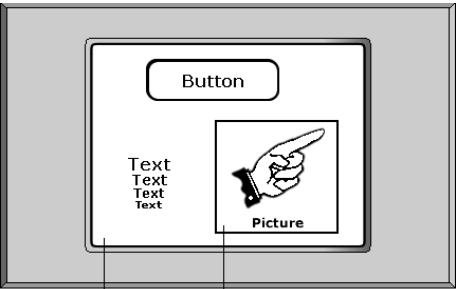
Function: Hides an element in a window and deactivates its functionalities. If the window is the one shown at present.

Parameters:

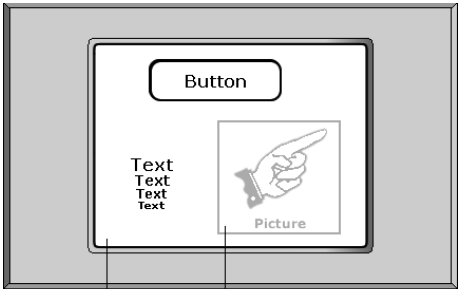
	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of the window
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

This subroutine effects that an element would not be displayed on LCD if the subroutine bTglShowWindow would have been called. If the window is the actual displayed window the LCD will be automatically updated. For displaying an element again call the subroutine bTglShow().



WindowID      ElementID  
figure 28: All elements shown



WindowID      ElementID  
figure 29: One element hidden

Sample program:

```

-----
' TGL_SHOW_HIDE.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

'' identifier of windows
#define WINDOW_ID              0

task main
    datalabel dlButton
    byte blReturn              ' return value of tgl subroutines
    word wlElementId           ' current identifier for creation of elements
    word wlBUTTON_ID           ' identifier if button

    #include TGL_DEVICE_DRIVERS_TP1000.INC

    *****
    ' INITIALIZATION
    *****
    call vTglInit()
    wlElementId = 0

    *****
    ' TGL ELEMENTS AND WINDOWS
    *****
    '' create and place a button without beep
    call bTglCreateButtonWnd( &
    31, 16, &                  ' width, height of element
    dlButton, 32, &            ' address, format width of bitmap
    TGL_KEY_ATTR_BEEP_OFF, &  ' key attributes auto repeat, beep, type
    wlElementId, WINDOW_ID, & ' identifier of element, window
    150, 110, &                ' x, y coordinate on LCD
    0h, &                      ' keycode
    blReturn )                 ' return code (0: OK exit  >0: error exit)

    '' let the button invert if pressed (for this window only)
    call bTglSetAttribute( &
    wlElementId, WINDOW_ID, & ' identifier of element, window
    TGL_INVERT, &             ' attribute to be set
    TGL_INVERTED, &           ' value of attribute
    blReturn )                 ' return code (0: OK exit  >0: error exit)

    '' save the identifier for later hiding and showing
    wlBUTTON_ID = wlElementId

    *****
    ' show and hide button
    *****
    call bTglShowWindow( WINDOW_ID, blReturn )
    while 1=1
        wait_duration 4000
        call bTglHide( &
        wlBUTTON_ID, &         ' identifier of element to be hidden
        WINDOW_ID, &           ' identifier of window with placed in element
        blReturn)              ' return code (0: OK exit  >0: error exit)

```



```
wait_duration 4000
call bTglShow( &
  w1BUTTON_ID, &      ' identifier of element to be shown
  WINDOW_ID, &        ' identifier of window with placed in element
  blReturn)           ' return code (0: OK exit >0: error exit)
endwhile

'*****
' FLASH
'*****

dlButton::
data filter "btn_Solo.bmp", "GRAPHFLT", 0      ' WxH=31x16 BmpWidth=32
end
```

## bTglShowText

call bTglShowText( ElementId, Text, Flag, Result )

Function: Displays a text in the graphic area of a text element like a label or a text button.

### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of label
Text	-	-	-	●	-	text to be displayed
Number	-	-	●	-	-	number to be displayed
Flag	●	-	-	-	-	TRUE=LCD will be updated immediately FALSE=LCD must be updated separately by calling vTglUpdate
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

The subroutine *bTglShowText* runs quite fast. The text is NOT saved permanently for the element. Using this subroutine the text is just displayed on the LCD without saving it. To set a permanent text for the element, please call *bTglSetText*.

! If you want to show variable texts with displaying a new window we suggest to call the subroutine *bTglSetText* for these texts **before** calling the subroutine *bTglShowWindow* even there is no need for saving these texts. If you call the subroutine *bTglShowText* **after** the subroutine *bTglShowWindow* you will see the time difference between the displayed window and the texts.

bTglShowLong, bTglShowWord, bTglShowByte

```
call bTglShowLong( ElementId, LongValue, Flag, Result )
call bTglShowWord( ElementId, WordValue, Flag, Result )
call bTglShowByte( ElementId, ByteValue, Flag, Result )
```

Function: Displays a numeric value in the graphic area of a text element like a label or a text button.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of label
LongValue	-	-	●	-	-	long value to be displayed
WordValue	-	●	-	-	-	word value to be displayed
ByteValue	●	-	-	-	-	byte value to be displayed
Flag	●	-	-	-	-	TRUE=LCD will be updated immediately FALSE=LCD must be updated separately by calling vTglUpdate
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

This subroutine you can use for displaying e.g. measurands on the LCD.

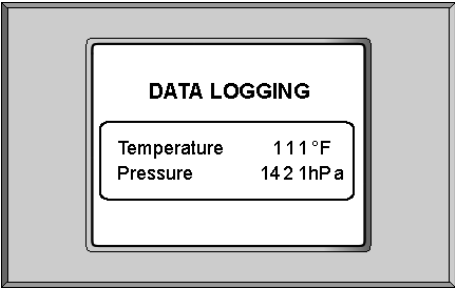


figure 30: Monitoring of Integer numbers

! For a more quiet impression of the displayed number we suggest using constant spacing and horizontal alignment right as font parameters of the text element.

## General Subroutines

The subroutine *bTglShowLong* runs quite fast. The text is NOT saved permanently for the element. Using this subroutine the text is just displayed on the LCD without saving it. To set a permanent text for the element, please call *bTglSetText*.

For more number formats use the Tiger-BASIC commands *stri\$* and *using* with *bTglShowText*.

Sample program:

```
'-----
' TGL_SLIDER_X_SHOW_VALUE.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0
' fonts
#define FONT_ID                0

task main
  datalabel d1SliderButton, d1SlideBar
  byte blReturn                ' return value of tgl subroutines
  word wlElementId             ' current identifier for elements
  word wlSLIDER_ID, wlLABEL_ID ' "constant" identifier for label
  long llSliderValue

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL FONTS
  *****
  call bTglCreateFontParams( &
  FONT_ID, &                  ' identifier of font
  "Valencia",10,"normal", &   ' name, size, type of font
  "right", "center", &        ' alignment horizontal, vertical
  "const center", 0, &        ' spacing type, blank
  -6, 0, &                    ' spacing char, vertical
  "imm", "char", &            ' overlay, wrap mode
  blReturn )                  ' return code (0: OK exit >0: error exit)
```

```

*****
' TGL ELEMENTS AND WINDOWS
*****
' create and place slide bar
call bTglCreatesliderWnd( &
168, 32, & ' width, height of element
-1599, 1600, & ' min, max value main direction
0, 0, & ' min, max value second direction or dummy
"X", & ' type of slider (main direction)
dlsSlideBar, 168, & ' address, format width of bitmap
wElementId, WINDOW_ID, & ' identifier of element, window
76, 120, & ' x, y coordinate on LCD
bIReturn ) ' return code (0: OK exit >0: error exit)
' save identifier for linking and later use
wSLIDER_ID = wElementId
' increment identifier for next element
wElementId = wElementId + 1
' create slider button
call bTglCreateGraphic( &
32, 16, & ' width, height of element
dlsSliderButton, 32, & ' address, format width of bitmap
wElementId, & ' identifier of element
bIReturn ) ' return code (0: OK exit >0: error exit)
' link slider button to slide bar
call bTglLink( &
wSLIDER_ID, & ' identifier of slide bar
wElementId, & ' identifier of slider button
bIReturn ) ' return code (0: OK exit >0: error exit)
' increment identifier for next element
wElementId = wElementId + 1

' label for displaying the slider value
call bTglCreateLabelVarWnd( &
80, 30, & ' width, height of element
FONT_ID, 3, & ' font identifier, frame thickness
wElementId, WINDOW_ID, & ' identifier of element, window
120, 60, & ' x, y coordinate on LCD
bIReturn ) ' return code (0: OK exit >0: error exit)
' save identifier for later use
wLABEL_ID = wElementId
' increment identifier for next element
wElementId = wElementId + 1

*****
' show current slider value
*****
call bTglShowWindow( WINDOW_ID, bIReturn )
loop 7FFFFFFFh
    call lTglGetSliderValue( &
        wSLIDER_ID, WINDOW_ID, & ' identifier of slider, window
        TGL_SL_OPT_VALUE, & ' option: read out current value/position
        lIsliderValue, & ' returned current slider value/position
        bIReturn ) ' return code (0: OK exit >0: error exit)
    call bTglShowLong( wLABEL_ID, lIsliderValue, TRUE, bIReturn )
    wait_duration 100
endloop

*****
' FLASH

```

```
*****  
dlSlideBar::  
data filter "chnl_x_slidebar.bmp", "GRAPHFLT", 0 ' WxH=168x32 BmpW=168  
dlSliderButton::  
data filter "chnl_x_slider_black.bmp", "GRAPHFLT", 0 ' WxH=32x16 BmpW=32  
end
```

bTglShowReal

call bTglShowReal( ElementId, Number, DecPlaces,Flag, Result )

Function: Displays a floating point value in the graphic area of a text element like a label or a text button.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of label
Number	-	-	-	-	●	floating point number to be displayed
DecPlaces	●	-	-	-	-	number of displayed decimal places
Flag	●	-	-	-	-	TRUE=LCD will be updated immediately FALSE=LCD must be updated separately by calling vTglUpdate
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

This subroutine you can use for displaying e.g. measurands on the LCD.



figure 31: Monitoring of floating point numbers

! For a more quiet impression of the displayed number we suggest using constant spacing and horizontal alignment right as font parameters of the text element.

## General Subroutines

The subroutine *bTg/ShowReal* runs quite fast. The text is NOT saved permanently for the element. Using this subroutine the text is just displayed on the LCD without saving it. To set a permanent text for the element, please call *bTg/SetText*.



bTglShowGraph

call bTglShowGraph( GraphicId, Data\$, DataWidth, FirstVal, Num, NumTotal, & LimInf, LimSup, Axis, Result )

Function:      Draws a scaled graph in the area of a graphic

Parameters:

	B	W	L	S	F	
GraphicId	●	-	-	-	-	identifier of a graphic for the drawing area
Data\$	-	-	--	●	-	memory for values to be drawn
DataWidth	●	-	-	-	-	Number of bytes of type of values in Data\$ TGL_BYTE, TGL_WORD, TGL_LONG, TGL_REAL
FirstVal	-	●	-	-	-	number of first value (not byte!) in string to be drawn 0 is number of first value in string
Num	-	●	-	-	-	number of all values (not bytes!) to be drawn 0 take all values in string from the value FirstVal
NumTotal	-	●	-	-	-	1 invalid value, must be at least 2 or 0 number of values in the finished graph can be more than the number of given values is needed for incremental drawing of a graph 0 lengthen graph to whole element width >0 continue graph from FirstVal on expects existing graph of values less FirstVal must be at least FirstVal+Num
LimInf, LimSup	-	-	-	-	●	min/max y-value of graph
Axis	●	-	-	-	-	TGL_NO_AXIS TGL_AXIS TGL_AXIS_SCALE_BINARY TGL_AXIS_SCALE_DECIMAL TGL_AXIS_LINES_H_BINARY TGL_AXIS_LINES_H_DECIMAL TGL_AXIS_LINES_HV_BINARY TGL_AXIS_LINES_HV_DECIMAL
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0    ok >0   error

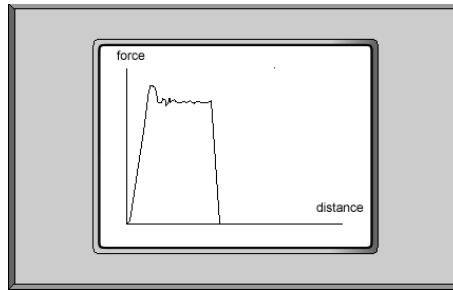


figure 32: Graph

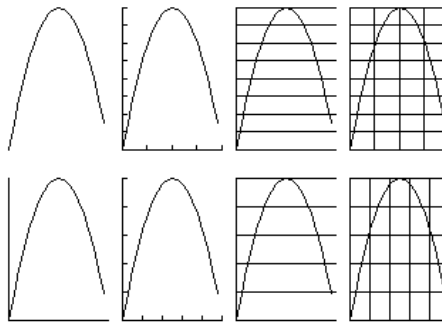


figure 33: axis types

define for axis	description
TGL_NO_AXIS	draw graph without any axis
TGL_AXIS	draw axis without any scale division
TGL_AXIS_SCALE_BINARY	draw axis with binary scale division
TGL_AXIS_SCALE_DECIMAL	draw axis with decimal scale division
TGL_AXIS_LINES_H_BINARY	draw binary horizontal artificial lines
TGL_AXIS_LINES_H_DECIMAL	draw decimal horizontal artificial lines
TGL_AXIS_LINES_HV_BINARY	draw binary horizontal and vertical artificial lines
TGL_AXIS_LINES_HV_DECIMAL	draw decimal horizontal and vertical artificial lines

## General Subroutines

Values can be easily stored in Data\$ by the fuctions NTOS\$ for integer variables and RTOS\$ for floating point variables.

```
' storing integer variables in a string
long llValue
sgData$ = ntos$( sgData$, blDataWidth*blIdx, blDataWidth, llValue )

' storing floating point variables in a string
real rlValue
sgData$ = ntos$( sgData$, TGL_REAL*blIdx, TGL_REAL, rlValue )
```

For an incremental drawing of a graph, e.g. incoming measurands, you must have at least 2 values, for starting with drawing of the graph. After drawing the first part you can add the next value by giving the last 2 values. For additional points you need not to draw the axis anymore. See example TGL\_SHOW\_GRAPH\_incremental.TIG.

```
' draw first 2 values of graph and axis
wlNum      = 2      ' number of values to be actually drawn (>2!)
wlNumTotal = 200     ' total number of expected values
rlLimInf   = 0       ' less than expected min value
rlLimSup   = 250     ' more than expected max value
blAxis     = TGL_AXIS ' draw first values with axis
wlFirstVal = 0
call bTglShowGraph( GRAPHIC_ID, sgData$, blDataWidth, &
wlFirstVal, wlNum, wlNumTotal, rlLimInf,rlLimSup, blAxis, blReturn )

' add secondly one value in graph
blAxis = TGL_NO_AXIS ' axis have been drawn already
for wlFirstVal = 1 to wlNumTotal
  wait_duration 1000
  call bTglShowGraph( GRAPHIC_ID, sgData$, blDataWidth, &
wlFirstVal, wlNum, wlNumTotal, rlLimInf,rlLimSup, blAxis, blReturn )
next
```

For erasing a graph from LCD just call *bTglHide*.

```
' erase graph from LCD
call bTglHide( wlGRAPH_ID, wlWindowId, blReturn )
```

You will get a nice graph when the points have a distance of at least 2 pixels. Otherwise when the points of the graph are too near by each other the graph will have some edges caused by roundings.

Please mind that the graph will not be saved with the element. That means that after reshewing the window the will not be shown unless you have called *bTglShowGraph*.

## General Subroutines

For drawing a graph in a given string as a user graphic instead of drawing in the area of a graphic, please call the subroutine *sTglDrawGraph* described in the chapter *User Graphic*.

vTglUpdate

call vTglUpdate()

Function: Updates the whole internal string on Lcd.  
Occasionally needed after calling *bTglShowText*, *bTglShowLong* or *bTglShowReal*

vTglUpdateParams

call vTglUpdateParams(Y, Height)

Function: Updates selected lines of the internal string on Lcd.  
Occasionally needed after calling *bTglShowText*, *bTglShowLong* or *bTglShowReal*

Parameters:

	B	W	L	S	F	
Y	-	●	-	-	-	offset for the line the LCD shall be updated from number of lines on LCD to be updated
Height	-	●	-	-	-	

bTglSetSize

```
call bTglSetSize( ElementId, Width,Height, Result )
```

Function:      Modify width and height of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Width, Height	-	●	-	-	-	size of element
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0    ok  
>0   error

wTglGetSize

```
call wTglGetSize( ElementId, Width,Height, Result )
```

Function:      Get width and height of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Width, Height	-	●	-	-	-	size of element
Result	●	-	-	-	-	error code, for details see table of error codes

**Return Values:**  
size of element  
error code, for details see table of error codes  
0    ok  
>0   error

bTglSetAddress

call bTglSetAddress( ElementId, Address, Result )

Function: Set new flash address for text or bitmap of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Address	-	-	●	-	-	flash address of text rep. bitmap
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

lTglGetAddress

call lTglSetAddress( ElementId, Address, Result )

Function: Return flash address for text or bitmap of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Address	-	-	●	-	-	flash address of text rep. bitmap
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

bTglSetText

call bTglSetText( ElementId, Text, Result )

Function: Removes old text and sets new text for an element with text. This text is saved permanently for this element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element with text: label textbutton
Text	-	-	-	●	-	new text for element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

This subroutine may be called just for elements with text. These are labels and text buttons.

The new set text will not be displayed on the LCD automatically. For displaying the new text on LCD call the subroutine *bTglShow* for each text element or *bTglShowWindow* for a refresh of all elements in one window.

If you just want to display information without saving, there would be a faster alternative by calling the subroutines *bTglShowText* for texts or *bTglShowLong* resp. *bTglShowReal* for numeric values.

Sample program:

```
'-----
' TGL_bTglSetText.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC
```



```

*****
' IDENTIFIER
*****
' fonts
#define FONT_ID          0
' windows
#define WINDOW_ID        0

task main
  byte blReturn
  word wlElementId      ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  call bTglCreateFontParams( FONT_ID, &          ' font identifier
  "Valencia",10,"normal", &          ' font name, size, type
  "center","center", &          ' alignement horizontal, vertical
  "prop",0,SPACING_CHAR_DEFAULT,0, & ' spacing type,blank,char,vertical
  "imm", "char", &          ' overlay, wrap mode
  blReturn )          ' return value

  call bTglCreateLabelWnd( 100, 40, &' width, height of label
  "text in label", FONT_ID, 5, &' text, font identifier, frame thickness
  wlElementId, WINDOW_ID, &          ' identifier of graphic, window
  110, 100, &          ' x, y coordinate on LCD
  blReturn )          ' return value

  *****
  ' bTglSetText saves text with the element.
  ' The old text will be deleted.
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
  wait_duration 2000
  loop 7FFFFFFFh
    call bTglSetText( wlElementId, "set text", blReturn )
    call bTglShow( wlElementId, WINDOW_ID, blReturn )
    wait_duration 1000
    call bTglSetText( wlElementId, "reset text", blReturn )
    call bTglShow( wlElementId, WINDOW_ID, blReturn )
    wait_duration 1000
  endloop
end

```

bTglSetFont

call bTglSetFont( ElementId, Font, Result )

Function: Set new font of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Font	●	-	-	-	-	font of element

Result	●	-	-	-	-	<b>Return Values:</b>
						error code, for details see table of error codes
						0 ok >0 error

bTglSetFrame

call bTglSetFrame( ElementId, Frame, Result )

Function: Set new frame of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Frame	●	-	-	-	-	frame thickness of element
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

bTglSetMargins

call bTglSetMargins( ElementId, Top,Bottom,Left,Right, Result )

Function: Set margins for the text graphic of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Top,Bottom, Left,Right	-	●	-	-	-	margins of text graphic
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

wTglGetMargins

call wTglGetMargins( ElementId, Top,Bottom,Left,Right, Result )

Function: Return margins for the text graphic of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Top,Bottom, Left,Right	-	●	-	-	-	<b>Return Values:</b> margins of text graphic
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

bTglSetCoordinates

```
call bTglSetCoordinates( ElementId, WindowId, X,Y, Result )
```

Function: Move placed element in a window onto new coordinates.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
X,Y	-	●	-	-	-	coordinates of an element in a window
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

wTglGetCoordinates

```
call wTglGetCoordinates( ElementId, WindowId, X,Y, Result )
```

Function: Returns coordinates of an element in a window.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
X,Y	-	●	-	-	-	coordinates of an element in a window
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

## General Subroutines

Sample program:

```
'-----
' TGL_bTglSetCoordinates_bTglGetCoordinates.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

'' windows
#define WINDOW_ID              1

task main
  datalabel dlButton          ' flash pointer on saved bitmap of button
  byte blReturn               ' return value for TGL subroutines
  word wlElementId            ' current identifier for creation of elements
  word wlX, wlY               ' coordinates on LCD of left top edge of element

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOOWS
  *****
  call bTglCreateButtonWnd( &
    56, 40, &                  ' width, height of element
    dlButton, 56, &             ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &     ' key attributes auto repeat, beep, type
    wlElementId, WINDOW_ID, &   ' identifier of element, window
    30, 30, &                   ' x,y coordinate on LCD
    04lh, &                     ' key code
    blReturn )                  ' return code (0: OK exit >0: error exit)

  *****
  ' show window with element on changing coordinates
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
  loop 7FFFFFFFh
    wait_duration 1000
    call wTglGetCoordinates( wlElementId, WINDOW_ID, wlX, wlY, blReturn )
    wlX = mod( wlX + 56 ,LCD_WIDTH -56)
    wlY = mod( wlY + 40 ,LCD_HEIGHT-40)
    call bTglSetCoordinates( wlElementId, WINDOW_ID, wlX, wlY, blReturn )
  endloop

  *****
  ' FLASH
  *****
  dlButton::
  data filter "num_ok.bmp", "GRAPHFLT", 0      ' WxH=56x40, bmpW=56
end
```

### bTglSetAttribute

call **bTglSetAttribute( ElementId, WindowId, Attribute, Value, Result )**

**Function:** Sets one attribute of an element in a window. The attributes concern the LCD output and touch panel functionality of the elements. For details see table below.

#### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of the window
Attribute	-	-	●	-	-	attribute of element (see table below)
Value	-	-	●	-	-	value of attribute (see table below)
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

#### Attributes for all elements

Attribute	Value	Description
TGL_ATTR_INV_STATE	TGL_NORMAL	show element in a NOT inverted state
	TGL_INVERTED	show element in an inverted state
TGL_ATTR_VISIBLE	TGL_HIDDEN	hide element from LCD occasionally deactivate touch panel functionality
	TGL_VISIBLE	show element on LCD occasionally activate touch panel functionality
TGL_ATTR_SHOW_MODE	TGL_SHOW_MODE_COPY	standard mode: graphic is copied to LCD screen
	TGL_SHOW_MODE_OR	element is ORed to LCD screen. This is used e.g. for background graphics.
	TGL_SHOW_MODE_AND	element is ANDed to LCD screen
	TGL_SHOW_MODE_XOR	element is XORed to LCD screen.

Attribute	Value	Description
TGL_ATTR_ROTATE	TGL_ROTATE_0	rotates bitmap resp. text graphic of element 0° to the right
	TGL_ROTATE_90	rotates bitmap resp. text graphic of element 90° to the right
	TGL_ROTATE_180	rotates bitmap resp. text graphic of element 180° to the right
	TGL_ROTATE_270	rotates bitmap resp. text graphic of element 270° to the right
TGL_ATTR_BLINK_MODE	TGL_BLINK_MODE_WHITE	change element with white area
	TGL_BLINK_MODE_BLACK	change element with black area
	TGL_BLINK_MODE_INVERT	invert element
	TGL_BLINK_MODE_ALTGRA	change bitmap/text with alternative bitmap/text
TGL_ATTR_BLINK_SPEED	TGL_BLINK_SPEED_OFF	switch off blinking
	TGL_BLINK_SPEED_SLOW	blink slow
	TGL_BLINK_SPEED_MID	blink in mid speed
	TGL_BLINK_SPEED_FAST	blink fast

Attributes for buttons and textbuttons only

Attribute	Value	Description
TGL_ATTR_SWITCH_STATE	TGL_STANDARD or TGL_INACT	set switch in inactive state
	TGL_ALTERNATIVE or TGL_ACT	set switch in active state
TGL_ATTR_INVERT	TGL_TRUE	invert push button for hold pressing invert switch in active state
	TGL_FALSE	no inversion for pressed push button or switch in active state



## General Subroutines

Attributes can be set directly after the creation of an element. You can set different attributes for the same element in different windows. The following code example creates a button which will invert during pressing.

```
call bTglCreateButtonWnd( &
31, 16, &                ' width, height of element
BUTTON, 32, &            ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, &  ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, & ' identifier of element, window
150, 110, &              ' x, y coordinate on LCD
0h, &                    ' keycode
blReturn )               ' return code (0: OK exit >0: error exit)

' let the button invert if it is pressed
call bTglSetAttribute( &
wElementId, WINDOW_ID, & ' identifier of element, window
TGL_ATTR_INVERT, TGL_TRUE,& ' attribute, value of attribute
blReturn )               ' return code (0: OK exit >0: error exit)
```

If you want to change the attributes of an element later you need to save the identifier of the element in a word variable after creation.

```
call bTglCreateGraphicWnd( &
31, 16, &                ' width, height of element
dlGraphicAddr, 32, &      ' address, format width of bitmap
wElementId, WINDOW_ID, & ' identifier of element, window
150, 110, &              ' x, y coordinate on LCD
blReturn )               ' return code (0: OK exit >0: error exit)

wlGRAPHIC_ID = wElementId ' save identifier of the element

' ' ...                    ' some code

' invert the graphic now
call bTglSetAttribute( &
wlGRAPHIC_ID, WINDOW_ID, & ' identifier of element, window
TGL_ATTR_INV_STATE, TGL_INVERTED,& ' attribute, value of attribute
blReturn )               ' return code (0: OK exit >0: error exit)
```

! Please mind that TGL\_ATTR\_INV\_STATE is a static attribute which set the inversion state for all elements. TGL\_ATTR\_INVERT is a dynamic attribute for the inversion of a pressed button!

When using the attribute TGL\_ROTATE please mind that the size of the element must be equal the size of the rotated bitmap. E.g you have a bitmap WxH=88x27 and want to rotate it for 90° to the right. The size of the element has the rotated values WxH=27x88. The format width of the bitmap must be given as it is saved in the flash memory.

```
dlHelloGraphic::
data filter "HelloGraphic.bmp", "GRAPHFLT", 0 ' WxH=88x27, bmpWidth=88

call bTglCreateGraphicWnd( &
27, 88, & ' width, height of element
dlHelloGraphic,88, & ' address, format width of bitmap
wElemenId,WINDOW_ID, & ' identifier of graphic, window
146, 135, & ' x,y coordinate on LCD
blReturn ) ' return code (0: OK exit >0: error exit)

call bTglSetAttribute( & '
wElemenId, WINDOW_ID, & ' identifier element, window
TGL_ROTATE, TGL_ROTATE_90, & ' attribute of element and its value
blReturn ) ' return code (0: OK exit >0: error exit)
```

With the attribute TGL\_ROTATE you can easily create rotated text graphics. Setting this attribute the LCD can be used edgewise as well as in landscape format.

The attribute TGL\_ATTR\_SHOW\_MODE determines how the element will be copied on LCD. Default is TGL\_SHOW\_MODE\_COPY. The elements will overlay the other elements in the order of their types and identifiers. If you want to see all elements which are overlaying each other you need to set the attributes TGL\_SHOW\_MODE\_OR or TGL\_SHOW\_MODE\_XOR. The Attribute TGL\_SHOW\_MODE\_AND can be used to hide a certain part of an other element.

The attributes TGL\_ATTR\_BLINK\_SPEED and TGL\_ATTR\_BLINK\_MODE can be set at any time. The new attribute will shown directly on LCD. Default for these attributes are TGL\_BLINK\_SPEED\_OFF and TGL\_BLINK\_MODE\_WHITE. If you want to let an element blink you need to set the attribute for speed.

The attribute TGL\_ATTR\_SWITCH\_STATE can be set by the subroutine bTglSetButtonState, too.

The attribute TGL\_ATTR\_VISIBLE can be set by the subroutines bTglShow and bTglHide, too.

ITglGetAttribute

call ITglGetAttribute( ElementId, WindowId, Attribute, Value, Result )

Function: Get value of an attribute of one attribute of a graphic, label, button or text button in a window. The attributes concern the layout of the elements. The value of the attribute determines if an element is displayed inverted, the rotation of its bitmap resp. its text graphic and the copy mode.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of the window
Attribute	-	-	●	-	-	attribute of element

	B	W	L	S	F	Return Values:
Value	-	-	●	-	-	attribute value
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

For details of the return values see subroutine *bTglSetAttribute* .

Sample program:

```
'-----
' TGL_BUTTON_beep_off_inverted.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
***** windows
#define WINDOW_ID      0

task main
  datalabel d1Button
  byte blReturn      ' return value of tgl subroutines
  word w1ElementId   ' current identifier for creation of elements
  word w1IbuFill     ' filling of the touch panel input buffer
  byte blKeycode     ' returned keycode
```

```
#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'  INITIALIZATION
'*****
call vTglInit()
wElementId = 0

'*****
'  TGL ELEMENTS AND WINDOWS
'*****
' create button without beep
call bTglCreateButtonWnd( &
31, 16, &                                ' width, height of element
dlButton, 32, &                          ' address, format width of bitmap
TGL_KEY_ATTR_BEEP_OFF, &                ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, &                ' identifier of element, window
150, 110, &                             ' x, y coordinate on LCD
0h, &                                   ' keycode
blReturn )                               ' return code (0: OK exit  >0: error exit)

' let the button invert if it is pressed
call bTglSetAttribute( &
wElementId, WINDOW_ID, &                ' identifier of element, window
TGL_INVERT, TGL_INVERTED, &            ' attribute, value of attribute
blReturn )                               ' return code (0: OK exit  >0: error exit)

'*****
' show button and get its keycode for further processing
'*****
call bTglShowWindow( WINDOW_ID, blReturn )
while 1=1
  get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill ' get buffer length
  if wIbuFill > 0 then                     ' check length of input buffer
    get #TP, #0, 1, blKeycode ' get keycode
  endif
endwhile

'*****
'  FLASH
'*****
dlButton::
data filter "btn_Solo.bmp", "GRAPHFLT", 0      ' WxH=31x16  bmpW=32
end
```

### vTglSetStandbyTime

call vTglSetStandbyTime( Seconds )

Function: Sets the time in seconds until the LCD changes into stand-by mode after the last touch on the panel.

#### Parameters:

	B	W	L	S	F	
Seconds	-	-	●	-	-	stand-by time in seconds 0=never change to stand-by mode

For stopping stand-by watching set standby-time to 0.

! Mind that the Tiger Graphic Library uses the system ticks to watch the stand-by mode. This means it is never allowed to use the function *set\_ticks()* with the library.

! Mind that calling of *bTglShowWindow* will lengthen the stand-by timeout. Never call this subroutine while you are waiting for stand-by.

ITglGetStandbyTime

call ITglGetStandbyTime( Seconds )

Function: Get the currently time in seconds until the LCD changes into stand-by mode after the last touch on the panel.

Parameters:

	B	W	L	S	F
Seconds	-	-	●	-	-

Return Values:  
current stand-by time of the system in seconds

## General Subroutines

Sample program:

```
-----
' TGL_STANDBY.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue" LCD

#include TigerGraphicLibrary.INC
*****
' IDENTIFIER
*****
' elements
#define BUTTON_ID              0
' windows
#define WINDOW_ID              1

task main
  datalabel dlButton           ' flash pointer on saved bitmap of button
  byte blReturn                ' return value for TGL subroutines
  word wlx, wly

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()

  *****
  ' TGL ELEMENTS AND WINDOOWS
  *****
  call bTglCreateButtonWnd( &
    56, 40, &                  ' width, height of element
    dlButton, 56, &             ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &     ' key attributes auto repeat, beep, type
    BUTTON_ID, WINDOW_ID, &     ' identifier of element, window
    104, 80, &                  ' x,y coordinate on LCD
    041h, &                     ' key code
    blReturn )                  ' return code (0: OK exit  >0: error exit)

  *****
  ' show window in stand-by mode
  *****
  call vTglSetStandbyTime(5) ' set stand-by time of 5s
  call bTglShowWindow( WINDOW_ID, blReturn )

  *****
  ' FLASH
  *****
  dlButton::
    data filter "num_ok.bmp", "GRAPHFLT", 0 ' WxH=56x40, bitmap width 56
end
```

bTglSetStandbyTermination

call bTglSetStandbyTermination( Mode, Result )

Function: Stop stand-by mode by event, e.g. an active device driver.

Parameters:

B

W

L

S

F

●

-

-

-

-

Mode

enables resp disables termination of stand-by mode by an event

●

-

-

-

-

Result

Return Values:

error code, for details see table of error codes

0 ok

>0 error

Mode	Functionality
TGL_STANDBY_TERM_LCD_ON	Stand-by mode will be terminated by active LCD driver
TGL_STANDBY_TERM_LCD_OFF	Stand-by mode will NOT be terminated by active LCD driver anymore

Sample program:

```
'-----
' TGL_STANDBY_LCD_terminated.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion  for "blue"  LCD

#include TigerGraphicLibrary.INC
'*****
' IDENTIFIER
'*****
' windows
#define WINDOW_ID              1

task main
  datalabel dlButton      ' flash pointer on saved bitmap of button
  byte blReturn           ' return value for TGL subroutines
```



```

word wIbuFill      ' filling of input buffer
word wElementId    ' current identifier for creation of elements
word wlBUTTON_ID   ' "constant" identifier of button

#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
' INITIALIZATION
'*****
call vTglInit()
wElementId = 0

'*****
' TGL ELEMENTS AND WINDOWS
'*****
wlBUTTON_ID = wElementId      ' save identifier of button for further use
call bTglCreateButtonWnd( &
56, 40, &                    ' width, height of element
dlButton, 56, &              ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, &     ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, &    ' identifier of element, window
104, 80, &                  ' x,y coordinate on LCD
041h, &                     ' key code
blReturn )                  ' return code (0: OK exit >0: error exit)

'*****
' show window in stand-by mode
'*****
call bTglShowWindow( WINDOW_ID, blReturn )

loop 7FFFFFFFh
'' set stand-by termination by LCD
call bTglSetStandbyTermination( TGL_STANDBY_TERM_LCD_ON, blReturn )

'' activate stand-by mode after button press
get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
while 0 = wIbuFill
    release_task
    get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
endwhile
call bTglSwitchStandby( TGL_ON, blReturn )

'' hide button and
'' deactivate stand-by mode with busy LCD after 5s
wait_duration 5000
call bTglHide( wlBUTTON_ID, WINDOW_ID, blReturn )

'*****

'' set stand-by termination by LCD off
call bTglSetStandbyTermination( TGL_STANDBY_TERM_LCD_OFF, blReturn )

'' activate stand-by mode after 5s
wait_duration 5000
call bTglSwitchStandby( TGL_ON, blReturn )
'' show button and
'' do NOT deactivate stand-by mode whether LCD is busy after 5s
wait_duration 5000
call bTglShow( wlBUTTON_ID, WINDOW_ID, blReturn )

```

```
'' wait for button press
'' -> stand-by mode must be finished by touch before!
get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
while 0 = wIbuFill
  release_task
  get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
endwhile

endloop

*****
' FLASH
*****
dlButton::
data filter "num_ok.bmp", "GRAPHFLT", 0 ' WxH=56x40, bitmap width 56
end
```

### bTglSwitchStandby

call bTglSwitchStandby( Flag, Result )

Function: Start or stop stand-by mode by command.

#### Parameters:

	B	W	L	S	F	
Flag	●	-	-	-	-	TGL_ON: start stand-by mode TGL_OFF: stop stand-by mode

	B	W	L	S	F	
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

Sample program:

```

-----
' TGL_STANDBY_switch.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD

#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              1

task main
    datalabel dlButton          ' flash pointer on saved bitmap of button
    byte blReturn               ' return value for TGL subroutines
    word wIbuFill               ' filling of input buffer
    word wElementId             ' current identifier for creation of elements

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
call vTglInit()
wElementId = 0
    
```

```

*****
' TGL ELEMENTS AND WINDOOWS
*****
call bTglCreateButtonWnd( &
56, 40, &                ' width, height of element
dlButton, 56, &          ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, &  ' key attributes auto repeat, beep, type
wElementId, WINDOW_ID, & ' identifier of element, window
104, 80, &              ' x,y coordinate on LCD
041h, &                 ' key code
blReturn )              ' return code (0: OK exit  >0: error exit)

*****
' show window in stand-by mode
*****
call bTglShowWindow( WINDOW_ID, blReturn )

loop 7FFFFFFFh

    '' activate stand-by mode after button press
    get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
    while 0 = wIbuFill
        release_task
        get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
    endwhile
    call bTglSwitchStandby( TGL_ON, blReturn )

    '' deactivate stand-by mode after 5s
    wait_duration 5000
    call bTglSwitchStandby( TGL_OFF, blReturn )
endloop

*****
' FLASH
*****
dlButton::
data filter "num_ok.bmp", "GRAPHFLT", 0 ' WxH=56x40, bitmap width 56
end

```

## vTglWaitTouchTp

call vTglWaitTouchTp()

Function: Wait for next touch on touch panel.

## vTglWaitReleaseTp

call vTglWaitReleaseTp()

Function: Wait for release of touch panel.

## bTglGetTouch

call bTglGetTouch(Flag)

Function: Return actual state if touch panel is touched.

### Parameters:

	B	W	L	S	F	
Flag	●	-	-	-	-	<b>Return Values:</b> TGL_TRUE: actually touched TGL_FALSE: actually released

## wTglGetTouchedElement

call `wTglGetTouchedElement( ElementId )`

Function: Get identifier from buffer of touched switches, sliders or listboxes.

### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	<b>Return Values:</b> identifier of touched element TGL_NO_ID: empty buffer

This buffer buffers the identifiers of internally administrated touch elements only  
Push buttons will not be buffered here.

Buffer works like first in first out buffer. In case of overflow the oldest entries will  
be overwritten. Buffer will be erased with window changing.

For exaple code see `TGL_wTglGetTouchedElement.TIG`.

## wTglGetNumTouchedElements

call wTglGetNumTouchedElements( Number )

Function: Get buffer filling for touched switches, sliders or listboxes.

### Parameters:

	B	W	L	S	F	Return Values:
Number	-	●	-	-	-	number of touched elements

Buffer works like first in first out buffer. In case of overflow the oldest entry will be overwritten. Buffer will be erased with window changing.

## bTglGetTouchedElementsFlag

call bTglGetTouchedElementsFlag( Flag )

Function: Return and reset buffer overflow flag for touched switches, sliders or listboxes.

### Parameters:

	B	W	L	S	F	
Flag	-	●	-	-	-	<b>Return Values:</b> TGL_TRUE: buffer overflow TGL_FALSE: no buffer overflow

Last entrys will be overwritten in case of overflow.



## vTglBeep

call bTglBeep( Number )

Function:      Beeps a number of times. (Tiger 2 only)

### Parameters:

	<b>B</b>	<b>W</b>	<b>L</b>	<b>S</b>	<b>F</b>	
Number	●	-	-	-	-	number of beeps

## Graphic

A graphic is a rectangular element placed in a window. The graphic is filled with a bitmap which is stored in the flash memory.

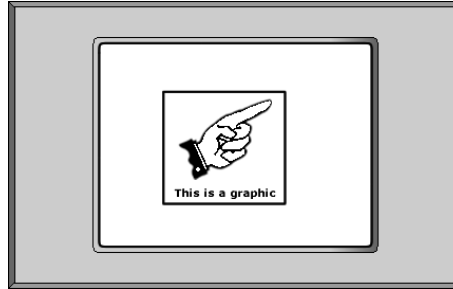


figure 34: Graphic

Subroutines:

- bTglCreateGraphic
- bTglPlaceGraphicInWindow
- bTglDockGraphicInWindow
- bTglCreateGraphicWnd
- bTglCreateGraphicDockWnd

Graphics can be used as extensions for other elements. They have to be linked with the subroutine *bTglLink*. A graphic linked with a slider is used as a slider button for the slidebar. If the slider value has changed, the slider is shown on the current position. It visualizes the current position for the user. The graphic element defines the background of a label or is used as alternative graphic of a button. If the button is pressed, the alternative graphic is shown on LCD.

Element	Functionality
Slider	The graphic is used as the variable slider of the slide bar (slider button).
Label	The graphic is used as background for the label. The graphic is XORed to the label.
Button	The graphic is saved as alternative graphic for the button. If the button is pressed by the user, the alternative graphic is shown, until the button is released again

bTglCreateGraphic

call bTglCreateGraphic( Width, Height, BmpAddr, BmpWidth, ElementId, Result )

Function: Creates a new bitmap graphic by storing its attributes width, height, bitmap address and bitmap width.

Parameters:

	B	W	L	S	F	
Width	-	●	-	-	-	width of graphic
Height	-	●	-	-	-	height of graphic
BmpAddr	-	-	●	-	-	address of the graphic bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the graphic in flash memory must be a multiple of 8
ElementId	-	●	-	-	-	unique identifier of this element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

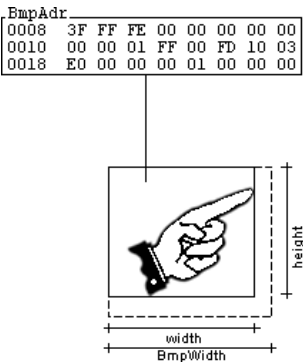


figure 35: Create graphic

bTglPlaceGraphicInWindow

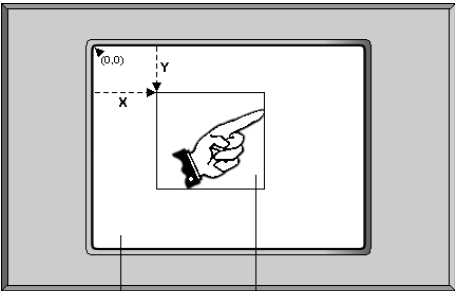
call bTglPlaceGraphicInWindow( ElementId, WindowId, X,Y, Result )

Function: Places a graphic in a window at position XY.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error



WindowID GraphicID  
figure 36: Place graphic in window

bTglDockGraphicInWindow

call bTglDockGraphicInWindow( ParentId, ChildId, WindowId, XRel, YRel, Option, & Result )

Function: Places a graphic in a window by docking the child element next to the parent element in one of 8 directions.

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

bTglCreateGraphicWnd

call bTglCreateGraphicWnd( Width, Height, BmpAddr, BmpWidth, ElementId, & WindowId, X,Y, Result )

Function: Creates a graphic and places it to a window at position XY.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of graphic
BmpAddr	-	-	●	-	-	address of the graphic bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the graphic in flash memory must be a multiple of 8
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

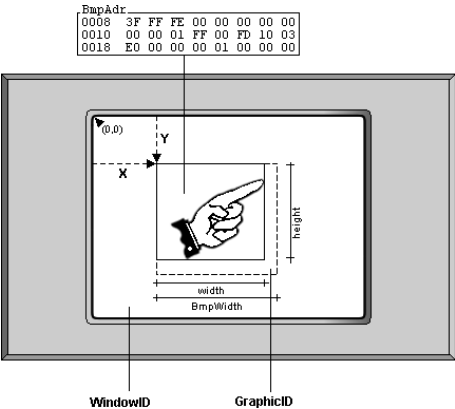


figure 37: Create and place graphic in a window

bTglCreateGraphicDockWnd

call bTglCreateGraphicDockWnd( Width, Height, BmpAddr, BmpWidth ParentId, & ChildId, WindowId, XRel, YRel, Option, Result )

Function: Creates a graphic and places it in a window by docking the child element next to the parent element in one of 8 directions.

Parameters:

	B	W	L	S	F	
Width	-	●	-	-	-	width of graphic
Height	-	●	-	-	-	height of graphic
BmpAddr	-	-	●	-	-	address of the graphic bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the graphic in flash memory must be a multiple of 8
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

Sample program:

```
'-----
' TGL_GRAPHIC_createWnd.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

'*****
' IDENTIFIER
'*****
' windows
#define WINDOW_ID              0

task main
  datalabel dlHelloGraphic
  byte blReturn                ' return value of the tgl subroutines
  word wElementId              ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '*****
  ' INITIALIZATION
  '*****
  call vTglInit()
  wElementId = 0

  '*****
  ' TGL ELEMENTS AND WINDOWS
  '*****
  call bTglCreateGraphicWnd( &
    88, 27, &                  ' width, height of element
    dlHelloGraphic, 88, &      ' address, format width of bitmap
    wElementId, WINDOW_ID, &   ' identifier of element, window
    116, 105, &                ' x,y coordinate on LCD
    blReturn )                 ' return code (0: OK exit >0: error exit)

  '*****
  ' show window
  '*****
  call bTglShowWindow( WINDOW_ID, blReturn )

  '*****
  ' FLASH
  '*****
  dlHelloGraphic::
  data filter "HelloGraphic.bmp", "GRAPHFLT", 0 ' WxH=88x27 BmpWidth=88end
```



## Label

A label is a rectangular element placed in a window. The label is filled by a text graphic. The text can be stored in flash or in string. Variable labels have no stored text. The text for these elements is given by a parameter of the subroutine *bTglShowText*. Optionally labels can be framed with variable frame thickness.

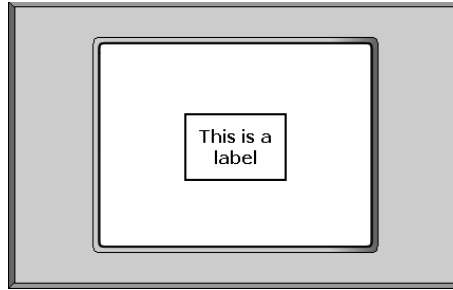


figure 38: Label

### Subroutines

- bTglCreateLabel
- bTglCreateLabelF
- bTglCreateLabelVar
- bTglPlaceLabelInWindow
- bTglCreateLabelWnd
- bTglCreateLabelFWnd
- bTglCreateLabelVarWnd

The text on text buttons is written with a graphic font. You have to create this font before using this element. Please see in chapter *Graphic Fonts* the subroutines *bTglCreateFont* or *bTglCreateFontParams* for creating your own fonts.

## bTglCreateLabel

```
call bTglCreateLabel ( Width, Height, Text$, FontId, Frame, ElementId, Result )
call bTglCreateLabelF ( Width, Height, TextAddr, FontId, Frame, ElementId, Result )
call bTglCreateLabelVar ( Width, Height, FontId, Frame, ElementId, Result )
```

Function: Creates a new label.

**bTglCreateLabel** Text is given as a constant parameter  
**bTglCreateLabelF** Text is passed by a flash address.  
 The address points on a 4 byte text length followed by the text itself  
**bTglCreateLabelVar** No Text is passed in this subroutine. Text can be displayed in elements area with *bTglShowText*.  
 Text can be saved with this element by calling the subroutine *bTglSetText*

### Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of label
Text\$	-	-	-	●	-	text in label
TextAddr	-	-	●	-	-	address of the 4 byte text length and text in flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame
ElementId	-	●	-	-	-	unique identifier of this element

### Return Values:

	B	W	L	S	F	
Result	●	-	-	-	-	error code, for details see table of error codes

0 ok  
 >0 error

The label will not be created, if the text graphic does not fit in the label. The fitting of the text graphic in the label depends on the chosen font, the text length and the frame thickness.

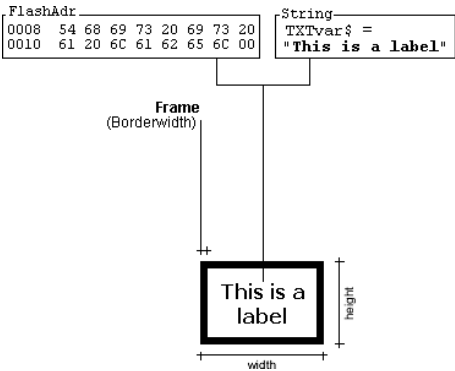


figure 39: Create a label

bTglPlaceLabelInWindow

call bTglPlaceLabelInWindow ( ElementId, WindowId, X,Y, Result )

Function: Places label in a window at position XY.  
For relative positioning to existing elements see subroutine *bTglDockLabelInWindow*

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of top left edge in window
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

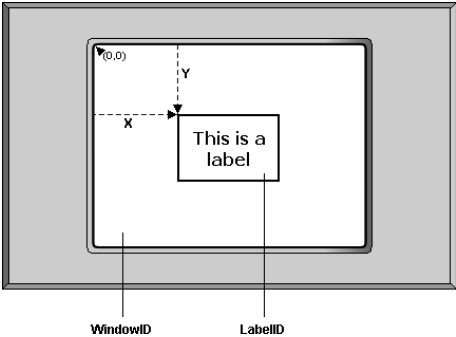


figure 40: Place label in a window

bTglDockLabelInWindow

call bTglDockLabelInWindow( ParentId, ElementId, WindowId, XRel,YRel, Option, Result )

Function: Places a label in a window by docking the new label as child element next to the existing parent element in one of 8 directions.  
For absolute positioning in window see subroutine *bTglPlaceLabelInWindow*

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of new element (child)
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

bTglCreateLabelWnd

```
call bTglCreateLabelWnd( Width, Height, Text, FontId, Frame, ElementId, &
                          WindowId, X,Y, Result )
call bTglCreateLabelFWnd( Width, Height, TextAddr,FontId, Frame, ElementId, &
                          WindowId, X,Y, Result )
call bTglCreateLabelVarWnd(Width, Height, FontId, Frame, ElementId, &
                           WindowId, X,Y, Result )
```

Function: Creates a label and places it to a window at global position XY.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of label
Text	-	-	-	●	-	text in label
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame
ElementId	-	●	-	-	-	unique identifier of new element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates in window
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

! The label will not be created, if the text graphic does not fit in the label. The fitting of the text graphic in the label depends on the choosen font, the text length and the frame thickness.

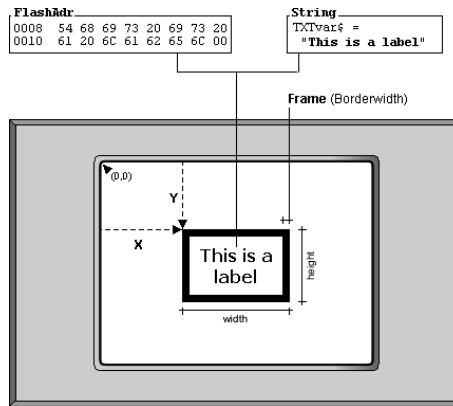


figure 41: Create and place label in a window

bTglCreateLabelDockWnd

```
call bTglCreateLabelDockWnd( Width, Height, Text, FontId, Frame, ParentId, &
                             ElementId, WindowId, XRel, YRel, Option, Result )
call bTglCreateLabelFDockWnd( Width, Height, TextAddr, FontId, Frame, ParentId, &
                              ElementId, WindowId, XRel, YRel, Option, Result )
call bTglCreateLabelVarDockWnd(Width, Height, FontId, Frame, ParentId &
                               ElementId, WindowId, XRel, YRel, Option, Result )
```

Function: Places a label in a window by docking the new label as child element next to the existing parent element in one of 8 directions.  
For absolute positioning in window see subroutine bPlaceLabelInWindow

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of label
Text	-	-	-	●	-	text in label
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of new element (child)
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT

Return Values:

Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error



For details about the docking subroutines please see the chapter *Docking*.

! The label will not be created, if the text graphic does not fit in the label. The fitting of the text graphic in the label depends on the choosen font, the text length and the frame thickness.

## Label

Sample program:

```
-----
' TGL_LABEL_createWnd.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0
' fonts
#define FONT_ID                0

task main
  byte blReturn                ' return value of tgl subroutines
  word wElementId              ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wElementId = 0

  *****
  ' TGL FONTS
  *****
  call bTglCreateFontParams( &
  FONT_ID, &                  ' identifier of font
  "Valencia", 10, "normal", & ' name, size, type of font
  "center", "center", &      ' alignment horizontal, vertical
  "prop", 0, &                ' spacing type, blank
  SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
  "imm", "char", &           ' overlay, wrap mode
  blReturn )                  ' return code (0: OK exit  >0: error exit)

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  call bTglCreateLabelWnd( &
  160, 40, &                  ' width, height of element
  "Hello Label", &            ' text in element
  FONT_ID, 5, &               ' font identifier, frame thickness
  wElementId, WINDOW_ID, &    ' identifier of element, window
  80, 100, &                  ' x, y coordinate on LCD
  blReturn )                  ' return code (0: OK exit  >0: error exit)

  *****
  ' show window
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
end
```

# Buttons

Buttons are rectangular areas on LCD containing a bitmap and touch panel functionality. There are buttons containing texts instead of bitmaps. These elements are called text buttons. See the chapter *Text Buttons* for this kind of a button.

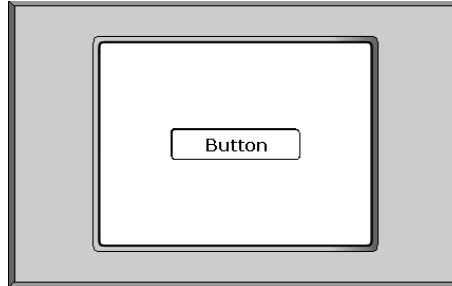


figure 42: Button

There are several features available for buttons. For monitoring a pressed button its bitmap can be inverted, or the button can give a beep. A nice feature is giving the button an alternative bitmap. This way a button with a 3D effect can be realized.



figure 43: Unpressed button bitmap



figure 44: Pressed button bitmap

Buttons can be used as switches. The state can be read easily by calling a function.

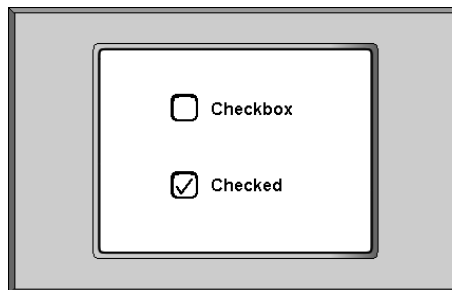


figure 45: Checkbox

Available subroutines for buttons:

- *bTglCreateButton*
- *bTglPlaceButtonInWindow*
- *bTglDockButtonInWindow*
- *bTglCreateButtonWnd*
- *bTglCreateButtonDockWnd*
- *bTglGetButtonState*
- *bTglSetButtonState*
- *bTglGetPushButtonState*
- *bTglGetKeycode*
- *bTglWaitKeycode*

See also:

- *bTglLink*
- *bTglSetAttribute*

bTglCreateButton

call bTglCreateButton(Width, Height, BmpAddr, BmpWidth, KeyAttr, ElementId, & Result )

Function:      Creates a new bitmap button.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the button in flash memory must be a multiple of 8
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0    ok >0 error

This is the first step to create a button or a switch button. By creating a button, the attributes of this element are saved and you can place the button in several windows. The next step is *bTglPlaceButtonInWindow* or *bTglDockButtonInWindow* to use the button in a window. You can perform these 2 steps in one function with *bTglCreateButtonWnd* or *bTglCreateButtonDockWnd*. The docking functions are for relative positioning to existing elements in the window.

To create a switch button, please set bit 6 from the variable KeyAttr.

## Buttons

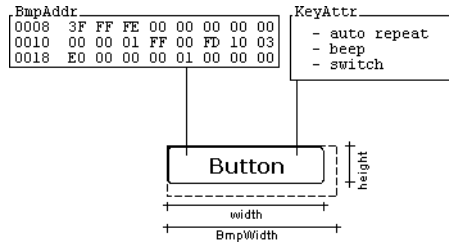


figure 46: Create button

Buttons can have different key attributes. These attributes are passed in the subroutines *bTglCreateButton*, *bTglCreateButtonWnd* or *bTglCreateButtonDockWnd* with parameter *KeyAttr*. For easy use pass these defines to the create functions

Attribute	Functionality
TGL_KEY_ATTR_AUTOREPEAT	(No) Filling the buffer using auto repeat mode
TGL_KEY_ATTR_NO_AUTOREPEAT	
TGL_KEY_ATTR_BEEP	Button generate (not) a beep (key click)
TGL_KEY_ATTR_NO_BEEP	
TGL_KEY_ATTR_STANDARD	Button is used as a push button (standard button)
TGL_KEY_ATTR_SWITCH	or switch button(e.g. Checkbox)
TGL_KEY_ATTR_DEFAULT	autorepeat on, beep on, push button

Create a button with default key attributes:

```
call bTglCreateButtonWnd(
31, 16, &           ' button size
dlButton, 32, &      ' address and format width of bitmap
TGL_KEY_ATTR_DEFAULT, & ' key attributes
wlElementId, &       ' unique identifier of this button
blReturn )           ' Return Code (0: OK Exit >0: Error Code)
```

Create switch button:

```
call bTglCreateButtonWnd(
31, 16, &           ' button size
dlButton, 32, &      ' address and format width of bitmap
TGL_KEY_ATTR_SWITCH, & ' key attributes
wlElementId, &       ' unique ID of this button
blReturn )           ' Return Code (0: OK Exit >0: Error Code)
```

## Buttons

Turn off beep:

```
call bTglCreateButtonWnd(  
31, 16, &          ' button size  
dlButton, 32, &    ' address and format width of bitmap  
TGL_KEY_ATTR_NO_BEEP, & ' key attributes  
wlElementId, &    ' unique ID of this button  
blReturn )        ' Return Code (0: OK Exit >0: Error Code)
```

Turn off auto repeat:

```
call bTglCreateButtonWnd(  
31, 16, &          ' button size  
dlButton, 32, &    ' address and format width of bitmap  
TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes  
wlElementId, &    ' unique ID of this button  
blReturn )        ' Return Code (0: OK Exit >0: Error Code)
```

For a combination of key attributes the defines can be ORed in a variable and passed afterwards.

Create Switch button without beep:

```
blKeyAttr = TGL_KEY_ATTR_SWITCH bitor TP_KEY_NO_BEEP  
call bTglCreateButtonWnd(  
31, 16, &          ' button size  
dlButton, 32, &    ' address and format width of bitmap  
blKeyAttr, &        ' key attributes: Beep OFF / Switch  
wlElementId, &    ' unique ID of this button  
blReturn )        ' Return Code (0: OK Exit >0: Error Code)
```

Turn off auto repeat and beep:

```
blKeyAttr = TP_KEY_NO_AUTOREPEAT bitor TP_KEY_NO_BEEP  
call bTglCreateButtonWnd(  
31, 16, &          ' button size  
dlButton, 32, &    ' address and format width of bitmap  
blKeyAttr, &        ' key attributes: No autorepeat / No beep  
wlElementId, &    ' unique ID of this button  
blReturn )        ' Return Code (0: OK Exit >0: Error Code)
```

These attributes are equal for all windows the elements are placed in. There are more attributes which can be different in each window ( see *bTglSetAttribute*).

## Buttons

### Push Buttons

You can use the inversion attribute to visualize that the push button is pressed. After releasing the button, the standard graphic is shown again without inversion. This is code to activate the inversion:

```
call bTglSetAttribute( &  
    wlElementId, &          ' identifier of button  
    ..WINDOW_ID, &          ' identifier of window  
    TGL_ATTR_INVERT,&       ' set attribute Inversion  
    TGL_TRUE, &             ' set attribute TRUE (Inversion enabled)  
    blReturn )              ' Return Code (0: OK Exit >0: Error Code)
```

Alternative button graphics are used to visualize that a push button is actually pressed. The alternative graphic is automatically shown when the button is pressed. After releasing the button, the standard graphic is shown again. You can generate 3D-effects instead of the standard inversion. After creating an element of type GRAPHIC, you have to link the graphic to the button.

```
call bTglLink( wlBUTTON_ID, wlGRAPHIC_ID, blReturn )
```

A rectangular button with rounded corners, a black border, and the text "about Tiger" in a bold, black, sans-serif font.

figure 47: Unpressed button bitmap

A rectangular button with rounded corners, a black border, and the text "about Tiger" in a bold, black, sans-serif font. The button appears slightly different from the unpressed state, possibly with a different background or shadow.

figure 48: Pressed button bitmap

### Switch Buttons

Buttons can be used as switches, if you set the parameter *KeyAttrin* *bTglCreateButton*, *bTglCreateButtonWnd* or *bTglCreateButtonDockWnd* to *TGL\_KEY\_ATTR\_SWITCH*. In this case, the button generates no keycode to the buffer. The button has 2 states (pressed / NOT pressed). The state of the button changes, if you press it. You can read out the state with *bTglGetButtonState* and set the state with *bTglSetButtonState*. If the state is 0, the standard bitmap is shown, otherwise the alternative graphic is shown if there is a linked graphic or the button will be inverted, if the attribute is set.



## Buttons

Read out current state:

```
call bTglGetButtonState( &
    wElementId, &          ' unique identifier of button
    WINDOW_ID, &           ' identifier of window
    blSwitchState, &       ' current state of switch
    blReturn )             ' Return Code (0: OK Exit >0: Error Code)
```

Set current state:

```
call bTglSetButtonState( &
    wElementId, & ' unique identifier of button
    WINDOW_ID, & ' identifier of window
    0, &         ' set to this value
    blReturn )   ' Return Code (0: OK Exit >0: Error Code)
```

Activate inversion:

```
call bTglSetAttribute( &
    wElementId, & ' element ID
    WINDOW_ID, & ' window ID
    TGL_INVERT, & ' set attribute Inversion
    TGL_TRUE, &   ' set attribute TRUE (Inversion enabled)
    blReturn )    ' Return Code (0: OK Exit >0: Error Code)
```

Alternative button graphics are used to visualize the current state of the button. The alternative graphic is automatically shown, when the state of the button is 1. If the state is 0, the standard graphic is shown again. You can generate 3D-effects instead of the standard inversion. After creating an element of type GRAPHIC, you have to link the graphic to the button.

```
call bTglLink( wlBUTTON_ID, wlGRAPHIC_ID, blReturn )
```

bTglPlaceButtonInWindow

```
call bTglPlaceButtonInWindow( ElementId, WindowId, X, Y, Code, Result )
call bTglPlaceButtonInWindowS( ElementId, WindowId, X, Y, CodeStr, Result )
```

Function: Places a button in a window at position X/Y.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinate on LCD
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode

Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes
						0 ok
						>0 error

After creating a button, you have to place the button into a window. It is possible to place one button in several windows, but never place the same button more than one time into one window.

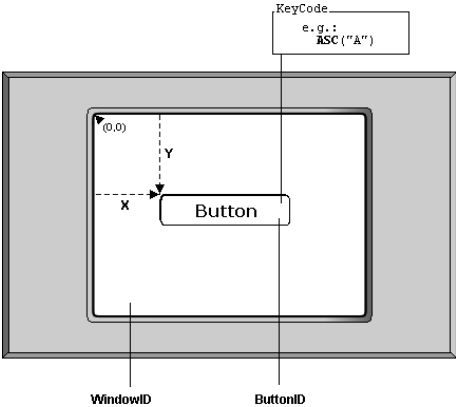


figure 49: Place a button in a window

## Buttons

Sample program:

```
'-----'
' TGL_BUTTON_create_place.TIG
'-----'
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                     ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  datalabel BUTTON
  byte blReturn                ' return value of tgl subroutines
  word wlElementId             ' current identifier for creation of elements
  word wlIbuFill               ' filling of the touch panel input buffer
  byte blKeycode               ' returned keycode

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  call bTglCreateButton( &
    31, 16, &                  ' width, height of element
    BUTTON, 32, &              ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &    ' key attributes auto repeat, beep, type
    wlElementId, &             ' identifier of element
    blReturn )                 ' return code (0: OK exit >0: error exit)

  call bTglPlaceButtonInWindow( &
    wlElementId, WINDOW_ID, &  ' identifier of element, window
    150, 110, &                ' x, y coordinate on LCD
    0h, &                      ' keycode
    blReturn )                 ' return code (0: OK exit >0: error exit)

  *****
  ' show button and get its keycode for further processing
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
  while 1=1
    get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill ' get buffer length
    if wlIbuFill > 0 then                    ' check length of input buffer
      get #TP, #0, 1, blKeycode ' get keycode
    endif
  endwhile
```

## Buttons

```
*****
' FLASH
*****
BUTTON::
data filter "btn_Solo.bmp", "GRAPHFLT", 0      ' WxH=31x16  bmpW=32
end
```

bTglDockButtonInWindow

```
call bTglDockButtonInWindow( ParentId, ElementId, WindowId, XRel, YRel, &  
                             Option, Code, Result )  
call bTglDockButtonInWindowS( ParentId, ElementId, WindowId, XRel, YRel, &  
                              Option, CodeStr, Result )
```

Function: Places a button in a window by docking next to an existing element in one of eight directions. For absolute positioning in window see *bTglPlaceButtonInWindow*.

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

## Buttons

Sample program:

```
-----
' TGL_BUTTON_create_dock.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  datalabel dlButton
  byte blReturn                ' return value of tgl subroutines
  word wIbuFill                ' filling of the touch panel input buffer
  byte blKeycode               ' returned keycode
  word wElementId              ' current identifier for creation of elements
  word wElemIdTmp              ' temporary saved identifier of element

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  ' create and place first first button
  call bTglCreateButtonWnd( &
    31, 16, &                  ' width, height of element
    dlButton, 32, &            ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &    ' key attributes auto repeat, beep, type
    wElementId, WINDOW_ID, &   ' identifier of element, window
    150, 110, &                ' x, y coordinate on LCD
    0h, &                      ' keycode
    blReturn )                 ' return code (0: OK exit >0: error exit)

  ' save identifier of current element for docking
  ' and increment it for the next element to be created
  wElemIdTmp = wElementId
  wElementId = wElementId + 1

  ' create second button without placing
  call bTglCreateButton( &
    31, 16, &                  ' width, height of element
    dlButton, 32, &            ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &    ' key attributes auto repeat, beep, type
    wElementId, &              ' identifier of element
    blReturn )                 ' return code (0: OK exit >0: error exit)
```

```

'' dock second button to the first button with the space of 1 pixel
call bTglDockButtonInWindow( &
wElemIdTmp, &          ' identifier of existing element (parent)
wElementId, &          ' identifier of new element (child)
WINDOW_ID, &          ' identifier of window
1, 0, &                ' x, y offset (space between the elements)
TGL_OPT_RIGHT, &      ' docking option
1h, &                  ' keycode
b1Return )             ' return code (0: OK exit  >0: error exit)

*****
' show button and get its keycode for further processing
*****
call bTglShowWindow( WINDOW_ID, b1Return )
while 1=1
    get #TP, #0, #UFCI_IBU_FILL, 0, w1IbuFill ' get buffer length
    if w1IbuFill > 0 then                     ' check input length of
buffer
        get #TP, #0, 1, b1Keycode            ' get keycode
    endif
endwhile

*****
' FLASH
*****
dlButton::
data filter "btn_Solo.bmp", "GRAPHFLT", 0    ' WxH=31x16 BmpWidth=32x16
end

```

## bTglCreateButtonWnd

```
call bTglCreateButtonWnd( Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                           ElementId, WindowId, X,Y, Code,   Result )
call bTglCreateButtonWndS( Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                           ElementId, WindowId, X,Y, CodeStr, Result )
```

Function: Creates a button and places it to a window at position X/Y.

## Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the button in flash memory must be a multiple of 8
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode

## Return Values:

Result	●	-	-	-	-	error code, for details see table of error codes
						0    ok
						>0   error

This function combines bTglCreateButton and bTglPlaceButtonInWindow in one step.

To create a switch button, please set BIT-6 from KeyAttr.



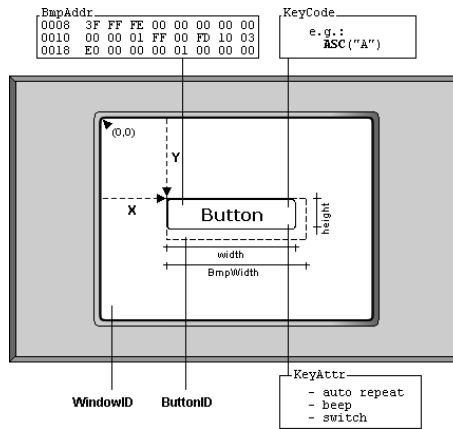


figure 50: Create and place button in a window

## Buttons

Sample program:

```
'-----
' TGL_BUTTON_createWnd.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  datalabel BUTTON
  byte blReturn                ' return value of tgl subroutines
  word wlElementId             ' current identifier for creation of elements
  word wlIbuFill               ' filling of the touch panel input buffer
  byte blKeycode               ' returned keycode

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  call bTglCreateButtonWnd( &
    31, 16, &                  ' width, height of element
    BUTTON, 32, &              ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &    ' key attributes auto repeat, beep, type
    wlElementId, WINDOW_ID, &  ' identifier of element, window
    144, 112, &                ' x, y coordinate on LCD
    0h, &                     ' keycode
    blReturn )                 ' return code (0: OK exit >0: error exit)

  *****
  ' show button and get its keycode for further processing
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
  while 1=1
    get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill  ' get buffer length
    if wlIbuFill > 0 then                      ' check input length of
buffer
      get #TP, #0, 1, blKeycode                ' get keycode
    endif
  endwhile
```

## Buttons

```
*****  
' FLASH  
*****  
BUTTON::  
data filter "btn_Solo.bmp", "GRAPHFLT", 0      ' WxH=31x16 BmpWidth=32x16  
end
```

## bTglCreateButtonDockWnd

```
call bTglCreateButtonDockWnd( Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                               ParentId, ElementId, WindowId, XRel, YRel, &
                               Option, Code,      Result )
```

```
call bTglCreateButtonDockWndS(Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                               ParentId,ElementId, WindowId, XRel, YRel, &
                               Option, CodeStr, Result )
```

Function: Creates and places a button in a window by docking next to an existing element in one of eight directions. For absolute positioning in window see *bTglCreateButtonWnd*.

## Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the button in flash memory must be a multiple of 8
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode

## Buttons

	B	W	L	S	F	
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

Sample program:

```

-----
' TGL_BUTTON_dockWnd.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  datalabel dlButton
  byte blReturn                ' return value of tgl subroutines
  word wLibuFill               ' filling of the touch panel input buffer
  byte blKeycode               ' returned keycode
  word wlElementId             ' current identifier for creation of elements
  word wlElemIdTmp             ' temporary saved identifier of element

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wlElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  '' create first button
  call bTglCreateButtonWnd( &
    31, 16, &                  ' width, height of element
    dlButton, 32, &             ' address, format width of bitmap
    TGL_KEY_ATTR_DEFAULT, &     ' key attributes auto repeat, beep, type
    wlElementId, WINDOW_ID, &   ' identifier of element, window
    150, 110, &                 ' x, y coordinate on LCD
    0h, &                       ' keycode
    blReturn )                  ' return code (0: OK exit >0: error exit)

  '' save identifier of current element for docking

```

```

'' and increment it for the next element to be created
wElemIdTmp = wElementId
wElementId = wElementId + 1

'' dock next button to the right with the space of 1 pixel
call bTglCreateButtonDockWnd( &
31, 16, &                                ' width, height of element
dlButton, 32, &                            ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, &                    ' key attributes auto repeat, beep, type
wElemIdTmp, &                              ' identifier of existing element (parent)
wElementId, &                              ' identifier of new element (child)
WINDOW_ID, &                              ' identifier of window
1, 0, &                                    ' x, y offset (space between the elements)
TGL_OPT_RIGHT, &                          ' docking option
1h, &                                     ' keycode
blReturn )                                ' return code (0: OK exit >0: error exit)

*****
' show button and get its keycode for further processing
*****
call bTglShowWindow( WINDOW_ID, blReturn )
while 1=1
  get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill  ' get buffer length
  if wIbuFill > 0 then                      ' check length of input buffer
    get #TP, #0, 1, blKeycode ' get keycode
  endif
endwhile

*****
' FLASH
*****
dlButton:: data filter "btn_Solo.bmp", "GRAPHFLT", 0 ' WxH=31x16 BmpW=32
end

```

### bTglCreateButtonDockWnd

```
call bTglCreateButtonDockWnd( Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                               ParentId, ElementId, WindowId, XRel, YRel, &
                               Option, Code,      Result )
call bTglCreateButtonDockWndS(Width, Height, BmpAddr, BmpWidth, KeyAttr, &
                               ParentId,ElementId, WindowId, XRel, YRel, &
                               Option, CodeStr, Result )
```

Function: Creates and places a button in a window by docking next to an existing element in one of eight directions. For absolute positioning in window see bTglCreateButtonWnd ().

#### Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the button in flash memory must be a multiple of 8
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode

Result

B	W	L	S	F
●	-	-	-	-

**Return Values:**  
error code, for details see table of error codes  
0    ok  
>0   error



bTglGetPushButtonState

```
call bTglGetPushButtonState( ElementId, State )
```

Function: Reads out the current state of a push button.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
State	●	-	-	-	-	

**Return Values:**  
current state of switch button  
TGL\_TRUE = actually pressed  
TGL\_FALSE = actually not pressed  
invalid identifier  
no button or switch  
button not placed in actual window  
button is not active

This function is only available for all button types, also for switch buttons. For getting the saved state of a switch call *bTglGetButtonState*.

bTglGetButtonState

```
call bTglGetButtonState( ElementId, WindowId, State, Result )
```

Function: Reads out the current state of the switch button.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
State	●	-	-	-	-	current state of switch button ( 0 / 1 )
Result	●	-	-	-	-	error code, for details see table of error codes

**Return Values:**  
current state of switch button ( 0 / 1 )  
error code, for details see table of error codes  
0 ok  
>0 error

This function is only available for switch buttons, not for standard buttons. You can read out the state every time, even if the window of the switch button is not active. State can be 0 or 1. To set the state manually, please use the function *bTglSetButtonState*.

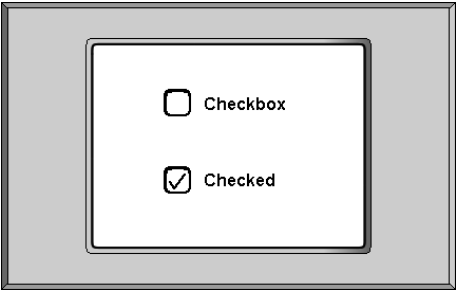


figure 51: Switch used as a checkbox

## Buttons

Example program:

```
'-----
' TGL_SWITCH_get_state.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
    dataLabel dlButton, dlGraphic
    byte blReturn              ' return value of tgl subroutines
    byte blSwitchState        ' current state of switch button
    word wlElementId          ' current identifier for creation of elements
    word wlSWITCH_ID          ' "constant" identifier for switch

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
call vTglInit()
wlElementId = 0

*****
' TGL ELEMENTS AND WINDOWS
*****
' create a switch button with the normal bitmap
call bTglCreateButtonWnd( &
101, 31, &                  ' width, height of element
dlButton, 104, &            ' address, format width of bitmap
TGL_KEY_ATTR_SWITCH, &     ' key attributes auto repeat, beep, type
wlElementId, WINDOW_ID, &  ' identifier of element, window
120, 100, &                ' x, y coordinate on LCD
0h, &                      ' keycode
blReturn )                  ' return code (0: OK exit >0: error exit)

' save identifier of current element for linking and further processing
' and increment it for the next element to be created
wlSWITCH_ID = wlElementId
wlElementId = wlElementId + 1

' create a graphic with an alternative bitmap for the button
call bTglCreateGraphic( &
101, 31, &                  ' width, height of element
dlGraphic, 104, &          ' address, format width of bitmap
wlElementId, &             ' identifier of element
blReturn )                  ' return code (0: OK exit >0: error exit)
```

```
' link the graphic with the alternative bitmap to the button
call bTglLink( &
wlsWITCH_ID, &          ' identifier of button
wlelementId, &          ' id of graphic with alternative bitmap
blReturn )              ' return code (0: OK exit >0: error exit)

*****
' get button states for further processing
*****
call bTglShowWindow( WINDOW_ID, blReturn )
while l=1
  call bTglGetButtonState( &
    wlsWITCH_ID, WINDOW_ID, &  ' identifier of element, window
    blSwitchState, &          ' current state of switch
    blReturn )                ' return code (0: OK exit >0: error exit)
endwhile

*****
' FLASH
*****
dlButton::
data filter "btn_abouttiger.bmp", "GRAPHFLT", 0          ' WxH=101x31 Bmp=104
dlGraphic::
data filter "btn_abouttiger_active.bmp", "GRAPHFLT", 0' WxH=101x31 Bmp=104
end
```

bTglSetButtonState

```
call bTglSetButtonState( ElementId, WindowId, State, Result )
```

Function: Sets the current state of the switch button.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
State	●	-	-	-	-	set current state of switch button ( 0 / 1 )
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

This function is only available for switch buttons, not for standard buttons. You can set the state every time, even if the window of the switch button is not active. State can be 0 or 1. To read out the current state, please use the function *bTglGetButtonState*.

## Buttons

Example program:

```
'-----
' TGL_SWITCH_set_state.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' identifier for windows
#define WINDOW_ID              0

task main
    datalabel dlButton, dlGraphic
    byte ever
    byte blReturn              ' return value of tgl subroutines
    word wlElementId           ' current identifier for creation of elements
    word wlsWITCH_ID           ' identifier of switch button

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
call vTglInit()
wlElementId = 0

*****
' TGL ELEMENTS AND WINDOWS
*****
' create a switch button with the normal bitmap
call bTglCreateButtonWnd( &
101, 31, &                    ' width, height of element
dlButton, 104, &              ' address, format width of bitmap
TGL_KEY_ATTR_SWITCH, &       ' key attributes auto repeat, beep, type
wlElementId, WINDOW_ID, &    ' identifier of element, window
120, 100, &                  ' x, y coordinate on LCD
0h, &                        ' keycode
blReturn )                    ' return code (0: OK exit >0: error exit)

' save identifier of current element for linking and switching
' and increment it for the next element to be created
wlsWITCH_ID = wlElementId
wlElementId = wlElementId + 1

' create a graphic with an alternative bitmap for the button
call bTglCreateGraphic( &
101, 31, &                    ' width, height of element
dlGraphic, 104, &             ' address, format width of bitmap
wlElementId, &                ' identifier of element
blReturn )                    ' return code (0: OK exit >0: error exit)
```

```

' link the graphic with the alternative bitmap to the button
call bTglLink( &
wlsWITCH_ID, &          ' identifier of button
wElementId, &          ' id of graphic with alternative bitmap
bIReturn )              ' return code (0: OK exit >0: error exit)

*****
' switch between button states
*****
call bTglShowWindow( WINDOW_ID, bIReturn )
for ever=0 to 0 step 0
' set switch button to state with standard bitmap
call bTglSetButtonState( &
wlsWITCH_ID, WINDOW_ID, & ' identifier of element, window
TGL_STANDARD, &          ' value of button state
bIReturn )                ' return code (0: OK exit >0: error exit)
wait_duration 1000

' set switch button to state with alternative bitmap
call bTglSetButtonState( &
wlsWITCH_ID, WINDOW_ID, & ' identifier of element, window
TGL_ALTERNATIVE, &        ' value of button state
bIReturn )                ' return code (0: OK exit >0: error exit)
wait_duration 1000
next

*****
' FLASH
*****
dlButton::
data filter "btn_abouttiger.bmp",          "GRAPHFLT",0 ' WxH=101x31 BmpW=104
dlGraphic::
data filter "btn_abouttiger_active.bmp",    "GRAPHFLT",0 ' WxH=101x31 BmpW=104
end

```

## bTglGetKeycode

call bTglGetKeycode ( Keycode, IbuFill )

Function: In case of buffer filling return keycode.

### Parameters:

	B	W	L	S	F	Return Values:
Keycode	●	-	-	-	-	keycode of button
IbuFill	-	●	-	-	-	input buffer filling

The returned keycode has been passed by the place function for buttons.

In case of buffer filling one byte will be read from touch panel input buffer

Use this function in a loop:

```
byte ever, blKeycode
word wIbuFill
for ever=0 to 0 step 0
  call bTglGetKeycode( blKeycode, wIbuFill )
  if 0 < wIbuFill then ` check buffer filling
    switchi blKeycode
      case 0:
        ` job for keycode=0
      case 1:
        ` job for keycode=1
      default:
        endswitch
    endif
  next ` endless loop
```



### bTglWaitKeycode

call bTglWaitKeycode ( Keycode)

Function: Wait until any standard button (not switches) has been pressed and return keycode.

#### Parameters:

	B	W	L	S	F	Return Values:
Keycode	●	-	-	-	-	keycode of button

The returned keycode has been passed by the place function for buttons.

This functions can be used, if nothing else should be done in the task for the graphical user interfase. This task wait until any button has been pressed. In case of other jobs better use function *bTglGetKeycode*.

Use this function in a loop:

```
byte blKeycode
call bTglWaitKeycode( blKeycode )
switchi blKeycode
case 0:
  \ job for keycode=0
case 1:
  \ job for keycode=1
default:
endswitch
```

# Text Button

Text buttons are rectangular areas in a window containing a text graphic and having touch panel functionality.

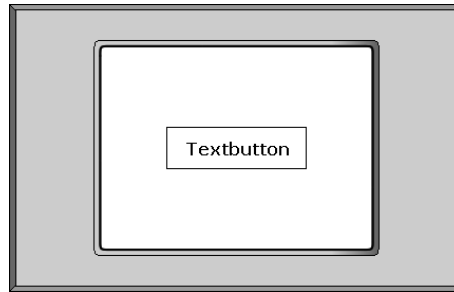


figure 52: Touch panel button with graphic text

Text buttons can be used as switches as you can do it with normal buttons. You can link an alternative text to the switch e.g. for a start/stop button:

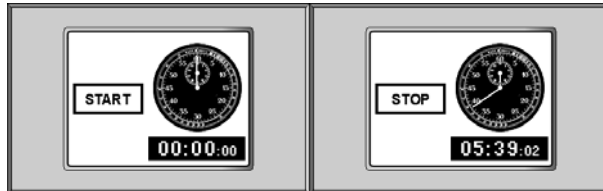


figure 53: Start/stop button

Please see chapter *Read out touch panel keyboard buffer* and *Auto repeat* from *Buttons* for reading out the buffer and handling of *TOUCHPANEL.TDD*

! The text on text buttons is written with a graphic font. You have to create this font before using this element. Please see chapter *Graphic Fonts* for creating your own fonts.

## Text Button

Available subroutines for text buttons:

- *bTglCreateTextButton*
- *bTglPlaceTextButtonInWindow*
- *bTglDockTextButtonInWindow*
- *bTglCreateTextButtonWnd*
- *bTglCreateTextButtonDockWnd*

See also:

- subroutines for buttons
- *bTglLink*
- *bTglSetAttribute*
- *bTglSetMargins*

There are two possibilities to pass the text for the text button. The standard subroutines store the text in RAM memory. In this case, the text is passed as string or constant. You need enough reserved memory space for the text pool. The reserved size is defined with `TGL_MAX_MEM_TEXTS_LEN` in the configuration file *TigerGraphicLibraryConf.tig*. Ensure all your text fits into the determined number of bytes.

```
call sTglCreateTextButton( 30,30,120,40,"Hello text button", 2, &
b1FONT_ID, w1BUTTON_ID, WINDOW_ID, b1Return )
```

The alternative is to save the text into the FLASH memory. In this case you need no RAM memory for the text of the text buttons. You can save many texts in FLASH memory. One possibility to save a string into FLASH is the instruction DATA STRING:

```
dlFlashAddr:
DATA STRING "Hello Textbutton"
```

Just pass the data label and the text appears correct in the text button:

```
call sTglCreateTextButtonF( 30,30,120,40, dlFlashAddr, 2, &
b1FONT_ID, w1BUTTON_ID, WINDOW_ID, b1Return )
```

If you use **Tiger 2**, the first four Bytes determine the length of the following text, for **Tiger 1** the first two Bytes determine the length of the text. DATA STRING handles this feature correct. Please notice this for manual storage in FLASH memory.

bTglCreateTextButton

```
call bTglCreateTextButton( Width, Height, Text, FontId, Frame, KeyAttr, &
                           ElementId, Result )
call bTglCreateTextButtonF( Width, Height, TextAddr, FontId, Frame, KeyAttr, &
                           ElementId, Result )
call bTglCreateTextButtonVar( Width, Height, FontId, Frame, KeyAttr, &
                              ElementId, Result )
```

Function: Creates a new text button.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of text button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off bit-6: 0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

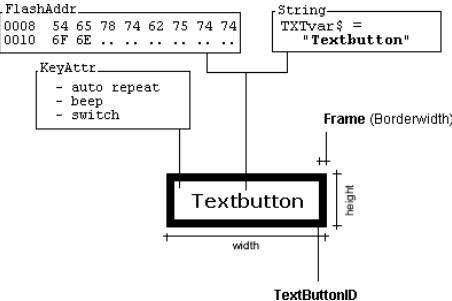


figure 54: Create text button

! The text button will not be created, if the text graphic does not fit in the text button. The fitting of the text graphic in the text button depends on the chosen font, the text length and the frame thickness.

For more Information about the key attributes see *bTglCreateButton*.

Additional attributes for text elements are margins. They can be set by the function *bTglSetMargins*:

```
call bTglSetMargins( wlTop,wlBottom,wlLeft,wlRight, wlElementId, blReturn )
```

bTglPlaceTextButtonInWindow

```
call bTglPlaceTextButtonInWindow( ElementId, WindowId, X, Y, Code, Result )
call bTglPlaceTextButtonInWindowS( ElementId, WindowId, X, Y, CodeStr, Result )
```

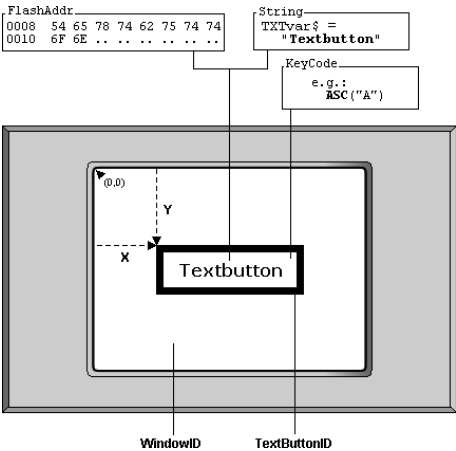
Function: Places a text button in a window at position X/Y.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode

Return Values:

Result  
● - - - - error code, for details see table of error codes  
0 ok  
>0 error



bTglDockTextButtonInWindow

```
call bTglDockTextButtonInWindow( ParentId, ElementId, WindowId, &
                                XRel, YRel, Option, Code, Result )
call bTglDockTextButtonInWindowS( ParentId, ElementId, WindowId, &
                                XRel, YRel, Option, CodeStr, Result )
```

Function: Places a Textbutton in a window by docking next to an existing element in one of eight directions. For absolute positioning in window see *bTglPlaceTextButtonInWindow()*.

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

The text button will not be created, if the text graphic does not fit in the text button. The fitting of the text graphic in the text button depends on the choosen font, the text length and the frame thickness.

## bTglCreateTextButtonWnd

```
call bTglCreateTextButtonWnd( Width, Height, Text, FontId, Frame, KeyAttr, &
                             ElementId, WindowId, X, Y, Code, Result )
call bTglCreateTextButtonFWnd( Width, Height, TextAddr, FontId, Frame, KeyAttr, &
                             ElementId, WindowId, X, Y, Code, Result )
call bTglCreateTextButtonVarWnd( Width, Height, FontId, Frame, KeyAttr, &
                                ElementId, WindowId, X, Y, Code, Result )
call bTglCreateTextButtonWndS( Width, Height, Text, FontId, Frame, KeyAttr, &
                                ElementId, WindowId, X, Y, CodeStr, Result )
call bTglCreateTextButtonFWndS( Width, Height, TextAddr, FontId, Frame, KeyAttr, &
                                ElementId, WindowId, X, Y, CodeStr, Result )
call bTglCreateTextButtonVarWndS( Width, Height, FontId, Frame, KeyAttr, &
                                  ElementId, WindowId, X, Y, CodeStr, Result )
```

Function: Creates a text button and places it to a window at position X/Y.

bTglCreateTextButtonWnd	The text is saved in RAM, you can pass the text as string or constant. The keycode is passed numerical
bTglCreateTextButtonFWnd	The text is saved in FLASH memory, you pass a label or FLASH address of string saved in FLASH (4 Byte length + text). The keycode is passed numerical
bTglCreateTextButtonVarWnd	No Text is passed. The keycode is passed numerical
bTglCreateTextButtonWndS:	The text is saved in RAM, you can pass the text as string or constant. The keycode is passed as character (string)
bTglCreateTextButtonFWndS	The text is saved in FLASH memory, you pass a label or FLASH address of string saved in FLASH (4 Byte length + text). The keycode is passed as character (string)
bTglCreateTextButtonVarWndS	No Text is passed. The keycode is passed as character (string)

### Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of text button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame



Text Button

	B	W	L	S	F	
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0    ok >0   error

The text button will not be created, if the text graphic does not fit in the text button. The fitting of the text graphic in the text button depends on the choosen font, the text length and the frame thickness.

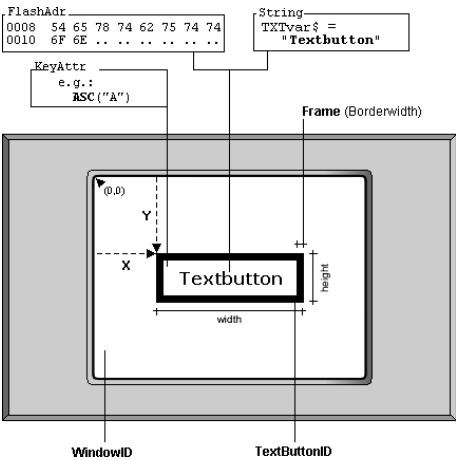


figure 56: Create and place a textbutton in a window

## **bTglCreateTextButtonDockWnd**

```
call bTglCreateTextButtonDockWnd( Width, Height, Text, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, Code, Result )  
call bTglCreateTextButtonFDockWnd( Width, Height, TextAddr, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, Code, Result )  
call bTglCreateTextButtonVarDockWnd( Width, Height, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, Code, Result )  
call bTglCreateTextButtonDockWndS( Width, Height, Text, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, CodeStr, Result )  
call bTglCreateTextButtonFDockWndS( Width, Height, TextAddr, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, CodeStr, Result )  
call bTglCreateTextButtonVarDockWndS( Width, Height, FontId, Frame, &  
KeyAttributes, ParentId, ElementId, WindowId, &  
XRel, YRel, Option, CodeStr, Result )
```

Function: Creates a text button and places it to a window by docking next to an existing element in one of eight directions. For absolute positioning see *bTglCreateTextButtonWnd*.

bTglCreateTextButtonDockWnd	The text is saved in RAM, you can pass the text as string or constant. The keycode is passed numerical
bTglCreateTextButtonFDockWnd	The text is saved in FLASH memory, you pass a label or FLASH address of string saved in FLASH (4 Byte length + text). The keycode is passed numerical
bTglCreateTextButtonVarDockWnd	No Text is passed. The keycode is passed numerical
bTglCreateTextButtonDockWndS:	The text is saved in RAM, you can pass the text as string or constant. The keycode is passed as character (string)
bTglCreateTextButtonFDockWndS	The text is saved in FLASH memory, you pass a label or FLASH address of string saved in FLASH (4 Byte length + text). The keycode is passed as character (string)
bTglCreateTextButtonVarDockWndS	No Text is passed. The keycode is passed as character (string)

## Text Button

### Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of text button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line width of frame
	B	W	L	S	F	
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat    0=on, 1=off bit-5: beep            0=on, 1=off bit-6:    0: standard button 1: switch button
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Code	●	-	-	-	-	binary keycode
CodeStr	-	-	-	●	-	string keycode
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0    ok >0   error

For details about the docking subroutines please see the chapter *Docking*.

## Text Button

Sample program:

```
'-----
' TGL_TEXTBUTTON.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

'' identifier of windows
#define WINDOW_ID              0

task main
    byte blReturn              ' return value of tgl subroutines
    string slKeyCode$(1)      ' touch panel input
    string slLcdOutput$(20h)
    word wElementId            ' current identifier for creation of elements
    word wLABEL_ID             ' "constant" identifier of label
    byte blFontId

    #include TGL_DEVICE_DRIVERS_TP1000.INC

    *****
    ' INITIALIZATION
    *****
    call vTglInit()
    wElementId = 0
    blFontId   = 0
    set_len$( slKeyCode$, 0 )
    set_len$( slLcdOutput$, 0 )

    *****
    ' TGL FONTS
    *****
    call bTglCreateFontParams( &
    blFontId, &                ' identifier of font
    "Valencia",10,"normal", &  ' name, size, type of font
    "center", "center", &      ' alignment horizontal, vertical
    "prop", 0, &                ' spacing type, blank
    SPACING_CHAR_DEFAULT, 0, &  ' spacing char, vertical
    "imm", "char", &           ' overlay, wrap mode
    blReturn )                  ' return code (0: OK exit  >0: error exit)

    *****
    ' TGL ELEMENTS AND WINDOWS
    *****
    call bTglCreateTextButtonWnd( &
    60, 40, &                  ' width, height of element
    "PRESS", &                  ' text in element
    blFontId, 5, &              ' text, font id, frame thickness
    TGL_KEY_ATTR_DEFAULT, &     ' key attributes: auto repeat,beep,no switch
    wElementId, WINDOW_ID, &    ' identifier of text button, window
    130, 100, &                 ' x, y coordinate on LCD
    41h, &                      ' keycode
    blReturn )                  ' return code (0: OK exit  >0: error exit)

    '' increment identifier for next element
    wElementId = wElementId + 1

    call bTglCreateLabelVarWnd( &
```

```

120, 40,&                                ' width, height of element
blFontId, 0, &                            ' identifier of font, frame thickness
wElementId, WINDOW_ID, &                 ' identifier of element, window
100, 40, &                               ' x, y coordinate on LCD
blReturn )                               ' return code (0: OK exit >0: error exit)

' save identifier of label to show the returned keycode
wLABEL_ID = wElementId

*****
' show and erase button code in label
*****
call bTglShowWindow( WINDOW_ID, blReturn )
loop 7FFFFFFh

' display code of pressed button
call vWaitForPressedButton()
get #TP, #0, 1, slKeyCode$               ' get single button code
slLcdOutput$ = "Button returns: " + slKeyCode$
call bTglSetText( wLABEL_ID, slLcdOutput$, blReturn )
call bTglShow( wLABEL_ID, WINDOW_ID, blReturn )

' erase displayed code of pressed button
call vWaitForPressedButton()
get #TP, #0, 0, slKeyCode$               ' read out buffer
call bTglSetText( wLABEL_ID, TGL_NO_TEXT, blReturn )
call bTglShow( wLABEL_ID, WINDOW_ID, blReturn )
endloop
end

sub vWaitForPressedButton()
word wIbuFill                            ' filling of input buffer of touch panel
wIbuFill = 0
get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
while wIbuFill = 0
    release_task
    get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
endwhile
end

```

## Slider

Sliders are touch elements for a continuous adjustment of a value by sliding over the touch panel. This element is used to create e.g. an equalizer by mixing different frequencies for a sound.

Example for the use of sliders:

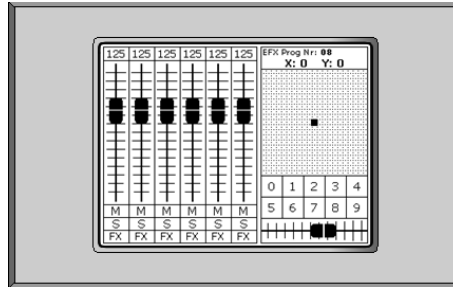


figure 57: Example for sliders

Available subroutines for sliders:

- *bTglCreateSlider*
- *bTglPlaceSliderInWindow*
- *bTglDockSliderInWindow*
- *bTglCreateSliderWnd*
- *bTglCreateSliderDockWnd*
- *bTglSetSliderValue*
- *lTglGetSliderValue*

See also:

- *bTglLink*
- *Graphic*

## Slider

There are 2 different types of sliders, the X-Slider and the Y-Slider. An X-Slider could look like this. You can change the value by sliding along the x-axis.

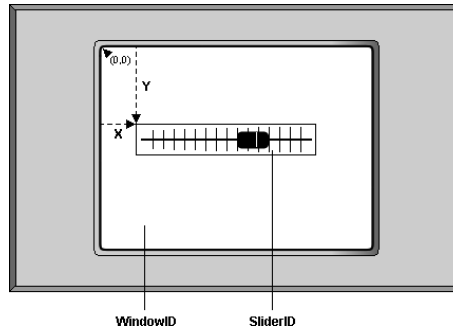


figure 58: X-Slider

This is an example of a Y-Slider. You can change the value by sliding along the y-axis.

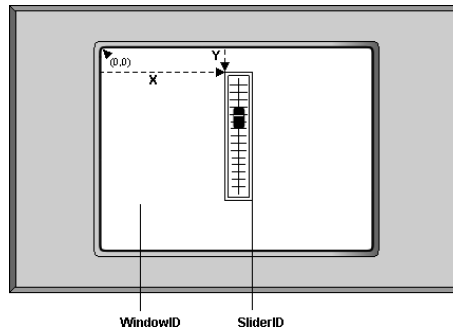


figure 59: Y-Slider

## Slider

The first step to create such a slider is to call the function `bTglCreateSliderWnd` or call `bTglCreateSlider` before calling `bTglPlaceSliderInWindow`. Both possibilities have the same effect.

```
call bTglCreateSliderWnd( &
31, 170, &          ' size of slider
200, -200, &        ' top, bottom value 1
0, 0, &             ' dummies for linear slider types
"Y", &              ' type of slider
dlslidebar, 32,&' address, format width of bitmap
wlsSLIDEBAR_ID,&     ' unique ID of this slidebar
WINDOW_ID, &        ' window ID to create this slider in
10, 20 &            ' X,Y coordinate on LCD
blReturn )          ' Return Code (0: OK Exit >0: Error Code)
```

A graphic linked with a slide bar is used as a variable slider (slider button). If the slider value has changed, the slider is shown on the current position. It visualizes the current position for the user. To create such a slider button, you must create a graphic with the button. For details about the graphic element, please look at chapter Graphic. After creating the graphic for the slider button, just link the graphic to the slidebar:

Example for creating a graphic:

```
'*****
' create graphic for slider
'*****
call bTglCreateGraphic( &
16,30, &              ' width, height of graphic
dlsliderButton, 16, & ' address, format width of bitmap
wlsSLIDERBUTTON_ID, & ' unique ID of this graphic
blReturn )            ' Return Code (0: OK Exit >0: Error Code)
```

Link graphic to slide bar:

```
'*****
' link graphic to slidebar
'*****
call bTglLink( &
wlsSLIDEBAR_ID, &      ' unique ID of slidebar
wlsSLIDERBUTTON_ID, &  ' unique ID of slider button graphic
blReturn )             ' Return Code (0: OK Exit >0: Error Code)
```



## Slider

Please notice, that a linked graphic will reduce the touch area of the sidebar. For an X-Slider, the half of the slider button width will be cut from the left and the right side of the sidebar. The Y-Slider will be scaled down at the top and the bottom. Please see the following graphics:

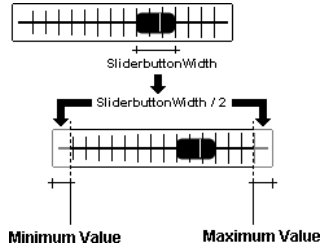


figure 60: Scaling x-slider

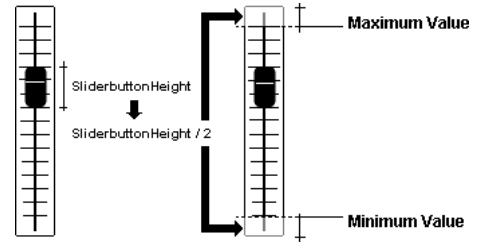


figure 61: Scaling y slider

The value of every slider is saved, even if you change the active window, the slider value will not be lost. To set a new slider value, please use the function *bTglSetSliderValue*. You can read out the current slider value with the function *lTglGetSliderValue*. The slider value can vary between the determined minimum and maximum value.

The minimum value can be greater than the maximum value. In this case the slider reverses.

Read out slider value:

```
call lTglGetSliderValue( &  
w1SLIDEBAR_ID, &      'unique ID of sidebar  
WINDOW_ID, &          ' unique ID of window  
TGL_SL_OPT_VALUE, &   ' option: read out current value of slider  
l1SliderValue, &      ' Return value: current value of slider  
b1Return )            ' Return Code (0: OK Exit >0: Error Code)
```

bTglCreateSlider

```
call bTglCreateSlider( Width, Height, Min, Max, Dummy, Dummy, XYFlag$, &
                      BmpAddr, BmpWidth, ElementId, Result )
```

Function:      Creates a new sidebar with the identifier ElementId.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of sidebar
Min, Max	-	-	●	-	-	value at minimum, maximum coordinate
Dummy	-	-	●	-	-	dummy value for linear sliders
XYFlag	-	-	-	●	-	"X": x-coordinate sidebar (left-right) "Y": y-coordinate sidebar (top-bottom)
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpWidth	-	●	-	-	-	width of the button in flash memory must be a multiple of 8
ElementId	-	●	-	-	-	unique identifier of this element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0    ok >0   error

You can place this slider with the same identifier several times in different windows, but never place two sliders with the same identifier in the same window!

To create a variable slider button for visualizing the current slider position, please create a graphic with the slider button and link the graphic to the slider with the function *bTglLink*.

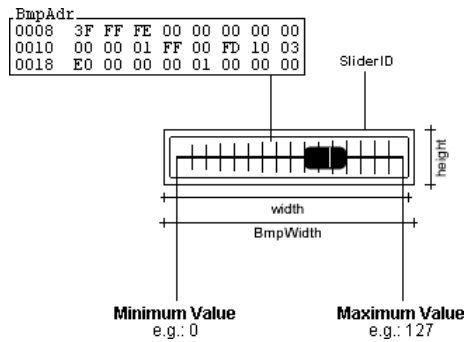


figure 62: Create x slider

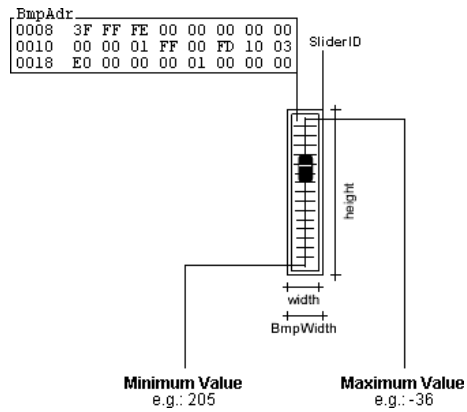


figure 63: Create y slider

bTglPlaceSliderInWindow

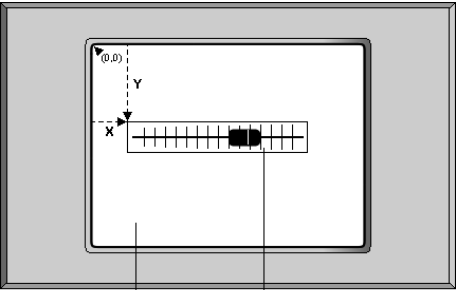
call bTglPlaceSliderInWindow( ElementId, WindowId, X, Y, Result )

Function: Places slider in a window at position X/Y.  
For relative positioning to existing elements see subroutine  
*bTglDockSliderInWindow*

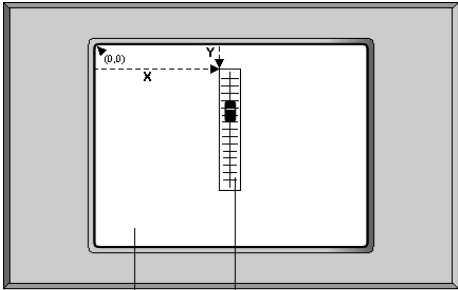
Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error



WindowID SliderID  
figure 64: Place an x slider in a window



WindowID SliderID  
figure 65: Place an y-slider in a window

## Slider

Sample program:

```
-----
' TGL_SLIDER_Y_create_place.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' elements
#define SLIDEBAR_ID      0
#define SLIDERBUTTON_ID 1
' windows
#define WINDOW_ID        0

task main
  datalabel d1SlideBar, d1SliderButton
  byte blReturn
  long l1SliderValue      ' current value of the slider

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  ' create slide bar
  call bTglCreateslider( &
    31, 170, &              ' width, height of element
    -200, 200, &            ' top, bottom value
    0, 0, &                 ' dummy values for linear sliders
    "Y", &                  ' type of slider (direction)
    d1SlideBar, 32, &        ' address, format width of bitmap
    SLIDEBAR_ID, &          ' identifier of element
    blReturn )              ' return code (0: OK exit  >0: error exit)

  ' place slide bar in window
  call bTglPlaceSliderInWindow( &
    SLIDEBAR_ID, WINDOW_ID, & ' identifier of element, window
    144, 44, &               ' x, y coordinate on LCD
    blReturn )               ' return code (0: OK exit  >0: error exit)

  ' create sliderbutton
  call bTglCreateGraphic( &
    16, 30, &                ' width, height of element
    d1SliderButton, 16, &    ' address, format width of bitmap
    SLIDERBUTTON_ID, &      ' identifier of element
    blReturn )              ' return code (0: OK exit  >0: error exit)

  ' link slider button to slide bar
  call bTglLink( &
    SLIDEBAR_ID, &           ' identifier of slide bar (slider)
    SLIDERBUTTON_ID,&        ' identifier of slider button (graphic)
```

```
blReturn )          ' return code (0: OK exit  >0: error exit)

'*****
' show window with slider
'*****
call bTglShowWindow( WINDOW_ID, blReturn )
while 1=1
  ' get current slider value for further processing
  call lTglGetSliderValue( &
    SLIDEBAR_ID, WINDOW_ID, &      ' identifier of slider, window
    TGL_SL_OPT_VALUE, &           ' option: read out current value/position
    llSliderValue, &              ' returned current slider value/position
    blReturn )                    ' return code (0: OK exit  >0: error exit)
endwhile

dlSlideBar::
data filter "chnl_slidebar.bmp",    "GRAPHFLT", 0 ' WxH=31x170 BmpW=32
dlSliderButton::
data filter "chnl_slider_black.bmp", "GRAPHFLT", 0 ' WxH=16x30 BmpW=16
end
```

bTglDockSliderInWindow

call bTglDockSliderInWindow( ParentId, ElementId, Window, XRel, YRel, Option, Return )

Function: Places a slide bar in a window by docking next to the existing parent element in one of 8 directions.  
For absolute positioning in window see subroutine *bPlaceLabellInWindow*

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of new element (child)
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

Sample program:

```

-----
' TGL_SLIDER_Y_create_dock.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' elements
#define SLIDEBAR_ID             0
#define SLIDERBUTTON_ID        1
#define SLIDEBAR_ID2           2
' windows
#define WINDOW_ID               0

task main
    datalabel CHNL_SLIDEBAR
    datalabel CHNL_SLIDER
    byte blReturn
    long llsliderValue          ' current value of the slider

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
call vTglInit()

*****
' create slider
*****
call bTglCreateSliderWnd( 31, & ' width of slider
    170, &          ' height of slider
    200, &          ' minimum value 1
    -200, &         ' maximum value 1
    200, &          ' minimum value 2
    -200, &         ' maximum value 2
    "Y", &          ' type of slider
    CHNL_SLIDEBAR, & ' address of bitmap
    32, &           ' width of bitmap
    SLIDEBAR_ID, &  ' unique ID of this sliderbar
    WINDOW_ID, &    ' window ID to create this slider in
    10, &           ' X-coordinate in LCD
    20, &           ' Y-coordinate in LCD
    blReturn )      ' Return Code (0: OK Exit >0: Error Code)

*****
' create graphic for slider
*****
call bTglCreateGraphic( 16, & ' width of graphic
    30, &          ' height of graphic
    CHNL_SLIDER, &   ' address of bitmap
    16, &           ' width of bitmap
    SLIDERBUTTON_ID, & ' unique ID of this graphic
    blReturn )      ' Return Code (0: OK Exit >0: Error Code)

```



```

*****
' link graphic to sidebar
*****
call bTglLink(SLIDEBAR_ID, & ' unique ID of sidebar
SLIDERBUTTON_ID, & ' unique ID of slider button graphic
blReturn) ' Return Code (0: OK Exit >0: Error Code)

*****
' create sidebar
*****
call bTglCreateslider( 31, & ' width of slider
170, & ' height of slider
200,& ' minimum value 1
-200, & ' maximum value 1
200, & ' minimum value 2
-200, & ' maximum value 2
"Y", & ' type of slider
CHNL_SLIDEBAR, &' address of bitmap
32, & ' width of bitmap
SLIDEBAR_ID2, &' unique ID of this sidebar
blReturn ) ' Return Code (0: OK Exit >0: Error Code)

*****
' dock slider in window
*****
call bTglDockSliderInWindow( SLIDEBAR_ID, & ' unique ID parent element
SLIDEBAR_ID2, &' unique ID of this sidebar
WINDOW_ID, & ' window ID to create this slider in
10, & ' X-coordinate offset from docking point
0, & ' Y-coordinate offset from docking point
TGL_OPT_RIGHT, &' docking option
blReturn ) ' Return Code (0: OK Exit >0: Error Code)

*****
' link graphic to sidebar
*****
call bTglLink(SLIDEBAR_ID2, & ' unique ID of sidebar
SLIDERBUTTON_ID, & ' unique ID of slider button graphic
blReturn) ' Return Code (0: OK Exit >0: Error Code)

*****
' show window
*****
call bTglShowWindow( WINDOW_ID, & ' window ID to show
blReturn ) ' Return Code (0: OK Exit >0: Error Code)
while 1=1
call lTglGetSliderValue(SLIDEBAR_ID, & ' unique ID of sidebar
WINDOW_ID, & ' unique ID of window
TGL_SL_OPT_VALUE, & ' option: read out current value of slider
llSliderValue, & ' Return value: current value of slider
blReturn) ' Return Code (0: OK Exit >0: Error Code)
endwhile

*****
' FLASH
*****
CHNL_SLIDEBAR::
data filter "chnl_slider.bmp", "GRAPHFLT", 0 ' WxH=31x170 BmpW=32
CHNL_SLIDER::

```

## Slider

```
data filter "chnl_slider_black.bmp", "GRAPHFLT", 0 ' WxH=16x30 BmpW=16  
end
```

bTglCreateSliderWnd

call bTglCreateSliderWnd( Width, Height, Min, Max, Dummy, Dummy, XYFlag\$, & BmpAddr, BmpWidth, ElementId, WindowId, X, Y, Result )

Function: Creates a slide bar and places it to a window at position X/Y.

Parameters:

	B	W	L	S	F	
Width	-	●	-	-	-	width of graphic
Height	-	●	-	-	-	height of graphic
Min	-	-	●	-	-	minimum value of slider
Max	-	-	●	-	-	maximum value of slider
Dummy	-	-	●	-	-	dummy value
Dummy	-	-	●	-	-	dummy value
XYFlag	-	-	-	●	-	"X": x-coordinate slide bar (left-right) "Y": y-coordinate slide bar (top-bottom)
BmpAddr	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidth	-	-	●	-	-	width of the graphic in flash memory Must be a multiple of 8
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	unique identifier of window
X	-	●	-	-	-	X-coordinate of left top edge
Y	-	●	-	-	-	Y-coordinate of left top edge
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

You can use this slider element in several times in different windows, but never more than 1 time in 1 window! To create a variable slider button for visualizing the current slider position, please create a graphic with the slider button and link the graphic to the slider with the function *bTglLink*.

bTglCreateSliderDockWnd

call bTglCreateSliderDockWnd(Width, Height, Min, Max, Dummy, Dummy, XYFlag\$, & BmpAddr, BmpWidth, ParentId, ElementId, WindowId, & X, Y, Option, Result)

Function: Places a slide bar in a window by docking next to an existing parent element in one of 8 directions.  
For absolute positioning in window see subroutine *bTglCreateSliderWnd*.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of graphic
Min, Max	-	-	●	-	-	value at minimum, maximun coordinate
Dummy	-	-	●	-	-	dummy value for linerar sliders
XYFlag	-	-	-	●	-	"X": x-coordinate slide bar (left-right) "Y": y-coordinate slide bar (top-bottom)
BmpAddr	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidth	-	●	-	-	-	width of the graphic in flash memory Must be a multiple of 8
ParentId	-	●	-	-	-	unique identifier of existing element (parent)
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	-	●	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

Sample program:

```

-----
' TGL_SLIDER_Y_dockWnd.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' elements
#define SLIDEBAR_ID              0
#define SLIDERBUTTON_ID          1
#define SLIDEBAR_ID2             2
' windows
#define WINDOW_ID                0

task main
    datalabel CHNL_SLIDEBAR
    datalabel CHNL_SLIDER
    byte blReturn
    long llsliderValue           ' current value of the slider

#include TGL_DEVICE_DRIVERS_TP1000.INC

*****
' INITIALIZATION
*****
call vTglInit()

*****
' create slider
*****
call bTglCreateSliderWnd( 31, & ' width of slider
    170, &                        ' height of slider
    200, &                        ' minimum value 1
    -200, &                       ' maximum value 1
    200, &                        ' minimum value 2
    -200, &                       ' maximum value 2
    "Y", &                        ' type of slider
    CHNL_SLIDEBAR, &              ' address of bitmap
    32, &                        ' width of bitmap
    SLIDEBAR_ID, &               ' unique ID of this sliderbar
    WINDOW_ID, &                 ' window ID to create this slider in
    10, &                        ' X-coordinate in LCD
    20, &                        ' Y-coordinate in LCD
    blReturn )                   ' Return Code (0: OK Exit >0: Error Code)

*****
' create graphic for slider
*****
call bTglCreateGraphic( 16, & ' width of graphic
    30, &                      ' height of graphic
    CHNL_SLIDER, &              ' address of bitmap
    16, &                      ' width of bitmap
    SLIDERBUTTON_ID, &          ' unique ID of this graphic
    blReturn )                  ' Return Code (0: OK Exit >0: Error Code)

```

```

*****
' link graphic to sidebar
*****
call bTglLink(SLIDEBAR_ID, & ' unique ID of sidebar
SLIDEBUTTON_ID, & ' unique ID of slider button graphic
blReturn) ' Return Code (0: OK Exit >0: Error Code)

*****
' create sidebar
*****
call bTglCreateSliderDockWnd( 31, & ' width of slider
170, & ' height of slider
200, & ' minimum value 1
-200, & ' maximum value 1
200, & ' minimum value 2
-200, & ' maximum value 2
"Y", & ' type of slider
CHNL_SLIDEBAR, & ' address of bitmap
32, & ' width of bitmap
SLIDEBAR_ID, & ' unique ID of the parent element
SLIDEBAR_ID2, & ' unique ID of this sidebar
WINDOW_ID, & ' window ID to create this slider in
10, & ' X-coordinate offset from docking point
0, & ' Y-coordinate offset from docking point
TGL_OPT_RIGHT, & ' docking option
blReturn ) ' Return Code (0: OK Exit >0: Error Code)

*****
' link graphic to sidebar
*****
call bTglLink(SLIDEBAR_ID2, & ' unique ID of sidebar
SLIDEBUTTON_ID, & ' unique ID of slider button graphic
blReturn) ' Return Code (0: OK Exit >0: Error Code)

*****
' show window
*****
call bTglShowWindow( WINDOW_ID, & ' window ID to show
blReturn ) ' Return Code (0: OK Exit >0: Error Code)

while 1=1
call lTglGetSliderValue(SLIDEBAR_ID, & ' unique ID of sidebar
WINDOW_ID, & ' unique ID of window
TGL_SL_OPT_VALUE, & ' option: read out current value of slider
lSliderValue, & ' Return value: current value of slider
blReturn) ' Return Code (0: OK Exit >0: Error Code)
endwhile

CHNL_SLIDEBAR::
data filter "chnl_sidebar.bmp", "GRAPHFLT", 0 ' WxH=31x170 BmpW=32
CHNL_SLIDER::
data filter "chnl_slider_black.bmp", "GRAPHFLT", 0 ' WxH=16x30 BmpW=16
end

```

bTglSetSliderValue

call bTglSetSliderValue( ElementId, WindowId, Value, Result )

Function: Sets the value of the selected slider and moves the slider button to the correct position. You can change the value of a slider every time, even if the window of the slide bar is not active. The slider button will be moved on the new position automatically.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
Value	-	-	●	-	-	slider is set to this value. If the slide bar has a slider button, the button is moved to the correct position
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error



ITglGetSliderValue

call ITglGetSliderValue( ElementId, WindowId, Option, Value, Result )

Function: Reads out current slider value of selected slider. You can read out the slider value all time, even if the window of the slider is not active. The actual value is saved permanent in RAM.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of slide bar
WindowId	-	●	-	-	-	unique identifier of window
Option	●	-	-	-	-	selects the value to read out TGL_SL_OPT_VALUE: reads out the current slider value TGL_SL_OPT_COORD: reads out the current position of the slider button on Lcd.
Value	-	-	●	-	-	<b>Return Values:</b> requested value is saved in this variable error code, for details see table of error codes 0 ok >0 error
Result	●	-	-	-	-	

## Slider

Sample program:

```
-----
' TGL_SLIDER_X_SHOW_VALUE.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0
' fonts
#define FONT_ID                0

task main
    datalabel dlSliderButton, dlSlideBar
    byte blReturn              ' return value of tgl subroutines
    word wlElementId          ' current identifier for elements
    word wlSLIDER_ID, wlLABEL_ID ' "constant" identifier for label
    long llSliderValue

    #include TGL_DEVICE_DRIVERS_TP1000.INC

    *****
    ' INITIALIZATION
    *****
    call vTglInit()
    wlElementId = 0

    *****
    ' TGL FONTS
    *****
    call bTglCreateFontParams( &
    FONT_ID, &                  ' identifier of font
    "Valencia",10,"normal", &   ' name, size, type of font
    "right", "center", &       ' alignment horizontal, vertical
    "const center", 0, &       ' spacing type, blank
    -6, 0, &                   ' spacing char, vertical
    "imm", "char", &           ' overlay, wrap mode
    blReturn )                 ' return code (0: OK exit >0: error exit)

    *****
    ' TGL ELEMENTS AND WINDOWS
    *****
    ' create and place slide bar
    call bTglCreatesliderWnd( &
    168, 32, &                  ' width, height of element
    -1599, 1600,&               ' min, max value main direction
    0, 0, &                    ' min, max value second direction or dummy
    "X", &                     ' type of slider (main direction)
    dlSlideBar, 168, &         ' address, format width of bitmap
    wlElementId, WINDOW_ID, &  ' identifier of element, window
    76, 120, &                 ' x, y coordinate on LCD
    blReturn )                 ' return code (0: OK exit >0: error exit)
    ' save identifier for linking and later use
    wlSLIDER_ID = wlElementId
```

```

'' increment identifier for next element
wElementId = wElementId + 1
' create slider button
call bTglCreateGraphic( &
32, 16, &                                ' width, height of element
dLSliderButton, 32, &                    ' address, format width of bitmap
wElementId, &                            ' identifier of element
bIReturn )                                ' return code (0: OK exit >0: error exit)
' link slider button to slide bar
call bTglLink( &
wSLIDER_ID, &                            ' identifier of slide bar
wElementId, &                            ' identifier of slider button
bIReturn )                                ' return code (0: OK exit >0: error exit)
'' increment identifier for next element
wElementId = wElementId + 1

' label for displaying the slider value
call bTglCreateLabelVarWnd( &
80, 30, &                                ' width, height of element
FONT_ID, 3, &                            ' font identifier, frame thickness
wElementId, WINDOW_ID, &                ' identifier of element, window
120, 60, &                                ' x, y coordinate on LCD
bIReturn )                                ' return code (0: OK exit >0: error exit)
'' save identifier for later use
wLABEL_ID = wElementId
'' increment identifier for next element
wElementId = wElementId + 1

*****
' show current slider value
*****
call bTglShowWindow( WINDOW_ID, bIReturn )
loop 7FFFFFFFh
    call lTglGetSliderValue( &
wSLIDER_ID, WINDOW_ID, &                ' identifier of slider, window
TGL_SL_OPT_VALUE, &                    ' option: read out current value/position
lISliderValue, &                        ' returned current slider value/position
bIReturn )                                ' return code (0: OK exit >0: error exit)
    call bTglShowLong( wLABEL_ID, lISliderValue, TRUE, bIReturn )
    wait_duration 100
endloop

*****
' FLASH
*****
dLSlideBar::
data filter "chnl_x_slidebar.bmp",      "GRAPHFLT", 0 ' WxH=168x32 BmpW=168
dLSliderButton::
data filter "chnl_x_slider_black.bmp", "GRAPHFLT", 0 ' WxH=32x16 BmpW=32
end

```

bTglSetSliderLimits

call bTglSetSliderLimits( ElementId, WindowId, Top,Bottom, Left,Right, Result )

Function: Sets the limits of the selected slider and moves the slider button to the correct position, if actually shown. Actual slider value is limited by slider limits.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
Top, Bottom	-	-	●	-	-	limits y axis, dummys for x sliders
Left, Right	-	-	●	-	-	limits x axis, dummys for y sliders
Result	●	-	-	-	-	

Return Values:

error code, for details see table of error codes  
0 ok  
>0 error

# Listbox

Listboxes are assembled elements of a view for the list items and two buttons or a slider for scrolling. Listboxes can be used for an easy user input of a determined selection of items.

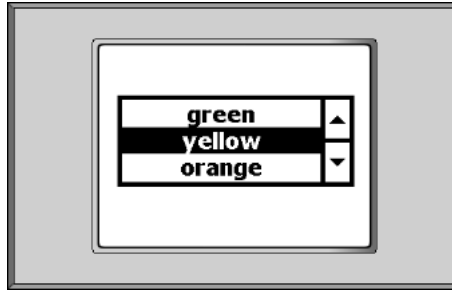


figure 66: Listbox

Listboxes can be used like the simple elements by creating, placing and showing them. All you have to do is passing a list of items with the creation of this element. The scrolling function will be internally administrated. You need not care for this. For getting an item just call the getting function.

Available subroutines for Listboxes:

- *bTglCreateListbox*
- *bTglPlaceListboxInWindow*
- *bTglDockListboxInWindow*
- *bTglCreateListboxWnd*
- *bTglCreateListboxDockWnd*
- *sTglGetListboxItem*
- *wTglGetListboxIndex*
- *bTglSetListboxIndex*

bTglCreateListbox

call bTglCreateListbox( Width, Height, List\$, FontId, Frame, WidthBut, KeyAttr, & ElementId, Result )

Function: Creates a new listbox with all items in the passed list.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
List\$	-	-	-	●	-	item list
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	frame and line thickness
WidthBut	-	●	-	-	-	scroll button width
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off
ElementId	-	●	-	-	-	unique identifier of this element
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

bTglPlaceListboxInWindow

```
call bTglPlaceListboxInWindow( ElementId, WindowId, X,Y, Result )
```

Function: Places a listbox in a window at position XY with the passed starting value.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge

Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error
--------	---	---	---	---	---	---

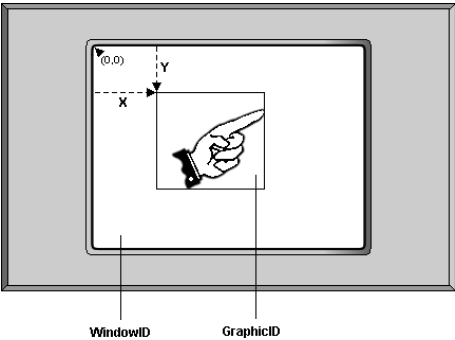


figure 67: Place listbox in window

bTglDockListboxInWindow

call bTglDockListboxInWindow( ParentId, ChildId, WindowId, XRel, YRel, Option, & Result )

Function: Places a listbox in a window by docking the child element next to the parent element in one of 8 directions.

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.



bTglCreateListboxWnd

call bTglCreateListboxWnd( Width, Height, List\$, FontId, Frame, WidthBut, KeyAttr, & ElementId, WindowId, X,Y, Result )

Function: Creates a listbox and places it to a window at position XY.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
List\$	-	-	-	●	-	item list
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	frame and line thickness
WidthBut	-	●	-	-	-	scroll button width
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

bTglCreateListboxDockWnd

call bTglCreateListboxDockWnd( Width, Height, List\$, FontId, Frame, WidthBut, & KeyAttr, ParentId, ChildId, WindowId, XRel, YRel, & Option, Result )

Function: Creates a listbox and places it in a window by docking the child element next to the parent element in one of 8 directions.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
List\$	-	-	-	●	-	item list
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	frame and line thickness
WidthBut	-	●	-	-	-	scroll button width
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

## Listbox

Sample program:

```
-----
' TGL_LISTBOX_createWnd.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' elements
#define LISTBOX_ID              0
#define LABEL_ID_KEY            1
#define LABEL_ID_SELECT        2
' windows
#define WINDOW_ID               0
' fonts
#define FONT_ID                 0

task main
  byte blReturn
  string slItem$(40h)

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  set_len$( slItem$, 0 )

  *****
  ' create font
  *****
  call bTglCreateFontParams(      &
    FONT_ID,                    &      ' font identifier
    "Valencia", 10, "normal",    &      ' font name, size, type
    "center","center",         &      ' alignment horizontal, vertical
    "prop",0,SPACING_CHAR_DEFAULT,0, &      ' spacing type, blank, char, vert.
    "imm", "char",              &      ' overlay, wrap mode
    blReturn )                  ' return message    0: Ok, >0: Error

  *****
  ' create a listbox and place it in a window
  *****
  call bTglCreateListBoxWnd(      &
    260,120,                    &      ' width, height of element
    "<0dh>s0<0dh>s1<0dh>s2<0dh>s3<0dh>s4<0dh>s5<0dh>s6<0dh>s7<0dh><0dh>", &
    &      ' list to be shown in element
    FONT_ID,                    &      ' font identifier
    5, 49,                      &      ' width of frame, button
    TGL_KEY_ATTR_DEFAULT,       &      ' key attrib. for touchpanel driver
    LISTBOX_ID, WINDOW_ID,      &      ' identifier of element, window
    30, 30,                     &      ' x,y coordinate in window
    blReturn )                  ' return message    0: Ok, >0: Error

  *****
```

## Listbox

```
' show window
'*****
call bTglShowWindow( WINDOW_ID, blReturn )

loop 7FFFFFFh
  call sTglGetListboxItem( LISTBOX_ID, slItem$, blReturn )
  wait_duration 200
endloop
end
```

sTglGetListboxItem

```
call sTglGetListboxItem( ElementId, Item$, Return
```

Function: Returns current item of listbox.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Item\$	-	-	●	-	-	current item of list
Result	●	-	-	-	-	error code, for details see table of error codes

**Return Values:**  
current item of list  
error code, for details see table of error codes  
0 ok  
>0 error

wTglGetListboxIndex

call wTglGetListboxIndex( ElementId, Index, Return

Function: Returns current item of listbox.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Index	-	●	-	-	-	current index of list
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

Return Values:

current index of list  
error code, for details see table of error codes  
0 ok  
>0 error

bTglSetListboxIndex

call sTglSetListboxIndex( ElementId, Index, Return )

Function: Set new current index of listbox. If Listbox is placed and shown in actually window, listbox will be updated automatically.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Index	-	●	-	-	-	new current indexof list
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

Return Values:

error code, for details see table of error codes  
0 ok  
>0 error

## Listbox

Sample program:

```
-----
' TGL_LISTBOX_set_get_index.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' elements
#define LISTBOX_ID              0
#define LABEL_ID_KEY            1
#define LABEL_ID_SELECT        2
' windows
#define WINDOW_ID               0
' fonts
#define FONT_ID                 0

task main
  byte blReturn
  word wlIndex

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()

  *****
  ' create font
  *****
  call bTglCreateFontParams(      &
    FONT_ID,                     & ' font identifier
    "Valencia", 18, "bold",      & ' font name, size, type
    "center","center",          & ' alignement horizontal, vertical
    "prop",0,SPACING_CHAR_DEFAULT,0, & ' spacing type, blank, char, vert.
    "imm", "char",              & ' overlay, wrap mode
    blReturn )                  ' return message    0: Ok, >0: Error

  *****
  ' create a listbox and place it in a window
  *****
  call bTglCreateListBoxWnd(      &
    260,100,                     & ' width, height of element
    "<0dh>violet<0dh>blue<0dh>turquoise<0dh>green<0dh>yellow<0dh>orange<0dh>",&
    & ' list to be shown in element
    FONT_ID,                     & ' font identifier
    5, 32,                       & ' width of frame, button
    TGL_KEY_ATTR_DEFAULT,        & ' key attrib. for touchpanel driver
    LISTBOX_ID, WINDOW_ID,       & ' identifier of element, window
    32, 70,                     & ' x,y coordinate in window
    blReturn )                  ' return message    0: Ok, >0: Error

  *****
  ' show window
```

```
*****
call bTglShowWindow( WINDOW_ID, blReturn )
randomize
loop 7FFFFFFh
    call wTglGetListboxIndex( LISTBOX_ID, wlIndex, blReturn )
    wait_duration 1000
    wlIndex = rnd(0)*8 shr 16
    call bTglSetListboxIndex( LISTBOX_ID, wlIndex, blReturn )
    wait_duration 1000
endloop
end
```



## Gauge

Gauges can be used as static elements like pie or bar charts and as dynamical elements like radial or linear indicators. Typical use is for displaying dial indicators, filling levels or time states.

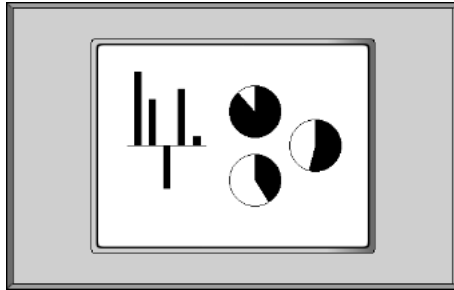


figure 68: Bar charts and pie charts

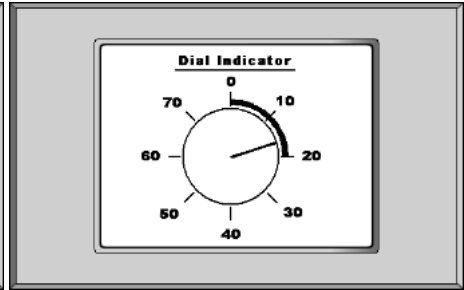


figure 69: Dial indicator

Gauges can be used like the simple elements by creating, placing and showing them. For dynamical use you just need to update the new value for this element.

Available subroutines for gauges:

- *bTglCreateGauge*
- *bTglPlaceGaugeInWindow*
- *bTglDockGaugeInWindow*
- *bTglCreateGaugeWnd*
- *bTglCreateGaugeDockWnd*
- *bTglShowGaugeValue*

bTglCreateGauge

call bTglCreateGauge( Width, Height, Min,Max, Type, Base, Frame, ElementId, Result )

Function:      Creates a new gauge.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
Min, Max	-	-	●	-	-	value limits
Type	●	-	-	-	-	type of gauge: radial or linear indicator bar or pie chart TGL_GA_TYPE_BAR TGL_GA_TYPE_PIE TGL_GA_TYPE_RADIAL
Base	-	●	-	-	-	position of minimum gauge value radial: 0..36000 centidegrees linear and radial: 0   =TGL_GA_BASE_RIGHT 9000 =TGL_GA_BASE_BOTTOM 18000 =TGL_GA_BASE_LEFT 27000 =TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness of outer circle resp. rectangle
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0   ok >0 error

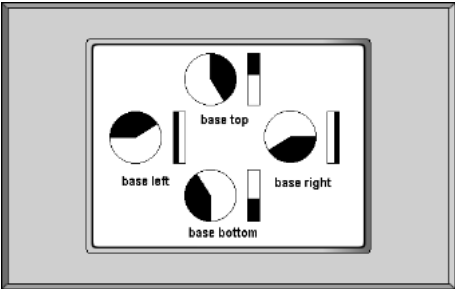


figure 70: Radial and linear gauge bases left, right, top, bottom

bTglPlaceGaugeInWindow

```
call bTglPlaceGaugeInWindow( ElementId, WindowId, X,Y, Value, Result )
```

Function: Places a Gauge in a window at position XY with the passed starting value.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
Value	-	-	●	-	-	starting value for gauge
Result	●	-	-	-	-	

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

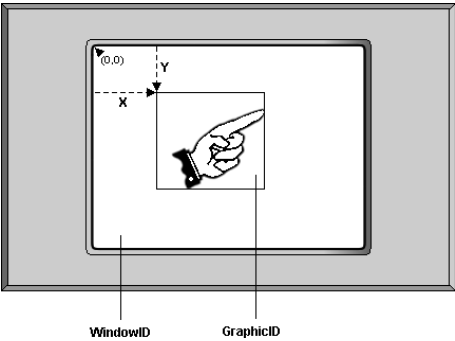


figure 71: Place Gauge in window

bTglDockGaugeInWindow

call bTglDockGaugeInWindow( ParentId, ChildId, WindowId, XRel, YRel, Option, & Value, Result )

Function: Places a Gauge in a window by docking the child element next to the parent element in one of 8 directions.

Parameters:

	B	W	L	S	F	
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT
Value	-	-	●	-	-	starting value for gauge
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For details about the docking subroutines please see the chapter *Docking*.

bTglCreateGaugeWnd

call bTglCreateGaugeWnd(Width, Height, Min,Max, Type, Base, Frame, & ElementId, WindowId, X,Y, Value, Result )

Function: Creates a Gauge and places it to a window at position XY.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
Min, Max	-	-	●	-	-	value limits
Type	●	-	-	-	-	type of gauge: radial or linear indicator bar or pie chart
Base	-	●	-	-	-	position of minimum gauge value radial: 0..36000 centidegrees linear and radial: 0 =TGL_GA_BASE_RIGHT 9000 =TGL_GA_BASE_BOTTOM 18000 =TGL_GA_BASE_LEFT 27000 =TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness of outer circle resp. rectangle
ElementId	-	●	-	-	-	unique identifier of this element
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
Value	-	-	●	-	-	starting value for gauge
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

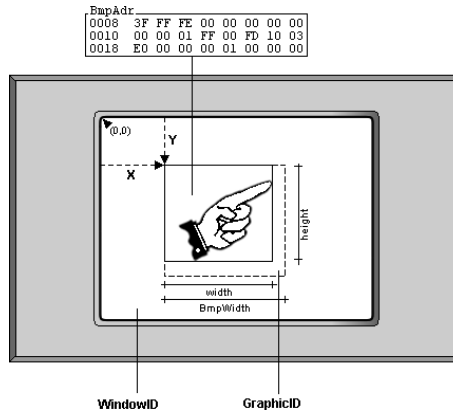


figure 72: Create and place Gauge in a window

## bTglCreateGaugeDockWnd

call bTglCreateGaugeDockWnd( Width, Height, Min,Max, Type, Base, Frame, & ParentId, ChildId, WindowId, XRel, YRel, Option, & Value, Result )

Function: Creates a Gauge and places it in a window by docking the child element next to the parent element in one of 8 directions.

### Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of element
Min, Max	-	-	●	-	-	value limits
Type	●	-	-	-	-	type of gauge: radial or linear indicator bar or pie chart
Base	-	●	-	-	-	position of minimum gauge value radial: 0..36000 centidegrees linear and radial: 0 =TGL_GA_BASE_RIGHT 9000 =TGL_GA_BASE_BOTTOM 18000 =TGL_GA_BASE_LEFT 27000 =TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness of outer circle resp. rectangle
ParentId	-	●	-	-	-	unique identifier of existing element in window
ChildId	-	●	-	-	-	unique identifier of new element in window
WindowId	-	●	-	-	-	unique identifier of window
XRel, YRel	-	●	-	-	-	relative coordinates of left top edge to docking point
Option	●	-	-	-	-	defines docking points in one of 8 directions TGL_OPT_RIGHT TGL_OPT_BOTTOM_RIGHT TGL_OPT_BOTTOM TGL_OPT_BOTTOM_LEFT TGL_OPT_LEFT TGL_OPT_TOP_LEFT TGL_OPT_TOP TGL_OPT_TOP_RIGHT

Gauge

	B	W	L	S	F	Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

For details about the docking subroutines please see the chapter *Docking*.



## Gauge

Sample program:

```
'-----
' TGL_GAUGE.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows
#define WINDOW_ID              0

task main
  byte blReturn                ' return value of tgl subroutines
  word wElementId              ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  wElementId = 0

  *****
  ' TGL ELEMENTS AND WINDOWS
  *****
  call bTglCreateGaugeWnd( &
    160,40, &                  ' size of the element
    0,100, &                   ' limits of values
    TGL_GA_TYPE_BAR, TGL_GA_BASE_LEFT, & ' chart type, location of base
    2, &                       ' frame thickness
    wElementId,WINDOW_ID, &    ' identifier of the element, window
    80,100, &                  ' coordinates in the window
    40, &                     ' saved value for element
    blReturn )                 ' return code (0=Ok, >0=Error)

  *****
  ' show window
  *****
  call bTglShowWindow( WINDOW_ID, blReturn )
end
```

## bTglShowGaugeValue

call `bTglShowGaugeValue( ElementId, Value, UpdateFlag, Result )`

Function: Show gauge with the passed value in internal string and update LCD output if update flag has been set.

### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
Value	-	-	●	-	-	value for gauge
UpdateFlag	●	-	-	-	-	TGL_TRUE: update LCD output TGL_FALSE: no LCD update

Result

**Return Values:**  
error code, for details see table of error codes  
0 ok  
>0 error

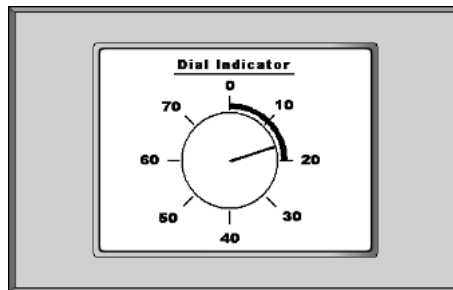


figure 73: Dial Indicator

Sample program:

```

-----
' TGL_GAUGE_dial_indicator.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

*****
' IDENTIFIER
*****
' windows

```

```
#define WINDOW_ID      0

long lgMin,lgMax ' limits of gauge
long lgValue     ' simulated value

task main
    datalabel dlMask      ' flash address for bitmap of background mask
    byte blReturn         ' return value of tgl subroutines
    word wlElementId      ' current identifier for creation of elements
    word wlGAUGE_ID       ' "constant" identifier for gauge
    long llValue          ' marker for changed value of lgValue

    #include TGL_DEVICE_DRIVERS_TP1000.INC

    *****
    ' INITIALIZATION
    *****
    call vTglInit()
    wlElementId = 0

    *****
    ' TGL ELEMENTS AND WINDOWS
    *****
    ' hand for dial indicator
    lgMin = 0
    lgMax = 8000
    call bTglCreateGaugeWnd( &
    118,118, &                ' size of the element
    lgMin,lgMax, &            ' limits of values
    TGL_GA_TYPE_PIE, 27000, & ' type of gauge, location of base
    3, &                      ' frame thickness
    wlElementId,WINDOW_ID, &  ' identifier of the element, window
    94,75, &                  ' coordinates in the window
    0, &                      ' saved value for element
    blReturn )                ' return code (0=Ok, >0=Error)
    ' save identifier for further processing
    wlGAUGE_ID = wlElementId
    ' increment current identifier for next element
    wlElementId = wlElementId + 1

    ' background mask for dial indicator
    call bTglCreateGraphicWnd( &
    LCD_WIDTH,LCD_HEIGHT, &   ' width, height of element
    dlMask, LCD_WIDTH, &      ' address, format width of bitmap
    wlElementId, WINDOW_ID, & ' identifier of element, window
    0,0, &                    ' x,y coordinate on LCD
    blReturn )                ' return code (0: OK exit >0: error exit)
    call bTglSetAttribute( &
    wlElementId, WINDOW_ID, & ' identifier of element, window
    TGL_ATTR_SHOW_MODE,TGL_SHOW_MODE_OR, & ' attribute and value of element
    blReturn )                ' return code (0: OK exit >0: error exit)

    *****
    ' show working dial indicator
    *****
    run_task tSimulateValue
    call bTglShowWindow( WINDOW_ID, blReturn )

    ' update dynamic chart when value has been changed
```

```

llValue = 0
loop 7FFFFFFFh
  if llValue <> lgValue then
    llValue = lgValue
    call bTglShowGaugeValue( wlGAUGE_ID, llValue, TGL_TRUE, blReturn )
  else
    release_task
  endif
endloop

'*****
' FLASH
'*****

dlMask::
data filter "Mask_Dial_Indicator.bmp", "GRAPHFLT", 0 ' WxH=320x240 FW=320
end

'-----
' simulate value changings from 40 to 40 every 100ms
'-----

task tSimulateValue
  lgValue = 1000
  randomize
  loop 7FFFFFFFh
    lgValue = limit( lgValue + ((rnd(0)*81) shr 16) - 40, lgMin, lgMax )
    wait_duration 100
  endloop
end

```

# Keyboard

A keyboard realizes easy user input with the touch panel. A keyboard is a sample of several elements which are placed in one or more windows using some fonts. These samples of elements are arranged in different styles which are shown below.

If you want to integrate a keyboard in your application you just have to initialize the keyboard with the style and font of your choice calling the subroutine *bTglInitKeyboard*. The keyboard will be shown on the LCD after calling the subroutine *sTglGetKeyboardInput*. This subroutine returns a string containing the user input for further processing.

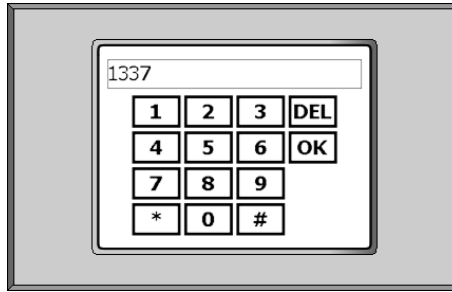


figure 74: Example for a keyboard

For a keyboard with the keys of your own choice you can call the subroutine *bTglInitKeyboardSelfmade*. You can handle this keyboard in the same way as the other keyboards by calling the subroutine *sTglGetKeyboardInput*.

Available subroutines:

- *bTglInitKeyboard*
- *bTglInitKeyboardSelfmade*
- *sTglGetKeyboardInput*

bTglInitKeyboard

call bTglInitKeyboard( Style, FontId, ElementId, WindowId, ElemIdKeybTxt, Result )

Function:        Initializes a keyboard for user input.

Parameters:

	B	W	L	S	F	
Style	-	●	-	-	-	look of keyboard
FontId	●	-	-	-	-	identifier for font of text in label for user input
ElementId	-	●	-	-	-	<b>Return Values:</b> IN: identifier of the first element of the keyboard OUT: first free identifier for the creation of next elements
WindowId	-	●	-	-	-	IN: identifier of the first window to place in the elements for the keyboard OUT: identifier of the next window to place in elements
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
Result	●	-	-	-	-	error code, for details see table of error codes 0    ok >0  error

! Mind creating the font for the keyboard user input for a correct initialization of the keyboard.

! Mind saving the identifiers of the first keyboard window and the label for the user input for calling the subroutine *sTglGetKeyboardInput*.

Internally the function creates a button for each key of the keyboard, a label for the user input and places them in windows. The number of used buttons and windows depends on the chosen style.

! The identifiers between the given first identifier for an element or a window and the returned identifier may NOT be used for other elements or windows. They are reserved for the elements which are assembled to one keyboard.

Needed numbers of identifiers for fonts, elements and windows for the

Keyboard

keyboards styles:

style	fonts	elements	windows
TGL_KEYB1_STYLE_ENG TGL_KEYB1_STYLE_GER TGL_KEYB1_STYLE_TRK TGL_KEYB1_STYLE_HUN	1	113 total 112 buttons 1 label	2
TGL_DIGSTYLE_1	1	15 total 14 buttons 1 label	1

TGL\_DIGSTYLE\_1



figure 75: Keyboard style digit block

TGL\_KEYB1\_STYLE\_ENG



figure 76: Keyboard style English unshifted



figure 77: Keyboard style English shifted

## Keyboard

### TGL\_KEYB1\_STYLE\_GER



figure 78: Keyboard style German unshifted



figure 79: Keyboard style German shifted

### TGL\_KEYB1\_STYLE\_TRK:



figure 80: Keyboard style Turkey unshifted



figure 81: Keyboard style Turkey shifted

### TGL\_KEYB1\_STYLE\_HUN:



figure 82: Keyboard style Hungarian unshifted



figure 83: Keyboard style Hungarian shifted



## Keyboard

Sample program:

```
'-----
TGL_KEYBOARD_STYLES.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

#define NUM_KEYBOARDS          4

task main
  byte blReturn                ' return value of tgl subroutines
  word wlElementId             ' current identifier for creation of
elements
  word wlWindowId              ' current identifier for creation of windows
  byte blFontId                ' current identifier for creation of fonts
  ' identifier of label for user input
  array wlELEM_ID_KEYB_TXT(NUM_KEYBOARDS) of word
  ' identifier of window for keyboard
  array wlWND_ID_KEYBOARD(NUM_KEYBOARDS) of word
  byte blFONT_ID_KEYB_VIEW      ' identifier of font for user input
  string slInput$(100h)         ' buffer for user input
  array wlStyle(NUM_KEYBOARDS) of word ' style of keyboard
  byte blIdx                    ' index of keyboard style

#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'  INITIALIZATION
'*****
call vTglInit()
wlElementId = 0
wlWindowId  = 0
blFontId    = 0
set_len$( slInput$, 0 )
wlStyle(0) = TGL_KEYB1_STYLE_ENG
wlStyle(1) = TGL_KEYB1_STYLE_GER
wlStyle(2) = TGL_KEYB1_STYLE_TRK
wlStyle(3) = TGL_DIGSTYLE_1

'*****
' create font for user input
'*****
' save identifier of font for user input
blFONT_ID_KEYB_VIEW = blFontId
call bTglCreateFontParams( &
  blFontId, &                ' identifier of font
  "Valencia", 18, "normal", & ' name, size, type of font
  "left", "top", &           ' alignment horizontal, vertical
  "prop", 0, &               ' spacing type, blank
  SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
  "imm", "char", &          ' overlay, wrap mode
  blReturn )                 ' return code (0: OK exit  >0: error exit)
```

```
*****
' initialize keyboards
*****
for blIdx = 0 to NUM_KEYBOARDS-1
  ' save identifier of first keyboard window
  wlWND_ID_KEYBOARD(blIdx) = wlWindowId
  call bTglInitKeyboard( wlStyle(blIdx), blFONT_ID_KEYB_VIEW, &
    wlElementId, wlWindowId, wlELEM_ID_KEYB_TXT(blIdx), blReturn )
next

*****
' show keyboard styles
*****
blIdx = 0
loop 7FFFFFFFh
  ' starting text for label for user input
  slInput$ = "User Input:"
  call sTglGetKeyboardInput( wlWND_ID_KEYBOARD(blIdx), &
    wlELEM_ID_KEYB_TXT(blIdx), 0FFh, slInput$, blReturn )

  '+++++
  '
  ' here you can analyse the user input, stored in slInput$
  '
  '+++++

  ' show next style
  blIdx = mod( blIdx+1, NUM_KEYBOARDS )
endloop
end
```

bTglInitKeyboardSelfmade

call bTglInitKeyboardSelfmade( FontIdView, FontIdKey, Frame, Distance, X, Y, Width, Height, KeyCodes\$, ElementId, WindowId, ElemIdKeybTxt, Result )

Function:        Initializes a keyboard for user input with the given keys of visible characters (no control characters).

Parameters:

	B	W	L	S	F	
FontIdView	●	-	-	-	-	identifier for font in label for user input
FontId	●	-	-	-	-	identifier for font in label for keys
Frame	●	-	-	-	-	frame thickness of keys
Distance	●	-	-	-	-	space between buttons
X, Y	-	●	-	-	-	x, y coordinate of top left edge of top left key
Width, Height	-	●	-	-	-	size of key
KeyCodes\$	-	-	-	●	-	keycodes and layout codes for keyboard

Return Values:

ElementId	-	●	-	-	-	IN: identifier of the first element of the keyboard OUT: first free identifier for the creation of next elements
WindowId	-	●	-	-	-	IN: identifier of the first window to place in the elements for the keyboard OUT: identifier of the next window to place in elements
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
Result	●	-	-	-	-	error code, for details see table of error codes 0    ok >0   error

Internally the function creates a text button for each key of the keyboard, a label for the user input and places them in windows. The number of used buttons and windows depends on the chosen style.

!        The identifiers between the given first identifier for an element or a window and the returned identifier may NOT be used for other elements or windows. They are reserved for the keyboard.

## Keyboard

Mind creating the fonts for the keyboard user input and the keys for a correct initialization of the keyboard.

Mind saving the identifiers of the keyboard window and the label for the user input for calling the subroutine *sTglGetKeyboardInput*.

The keys are arranged in a block. The first key in the string *KeyCodes\$* is for the top left key. The second key will be placed on the right side of the first key. For placing a key in the next row you have to insert the control character *\_NEXT\_ROW*. For your selfmade Keyboard there are two control keys for deleting the last character *\_DEL* and for finishing the user input *\_OK*.

Control characters in the keycode string *KeyCodes\$* for the keyboard:

control character	description
_NEXT_ROW	place the next key in the next row
_NEXT_COL	place the next key in the next column
_NEXT_KEYB	place the next key in the next key set
_OK	finish the user input
_DEL	delete the last character
_CHANGE	change the key set
_PLUSMINUS	multiplies a numerical input value with -1
_DOT	set unique decimal point for a numerical input value

Needed numbers of identifiers for fonts, elements and windows for the selfmade keyboard:

fonts	elements	windows
2	1 text button for each key 1 label	1

### Example 1 *KeyCodes\$*

```
KeyCodes$ = "123"+chr$(_DEL)+chr$(_NEXT_ROW) & ' keys row 1  
+          "456"+chr$(_OK) +chr$(_NEXT_ROW) & ' keys row 2  
+          "789"          +chr$(_NEXT_ROW) & ' keys row 3  
+          "*0#"          ' keys row 4
```



figure 84: Example 1 for selfmade keyboard

### Example 2 *KeyCodes\$*

```
KeyCodes$ = "789"+chr$(_DEL)+chr$(_NEXT_ROW) & ' keys row 1  
+          "456"+chr$(_OK) +chr$(_NEXT_ROW) & ' keys row 2  
+          "123"          +chr$(_NEXT_ROW) & ' keys row 3  
+chr$(_NEXT_COL)+"0"          ' keys row 4
```

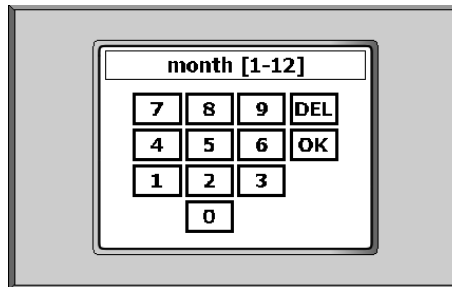


figure 85: Example 2 for selfmade keyboard

## Keyboard

You can widen a button to a multiple width by passing a keycode twice or more. This is often used for functional buttons like blank or ok button.

By passing the code `_NEXT_KEYB` a second keyboard will be generated e.g. for shifted keys. (see example `TGL_KEYBOARD_selfmade_shift.TIG.`)

### Example 3 *KeyCodes\$*

```
slKeyCodes$ = "1234567890"+chr$(_DEL)+chr$(_DEL) +chr$(_NEXT_ROW) & ' row 1
+          "qwertyuiopü+" +chr$(_NEXT_ROW) & ' row 2
+          "asdfghjklöä#" +chr$(_NEXT_ROW) & ' row 3
+          "<60>yxcvbnm,.-@" +chr$(_NEXT_ROW) & ' row 4
+ chr$(_CHANGE)+chr$(_CHANGE)+"[ ] ^\"+chr$(_OK)+chr$(_OK) & ' row 5
+ chr$(_NEXT_KEYB) &
+          "!<34>$%<38>/()="+chr$(_DEL)+chr$(_DEL) +chr$(_NEXT_ROW) & ' row 1
+          "QWERTZUIOPÜ*" +chr$(_NEXT_ROW) & ' row 2
+          "ASDFGHJKLÖÄ<39>" +chr$(_NEXT_ROW) & ' row 3
+          "<62>YXCVBNM;:_|" +chr$(_NEXT_ROW) & ' row 4
+ chr$(_CHANGE)+chr$(_CHANGE)+"{ } °~"+chr$(_OK)+chr$(_OK) ' row 5
+chr$(_NEXT_COL)+"0"
```

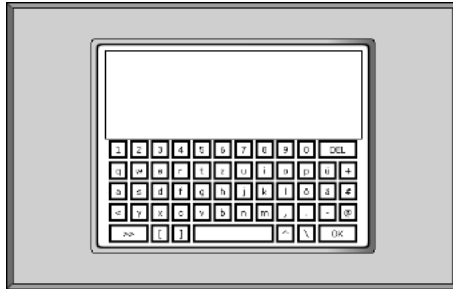


figure 86: unshifted keys

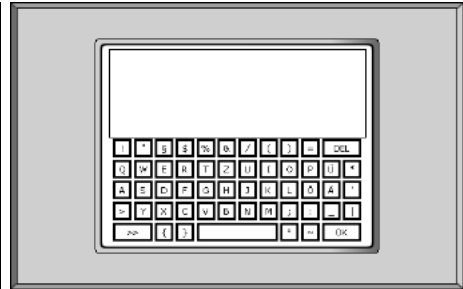


figure 87: shifted keys

## Keyboard

Sample program:

```
'-----
TGL_KEYBOARD_selfmade.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal      for "white" LCD
'                                  ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

#define NUM_KEYBOARD_STYLES 6

task main
  byte blReturn      ' return value of tgl subroutines
  word wElementId    ' current identifier for creation of elements
  word wWindowId     ' current identifier for creation of windows
  byte blFontId      ' current identifier for creation of fonts
  word wELEM_ID_KEYB_TXT ' identifier of label for user input
  word wWIND_ID_KEYBOARD ' identifier of window for keyboard
  byte blFONT_ID_KEYB_VIEW ' identifier of font for user input
  byte blFONT_ID_KEYB_KEY ' identifier of font for key
  string sInput$(100h) ' buffer for user input
  byte blFrame       ' frame thickness of keys
  word wDistance     ' space between keys
  word wLX, wLY      ' top left edge of top left key
  word wLWidth, wLHeight ' key size
  string sKeyCodes$(80) ' keycodes and layout codes for keyboard

#include TGL_DEVICE_DRIVERS_TP1000.INC

'*****
'  INITIALIZATION
'*****
call vTglInit()
wElementId = 0
wWindowId  = 0
blFontId   = 0
set_len$( sInput$, 0 )

'*****
'  create your own keyboard style here
'*****
blFrame      = 4
wDistance    = 4
wLWidth      = 60
wLHeight     = 40
' place a block of 4x4 keys in the middle of the free LCD
' under the label for the user input (height=36)
wLX          = (LCD_WIDTH - (4*wLWidth+3*wDistance))/2
wLY          = 36 + (LCD_HEIGHT - 36 - (4*wLHeight+3*wDistance))/2
sKeyCodes$ = "123"+chr$( _DEL )+chr$( _NEXT_ROW ) & ' keys row 1
+           "456"+chr$( _OK )+chr$( _NEXT_ROW ) & ' keys row 2
+           "789" +chr$( _NEXT_ROW ) & ' keys row 3
+           "*0#" ' keys row 4

call bTglCreateFontParams( &
  blFontId, & ' identifier of font
  "Valencia", 18, "normal", & ' name, size, type of font
  "left", "top", & ' alignment horizontal, vertical
  "prop", 0, & ' spacing type, blank
  SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
```

```

"imm", "char", &                                ' overlay, wrap mode
blReturn )                                       ' return code (0: OK exit  >0: error exit)
' save identifier of font for user input
blFONT_ID_KEYB_VIEW = blFontId

blFontId = blFontId + 1
call bTglCreateFontParams( &
blFontId, &                                     ' identifier of font
"Valencia", 18, "bold", &                       ' name, size, type of font
"center", "center", &                           ' alignment horizontal, vertical
"prop", 0, &                                     ' spacing type, blank
SPACING_CHAR_DEFAULT, 0, &                       ' spacing char, vertical
"imm", "char", &                                 ' overlay, wrap mode
blReturn )                                       ' return code (0: OK exit  >0: error exit)
' save identifier of font for keys
blFONT_ID_KEYB_KEY = blFontId

*****
' initialize keyboard
*****
' save identifier of keyboard window
wlWND_ID_KEYBOARD = wlWindowId
call bTglInitKeyboardSelfmade( blFONT_ID_KEYB_VIEW, blFONT_ID_KEYB_KEY, &
blFrame, wlDistance, wlX, wlY, wlWidth, wlHeight, slKeyCodes$, &
wlelementId, wlWindowId, wleLEM_ID_KEYB_TXT, blReturn )

*****
' show selfmade keyboard style
*****
loop 7FFFFFFh
' starting text for label for user input
slInput$ = "Input number:"
call sTglGetKeyboardInput( wlWND_ID_KEYBOARD, wleLEM_ID_KEYB_TXT, &
0FFh, slInput$, blReturn )

'+++++
'
' here you can analyse the user input
' stored in the string buffer slInput$
'
'+++++

endloop
end

```



sTglGetKeyboardInput

call sTglGetKeyboardInput( WndIdUnshifted, ElemIdKeybTxt, MaxLenInput, Input\$, Result )

Function: Shows keyboard in an own window, informs the user about the wanted input, displays and return the user input.

Parameters:

	B	W	L	S	F	
WndIdUnshifted	-	●	-	-	-	identifier of the first keyboard window
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
MaxLenInput	-	●	-	-	-	maximal input length
Input\$	-	-	-	●	-	IN info text for the user OUT user input
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

The maximal input length is limited by 3 factors:

- *MaxLenInput*
- maximal length of *Input\$*
- maximal fitting characters in the label for the user input

For a longer input length you can choose a small and narrow font for the initialization of the keyboard, e.g VALENCIA\_8\_NORMAL.

## Keyboard

Sample program:

```
-----
' TGL_KEYBOARD_create.TIG
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal      for "white" LCD
'                                ' 1 = inversion for "blue" LCD
#include TigerGraphicLibrary.INC

task main
    byte blReturn                ' return value of tgl subroutines
    word wlElementId             ' current identifier for elements
    word wlWindowId              ' current identifier for creation of windows
    byte blFontId               ' current identifier for creation of fonts
    word wlELEM_ID_KEYB_TXT      ' identifier of label for user input
    word wlWND_ID_KEYBOARD       ' identifier of window for keyboard
    byte blFONT_ID_KEYB_VIEW     ' identifier of font for user input
    string slInput$(100h)       ' buffer for user input

#include TGL_DEVICE_DRIVERS_TP1000.INC
*****
' INITIALIZATION
*****
call vTglInit()
wlElementId = 0
wlWindowId  = 0
blFontId    = 0
set_len$( slInput$, 0 )

call bTglCreateFontParams( &
    blFontId, &                ' identifier of font
    "Valencia", 18, "normal", & ' name, size, type of font
    "left", "top", &           ' alignment horizontal, vertical
    "prop", 0, &               ' spacing type, blank
    SPACING_CHAR_DEFAULT, 0, & ' spacing char, vertical
    "imm", "char", &          ' overlay, wrap mode
    blReturn )                 ' return code (0: OK exit >0: error exit)
' save identifier of font for user input
blFONT_ID_KEYB_VIEW = blFontId

*****
' initialize keyboard
*****
' save identifier of keyboard window
wlWND_ID_KEYBOARD = wlWindowId
call bTglInitKeyboard( TGL_KEYB1_STYLE_ENG, blFONT_ID_KEYB_VIEW, &
    wlElementId, wlWindowId, wlELEM_ID_KEYB_TXT, blReturn )

*****
' show keyboard and get user input
*****
loop 7FFFFFFFh
    ' starting text for label for user input
    slInput$ = "Input text:"
    call sTglGetKeyboardInput( wlWND_ID_KEYBOARD, wlELEM_ID_KEYB_TXT, &
        0FFh, slInput$, blReturn )
    ' HERE YOU CAN ANALYZE THE USER INPUT SAVED IN slInput$
endloop
end
```

sTglGetKeybInputTimeout

```
call sTglGetKeybInputTimeout( WndIdUnshifted, ElemIdKeybTxt, MaxLenInput,
                               Input$, Timeout, Result )
```

Function: Shows keyboard in an own window, informs the user about the wanted input, displays and return the user input. Stop keyboard input after timeout and abort itself.

Parameters:

	B	W	L	S	F	
WndIdUnshifted	-	●	-	-	-	identifier of the first keyboard window
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
MaxLenInput	-	●	-	-	-	maximal input length
Input\$	-	-	-	●	-	IN info text for the user OUT user input
Timeout-	-	-	●	-	-	stop keyboard input after this idle time
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

- The maximal input length is limited by 3 factors:
- *MaxLenInput*
  - maximal length of *Input\$*
  - maximal fitting characters in the label for the user input

For a longer input length you can choose a small and narrow font for the initialization of the keyboard, e.g VALENCIA\_8\_NORMAL.

# RTC Applications

The RTC applications are for displaying and setting date and time of the real time clock. An RTC application is a sample of several elements which are placed in one or more windows using some fonts. These samples of elements are arranged in different styles which are shown below.

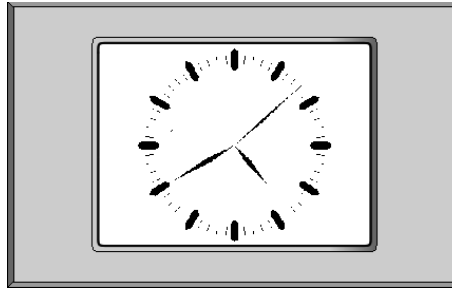


figure 88: Example for RTC application

If you want to integrate an RTC application in your application you just need to initialize the application with the style and font of your choice calling the subroutine *bTglInitRtc*. For setting the RTC call *bTglSetRtc*.

Available subroutines:

- *bTglInitRtc*
- *bTglSetRtc*

## bTgllnitRtc

call **bTgllnitRtc( Style, FontId, ElementId, WindowId, ElemIdRtc, Result )**

Function: Creates elements for an application for adjusting the real time clock and places them in windows.

### Parameters:

	B	W	L	S	F	
Style-	-	●	-	-	-	style of application for setting the real time clock
FontId	●	-	-	-	-	<b>Return Values:</b> IN: current identifier of fonts OUT: next free identifier of fonts IN: current identifier of elements OUT: next free identifier of element IN: current identifier of windows identifier of window for calling the application OUT: next free identifier of window identifier of element for calling the application error code, for details see table of error codes 0 ok >0 error
ElementId	-	●	-	-	-	
WindowId	-	●	-	-	-	
ElemIdRtc	-	●	-	-	-	
Result	●	-	-	-	-	

! Mind creating the font for the RTC application for a correct initialization.

! Mind saving the identifiers of the first RTC application window and the returned label for the for calling the subroutine *bTglSetRtc*.

Internally the function creates text buttons and labels for the displayed time and date, a button for each key of the keyboard for the user input, a label for the user input and places them in windows. Additionally there could be created some new fonts by changing the font parameters. The number of used fonts, elements and windows depends on the chosen style.

! The identifiers between the given first identifier for a font, an element or a window and the returned identifier may NOT be used for other fonts, elements or windows. They are reserved for this RTC application.

Needed numbers of identifiers for fonts, elements and windows for the RTC application styles:

style	fonts	elements	windows
TGL_RTC_STYLE_1	2	22 total 19 text buttons 3 labels	2

TGL\_RTC\_STYLE\_1:



figure 89: TGL\_RTC\_STYLE\_1 date and time



figure 90: TGL\_RTC\_STYLE\_1 keyboard

## bTglSetRtc

call `bTglSetRtc( WindowId, ElementId, Result )`

Function: Shows RTC application of chosen style and set the time using the user input.

### Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of element for RTC application
WindowId	-	●	-	-	-	identifier of window for RTC application
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

For setting the date or time, please touch the number on the LCD

### Sample program:

```

-----
' TGL_RTC_APPLICATION_STYLES.TIG
-----
#include TigerGraphicLibrary.INC

task main
    *****
    ' touch panel vars
    *****
    word wLibuFill
    byte blKeycode

    *****
    ' TGL general vars
    *****
    byte blReturn      ' return value of tgl subroutines
    word wElementId    ' current identifier for creation of elements
    word wWindowId     ' current identifier for creation of windows
    byte blFontId      ' current identifier for creation of fonts
    
```

```

*****
' RTC vars
*****
string sInput$(100h) ' buffer for user input
byte blFONT_ID_rtc ' identifier for font of rtc application
word wlWND_ID_rtc ' identifier for window of rtc application
word wlELEM_ID_rtc ' identifier for element of rtc application

*****
' device drivers
*****
#include TGL_DEVICE_DRIVERS_TP1000.INC
install_device #RTC, "RTC1.TD2"

*****
' INITIALIZATION
*****
call vTglInit()
wlElementId = 0
wlWindowId = 0
blFontId = 0
set_len$( sInput$, 0 )

*****
' RTC application
*****
blFONT_ID_rtc = blFontId ' save identifier of font for rtc
call bTglCreateFont( &
blFontId, & ' identifier of font
"Valencia", 18, "bold", & ' name, size, type of font
blReturn ) ' return code (0: OK exit >0: error exit)

wlWND_ID_rtc = wlWindowId
call bTglInitRtc( TGL_RTC_STYLE_1, blFontId, wlElementId, &
wlWindowId, wlELEM_ID_rtc, blReturn )

*****
' start application
*****
loop 7FFFFFFFh
call bTglSetRtc( wlWND_ID_rtc, wlELEM_ID_rtc, blReturn )
endloop
end

```



# Text Graphics

With the graphic fonts you can choose between many bitmap fonts of various sizes and types. The Tiger Graphic Library provides a tool to use these fonts as easy as every other font.



figure 91: Various graphic fonts

In addition to the choice of the font, its size and its type the Tiger Graphic Library provides further attributes for the layout of texts. You are able to determine the alignment and spacing of the characters. You can build texts with graphic fonts by wrapping lines after characters or words. You can build all texts on prepared backgrounds.

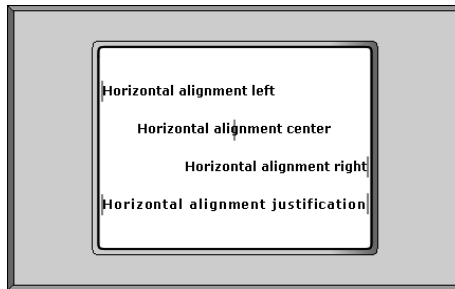


figure 92: Horizontal alignment

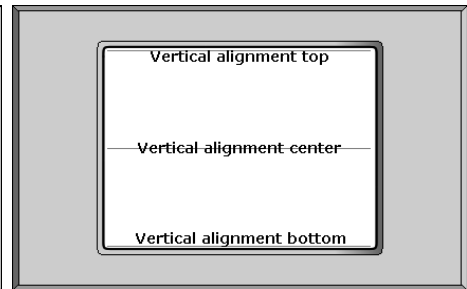


figure 93: Vertical alignment

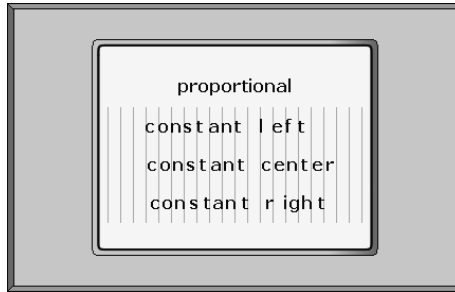


figure 94: Character spacing

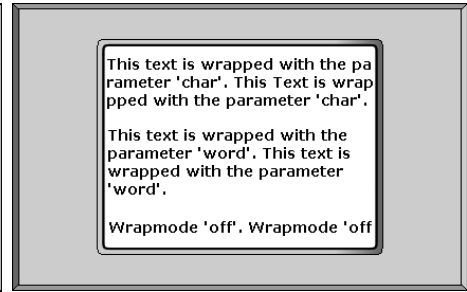


figure 95: Text wrapping

In the Tiger Graphic Library the graphic fonts are used with some types of elements like labels and text buttons.

Available subroutines for graphic fonts are:

- *bTglCreateFont*
- *bTglCreateFontParams*
- *bTglDeleteFont*
- *bTglSetFontBmp*
- *bTglGetFontBmp*
- *bTglSetFontParams*
- *bTglGetFontParams*
- *sTglBuildTextGraphic*
- *lTglCalcTextToWindow*
- *lTglGetFontHeight*
- *lTglCalcTextGraphicWidth*
- *lTglCalcLineWidths*

### Choosing the Graphic Fonts

Available fonts are:

Name	Type	Size
Amsterdam	bold	8,11,16,21
Atlanta	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Helsinki	normal	7,8,9,10,11,12,14,18,22,26
	bold	10,12,14,18,22,26,28,32,52,56,60
Istanbul	normal	8,10,11,12,14,18,21
Stockholm	normal	8,10,11,12,14,18,21
Tokio	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Valencia	normal, bold, italic, bold italic	8,10,11,12,14,18,21
	normal, bold	24,36,48

You choose the fonts by modifying the configuration file *TigerGraphicLibraryConf.inc*. Copy this file from the directory of the Tiger BASIC program on your PC into the same directory of the .TIG file of your project. Usually the path to the directory with the configuration file is *C:\Programme\Wilke Technology\Tiger Basic 5.3\TigerGraphicLibrary*.

! If you configure the Tiger Graphic Library for your project, NEVER change the original configuration file in the directory of the Tiger Graphic Library. This file with its standard configuration runs with all the examples of the Tiger Graphic Library. Normally the standard configuration would work with little programs you will write, too.

If you want to use e.g. VALENCIA\_18\_BOLD you should activate the following code lines in the configuration file:

```
#define VALENCIA_18_BOLD
#include TGL_GRAFO_VALENCIA.INC
```

For details about the configuration file of the Tiger Graphic Library see chapter *Configuration* and its sections for the graphic fonts.

### Specific Parameters of Graphic Fonts

For desining text graphics it could be helpful to know some font parameters to be able to calculate the exact pixel positions.

*Line height* is the height from the highest pixel to the lowest pixel of all characters in the font bitmap. If you need to verify the line height, just pass the additional number pixels with the parameter horizontal spacing. Passing a negative value for this parameter, you will get overlapping lines. This need not disturb, because there are only a few characters, which have pixels in the highest or lowest position.

*Max width* is the pixel width of the widest character in the font. This ist mostly *W*, *™* or *‰*.

Character spacing is the number of white pixels between 2 characters. For proportional spacing type there will be for each font bitmap default spacing stored in the flash memory. Passing the define *SPACING\_CHAR\_DEFAULT* with the functions *bTglCreateFont* or *bTglSetFontParams* for a proportional spacing type these spacings will be taken.

For *constant spacing types* the character with the maximal width would determine the distance of the characters. Going like this a zero has to be passed for this parameter. As normally only alpha numerical characters would be used this width would be too wide, so the default value should have a negative value. The value in the table takes attention to the most width alpha numeric character of the charset, mostly *W*, *M*, *H* or *m*. For a nice looking, you should pass *constant center* for the parameter *constant spacing*.

For a view of a variable value you should choose a constant spacing type. Otherwise if you choose proportional spacing type a flatterring will occur with each changing of the value because of the unequal widths of the digits. For an optimized spacing for constant digit distances, pass the define *SPACING\_CHAR\_DEFAULT\_DIGIT* with the functions *bTglCreateFont* or *bTglSetFontParams* and a constant spacing type.

font	line height	max. width	default character spacing (proportional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
AMSTERDAM_8_BOLD	11	12	1	0	-5
AMSTERDAM_11_BOLD	14	14	1	0	-4
AMSTERDAM_14_BOLD	21	21	2	0	-6
AMSTERDAM_21_BOLD	28	28	3	0	-8
ATLANTA_8_NORMAL	15	11	1	0	-4
ATLANTA_8_BOLD	15	11	1	0	-4
ATLANTA_8_ITALIC	15	12	1	-2	-6
ATLANTA_8_BOLD_ITALIC	14	12	1	-1	-4
ATLANTA_10_NORMAL	17	14	1	-1	-6
ATLANTA_10_BOLD	18	13	1	0	-5
ATLANTA_10_ITALIC	17	14	1	-1	-6
ATLANTA_10_BOLD_ITALIC	17	14	1	-1	-6
ATLANTA_11_NORMAL	18	15	1	0	-7
ATLANTA_11_BOLD	19	18	1	-5	-9
ATLANTA_11_ITALIC	18	16	1	-2	-7
ATLANTA_11_BOLD_ITALIC	18	17	1	-3	-8
ATLANTA_12_NORMAL	19	17	1	-2	-7
ATLANTA_12_BOLD	20	18	1	-3	-8
ATLANTA_12_ITALIC	20	17	1	-2	-7
ATLANTA_12_BOLD_ITALIC	19	17	1	-2	-7
ATLANTA_14_NORMAL	22	19	2	0	-7
ATLANTA_14_BOLD	23	21	2	-4	-9
ATLANTA_14_ITALIC	23	20	2	-3	-8
ATLANTA_14_BOLD_ITALIC	23	21	2	-2	-8
ATLANTA_18_NORMAL	28	25	2	0	-11
ATLANTA_18_BOLD	28	25	2	-2	-10

## Text Graphics

font	line height	max. width	default character spacing (proportional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
ATLANTA_18_ITALIC	29	27	2	-4	-12
ATLANTA_18_BOLD_ITALIC	28	26	2	-4	-11
ATLANTA_21_NORMAL	33	29	3	0	-11
ATLANTA_21_BOLD	34	29	3	-2	-11
ATLANTA_21_ITALIC	33	29	3	-2	-10
ATLANTA_21_BOLD_ITALIC	34	29	3	-3	-10
HELSINKI_7_NORMAL	11	7	1	0	-2
HELSINKI_8_NORMAL	12	9	1	0	-3
HELSINKI_9_NORMAL	14	9	1	0	-3
HELSINKI_10_BOLD	16	11	1	0	-3
HELSINKI_10_NORMAL	15	12	1	0	-3
HELSINKI_11_NORMAL	17	12	1	0	-4
HELSINKI_12_BOLD	17	12	1	0	-3
HELSINKI_12_NORMAL	18	14	1	0	-5
HELSINKI_14_BOLD	21	15	2	0	-4
HELSINKI_14_NORMAL	21	17	2	0	-6
HELSINKI_18_BOLD	26	19	2	0	-6
HELSINKI_18_NORMAL	26	22	2	0	-8
HELSINKI_22_BOLD	33	23	3	0	-6
HELSINKI_22_NORMAL	33	27	3	0	-9
HELSINKI_26_BOLD	40	28	3	0	-8
HELSINKI_26_NORMAL	39	33	3	0	-12
HELSINKI_28_BOLD	44	36	4	0	-10
HELSINKI_32_BOLD	50	34	4	0	-15
HELSINKI_52_BOLD	82	66	7	0	-22
HELSINKI_56_BOLD	88	72	8	0	-24
HELSINKI_60_BOLD	95	79	8	0	-27

font	line height	max. width	default character spacing (proportional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
ISTANBUL_8_NORMAL	14	11	1	-2	-5
ISTANBUL_10_NORMAL	17	13	1	-4	-5
ISTANBUL_11_NORMAL	19	15	1	-4	-6
ISTANBUL_12_NORMAL	20	16	1	-3	-7
ISTANBUL_14_NORMAL	21	17	2	-4	-7
ISTANBUL_18_NORMAL	29	25	2	-8	-13
ISTANBUL_21_NORMAL	36	29	3	-8	-13
STOCKHOLM_8_NORMAL	15	14	1	0	-7
STOCKHOLM_10_NORMAL	16	16	1	-1	-8
STOCKHOLM_11_NORMAL	17	18	1	-3	-10
STOCKHOLM_12_NORMAL	19	21	1	-4	-11
STOCKHOLM_14_NORMAL	20	22	2	-5	-11
STOCKHOLM_18_NORMAL	25	29	2	-5	-16
STOCKHOLM_21_NORMAL	31	35	3	-4	-18
TOKIO_8_NORMAL	14	7	1	0	-1
TOKIO_8_BOLD	15	9	1	0	-2
TOKIO_8_ITALIC	14	10	1	0	-3
TOKIO_8_BOLD_ITALIC	15	9	1	0	-1
TOKIO_10_NORMAL	19	10	1	0	-1
TOKIO_10_BOLD	19	12	1	-1	-3
TOKIO_10_ITALIC	18	13	1	0	-4
TOKIO_10_BOLD_ITALIC	19	12	1	0	-1
TOKIO_11_NORMAL	20	11	1	0	-2
TOKIO_11_BOLD	20	13	1	-1	-3
TOKIO_11_ITALIC	20	14	1	-1	-4
TOKIO_11_BOLD_ITALIC	20	15	1	-2	-3
TOKIO_12_NORMAL	21	12	1	0	-2

font	line height	max. width	default character spacing (proportional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
TOKIO_12_BOLD	21	13	1	0	-2
TOKIO_12_ITALIC	21	16	1	0	-5
TOKIO_12_BOLD_ITALIC	21	16	1	0	-3
TOKIO_14_NORMAL	25	14	2	0	-2
TOKIO_14_BOLD	26	18	2	-1	-5
TOKIO_14_ITALIC	25	18	2	0	-4
TOKIO_14_BOLD_ITALIC	25	18	2	0	-3
TOKIO_18_NORMAL	31	22	2	0	-6
TOKIO_18_BOLD	31	22	2	-1	-5
TOKIO_18_ITALIC	31	24	2	0	-6
TOKIO_18_BOLD_ITALIC	30	22	2	0	-2
TOKIO_21_NORMAL	35	24	3	0	-6
TOKIO_21_BOLD	36	27	3	0	-6
TOKIO_21_ITALIC	35	25	3	0	-3
TOKIO_21_BOLD_ITALIC	35	25	3	0	-2
VALENCIA_8_NORMAL	13	15	1	-6	-8
VALENCIA_8_BOLD	13	18	1	-8	-10
VALENCIA_8_ITALIC	13	15	1	-5	-8
VALENCIA_8_BOLD_ITALIC	13	18	1	-7	-10
VALENCIA_10_NORMAL	15	17	1	-6	-9
VALENCIA_10_BOLD	15	22	1	-10	-12
VALENCIA_10_ITALIC	15	17	1	-5	-8
VALENCIA_10_BOLD_ITALIC	15	21	1	-7	-10
VALENCIA_11_NORMAL	17	21	1	-8	-10
VALENCIA_11_BOLD	17	21	1	-7	-9
VALENCIA_11_ITALIC	17	21	1	-7	-9
VALENCIA_11_BOLD_ITALIC	21	17	1	-6	-8



font	line height	max. width	default character spacing (proportional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
VALENCIA_12_NORMAL	18	21	1	-7	-9
VALENCIA_12_BOLD	18	21	1	-7	-10
VALENCIA_12_ITALIC	18	20	1	-7	-8
VALENCIA_12_BOLD_ITALIC	18	21	1	-8	-8
VALENCIA_14_NORMAL	23	26	2	-9	-13
VALENCIA_14_BOLD	23	29	2	-10	-15
VALENCIA_14_ITALIC	23	26	2	-9	-11
VALENCIA_14_BOLD_ITALIC	23	27	2	-6	-10
VALENCIA_18_NORMAL	29	29	2	-7	-13
VALENCIA_18_BOLD	29	29	2	-4	-12
VALENCIA_18_ITALIC	29	29	2	-8	-11
VALENCIA_18_BOLD_ITALIC	29	29	2	-3	-8
VALENCIA_21_NORMAL	35	33	3	-6	-14
VALENCIA_21_BOLD	35	45	3	-14	-23
VALENCIA_21_ITALIC	35	39	3	-12	-17
VALENCIA_21_BOLD_ITALIC	35	44	3	-13	-20
VALENCIA_24_NORMAL	40	40	3	-8	-17
VALENCIA_24_BOLD	38	47	3	-9	-22
VALENCIA_36_NORMAL	60	55	5	-8	-21
VALENCIA_36_BOLD	57	65	5	-10	-27
VALENCIA_48_NORMAL	79	73	6	-10	-28
VALENCIA_48_BOLD	77	92	6	-19	-33

### Designing Graphic Texts

As you know it in writing programs for the PC you can design the text by its spacing, alignment, wrapping mode. Finally you can integrate lay text over a prepared background by using different overlay modes.

You can determine spacing in different ways. You are able to determine the number of white pixels between the characters. The number of additional pixels on the left and the right of a blank can be given, too. In case of using constant character spacing you can choose between left, right or center alignment of the characters as you can see it in the following figure.

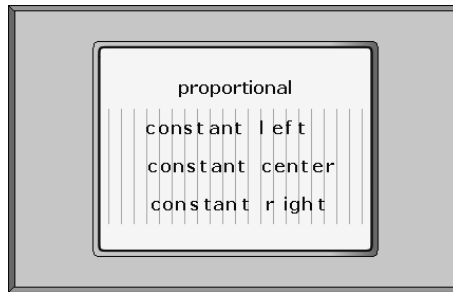


figure 96: Spacing of fonts

For the horizontal alignment of the text lines you can choose between left, right center and center justification. The vertical alignment can be top, center and bottom.

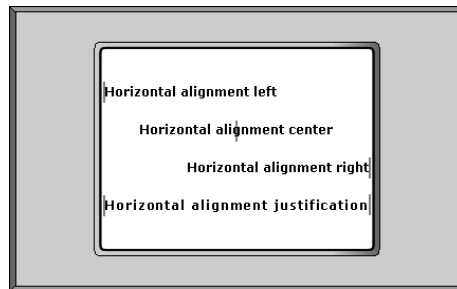


figure 97: Horizontal alignment

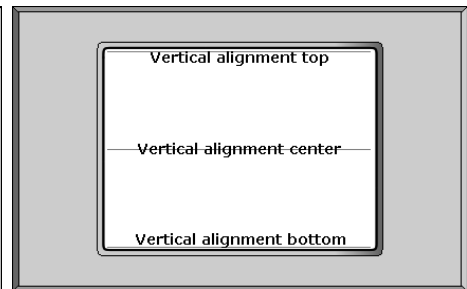


figure 98: Vertical alignment

You can wrap the text lines in different ways. Char wrapping wraps the text after the last char that completely fits into the line. Word wrapping wraps the text after the last word that completely fits into the line. Wrapmode off means no matter how long the text is only one line will be shown.

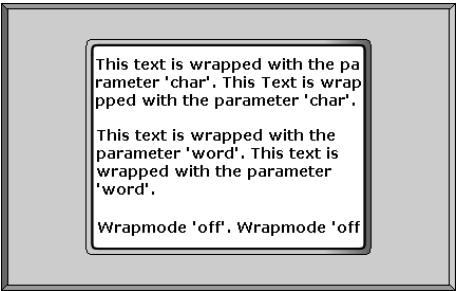


figure 99: Text wrapping

The different modes you can lay the text graphic over the background you can see in the following table:

Overlay mode	Description
IMM	Copy pixels immanently without any modification
AND	„Black" pixels are added.
OR	„White" pixels are added
XOR	„1" pixels invert colour value in destination area

## Codes for Normal and Special Chars

The Tiger-BASIC™ language does not accept all special chars in the programm code. The solution is using the Tiger-BASIC™ codes for these characters. You can insert the Tiger-BASIC™ code in each string position between the other characters.

For writing "The price is 29€." use the following code

```
Text$ = "The price is 29<80h>."
```

This table is for the fonts ATLANTA, TOKIO, ISTANBUL, VALENCIA and STOCKHOLM.

char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code
BLANK	0020	<20h>	3	0033	<33h>	G	0047	<47h>
!	0021	<21h>	4	0034	<34h>	H	0048	<48h>
"	0022	<22h>	5	0035	<35h>	I	0049	<49h>
#	0023	<23h>	6	0036	<36h>	J	004a	<4Ah>
\$	0024	<24h>	7	0037	<37h>	K	004b	<4Bh>
%	0025	<25h>	8	0038	<38h>	L	004c	<4Ch>
&	0026	<26h>	9	0039	<39h>	M	004d	<4Dh>
'	0027	<27h>	:	003a	<3Ah>	N	004e	<4Eh>
(	0028	<28h>	;	003b	<3Bh>	O	004f	<4Fh>
)	0029	<29h>	<	003c	<3Ch>	P	0050	<50h>
*	002a	<2Ah>	=	003d	<3Dh>	Q	0051	<51h>
+	002b	<2Bh>	>	003e	<3Eh>	R	0052	<52h>
,	002c	<2Ch>	?	003f	<3Fh>	S	0053	<53h>
-	002d	<2Dh>	@	0040	<40h>	T	0054	<54h>
.	002e	<2Eh>	A	0041	<41h>	U	0055	<55h>
/	002f	<2Fh>	B	0042	<42h>	V	0056	<56h>
0	0030	<30h>	C	0043	<43h>	W	0057	<57h>
1	0031	<31h>	D	0044	<44h>	X	0058	<58h>
2	0032	<32h>	E	0045	<45h>	Y	0059	<59h>
			F	0046	<46h>	Z	005a	<5Ah>

char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code
[	005b	<5Bh>	z	007a	<7Ah>	™	2122	<99h>
\	005c	<5Ch>	{	007b	<7Bh>	š	0161	<9Ah>
]	005d	<5Dh>		007c	<7Ch>	”	02dd	<9Bh>
^	005e	<5Eh>	}	007d	<7Dh>	œ	0153	<9Ch>
_	005f	<5Fh>	~	007e	<7Eh>	free	free	<9Dh>
`	0060	<60h>	free	free	<7Fh>	ž	017e	<9Eh>
a	0061	<61h>	€	20ac	<80h>	free	free	<9Fh>
b	0062	<62h>	free	free	<81h>	free	free	<A0h>
c	0063	<63h>	free	free	<82h>	i	00a1	<A1h>
d	0064	<64h>	free	free	<83h>	¢	00a2	<A2h>
e	0065	<65h>	"	0022	<84h>	£	00a3	<A3h>
f	0066	<66h>	free	free	<85h>	free	free	<A4h>
g	0067	<67h>	free	free	<86h>	¥	00a5	<A5h>
h	0068	<68h>	free	free	<87h>	free	free	<A6h>
i	0069	<69h>	^	02c6	<88h>	§	00a7	<A7h>
j	006a	<6Ah>	‰	2030	<89h>	”	00a8	<A8h>
k	006b	<6Bh>	Š	0160	<8Ah>	©	00a9	<A9h>
l	006c	<6Ch>	free	free	<8Bh>	free	free	<AAh>
m	006d	<6Dh>	Œ	0152	<8Ch>	free	free	<ABh>
n	006e	<6Eh>	free	free	<8Dh>	free	free	<ACh>
o	006f	<6Fh>	Ž	017d	<8Eh>	ÿ	0178	<ADh>
p	0070	<70h>	free	free	<8Fh>	®	00ae	<AEh>
q	0071	<71h>	free	free	<90h>	free	free	<AFh>
r	0072	<72h>	free	free	<91h>	°	00b0	<B0h>
s	0073	<73h>	free	free	<92h>	±	00b1	<B1h>
t	0074	<74h>	"	0022	<93h>	free	free	<B2h>
u	0075	<75h>	Ů	0150	<94h>	free	free	<B3h>
v	0076	<76h>	Ů	0170	<95h>	’	00b4	<B4h>
w	0077	<77h>	ő	0151	<96h>	μ	00b5	<B5h>
x	0078	<78h>	ú	0171	<97h>	·	02d9	<B6h>
y	0079	<79h>	~	02dc	<98h>	˘	02d8	<B7h>

char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code	char	Uni code	Tiger BASIC code
Š	00b8	<B8h>	Ñ	00d1	<D1h>	ê	00ea	<EAh>
Ĝ	011e	<B9h>	Ò	00d2	<D2h>	ë	00eb	<EBh>
ğ	011f	<BAh>	Ó	00d3	<D3h>	ì	00ec	<ECh>
Ş	015e	<BBh>	Ô	00d4	<D4h>	í	00ed	<EDh>
ş	015f	<BCh>	Õ	00d5	<D5h>	î	00ee	<EEh>
İ	0130	<BDh>	Ö	00d6	<D6h>	ï	00ef	<EFh>
ı	0131	<BEh>	×	00d7	<D7h>	ð	00f0	<F0h>
ı	00bf	<BFh>	Ø	00d8	<D8h>	ñ	00f1	<F1h>
À	00c0	<C0h>	Ù	00d9	<D9h>	ò	00f2	<F2h>
Á	00c1	<C1h>	Ú	00da	<DAh>	ó	00f3	<F3h>
Â	00c2	<C2h>	Û	00db	<DBh>	ô	00f4	<F4h>
Ã	00c3	<C3h>	Ü	00dc	<DCh>	õ	00f5	<F5h>
Ä	00c4	<C4h>	Ý	00dd	<DDh>	ö	00f6	<F6h>
Å	00c5	<C5h>	þ	00de	<DEh>	÷	00f7	<F7h>
Æ	00c6	<C6h>	ß	00df	<DFh>	ø	00f8	<F8h>
Ç	00c7	<C7h>	à	00e0	<E0h>	ù	00f9	<F9h>
È	00c8	<C8h>	á	00e1	<E1h>	ú	00fa	<FAh>
É	00c9	<C9h>	â	00e2	<E2h>	û	00fb	<FBh>
Ê	00ca	<CAh>	ã	00e3	<E3h>	ü	00fc	<FCh>
Ë	00cb	<CBh>	ä	00e4	<E4h>	ý	00fd	<FDh>
Ì	00cc	<CCh>	å	00e5	<E5h>	þ	00fe	<FEh>
Í	00cd	<CDh>	æ	00e6	<E6h>	ÿ	00ff	<FFh>
Î	00ce	<CEh>	ç	00e7	<E7h>			
Ï	00cf	<CFh>	è	00e8	<E8h>			
Ð	00d0	<D0h>	é	00e9	<E9h>			

Codes for Control Chars

As you can see in the table above not all Tiger-BASIC™ codes are used for normal and special chars. The bytes from <00h> to <1Fh> are reserved for the use as control bytes.

All these control codes can be used by putting them right into the text. In the following example the text will change the font from the given Font ID to Font ID '1' after the word 'brown'.

```
Text$ = "The quick brown <1Ch><01h> fox jumps over the lazy dog."
```

code	used for
<09h>	Tab . next tab position is multiple of 64 pixels
<0Dh>	Carriage Return + Line Feed
<1Ch>	Font Select – next byte is a new identifier of a font, not a regular char. Font must be created before.

The following sample program will show this graphic on the display.



figure 100: Change font in one text

```

'-----
' FONT_ChangeFontInText.TIG
'-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
'                                  ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

#define FONT_ID_0      0
#define FONT_ID_1      1
#define FONT_ID_2      2
#define FONT_ID_3      3
#define FONT_ID_4      4

string sgLcd$(LCD_SIZE) ' content for LCD output

task main
  byte blReturn ' return value for tgl subroutines

  #include TGL_DEVICE_DRIVERS_TP1000.INC
  '*****
  ' Initialization
  '*****
  call vTglInit()
  sgLcd$ = fill$( "00%", LCD_SIZE )

  '*****
  ' create elements
  '*****
  '
  '          Id,          name,          size,type,          return code
  call bTglCreateFont( FONT_ID_0, "Stockholm", 18, "normal", blReturn )
  call bTglCreateFont( FONT_ID_1, "Valencia",  10, "normal", blReturn )
  call bTglCreateFont( FONT_ID_2, "Tokio",     14, "bold",   blReturn )
  call bTglCreateFont( FONT_ID_3, "Atlanta",   12, "italic", blReturn )
  call bTglCreateFont( FONT_ID_4, "Istanbul",  21, "normal", blReturn )

  call bTglSetFontParams(&
  FONT_ID_0, "center","center","prop",0, &' id, alignments H,V,type,blank
  SPACING_CHAR_DEFAULT, 0, "imm", & ' spacing char,V, overlay mode,
  "word", blReturn )                ' wrapmode, return

  '*****
  ' show elements
  '*****
  '
  '          dst,          txt
  call sTglBuildTextGraphic( sgLcd$, &
  "The <1Ch><01h>quick brown <1Ch><02h>fox jumps <1Ch><03h>over &
  <1Ch><04h>the lazy <1Ch><00h>dog.", &
  LCD_WIDTH,LCD_HEIGHT, 0,0,      320,240,FALSE,FONT_ID_0, blReturn )
  '          wDst,hDst,      xWnd,yWnd,wWnd,hWnd, inv,fontId,      return code
  '*****
  ' show text on display
  '*****
  call vTglShowUserGraphic( sgLcd$ )
end

```



### Graphic Fonts Solo

You can program with the graphic fonts in different ways. For standard needs we suggest just creating some labels, place them in windows and show them on LCD as it is described in chapter *Labels*.

There are reasons to work with the graphic fonts alone, without using any elements. One reason could be saving memory but the need of dynamical built text graphics. Another reason could be some special needs, e.g. building a text graphic of a bigger size than the LCD which could be shown by scrolling.

If you program with the graphic fonts without labels you have to follow these steps:

1. Choose the fonts for your project in the configuration file
2. Include the needed graphic fonts
3. Set the directions and states of the BASIC-Tiger ports
4. Install the device driver for the LCD
5. Initialize the Tiger Graphic Library
6. Assemble the attributes for the text layout by creating a font
7. Declare and initialize a string variable for the graphic text
8. Build a graphic text
9. Place the graphic text in an output string for the LCD
10. Display the graphic text on the LCD

**!** NEVER build a text graphic by pass an uninitialized string. Before filling a string with a text graphic ensure that the string has the right length for the given graphic area.

Please mind creating a graphic font before building and placing a graphic text. Creating a font means defining an identifier for the font and its attributes.

### Including Graphic Fonts

You have to make the program known that it should work with the graphic fonts. The graphic fonts are part of the Tiger Graphic Library. So you just have to include the Tiger Graphic Library. Do this right in the beginning of your program.

```
#include TigerGraphicLibrary.INC
```

If you will program without any element or window you can copy the configuration file

*C:\Programme\Wilke Technology\Tiger Basic 5.3\TigerGraphicLibrary\TigerGraphicLibraryConf.INC*

into your project directory and activate the master defines in the like this for saving memory.

```
#define TGL_ELEMENTS           ' activates elements and windows
#define TGL_TOUCHPANEL        ' activate all touch panel functions
#define TGL_GRAPHIC_FONTS     ' activate all graphic font functions
#define TGL_KEYBOARDS         ' activate keyboard applications
#define TGL_RTC_APPLICATIONS  ' activate RTC applications
```

For details see chapter *Configuration* sections *Master Defines* and *Memory for Graphic Fonts*.

### Hardware Configuration for the LCD

For the use of the LCD please set the BASIC Tiger™ ports. The LCD must get a reset and its backlight has to be switched on. The reset of the LCD should be done before installing the device driver for the LCD.

```
dir_pin 8, 5, 0           ' L85: output for reset of LCD
out 8, 00100000b, 0       ' shut down the LCD
out 8, 00100000b, 255     ' start the LCD
wait_duration 100         ' starting time for the LCD

dir_pin 8, 2, 0           ' L82: output for backlight of LCD
out 8, 00000100b, 0       ' switch backlight of LCD on
```

Now please install the device driver for the LCD. Make sure using the device number LCD as they are used in the Tiger Graphic Library. For details see the device driver manual. We suggest installing all the device drivers you want to use in your project in the task main.

```
install_device #LCD, "LCD-S1D13700.TD2",0 ,0, 0EEH, 1, 250, 2, 0
```

For easy installing of the LCD in one code line only include the following file. Certainly you can modify this file by your own needs.

```
#include TGL_DEVICE_DRIVERS_TP1000.INC
```

Before the calling of any subroutine of the Tiger Graphic Library initialize the graphic fonts.

```
call vTglInit()
```

For putting several strings quickly on LCD, please mind the busy time of the LCD. For details see the device driver manual for the LCD. You need not care about this by using the output subroutines *vPutStringToLcd* and *vPutStringToLcdParams*

sample program:

```
-----
' TGL_GRAPHIC_FONTS_solo
-----
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion for "blue"  LCD
#include TigerGraphicLibrary.INC

string sgLcd$(LCD_SIZE)

task main
  byte blReturn                    ' return value for TGL subroutines
  string slText$

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  *****
  ' INITIALIZATION
  *****
  call vTglInit()
  sgLcd$ = fill$( "00%", LCD_SIZE)

  *****
  ' FONTS
  *****
  '                               Id, name, size, type, return
  call bTglCreateFont( 0, "Valencia", 18, "bold", blReturn )

  *****
  ' BUILD TEXT GRAPHIC
  *****
  slText$ = "Hello graphic fonts!"
  call sTglBuildTextGraphic( &
    sgLcd$, slText$, &          ' graphical destination area, text
    LCD_WIDTH,LCD_HEIGHT, &    ' format width, height of destination
    80,60, &                    ' x,y coordinates
    160,120, &                  ' width, height of text graphic
    FALSE,0, &                  ' inverting flag, frame thickness
    blReturn )                  ' return value (0=Ok, >0=Error)

  *****
  ' SHOW TEXT GRAPHIC
  *****
  put #LCD, #1, sgLcd$          ' show sgLcd$ on LCD
end
```

## bTglCreateFont

call `bTglCreateFont( FontId, FontName$, FontSize, FontType$, Result )`

Function: Creates a new font with default parameters.

### Parameters:

	B	W	L	S	F	
FontId	●	-	-	-	-	unique identifier of this font
FontName\$	-	-	-	●	-	name of this fonts family
FontSize	●	-	-	-	-	size of chosen font
FontType\$	-	-	-	●	-	type of chosen font

### Return Values:

Result	●	-	-	-	-	error code, for details see table of error codes
	0					ok
	>0					error

Available fonts are:

Name	Type	Size
Amsterdam	bold	8,11,16,21
Atlanta	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Helsinki	normal	7,8,9,10,11,12,14,18,22,26
	bold	10,12,14,18,22,26,28,32,52
	bold	56,60
Istanbul	normal	8,10,11,12,14,18,21
Stockholm	normal	8,10,11,12,14,18,21
Tokio	normal, bold, italic, bold italic	8,10,11,12,14,18,21
Valencia	normal, bold, italic, bold italic	8,10,11,12,14,18,21
	normal, bold	24,36,48

default Parameters	
horizontal alignment	left
vertical alignment	top
spacing	proportional with font specific default values

default Parameters	
overlay mode	immanent copy
line wrapping after	word

bTglCreateFontParams

```
call bTglCreateFontParams( FontId, FontName$, FontSize, FontType$, &
AlignHorizontal$, AlignVertical$, SpacingType$, &
SpacingBlank, SpacingChar, & Spacing Vertical, &
OverlayMode$, WrapMode$, Result )
```

Function: Stores all font attributes in an internal global parameter string handled by FontId.

Parameters:

	B	W	L	S	F	
FontId	●	-	-	-	-	unique identifier of this font
FontName\$	-	-	-	●	-	name of this fonts family
FontSize	●	-	-	-	-	size of chosen font
FontType\$	-	-	-	●	-	type of chosen font
AlignHorizontal\$	-	-	-	●	-	horizontal alignment of text
AlignVertical\$	-	-	-	●	-	vertical alignment of text
SpacingType\$	-	-	-	●	-	spacing type of text
SpacingBlank	-	-	●	-	-	additional spacing pixels for blank
SpacingChar	-	-	●	-	-	spacing pixels after every character
SpacingVertical	-	-	●	-	-	spacing pixel lines between two lines
OverlayMode\$	-	-	-	●	-	graphic copy mode – text graphic into screen
WrapMode\$	-	-	-	●	-	mode of text wrapping
Result	●	-	-	-	-	<b>Return Values:</b> error code, for details see table of error codes 0 ok >0 error

! Mind having activated the codelines for the fonts in the configuration file *TigerGraphicLibraryConf.INC*. We suggest activating only those fonts you really need. Including all available fonts would take too much flash memory. For details see chapter *Configuration*.

Parameters	Options
FontName\$	Helsinki, Amsterdam, Tokio, Istanbul, Atlanta, Valencia,

Parameters	Options
	Stockholm
FontType\$	normal, bold, italic, bold italic
AlignHorizontal\$	left, center, right, just
AlignVertical\$	top, center, bottom
SpacingType\$	prop, const left, const center, const right
OverlayMode\$	imm, or, and, xor
WrapMode\$	char, word, off

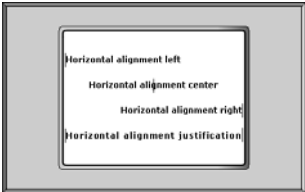


figure 101: Horizontal alignment

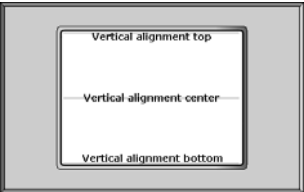


figure 102: Vertical alignment

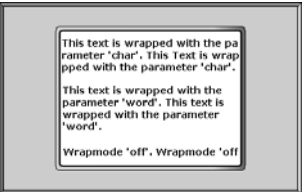


figure 90: Text wrapping

Overlay mode	Description
IMM	Copy pixels immanently without any modification (default)
AND	„Black" pixels are added.
OR	„White" pixels are added
XOR	„1" pixels invert colour value in destination area

The distance between the characters, which is determined by the parameter *spacing char* is a font specific attribute. Normally you can pass the define *SPACING\_CHAR\_DEFAULT* for this parameter. If you like to determine the number of pixels by your own, you can pass your own number. Negative numbers are allowed, too. This would let the characters overlap.

For a view of a variable value you should choose a constant spacing type. Otherwise if you choose proportional spacing type a flatterring will occur with each changing of the value because of the unequal widths of the digits. For an optimized spacing for constant digit distances, pass the define *SPACING\_CHAR\_DEFAULT\_DIGIT*. For a code example see *TGL\_SLIDER\_X\_show\_value.TIG*.



bTglSetFontParams

```
call bTglSetFontParams( FontId, AlignHorizontal$, AlignVertical$, SpacingType$, &
                        SpacingBlank, SpacingChar, Spacing Vertical, &
                        OverlayMode$, WrapMode$, Result )
```

Function: Set new parameters for a font.

Parameters:

	B	W	L	S	F	
FontId	●	-	-	-	-	unique identifier of this font
AlignHorizontal\$	-	-	-	●	-	horizontal alignment of text
AlignVertical\$	-	-	-	●	-	vertical alignment of text
SpacingType\$	-	-	-	●	-	spacing type of text
SpacingBlank	-	-	●	-	-	additional spacing pixels for blank
SpacingChar	-	-	●	-	-	spacing pixels after every character
SpacingVertical	-	-	●	-	-	spacing pixel lines between two lines
OverlayMode\$	-	-	-	●	-	graphic copy mode – text graphic into screen
WrapMode\$	-	-	-	●	-	mode of text wrapping

Return Values:

Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

Parameters	Options
FontName\$	Helsinki, Amsterdam, Tokio, Istanbul, Atlanta, Valencia, Stockholm
FontType\$	normal, bold, italic, bold italic
AlignHorizontal\$	left, center, right, just
AlignVertical\$	top, center, bottom
SpacingType\$	prop, const left, const center, const right
OverlayMode\$	imm, or, and, xor
WrapMode\$	char, word, off

The distance between the characters, which is determined by the parameter *spacing char* is a font specific attribute. Normally you can pass the define

*SPACING\_CHAR\_DEFAULT* for this parameter. If you like to determine the number of pixels by your own, you can pass your own number. Negative numbers are allowed, too. This would let the characters overlap.

For a view of a variable value you should choose a constant spacing type. Otherwise if you choose proportional spacing type a flatterer will occur with each changing of the value because of the unequal widths of the digits. For an optimized spacing for constant digit distances, pass the define *SPACING\_CHAR\_DEFAULT\_DIGIT*. For a code example see *TGL\_SLIDER\_X\_show\_value.TIG*.

## bTglGetFontParams

```
call bTglGetFontParams( FontId, FontName$, FontSize, FontType$, &
                        AlignHorizontal$, AlignVertical$, SpacingType$, &
                        SpacingBlank, SpacingChar, & Spacing Vertical, &
                        OverlayMode$, WrapMode$, Return )
```

Function: Returns actual parameters of a font.

### Parameters:

	B	W	L	S	F	
FontId	●	-	-	-	-	unique identifier of this font
FontName\$	-	-	-	●	-	name of this fonts family
FontSize	●	-	-	-	-	size of chosen font
FontType\$	-	-	-	●	-	type of chosen font
AlignHorizontal\$	-	-	-	●	-	horizontal alignment of text
AlignVertical\$	-	-	-	●	-	vertical alignment of text
SpacingType\$	-	-	-	●	-	spacing type of text
SpacingBlank	-	-	●	-	-	additional spacing pixels for blank
SpacingChar	-	-	●	-	-	spacing pixels after every character
SpacingVertical	-	-	●	-	-	spacing pixel lines between two lines
OverlayMode\$	-	-	-	●	-	graphic copy mode – text graphic into screen
WrapMode\$	-	-	-	●	-	mode of text wrapping
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

Parameters	Options
FontName\$	Helsinki, Amsterdam, Tokio, Istanbul, Atlanta, Valencia, Stockholm
FontType\$	normal, bold, italic, bold italic
AlignHorizontal\$	left, center, right, just
AlignVertical\$	top, center, bottom
SpacingType\$	prop, const left, const center, const right
OverlayMode\$	imm, or, and, xor
WrapMode\$	char, word, off

sTglBuildTextGraphic

```
call sTglBuildTextGraphic( GraphicOut$, Text$, DestWidth, DestHeight, &
                           WndPosX, WndPosY, WndWidth, WndHeight, InvertFlag, &
                           FontId, Result )
```

Function: Builds a text graphic and places it in a graphic area. Optionally the text graphic can be inverted.

Parameters:

	B	W	L	S	F	
Text\$	-	-	-	●	-	this text will be put into graphic
DestWidth	-	-	●	-	-	width of destination area (multiple of 8)
DestHeight	-	-	●	-	-	height of destination area
WndPosX	-	-	●	-	-	X Position of top left corner of graphic box
WndPosY	-	-	●	-	-	Y Position of top left corner of graphic box
WndWidth	-	-	●	-	-	width of graphic box
WndHeight	-	-	●	-	-	height of graphic box
InvertFlag	●	-	-	-	-	TRUE = show graphic box inverted FALSE = show graphic box normal
FontId	●	-	-	-	-	unique identifier of this font
GraphicOut\$	-	-	-	●	-	graphic area with formatted graphic text
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

Return Values:

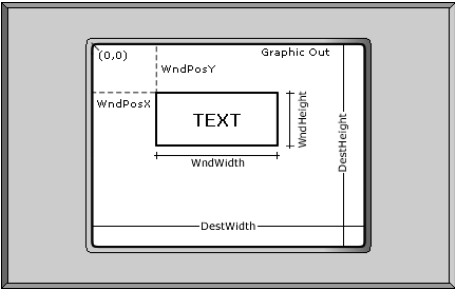


figure 103: Placing a text graphic on LCD

Ensure that the string for the text graphic has the correct maximal size. You want to build a text graphic with the width of 32 and the height of 100 pixels. The string for the graphic must have the length of at least  $32 \cdot 100 / 8 = 400$  bytes.

The string must have been initialized before calling the subroutine. You can do this by the functions `set_len$` or `fill$`. The string need not to be empty. The content will be used as background for the string, if you build the string with an overlay mode OR, AND or XOR.

! Mind that this subroutine works only with a limited number of characters `TGL_TXT_GRAPHIC_TXT_LEN` as it is determined in the file *TigerGraphicLibraryConf.INC*. The default value is 512 characters. You can increase this value if you have enough RAM. Otherwise please call the subroutine *ITglCalcTextToWindow*.

ITglCalcTextToWindow

```
call ITglCalcTextToWindow( NumCharsWnd, NumLinesWnd, MaxNumLines, &
                           Text$, FontId, WndWidth, WndHeight, Result )
```

Function: Calculates how many characters of a given text fit as graphic characters in a box of given width and height using the parameters for the text design. Additionally it returns the number of lines the characters of the text will fill in the box and the maximum of lines fitting in the box.

Parameters:

	B	W	L	S	F	
Text\$	-	-	-	●	-	this text is calculated with
FontId	●	-	-	-	-	unique identifier of this font
WndWidth	-	-	●	-	-	width of graphic box
WndHeight	-	-	●	-	-	height of graphic box
<b>Return Values:</b>						
NumCharsWnd	-	-	●	-	-	number of characters out of Text\$ which fit in graphic box formatted by FontId
NumLinesWnd	-	●	-	-	-	number of lines out of Text\$ which fit in graphic box formatted by FontId
MaxNumLines	-	●	-	-	-	maximum number of lines that are possible in graphic box formatted by FontId
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

# ITglGetLineHeight

call ITglGetLineHeight( LineHeight, FontId )

Function: Returns height of one text line of chosen font.

Parameters:

	B	W	L	S	F	
FontId	●	-	-	-	-	unique identifier of this font
LineHeight	-	-	●	-	-	<b>Return Value:</b> height of one line of text in chosen font

The height of a text line is defined by the distance of the highest and lowest pixel position of all character bitmaps.

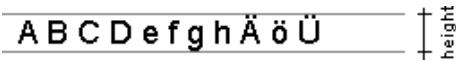


figure 104: Line height

For details see table in section *Specific Parameters of Graphic Fonts*

ITglCalcTextGraphicWidth

```
call ITglCalcTextGraphicWidth( NumPixels, Text$, FontId, Result )
```

Function:        Calculates the pixel width of the text graphic without any line wrapping.

Parameters:

	B	W	L	S	F	
Text\$	-	-	-	●	-	given text
FontId	●	-	-	-	-	identifier of font

NumPixels	-	-	●	-	-	text width in pixels
Result	●	-	-	-	-	error code, for details see table of error codes

0

ok

>0

error



# User Graphic

For easy displaying your own graphic the Tiger Graphic Library provides you some special subroutines for LCD output. To be sure not to be disturbed by the outputs of the Tiger Graphic Library please use these subroutines.

Available special subroutines for LCD output in the Tiger Graphic Library:

- *vTglShowUserGraphic*
- *vTglShowUserGraphicParams*
- *vTglHideUserGraphic*
- *sTglGetWindowGraphic*
- *vTglPutWindowGraphic*
- *vTglClearWindowGraphic*
- *vPutStringToLcd*
- *vPutStringToLcdParams*

! For LCD outputs mind using the one of the two LCD layers the Tiger Graphic Library does not use. The LCD device driver will "or" both LCD layers. In the default configuration the Tiger Graphic Library uses the layer 1. You can determine this layer by the define *SHOW\_WINDOW\_LAYER* in the file *TigerGraphicLibraryDefs.INC*. If you use the special subroutines for LCD output you need not care about this!

## User Graphic

A nice example for displaying self made graphics and using the Tiger Graphic Library in one application is given by the program TGL\_USER\_GRAPHIC\_analog\_clock.TIG.

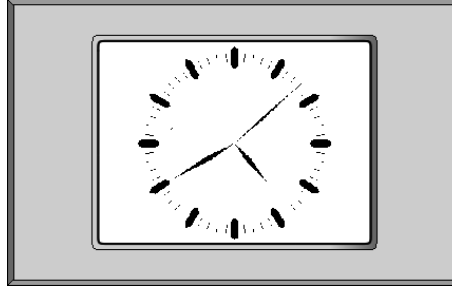


figure 105: Self made graphic analog clock

sample program:

```
#define LCD_INVERSION_MODE      0      ' 0 = normal    for "white" LCD
                                     ' 1 = inversion for "blue" LCD

#include TigerGraphicLibrary.INC

' parameter of draw functions
#define X_REF_CLOCK_CENTER      160
#define Y_REF_CLOCK_CENTER      120
#define MODE_INV_LINE           3
#define MODE_BLACK_LINE         0
#define PEN_CLOCK               0

' free LCD layer for user when using the Tiger Graphic Library
#define TGL_LCD_LAYER_USER      2

' global strings for user graphic
string sgLcd$(LCD_SIZE), sgLcdBackground$(LCD_SIZE)

task main
    *****
    ' declaration of variables
    *****
    ' touch panel
    word wIbuFill                ' filling of touch panel input buffer
    byte blKeycode

    ' TGL general
    byte blReturn                ' return value of tgl subroutines
    word wElementId              ' current identifier for creation of elements
    word wWindowId               ' current identifier for creation of windows
    byte blFontId                ' current identifier for creation of fonts
    byte ever

    ' RTC application
```

```

string slInput$(100h) ' buffer for user input
byte  blFONT_ID_rtc   ' identifier for font    of rtc application
word  wlWND_ID_rtc    ' identifier for window  of rtc application
word  wlELEM_ID_rtc    ' identifier for element of rtc application

'' user graphic clock
word  wlWND_ID_clock  ' identifier for window of user graphic clock
string slDate$(8), slDateOld$(8)
long  llRot           ' rotation of line to be drawn

*****
' device drivers
*****
#include TGL_DEVICE_DRIVERS_TP1000.INC
install_device #RTC, "RTCl.TD2"

*****
' INITIALIZATION
*****
'' variables
call vTglInit()
wElementId = 0
wWindowId  = 0
blFontId   = 0
set_len$( slInput$, 0 )
set_len$( slDate$, 0 )
set_len$( slDateOld$, 0 )

'' RTC application
blFONT_ID_rtc = blFontId      ' save identifier of font    for rtc
call bTglCreateFont( &
blFontId, &                  ' identifier of font
"Valencia", 18, "bold", &    ' name, size, type of font
blReturn )                   ' return code (0: OK exit >0: error exit)
wlWND_ID_rtc = wWindowId
call bTglInitRtc( TGL_RTC_STYLE_1, blFONT_ID_rtc, wElementId, &
wWindowId, wlELEM_ID_rtc, blReturn )

'' draw the clocks face / dial
sgLcdBackground$ = fill$( "00%", LCD_SIZE ) ' clear LCD string
for llRot = 0 to 33000 step 3000 ' angles for dial hour (12lines,30deg)
draw_line( sgLcdBackground$,LCD_WIDTH,LCD_HEIGHT, &
X_REF_CLOCK_CENTER,Y_REF_CLOCK_CENTER, 0,-50, 2,-48, &
2400,2400, llRot,MODE_INV_LINE,PEN_CLOCK )
draw_next_line( 2,-40, PEN_CLOCK )
draw_next_line( 0,-38, PEN_CLOCK )
draw_next_line( -2,-40, PEN_CLOCK )
draw_next_line( -2,-48, PEN_CLOCK )
close_line( PEN_CLOCK )
next
fill_area( sgLcdBackground$ ) ' fill screen area with 1-bits
for llRot = 0 to 35400 step 600 ' angles for dial second (60lines,6deg)
draw_line( sgLcdBackground$,LCD_WIDTH,LCD_HEIGHT, &
X_REF_CLOCK_CENTER,Y_REF_CLOCK_CENTER, 0,-46, 0,-42, &
2400,2400, llRot,MODE_BLACK_LINE,PEN_CLOCK )
next

'' define whole clock window as a button
wlWND_ID_clock = wWindowId

```

```

call bTglCreateButtonWnd( &
LCD_WIDTH, LCD_HEIGHT, &      ' width, height of element
0, 0, &                        ' address, format width of bitmap
TGL_KEY_ATTR_DEFAULT, &      ' key attributes auto repeat, beep, type
wElementId, wWindowId, &      ' identifier of element, window
0, 0, &                        ' x, y coordinate on LCD
0h, &                          ' keycode
bIReturn )                     ' return code (0: OK exit >0: error exit)

'*****
' start program
'*****
call bTglShowWindow( wWIND_ID_clock, bIReturn )
for ever = 0 to 0 step 0

    ' set new time
    get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill
    if 0 < wIbuFill then
        call bTglSetRtc( wWIND_ID_rtc, wELEM_ID_rtc, bIReturn )
        call bTglShowWindow( wWIND_ID_clock, bIReturn )
    endif

    ' update time on LCD secondly
    get #rtc, #3, 0, sIDate$ ' h,m,s,d,M,yy,D
    if sIDateOld$ <> sIDate$ then
        sIDateOld$ = sIDate$
        call sUpdateClock( sIDate$ )
    endif

    release_task
next ' end of endless loop
end

'-----
' update time on LCD
'-----
sub sUpdateClock( string spDate$ )
    byte bIReturn          ' return value of tgl subroutines
    string sIDate$(8)
    byte bISec, bIMin, bIHour      ' time
    long llRotHour, llRotMin, llRotSec ' angles

    ' extract time
    bISec = nfoms( spDate$, 2, TGL_BYTE )
    bIMin = nfoms( spDate$, 1, TGL_BYTE )
    bIHour = nfoms( spDate$, 0, TGL_BYTE )

    ' calculate angles for sec, min and hour
    llRotSec = bISec *600 + 18000      ' bISec*(36000/60) + 18000
    llRotMin = bIMin *600 + 18000      ' bIMin*(36000/60) + 18000
    llRotHour = bIHour*3000 + 18000 + bIMin*50 ' bIHour*(36000/12)+ 18000
    '                                     ' + (bIMin*(36000/720))

    sgLcd$ = fill$( "00%", LCD_SIZE ) ' clear LCD string
    '                                     scale-X/Y
    call sDrawWatchHand( llRotHour, 4000,2600, sgLcd$ )
    call sDrawWatchHand( llRotMin, 3800,4000, sgLcd$ )
    call sDrawWatchHand( llRotSec, 1900,4600, sgLcd$ )

```

```
fill_area( sgLcd$ ) ' fill screen area with 1-bits

' update clock on LCD
sgLcd$ = or2$( sgLcd$, sgLcdBackground$ )
call vTglShowUserGraphic( sgLcd$ )
end

'-----
' draw a watch hand
'-----
sub sDrawWatchHand( long lpRot, lpXScale,lpYScale; var string spvDst$ )
draw_line( spvDst$,LCD_WIDTH,LCD_HEIGHT, &
X_REF_CLOCK_CENTER,Y_REF_CLOCK_CENTER, 0,0, 1,8, &
lpXScale,lpYScale, lpRot,MODE_INV_LINE,PEN_CLOCK )
draw_next_line( 0, 24, PEN_CLOCK )
draw_next_line( -1, 8, PEN_CLOCK )
close_line( PEN_CLOCK )
end
```

### vTglShowUserGraphic

```
call vTglShowUserGraphic( Graphic$ )  
call vTglShowUserGraphicF( GraphicAddr )
```

Function: Display a user made graphic on the second LCD layer which the Tiger Graphic Library will not use.

#### Parameters:

	B	W	L	S	F	
Graphic\$	-	-	-	●	-	user graphic
GraphicAddr-	-	-	●	-	-	flash address of user graphic

If you call this subroutine ensure that the graphic has a size of LCD\_SIZE bytes to fill the whole LCD.

For updating just a few rows of the LCD with a graphic of less bytes than the LCD needs, please call the subroutine *vTglShowUserGraphicParams* resp. *vTglShowUserGraphicFParams*.

For using the other LCD layer which is used by the Tiger Graphic Library, please call the subroutine *sTglPutWindowGraphic* resp. *sTglPutWindowGraphicF*.

vTglShowUserGraphicParams

```
call vTglShowUserGraphicParams( Graphic$, OffsLcd, OffsStr, Length )
call vTglShowUserGraphicFParams( GraphicAddr, OffsLcd, OffsStr, Length )
```

Function: Display a user made graphic on selected rows of the second LCD layer which the Tiger Graphic Library will not use.

Parameters:

	B	W	L	S	F	
Graphic\$	-	-	-	●	-	user graphic
GraphicAddr-	-	-	●	-	-	flash address of user graphic
OffsLcd	-	-	-●	-	-	offset on LCD
OffsStr	-	-	-●	-	-	offset in string for user graphic
Length	-	-	-●	-	-	number of bytes to be displayed on LCD
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

If your graphic would fill the whole LCD, please call the subroutine *vTglShowUserGraphic*.

For using the other LCD layer which is used by the Tiger Graphic Library, please call the subroutine *sTglPutWindowGraphic*.

## vTglHideUserGraphic

call vTglHideUserGraphic()

Function: Clear the second LCD layer which the Tiger Graphic Library will not use.

For clearing the other LCD layer which is used by the Tiger Graphic Library, please call the subroutine *vTglClearWindowGraphic*. The subroutine *sTglHideWindow* would deactivate any touch panel functionality, too.



sTglGetWindowGraphic

call sTglGetWindowGraphic( WndGraphic, Result )

Function: This function returns a copy of the internal string for the graphic of the elements of the actual window

Parameters:

	B	W	L	S	F
WndGraphic	-	-	-	●	-
Result	●	-	-	-	-

**Return Values:**  
LCD content of the current window  
error code, for details see table of error codes  
0 ok  
>0 error

For getting the current LCD output of both layers use the following code:

```
long llAddr
string spvCopyScreen$(LCD_SIZE)
get #LCD, #0, #UFCI_GRA_HOME, 0, llAddr ' start address of graphics
get #LCD, #llAddr, 0, spvCopyScreen$    ' read out graphical content
```

vTglPutWindowGraphic

```
call vTglPutWindowGraphic( Graphic$ )
call vTglPutWindowGraphicF(GraphicAddr)
```

Function: This function changes the actual shown window of the Tiger Graphic Library by the given graphic using the LCD layer of the Tiger Graphic Library.

Parameters:

	B	W	L	S	F	
Graphic\$	-	-	-	●	-	Graphic which will be shown on the LCD layer which is used by the Tiger Graphic Library
GraphicAddr-	-	-	●	-	-	flash address of user graphic

Mind that the Tiger Graphic Library will use this layer for its graphical outputs, too. For avoiding any collisions, please call the subroutine *vTglShowUserGraphic* resp. *vTglShowUserGraphicF* which uses the alternative LCD layer.

## vTglClearWindowGraphic

call vTglClearWindowGraphic()

Function: Clear the LCD layer which the Tiger Graphic Library is using.

For clearing the other LCD layer which is not used by the Tiger Graphic Library, please call the subroutine *vTglHideUserGraphic*.

vTglPutStringToLcd

```
sub vTglPutStringToLcd( String$, Layer, DevNo )
```

Function:      Puts a string on the LCD paying attention to the busy time of the LCD.  
Should be called only if no elements or windows are used in the program.

Parameters:

	B	W	L	S	F	
String\$	-	-	-	●	-	source string
Layer	●	-	-	-	-	number of the LCD layer 1 or 2
DevNo	-	-	●	-	-	device number in the program

If you work with elements and windows in your program, please call the subroutines *vTglShowUserGraphic*, *vTglShowUserGraphicParams* or *sTglPutWindowGraphic* for LCD outputs to avoid conflicts with the LCD output controlled by tge TigerGraphic Library.

For showing just a few rows of the LCD with a graphic of less bytes than the LCD needs, please call the subroutine *vTglPutStringToLcdParams*.

vTglPutStringToLcdParams

call vTglPutStringToLcdParams( String\$, Layer, DevNo, OffsLcd, OffsStr, Length )

Function: Puts a string on selected rows of the LCD paying attention to the busy time of the LCD.  
Should be called only if no elements and windows are used in the program.

Parameters:

	B	W	L	S	F	
String\$	-	-	-	●	-	source string
Layer	●	-	-	-	-	number of the LCD layer 1 or 2
DevNo	-	-	●	-	-	device number in the program
OffsLcd	-	-	-●	-	-	offset on LCD
OffsStr	-	-	-●	-	-	offset in string for user graphic
Length	-	-	-●	-	-	number of bytes to be displayed on LCD

If you work with elements and windows in your program, please call the subroutines *vTglShowUserGraphic*, *vTglShowUserGraphicParams* or *sTglPutWindowGraphic* for LCD outputs to avoid conflicts with the LCD output controlled by tge TigerGraphic Library.

For filling the whole LCD, please call the subroutine *vTglPutStringToLcd*.

vTglPutFlashToLcd

call vTglPutFlashToLcd( Addr, Layer, DevNo, OffsLcd, OffsStr, Length )

Function: Puts a string on selected rows of the LCD paying attention to the busy time of the LCD.  
Should be called only if no elements and windows are used in the program.

Parameters:

	B	W	L	S	F	
Addr	-	-	-	●	-	flash address
Layer	●	-	-	-	-	number of the LCD layer 1 or 2
DevNo	-	-	●	-	-	device number in the program
OffsLcd	-	-	-	●	-	offset on LCD
OffsStr	-	-	-	●	-	offset in string for user graphic
Length	-	-	-	●	-	number of bytes to be displayed on LCD

If you work with elements and windows in your program, please call the subroutines *vTglShowUserGraphic*, *vTglShowUserGraphicParams* or *sTglPutWindowGraphic* for LCD outputs to avoid conflicts with the LCD output controlled by tge TigerGraphic Library

For putting strings on LCD, please call the subroutines *vTglPutStringToLcd* resp. *vTglPutStringsToLcdParams*.

# Graphical Functions

In this chapter you will find some goodies for often needed graphics, e.g. drawing a graph of measurands or visualizing values statically by charts or dynamically by gauges.

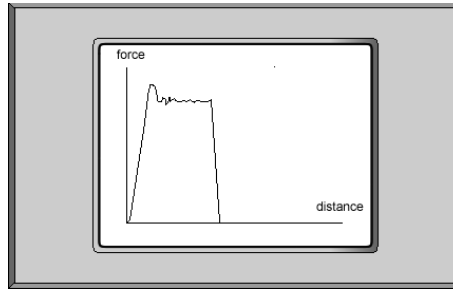


figure 106: Graph

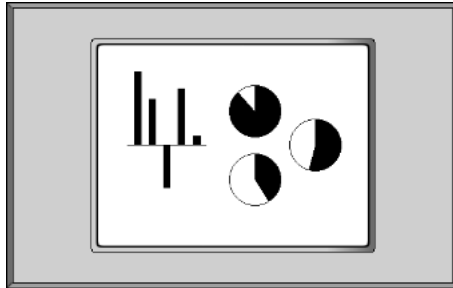


figure 107: Charts

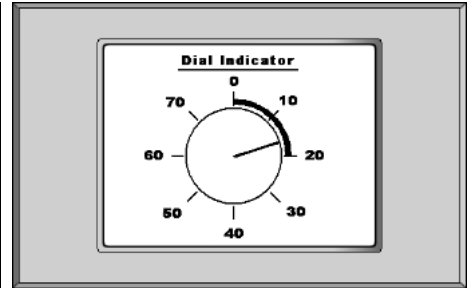


figure 108: Gauges

For the use of graphs, charts and gauges with elements please see the chapters General Subroutines and Gauges.

sTglDrawGraph

call sTglDrawGraph(WFmt,HFmt, Width,Height, X,Y, Data\$, DataWidth, & FirstVal, Num, NumTotal, LimInf,LimSup, Axis, Graph\$, Result )

Function:      Draws a scaled graph in a rectangle area in a destination area

Parameters:

	B	W	L	S	F	
WFmt,HFmt	-	●	-	-	-	format width, height of the destination WFmt must be a multiple of 8
Width,Height	-	●	-	-	-	size of the area for the graph
X,Y	-	●	-	-	-	position of the graph's area in the destination
FirstVal	-	●	-	-	-	number of first value (not byte!) in string to be drawn 0 is number of first value in string
Num	-	●	-	-	-	number of all values (not bytes!) to be drawn 0 take all values in string from the value FirstVal
NumTotal	-	●	-	-	-	1 invalid value, must be at least 2 or 0 number of values in the finished graph can be more than the number of given values is needed for incremental drawing of a graph 0 lengthen graph to whole element width >0 continue graph from FirstVal on expects existing graph of values less FirstVal must be at least FirstVal+Num
LimInf,LimSup	-	-	-	-	●	number of the LCD layer 1 or 2
Axis	●	-	-	-	-	TGL_NO_AXIS TGL_AXIS TGL_AXIS_SCALE_BINARY TGL_AXIS_SCALE_DECIMAL TGL_AXIS_LINES_H_BINARY TGL_AXIS_LINES_H_DECIMAL TGL_AXIS_LINES_HV_BINARY TGL_AXIS_LINES_HV_DECIMAL
Graph\$	-	-	●	-	-	<b>Return Values:</b> graphical data code
Result	●	-	-	-	-	error code, for details see table of error codes 0    ok >0   error



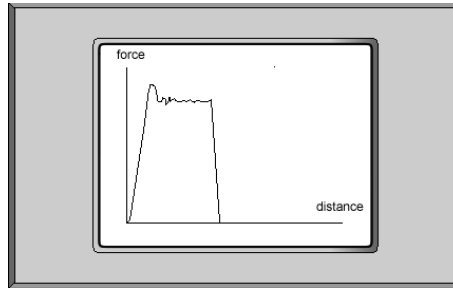


figure 109: Graph

For drawing a graph in an element "graphic", please call the subroutine *bTglShowGraph* described in the chapter *General Subroutines*.

## Touch Panel

For reading out the touch panel buffer the Tiger Graphic Library provides the functions *bTglGetKeycode* and *bTglWaitKeycode*. Further touch panel functions are described in chapter *General Functions*.

For a more direct control of the touch panel it is possible to use the device driver functions of the *TOUCHPANEL.TDD*. For detailed information please see the touch panel device driver manual *Touchpanel\_xx.pdf*.

For the device driver number in the Tiger Graphic Library use the define *TP*.

### Read out Touch Panel Keyboard Buffer

Use this command to check, if there are bytes in the input buffer. If the variable is greater than 0, you can read out the buffer, otherwise the buffer is empty.

#### GET #TP, #0, #UFCI\_IBU\_FILL, Number, Variable

**Number** is a constant, a variable or expression of the data type BYTE, WORD, LONG and specifies the length of output.

**Variable** is a variable of the data type BYTE, WORD, LONG or STRING to read out the number of bytes in input buffer.

If there is at least one byte in the buffer, you can read out the generated keycode from TOUCHPANEL.TDD with following command.

#### GET #TP, #0, Number, Variable

**Number** is a constant, a variable or expression of the data type BYTE, WORD, LONG and specifies the length of output.

**Variable** is a variable of the data type BYTE, WORD, LONG or STRING to read out input buffer from touch panel keyboard.

Read out TOUCHPANEL.TDD buffer:

```
get #TP, #0, #UFCI_IBU_FILL, 0, wIbuFill ' get buffer length
if 0 < wIbuFill then                      ' check length of input buffer
  get #TP, #0, 0, sIInput$                ' get all generated keys
endif
```

### Auto Repeat

PUT #TP, #0, #TP\_AUTOREPEAT\_DELAY, n1

PUT #TP, #0, #TP\_AUTOREPEAT\_RATE, n2

This command adjusts the delay and repeat rate of the touch panel keyboard.

**n1** is a constant, variable or expression of the data type BYTE, WORD, LONG and specifies the delay (time between keystroke and generation of the 1st code) in ms (0=inactive)

**n2** is a constant, variable or expression of the data type BYTE, WORD, LONG and specifies the repeat rate in ms (a new code is generated by the keyboard every n2 x ms)

The auto repeat function can be activated in *TOUCHPANEL.TDD*. If keys are to receive no auto-repeat function they have a special key attribute (See key attributes, BIT-4). The auto repeat function is global, if you want to deactivate it for another window, please deactivate it with this command, before showing the new window.

# Templates

In this chapter we will show some example applications using the Tiger Graphic Library which can be used as templates for your own projects. Unlike the *Integrated Applications* these templates are not part of the Tiger Graphic Library. If you want to use one of their subroutines please copy them into your own program code and modify them for your own standards.

Available templates for the Tiger Graphic Library:

- Demo Menu

### Demo Menu

The file "TGL\_MENU.TIG" shows a simple menu created with the TP-1000. The window consists of the background graphic, which is ORed into the window. This is necessary, because the format of the bitmap is 320\*240  $\Rightarrow$  the whole LCD. Otherwise the other elements would be overwritten. 7 Buttons are created and every button is linked with an alternative graphic. If the button is pressed, the alternative graphic is shown. This has a 3D effect. If a menu button is pressed, a "switch...case" decided, which button was pressed. You could put your code there. This menu is also part of the TP-1000 Demo program.

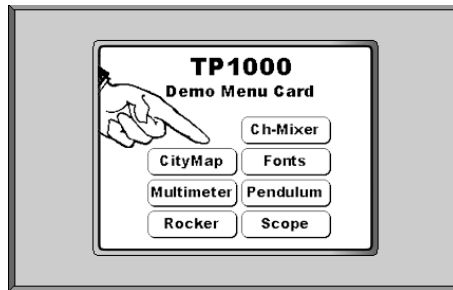


figure 110: Application menu

## Templates

TGL\_MENU.INC:

```
word wgIdentifier

#include TigerGraphicLibrary.INC

'=====
' Menu buttons parameter
'=====
#define PD_BUTTON_HEIGHT      31
#define PD_BUTTON_WIDTH       101
#define PD_BUTTON_BMP_WIDTH   104

#define PD_NUM_OF_BUTTONS     7

#define PD_MENU_WINDOW        0

'=====
' Menu 1 buttons
'=====
#define PD_MENU_MIXER          0
#define PD_MENU_CITYMAP        1
#define PD_MENU_FONTS          2
#define PD_MENU_MULTIMETER     3
#define PD_MENU_PENDULUM       4
#define PD_MENU_ROCKER         5
#define PD_MENU_SCOPE          6

task main
    datalabel BUTTON_0, MENU_BACKGROUND
    long llButtonAdr
    word wlCurWindow
    word wlParentId          ' identifier of parent element for docking
    byte blDockingOption     ' current docking option
    byte blReturnCode        ' keycode
    long llXOffs             ' x-coordinate offset for docking
    long llYOffs             ' y-coordinate offset for docking
    word wlBackgroundIdentifier ' unique identifier of background
    long llBmpLen
    byte blReturn

    llBmpLen = (PD_BUTTON_HEIGHT * PD_BUTTON_BMP_WIDTH) / 8

    '.....
    ' beeper
    '.....
    dir_pin 4, 2, 0

    '.....
    ' reset LCD
    '.....
    DIR_PIN 8, 5, 0          ' L85 is Reset pin of the LCD
    OUT 8, 00100000b, 0      ' Reset the LCD
    OUT 8, 00100000b, 255    ' Be sure there is no reset more
    WAIT_DURATION 100        ' wait that the LCD is not busy

    DIR_PIN 8, 2, 0
```

```

OUT 8, 00000100B, 0          ' Backlight on

'.....
' install device drivers
'.....
INSTALL_DEVICE #TP, "TOUCHPANEL.TDD", TP_TYP_1
INSTALL_DEVICE #LCD, "LCD-S1D13700.TD2",0,0,0EEH, 1, 250, 2, 0

'.....
' create buttons
'.....
call vTglInit
llButtonAdr = BUTTON_0          ' address of the first button

wgIdentifier = 0

'.....
' create background (graphic element)
'.....
wlBackgroundIdentifier = wgIdentifier ' save identifier of background
call bTglCreateGraphic( LCD_WIDTH,LCD_HEIGHT, MENU_BACKGROUND,LCD_WIDTH, &
wlBackgroundIdentifier, blReturn ) ' create background element
wgIdentifier = wgIdentifier + 1

wlCurWindow = PD_MENU_WINDOW

'.....
' create menu buttons
'.....
blReturnCode = 0

call bTglCreateButtonWnd( PD_BUTTON_WIDTH, PD_BUTTON_HEIGHT,&
llButtonAdr, PD_BUTTON_BMP_WIDTH, 0, wgIdentifier, wlCurWindow,&
161, 82, blReturnCode, blReturn)

blDockingOption = TGL_OPT_BOTTOM_LEFT ' dock to the bottom left first
llXOffs = -4
llYOffs = 4
wlParentId = wgIdentifier          ' save parent element
call lCreateButtonInc(blReturnCode, wgIdentifier, llButtonAdr, llBmpLen)

call bTglCreateGraphic( PD_BUTTON_WIDTH, PD_BUTTON_HEIGHT, llButtonAdr, &
PD_BUTTON_BMP_WIDTH, wgIdentifier, blReturn ) ' background
call bTglLink(wlParentId, wgIdentifier, blReturn)
wgIdentifier = wgIdentifier + 1
llButtonAdr = llButtonAdr + llBmpLen

loop PD_NUM_OF_BUTTONS-1

call bTglCreateButtonDockWnd( PD_BUTTON_WIDTH, PD_BUTTON_HEIGHT,&
llButtonAdr, PD_BUTTON_BMP_WIDTH, 0, wlParentId, wgIdentifier,&
wlCurWindow, llXOffs, llYOffs, blDockingOption, &
blReturnCode, blReturn)
wlParentId = wgIdentifier          ' next parent element
if blDockingOption = TGL_OPT_BOTTOM_LEFT then
blDockingOption = TGL_OPT_RIGHT          ' dock to the right now

```



```

        llXOffs = 4                                ' space: 4 Pixel
        llYOffs = 0                                '
    else
        blDockingOption = TGL_OPT_BOTTOM_LEFT      ' dock to the bottom left
        llXOffs = -4                                '
        llYOffs = 4                                ' space: 4 Pixel
    endif
    call lCreateButtonInc(blReturnCode, wgIdentifier, llButtonAdr, llBmpLen)
    call bTglCreateGraphic( PD_BUTTON_WIDTH, PD_BUTTON_HEIGHT, llButtonAdr, &
        PD_BUTTON_BMP_WIDTH, wgIdentifier, blReturn )
    call bTglLink(wlParentId, wgIdentifier, blReturn)
    wgIdentifier = wgIdentifier + 1
    llButtonAdr = llButtonAdr + llBmpLen
endloop

call bTglPlaceGraphicInWindow( wlBackgroundIdentifier, wlCurWindow, &
    0, 0, blReturn )    ' place background graphic

' set graphic mode OR
call bTglSetAttribute(wlBackgroundIdentifier, wlCurWindow, &
    TGL_SHOW_MODE, TGL_SHOW_MODE_OR, blReturn)

' .....
' show start menu
' .....
call bTglShowWindow(PD_MENU_WINDOW, blReturn)

' .....
' read out buffer
' .....
byte key_index
long ibu_fill                                ' input buffer len of touch panel driver
while l=1                                    ' <===== LOOP =====>
    ibu_fill = 0                            ' init
    while ibu_fill = 0                      ' <==== LOOP =====>
        GET #TP, #0, #UFCI_IBU_FILL, 0, ibu_fill    ' get buffer length
        release_task                            '
    endwhile                                ' <==== LOOP =====>
    GET #TP, #0, 1, key_index                ' get index of key

    switch key_index
    CASE PD_MENU_MIXER:                      ' Mixer Menu
        ' place your code here
    CASE PD_MENU_MULTIMETER:
        ' place your code here
    CASE PD_MENU_FONTS:
        ' place your code here
    CASE PD_MENU PENDULUM:
        ' place your code here
    CASE PD_MENU_CITYMAP:
        ' place your code here
    CASE PD_MENU_SCOPE:
        ' place your code here
    CASE PD_MENU_ROCKER:
        ' place your code here
    endswitch

endwhile                                ' <===== LOOP =====>

```

```
BUTTON_0::
DATA FILTER "btn_chmixer.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_chmixer_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_citymap.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_citymap_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_fonts.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_fonts_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_multimeter.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_multimeter_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_pendulum.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_pendulum_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_rocker.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_rocker_active.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_scope.bmp", "GRAPHFLT", 0      '
DATA FILTER "btn_scope_active.bmp", "GRAPHFLT", 0      '

MENU_BACKGROUND::
DATA FILTER "background_320x240.bmp", "GRAPHFLT", 0      '
end

sub lCreateButtonInc(var byte bpvReturnCode; var word wpvIdentifier; &
    var long lpvButtonAdr; long lpBmpLen)
    bpvReturnCode = bpvReturnCode + 1
    wpvIdentifier = wpvIdentifier + 1
    lpvButtonAdr = lpvButtonAdr + lpBmpLen      ' increment to next Bitmap
end
```

## Error Codes

Each subroutine of the Tiger Graphic Library returns a result about its operation. It helps you debugging your program. The return value informs you about the following details:

- validity of used parameters
- correctness of usage
- size of reserved memory in the user configuration file  
*TigerGraphicLibraryConf.INC*

No.	Name	Description
0	TGL_MSG_OK	OK Exit
1	TGL_ERR_ELEMENT_INVA LID	You have to reserve more space for elements in file <i>TigerGraphicLibraryConf.INC</i> . Increase the value of <i>TGL_MAX_NUM_ELEMENTS</i> or use smaller identifier, if available.
2	TGL_ERR_WINDOW_INVA LID	The identifier of the window is too large. If you want to use more windows, please increase the value of <i>TGL_MAX_NUM_WINDOWS</i> in <i>TigerGraphicLibraryConf.INC</i>
3	TGL_ERR_WINDOW_STR_ LEN	Reserved RAM for elements in windows is full. Increase the value of <i>TGL_WINDOW_ATTRIBUTES_LEN</i> in <i>TigerGraphicLibraryConf.INC</i>
4	TGL_ERR_ID_EXISTING	Element of same identifier is existing already, please chose another identifier or delete the element of this number
5	TGL_ERR_ELEMENT_NOT _FOUND_WND	Element has not been placed in this window. Please place the element in the window.
6	TGL_ERR_TOO_MANY_EL EMENTS	You have to reserve more space for specified elements in file <i>TigerGraphicLibraryConf.INC</i> . Increase the value of maximum number of specified element. E.g. increase <i>TGL_MAX_NUM_BUTTONS</i> , if specified element is a button.
7	TGL_ERR_ALREADY_PLAC ED	This element is already placed in this window. NEVER place one element several times in the same window.

No.	Name	Description
8	TGL_ERR_DACC_STRI_FU LL	Direct Access String for TOUCHPANEL.TDD is too short. Please increase TGL_BUTTON_DACC_LEN or TGL_SLIDER_DACC_LEN in TigerGraphicLibraryConf.INC
9	TGL_ERR_INVALID_ATTRI BUTE	Attribute does not exist
10	TGL_ERR_INVALID_ATTRI BUTE_VALUE	Value for attribute is not allowed
11	TGL_ERR_INVALID_POSIT ION	Position in string does not exist
12	TGL_ERR_INVALID_SIZE	Size of element is not valid. E.g.: The slider button must fit into the slide bar
13	TGL_ERR_ELEMENT_NOT _ACTIVE	Element is not active at the moment. Element must be active for this operation. The window must be active and the element must be shown.
14	TGL_ERR_INVALID_PARA METER	Selected option is not available
15	TGL_MSG_NO_CHANGE	Element has not changed (no inversion ...)
16	TGL_ERR_TEXT_TOO_LO NG	Text does not fit into the element. Please cut one or more chars and try again.
17	TGL_ERR_ELEMENT_NOT _FOUND	Element not created yet
18	free	
19	free	
20	TGL_ERR_INVALID_BUT_ CODE	Keycode for button must be between 0 and 255 (1 byte).
21	TGL_ERR_TEXT_MEM_OV ERFLOW	Not enough reserved RAM for text. Please increase <i>TGL_MAX_MEM_TEXTS_LEN</i> in TigerGraphicLibraryConf.INC
23	TGL_ERR_INVALID_LINK	This link is not allowed. Please read documentation for <i>bTglLink</i>
24	free	
25	TGL_ERR_NO_GFK_TEXT_ AREA	No area for text in element, caused by too wide frame. Please decrement frame thickness or size up element.
26	free	

No.	Name	Description
30	TGL_ERR_INVALID_TYPE	Operation for this type of element not allowed. Please see documentation of this subroutine.
31	TGL_ERR_NO_SWITCH	Button must be declared as switch. This operation is not allowed for standard buttons. To declare a button as switch, please set the key attribute to <i>TGL_KEY_ATTR_SWITCH</i>
32	TGL_ERR_LCD_CPY_STR_TO_SMALL	Passed parameter for copy of window graphic is smaller than <i>LCD_SIZE</i>
40	TGL_ERR_TYPE_INVALID	not existing type or invalid type for this function
41	TGL_ERR_LIST_OVERFLOW	invalid index of list
42	TGL_ERR_NO_FLASH_ADDRESS	text resp. bitmap has no flash address
43	TGL_ERR_BASE_INVALID	invalid base value for this type resp. this element
44	free	
50	TGL_ERR_ELEMENT_OUT_OF_LCD_AREA	The element does not fit in window. Please check placing coordinates of element.
51	TGL_ERR_TOO_MANY_BLINK	too many blinking elements in one window
52	free	
60	TGL_ERR_INTERNAL_KEYCODE_OVERFLOW	Too many internal keycodes. Increase <i>TGL_MAX_NUM_BUTTONS_IN_WINDOW</i> in <i>TigerGraphicLibraryConf.TIG</i>
61	TGL_ERR_MISSING_INTERNAL_KEYCODE	Fatal error. Please call support
62	free	
63	free	
81	TGL_ERR_FONT_TASKS_OVERFLOW	<i>sCreateTxtGraphic</i> is called in too many tasks. Please increase <i>TGL_MAX_NUM_TXT_GRAPHIC_STR</i> in <i>TigerGraphicLibraryConf.INC</i>
82	TGL_ERR_FONT_GRAPHIC_OVERFLOW	Text graphic does not fit in the internal string. Occasionally increase <i>TGL_TXT_GRAPHIC_LEN</i> in <i>TigerGraphicLibraryConf.INC</i>
83	TGL_ERR_FONT_INVALID	Index of format string is too high. Please increase <i>TGL_MAX_NUM_FONTS</i> in <i>TigerGraphicLibraryConf.INC</i>

No.	Name	Description
84	TGL_ERR_FONT_NAME_INVALID	Invalid font name choice. Please see documentation of <i>bTglCreateFont</i> .
85	TGL_ERR_FONT_TYPE_INVALID	Invalid font type choice. Please see documentation of <i>bTglCreateFont</i> .
86	TGL_ERR_FONT_ALIGN_VERTICAL_INVALID	Invalid vertical alignment parameter. Please see documentation of <i>bTglCreateFontParams</i> .
87	TGL_ERR_FONT_ALIGN_HORIZONTAL_INVALID	Invalid horizontal alignment parameter. Please see documentation of <i>bTglCreateFontParams</i> .
88	TGL_ERR_FONT_SPACING_TYPE_INVALID	Invalid spacing type parameter
89	TGL_ERR_FONT_OVERLAY_INVALID	Invalid overlay mode. Please see documentation of <i>bTglCreateFontParams</i> .
90	TGL_ERR_FONT_WRAP_INVALID	Invalid wrapping option. Please see documentation of <i>bTglCreateFontParams</i> .
91	TGL_ERR_FONT_DIRECTION_INVALID	Invalid writing direction for text
92	TGL_ERR_FONT_NOT_EXISTING	Font has not been created or its parameters are damaged.
93	TGL_ERR_FONT_ALREADY_EXISTING	Font has been already created
94	TGL_ERR_FONT_SIZE_INVALID	Invalid font size for this font
95	TGL_ERR_FONT_NOT_INCLUDED	Decomment font in TigerGraphicLibraryConf.INC
96	TGL_ERR_TEXT_VARIABLE_OVERFLOW	Internal variable overflow. Increment TGL_TXT_GRAPHIC_TXT_LEN in TigerGraphicLibraryConf.INC
97	TGL_ERR_TEXT_GRAPHIC_VARIABLE_OVERFLOW	Internal variable overflow. Increment TGL_TXT_GRAPHIC_LEN in TigerGraphicLibraryConf.INC
98	free	
100	TGL_ERR_FATAL	Fatal system error => please check with programmers
101	free	
110	TGL_ERR_INVALID_STYLE	Style is not available for this subroutine. Please see description of subroutine.

## Error Codes

No.	Name	Description
111	TGL_ERR_GRAPH_INVALID_DATAWIDTH	Invalid data width for the graph's values. Please see description of subroutine.
112	TGL_ERR_GRAPH_INVALID_NUM_VAL	Invalid number of values for a graph
113	free	
150	TGL_NO_ALT_GRA	No alternative graphic available for this element. Please call <i>vTgLink</i> for setting an alternative graphic for an element.

## Overview of Example Programs

Name	Type	Description
GRAPHIC_FONTS_solo.TIG	graphic fonts	demonstrates how to work with graphic fonts without using elements and windows
TGL_EXAMPLE_BUTTON.TIG	button	getting started example for creation of a button on the touch panel
TGL_EXAMPLE_SLIDER.TIG	slider	getting started example for creation of a slider on the touch panel
TGL_GraphicalUserInterface.TIG	GUI	demonstrates an easy way of programming a graphic user interface
TGL_KEYBOARD_selfmade.TIG	keyboard	demonstrates how to generate a keyboard with the keys of your choice with one command only
TGL_KEYBOARD_selfmade_plus_text.TIG	keyboard	demonstrates how to place additionally labels in a selfmade keyboard
TGL_KEYBOARD_selfmade_shift.TIG	keyboard	Demonstrates how to create a keyboard with keys of your own choice including <ul style="list-style-type: none"> <li>- a special shift key for changing the keys</li> <li>- change of the size for the user input</li> <li>- up sized single keys</li> </ul> Additionally this example gives you an example how to organize your code for a graphical user interface
TGL_KEYBOARD_STYLE_S.TIG	keyboard	shows all predefined keyboard styles
TGL_STEP_1_button.TIG	button	explain all steps which are necessary to create a button on the LCD
TGL_STEP_2_label.TIG	label	explain all steps which are necessary to show a graphic text on the LCD
TGL_BLINK.TIG	general	shows a blinking element
TGL_BLINK_modes.TIG	general	shows blinking elements in all blinking modes and blinking speeds
TGL_bTglDeleteElement.TIG	general	demonstrates how to delete and recreate an element
TGL_bTglDeleteElementFromWindow.TIG	general	demonstrates how to delete an element from a window without deleting the element itself.



Name	Type	Description
TGL_bTglSetAttribute.TIG	general	set attributes of an element
TGL_bTglSetCoordinates_bTglGetCoordinates.TIG	general	demonstrates how to set and get the coordinates of an element in a window
TGL_bTglSetText.TIG	general	changes the text which is saved with a label
TGL_bTglShowText.TIG	general	displays a text using the area of a label without changing its text
TGL_BUTTON_alternative_bitmap.TIG	button	Creates button with alternative graphic (graphic is shown, when button is pressed)
TGL_BUTTON_beep_off_inverted.TIG	button	Creates a button without beep and with inversion
TGL_BUTTON_create_dock.TIG	button	Example for using the 2 functions create and docking in window for buttons
TGL_BUTTON_create_place.TIG	button	Example for using the single create and place functions for buttons
TGL_BUTTON_createWnd.TIG	button	Creates a standard button with beep calling the combined create and place subroutine
TGL_BUTTON_pressed_states.TIG	button	demonstrates the possibilities of marking a pressed element
TGL_BUTTON_rotate.TIG	button	demonstrates how to rotate buttons
TGL_CHART_linear_gauge	gauge	Shows linear chart.
TGL_CHECKBOX.TIG	button	Creates a checkbox
TGL_GAUGE.TIG	gauge	Creates a linear gauge with a constant value
TGL_GAUGE_dial_indicator.TIG	gauge	Shows a working dial indicator
TGL_GAUGE_speedometer.TIG	gauge	Shows a dial indicator styled as speedometer
TGL_GAUGE_types.TIG	gauge	Show different types of gauges
TGL_GRAPHIC_createWnd.TIG	graphic	Creates a graphic and places it in a window in one go
TGL_GRAPHIC_dockWnd.TIG	graphic	Creates a graphic and places it in a window relative to an existing element
TGL_GRAPHIC_rotate.TIG	graphic	demonstrates how to create rotated graphics

## Overview of Example Programs

Name	Type	Description
G		
TGL_KEYBOARD.TIG	keyboard	demonstrates how to initialize one style of the keyboards.
TGL_LABEL_background.TIG	label	Create a label with a bitmap background
TGL_LABEL_background_varText.TIG	label	Create a label with a bitmap background and shows texts.
TGL_LABEL_createFWnd.TIG	label	Creates and places a label in one go. The text is given by a flash address.
TGL_LABEL_createWnd.TIG	label	Creates a label and places it in a window in one go
TGL_LABEL_dockWnd.TIG	label	Creates a label and places it in a window relative to an existing element
TGL_LABEL_rotate.TIG	label	demonstrates how to create rotated labels
TGL_LISTBOX_createWnd.TIG	listbox	Create and place a listbox in a window
TGL_LISTBOX_set_get_index.TIG	listbox	Set and get indices of listboxes
TGL_ITglCalcTextToWindow.TIG	graphic fonts	Calculates number of graphic characters fitting in an area
TGL_MENU.TIG	template	Demonstrates how to realize a simple menu for your TP-1000
TGL_RTC_APPLICATION.TIG	rtc application	Demonstrates how to initialize one style of the RTC applications.
TGL_SHOW_GRAPH	general	Demonstrates how to build and show a graph of given values, e.g. measurands
TGL_SHOW_GRAPH_axes	general	Shows the types for axes of graphs
TGL_SHOW_GRAPH_clear	general	Demonstrates how to erase a graph from LCD
TGL_SHOW_GRAPH_incremental	general	Demonstrates how to build and show a graph of given values, e.g. measurands
TGL_SHOW_HIDE.TIG	general	Demonstrates the functionality of show and hide

## Overview of Example Programs

Name	Type	Description
		with the element button
TGL_SLIDER_X_set_value.TIG	slider	Demonstrates how to set a slider value by program code
TGL_SLIDER_X_show_value.TIG	slider	Shows the actual slider value
TGL_SLIDER_X_show_value_rotated.TIG	slider	Displays the value of an rotated slider.
TGL_SLIDER_XandY_show_value.TIG	slider	Shows the values of an x and y slider
TGL_SLIDER_Y_create_dock.TIG	slider	Demonstrates how to create a slider and placing it relative to an existing element
TGL_SLIDER_Y_create_place.TIG	slider	Demonstrates how to create a slider and placing it in a window
TGL_SLIDER_Y_createWindow.TIG	slider	Demonstrates how to create and place a slider in one go
TGL_SLIDER_Y_dockWindow.TIG	slider	Demonstrates how to create and dock a slider relative to an other element in one go
TGL_SLIDER_Y_preset.TIG	slider	Demonstrates how to preset a slider value
TGL_SLIDER_Y_set_value.TIG	slider	Demonstrates how to set a slider value by program code
TGL_SLIDER_Y_show_value.TIG	slider	Shows the actual slider value
TGL_SLIDER_Y_show_value_preset.TIG	slider	Demonstrates how to preset a slider value and show its current value
TGL_SOME_WINDOWS.TIG	template	Demonstrates how to switch between windows
TGL_STANDBY.TIG	general	Demonstrates how to use the stand-by function of the Tiger Graphic Library
TGL_STANDBY_LCD_terminated.TIG	general	Demonstrates how to leave stand-by mode by LCD activity
TGL_STANDBY_switch.TIG	general	Demonstrates how to switch to stand-by mode by button
TGL_SWITCH_get_state.TIG	button	Get the state of a switch
TGL_SWITCH_inverted.TIG	button	Creates a switch button without beep and with

Name	Type	Description
IG		inversion
TGL_SWITCH_save_states.TIG	button	Demonstrates how switches hold their states in case of changed windows
TGL_SWITCH_set_state.TIG	buttons	Creates a switch button with alternative graphic and changes the state of the switch button with BASIC Code
TGL_SWITCHES.TIG	button	Example for several switches in one window
TGL_TEXTBUTTON_alternative_text.TIG	text button	Demonstrates how to create a text button showing an alternative text when it is pressed
TGL_TEXTBUTTON_alternative_text_rotate.TIG	text button	Demonstrates how to create a rotated text button showing an alternative text when it is pressed
TGL_TEXTBUTTON_createWnd.TIG	text button	Demonstrates how to create and place a text button in one go
TGL_TEXTBUTTON_inverted_switch.TIG	text button	Creates an inverting text button switch
TGL_TEXTBUTTON_rotate.TIG	text button	Demonstrates how to create rotated text buttons
TGL_TEXTBUTTON_switch.TIG	text button	Demonstrates how to use a text button as switch showing an alternative text
TGL_TEXTBUTTONS_DOCKING_OPTIONS.TIG	text button	Demonstrates how to place text buttons relative to an existing element
TGL_USER_GRAPHIC_analog_clock.TIG	user graphic	Example for showing a selfmade user graphic on LCD
TGL_wTglGetTouchedElement.TIG	general	Efficient polling of switches, sliders and listboxes.
FONT_ChangeFontInText.TIG	graphic fonts	Demonstrates how to change the font in one text
FONT_TYPES.TIG	graphic fonts	Shows all font types
FONT_AllFonts.TIG	graphic fonts	Shows all existing graphic fonts
FONT_VariousFonts.TIG	graphic fonts	Shows selected graphic fonts

Name	Type	Description
FONT_ALIGNMENT_AllDirections.TIG	graphic fonts	Shows alignments in all directions
FONT_CHARSET_Hungarian	graphic fonts	Shows Hungarian charset
FONT_HelloWorld.TIG	graphic fonts	Hello world program for graphic fonts
FONT_SPACING_AllTypes.TIG	graphic fonts	Shows all character spacing types
FONT_SPECIAL_CHARS.TIG	graphic fonts	Shows all special chars
FONT_WRAPPING_AllTypes.TIG	graphic fonts	Shows different line wrapping types for texts

## Overview of applications

Name	Description
Graphic_Fonts_Demo.TIG	Show all fonts of the Tiger Graphic Library
TP1000_Demo.tig	Demonstrate many functionalities of the Tiger Graphic Library
TP-1000_DEMO_Movie.TIG	Shows intro movie of TP1000_DEMO
TP1000_DMX_DEMO.TIG	Application for a DMX512 controller
Doorbells.TIG	Demo for doorbells in apartment houses
Dot_Slider.TIG	Dot styled slider
PlusMinus_Slider.TIG	Plus minus styled slider
TwoDots_Slider.TIG	Two dot slider
TERMINAL.tig	Demo for a VT100 simulation by TP1000
Wellness.TIG	Control unit for lightning and wellness programs

## Overview of Include Files

Name	Description
Define_a.INC	general symbol definitions
Ufunc4.INC	definitions user function codes
TP_CALIBRATE.INC	basic touch panel definitions and subroutines
TigerBasicLibrary.INC	general Tiger-BASIC subroutines
TigerGraphicLibrary.INC	file inclusions
TigerGraphicLibraryClbks.INC	callback tasks
TigerGraphicLibraryConf.INC	user configurations
TigerGraphicLibraryDefs.INC	definitions and error codes
TigerGraphicLibraryDoc.INC	details about versions
TigerGraphicLibraryGlobs.INC	global variables
TigerGraphicLibrarySubs.INC	internal subroutines
TGL_BUTTON.INC	subroutines for buttons and text buttons
TGL_CHECKS.INC	subroutines for checking validity of parameters and data consistency
TGL_GAUGE.INC	subroutines for radial and linear gauges
TGL_GENERAL.INC	user callable general subroutines
TGL_GRAPHIC.INC	subroutines for graphics
TGL_GRAPHICAL_FUNCTIONS.INC	general graphical functions
TGL_KEYBOARD.INC	subroutines for keyboard applications
TGL_KEYBOARD_DEFS.INC	definitions for keyboard applications
TGL_LABEL.INC	subroutines for labels
TGL_LISTBOX.INC	subroutines for listboxes
TGL_RTC.INC	subroutines for real time clock applications
TGL_RTC_DEFS.INC	definitions for real time clock applications
TGL_SLIDER.INC	subroutines for sliders
TGL_USER_GRAPHIC.INC	subroutines for handling of own user graphic using the Tiger Graphic Library

## Overview of Include Files

Name	Description
TGL_lockLowLevel.INC	code lines for tgl locking in internal tgl task
TGL_DEVICE_DRIVERS_TP1000	installation of device drivers for the TP1000
TGL_GRAFO__callables_i.inc	callable subroutines for users
TGL_GRAFO__defines_d.inc	definitions for graphic fonts
TGL_GRAFO__font_id2_d.inc	definitions for id2 fonts
TGL_GRAFO__globals_v.inc	global variables for graphic fonts
TGL_GRAFO__subs_i.inc	internal subroutines for graphic fonts
TGL_GRAFO__txt_ctrl_d.inc	definitions for text control chars
e.g. FONT_ID2_4_VALENCIA.INC	These files hold a list of bitmap-fonts to be used on graphical devices as LCD
e.g. FONT_ID2_VALENCIA_10_NORMAL.INC	These files hold head-information, width table and pixel data of a bitmap-font



## Documentation History

Version of	Description / Changes
1.00	first version
1.01	new functionalities new examples new include files bug fixes
1.02	bug fixes e.g.: - vTglGetButtonState for text button switches - lTglCalcTextToWindow for special chars - exact alignments - bTglShow: inverted switches new examples - Doorbells - TGL_BUTTON_pressed_states.TIG - TGL_SWITCH_save_states.TIG enhanced TP1000demo accelerations e.g. keyboards lower need of RAM
1.03	bug fixes - graphic fonts: constant center spacing
1.04	bug fixes - bTglCreateTextButtonF - touch panel handling sgTglShowButton
1.05	changes - acceleration of graphic fonts especially special chars new examples - FONT_ChangeFontInText.TIG - TGL_BLINK.TIG - TGL_BLINK_modes.TIG - TGL_STANDBY.TIG - TGL_STANDBY_LCD_terminated - TGL_STANDBY_LCD_terminated new applications - Graphic_Fonts_Demo.TIG

Version of	Description / Changes
	<ul style="list-style-type: none"> <li>- TP1000_DMX_Demo.TIG</li> <li>- Doorbells.TIG</li> <li>- TERMINAL.tig</li> </ul> bug fixes new features <ul style="list-style-type: none"> <li>- bigger fonts,</li> <li>- new font family Stockholm</li> <li>- new functionality font change in text</li> </ul>
1.06	bug fixes <ul style="list-style-type: none"> <li>- bTglSetAttribute - TGL_ATTR_INVERT</li> </ul>
1.07	bug fixes <ul style="list-style-type: none"> <li>- bTglGetSliderValue</li> <li>- bTglSetAttribute - TGL_ATTR_INVERT</li> </ul> enhancements <ul style="list-style-type: none"> <li>- gauges</li> <li>- blink</li> <li>- standby</li> </ul>
1.08	bug fixes <ul style="list-style-type: none"> <li>- bTglShowText for textbuttons</li> <li>- textbutton with text stored in flash memory</li> <li>- bTglSetSliderLimits</li> <li>- selfmade keyboard with changing of keys</li> <li>- selfmade keyboard with wide buttons</li> </ul> applications <ul style="list-style-type: none"> <li>- styled sliders</li> </ul>
1.09	bug fixes <ul style="list-style-type: none"> <li>- btglShowText for textbuttons</li> <li>- textbutton with text stored in flash memory</li> <li>- blinking of textbuttons and labels</li> </ul> enhancements <ul style="list-style-type: none"> <li>- set inversion state of elements TGL_ATTR_INV_STATE</li> </ul>
1.10	enhancements <ul style="list-style-type: none"> <li>- sTglGetKeyblnputTimeout</li> <li>- bTglCreateFont tolerant parameters resp. " _-"</li> <li>- lTglGetFontHeight</li> <li>- bTglSetMargins/bTglGetMargins</li> </ul>

Version of	Description / Changes
	<ul style="list-style-type: none"> <li>- bTglGetTouch</li> <li>- bTglGetPushButtonState</li> </ul> bug fixes <ul style="list-style-type: none"> <li>- cursor off</li> <li>- text position keyboard view</li> <li>- create text elements check font</li> </ul>
1.11	bug fixes <ul style="list-style-type: none"> <li>- bTglGetButtonState/bTglSetButtonState</li> <li>- switches: non following identifiers</li> <li>- blink text elements</li> </ul> enhancements <ul style="list-style-type: none"> <li>- delete text elements</li> </ul>
1.12	enhancements <ul style="list-style-type: none"> <li>- new element listbox</li> <li>- bTglCreateFont/bTglCreateFontParams/bTglSetFont</li> <li>- new error message TGL_ERR_FONT_NOT_INCLUDED</li> <li>- default spacing for constant spacing of letters and digits</li> </ul> bug fixes <ul style="list-style-type: none"> <li>- bTglSetText/bTglShow text refresh on LCD</li> <li>- key beep after vTglHideWindow</li> <li>- alignments in bitmaps helsinki 52,56,60</li> </ul>
1.13	enhancements <ul style="list-style-type: none"> <li>- rtc style 1 -&gt; setting of day of week</li> </ul> bug fixes <ul style="list-style-type: none"> <li>- update bars base right,top,bottom</li> </ul>
1.14	enhancements <ul style="list-style-type: none"> <li>- fully multitasking ability</li> <li>- new error codes for internal string overflow using graphic fonts</li> <li>- step by step lessons for the programming of applications using the Tiger Graphic Library</li> </ul> bug fixes <ul style="list-style-type: none"> <li>- line height with font select</li> <li>- line wrapping by cr</li> <li>- missing initializations in the keyboard and checking subroutines</li> </ul>