Tiger Graphic Library
V2.02

Index

Tiger Graphic Library V2.02 1

Index 2

Introduction 10

Programming with the Tiger Graphic Library 13

Basic Program Structure for Applications 15
- Show a Window with Elements on LCD 18
- Realize a Graphical User Interface 32
- Organize Window Changing 45
- Modifiy and Update Elements 51
- Additional Features 57
  - Free User Flash Sectors 58
  - Saving Data in Eeprom 59
  - Stand-by 61
  - Password Protection 62
  - Blinking Elements 63
  - Moving Elements 64

Components and its Handling 65
- Elements 65
- Windows 66
- Creating and Placing Elements 67
- Docking 68
- Integrated Applications 69
- Graphic Fonts 70
- User Graphics 71
- Graphical Functions 72
- Tips and Tricks 74

Configuration 76
- General Settings 77
- Master Defines 78
- LCD Settings 80
- Memory for Elements 82
- Memory for Windows 83
- Memory for Graphic Fonts 84
- Choice of Graphic Fonts 86
- Choice of Keyboards 87
- Choice of RTC Applications 88
- Hardware 89

Versions 90

Application Programming Interface (API) 91

General Subroutines 92

bTglInit 94
bTglLink 95
bTglDeleteElement 97
bTglDeleteElementFromWindow 99
bTglShowWindow 101
bTglHideWindow 102
bTglShow 103
bTglHide 104
bTglShowText 107
bTglShowLong, bTglShowWord, bTglShowByte 108
bTglShowReal 112
bTglShowDetail, ShowDetailF 114
bTglShowGraph 115
bTglHideGraph 119
bTglUpdateScope 120
vTglUpdate 121
vTglUpdateParams 121
wTglGetType 122
bTglSetSize 123
wTglGetSize 123
bTglSetAddress 124
lTglGetAddress 124
bTglSetBmpWidth 125
wTglGetBmpWidth 125
bTglSetText 126
sTglGetText 128
bTglSetFont 129
bTglSetFrame 130
bTglSetMargins 131
wTglGetMargins 131
bTglSetCoordinates 132
wTglGetCoordinates 132
bTglSetAttribute 134
lTglGetAttribute 139
vTglSetStandbyTime 141
lTglGetStandbyTime 142
bTglSetStandbyTermination 144
bTglSwitchStandby 147

vTglDelayStandby	149
lTglGetStandbyState	150
vTglWaitTouchTp	151
vTglWaitReleaseTp	151
bTglGetTouch	151
wTglGetTouchedElement	152
wTglGetNumTouchedElements	153
bTglGetTouchedElementsFlag	154
bTglCalibrateTp	155
vTglBeep	156
Graphic	**157**
wTglInitMaskWnd	159
bTglCreateGraphicWnd	160
bTglCreateGraphicDockWnd	162
bTglCreateGraphic	163
bTglPlaceGraphicInWindow	164
bTglDockGraphicInWindow	165
Label	**166**
bTglCreateLabelWnd, bTglCreateLabelFWnd, bTglCreateLabelVarWnd	167
bTglCreateLabelDockWnd, bTglCreateLabelFDockWnd, bTglCreateLabelVarDockWnd	170
bTglCreateLabel, bTglCreateLabelF, bTglCreateLabelVar	172
bTglPlaceLabelInWindow	174
bTglDockLabelInWindow	175
Button	**176**
Push Button	178
Switch Button	179
wTglInitPushButtonWnd	181
wTglInitPushButton	183
wTglInitSwitchButtonWnd	184
wTglInitSwitchButton	186
bTglCreateButtonWnd, bTglCreateButtonWndS	187
bTglCreateButtonDockWnd, bTglCreateButtonDockWndS	190
bTglCreateButton	193
bTglPlaceButtonInWindow, bTglPlaceButtonInWindowS	197
bTglDockButtonInWindow, bTglDockButtonInWindowS	199
bTglSetKeyAttributes	202
bTglGetKeyAttributes	203
bTglGetKeycode	204
bTglWaitKeycode	205
bTglGetPushButtonState	206

bTglSetButtonState 207
bTglGetButtonState 210

Text Button **213**
wTglInitPushTextButtonWnd, wTglInitPushTextButtonFWnd 216
wTglInitPushTextButton, wTglInitPushTextButtonF 217
wTglInitSwitchTextButtonWnd, wTglInitSwitchTextButtonFWnd 218
wTglInitSwitchTextButton, wTglInitSwitchTextButtonF 220
bTglCreateTextButtonWnd, bTglCreateTextButtonFWnd,
bTglCreateTextButtonVarWnd, bTglCreateTextButtonWndS,
bTglCreateTextButtonFWndS, bTglCreateTextButtonVarWndS 221
bTglCreateTextButtonDockWnd, bTglCreateTextButtonFDockWnd,
bTglCreateTextButtonVarDockWnd, bTglCreateTextButtonDockWndS,
bTglCreateTextButtonFDockWndS, bTglCreateTextButtonVarDockWndS 224
bTglCreateTextButton, bTglCreateTextButtonF, bTglCreateTextButtonVar 228
bTglPlaceTextButtonInWindow, bTglPlaceTextButtonInWindowS 230
bTglDockTextButtonInWindow, bTglDockTextButtonInWindowS 231

Slider **233**
wTglInitSliderWnd 237
wTglInitSlider 238
bTglCreateSliderWnd 239
bTglCreateSliderDockWnd 241
bTglCreateSlider 244
bTglPlaceSliderInWindow 246
bTglDockSliderInWindow 249
bTglSetSliderValue 252
lTglGetSliderValue 253
bTglSetSliderLimits 256

Listbox **257**
bTglCreateListboxWnd, bTglCreateListBoxFWnd 258
bTglCreateListboxDockWnd, bTglCreateListBoxFDockWnd 259
bTglCreateListbox, bTglCreateListboxF 262
bTglPlaceListboxInWindow 263
bTglDockListboxInWindow 264
sTglGetListboxItem 265
wTglGetListboxIndex 266
bTglSetListboxIndex 266

Gauge **269**
bTglCreateGaugeWnd 270
bTglCreateGaugeDockWnd 272
bTglCreateGauge 273

bTglPlaceGaugeInWindow 274
bTglDockGaugeInWindow 275
bTglShowGaugeValue 276

Keyboard **279**
wTglInitKeyboard 280
wTglInitKeyboardSelfmade 288
sTglGetKeyboardInput 297
sTglGetKeybInputTimeout 298
sTglEditText 299
sTglGetKeybPassword 300
sTglGetKeybPasswordTimeout 301
sTglGetKeybParams 302

RTC Applications **303**
bTglInitRtc 304
bTglSetRtc 306

Text Graphic **308**
Choosing the Graphic Fonts 310
Specific Parameters of Graphic Fonts 311
Designing Graphic Texts 317
Codes for Normal and Special Chars 319
Codes for Control Chars 322
Graphic Fonts Solo 324
Including Graphic Fonts 325
Hardware Configuration for the LCD 326
bTglCreateFont 328
bTglCreateFontParams 330
bTglSetFontParams 332
bTglGetFontParams 334
sTglBuildText, sTglBuildTextF 335
lTglCalcTextToWindow, lTglCalcTextToWindowF 337
lTglGetLineHeight 338
lTglCalcTextGraphicWidth 339

User Graphic **340**
vTglShowUserGraphic, vTglShowUserGraphicF 345
vTglShowUserGraphicParams, vTglShowUserGraphicParams 346
vTglHideUserGraphic 347
sTglGetWindowGraphic 348
vTglPutWindowGraphic, vTglPutWindowGraphic 349
vTglClearWindowGraphic 350
vTglPutStringToLcd 351

vTglPutStringToLcdParams 352
vTglPutFlashToLcd 353

Graphical Functions 354
sTglDrawRectangle 356
sTglDrawFrame 357
sTglDrawBar 358
sTglDraw 359
wTglCalcCircleParams 360
sTglDrawPie 361
sTglUpdatePie 362
sTglDrawHand 363
sTglUpdateHand 364
sTglDrawListbox 365
sTglUpdateScope 366
sTglDrawGraph 367
sTglDrawAxes 369
sTglClearGraph 370
sTglRotate, sTglRotateF 371
sTglRotateFToDst 373
sTglRotateFToDstOffs 374
sTglGraphicMove 375
sTglGraphicErase 376
sTglGraphicInvert 377

Eeprom 378
sReadStringFromEeprom 379
vWriteStringToEeprom 380

Touch Panel 381
Read out Touch Panel Keyboard Buffer 382
Auto Repeat 383

Templates 384
Demo Menu 385

Error Codes 390

Overview of Example Programs 395

Overview of applications 401

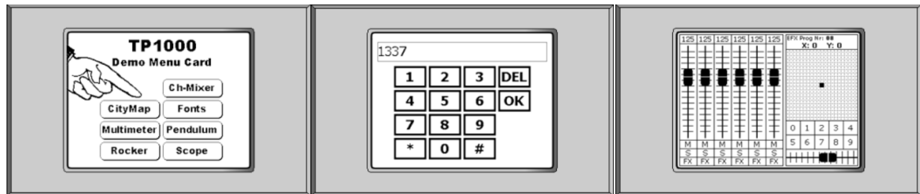
Overview of Include Files 402

Documentation History 404

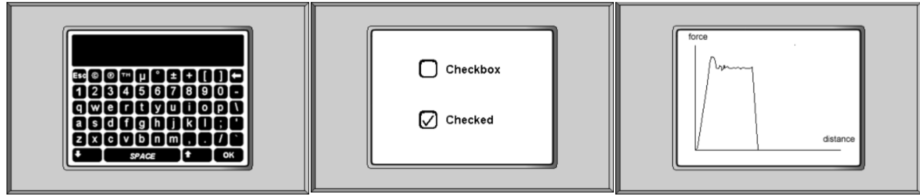
Blank Page

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.

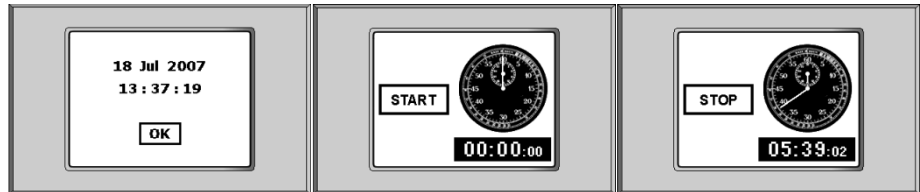


Now with the Tiger Graphic Library you are able to realize your project in a minimum of time and a maximum of user friendliness. Its great features will make it easy for you to give your program an individual nice looking without needs for a longer development time. You will find lots of example programs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



The Tiger Graphic Library provides subroutines for:

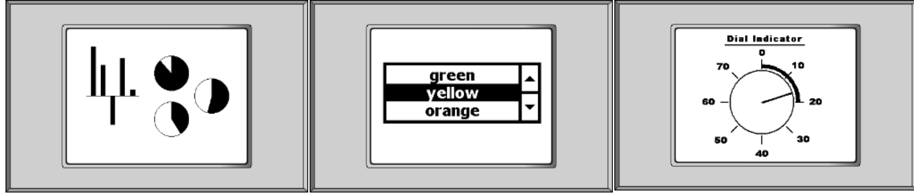
- creating buttons, switches and sliders for the touch panel
- listboxes for easy item selections
- graphic fonts with many special chars
- administrating whole pages containing touch panel functionalities and graphical elements for the LCD
- touch panel keyboards in different styles
- setting and displaying 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
- using the LCD in portrait, landscape, upright and upside-down mode



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.: *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Applications\TP1000_Demo\TP1000_Demo.tig*

For a hello world program for a button and a label please see *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Manuals\TigerGraphicLibrary\QuickStart_TigerGraphicLibrary_vXXX_yy.pdf*

For your first application with the Tiger Graphic Library we would suppose to start with the chapter *Programming with the Tiger Graphic Library*.

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 ready for cut and paste into your code *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary.*

Programming with the Tiger Graphic Library

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.

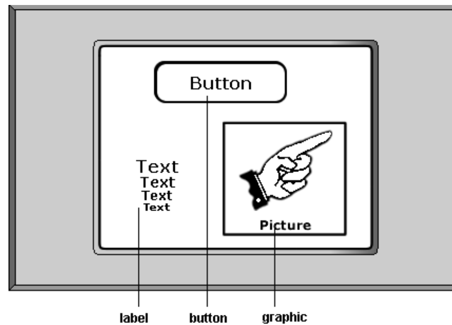


figure 1: Elements of the Tiger Graphic Library

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.

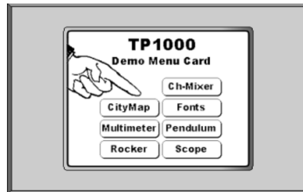


figure 2 Menu

figure 3: Graphical user interface

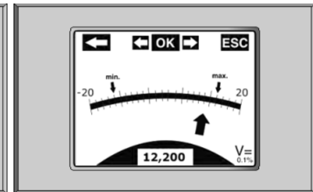


figure 4 Changing information

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 global variables 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. Save the bitmaps and texts for the elements in the flash memory
8. Copy the configuration file *TigerGraphicLibraryConf.INC* in your project directory ans apply the used fonts.
9. Create the elements and place them in windows and save the identifiers for those elements to work with later in the program
10. Display the window with its elements on the LCD
11. Watch the touched elements and occasionally update the element contents or change the window

All code examples are written for the TP1000 hardware.

Basic Program Structure for Applications

For applications with several windows there are normally a lot of elements to be created and handled. This causes initializations for each window and an execution loop for all the touch fields in the actually shown window. You will need to produce so many code lines that a clear program structure will help you to hold the overview. We propose a structure of initialization and execution subroutines for each window and an own administration subroutine which handles the window changing.

basic tasks	typical subroutine names	functionality	types of typically called tgl subroutines
Initialization	wInitWindowX	Fill windows with elements	Create Place Set Attributes Init
Execution	wExecWindowX	Watch touches Update elements Hide elements Occ. watch other inputs	Get Keycode Get Slider Value Get Switch State Show Text Update Gauge Show Graph Update Scope
Administration	wAdministrateWin dows	Change current window on lcd	Show Window (Exec Window)

Initialization means creating and placing of elements in one window. Additionally attributes of these elements like texts, bitmaps, inversion states, blink modes etc. can be set here. Fonts and elements which should be placed in several windows, we propose to initialize in an own subroutine *wInitGeneral*.

Execution means watching the touch areas of buttons, switches and sliders. Normally this is done in a loop starting with requests for all interactive elements and react as needed. Changes of internal variables in the application caused by analog measurands or digital signals which stand in relation to the actually shown window are handled here, too. This could be e.g. the update of gauges or the changing of element attributes like texts, bitmaps or inversion states.

The window changing we propose to handle in an administration subroutine. This subroutine represents the root level of the subroutines for the graphical user interface. From here all the execution subroutines will be called. Each window

changing leads back to this central subroutine. This subroutine is composed of the determination of the first window, one single call of the bTglShowWindow subroutine and a selection of the execution subroutine the actually shown window.

This leads to a modular program structure in which windows can easyly be added and modified without need of knowing the whole program code (see pseudo code below for code structure). The subroutines with *tgl* as part in their names are part of the Tiger Graphic Library. The subroutines without *tgl* in their names must be written by the programmer.

pseudo code example:

```
  word wlElementId
  call wInitGeneral( wlElementId.)
  call wInitWindowA( wlElementId )
  call wInitWindowB( wlElementId )
  ...
  call AdministrateWindows()
end

sub AdministrateWindows()
  word wlWindowId
  wlWindowId = WindowId1
  for ever=0 to 0 step 0
    call bTglShowWindow( wlWindowId )
    switch WindowId
    case WID_1:
      call wExecWindowA( wlWindowId )
    case WID_2:
      call wExecWindowB( wlWindowId )
    ...
    endswitch
  next
end

sub wInitGeneral( var word wpvElementId )
  call bTglCreateFont()
  ...
  call bTglCreateElement(()
  ...
  call bTglSetAttribute()
  ...
end
```

```
sub wInitWindowA( var word wpvElementId )
  call bTglCreateElementWnd()
  ...
  call bTglPlaceElementInWindow()
  ...
  call bTglSetAttribute()
  ...
end
sub wExecWindowA( &
  var word wpvWindowId )
  for ever=0 to 0 step 0
    call bTglGetKeycode()
    switch Keycode
    case KEY_1:
    case KEY_2:
    ...
    endswitch
    call wTglGetTouchedElement()
    ...
  next
end

sub wInitWindowB( var word wpvElementId )
  call bTglCreateElementWnd()
  ...
  call bTglPlaceElementInWindow()
  ...
  call bTglSetAttribute()
  ...
end
sub wExecWindowB( var word wpvWindowId )
  for ever=0 to 0 step 0
    call bTglGetKeycode()
    switch Keycode
    case KEY_1:
    case KEY_2:
    ...
    endswitch
    call wTglGetTouchedElement()
    ...
  next
end
```

Please mind the passed parameters for the initialization and execution subroutines which the programmer has to write for each window. In our structure concept we propose to increment the identifier for the elements after each creation.

This way new elements can be added at every time without taking care of double identifiers.

The initialization subroutines expect a free identifier to pass it to the next tgl creation subroutine. The returned identifier which has been incremented after each creation is the next free identifier for further elements in the next windows.

The execution subroutines get passed the identifier of the actually shown window. When the window should be changed just change this identifier and return this value to the administration subroutine.

The administration subroutine is the third basic subroutine in our structure for programs using the Tiger Graphic Library the programmer has to write. This subroutine handles the current window, shows it on the LCD and calls the corresponding execution subroutine.

For a simple example code using this structure see the files in *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_3WindowChanging*.

In the directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION* you will find more code examples focusing different tasks of programming with the Tiger Graphic Library with increasing complexity. These are elucidated in the next chapters.

For a template for your next applications see *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_TEMPLATE*.

Show a Window with Elements on LCD

This is the first of some example programs which lead to a general program structure for applications using the Tiger Graphic Library. This example demonstrates how to create elements, place them in a window show the elements on lcd and activate touch fields. You will find this example in *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_1CreateElements*

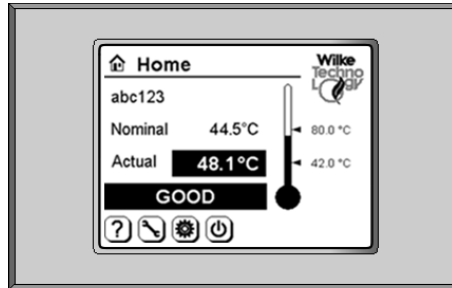


figure 6: Home window of an application

We start having a look in the file *APPLICATION.TIG*. Before we explain the use uf the subroutines there are some basic steps to do which are essential for the use of the Tiger Graphic Library.

The first 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 second step please determine the numbers for the identifiers of the windows and fonts in which the elements should be shown. Start with number 0 for the first window and for the first font. You will need these identifiers when you place elements in or show it on the LCD. The definitions will substitute the numbers in the code for an easier readability.

```
'' window identifiers
#define WID_Home                0
'' font identifiers
#define FID_Nominal             0
#define FID_Actual              1
#define FID_State               2
#define FID_ProductCode         3
```

In step 3 the file *APPLICATION_gui.inc* will be included which contains all subroutines for the graphical user interface. For this example this file contains the subroutine for the creation and placing of the elements.

```
#include APPLICATION_gui.inc
```

In step 4 the local variables are declared. The byte variable *blReturn* is for the return value of the subroutines of the Tiger Graphic Library. This variable reports the success of the execution of the last tgl subroutine. In case of a returned value unequals zero the table of error codes in the end of this manual will help you to identify the invalid parameter. The word variable *wlElementId* is the incremental identifier for the elements. This variable will start with zero and will be incremented after the creation of each element. This grants avoiding double identifiers for the multitude of elements which would lead to errors.

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

In step 5 the BASIC-Tiger™ ports will be set and the device drivers for the touch panel and the LCD will be installed. The included file TGL_DEVICE_DRIVERS_TP1000.INC is for installing the device drivers of a TP1000 board. For details see the program code of the files *TGL_DEVICE_DRIVERS_TP1000.INC TigerGraphicLibraryDefs.INC* and *TigerGraphicLibraryConf.INC* in the directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\TigerGraphicLibrary* and the data sheet for the TP1000.

```
#include TGL_DEVICE_DRIVERS_TP1000.INC
```

In step 6 the Tiger Graphic Library is initialized. This is done before calling any other subroutine of the Tiger Graphic Library. The subroutine *bTglInit* will initialize all internal variables and start an internal task. After these steps all the subroutines of the Tiger Graphic Library may be called. Just mind the order of creating, placing before showing, getting and setting.

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 determinating 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.

As written above we initialize the incremental identifier for the elements with zero. With the incrementation after each creation of an element we are able to create 2000 elements per default. The maximum number of created elements can be determined in the configuration file *TigerGraphicLibraryConf.INC.*

The subroutine *wInitHome* will fill the window named home with elements. This subroutine must be written by the programmer. We propose to write one subroutine for each window. In the example code you will identify subroutines which must be written by the programmer by a missing tgl in the name. We propose to pass the constant identifier of the window, the incremental identifier and the flag for the tgl error code.

```
call bTglInit( blReturn ) ' initialize internal variables and start internal
task
wlElementId = 0
call wInitHome( WID_Home, wlElementId, blReturn )
```

The subroutine *wInitHome* is placed in the file *APPLICATION_gui.inc* we have already included in this program example. We propose to pool files which belongs together in include files. This way you will get a modular program code which can be easily recycled and included for further applications.

The first lines (step 6i) in *APPLICATION_gui.inc* describe the seizure of the lcd we will use in our example application. We do this by dividing the graphical area in squares. Then we define aliases for the left top edges of these squares. This way we reduce the number of positions from 320x240 pixels to now 8x6 units. You need not do this in your applications but this compartmentation will help you to place your elements in clear lines and grants a good compromise of element size and number of lines. For an easier use of this 8x6 LCD seizure there are definitions already implemented in the library. You are certainly free in using this seizure with the library definitions or to create your own seizure.

```
' LCD_SEIZURE LANDSCAPE
' =====================
'             32      73     114     155     196     237     278     319
'       0       41      82      123     164     205     246     287
'       X0      X1      X2      X3      X4      X5      X6      X7
'   1 Y0+------++------++------++------++------++------++------++------+Y0
'       |      ||                                      ||              |
'       | ICON ||TITLE                                 ||              |
'       |      ||                                      ||              |
'  33   +------++------++------++------++------++------++     LOGO     +
'  42 Y1+------++------++------++------++------++------++              +Y1
'       |                                              ||              |
'       |                                              ||              |
'       |                                              ||              |
'  74   +------++------++------++------++------++------++------++------+
'  83 Y2+------++------++------++------++------++------++------++------+Y2
'       |                                                              |
'       |                                                              |
'       |                                                              |
' 115   +------++------++------++------++------++------++------++------+
' 124 Y3+------++------++------++------++------++------++------++------+Y3
'       |                                                              |
'       |                                                              |
'       |                                                              |
' 156   +------++------++------++------++------++------++------++------+
' 165 Y4+------++------++------++------++------++------++------++------+Y4
'       |                                                              |
'       |                                                              |
'       |                                                              |
' 197   +------++------++------++------++------++------++------++------+
' 206 Y5+------++------++------++------++------++------++------++------+Y5
'       |      ||      ||      ||      ||      ||      ||      ||      |
'       |BUTTON||BUTTON||BUTTON||BUTTON||BUTTON||BUTTON||BUTTON||BUTTON|
'       |      ||      ||      ||      ||      ||      ||      ||      |
' 238   +------++------++------++------++------++------++------++------+
'       X0      X1      X2      X3      X4      X5      X6      X7
'       0       41      82      123     164     205     246     287
'             32      73     114     155     196     237     278     319
'
```

```
' LCD_SEIZURE PORTRAIT
' ====================
'             33      74     115     156     197     238
'       1       42      83      124     165     206
'       X0      X1      X2      X3      X4      X5
'   0 Y0+------++------++------++------++------++------+Y0 0
'       |      ||                      ||              |
'       | ICON ||                      ||              |
'       |      ||                      ||              |
'  32   +------++ TITLE                ++     LOGO     +   32
'  41 Y1+------++                      ++              +Y1 41
'       |                              ||              |
'       |                              ||              |
'       |                              ||              |
'  73   +------++------++------++------++------++------+   73
'  82 Y2+------++------++------++------++------++------+Y2 82
'       |                                              |
'       |                                              |
'       |                                              |
' 114   +------++------++------++------++------++------+   114
' 123 Y3+------++------++------++------++------++------+Y3 123
'       |                                              |
'       |                                              |
'       |                                              |
' 155   +------++------++------++------++------++------+   155
' 164 Y4+------++------++------++------++------++------+Y4 164
'       |                                              |
'       |                                              |
'       |                                              |
' 196   +------++------++------++------++------++------+   196
' 205 Y5+------++------++------++------++------++------+Y5 205
'       |                                              |
'       |                                              |
'       |                                              |
' 237   +------++------++------++------++------++------+   237
' 246 Y6+------++------++------++------++------++------+Y6 246
'       |                                              |
'       |                                              |
'       |                                              |
' 288   +------++------++------++------++------++------+   288
' 205 Y7+------++------++------++------++------++------+Y7 205
'       |      ||      ||      ||      ||      ||      |
'       |BUTTON||BUTTON||BUTTON||BUTTON||BUTTON||BUTTON|
'       |      ||      ||      ||      ||      ||      |
' 319   +------++------++------++------++------++------+   219
'       X0      X1      X2      X3      X4      X5
'       1       42      83      124     165     206
'             33      74     115     156     197     238
'
#define TGL_H           33 ' height
#define TGL_W           33 ' width
#define TGL_D            8 ' column resp. row distance
#ifdef TGL_LANDSCAPE
#define TGL_X            0
#define TGL_Y            1
#else ' TGL_PORTRAIT
#define TGL_X            1
#define TGL_Y            0
#endif
```

```
#define TGL_H1          TGL_H
#define TGL_H2   (2*TGL_H+1*TGL_D)
#define TGL_H3   (3*TGL_H+2*TGL_D)
#define TGL_H4   (4*TGL_H+3*TGL_D)
#define TGL_H5   (5*TGL_H+4*TGL_D)
#define TGL_H6   (6*TGL_H+5*TGL_D)
#define TGL_H7   (7*TGL_H+6*TGL_D)
#define TGL_H8   (8*TGL_H+7*TGL_D)
#define TGL_W1          TGL_W
#define TGL_W2   (2*TGL_W+1*TGL_D)
#define TGL_W3   (3*TGL_W+2*TGL_D)
#define TGL_W4   (4*TGL_W+3*TGL_D)
#define TGL_W5   (5*TGL_W+4*TGL_D)
#define TGL_W6   (6*TGL_W+5*TGL_D)
#define TGL_W7   (7*TGL_W+6*TGL_D)
#define TGL_W8   (8*TGL_W+7*TGL_D)
#define TGL_X0          TGL_X
#define TGL_X1  (TGL_X0+TGL_H+TGL_D)
#define TGL_X2  (TGL_X1+TGL_H+TGL_D)
#define TGL_X3  (TGL_X2+TGL_H+TGL_D)
#define TGL_X4  (TGL_X3+TGL_H+TGL_D)
#define TGL_X5  (TGL_X4+TGL_H+TGL_D)
#define TGL_X6  (TGL_X5+TGL_H+TGL_D)
#define TGL_X7  (TGL_X6+TGL_H+TGL_D)
#define TGL_Y0          TGL_Y
#define TGL_Y1  (TGL_Y0+TGL_H+TGL_D)
#define TGL_Y2  (TGL_Y1+TGL_H+TGL_D)
#define TGL_Y3  (TGL_Y2+TGL_H+TGL_D)
#define TGL_Y4  (TGL_Y3+TGL_H+TGL_D)
#define TGL_Y5  (TGL_Y4+TGL_H+TGL_D)
#define TGL_Y6  (TGL_Y5+TGL_H+TGL_D)
#define TGL_Y7  (TGL_Y6+TGL_H+TGL_D)
```

Before we create the elements the fonts must be created (step 6ii) because they are needed for the text elements. Please see the api below for the specification of the subroutines. Please mind the checking of the return values after each calling of an tgl subroutine. The initialization subroutine will be aborded immediately in case of faillure and return the error code of the last failed subroutine.

Mind that the font bitmaps for the font must be applied in the configuration file *TigerGraphicLibraryConf.INC*. This file is included internally. You find the default configuration file in
%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\TigerGraphicLibrary.
Mind to use copys of this file and never to change the original file. Otherwise most of the example programs may produce errors! Activating too many fonts which are never used in a program would waste a lot flash memory. Check which fonts you really need! For more information see sections *Graphic Fonts* in chapter *Components and its Handling* and *Choosing the Graphic Fonts* in chapter *Text Graphics*

Please have a look on the font parameters for letters and digits. For the product code string it is chosen a proportional character spacing of a default spacing width (*"prop", SPACING_CHAR_DEFAULT*). For the displaying of the value of the nominal temperature a constant spacing type is chosen. All fonts in the Tiger Graphic Library can be used proportional or constant. For the optimizing of the spacing for digits please pass *SPACING_CHAR_DEFAULT_DIGIT*.

```
call bTglCreateFontParams( &
FID_ProductCode, &              ' identifier of font
"Valencia", 14, "normal", &     ' name, size, type of font
"left", "center", &             ' alignment horizontal, vertical
"prop", 0, &                    ' spacing type, blank
SPACING_CHAR_DEFAULT,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

call bTglCreateFontParams( &
FID_Nominal, &                  ' identifier of font
"Valencia", 14, "normal", &     ' 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
```

In step 6iii we propose to work with masks. A mask is a monochrome bitmap in the same size as the LCD (here 320x240). The mask contents all graphics which are constant and are not modified during the program. Using masks simplifys the realization of a good looking application, reduces the number of needed elements and speeds up the window changing. The bitmap can easily be made with any external graphic program. Save the bitmap in the project directory. For a better overwiew over the project files the Tiger Graphic Library includes per default the files in the directory *bitmaps*. If the bitmap is saved at other places, just pass the path to the data instruction. To be available in the program it must be poked in the flash by the instruction *data filter*. The datalabel which is placed just before the data instruction saves the flash address which must be passed to the tgl subroutines. Datalabels are always global even though they are declared locally. The called initialation subroutine creates internally an element of the type *graphic* which is copied in an ORed mode on the screen. Initialization subroutines of the Tiger Graphic Library increment the identifier for elements internally.

```
datalabel dlMaskHome
dlMaskHome::
data filter "MaskHome.bmp", "GRAPHFLT", 0     ' WxH=320x240  BmpWidth=320
call wTglInitMaskWnd( &
dlMaskHome, &                    ' flash address of bitmap
wpWindowId, wpvElementId, &      ' identifier of window, element
bpvReturn )                      ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif
```

In step 6iv four bitmap buttons will be created and placed. As these elements are similar, we do it in a loop. The subroutines with a *Wnd* in their name combine the *create* and *place* subroutines. This reduces the calls in your program. For the next element the identifier for elements will be incremented internally.

Mind that Tiger-BASIC handles only with bitmaps which have a format width of a multiple of eight. This means, that the last 1 to 6 pixel columns in the bitmap could be padding pixels. The elements in the Tiger Graphic Library can have any width. The number of bytes the bitmap needs in the flash memory can be calculated by *height*width/8*. You can save the bitmaps in the directory *bitmaps* of your project. In this program example we use bitmaps which are part of the Tiger Graphic Library. See *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\TigerGraphicLibrary\Bitmaps* for more bitmaps.

```
  call bTglCreateButtonWnd( &
  TGL_W1, TGL_H1, &   ' width, height of element
  llAddr, 40, &                ' address, format width of bitmap
  TGL_KEY_ATTR_AUTOREPEAT_OFF, &' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, & ' identifier of element, window
  wlX, TGL_Y5, &               ' x, y coordinate on lcd
  blKeycode, &                 ' keycode
  bpvReturn )                  ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
```

In step 6v text elements will be created. Mind the passed font identifier. If the font does not exist yet the subroutine will return an error. Further text elements are text buttons and listboxes. Some attributes of elements can not be determined by parameters in the create and place subroutines. In this example the label for the actual temperature should be displayed inverted. See the api below in this manual for more attributes which can be set.

```
call bTglCreateLabelWnd( &
80, TGL_H1, &                     ' width, height of element
"48.7", &                         ' text
FID_Actual, 0, &                  ' font identifier, frame thickness
wpvElementId, wpWindowId, &       ' identifier of element, window
TGL_X2, TGL_Y3, &                 ' x, y coordinate on lcd
bpvReturn )                       ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif
call bTglSetAttribute( wpvElementId, wpWindowId, &
TGL_ATTR_INV_STATE, TGL_INVERTED, bpvReturn )
if bpvReturn <> TGL_MSG_OK then
  return
endif
wpvElementId = wpvElementId + 1
```

In step 6iv a gauge is created and placed. In this example we choosed a rectangular gauge which fits exactly in the thermometer in the mask.

```
call bTglCreateGaugeWnd( &
7, 123, &                                 ' width, height of element
0,1023, &                                 ' limits of values
TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM,&     ' type, location of base
0, &                                      ' frame thickness
wpvElementId, wpWindowId, &               ' identifier of the element, window
219, 46, &                                ' 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
wpvElementId = wpvElementId + 1
```

In step 7 (.tig file) we display the elements 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*. The window itself need not be created. All windows are existing already with or without placed elements. A shown empty window would lead to an empty LCD and an inactive touch area.

```
call bTglShowWindow( WID_Home, blReturn )
```

Sample program:

```
'----------------------------------------------------------------------------
' TGL_APPLICATION_1CreateElements/APPLICATION.TIG
' TGL_APPLICATION_1CreateElements/APPLICATION_gui.INC
'----------------------------------------------------------------------------
'****************************************************************************
```

```
'       Step 1: Include the Tiger Graphic Library files
'****************************************************************************
#include TigerGraphicLibrary.INC

'****************************************************************************
'       Step 2: Identifiers for the Tiger Graphic Library
'****************************************************************************
'' window identifiers
#define WID_Home                0
'' font identifiers
#define FID_Nominal             0
#define FID_Actual              1
#define FID_State               2
#define FID_ProductCode         3

'****************************************************************************
'       Step 3: Include user specific application files
'****************************************************************************
#include APPLICATION_gui.inc

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

  '**************************************************************************
  '     Step 4: Declaration of local variables
  '**************************************************************************
  byte blReturn         ' return value for TGL subroutines
  word wlElementId      ' incremental identifier for elements

  '**************************************************************************
  '     Step 5: Installation of device drivers for touch panel and LCD
  '**************************************************************************
  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '**************************************************************************
  '     Step 6: Initializations
  '                i) Tiger Graphic Library
  '               ii) fonts, elements and windows
  '**************************************************************************
  call bTglInit( blReturn )
  wlElementId = 0
  call wInitHome( WID_Home, wlElementId, blReturn )

  '**************************************************************************
  '     Step 7: Displaying the window on the LCD
  '**************************************************************************
  call bTglShowWindow( WID_Home, blReturn )
end

'----------------------------------------------------------------------------
' wInitHome:
'----------------------------------------------------------------------------
' Create and place all fonts and elements for the home window
' Check tgl return value and abort subroutine in case of failure
' Return next free identifier for further elements to be created
'----------------------------------------------------------------------------
' 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 wInitHome( word wpWindowId; var word wpvElementId; var byte bpvReturn )
  long llAddr
  word wlX
  byte blButton, blKeycode

  '*************************************************************************
  '     Step 8ii: fonts
  '*************************************************************************
  call bTglCreateFontParams( &
  FID_ProductCode, &              ' identifier of font
  "Valencia", 14, "normal", &     ' name, size, type of font
  "left", "center", &             ' alignment horizontal, vertical
  "prop", 0, &                    ' spacing type, blank
  SPACING_CHAR_DEFAULT,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

  call bTglCreateFontParams( &
  FID_Nominal, &                  ' identifier of font
  "Valencia", 14, "normal", &     ' 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

  call bTglCreateFontParams( &
  FID_Actual, &                   ' identifier of font
  "Valencia", 14, "normal", &     ' 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

  call bTglCreateFontParams( &
  FID_State, &                    ' 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
  bpvReturn )                     ' return code (0: OK exit  >0: error exit)
```

```
if bpvReturn <> TGL_MSG_OK then
  return
endif

'**************************************************************************
'     step 8iii: background mask
'**************************************************************************
datalabel dlMaskHome
dlMaskHome::
data filter "MaskHome.bmp", "GRAPHFLT", 0      ' WxH=320x240  BmpWidth=320
call wTglInitMaskWnd( &
dlMaskHome, &                   ' flash address of bitmap
wpWindowId, wpvElementId, &     ' identifier of window, element
bpvReturn )                     ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif

'**************************************************************************
'     step 8iv: bitmap elements
'**************************************************************************
' menu bar
' source path to bitmaps (automatical included by the TGL)
' %ProgramData%\Wilke Technology\Tiger Basic 5.4\
'   Libraries\TigerGraphicLibrary\Bitmaps\Buttons\Style3
datalabel dlMenuButtons
dlMenuButtons::
data filter "TGL_BUTTON_STYLE_3_iconSettings_1.bmp",  "GRAPHFLT", 0
data filter "TGL_BUTTON_STYLE_3_iconService_1.bmp",   "GRAPHFLT", 0
data filter "TGL_BUTTON_STYLE_3_iconHelp_1.bmp",      "GRAPHFLT", 0
data filter "TGL_BUTTON_STYLE_3_iconOnOff_1.bmp",     "GRAPHFLT", 0
' WxH=33x33  BmpWidth=40
#define BMP_LEN_MenuButton    (33 * 40/8) ' = (H * BmpWidth/8)
#define KEY_Settings          0
#define KEY_Calibration       1
#define KEY_Help              2
#define KEY_OnOff             3
#define NUM_MenuButtons       4
for blButton = 0 to NUM_MenuButtons-1
  llAddr = dlMenuButtons + blButton*BMP_LEN_MenuButton
  wlX    = blButton*(TGL_W+TGL_D)
  switchi blButton
  case 0:
    blKeycode   = KEY_Help
  case 1:
    blKeycode   = KEY_Calibration
  case 2:
    blKeycode   = KEY_Settings
  case 3:
    blKeycode   = KEY_OnOff
  endswitch
  call bTglCreateButtonWnd( &
  TGL_W1, TGL_H1, &   ' width, height of element
  llAddr, 40, &               ' address, format width of bitmap
  TGL_KEY_ATTR_AUTOREPEAT_OFF, &' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, & ' identifier of element, window
  wlX, TGL_Y5, &              ' x, y coordinate on lcd
  blKeycode, &                ' keycode
  bpvReturn )                 ' return code (0: OK exit  >0: error exit)
```

```
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    wpvElementId = wpvElementId + 1
  next

  '**************************************************************************
  '      step 8v: text elements
  '**************************************************************************
  call bTglCreateLabelWnd( &
  TGL_W5, TGL_H1, &              ' width, height of element
  "ABC123", &                    ' text
  FID_ProductCode, 0, &          ' font identifier, frame thickness
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X0, TGL_Y1, &              ' 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 bTglCreateLabelWnd( &
  80, TGL_H1, &                  ' width, height of element
  "50.0", &                      ' text
  FID_Nominal, 0, &              ' font identifier, frame thickness
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X2, TGL_Y2, &              ' 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 bTglCreateLabelWnd( &
  80, TGL_H1, &                  ' width, height of element
  "48.7", &                      ' text
  FID_Actual, 0, &               ' font identifier, frame thickness
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X2, TGL_Y3, &              ' x, y coordinate on lcd
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  call bTglSetAttribute( wpvElementId, wpWindowId, &
  TGL_ATTR_INV_STATE, TGL_INVERTED, bpvReturn )
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1

  call bTglCreateLabelWnd( &
  TGL_W5, TGL_H1, &              ' width, height of element
  "GOOD", &                      ' text
  FID_State, 0, &                ' font identifier, frame thickness
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X0, TGL_Y4, &              ' x, y coordinate on lcd
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
```

```
  endif
  call bTglSetAttribute( wpvElementId, wpWindowId, &
  TGL_ATTR_INV_STATE, TGL_INVERTED, bpvReturn )
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1

  '*************************************************************************
  '     step 8vi: gauges
  '*************************************************************************
  call bTglCreateGaugeWnd( &
  7, 123, &                              ' width, height of element
  0,1023, &                              ' limits of values
  TGL_GA_TYPE_BAR, TGL_GA_BASE_BOTTOM,& ' type, location of base
  0, &                                   ' frame thickness
  wpvElementId, wpWindowId, &            ' identifier of the element, window
  219, 46, &                             ' 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
  wpvElementId = wpvElementId + 1
end
```

Realize a Graphical User Interface

In this chapter it is demonstrated step by step how to realize a graphical user interface (gui) as it is e. g. needed for the program settings. We suggest copying the code fragments directly into your future projects. You will find this example in *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_2UserInterface*

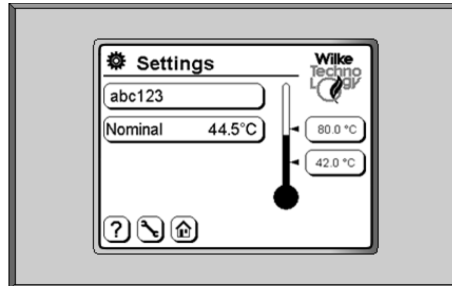


figure 7: Settings window of application

In step 1 the Tiger Graphic Library will be included.

```
#include TigerGraphicLibrary.INC
```

In step 2 the Idenifiers for fonts and windows will be defined. Additionally global variables are declared for identifiers of those elements we will need to pass to subroutines in the further program code.

```
'' window identifiers
#define WID_Settings             0
#define WID_Keyboard             1
#define WID_Keyboard_shifted     2 ' internally initialized window
#define WID_DigitBlock           3
'' font identifiers
#define FID_ProductCode          0
#define FID_Nominal              1
#define FID_Limit                2
#define FID_Keyboard             3
#define FID_DigitBlock           FID_Keyboard
'' element identifiers
word wgEID_Keyboard, wgEID_DigitBlock
```

In step 3 some global variables are defined which are central for this application.

```
#define MAX_LEN_PRODUCTCODE     10h
string sgProductCode$(MAX_LEN_PRODUCTCODE)
real rgTemperatureNom, rgTemperatureMin, rgTemperatureMax
```

In step 4 the file *APPLICATION_gui.inc* is included which contains all subroutines for the graphical user interface. These are initialization subroutines where the windows will be filled with elements and execution subroutines for the handling of the buttons in the windows.

```
#include APPLICATION_gui.inc
```

After the declaration of some local variables in step 5 and the installation of the device drivers in step 5 the initializations are done in step 7. For acceleration we set a higher task prio for the initializing task *main*.

Elements and fonts which are needed in more than one window we propose to create in an own subroutine *wInitGeneral*.

The windows for the keyboards can be initialized by the tgl subroutine *bTglInitKeyboard*. You will get the whole keyboard functionality with one call only. The identifiers of the elements 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 *WID_keyboard* must be parsed into a variable before it can be passed to the initialization subroutine. If you debug the return value of *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.

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

#include TGL_DEVICE_DRIVERS_TP1000.INC

set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call bTglInit( blReturn )
wlElementId = 0
call wInitGeneral( wlElementId, blReturn )
call wInitSettings( WID_Settings, wlElementId, blReturn )
wlWindowId = WID_Keyboard
call bTglInitKeyboard( &
TGL_KEYB1_STYLE_ENG, FID_Keyboard, & ' keyboard style, font
wlElementId, wlWindowId, &    ' identifier of element, window
wgEID_Keyboard, &             ' identifier for keyboard view
blReturn )                    ' return code (0: OK exit  >0: error exit)
wlWindowId = WID_DigitBlock
call bTglInitKeyboard( &
TGL_DIGSTYLE_1, FID_DigitBlock, & ' keyboard style, font
wlElementId, wlWindowId, &    ' identifier of element, window
wgEID_DigitBlock, &           ' identifier for keyboard view
blReturn )                    ' return code (0: OK exit  >0: error exit)
```

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.

```
wgEID_ProductCode = wpvElementId
call bTglCreateTextButtonWnd( &
TGL_W5a, TGL_H1, &            ' width, height of element
"ABC123", &                   ' text
FID_ProductCode, 0, &         ' font identifier, frame thickness
TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
wpvElementId, wpWindowId, &   ' identifier of element, window
TGL_X0a, TGL_Y1, &            ' x, y coordinate on lcd
KEY_ProductCode, &            ' keycode
bpvReturn )                   ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif
wpvElementId = wpvElementId + 1
```

Mind the “a” in the parameter definitions for size and placement. We enhanced our internal definitions for the LCD seizure to reach some more standard positions.

```
#define TGL_H           33 ' height
#define TGL_W           33 ' width
#define TGL_a            3 ' cell margin
#define TGL_b            2 ' margin offset unpressed button
#define TGL_D            8 ' column resp. row distance
#ifdef TGL_LANDSCAPE
#define TGL_X            0
#define TGL_Y            1
#else ' TGL_PORTRAIT
#define TGL_X            1
#define TGL_Y            0
#endif

#define TGL_H1          TGL_H
#define TGL_H2   (2*TGL_H+1*TGL_D)
#define TGL_H3   (3*TGL_H+2*TGL_D)
#define TGL_H4   (4*TGL_H+3*TGL_D)
#define TGL_H5   (5*TGL_H+4*TGL_D)
#define TGL_H6   (6*TGL_H+5*TGL_D)
#define TGL_H7   (7*TGL_H+6*TGL_D)
#define TGL_H8   (8*TGL_H+7*TGL_D)
#define TGL_W1          TGL_W
#define TGL_W2   (2*TGL_W+1*TGL_D)
#define TGL_W3   (3*TGL_W+2*TGL_D)
#define TGL_W4   (4*TGL_W+3*TGL_D)
#define TGL_W5   (5*TGL_W+4*TGL_D)
#define TGL_W6   (6*TGL_W+5*TGL_D)
#define TGL_W7   (7*TGL_W+6*TGL_D)
#define TGL_W8   (8*TGL_W+7*TGL_D)
#define TGL_X0          TGL_X
#define TGL_X1  (TGL_X0+TGL_H+TGL_D)
#define TGL_X2  (TGL_X1+TGL_H+TGL_D)
#define TGL_X3  (TGL_X2+TGL_H+TGL_D)
#define TGL_X4  (TGL_X3+TGL_H+TGL_D)
#define TGL_X5  (TGL_X4+TGL_H+TGL_D)
#define TGL_X6  (TGL_X5+TGL_H+TGL_D)
#define TGL_X7  (TGL_X6+TGL_H+TGL_D)
#define TGL_Y0          TGL_Y
#define TGL_Y1  (TGL_Y0+TGL_H+TGL_D)
#define TGL_Y2  (TGL_Y1+TGL_H+TGL_D)
#define TGL_Y3  (TGL_Y2+TGL_H+TGL_D)
#define TGL_Y4  (TGL_Y3+TGL_H+TGL_D)
#define TGL_Y5  (TGL_Y4+TGL_H+TGL_D)
#define TGL_Y6  (TGL_Y5+TGL_H+TGL_D)
#define TGL_Y7  (TGL_Y6+TGL_H+TGL_D)

#define TGL_H1a (TGL_H1-(2*TGL_a))
#define TGL_H2a (TGL_H2-(2*TGL_a))
#define TGL_H3a (TGL_H3-(2*TGL_a))
#define TGL_H4a (TGL_H4-(2*TGL_a))
#define TGL_H5a (TGL_H5-(2*TGL_a))
#define TGL_H6a (TGL_H6-(2*TGL_a))
#define TGL_H7a (TGL_H7-(2*TGL_a))
#define TGL_H8a (TGL_H8-(2*TGL_a))
#define TGL_W1a (TGL_W1-(2*TGL_a))
#define TGL_W2a (TGL_W2-(2*TGL_a))
#define TGL_W3a (TGL_W3-(2*TGL_a))
#define TGL_W4a (TGL_W4-(2*TGL_a))
#define TGL_W5a (TGL_W5-(2*TGL_a))
```

```
#define TGL_W6a (TGL_W6-(2*TGL_a))
#define TGL_W7a (TGL_W7-(2*TGL_a))
#define TGL_W8a (TGL_W8-(2*TGL_a))
#define TGL_X0a (TGL_X0+TGL_a)
#define TGL_X1a (TGL_X1+TGL_a)
#define TGL_X2a (TGL_X2+TGL_a)
#define TGL_X3a (TGL_X3+TGL_a)
#define TGL_X4a (TGL_X4+TGL_a)
#define TGL_X5a (TGL_X5+TGL_a)
#define TGL_X6a (TGL_X6+TGL_a)
#define TGL_X7a (TGL_X7+TGL_a)
#define TGL_Y0a (TGL_Y0+TGL_a)
#define TGL_Y1a (TGL_Y1+TGL_a)
#define TGL_Y2a (TGL_Y2+TGL_a)
#define TGL_Y3a (TGL_Y3+TGL_a)
#define TGL_Y4a (TGL_Y4+TGL_a)
#define TGL_Y5a (TGL_Y5+TGL_a)
#define TGL_Y6a (TGL_Y6+TGL_a)
#define TGL_Y7a (TGL_Y7+TGL_a)
#define TGL_H1b (TGL_H1a-TGL_b)
#define TGL_H2b (TGL_H2a-TGL_b)
#define TGL_H3b (TGL_H3a-TGL_b)
#define TGL_H4b (TGL_H4a-TGL_b)
#define TGL_H5b (TGL_H5a-TGL_b)
#define TGL_H6b (TGL_H6a-TGL_b)
#define TGL_H7b (TGL_H7a-TGL_b)
#define TGL_H8b (TGL_H8a-TGL_b)
#define TGL_W1b (TGL_W1a-TGL_b)
#define TGL_W2b (TGL_W2a-TGL_b)
#define TGL_W3b (TGL_W3a-TGL_b)
#define TGL_W4b (TGL_W4a-TGL_b)
#define TGL_W5b (TGL_W5a-TGL_b)
#define TGL_W6b (TGL_W6a-TGL_b)
#define TGL_W7b (TGL_W7a-TGL_b)
#define TGL_W8b (TGL_W8a-TGL_b)
```

For the next steps see *APPLICATION_gui.INC.* Here we describe only the new ideas of this example. See the last chapter for more information.

In step 8i the variables for identifiers of some elements of this window are declared. The saving of these identifiers enables the modification of these elements later in the program.

```
word wgEID_ProductCode,    wgEID_TemperatureNom
word wgEID_TemperatureMin, wgEID_TemperatureMax
```

In step 8ii the keycodes for the touch elements of this window are determined.

```
#define KEY_ProductCode          0
#define KEY_TemperatureNom       1
#define KEY_TemperatureMin       2
#define KEY_TemperatureMax       3
```

In step 7iii text buttons are created and placed as touch elements. As the texts should be changed by user input the identifiers are saved before the creation.

```
  wgEID_ProductCode = wpvElementId
  call bTglCreateTextButtonWnd( &
  TGL_W5a, TGL_H1, &     ' width, height of element
  "ABC123", &                  ' text
  FID_ProductCode, 0, &        ' font identifier, frame thickness
  TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, &   ' identifier of element, window
  TGL_X0a, TGL_Y1, &           ' x, y coordinate on lcd
  KEY_ProductCode, &           ' keycode
  bpvReturn )                  ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
```

After the initializations in step 9 in the file *APPLICATION.TIG* the task priority is resetted for normal running. The window and the elements are shown on lcd and the pressing of the buttons is watched in an endless loop.

```
  set_task_prio main, 1 ' reset task priority for usual running
  wlWindowId = WID_Settings
  for ever = 0 to 0 step 0
    call bTglShowWindow( wlWindowId, blReturn )
    call wExecSettings(  wlWindowId, blReturn )
  next ' endless loop
```

In step 9iv the programs 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_ProductCode:
    ''…
    case KEY_TemperatureNom:
    ''…
    case KEY_TemperatureMin:
    ''…
    case KEY_TemperatureMax:
    ''…
    endswitch
```

In step 9v a keyboard will be shown on LCD for the user input. It is possible to show a greeting text in the keyboard view by filling the string for the user input *slText$*. All the functionality of getting the keyboard input is done by a single calling.

```
    slText$ = "Product Code"
    call sTglGetKeyboardInput( &
    WID_keyboard, wgEID_Keyboard, &' identifiers for keyboard
    10h, slText$, &      ' maximal user input length, user input
    bpvReturn )          ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
```

When the user finished his input, the input normally is checked. We do this in step 9vi. If there is a returned text, the text of the text button is replaced by the new text.

```
    if 0 < len(slText$) then
      sgProductCode$ = slText$
      call bTglSetText( wgEID_ProductCode, sgProductCode$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif
```

In this moment the text is not shown on lcd yet. The window with the keyboard is still shown. To change back to the settings window we end the subroutine after the switch case. Back in the endless loop in the task *main* we change again to the settings window with the new text in the touched element.

Sample program:

```
'----------------------------------------------------------------------------
' TGL_APPLICATION_2UserInterface/APPLICATION.TIG
' TGL_APPLICATION_2UserInterface/APPLICATION_gui.INC
'----------------------------------------------------------------------------
' required software:    Tiger Graphic Library V1.15
' required hardware:    TP-1000
'----------------------------------------------------------------------------

'****************************************************************************
'       Step 1: Include the files for the Tiger Graphic Library
'****************************************************************************
#include TigerGraphicLibrary.INC

'****************************************************************************
'       Step 2: Identifiers for the Tiger Graphic Library
'****************************************************************************
'' window identifiers
#define WID_Settings            0
#define WID_Keyboard            1
#define WID_Keyboard_shifted    2 ' placeholder for internally initialized
window
```

```
#define WID_DigitBlock          3
'' font identifiers
#define FID_ProductCode         0
#define FID_Nominal             1
#define FID_Limit               2
#define FID_Keyboard            3
#define FID_DigitBlock          FID_Keyboard
'' element identifiers
word wgEID_Keyboard, wgEID_DigitBlock

'****************************************************************************
'       Step 3: Application specific definitions and global variables
'****************************************************************************
#define MAX_LEN_PRODUCTCODE     10h
string sgProductCode$(MAX_LEN_PRODUCTCODE)
real rgTemperatureNom, rgTemperatureMin, rgTemperatureMax

'****************************************************************************
'       Step 5: Include user specific application files
'****************************************************************************
#include APPLICATION_gui.inc


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

  '**************************************************************************
  '     Step 4: Declaration of local variables
  '**************************************************************************
  byte blReturn          ' return value for TGL subroutines
  word wlElementId       ' incremental identifier for elements
  word wlWindowId        ' identifier for windows
  byte ever              ' endless loop

  '**************************************************************************
  '     Step 5: Installation of device drivers for touch panel and LCD
  '**************************************************************************
  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '**************************************************************************
  '     Step 6: Initializations
  '                i) Tiger Graphic Library
  '               ii) fonts, elements and windows
  '**************************************************************************
  set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
  call bTglInit( blReturn )
  wlElementId = 0
  call wInitGeneral( wlElementId, blReturn )
  call wInitSettings( WID_Settings, wlElementId, blReturn )
  wlWindowId = WID_Keyboard
  call bTglInitKeyboard( &
  TGL_KEYB1_STYLE_ENG, FID_Keyboard, & ' keyboard style, font
  wlElementId, wlWindowId, &     ' identifier of element, window
  wgEID_Keyboard, &              ' identifier for keyboard view
  blReturn )                     ' return code (0: OK exit  >0: error exit)
  wlWindowId = WID_DigitBlock
  call bTglInitKeyboard( &
```

```
  TGL_DIGSTYLE_1, FID_DigitBlock, & ' keyboard style, font
  wlElementId, wlWindowId, &     ' identifier of element, window
  wgEID_DigitBlock, &            ' identifier for keyboard view
  blReturn )                     ' return code (0: OK exit  >0: error exit)

  '**************************************************************************
  '      Step 7: Display the window on the LCD and watch the window buttons
  '**************************************************************************
  set_task_prio main, 1 ' reset task priority for usual running
  wlWindowId = WID_Settings
  for ever = 0 to 0 step 0
    call bTglShowWindow( wlWindowId, blReturn )
    call wExecSettings(  wlWindowId, blReturn )
  next ' endless loop
end

'----------------------------------------------------------------------------
' wInitGeneral:
'----------------------------------------------------------------------------
' Create fonts and general elements which are placed in several windows.
' Here: fonts only
'----------------------------------------------------------------------------
' 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( &
  FID_ProductCode, &             ' identifier of font
  "Valencia", 14, "normal", &    ' name, size, type of font
  "left", "center", &            ' alignment horizontal, vertical
  "prop", 0, &                   ' spacing type, blank
  SPACING_CHAR_DEFAULT,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

  call bTglCreateFontParams( &
  FID_Nominal, &                 ' identifier of font
  "Valencia", 14, "normal", &    ' 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

  call bTglCreateFontParams( &
  FID_Limit, &                   ' identifier of font
  "Valencia", 10, "normal", &    ' 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

  call bTglCreateFontParams( &
  FID_Keyboard, &                ' 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
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
end


'----------------------------------------------------------------------------
' wInitSettings:
'----------------------------------------------------------------------------
' 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)
'----------------------------------------------------------------------------
'****************************************************************************
'       Step 8i: element identifiers for this window
'****************************************************************************
word wgEID_ProductCode,    wgEID_TemperatureNom
word wgEID_TemperatureMin, wgEID_TemperatureMax
'****************************************************************************
'       Step 8ii: keycodes for this window
'****************************************************************************
#define KEY_ProductCode         0
#define KEY_TemperatureNom      1
#define KEY_TemperatureMin      2
#define KEY_TemperatureMax      3
sub wInitSettings( word wpWindowId; var word wpvElementId; &
  var byte bpvReturn )
  word wlElementId, wlY
  string slText$(10h)
  set_len$( slText$, 0 )

  '**************************************
  ' background mask
  '**************************************
  datalabel dlMaskSettings
  dlMaskSettings::
  data filter "MaskSettings.bmp", "GRAPHFLT", 0 ' WxH=320x240  BmpWidth=320
  call wTglInitMaskWnd( &
```

```
  dlMaskSettings, &              ' flash address of bitmap
  wpWindowId, wpvElementId, &    ' identifier of window, element
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif

  '************************************
  ' more elements
  '************************************

'****************************************************************************
  '     Step 8iii: save element identifier for this element

'****************************************************************************
  wgEID_ProductCode = wpvElementId
  call bTglCreateTextButtonWnd( &
  TGL_W5a, TGL_H1, &     ' width, height of element
  "ABC123", &                    ' text
  FID_ProductCode, 0, &          ' font identifier, frame thickness
  TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X0a, TGL_Y1, &             ' x, y coordinate on lcd
  KEY_ProductCode, &             ' keycode
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1

  wgEID_TemperatureNom = wpvElementId
  call bTglCreateTextButtonWnd( &
  80, TGL_H1, &                  ' width, height of element
  "50.0", &                      ' text
  FID_Nominal, 0, &              ' font identifier, frame thickness
  TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X2, TGL_Y2, &              ' x, y coordinate on lcd
  KEY_TemperatureNom, &          ' keycode
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1

  wgEID_TemperatureMax = wpvElementId
  call bTglCreateTextButtonWnd( &
  40, TGL_H1, &                  ' width, height of element
  "80.0", &                      ' text
  FID_Limit, 0, &                ' font identifier, frame thickness
  TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X6a, TGL_Y2, &             ' x, y coordinate on lcd
  KEY_TemperatureMax, &          ' keycode
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
```

```
  wgEID_TemperatureMin = wpvElementId
  call bTglCreateTextButtonWnd( &
  40, TGL_H1, &                   ' width, height of element
  "42.0", &                       ' text
  FID_Limit, 0, &                 ' font identifier, frame thickness
  TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
  wpvElementId, wpWindowId, &    ' identifier of element, window
  TGL_X6a, TGL_Y3, &             ' x, y coordinate on lcd
  KEY_TemperatureMin, &          ' keycode
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
end
'----------------------------------------------------------------------------
' wExecSettings:
'----------------------------------------------------------------------------
' 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 wExecSettings( var word wpvWindowId; var byte bpvReturn )
  byte blKeycode        ' button return code
  string slText$(10h)   ' user input
  set_len$( slText$, 0 )

  '**************************************************************************
  '     Step 8iv: get keyboard input
'**************************************************************************
  call bTglWaitKeycode( blKeycode )
  switchi blKeycode

  case KEY_ProductCode:
    '***********************************************************************
    '   Step 8v: get keyboard input
    '***********************************************************************
    slText$ = "Product Code"
    call sTglGetKeyboardInput( &
    WID_keyboard, wgEID_Keyboard, &' identifiers for keyboard
    10h, slText$, &      ' maximal user input length, user input
    bpvReturn )          ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    '***********************************************************************
    '   Step 8vi: check valid keyboard input and change text in element
    '***********************************************************************
    if 0 < len(slText$) then
      sgProductCode$ = slText$
      call bTglSetText( wgEID_ProductCode, sgProductCode$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
```

```
    endif

  case KEY_TemperatureNom:
    slText$ = "T nominal"
    call sTglGetKeyboardInput( &
    WID_DigitBlock, wgEID_DigitBlock, &' identifiers for keyboard
    7, slText$, &                ' maximal user input length, user input
    bpvReturn )          ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    if 0 < len(slText$) then
      rgTemperatureNom = val_real( slText$, "." )
      call bTglSetText( wgEID_TemperatureNom, slText$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif

  case KEY_TemperatureMin:
    slText$ = "T min"
    call sTglGetKeyboardInput( &
    WID_DigitBlock, wgEID_DigitBlock, &' identifiers for keyboard
    7, slText$, &                ' maximal user input length, user input
    bpvReturn )          ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    if 0 < len(slText$) then
      rgTemperatureMin = val_real( slText$, "." )
      call bTglSetText( wgEID_TemperatureMin, slText$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif

  case KEY_TemperatureMax:
    slText$ = "T max"
    call sTglGetKeyboardInput( &
    WID_DigitBlock, wgEID_DigitBlock, &' identifiers for keyboard
    7, slText$, &                ' maximal user input length, user input
    bpvReturn )          ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    if 0 < len(slText$) then
      rgTemperatureMax = val_real( slText$, "." )
      call bTglSetText( wgEID_TemperatureMax, slText$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif
  endswitch ' blKeycode
end
```

Organize Window Changing

This chapter explaines step by step how to change between two or more windows. We suggest copying the code fragments directly into your future projects. You will find this example in *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_3WindowChanging*

In step 1 the Tiger Graphic Library will be included.

```
#include TigerGraphicLibrary.INC
```

In step 2 the idenifiers for windows will be defined. For easier later addition of more windows we group the number of identifiers.

```
'' window identifiers
'' primary
#define WID_Home                10
#define WID_Settings            11
#define WID_Calibration         12
'' operation
#define WID_Start               20
#define WID_Stop                21
'' help
#define WID_HelpHome            30
#define WID_HelpHome2           31
#define WID_HelpHome3           32
#define WID_HelpSettings        33
#define WID_HelpCalibration     34
```

In step 3 the file *APPLICATION_gui.inc* will be included which contains all subroutines for the graphical user interface. This means initialization subroutines where the windows will be filled with elements and execution subroutines for the handling of the buttons in the windows.

```
#include APPLICATION_gui.inc
```

After the declaration of some local variables in step 4 and the installation of the device drivers in step 5 the initializations are done in step 6. For acceleration of the initialization a higher task prio for the initializing task main is set.

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

#include TGL_DEVICE_DRIVERS_TP1000.INC

set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
call bTglInit( blReturn )
wlElementId = 0
call wInitGeneral(                             wlElementId, blReturn )
call wInitHome(           WID_Home,            wlElementId, blReturn )
call wInitSettings(       WID_Settings,        wlElementId, blReturn )
call wInitCalibration(    WID_Calibration,     wlElementId, blReturn )
call wInitStart(          WID_Start,           wlElementId, blReturn )
call wInitHelpHome(       WID_HelpHome,        wlElementId, blReturn )
call wInitHelpHome2(      WID_HelpHome2,       wlElementId, blReturn )
call wInitHelpHome3(      WID_HelpHome3,       wlElementId, blReturn )
call wInitHelpSettings(   WID_HelpSettings,    wlElementId, blReturn )
call wInitHelpCalibration(WID_HelpCalibration,wlElementId, blReturn )
```

For the sub steps of step 6 see *APPLICATION_gui.INC.* Here we describe only the new ideas of this example. See the last chapter for more information.

In addition to the creating subroutines for elements you will find some initialization subroutines as part of the Tiger Graphic Library like *wTglInitPushButton*. These subroutines create one or more elements occasionally set some spezial attributes which lead to more complex elements or element groups. If there is a *Wnd* in the name these elements are placed in a window. The identifier of the elements will be incremented internally. That ist he reason why the variable of the identifier for element is not incremented afterwards.

```
wgEID_MenuButtons = wpvElementId
for blIdx = 0 to NUM_MenuButtons-1
  llAddr  = dlMenuButtons + blIdx*2*BMP_LEN_MenuButton
  llAddr2 = llAddr + BMP_LEN_MenuButton
  call wTglInitPushButton( &
  TGL_W1, TGL_H1, &   ' width, height of element
  llAddr, llAddr2, &          ' addresses of normal, alternative graphic
  wpvElementId, &             ' identifier of element
  bpvReturn )                 ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
next
```

The execution subroutines watches he touched buttons (step 6ii), change the identifier for the actually shown window and return this new identifier to the calling window administration subroutine.

```
call bTglWaitKeycode( blKeycode )
switchi blKeycode
case KEY_Help:
  wpvWindowId = WID_HelpHome
case KEY_Settings:
  wpvWindowId = WID_Settings
case KEY_Calibration:
  wpvWindowId = WID_Calibration
case KEY_OnOff:
  wpvWindowId = WID_Start
endswitch ' blKeycode
```

For a fast task switching in step 8 (APPLICATION.TIG) the priority is resetted to 1 when the program enters its main loop. The main loop for this example program is done in *bAdministrateWindows.* Here the execution subroutines for the windows are called.

```
set_task_prio main, 1 ' reset task priority for usual running
call bAdministrateWindows( blReturn )
```

For the administration of the shown windows we enter the endless loop with the first window to be shown in step 8.

```
wlWindowId = WID_Home
for ever = 0 to 0 step 0
…
next
```

In step 9 the current window is shown on lcd.

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

In step 10 the program divides into the executione subroutines of the windows. The return value is checked after the switch case construction.

```
        switchi wlWindowId

      '' primary
    case WID_Home:
      call wExecHome( wlWindowId, bpvReturn )
    case WID_Settings:
      call wExecSettings( wlWindowId, bpvReturn )
    case WID_Calibration:
      call wExecCalibration( wlWindowId, bpvReturn )

      '' operation
    case WID_Start:
      call wExecStart( wlWindowId, bpvReturn )

      '' help
    case WID_HelpHome:
      call wExecHelpHome( wlWindowId, bpvReturn )
    case WID_HelpHome2:
      call wExecHelpHome2( wlWindowId, bpvReturn )
    case WID_HelpHome3:
      call wExecHelpHome3( wlWindowId, bpvReturn )
    case WID_HelpSettings:
      call wExecHelpSettings( wlWindowId, bpvReturn )
    case WID_HelpCalibration:
      call wExecHelpCalibration( wlWindowId, bpvReturn )

    endswitch ' wlWindowId
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
```

Sample program:

```
'----------------------------------------------------------------------------
' TGL_APPLICATION_3WindowChanging/APPLICATION.TIG
'----------------------------------------------------------------------------

'****************************************************************************
'       Step 1: Include the files for the Tiger Graphic Library
'****************************************************************************
#include TigerGraphicLibrary.INC

'****************************************************************************
'       Step 2: Identifiers for the Tiger Graphic Library
'****************************************************************************
'' window identifiers
'' primary
#define WID_Home                10
#define WID_Settings            11
#define WID_Calibration         12
'' operation
#define WID_Start               20
#define WID_Stop                21
'' help
#define WID_HelpHome            30
#define WID_HelpHome2           31
```

```
#define WID_HelpHome3           32
#define WID_HelpSettings        33
#define WID_HelpCalibration     34

'****************************************************************************
'       Step 3: Inclusion of user specific application files
'****************************************************************************
#include APPLICATION_Gui.INC

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

  '**************************************************************************
  '     Step 4: Declaration of local variables
  '**************************************************************************
  byte blReturn         ' return value for TGL subroutines
  word wlElementId      ' incremental identifier for elements
  word wlWindowId       ' identifier for windows

  '**************************************************************************
  '     Step 5: Install the device drivers for the touch panel and the LCD
  '**************************************************************************
  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '**************************************************************************
  '     Step 6: Initialize
  '                i) Tiger Graphic Library
  '               ii) fonts, elements and windows
  '**************************************************************************
  set_task_prio main, TGL_INIT_TASK_PRIO ' speed up task for initializations
  call bTglInit( blReturn )
  wlElementId = 0
  call wInitGeneral(                            wlElementId, blReturn )
  call wInitHome(           WID_Home,           wlElementId, blReturn )
  call wInitSettings(       WID_Settings,       wlElementId, blReturn )
  call wInitCalibration(    WID_Calibration,    wlElementId, blReturn )
  call wInitStart(          WID_Start,          wlElementId, blReturn )
  call wInitHelpHome(       WID_HelpHome,       wlElementId, blReturn )
  call wInitHelpHome2(      WID_HelpHome2,      wlElementId, blReturn )
  call wInitHelpHome3(      WID_HelpHome3,      wlElementId, blReturn )
  call wInitHelpSettings(   WID_HelpSettings,   wlElementId, blReturn )
  call wInitHelpCalibration(WID_HelpCalibration,wlElementId, blReturn )


  '**************************************************************************
  '     Step 7: 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


'----------------------------------------------------------------------------
' bAdministrateWindows:
'----------------------------------------------------------------------------
' Show start window and administrate the change of windows
```

```
'---------------------------------------------------------------------------
' RETURN VALUES:
'   bpvReturn           tgl return value (0=OK, >0=error)
'---------------------------------------------------------------------------
sub bAdministrateWindows( var byte bpvReturn )
  byte ever        ' endless loop
  word wlWindowId ' identifier of currently shown window

  '*************************************************************************
  '     Step 8: determin start window
  '*************************************************************************
  wlWindowId = WID_Home
  for ever = 0 to 0 step 0

    '***********************************************************************
    '   Step 9: show current window
    '***********************************************************************
    call bTglShowWindow( wlWindowId, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
      return
    endif

    '***********************************************************************
    '   Step 10: watch window touches and switch the window
    '***********************************************************************
    switchi wlWindowId

      '' primary
    case WID_Home:
      call wExecHome( wlWindowId, bpvReturn )
    case WID_Settings:
      call wExecSettings( wlWindowId, bpvReturn )
    case WID_Calibration:
      call wExecCalibration( wlWindowId, bpvReturn )

      '' operation
    case WID_Start:
      call wExecStart( wlWindowId, bpvReturn )

      '' help
    case WID_HelpHome:
      call wExecHelpHome( wlWindowId, bpvReturn )
    case WID_HelpHome2:
      call wExecHelpHome2( wlWindowId, bpvReturn )
    case WID_HelpHome3:
      call wExecHelpHome3( wlWindowId, bpvReturn )
    case WID_HelpSettings:
      call wExecHelpSettings( wlWindowId, bpvReturn )
    case WID_HelpCalibration:
      call wExecHelpCalibration( wlWindowId, bpvReturn )

    endswitch ' wlWindowId
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
  next ' endless loop
end
```

Modifiy and Update Elements

This application could stand for a real application for the controlling of a heater In this chapter we demonstrate a simple but working application realizing a graphical user interface, window changing and displaying values of a measuring task in a graphical mode.

The explanations in this chapter assume the knowledge of the chapters before. Here we will explain the new ideas only. If you agree with the proposed program structure this code example can be taken as template for new applications. You will find this example in the directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\ TGL_APPLICATION_4ElementShowingUpdating*

After 3 standard steps The first new idea is the included file *APPLICATION_Temperature.INC* in step 4. All the subroutines which belong to the heater functionality are assembled in this file. Including files instead of a long *.TIG* file enables an easy inclusion of subroutines further applications without reinventing the wheel again or copying code lines. In this subroutine you will find an independent measuring task *tTemperature* which provides secondly the actual measured temperature and controls the heater. In addition to the task there are some short subroutines for converting analog input into real temperature values and for calibrating the heater.

```
#include APPLICATION_Temperature.INC
#include APPLICATION_Gui.INC
```

Mind the order of the include files! Subroutines which are called in include files must be placed in files which have been already included. E. g. a window execution subroutine which is placed in *APPLICATION_Gui.INC* and shows the actual temperature needs to call subroutines of *APPLICATION_Temperature.INC*. This means *APPLICATION_Temperature.INC* must be included before *APPLICATION_Gui.INC.*

The next new idea you will find in step 8. Here is a starting screen shown on lcd. This may be done directly after the installation of the device drivers. The starting screen bridges over the initialization time of the windows which could last some seconds. The more windows and elements you have the longer the initialization time will be. The starting output would suggest a short booting time.

```
  datalabel dlMaskWelcome
  dlMaskWelcome::
  data filter "MaskWelcome.bmp", "GRAPHFLT", 0  ' WxH=320x240  BmpWidth=320
  call vTglPutWindowGraphicF( dlMaskWelcome )
```

Mind that normally tgl subroutines must not be called before the initialization subroutine bTglInit has been called, but some subroutines like *vTglPutWindowGraphicF* may be called whenever you need them!

In step 9 the task for the temperature measuring and heater controlling is started. This task runs in an endless loop. For a fast task switching please mind the instructions for releasing the task in the end of loops. Releasing instructions are *release_task*, *wait_duration*, *wait_next* and *wait_clock*.

```
run_task tTemperature

task tTemperature
  byte ever
  word wlDigit
  randomize
  wait_next 1000
  for ever=0 to 0 step 0
    '' get temperature
    '' HERE: simulate 10-bit measurand between 503 and 519
    wlDigit = 503 + ((rnd(0)*16) shr 16)
    call rDigitToTemperature( wlDigit, rgTemperatureAct )

    '' control temperature
    '' HERE: too simple for a working controlling!
    if rgTemperatureAct < rgTemperatureNom then
      wgTemperatureDigitOut = limit(wgTemperatureDigitOut+1, 0, 1023)
    else
      wgTemperatureDigitOut = limit(wgTemperatureDigitOut-1, 0, 1023)
    endif

    wait_next
  next
end
```

In more complex windows the execution subroutines (*APPLICATION_gui.INC)* sometimes need to check more than just the pressed buttons. In the home window (*wExecHome*) a switch and the actual temperature need to be checked, too. In this case it is not possible just to wait for the next pressed button as the subroutine *bTglWaitButton* would do and we did in the examples before.

In step 13i (*wExecHome*) it is shown how to check the buttons without blocking the whole loop. You just check the filling of the button input buffer. Only if there are pressed buttons there is a valid keycode.

```
call bTglGetKeycode( blKeycode, wlIbuFill )
if 0 < wlIbuFill then
  switchi blKeycode
  case KEY_Help:
    wpvWindowId = WID_HelpHome
    return
  case KEY_Settings:
    wpvWindowId = WID_Settings
    return
  case KEY_Calibration:
    wpvWindowId = WID_Calibration
    return
  endswitch ' blKeycode
endif
```

In step 13ii (*wExecHome*) the internally administrated touch elements are checked. The subroutine *wTglGetTouchedElement* returns the identifiers of a touched switch, slider or listbox. In case of no touched element *TGL_NO_ID* is returned. If an element has been pressed the type specific subroutines *bTglGetButtonState*, *lTglGetSliderValue sTglGetListboxItem* or *wTglGetListboxIndex* give more information.

```
call wTglGetTouchedElement( wlElementId )
if wlElementId = wgEID_MenuButtons + 2*IDX_OnOff then
  call bTglGetButtonState( wlElementId, wpvWindowId, blState, bpvReturn)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  switchi blState
  case 0:
    wpvWindowId = WID_Stop
  case 1:
    wpvWindowId = WID_Start
  endswitch
  return
endif
```

In step 13iii (*wExecHome*) the temperature is checked and updated. For showing new information on lcd the Tiger Graphic Library provides show subroutines for the different types of elements. There are two basic ideas of showing information on LCD.

The first is the use of the graphic area of currently shown elements without saving the information with the element. You will find several subroutines like e.g. *bTglShowText* for strings, *bTglShowLong* for numerical values, *bTglShowGaugeValue* for gauges. For more show subroutines please see the API below in this manual. For an optimized lcd update these subroutines get passed a flag if the lcd should be updated immediately or not. If more than one element should be updated *TGL_FALSE* should be passed. After finishing all callings of show subroutines *vTglUpdate* is called for a single lcd update in one go.

```
  if rlTemperatureAct <> rgTemperatureAct then
    '' save new measurand
    rlTemperatureAct = rgTemperatureAct
    '' update internally first without updating lcd
    call sTblStrR( rlTemperatureAct, DECIMAL_PLACES,DECIMAL_POINT, slText$ )
    call bTglShowText( wgEID_TemperatureAct, slText$, &
    TGL_FALSE, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    call wTemperatureToDigit( rgTemperatureAct, wlDigit )
    llValue = wlDigit
    call bTglShowGaugeValue( wgEID_TemperatureGauge, llValue, &
    TGL_FALSE, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    '' status
    if rgTemperatureMin <= rlTemperatureAct &
      and rlTemperatureAct <= rgTemperatureMax then
      slTemperatureState2$ = "GOOD"
    else
      slTemperatureState2$ = "BAD"
    endif
    if slTemperatureState$ <> slTemperatureState2$ then
      slTemperatureState$ = slTemperatureState2$
      call bTglShowText( wgEID_TemperatureState, slTemperatureState$, &
      TGL_FALSE, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif
    '' update now the whole lcd in one go
    call vTglUpdate()   endif ' wlTemperatureAct <> wgTemperatureAct
  endif ' wlTemperatureAct <> wgTemperatureAct
```

Mind that show subroutines will not save the modifications with the element. In case of a window changing the information will get loosed! If you call a show subroutine for an element which is not placed in the currently shown window the subroutine will have no effect.

In step 13iv (*wInitSettings*) is demonstrated, how to define hidden buttons. These are touch fields with no bitmap which can be placed everywhere on the lcd. We use a hidden button here to avoid redundant data saving. We need to show the product code in the home window and in the settings window. This means we place the same element in these two windows. As we do not need a textbutton in the home window, we choosed a label where the product code is saved with. For the button functionality we underlay the label in the settings window with a button. Pass the parameters *TGL_NO_ADDR* and *TGL_NO_BMP* to make the button invisible.

```
call bTglPlaceLabelInWindow( &
 wgEID_ProductCode, wpWindowId, &      ' identifier of element, window
 TGL_X0a, TGL_Y1a, &            ' x, y coordinate on lcd
```

```
bpvReturn )                    ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif
call bTglCreateButtonWnd( &
TGL_W5, TGL_H1, &      ' width, height of element
TGL_NO_ADDR, TGL_NO_BMP, &     ' address, format width of bitmap
TGL_KEY_ATTR_NO_AUTOREPEAT, & ' key attributes auto repeat, beep, type
wpvElementId, wpWindowId, &    ' identifier of element, window
TGL_X0, TGL_Y1, &              ' x, y coordinate on lcd
KEY_ProductCode, &             ' keycode
bpvReturn )                    ' return code (0: OK exit  >0: error exit)
if bpvReturn <> TGL_MSG_OK then
  return
endif
wpvElementId = wpvElementId + 1
```

Step 13v (*wExecSettings*) demonstrates how the information can be saved with the element with a setting subroutine. The setting subroutine does not mind the placement in a certain window. The information will be saved with the element and shown whenever a window where this element is placed in is called to be shown on lcd.

```
    slText$ = "Product Code"
    call sTglGetKeyboardInput( &
    WID_keyboard, wgEID_Keyboard, &' identifiers for keyboard
    10h, slText$, &    ' maximal user input length, user input
    bpvReturn )                ' return code (0: OK exit  >0: error exit)
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    if 0 < len(slText$) then
      sgProductCode$ = slText$
      call bTglSetText( wgEID_ProductCode, sgProductCode$, bpvReturn )
      if bpvReturn <> TGL_MSG_OK then
        return
      endif
    endif
    return
```

Mind that saving information with an element with the setting subroutines does not mean updating information on the lcd! The new information will be shown not before a show subroutine has been called afterwards! In our example the new information will appear with the calling of *bTglShowWindow* after the return instruction in *wAdministrateWindows.*

In step 14 (*APPLICATION.TIG*) it is demonstrated how the tgl return code can be used for additional application specific error codes. This could help debugging the program.

```
    '' invalid window
    bpvReturn = 0FFh
```

To avoid later errors please mind this:

NEVER call a tgl subroutine when the task switching is disabled! Otherwise the program could hang in an endless loop because of disabled internal tgl tasks which grant a correct multitasking functionality. Interrupt tasks disable the task switching, too!

For the whole sample program please see *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_ 4ElementShowingUpdating*

Additional Features

In this chapter we demonstrate how to integrate some additional features of the Tiger Graphic Library. In the described example the graphical user interface is supported by blinking elements, Elements are moved on the lcd, the application data is saved in the on board eeprom and the stand-by mode is activated. For more features see the API below in this manual.

The explanations in the following subchapters assume the knowledge of the chapters before. Each following chapter explain one new feature. You will find the code example in
%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_BASIC_APPLICATION\TGL_APPLICATION_ 5EepromStandbyBlink

Free User Flash Sectors

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

```
datalabel dlTglFirstFreeAddr
dlTglFirstFreeAddr::
```

To make sure that there is no more flash data which is poked with the download, Set this datalabel AFTER all file inclusions in the beginning of the task main.

Saving Data in Eeprom

Normally there are some user configurations which must be saved for resets or restarts of the program. For an easy handling there is an on board eeprom on the TP1000. In step 4 an include file for the user configuration of this application is included.

The calibration parameters for the touchpanel are already saved on this eeprom. The addresses can be set in the configuration file *TigerGraphicLibraryConf.INC.* Default addresses for these parameters are the highest addresses from 65509 to 65535.

```
#include APPLICATION_Eeprom.INC
```

Mind the order of the include files! Subroutines which are called in include files must be placed in files which have been already included. E. g. a window execution subroutine which is placed in *APPLICATION_Gui.INC* and shows the actual temperature needs to call subroutines of *APPLICATION_Temperature.INC*. This means *APPLICATION_Temperature.INC* must be included before *APPLICATION_Gui.INC.*

To hold a good overview over the saved data in the eeprom we propose to determine a memory seizure. A recursive definition of the data addresses will help you avoiding overlapping data.

```
#define EEPROM_StartData        0
#define EEPROM_Pwd              0
#define EEPROM_ProductCode      (EEPROM_Pwd + MAX_LEN_PWD)
#define EEPROM_TemperatureNom   (EEPROM_ProductCode + MAX_LEN_PRODUCTCODE)
#define EEPROM_TemperatureMin   (EEPROM_TemperatureNom + REAL_LEN)
#define EEPROM_TemperatureMax   (EEPROM_TemperatureMin + REAL_LEN)
#define EEPROM_TemperatureAxisInterception (EEPROM_TemperatureMax+REAL_LEN)
#define EEPROM_DataLen  (EEPROM_TemperatureAxisInterception+REAL_LEN)
```

The i²c setup for the eeprom ports and pins is done with the installation of the device drivers in step 6.

```
#include TGL_DEVICE_DRIVERS_TP1000.INC
```

With each program start the program get its settings form the eeprom. This done with the other initializations in step 8.

```
call vInitEeprom()
```

The eeprom initialization subroutine devides the first start after download from subsequent starts. In this program in the 128bytes flash area which is handled by the functions *serial_no$* and *set_serial_no* is used to set a flag for the first start. In this case the default settings will be saved in the eeprom.

```
sub vInitEeprom()
  long llFlag
  '' check first start
  if serial_no$( 0, BYTE_LEN ) = "FF"% then ' first start after download
    call vSetDfltEeprom()
    call vWriteToEeprom()
    llFlag = set_serial_no( "00"%, 0, 1 )
  else                                        ' later starts
    call vReadFromEeprom()
  endif
end
```

If a setting is changed like in the execution subroutine *wExecSettings* (*APPLICATION_gui.INC*) just overwrite the old value in the eeprom e. g.

```
call vWriteStringToEeprom( slText$, EEPROM_ProductCode )
```

Stand-by

The TigerGraphicLibrary provides a stand-by mode. Stand-by means switching off the backlight for energy saving. The stand-by mode will be activated in case of missing user inputs for a certain time. The stand-by mode will be left with the next touch. See the stand-by subroutines in the api for more features.

Stand-by functionality is activated by setting the standby time as it is done in step 12.

```
call vTglSetStandbyTime( 60 )
```

The stand-by time can be changed whenever you want in your program by recalling the setting subroutine. Passing zero means deactivating stand-by functionality.

Password Protection

A password protection for windows can be easily realized by a password request for selected windows. In our example program we have integrated this password request in step 13 in the window administration subroutine *bAdministrateWindows.*

```
if wlWindowId = WID_Calibration and bgLoggedIn = FALSE then
  slPwd$ = "Password"
  call sTglGetKeyboardInput( &
  WID_Keyboard, wgEID_Keyboard, &' identifiers for keyboard
  MAX_LEN_PWD, slPwd$, &     ' maximal user input length, user input
  bpvReturn )                ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  '' HERE: let all passwords be valid
  ''''if slPwd$ <> sgPwd$ and slPwd$ <> MASTER_PWD then
  if 0 = len(slPwd$) then
    wlWindowId = wlWID_Old
  endif
endif
```

The password will be requested when a protected window should be shown. The flag *bgLoggedIn* helps avoiding a multiple login. If whiched, the password input can be hidden by “*”. See *sTglGetKeybPassword* for more information.

In case of an invalid password, the identifier of the last window must be saved. to return to this window. This is done in step 14.

```
wlWID_Old = wlWindowId
```

If the password can be set by the user as in this application, it is helpful to define a master password which is always valid. The master password enables the setting of a new password in case of a forgotten password.

Blinking Elements

Blinking elements can help to hightlight actual and important information. In the code example we use blinking elements to guide user. E.g. if in the settings window (*APPLICATION_gui.INC*) something has been changed, the save button begins to blink. This reminds the user to press the save button to finish the changings. The savebutton itself stops blinking after pressing.

```
switchi blKeycode
case KEY_Save:
  call bTglSetAttribute( wlEID_Save, wpvWindowId, &
  TGL_ATTR_BLINK_SPEED, TGL_BLINK_SPEED_OFF, bpvReturn )
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
    …
case KEY_ProductCode:
    …
    call bTglSetAttribute( wlEID_Save, WID_Settings, &
    TGL_ATTR_BLINK_SPEED, TGL_BLINK_SPEED_SLOW, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
```

The password will be requested when a protected window should be shown. The flag *bgLoggedIn* helps avoiding a multiple login.

In case of an invalid password, the identifier of the last window must be saved. to return to this window. This is done in step 14.

```
wlWID_Old = wlWindowId
```

If the password can be set by the user as in this application, it is helpful to define a master password which is always valid. The master password enables the setting of a new password in case of a forgotten password.

Moving Elements

The coordinates of elements in a window can be changed with the subroutine *bTglSetCoordinates*. In this application the limit pointers are moved in case of changings oft the limits or and after a calibration. The movement is done in the subroutine *bSetGaugePtr* (*APPLICATION_gui.INC*).

```
call bTglSetCoordinates( wlElementId, wlWindowId, &
227, wlY, bpvReturn )
```

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.

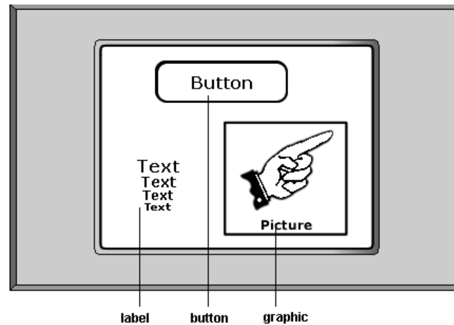


figure 8: Elements of the Tiger Graphic Library

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 9: Input number

figure 10: 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.

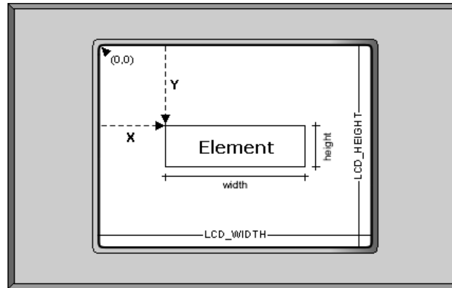
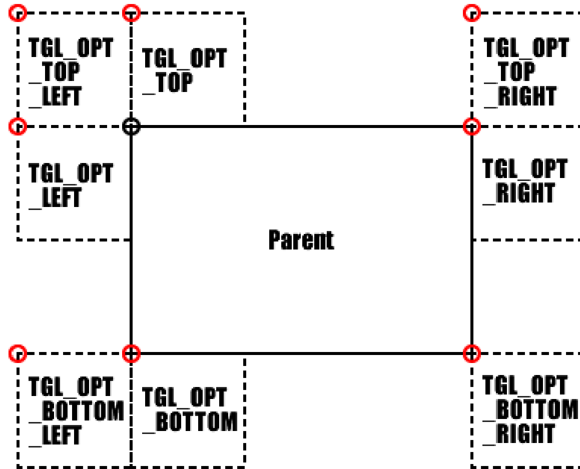


figure 11: Creating and placing elements

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 alternative way to place elements relative to other elements. You can move the docking point with the x-/y-offset.



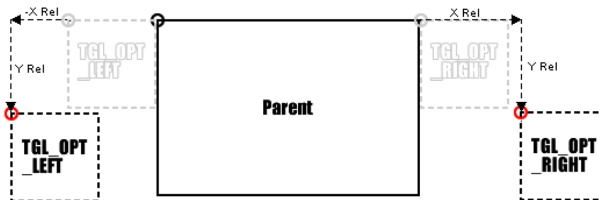


figure 12: Docking elements

Integrated Applications

Integrated applications are no elements, but consist of one or more predefines 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.

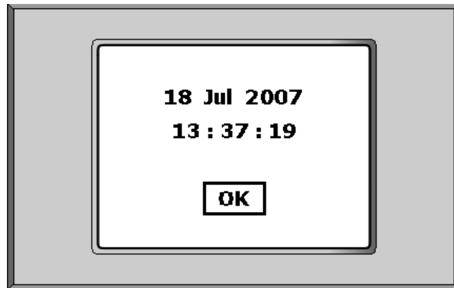


figure 13: RTC application as integrated application

figure 14: 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 15: 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 Button* 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:

- *bTglCreateFont*
- *bTglCreateFontParams*
- *bTglSetFontBmp*
- *bTglChangeFont*
- *bTglDeleteFont*
- *bTglSetFontParams*
- *bTglGetFontParams*
- *sTglBuildTextGraphic*
- *sTglBuildTextGraphicRotated*
- *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*

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*
- *wTglCalcCircleParams*
- *sTglDrawPie*
- *sTglUpdatePie*
- *sTglDrawHand*
- *sTglUpdateScope*
- *sTglDrawGraph*
- *sTglClearGraph*
- *sTglRotate*
- *sTglRotateToDst*
- *sTglDrawLine*
- *sTglDrawNextLine*
- *sTglGraphicMove*
- *sTglGraphicErase*
- *sTglGraphicInvert*

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.

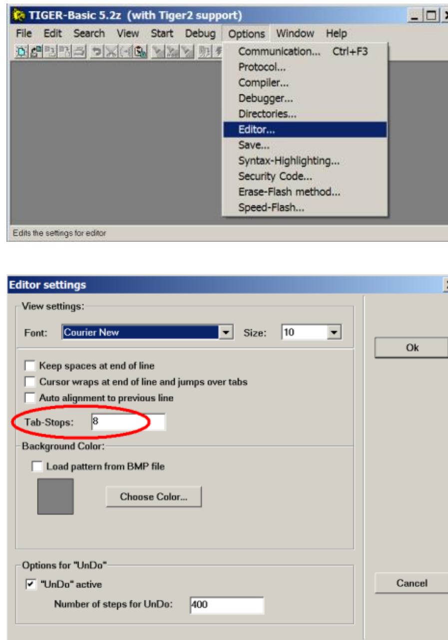


figure 16: Editor settings

Tips and Tricks

Checklist for your program code

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 global variables 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. Save the bitmaps and texts for the elements in the flash memory
8. Copy the configuration file TigerGraphicLibraryConf.INC in your project directory ans apply the used fonts.
9. Create the elements and place them in windows and save the identifiers for those elements to work with later in the program
10. Display the window with its elements on the LCD
11. Watch the touched elements and occasionally update the element contents or change the window

Each subroutine of the Tiger Graphic Library returns a result about its operation. It helps you debugging your program. The return value reports a valid or invalid handling of subroutines. If an element in your program does not work as you expect, please check this return value and see the error code table in the end of this manual. For a target-oriented search start controlling your passed parameters to the tgl subroutines in the following order:

- creation of elements or initialization of the integrated applications
- 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 could hang in an endless loop because of disabled internal tgl tasks which grant a correct multitasking functionality. Interrupt tasks disable the task switching, too!

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 *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\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.

If it is necessary there could be more configurations done with changing the definition file *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\TigerGraphicLibrary\TigerGraphicLibraryDefs.INC.* See the inline comments for more information. The handling is similar to the configuration file. Only do changes in a copy of this file in your project directory.

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 erors 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 RAM, FLASH memory and shorten compiling time. The default configuration actually takes about 320kb code (=5 flash sectors), 66kb flash memory (=2 flash sectors) and 517kb RAM. Please mind that you need nearly no additional memory for your coded graphical user interface.

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_APPLICATIO NS	Activates all graphical RTC applicati

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
```

LCD Settings

There are different types of LCDs. For the LCD type with black dots and a white background you should choose the normal mode *LCD_INVERSION_MODE=0*. For LCD types with white dots and a blue background the output should be inverted *LCD_INVERSION_MODE=1*.

```
#define LCD_INVERSION_MODE      0 ' 0=normal   black dots white background
'                                 ' 1=inverted white dots blue  background
```

If you like to use the LCD in an other orientation than upright landscape you have 2 possibilities. Rotate single elements on the unrotated LCD or rotate the whole LCD with all its elements.

For the first possibitity rotate each element which needs to be rotated by calling the subroutine *bTglSetAttribute*. The coordinate system stays as it is for the LCD in upright landscape format. All parameters must be passed fitting to this coordinate system. This is the solution for a speed optimized application if the bitmaps are included pre-rotated.

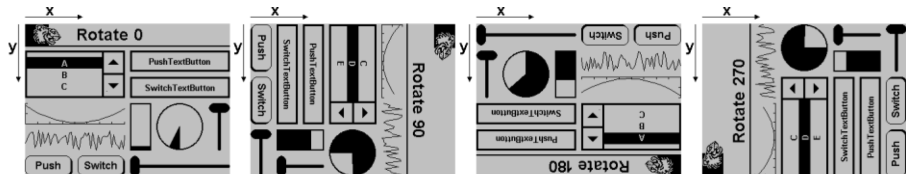


figure 17: Rotated elements on unrotated LCD

For examples see directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_Rotate*

The second possibility is to define a global LCD orientation. With this definition, the whole LCD area will be rotated. The graphical reference point will be replaced to the top left edge in the rotated area. This means a new coordinate system. Element sizes and coordinates must be passed now, as they are in the rotated area. This solution is a little bit easier for the parameters but the showing of the elements will take time for the rotations.

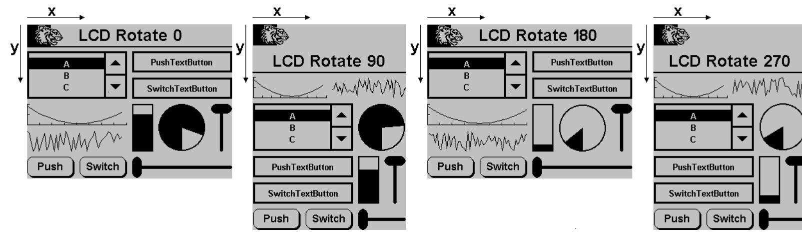


figure 18: Unrotated elements on rotated LCD

If the LCD should be used in upright portrait mode, the following definitions must be activated:

```
'#define TGL_LCD_ROTATE_0        ' upright landscape     reset button top
#define TGL_LCD_ROTATE_90        ' upright portrait      reset button left
'#define TGL_LCD_ROTATE_180      ' upside-down landscape reset button bottom
'#define TGL_LCD_ROTATE_270      ' upside-down portrait  reset button right
```

For examples see directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Examples\TigerGraphicLibrary\TGL_LcdOrientation*

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	8
TGL_MAX_NUM_GRAPHICS	10
TGL_MAX_NUM_LABELS	19
TGL_MAX_NUM_BUTTONS	11
TGL_MAX_NUM_TEXT_BUTTONS	20
TGL_MAX_NUM_SLIDERS	44
TGL_MAX_NUM_GAUGES	16
TGL_MAX_NUM_LISTBOXES	16

Example: You need 100 buttons in your application. This will reserve 1100 bytes of RAM. Each button needs 11 bytes => 100*11=1100 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
TGL_MAX_NUM_WINDOWS	maximal numbers of windows
TGL_WINDOW_ATTRIBUTES_LEN	size of memory pool for specific attributes of elements in windows
TGL_MAX_NUM_BLINK	maximum number of blinking elements in the same window
TGL_MAX_NUM_BUTTONS_IN_WINDOW	maximal number of buttons in one window
TGL_MAX_NUM_SLIDERS_IN_WINDOW	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 built 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
TGL_GRA_WIDTH	maximum width of text graphics
TGL_GRA_HEIGHT	maximum height of text graphics
TGL_TXT_GRAPHIC_TXT_LEN	maximum text length for building text graphics in one go
TGL_MAX_NUM_FONTS	maximum number of created fonts

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

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
```

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 19: Keyboard Hungarian style unshifted

figure 20: 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_APPLICATIOPNS 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 21: displaying time and date in digital style 1

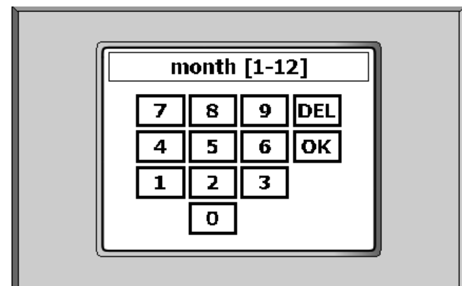


figure 22: setting time and date in digital style 1

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 definitions fitting to the TP1000.

For the types of the touch panel see the table below. For details see the manual of the touch panel device driver. For the TP1000 it is fundamental, if you use the analog inputs (TP_TYP_0) or not (TP_TYP_1).

```
#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 ANOLOG1 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 ANOLOG1 or ANALOG2 parallel to the TOUCHPANEL)
5	TP_TYP_5	TEC-1000 (stand alone, without other analog device drivers)

From the eeprom address *TP_CALIB_EEPROM_ADDR* on are 26 bytes reserved for touch panel calibration. Mind this when using the eeprom. The default adresses are the last eeprom adresses.

```
#define TP_CALIB_EEPROM_ADDR    (SIZE_EEPROM-1-TP_CALIB_LEN)
```

For explicit calibration please call *bTglCalibrateTp*.

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 V1.14 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
'#define TGL_V1_15
'#define TGL_V1_16
'#define TGL_V2_00
```

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.

Application Programming Interface (API)

The following chapters describe the application programming interface (API) of the Tiger Graphic Library. The descriptions are assembled in thematical groups. You will find the same groups as include files for the Tiger GraphicLibrary in the directory *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\TigerGraphicLibrary.*

In the first chapter are general subroutines. Specific subroutines for the elements follow. For a fast application development there are subroutines for whole keyboards and rtc handling. One chapter is for additional subroutines for text graphics which give a more elemental control over the graphic fonts. The subroutines in chapter *user graphic* help putting graphics on lcd in a secure mode without disturbing the internally generated lcd outputs. Graphical functions are subroutine the Library calls itself. Passing your own variables will give you the full control.

General Subroutines

The subroutines described here are for all types of elements.

Subroutine for the initialization of the Tiger Graphic Library

- *bTglInit*

Subroutines for showing and hiding elements

- *bTglShowWindow, bTglHideWindow*
- *bTglShow, bTglHide*
- *bTglShowText*
- *bTglShowLong, bTglShowWord, bTglShowByte, bTglShowReal*
- *bTglShowGraph, bTglHideGraph*
- *vTglUpdate, vTglUpdateParams*
- *bTglUpdateScope*

Subroutines for modifying and deleting elements or getting element attributes

- *bTglLink*
- *bTglDeleteElement*
- *bTglDeleteElementFromWindow*
- *bTglSetSize, wTglGetSize*
- *bTglSetAddress, lTglGetAddress*
- *bTglSetBmpWidth, wTglGetBmpWidth*
- *bTglSetText, sTglGetText*
- *bTglSetFont*
- *bTglSetFrame*
- *bTglSetMargins, wTglGetMargins*
- *bTglSetCoordinates, wTglGetCoordinates*
- *bTglSetLimits*
- *bTglSetAttribute, bTglGetAttribute*

Subroutines for controlling LCD and touch panel

- *vTglSetStandbyTime, lTglGetStandbyTime*
- *bTglSetStandbyTermination*
- *vTglDelayStandby*
- *bTglSwitchStandby*
- *bTglGetStandbyState*
- *vTglWaitTouchTp*
- *vTglWaitReleaseTp*
- *bTglGetTouch*
- *wTglGetTouchedElement*
- *wTglGetNumTouchedElements*
- *bTglGetTouchedElementsFlag*
- *bTglCalibrateTp*
- *vTglBeep*

bTglInit

call bTglInit(blReturn)

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

Parameters:

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

! Call *bTglInit* for initialization **before** calling any other subroutine of the Tiger Graphic Library.

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
						Return Values:
Result	●	-	-	-	-	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.

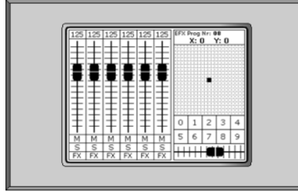


figure 23: Linked elements: slider buttons on sliderbars

figure 24: Linked elements dark background for user input

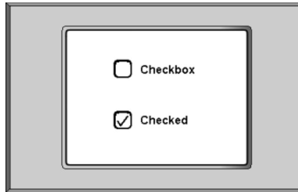


figure 25: Linked elements empty and checked boxes

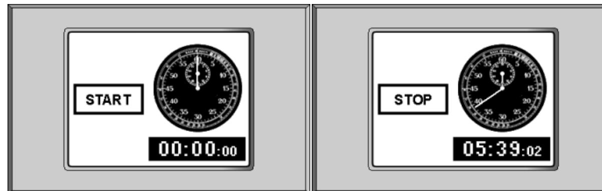


figure 26: Linked element showing alternative texts "start" "stop"

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
						Return Values:
Result	●	-	-	-	-	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
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  wlElementId = 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
wlElementId, 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( wlElementId, 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
  wlElementId, WINDOW_ID, &      ' identifier of element, window
  30, 30, &                      ' x,y coordinate on LCD
  041h,      &                   ' key code
  blReturn )                     ' return value
  call bTglShow( wlElementId, 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
						Return Values:
Result	●	-	-	-	-	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
'-----------------------------------------------------------------------------
#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      ' current identifier for creation of elements

   #include TGL_DEVICE_DRIVERS_TP1000.INC

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

```
  call bTglInit( blReturn )
  wlElementId = 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
  wlElementId, 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(  wlElementId, WINDOW_ID, blReturn )

    '' after 2s replace button in window and show it on LCD
    wait_duration 2000
    call bTglPlaceButtonInWindow( &
    wlElementId, WINDOW_ID, &       ' identifier of element, window
    30, 30, &                       ' x,y coordinate on LCD
    041h,       &                   ' key code
    blReturn )                      ' return value
    call bTglShow( wlElementId, 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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

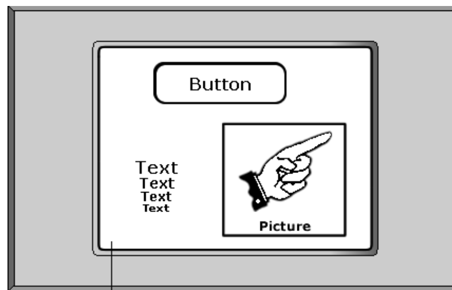


figure 27: Show window

! 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.

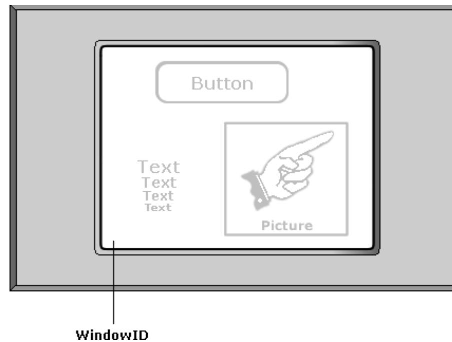


figure 28: Hide window

! 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
						Return Values:
Result	●	-	-	-	-	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*.

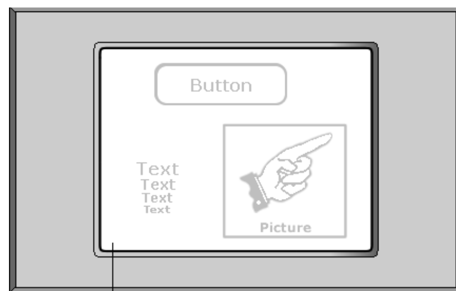


figure 29: All elements hidden

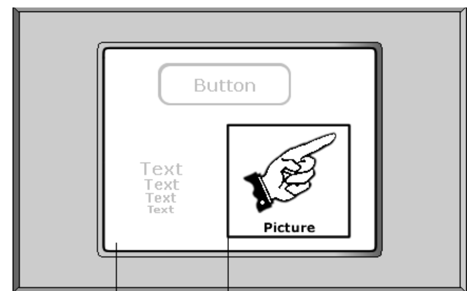


figure 30: One element shown

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
						Return Values:
Result	●	-	-	-	-	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().

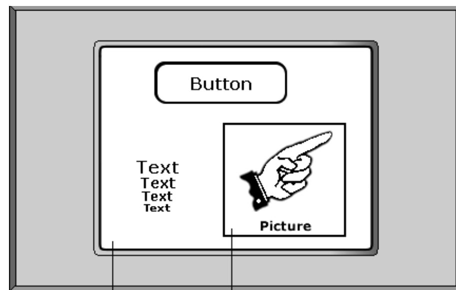


figure 31: All elements shown

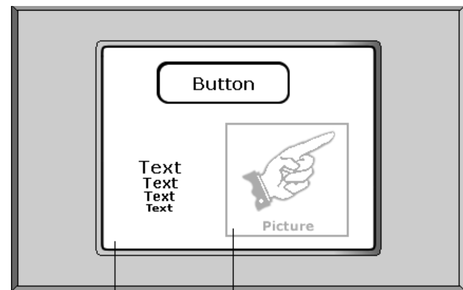


figure 32: One element hidden

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_SHOW_HIDE.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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( &
```

```
    wlBUTTON_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
						Return Values:
Result	●	-	-	-	-	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, WordWalue, 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
						Return Values:
Result	●	-	-	-	-	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.

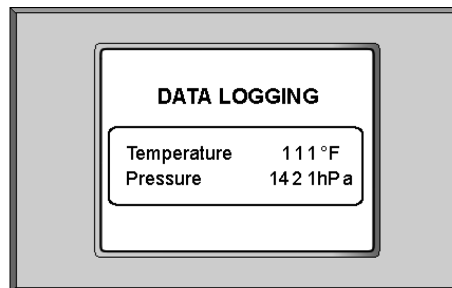


figure 33: Monitoring of integer numbers

! 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.

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
'----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
wlElementId = wlElementId + 1
' create slider button
call bTglCreateGraphic( &
32, 16, &                       ' width, height of element
dlSliderButton, 32, &           ' address, format width of bitmap
wlElementId, &                  ' identifier of element
blReturn )                      ' return code (0: OK exit  >0: error exit)
' link slider button to slide bar
call bTglLink( &
wlSLIDER_ID, &                  ' identifier of slide bar
wlElementId,&                   ' identifier of slider button
blReturn )                      ' return code (0: OK exit  >0: error exit)
'' increment identifier for next element
wlElementId = wlElementId + 1

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

'*********************************************
' show current slider value
'*********************************************
call bTglShowWindow( WINDOW_ID, blReturn )
loop 7FFFFFFFh
  call lTglGetSliderValue( &
  wlSLIDER_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)
  call bTglShowLong( wlLABEL_ID, llSliderValue, TRUE, blReturn )
  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
						Return Values:
Result	●	-	-	-	-	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.

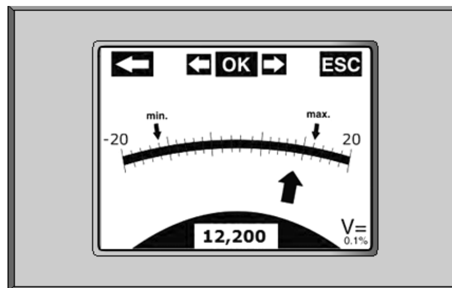


figure 34: Monitoring of floating point numbers

! 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.

The subroutine *bTglShowReal* 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*.

bTglShowDetail, ShowDetailF

```
call bTglShowDetail(   GraphicId, Src$,   WSrc, Width,Height, X,Y, Result )
call bTglShowDetailF(  GraphicId, Addr,   WSrc, Width,Height, X,Y, Result )
```

Function: Shows a detail of a bitmap in the area of an element of type *graphic.* Bitmap is stored in RAM resp. FLASH memory.

Parameters:

	B	W	L	S	F	
GraphicId	●	-	-	-	-	identifier of a graphic
Src$	-	-	-	●	-	contains bitmap
Addr	-	-	●	-	-	first address of bitmap in flash memory
WSrc	-	-	●	-	-	bitmap format width (must be a multiple of 8!)
Width, Height	-	-	●	-	-	bitmap pixel size
X,Y	-	-	●	-	-	position of left top edge of bitmap detail
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

bTglShowGraph

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

Function: Draws a scaled graph in the area of the element of type *graphic.*

Parameters:

	B	W	L	S	F	
GraphicId	●	-	-	-	-	identifier of a graphic for the drawing area
Data$	-	-	-	●	-	buffer containing numerical values of the same type
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 1 invalid value, must be at least 2 or 0
NumTotal	-	●	-	-	-	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 **Return Values:**
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

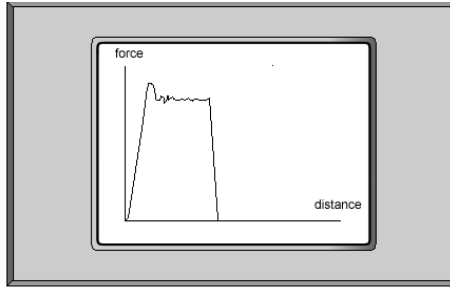


figure 35: Graph

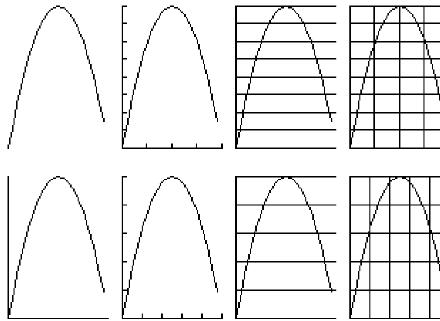


figure 36: axis types

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

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
```

If the axes should be drawn alone without any value, pass an empty string *Data$.*

For clearing the graphs line call *bTglHideGraph.* For erasing a graph inclusive its axes 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 reshowing the window the will not be shown unless you have called *bTglShowGraph*.

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 *Graphical Functions.*

bTglHideGraph

call bTglHideGraph(GraphicId, Axis, Result)

Function: Clears the graph's line. Axes will stay uncleared!

Parameters:

	B	W	L	S	F	
GraphicId	●	-	-	-	-	identifier of a graphic for the drawing area
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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

bTglUpdateScope

call bTglUpdateScope(ElementId, Data$,Size, Min,Max, Step, Direction, Return)

Function: Update graph of oszilloscope with new values.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of element
Data$	-	-	-	●	-	old and new value of graph, LSB first
Size	●	-	-	-	-	variable type of value 1=byte, 2=word, 4=long, 8=real
Min,Max	-	-	-	-	●	limits for values of graph
Step	-	●	-	-	-	horizontal pixel translation of graph for these values
Direction	●	-	-	-	-	translation direction of graph TGL_DIR_RIGHT, TGL_DIR_LEFT TGL_DIR_BOTTOM, TGL_DIR_TOP
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

The values can be easily written in Data$ by calling the functions ntos$, rtos$ or pushn.

vTglUpdate

```
call vTglUpdate()
```

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

vTglUpdateParams

```
call vTglUpdateParams( Y, Height )
```

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

Parameters:

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

wTglGetType

```
call wTglGetType( ElementId, Type, Result )
```

Function: Get type of an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
						Return Values:
Type	-	●	-	-	-	type of element
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
Width, Height	-	●	-	-	-	size of element
Result	●	-	-	-	-	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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
Address	-	-	●	-	-	flash address of text rep. bitmap
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

bTglSetBmpWidth

call bTglSetBmpWidth(ElementId, BmpWidth, Result)

Function: Set format width of bitmap related to an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of element
BmpWidth	-	-	●	-	-	format width of bitmap
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						›0 error

wTglGetBmpWidth

call wTglSetBmpWidth (ElementId, BmpWidth, Result)

Function: Return format width of bitmap related to an element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of element
						Return Values:
BmpWidth	-	-	●	-	-	format width of bitmap
Result	●	-	-	-	-	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:
Text	-	-	-	●	-	new text for element
						Return Values:
Result	●	-	-	-	-	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 listboxes, 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
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
```

sTglGetText

```
call sTglGetText( ElementId, Text, Result )
```

Function: Return text which is saved with the text element.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element with text:
						Return Values:
Text	-	-	-	●	-	text of element
Result	●	-	-	-	-	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 listboxes, labels, and text buttons.

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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
Top,Bottom, Left,Right	-	●	-	-	-	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
						Return Values:
Result	●	-	-	-	-	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
						Return Values:
X,Y	-	●	-	-	-	coordinates of an element in a window
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						›0 error

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_bTglSetCoordinates_bTglGetCoordinates.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
  041h, &                         ' 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)
						Return Values:
Result	●	-	-	-	-	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 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270	rotates element content (bitmap, text,gauge,graph) xxx° 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

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

wlGRAPHIC_ID = wlElementId ' 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!

The idea of the attribute *TGL_ATTR_ROTATE* in the Tiger Graphic Library is a rotation of a single element on the LCD used upright landscape. The parameters for the creation and placement of a rotated element must be passed as the parameters are for a rectangle placed on the unrotated LCD used upright landscape. The size of the created element must be equal the size of the included bitmap after its rotation.

E. g. you have a bitmap WxH=88x27 and want to rotate it for 90° to the right. The size of the created element has the rotated values WxH=27x88. The format width of the bitmap must be given as it is saved in the flash memory as a multiple of 8. For this example it is 88.

```
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
wlElemenId,WINDOW_ID, &         ' identifier of graphic, window
146, 135, &                     ' x,y coordinate on LCD
blReturn )                      ' return code (0: OK exit  >0: error exit)

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

With the attribute TGL_ATTR_ROTATE the content of the **single** element will be shown right rotated. Contents could be bitmaps, texts, graphs, gauges or listboxes. Setting this attribute for the elements the LCD can be used in portrait format as well as in landscape format. Normally only variable texts and graphs need to be rotated during runtime. Bitmaps can be rotated before their inclusion, gauges can be chosen with the needed base and fix elements can be normally assembled on a single graphic which is used as background mask for the window. See *wTglInitMaskWnd* for more information.

If **all** elements shall be rotated, you can alternatively rotate the global LCD orientation by configuring the TGL. Please see section *LCD settings* in chapter *Configuration* for detailed information. Mind that all rotations have to be done during run-time.

!

If bitmap elements are created by usin tgl-init-subroutines, the format width of the saved bitmap will be calculated internally as the next multiple of 8 to the width. If the bitmap(s) should be rotated for the element afterwards for 90° or 270° by calling *bTglSetAttribute* the calculated format width can differ to the existing format width. This has no effect for the correct working of the TGL subroutines and may be ignored, if the saved bitmap has the format width of the next multople of 8. If not set the correct format width by calling the setter *bTglSetBmpWidth*. The getter *wTglGetBmpWidth* will return the saved value.

!

Bitmaps of simple graphics and buttons can be rotated BEFORE their inclusion. This way the rotation is already done for each showing of this bitmap element and need no CPU load during the running program.

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 pther elements in the orter 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 ce 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.

lTglGetAttribute

call lTglGetAttribute(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
						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
'-----------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'***************************************
' IDENTIFIER
'***************************************' 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 wlIbuFill        ' filling of the touch panel input buffer
  byte blKeycode        ' returned keycode

  #include TGL_DEVICE_DRIVERS_TP1000.INC
```

```
'***********************************
' INITIALIZATION
'***********************************
call bTglInit( blReturn )
wlElementId = 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
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 it is pressed
call bTglSetAttribute( &
wlElementId, 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, wlIbuFill   ' get buffer length
  if wlIbuFill > 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 stand-by 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.

lTglGetStandbyTime

call lTglGetStandbyTime(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	Return Values:
Seconds	-	-	●	-	-	current stand-by time of the system in seconds

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_STANDBY.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )

  '***********************************
  ' 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 (= running output)
TGL_STANDBY_TERM _LCD_OFF	Stand-by mode will NOT be terminated anymore by active LCD driver (= running output)

This function (re)sets only additional modes in which the stand-by mode will be left. For the (de)activation of stand-by functionality call *vTglSetStandbyTime.*

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_STANDBY_LCD_terminated.TIG
'-----------------------------------------------------------------------------
#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 wlIbuFill          ' filling of input buffer
word wlElementId        ' current identifier for creation of elements
word wlBUTTON_ID        ' "constant" identifier of button

#include TGL_DEVICE_DRIVERS_TP1000.INC

'***********************************
' INITIALIZATION
'***********************************
call bTglInit( blReturn )
wlElementId = 0

'***********************************
' TGL ELEMENTS AND WINDOOWS
'***********************************
wlBUTTON_ID = wlElementId       ' 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
wlElementId, 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, wlIbuFill
  while 0 = wlIbuFill
    release_task
    get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
  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, wlIbuFill
    while 0 = wlIbuFill
      release_task
      get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
    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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

Sample program:

```
'-------------------------------------------------------------------------------
' TGL_STANDBY_switch.TIG
'-------------------------------------------------------------------------------
#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 wlIbuFill        ' filling of input buffer
  word wlElementId      ' current identifier for creation of elements

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '***********************************
  ' INITIALIZATION
  '***********************************
  call bTglInit( blReturn )
  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
  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, wlIbuFill
    while 0 = wlIbuFill
      release_task
      get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
    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
```

vTglDelayStandby

```
call vTglDelayStandby()
```

Function: Delays the switching in the stand-by mode.

Touch elements like buttons, switches and sliders delays the switching in the stand-by mode automatically. With this subroutine the user can delay the switching everywhere in the program.

lTglGetStandbyState

```
call lTglGetStandbyState( State )
```

Function: Get the stand-by state.

Parameters:

	B	W	L	S	F	Return Values:
State	●	-	-	-	-	stand-by state TGL_ON/TGL_OFF

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	
						Return Values:
Flag	●	-	-	-	-	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	Return Values:
ElementId	-	●	-	-	-	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	
						Return Values:
Flag	-	●	-	-	-	TGL_TRUE: buffer overflow TGL_FALSE: no buffer overflow

Last entrys will be overwritten in case of overflow.

bTglCalibrateTp

```
call bTglCalibrateTp( Result )
```

Function: Forces the touch panel calibration.

Parameters:

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

vTglBeep

```
call vTglBeep( Number )
```

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

Parameters:

	B	W	L	S	F	
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.

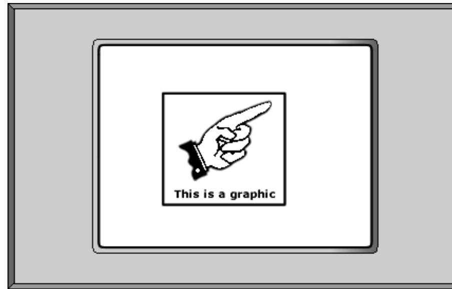


figure 37: Graphic

Subroutines:

- *wTglInitMaskWnd*
- *bTglCreateGraphicWnd, bTglCreateGraphicDockWnd*
- *bTglCreateGraphic*
- *bTglPlaceGraphicInWindow, bTglDockGraphicInWindow*

See also:

- *bTglLink*
- *bTglSetAttribute*

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

Element	Functionality
	again

Graphics can be used as graphical areas for graphs and scopes. Pass the parameters TGL_NO_ADDR and TGL_NO_BMP as flash address and format width of bitmap, to create an empty graphic.

```
call bTglCreateGraphicWnd( &
200, 200, &                         ' width, height of element
TGL_NO_ADDR, TGL_NO_BMP, &          ' address, format width of bitmap
wpvElementId, wpWindowId, &         ' identifier of element, window
60, 20, &                           ' x,y coordinate on LCD
bpvReturn )                         ' return code (0: OK exit  >0: error exit)
```

wTglInitMaskWnd

call wTglInitMaskWnd(BmpAddr, WindowId, X,Y, ElementId, Result)

Function: Creates a background mask for the whole lcd and places it in a window at position XY. Increments the identifier for 1 created element.

Parameters:

	B	W	L	S	F	
BmpAddr	-	-	●	-	-	address of the graphic bitmap in flash memory
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
						Return Values:
ElementId	-	●	-	-	-	IN: first free identifier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

Size of bitmap is always size of lcd.

Bitmap will be ORed to other elements.

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

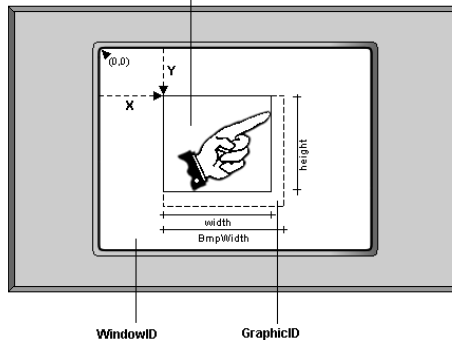


figure 38: Create and place graphic in a window

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_GRAPHIC_createWnd.TIG
'-----------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

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

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

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '*************************************
  ' INITIALIZATION
  '*************************************
  call bTglInit( blReturn )
  wlElementId = 0

  '*************************************
  ' TGL ELEMENTS AND WINDOWS
  '*************************************
  call bTglCreateGraphicWnd( &
  88, 27, &                      ' width, height of element
  dlHelloGraphic, 88, &          ' address, format width of bitmap
  wlElementId, 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
```

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 **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.*

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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

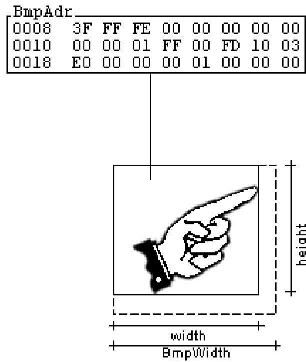


figure 39: Create graphic

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

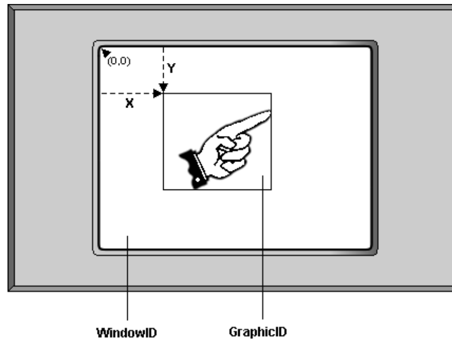


figure 40: Place graphic in window

It is possible to place one element in several windows. It is not allowed to place the same element twice or more in the same 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 **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.*

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.

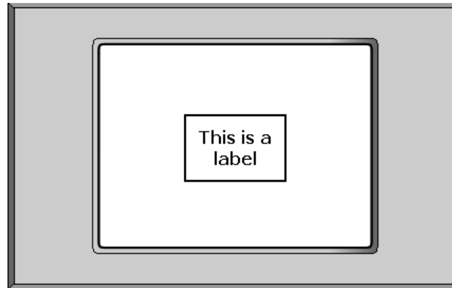


figure 41: Label

Subroutines

- *bTglCreateLabelWnd, bTglCreateLabelFWnd, bTglCreateLabelVarWnd, bTglCreateLabelDockWnd, bTglCreateLabelFDockWnd, bTglCreateLabelVarDockWnd*
- *bTglCreateLabel, bTglCreateLabelF, bTglCreateLabelVar*
- *bTglPlaceLabelInWindow, bTglDockLabelInWindow*

See also:

- *bTglLink*
- *bTglSetAttribute*

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.

bTglCreateLabelWnd, bTglCreateLabelFWnd, bTglCreateLabelVarWnd

```
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 thickness of frame
ElementId	-	●	-	-	-	unique identifier of new element
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates in window
						Return Values:
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 choosen font, the text length and the frame thickness.

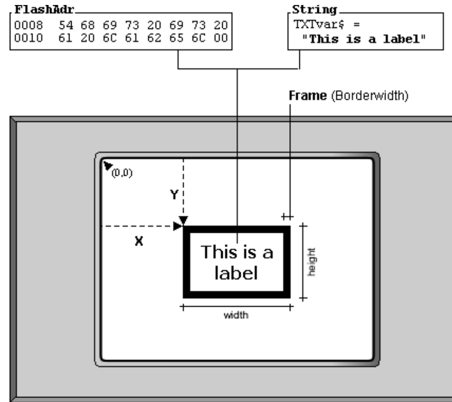


figure 42: Create and place label in a window

Sample program:

```
'--------------------------------------------------------------------------
' TGL_LABEL_createWnd.TIG
'--------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

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

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

   #include TGL_DEVICE_DRIVERS_TP1000.INC

  '*************************************
  ' INITIALIZATION
  '*************************************
  call bTglInit( blReturn )
  wlElementId = 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
  wlElementId, 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
```

bTglCreateLabelDockWnd, bTglCreateLabelFDockWnd, bTglCreateLabelVarDockWnd

```
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 thickness 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.

bTglCreateLabel, bTglCreateLabelF, bTglCreateLabelVar

```
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 thickness of frame
ElementId	-	●	-	-	-	unique identifier of this element
						Return Values:
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 choosen font, the text length and the frame thickness.

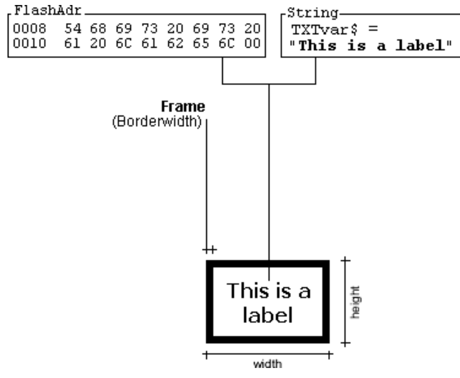


figure 43: Create a label

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

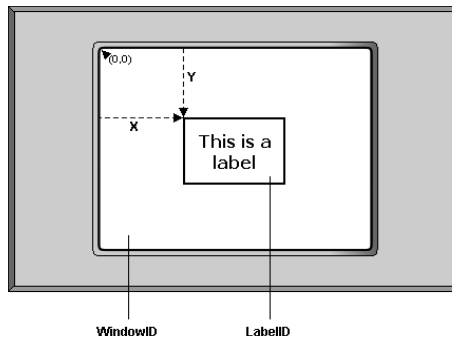


figure 44: Place label in a window

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
						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.*

Button

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.

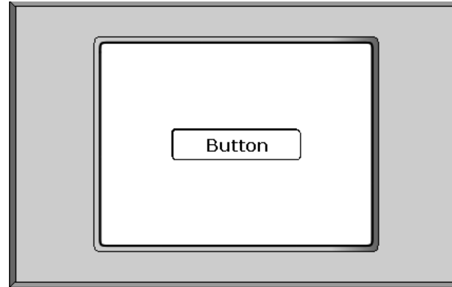


figure 45: Button

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 46: Unpressed button bitmap

figure 47: Pressed button bitmap

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

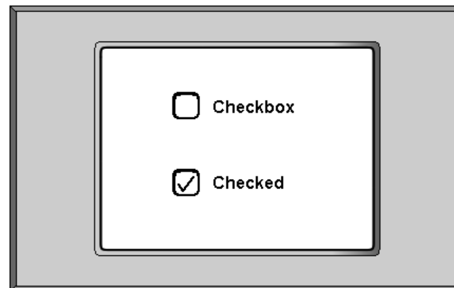


figure 48: Checkbox

Available subroutines for buttons:

- *wTglInitPushButtonWnd, wTglInitPushButton*
- *wTglInitSwitchButtonWnd, wTglInitSwitchButton*
- *bTglCreateButtonWnd, bTglCreateButtonWndS, bTglCreateButtonDockWnd, bTglCreateButtonDockWndS*
- *bTglCreateButton*
- *bTglPlaceButtonInWindow, bTglPlaceButtonInWindowS, bTglDockButtonInWindow, bTglDockButtonInWindowS*

- *bTglGetKeycode*
- *bTglWaitKeycode*
- *bTglGetPushButtonState*
- *bTglSetButtonState, bTglGetButtonState*

See also:

- *bTglLink*
- *bTglSetAttribute*

Push Button

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 )
```

figure 49: Unpressed button bitmap

figure 50: Pressed button bitmap

See the initializations subroutines *wTglInitPushButton* and *wTglInitPushButtonWnd* for an easy way of creating of these elements in one call.

Switch Button

Buttons can be used as switches, if you set the parameter *KeyAttr* in *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.

Read out current state:

```
call bTglGetButtonState( &
  wlElementId, &        ' 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( &
  wlElementId, & ' 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( &
  wlElementId, & ' 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 )
```

See the initializations subroutines *wTglInitSwitchButton* and *wTglInitSwitchButtonWnd* for an easy way of creating of these elements in one call.

wTglInitPushButtonWnd

call wTglInitPushButtonWnd(Width, Height, BmpAddr, BmpAddr2 WindowId, X,Y, Code, ElementId, Result)

Function: Creates a push button which changes bitmaps during pressing time and places it to a window at position X/Y. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpAddr2	-	-	●	-	-	address of alternative bitmap TGL_NO_ADDR: button inverts while pressed
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Code	●	-	-	-	-	binary keycode
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create place and link for buttons in one go. For the alternative bitmap an element of type graphic will be created.

The format width of the saved bitmap will be calculated internally as the next multiple of 8 to wpWidth. If the bitmap(s) should be rotated for the element afterwards for 90° or 270° by calling bTglSetAttribute() the calculated format width can differ to the existing format width. This has no effect for the correct working of the TGL subroutines and may be ignored, if the saved bitmap has the format width of the next multople of 8. If not set the correct format width by calling the setter bTglSetBmpWidth. The getter wTglGetBmpWidth will return the saved value.

Default for key attributes is autorepeat off and beep on.

If TGL_NO_ADDR is passed for BmpAddr2 the button inverts while it is pressed instead of showing an alternative bitmap.

wTglInitPushButton

call wTglInitPushButton(Width, Height, BmpAddr, BmpAddr2, ElementId, Result)

Function: Creates a push button which changes bitmaps during pressing time. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpAddr2	-	-	●	-	-	address of alternative bitmap TGL_NO_ADDR: button inverts while pressed
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create and link for buttons in one go. For the alternative bitmap an element of type graphic will be created.

!

The format width of the saved bitmap will be calculated internally as the next multiple of 8 to wpWidth. If the bitmap(s) should be rotated for the element afterwards for 90° or 270° by calling bTglSetAttribute() the calculated format width can differ to the existing format width. This has no effect for the correct working of the TGL subroutines and may be ignored, if the saved bitmap has the format width of the next multople of 8. If not set the correct format width by calling the setter bTglSetBmpWidth. The getter wTglGetBmpWidth will return the saved value.

Default for key attributes is autorepeat off and beep on.

If BmpAddr2= TGL_NO_ADDR the button inverts while it is pressed.

wTglInitSwitchButtonWnd

call wTglInitSwitchButtonWnd(Width, Height, BmpAddr, BmpAddr2 WindowId, X,Y, ElementId, Result)

Function: Creates a switch button which changes bitmaps after each pressing and places it to a window at position X/Y. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpAddr2	-	-	●	-	-	address of alternative bitmap
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create place and link for buttons in one go. For the alternative bitmap an element of type graphic will be created.

!

The format width of the saved bitmap will be calculated internally as the next multiple of 8 to wpWidth. If the bitmap(s) should be rotated for the element afterwards for 90° or 270° by calling bTglSetAttribute() the calculated format width can differ to the existing format width. This has no effect for the correct working of the TGL subroutines and may be ignored, if the saved bitmap has the format width of the next multople of 8. If not set the correct format width by calling the setter bTglSetBmpWidth. The getter wTglGetBmpWidth will return the saved value.

Default for key attributes is autorepeat off and beep on. The keycode for switch buttons will be administrated internally.

The switch state can be controlled by the subroutines *bTglGetButtonState* and *bTglSetButtonState.*

wTglInitSwitchButton

call wTglInitSwitchButton(Width, Height, BmpAddr, BmpAddr2, ElementId, Result)

Function: Creates a switch button which changes bitmaps after each pressing. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
BmpAddr	-	-	●	-	-	address of the button bitmap in flash memory
BmpAddr2	-	-	●	-	-	address of alternative bitmap
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create and link for buttons in one go. For the alternative bitmap an element of type graphic will be created.

The format width of the saved bitmap will be calculated internally as the next multiple of 8 to wpWidth. If the bitmap(s) should be rotated for the element afterwards for 90° or 270° by calling bTglSetAttribute() the calculated format width can differ to the existing format width. This has no effect for the correct working of the TGL subroutines and may be ignored, if the saved bitmap has the format width of the next multople of 8. If not set the correct format width by calling the setter bTglSetBmpWidth. The getter wTglGetBmpWidth will return the saved value.

Default for key attributes is beep on.

The switch state can be controlled by the subroutines *bTglGetButtonState* and *bTglSetButtonState.*

bTglCreateButtonWnd, bTglCreateButtonWndS

```
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: mode 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.

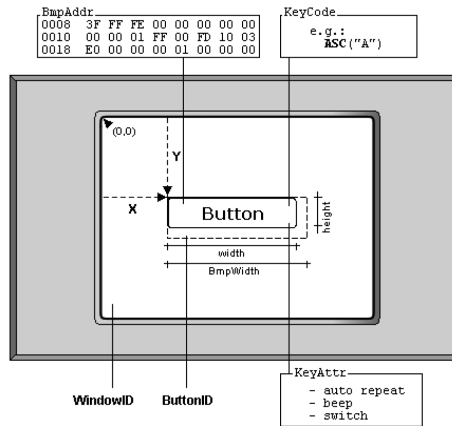


figure 51: Create and place button in a window

Sample program:

```
'-------------------------------------------------------------------------
' TGL_BUTTON_createWnd.TIG
'-------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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

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

bTglCreateButtonDockWnd, bTglCreateButtonDockWndS

```
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: mode 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

	B	W	L	S	F	
Result	●	-	-	-	-	**Return Values:** 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
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
  wlElemIdTmp = wlElementId
```

```
  wlElementId = wlElementId + 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
  wlElemIdTmp, &                  ' identifier of existing element (parent)
  wlElementId, &                  ' 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, wlIbuFill   ' get buffer length
    if wlIbuFill > 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
```

bTglCreateButton

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

Function: Creates a new bitmap button resp. switch.

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: mode 0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
						Return Values:
Result	●	-	-	-	-	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.

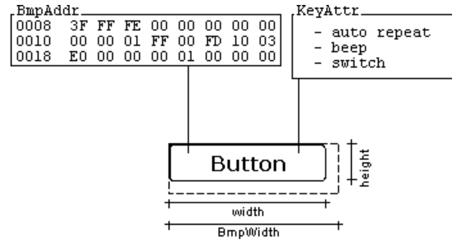


figure 52: Create button

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_ON TGL_KEY_ATTR_AUTOREPEAT_OFF	(No) Filling the buffer using auto repeat mode
TGL_KEY_ATTR_BEEP_ON TGL_KEY_ATTR_BEEP_OFF	Button generate (not) a beep (key click)
TGL_KEY_ATTR_STANDARD TGL_KEY_ATTR_SWITCH	Button is used as a standard button (=push button) or switch button(e.g. checkbox)
TGL_KEY_ATTR_DEFAULT	autorepeat on, beep on, push button

Create a button with default key attributes:

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

Create a switch button:

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

Turn off beep:

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

Turn off auto repeat:

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

For a combination of kea attributes the difines ca be ORed in a variable and passed afterwards.

Create Switch button without beep:

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

Turn off auto repeat and beep:

```
blKeyAttr = TGL_KEY_AUTOREPEAT_OFF bitor TGL_KEY_BEEP_OFF
call bTglCreateButton(
31, 16, &                 ' button size
dlButton, 32, &           ' address and format width of bitmap
blKeyAttr, &              ' key attributes: No autorepeat / No beep
wpvElementId, &           ' unique ID of this button
bpvReturn )               ' Return Code (0: OK Exit  >0: Error Code)
```

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

A hidden button can be created by passing the parameters *TGL_NO_ADDR* and *TGL_NO_BMP* with the creation.

```
call bTglCreateButtonWnd(
31, 16, &                ' button size
TGL_NO_ADDR, TGL_NO_BMP, &' address and format width of bitmap
TGL_KEY_ATTR_DEFAULT, &          ' key attributes: No autorepeat / No beep
wlElementId, &           ' unique ID of this button
blReturn )               ' Return Code (0: OK Exit  >0: Error Code)
```

bTglPlaceButtonInWindow, bTglPlaceButtonInWindowS

```
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
						Return Values:
Result	●	-	-	-	-	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.

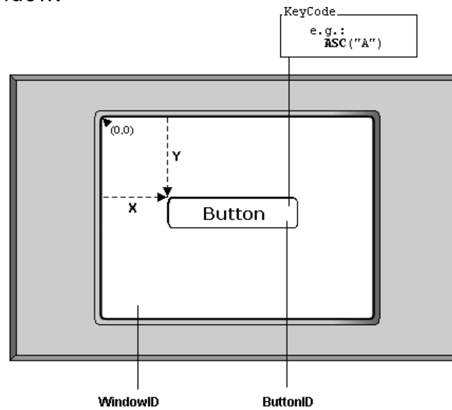


figure 53: Place a button in a window

Sample program:

```
'-----------------------------------------------------------------------------'
' TGL_BUTTON_create_place.TIG
'-----------------------------------------------------------------------------'
#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 bTglInit( blReturn )
  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
  '*********************************
  ' FLASH
  '*********************************
  BUTTON::
  data filter "btn_Solo.bmp", "GRAPHFLT", 0     ' WxH=31x16  bmpW=32
end
```

bTglDockButtonInWindow, bTglDockButtonInWindowS

```
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
						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.*

Sample program:

```
'-----------------------------------------------------------------------
' TGL_BUTTON_create_dock.TIG
'-----------------------------------------------------------------------
#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 bTglInit( blReturn )
  wlElementId = 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
  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
  wlElemIdTmp = wlElementId
  wlElementId = wlElementId + 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
  wlElementId, &                ' 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( &
  wlElemIdTmp, &                ' identifier of existing element (parent)
  wlElementId, &                ' 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, wlIbuFill    ' get buffer length
    if wlIbuFill > 0 then                        ' check input length of
buffer
      get #TP, #0, 1, blKeycode                  ' get keycode
    endif
  endwhile

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

bTglSetKeyAttributes

call bTglSetKeyAtrributes(ElementId, KeyAttr, Result)

Function: Set the key attributes auto repeat, beep and mode.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of this element
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off bit-6: mode 0: standard button 1: switch button
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

bTglGetKeyAttributes

```
call bTglGetKeyAtrributes( ElementId, KeyAttr, Result )
```

Function: Get the key attributes auto repeat, beep and mode.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	identifier of this element
						Return Values:
KeyAttr	●	-	-	-	-	key attributes for touch panel driver bit-4: auto repeat 0=on, 1=off bit-5: beep 0=on, 1=off bit-6: mode 0: standard button 1: switch button
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

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 wlIbuFill
for ever=0 to 0 step 0
  call bTglGetKeycode( blKeycode, wlIbuFill )
  if 0 < wlIbuFill 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
```

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
						Return Values:
State	●	-	-	-	-	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*.

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 TGL_STANDARD = not pressed TGL_ALTERNATIVE = pressed
						Return Values:
Result	●	-	-	-	-	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 TGL_STANDARD or TGL_ALTERNATIVE . To read out the current state, please use the function *bTglGetButtonState*.

Example program:

```
'------------------------------------------------------------------------------
' TGL_SWITCH_set_state.TIG
'------------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
  wlElementId, &                 ' id of graphic with alternative bitmap
  blReturn )                     ' return code (0: OK exit  >0: error exit)

  '***********************************
```

```
  ' switch between button states
  '************************************
  call bTglShowWindow( WINDOW_ID, blReturn )
  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
    blReturn )                   ' 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
    blReturn )                   ' 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
```

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
						Return Values:
State	●	-	-	-	-	current state of switch button TGL_STANDARD = not pressed TGL_ALTERNATIVE = pressed
Result	●	-	-	-	-	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 TGL_STANDARD or TGL_ALTERNATIVE. To set the state manually, please use the function *bTglSetButtonState*.

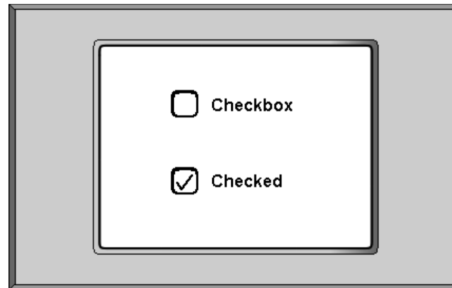


figure 54: Switch used as a checkbox

Example program:

```
'-----------------------------------------------------------------------------
' TGL_SWITCH_get_state.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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 1=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
```

Text Button

Text buttons are rectangular areas in a window containing a text graphic and having touch panel functionality.

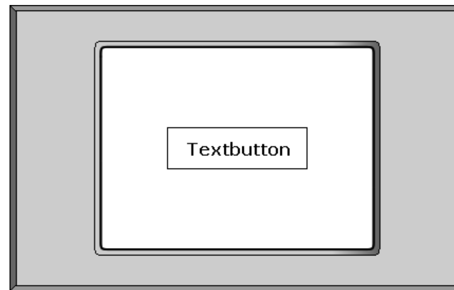


figure 55: Touch panel button with graphic text

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:

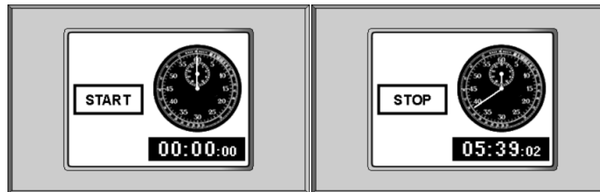


figure 56: Start/stop button

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.

Available subroutines for text buttons:

- *wTglInitPushTextButtonWnd, wTglInitPushTextButtonFWnd, wTglInitPushTextButton, wTglInitPushTextButtonF*
- *wTglInitSwitchTextButtonWnd, wTglInitSwitchTextButtonFWnd, wTglInitSwitchTextButton, wTglInitSwitchTextButtonF*
- *bTglCreateTextButtonWnd, bTglCreateTextButtonFWnd, bTglCreateTextButtonVarWnd bTglCreateTextButtonWndS, bTglCreateTextButtonFWndS, bTglCreateTextButtonVarWndS, bTglCreateTextButtonDockWnd, bTglCreateTextButtonFDockWnd, bTglCreateTextButtonVarDockWnd, bTglCreateTextButtonDockWndS, bTglCreateTextButtonFDockWndS, bTglCreateTextButtonDockVarWndS*
- *bTglCreateTextButton, bTglCreateTextButtonF, bTglCreateTextButtonVar*
- *bTglPlaceTextButtonInWindow, bTglPlacetextButtonInWindowS, bTglDockTextButtonInWindow, bTglDockTextButtonInWindowS*

See also:

- *bTglGetKeycode*
- *bTglWaitKeycode*
- *bTglGetPushButtonState*
- *bTglSetButtonState, bTglGetButtonState*
- *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, &
blFONT_ID, wlBUTTON_ID, WINDOW_ID, blReturn )
```

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, &
  FONT_ID, wpvElementId, wpWindowId, bpvReturn )
```

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.

wTglInitPushTextButtonWnd, wTglInitPushTextButtonFWnd

```
call wTglInitPushTextButtonWnd(   Width, Height, Text,       Text2,       Font, Frame,
                                  WindowId, X,Y, Keycode, ElementId,  Result )
call wTglInitPushTextButtonFWnd(  Width, Height, Textaddr,Textaddr2, Font, Frame,
                                  WindowId, X,Y, Keycode, ElementId, Result )
```

Function: Creates a push text button which changes bitmaps during pressing and places it to a window at position X/Y. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line thickness of frame
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
Keycode	●	-	-	-	-	button return code
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create place and link for buttons in one go. For the alternative text an element of type label will be created.

Default for key attributes are autorepeat off and beep on.

If the alternative pushed state should be inverted instead of an alternative text pass TGL_NO_TEXT with Text2$ resp. TGL_NO_ADDR with TextAddr.

wTglInitPushTextButton, wTglInitPushTextButtonF

call wTglInitPushTextButton(	Width, Height, Text,	Text2,	Font, Frame,	
	ElementId, Result)			
call wTglInitPushTextButtonF(	Width, Height, Textaddr,Textaddr2,		Font, Frame,	
	ElementId, Result)			

Function: Creates a push text button which changes texts after each pressing. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line thickness of frame
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements
						OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						>0 error

This initialization subroutine assembles tgl subroutines create and link for buttons in one go. For the alternative text an element of type label will be created.

Default for key attributes is beep on.

wTglInitSwitchTextButtonWnd, wTglInitSwitchTextButtonFWnd

```
call wTglInitSwitchTextButtonWnd( Width, Height, Text,      Text2,      Font, Frame,
                                  WindowId, X,Y, ElementId,  Result )
call wTglInitSwitchTextButtonFWnd(Width, Height, Textaddr,Textaddr2, Font, Frame,
                                  WindowId, X,Y ElementId, Result )
```

Function: Creates a switch text button which changes bitmaps after each pressing and places it to a window at position X/Y. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line thickness of frame
WindowId	-	●	-	-	-	identifier of window
X, Y	-	●	-	-	-	coordinates on LCD
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

This initialization subroutine assembles tgl subroutines create place and link for buttons in one go. For the alternative bitmap an element of type label will be created.

Default for key attributes is autorepeat off and beep on. The keycode for switch buttons will be administrated internally.

The switch state can be controlled by the subroutines *bTglGetButtonState* and *bTglSetButtonState.*

If the Alternative switch state should be inverted instead of an alternative text pass *TGL_NO_TEXT* with Text2$ resp. *TGL_NO_ADDR* with TextAddr.

wTglInitSwitchTextButton, wTglInitSwitchTextButtonF

```
call wTglInitSwitchTextButton(  Width, Height, Text,       Text2,       Font, Frame,
                                ElementId,  Result )
call wTglInitSwitchTextButtonF(Width, Height, Textaddr,Textaddr2, Font, Frame,
                                ElementId, Result )
```

Function: Creates a switch text button which changes texts after each pressing. Inrements the identifier for 2 created elements.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of button
Text	-	-	-	●	-	text in text button
TextAddr	-	-	●	-	-	address of the text in the flash memory
FontId	●	-	-	-	-	identifier of font
Frame	●	-	-	-	-	line thickness of frame
						Return Values:
ElementId	-	●	-	-	-	IN: first free identfier for these elements OUT: next free identifier for further elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

This initialization subroutine assembles tgl subroutines create and link for buttons in one go. For the alternative bitmap an element of type label will be created.

Default for key attributes is beep on.

The switch state can be controlled by the subroutines *bTglGetButtonState* and *bTglSetButtonState.*

bTglCreateTextButtonWnd, bTglCreateTextButtonFWnd, bTglCreateTextButtonVarWnd, bTglCreateTextButtonWndS, bTglCreateTextButtonFWndS, bTglCreateTextButtonVarWndS

```
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 thickness 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: mode 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

! 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.

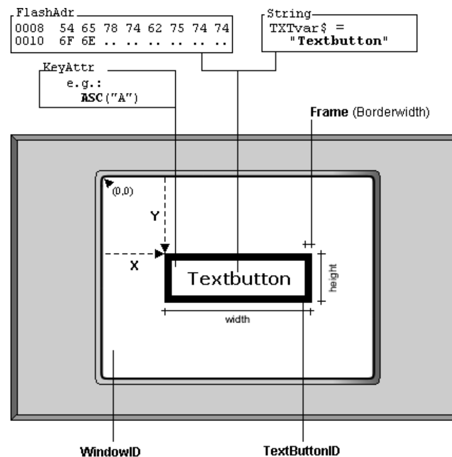


figure 57: Create and place a textbutton in a window

bTglCreateTextButtonDockWnd, bTglCreateTextButtonFDockWnd, bTglCreateTextButtonVarDockWnd, bTglCreateTextButtonDockWndS, bTglCreateTextButtonFDockWndS, bTglCreateTextButtonVarDockWndS

```
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)

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 thickness 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: mode 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
						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.*

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_TEXTBUTTON.TIG
'-----------------------------------------------------------------------------
#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 wlElementId       ' current identifier for creation of elements
  word wlLABEL_ID        ' "constant" identifier of label
  byte blFontId

   #include TGL_DEVICE_DRIVERS_TP1000.INC

  '*************************************
  ' INITIALIZATION
  '*************************************
  call bTglInit( blReturn )
  wlElementId = 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
  wlElementId, 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
  wlElementId = wlElementId + 1

  call bTglCreateLabelVarWnd( &
  120, 40,&                        ' width, height of element
  blFontId, 0, &                   ' identifier of font, frame thickness
```

```
  wlElementId, 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
  wlLABEL_ID = wlElementId

  '*************************************
  ' show and erase button code in label
  '*************************************
  call bTglShowWindow( WINDOW_ID, blReturn )
  loop 7FFFFFFFh

    ' display code of pressed button
    call vWaitForPressedButton()
    get #TP, #0, 1, slKeyCode$          ' get single button code
    slLcdOutput$ = "Button returns: " + slKeyCode$
    call bTglSetText( wlLABEL_ID, slLcdOutput$, blReturn )
    call bTglShow( wlLABEL_ID, WINDOW_ID, blReturn )

    ' erase displayed code of pressed button
    call vWaitForPressedButton()
    get #TP, #0, 0, slKeyCode$          ' read out buffer
    call bTglSetText( wlLABEL_ID, TGL_NO_TEXT, blReturn )
    call bTglShow( wlLABEL_ID, WINDOW_ID, blReturn )
  endloop
end

sub vWaitForPressedButton()
  word wlIbuFill       ' filling of input buffer of touch panel
  wlIbuFill = 0
  get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
  while wlIbuFill = 0
    release_task
    get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
  endwhile
end
```

bTglCreateTextButton, bTglCreateTextButtonF, bTglCreateTextButtonVar

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 thickness 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: mode 0: standard button 1: switch button
ElementId	-	●	-	-	-	unique identifier of this element
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

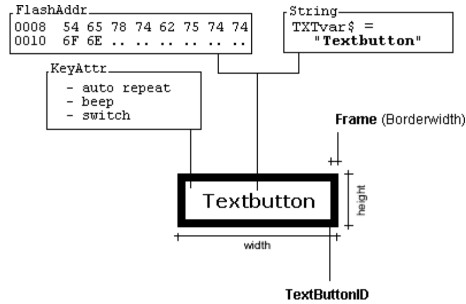


figure 58: Create text button

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 choosen 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, bTglPlaceTextButtonInWindowS

```
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

FlashAddr
0008 54 65 78 74 62 75 74 74
0010 6F 6E .. ..

String
TXTvar$ =
"Textbutton"

KeyCode
e.g.:
ASC("A")

(0,0)
Y
X
Textbutton

WindowID
TextButtonID

figure 59: Place text button in a window

bTglDockTextButtonInWindow, bTglDockTextButtonInWindowS

```
call bTglDockTextButtonInWindow(    ParentId, ElementId, WindowId, &
                                    XRel, YTRel, 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
						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 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.

Slider

Sliders are touch elements for a continuously 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:

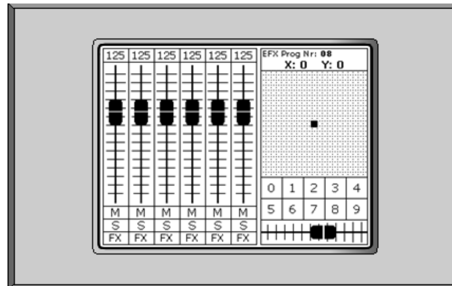


figure 60: Example for sliders

Available subroutines for sliders:

- *wTglInitSliderWnd, wTglInitSlider*
- *bTglCreateSliderWnd, bTglCreateSliderDockWnd*
- *bTglCreateSlider*
- *bTglPlaceSliderInWindow, bTglDockSliderInWindow*
- *lTglGetSliderValue, bTglSetSliderValue*
- *bTglSetSliderLimits*

See also:

- *bTglLink*
- *Graphic*

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.

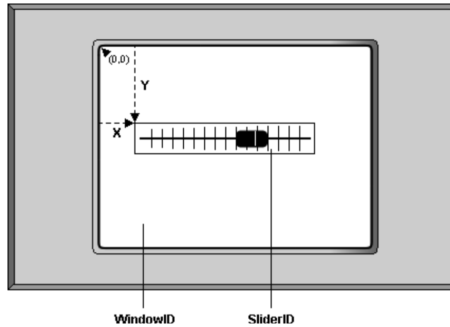


figure 61: X-Slider

This is an example of a Y-Slider. You can change the value by sliding along the y-axis.

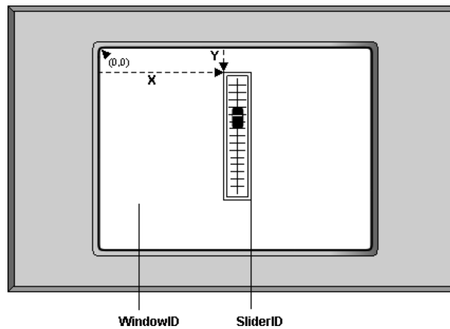


figure 62: Y-Slider

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, &          ' dummys for linear slider types
"Y", &           ' type of slider
dlSlidebar, 32,&' address, format width of bitmap
wpvElementId,&   ' unique ID of this slidebar
wpWindowId, &    ' window ID to create this slider in
10, 20 &         ' X,Y coordinate on LCD
bpvReturn )      ' 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. Alternatively you can call the initialization subroutines *wTglInitSlider* or *wTglInitSliderWnd*.

Example for creating a graphic:

```
'********************************************************
' create graphic for slider
'********************************************************
call bTglCreateGraphic( &
16,30, &                ' width, height of graphic
dlSliderButton, 16, & ' address, format width of bitmap
wpvElementId, &         ' unique ID of this graphic
bpvReturn )             ' Return Code (0: OK Exit  >0: Error Code)
```

Link graphic to slidebar:

```
'********************************************************
' link graphic to slidebar
'********************************************************
call bTglLink( &
wgEID_SLIDEBAR, &       ' unique ID of slidebar
wgEID_SLIDERBUTTON, & ' unique ID of slider button graphic
bpvReturn )             ' Return Code (0: OK Exit  >0: Error Code)
```

Please notice, that a linked graphic will reduce the touch area of the slidebar. For an X-Slider, the half of the slider button width will be cut from the left and the right side of the slidebar. The Y-Slider will be scaled down at the top and the bottom. Please see the following graphics:

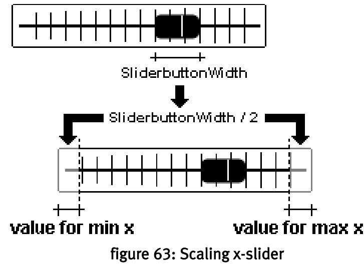


figure 63: Scaling x-slider

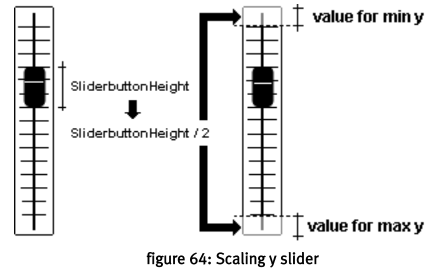


figure 64: Scaling y slider

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( &
wgEID_SLIDEBAR, &         ' unique ID of slidebar
wpvWindowId, &            ' unique ID of window
TGL_SL_OPT_VALUE, &       ' option: read out current value of slider
llSliderValue, &          ' Return value: current value of slider
bpvReturn )               ' Return Code (0: OK Exit  >0: Error Code)
```

wTglInitSliderWnd

call wTglInitSliderWnd(Width, Height, BmpAddr, BmpWidth, MinXY, MaxXY, &
WBut, HBut, BmpAddrBut, BmpWidthBut, &
WindowId, X, Y, ElementId, Result)

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

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of slidebar
BmpAddr	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidth	-	●	-	-	-	bitmap with slidebar (multiple of 8!)
MinXY, MaxXY	-	-	●	-	-	value at minimum, maximum coordinate
WBut, HBut	-	●	-	-	-	size of slider button
BmpAddrBut	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidthBut	-	●	-	-	-	bitmap with slider button (multiple of 8!)
WindowId	-	●	-	-	-	unique identifier of window
X, Y	-	●	-	-	-	coordinates of left top edge
						Return Values:
ElementId	-	●	-	-	-	IN: first free identifier for these elements OUT: free identifier for next element
Result	●	-	-	-	-	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!

BmpAddrBut = TGL_NO_ADDR creates a slider without button.

The orientation of the slider is determined by the higher value of width and height. Width ‹= height means an Y-slider otherwise an X-slider.

wTglInitSlider

call wTglInitSliderWnd(Width, Height, BmpAddr, BmpWidth, MinXY, MaxXY, &
WBut, HBut, BmpAddrBut, BmpWidthBut, ElementId, Result)

Function: Creates a slidebar incl. a linked slider button.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of slidebar
BmpAddr	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidth	-	●	-	-	-	bitmap with slidebar (multiple of 8!)
MinXY, MaxXY	-	-	●	-	-	value at minimum, maximum coordinate
WBut, HBut	-	●	-	-	-	size of slider button
BmpAddrBut	-	-	●	-	-	address of the slide bar bitmap in flash memory
BmpWidthBut	-	●	-	-	-	bitmap with slider button (multiple of 8!)
						Return Values:
ElementId	-	●	-	-	-	IN: first free identifier for these elements OUT: free identifier for next element
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

The orientation of the slider is determined by the higher value of width and height. Width ‹= height means an Y-slider otherwise an X-slider.

BmpAddrBut = TGL_NO_ADDR creates a slider without button.

You can use this slider element in several times in different windows, but never more than 1 time in 1 window!

bTglCreateSliderWnd

```
call bTglCreateSliderWnd( Width, Height, MinXY, MaxXY, 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, Height	-	●	-	-	-	size of slidebar
MinXY, MaxXY	-	-	●	-	-	value at minimum, maximum coordinate
Dummy	-	-	●	-	-	dummy value for linear sliders
XYFlag	-	-	-	●	-	"X": x-coordinate slidebar (left-right) "Y": y-coordinate slidebar (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, Y	-	●	-	-	-	coordinates of left top edge
						Return Values:
Result	●	-	-	-	-	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*. Alternatively you can call *wTglInitSlider*.

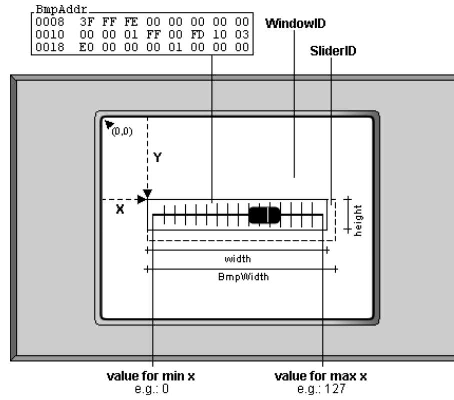


figure 65: Create and place slider in a window

bTglCreateSliderDockWnd

```
call bTglCreateSliderDockWnd(Width, Height, MinXY, MaxXY, 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
MinXY, MaxXY	-	-	●	-	-	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 **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*.

Sample program:

```
'-----------------------------------------------------------------------
' TGL_SLIDER_Y_dockWnd.TIG
'-----------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'*************************************
' IDENTIFIER
'*************************************
' elements
#define SLIDERBAR_ID            0
#define SLIDERBUTTON_ID         1
#define SLIDERBAR_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 bTglInit( blReturn )

  '*******************************************************
  ' create slidebar
  '*******************************************************
  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
        SLIDERBAR_ID, & ' unique ID of this slidebar
        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 slidebar
'*******************************************************
call bTglLink(SLIDERBAR_ID, & ' unique ID of slidebar
SLIDERBUTTON_ID, &     ' unique ID of slider button graphic
blReturn)              ' Return Code (0: OK Exit  >0: Error Code)

'*******************************************************
' create slidebar
'*******************************************************
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
      SLIDERBAR_ID, & ' unique ID of the parent element
      SLIDERBAR_ID2, &' unique ID of this slidebar
      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 slidebar
'*******************************************************
call bTglLink(SLIDERBAR_ID2, & ' unique ID of slidebar
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(SLIDERBAR_ID, & ' unique ID of slidebar
  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

CHNL_SLIDEBAR::
data filter "chnl_slidebar.bmp", "GRAPHFLT", 0     ' WxH=31x170  BmpW=32
CHNL_SLIDER::
data filter "chnl_slider_black.bmp", "GRAPHFLT", 0 ' WxH=16x30   BmpW=16
end
```

bTglCreateSlider

```
call bTglCreateSlider( Width, Height, MinXY, MaxXY, Dummy, Dummy, XYFlag$, &
                       BmpAddr, BmpWidth, ElementId, Result )
```

Function: Creates a new slidebar with the identifier ElementId.

Parameters:

	B	W	L	S	F	
Width, Height	-	●	-	-	-	size of slidebar
MinXY, MaxXY	-	-	●	-	-	value at minimum, maximum coordinate
Dummy	-	-	●	-	-	dummy value for linear sliders
XYFlag	-	-	-	●	-	"X": x-coordinate slidebar (left-right) "Y": y-coordinate slidebar (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
						Return Values:
Result	●	-	-	-	-	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*. Alternatively you can call *wTglInitSlider*.

The parameter names Min and Max are not the If you need to create a rotated slider the easyest would be to swich the XYFlag. isualizing the current slider position, please create a graphic with the slider button and link the graphic to the slider with the function *bTglLink*. Alternatively you can call *wTglInitSlider*.

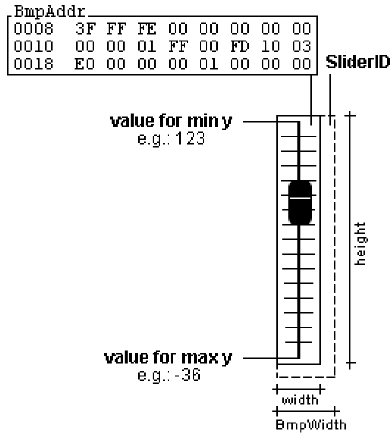


figure 66: Create y slider

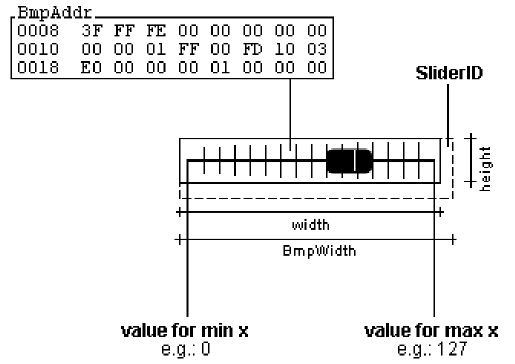


figure 67: Create x slider

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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						›0 error

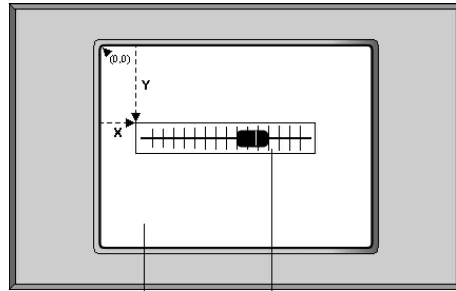


figure 68: Place an x slider in a window

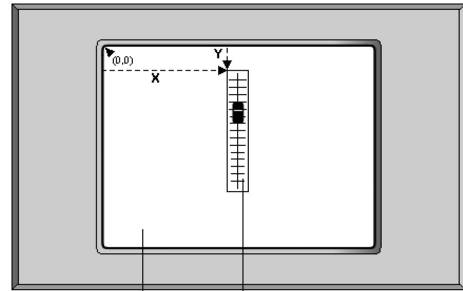


figure 69: Place an y-slider in a window

Sample program:

```
'------------------------------------------------------------------------
' TGL_SLIDER_Y_create_place.TIG
'------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'***************************************
' IDENTIFIER
'***************************************
' elements
#define SLIDEBAR_ID     0
#define SLIDERBUTTON_ID 1
' windows
#define WINDOW_ID       0

task main
  datalabel dlSlideBar, dlSliderButton
  byte blReturn
  long llSliderValue            ' current value of the slider

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '***********************************
  ' INITIALIZATION
  '***********************************
  call bTglInit( blReturn )

  '*************************************
  ' 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)
  dlSlideBar, 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
  dlSliderButton, 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 *bPlaceLabelInWindow*

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
						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.*

Sample program:

```
'-----------------------------------------------------------------------
' TGL_SLIDER_Y_create_dock.TIG
'-----------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'**************************************
' IDENTIFIER
'**************************************
' elements
#define SLIDERBAR_ID            0
#define SLIDERBUTTON_ID         1
#define SLIDERBAR_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 bTglInit( blReturn )

  '*******************************************************
  ' create slidebar
  '*******************************************************
  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
        SLIDERBAR_ID, & ' unique ID of this slidebar
        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 slidebar
  '*******************************************************
```

```
  call bTglLink(SLIDERBAR_ID, & ' unique ID of slidebar
  SLIDERBUTTON_ID, &    ' unique ID of slider button graphic
  blReturn)             ' Return Code (0: OK Exit  >0: Error Code)

  '********************************************************
  ' create slidebar
  '********************************************************
  call bTglCreateSlider( 31, & ' width of slider
        170, &          ' height of slider
        200, -200, &    ' minimum, maximum value 1
        200, -200, &    ' minimum, maximum value 2
        "Y", &          ' type of slider
        CHNL_SLIDEBAR, &' address of bitmap
        32, &           ' width of bitmap
        SLIDERBAR_ID2, &' unique ID of this slidebar
        blReturn )      ' Return Code (0: OK Exit  >0: Error Code)

  '********************************************************
  ' dock slider in window
  '********************************************************
  call bTglDockSliderInWindow( SLIDERBAR_ID, &  ' unique ID parent element
        SLIDERBAR_ID2, &' unique ID of this slidebar
        WINDOW_ID, &    ' window ID to create this slider in
        10, 0, &        ' X,Y coordinate offsets from docking point
        TGL_OPT_RIGHT, &' docking option
        blReturn )      ' Return Code (0: OK Exit  >0: Error Code)

  '********************************************************
  ' link graphic to slidebar
  '********************************************************
  call bTglLink(SLIDERBAR_ID2, & ' unique ID of slidebar
  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(SLIDERBAR_ID, & ' unique ID of slidebar
    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_slidebar.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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

lTglGetSliderValue

call lTglGetSliderValue(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.
						Return Values:
Value	-	-	●	-	-	requested value is saved in this variable
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

Returned coodinates are always for the unrotated LCD. A global LCD rotation by configuration has no influence on these returned values!

Sample program:

```
'----------------------------------------------------------------------
' TGL_SLIDER_X_SHOW_VALUE.TIG
'----------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
  wlElementId = wlElementId + 1
```

```
  ' create slider button
  call bTglCreateGraphic( &
  32, 16, &                      ' width, height of element
  dlSliderButton, 32, &          ' address, format width of bitmap
  wlElementId, &                 ' identifier of element
  blReturn )                     ' return code (0: OK exit  >0: error exit)
  ' link slider button to slide bar
  call bTglLink( &
  wlSLIDER_ID, &                 ' identifier of slide bar
  wlElementId,&                  ' identifier of slider button
  blReturn )                     ' return code (0: OK exit  >0: error exit)
  '' increment identifier for next element
  wlElementId = wlElementId + 1

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

  '*********************************************
  ' show current slider value
  '*********************************************
  call bTglShowWindow( WINDOW_ID, blReturn )
  loop 7FFFFFFFh
    call lTglGetSliderValue( &
    wlSLIDER_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)
    call bTglShowLong( wlLABEL_ID, llSliderValue, TRUE, blReturn )
    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, 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
Top, Bottom	-	-	●	-	-	limits y axis, dummys for x sliders
Left, Right	-	-	●	-	-	limits x axis, dummys for y sliders
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

The parameters *Left/Right* are dummys for X-sliders, and *Top/Bottom* for Y-Sliders!

Listbox

Listboxes are assembled elements of a view for the list items end two buttons or a slider for scrolling. Listboxes can be used for an easy user input of a determined selection of items.

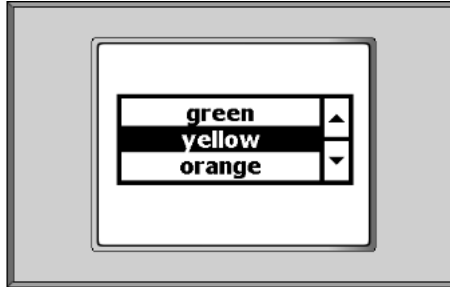


figure 70: Listbox

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:

- *bTglCreateListboxWnd, bTglCreateListBoxFWnd, bTglCreateListboxDockWnd, bTglCreateListboxFDockWnd*
- *bTglCreateListbox, bTglCreateListBoxF*
- *bTglPlaceListboxInWindow, bTglDockListboxInWindow*
- *sTglGetListboxItem*
- *wTglGetListboxIndex, bTglSetListboxIndex*

bTglCreateListboxWnd, bTglCreateListBoxFWnd

```
call bTglCreateListboxWnd(   Width, Height, List$, FontId, Frame, WidthBut, KeyAttr, &
                             ElementId, WindowId, X,Y, Result )
call bTglCreateListboxFWnd(  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
List	-	-	●	-	-	flash addr of 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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

bTglCreateListboxDockWnd, bTglCreateListBoxFDockWnd

```
call bTglCreateListboxDockWnd( Width, Height, List$, FontId, Frame, WidthBut, &
                               KeyAttr, ParentId, ChildId, WindowId, XRel, YRel, &
                               Option, Result )
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
List	-	-	●	-	-	flash addr of 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 **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.*

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_LISTBOX_createWnd.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  set_len$( slItem$, 0 )

  '********************************************************
  ' create font
  '********************************************************
  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, 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


  '********************************************************
  ' show window
  '********************************************************
```

```
  call bTglShowWindow( WINDOW_ID, blReturn )

  loop 7FFFFFFFh
    call sTglGetListboxItem( LISTBOX_ID, slItem$, blReturn )
    wait_duration 200
  endloop
end
```

bTglCreateListbox, bTglCreateListboxF

```
call bTglCreateListbox( Width, Height, List$, FontId, Frame, WidthBut, KeyAttr, &
                        ElementId, Result )
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
List	-	-	●	-	-	flash addr of 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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

The TGL listboxes are formatted like those of the Tiger BASIC functions *select$*/ *index$*. The first byte determines the separator. The list must be closed with a double separator.

```
list1$ = ",John,Mary,Harry,Mathilde,Nobody,,"
list2$ = "<0Dh>A<0Dh>B<0Dh>C<0Dh><0Dh>"
```

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

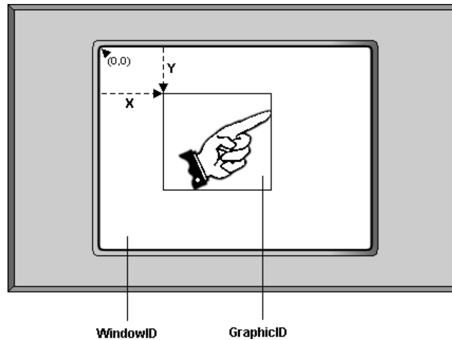


figure 71: Place listbox in window

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
						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.*

sTglGetListboxItem

```
call sTglGetListboxItem( ElementId, Item$, Return )
```

Function: Returns current item of listbox.

Parameters:

	B	W	L	S	F	
ElementId	-	●	-	-	-	unique identifier of element
						Return Values:
Item$	-	-	●	-	-	current item of list
Result	●	-	-	-	-	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
						Return Values:
Index	-	●	-	-	-	current index of list
Result	●	-	-	-	-	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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes
						0 ok
						›0 error

Sample program:

```
'------------------------------------------------------------------------------
' TGL_LISTBOX_set_get_index.TIG
'------------------------------------------------------------------------------
#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 bTglInit( blReturn )

  '******************************************************
  ' 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 7FFFFFFFh
    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.

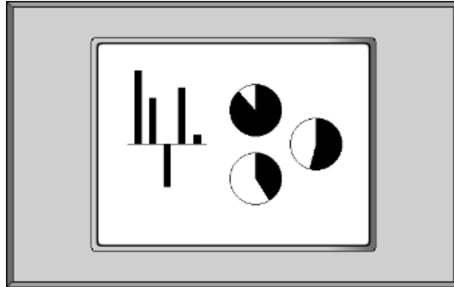


figure 72: Bar charts and pie charts

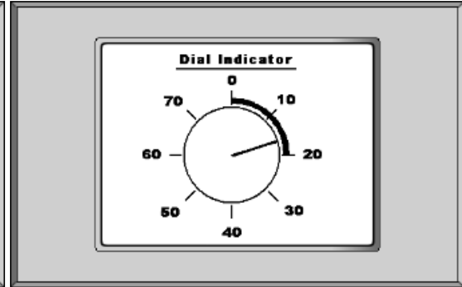


figure 73: Dial indicator

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:

- *bTglCreateGaugeWnd*
- *bTglCreateGaugeDockWnd*
- *bTglCreateGauge*
- *bTglPlaceGaugeInWindow*
- *bTglDockGaugeInWindow*
- *bTglShowGaugeValue*

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

Sample program:

```
'---------------------------------------------------------------------------
' TGL_GAUGE.TIG
'---------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

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

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

   #include TGL_DEVICE_DRIVERS_TP1000.INC

  '************************************
  ' INITIALIZATION
  '************************************
  call bTglInit( blReturn )
  wlElementId = 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
  wlElementId,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
```

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 **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.*

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
ElementId	-	●	-	-	-	unique identifier of element
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

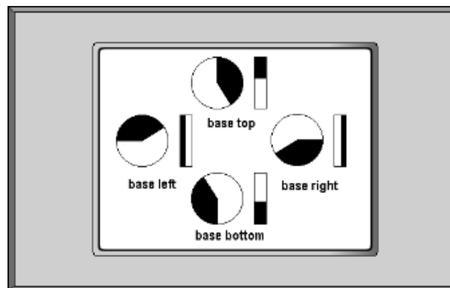


figure 74: Radial and linear gauge bases left, right, top, bottom

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
						Return Values:
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

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
						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.*

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

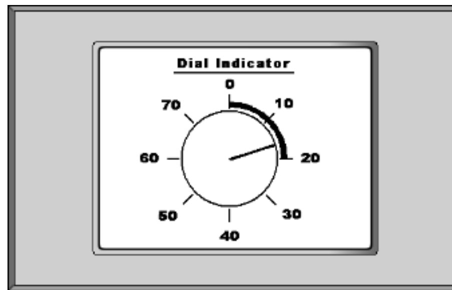


figure 75: Dial Indicator

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_GAUGE_dial_indicator.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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.

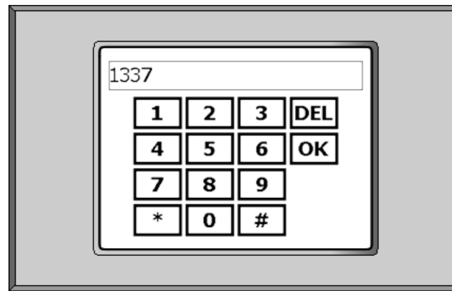


figure 76: Example for a keyboard

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:

- *wTglInitKeyboard*
- *wTglInitKeyboardSelfmade*
- *sTglGetKeyboardInput*
- *sTglGetKeybInputTimeout*
- *sTglEditText*
- *sTglGetKeybPassword*
- *sTglGetKeybPasswordTimeout*
- *sTglGetKeybParams*

wTglInitKeyboard

call wTglInitKeyboard(WindowId, Style, FontId, ElemIdKeybTxt, ElementId, Result)

Function: Initializes a bitmap keyboard for user input.

Parameters:

	B	W	L	S	F	
WindowId	-	●	-	-	-	identifier of the first window to place in the elements for the keyboard depending on KeyCodes$ more than one window may be filled
Style	-	●	-	-	-	look of keyboard
FontId	●	-	-	-	-	identifier for font of text in label for user input
						Return Values:
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
ElementId	-	●	-	-	-	IN: identifier of the first element of the keyboard OUT: first free identifier for the creation of next elements
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

There must be an existing font for a correct initialization of the keyboard. Call bTglCreateFont or pass parameters as follows with the creation of the font:

alignment horizontal	"left"
alignment vertical	"top"
spacing type	"prop"
spacing blank	0
spacing char	SPACING_CHAR_DEFAULT
spacing vertical	0
overlay mode	"imm"
wrap mode	"char"

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 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_KEYB1_STYLE_HEX	1	20 total 19 buttons 1 label	1

TGL_DIGSTYLE_1

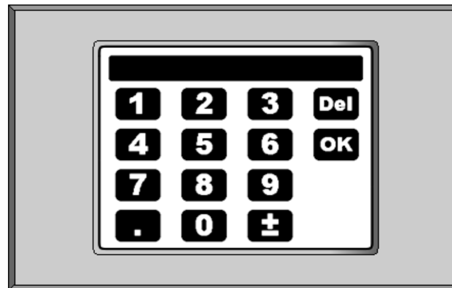


figure 77: Keyboard style digit block

TGL_KEYB1_STYLE_HEX:

figure 78: Keyboard style hex block

TGL_KEYB1_STYLE_ENG

figure 79: Keyboard style English unshifted

figure 80: Keyboard style English shifted

TGL_KEYB1_STYLE_GER

figure 81: Keyboard style German unshifted

figure 82: Keyboard style German shifted

TGL_KEYB1_STYLE_TRK:

figure 83: Keyboard style Turkey unshifted

figure 84: Keyboard style Turkey shifted

TGL_KEYB1_STYLE_HUN:

figure 85: Keyboard style Hungarian unshifted

figure 86: Keyboard style Hungarian shifted

Sample program:

```
'-----------------------------------------------------------------------------
' TGL_KEYBOARD.TIG
'-----------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'' window identifiers
#define WID_Text                0
#define WID_KeybUnshifted       1
#define WID_KeybShifted         2

'' font identifiers
#define FID_KeybView            0
#define FID_Text                1

'' element identifiers
word wgEID_KeybView

'' application specific definitions and global variables
#define DFLT_Text       "Please press text for a new text input!"
#define MAX_LEN_Text    80h
string sgInput$(MAX_LEN_Text)
#define REQUEST_Input   "Please input some text!"

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

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '**********************************
  ' INITIALIZATIONS
  '**********************************
  set_task_prio main, TGL_INIT_TASK_PRIO ' speed up for tgl initialization
  wlElementId = 0
  call bTglInit( blReturn )
  call wInitGeneral (              wlElementId, blReturn )
  call wTglInitKeyboard( WID_KeybUnshifted, TGL_KEYB1_STYLE_ENG &
  ,  FID_KeybView, wgEID_KeybView, wlElementId, blReturn )
  call wInitText(        WID_Text, wlElementId, blReturn )

  '**********************************
  ' START GUI
  '**********************************
  set_task_prio main, 1 ' reset task priority for usual running
  call bAdministrateWindows( blReturn )
end


'-----------------------------------------------------------------------------
' wInitGeneral:
'-----------------------------------------------------------------------------
' initialize fonts and general elements
'-----------------------------------------------------------------------------
' RETURN VALUES:
'       wpvElementId    IN:  first free identifier for these elements
'                       OUT: next free identifier for further elements
'       bpvReturn       error code: 0=OK, <0=ERROR
```

```
'------------------------------------------------------------------------------
sub wInitGeneral( var word wpvElementId; var byte bpvReturn )
  call bTglCreateFontParams( &
  FID_KeybView, &                 ' identifier of font
  "Valencia", 18, "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
  bpvReturn )                     ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif

  call bTglCreateFontParams( &
  FID_Text, &                     ' 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", "word", &                ' overlay, wrap mode
  bpvReturn )                     ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
end


'------------------------------------------------------------------------------
' wInitText:
'------------------------------------------------------------------------------
' empty window
'------------------------------------------------------------------------------
' PARAMETERS:
'       wpWindowId      identifier for window
' RETURN VALUES:
'       wpvElementId    IN:  first free identifier for these elements
'                       OUT: next free identifier for further elements
'       bpvReturn       error code: 0=OK, <0=ERROR
'------------------------------------------------------------------------------
#define KEY_Text        0
word wgEID_Text
sub wInitText( word wpWindowId; var word wpvElementId; var byte bpvReturn )
  wgEID_Text = wpvElementId
  call bTglCreateTextButtonWnd( &
  260, 180, &                     ' width, height of element
  DFLT_Text, &                    ' text in element
  FID_Text, 5, &                  ' text, font id, frame thickness
  TGL_KEY_ATTR_AUTOREPEAT_OFF, &' key attributes: auto repeat,beep,no switch
  wpvElementId, wpWindowId, &     ' identifier of text button, window
  30, 30, &                       ' x, y coordinate on lcd
  KEY_Text, &                     ' keycode
  bpvReturn )                     ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
end
'------------------------------------------------------------------------------
```

```
' wExecText:
'--------------------------------------------------------------------------
' Edit text
'--------------------------------------------------------------------------
' 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 wExecText( var word wpvWindowId; var byte bpvReturn )
  byte ever, blKeycode
  word wlIbuFill
  for ever = 0 to 0 step 0
    call bTglGetKeycode( blKeycode, wlIbuFill )
    if 0 < wlIbuFill then
      switchi blKeycode

      case KEY_Text:
        sgInput$ = REQUEST_Input
        set_task_prio main, 4 ' speed up task for keyboard
        call sTglGetKeyboardInput( WID_KeybUnshifted, wgEID_KeybView, &
        MAX_LEN_Text, sgInput$, bpvReturn )
        set_task_prio main, 1 ' reset task prio fur usual running
        if bpvReturn <> TGL_MSG_OK then
          return
        endif
        if 0 < len(sgInput$) then
          call bTglSetText( wgEID_Text, sgInput$, bpvReturn )
          if bpvReturn <> TGL_MSG_OK then
            return
          endif
        endif
        return

      endswitch ' blKeycode
    endif ' 0 < wlIbuFil
  next ' for ever
end


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

  wlWindowId = WID_Text
  for ever=0 to 0 step 0
    call bTglShowWindow( wlWindowId, bpvReturn ) ' no effect for this
example!
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
```

```
    switchi wlWindowId

    case WID_Text:
      call wExecText( wlWindowId, bpvReturn )

    default:
      bpvReturn = 0FFh
      return
    endswitch
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
  next
end
```

wTglInitKeyboardSelfmade

```
call wTglInitKeyboardSelfmade( WindowId, FontIdView, FontIdKey, Frame, Distance,&
                               X,Y, Width,Height, KeyCodes$, ElemIdKeybTxt,  &
                               ElementId, Result )
```

Function: Initializes a keyboard for user input with the given keys of visible characters (no control characters).

Parameters:

	B	W	L	S	F	
WindowId	-	●	-	-	-	identifier of the first window to place in the elements for the keyboard depending on KeyCodes$ more than one window may be filled
FonIdView	●	-	-	-	-	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:
ElemIdKeybTxt	-	●	-	-	-	identifier of the label for the user input
ElementId	-	●	-	-	-	IN: identifier of the first element of the keyboard OUT: first free identifier for the creation of next elements
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 and the returned identifier may NOT be used for other elements or windows. They are reserved for the 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$*

```
slKeyCodes$ = "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
```

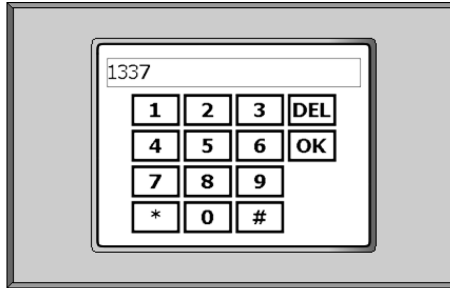


figure 87: Example 1 for selfmade keyboard

Example 2 *KeyCodes$*

```
slKeyCodes$ =     "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
```

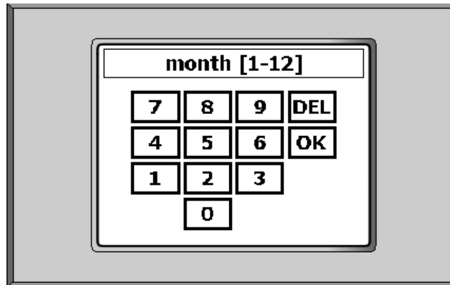


figure 88: Example 2 for selfmade keyboard

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$(_NEXT_ROW) & ' row 1
+             "qwertzuiopü+"                      +chr$(_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"
```

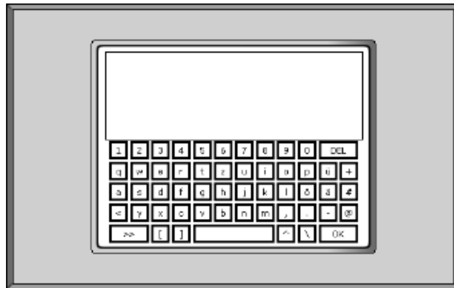


figure 89: unshifted keys

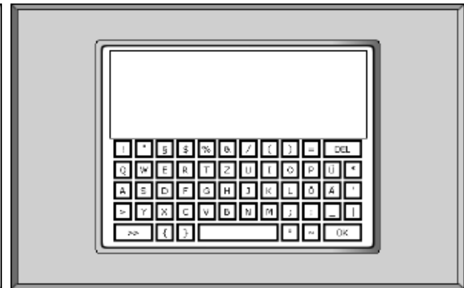


figure 90: shifted keys

Sample program:

```
'-----------------------------------------------------------------------------
TGL_KEYBOARD_selfmade.TIG
'-----------------------------------------------------------------------------
#include TigerGraphicLibrary.INC

'' window identifiers
#define WID_Text                0
#define WID_Keyboard            1

'' font identifiers
#define FID_KeybView            0
#define FID_KeybKey             1
#define FID_Text                2

'' element identifiers
word wgEID_KeybView

'' application specific definitions and global variables
#define DFLT_Text       "Please press for telephone number input!"
#define MAX_LEN_Text    20
string sgInput$(MAX_LEN_Text)
#define REQUEST_Input   "Please input number!"

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

  #include TGL_DEVICE_DRIVERS_TP1000.INC

  '**********************************
  ' INITIALIZATIONS
  '**********************************
  set_task_prio main, TGL_INIT_TASK_PRIO ' speed up for tgl initialization
  call bTglInit( blReturn )
  wlElementId = 0
  blReturn    = TGL_MSG_OK
  call wInitGeneral (               wlElementId, blReturn )
  call wInitKeyboard( WID_Keyboard, wlElementId, blReturn )
  call wInitText(     WID_Text,     wlElementId, blReturn )

  '**********************************
  ' START GUI
  '**********************************
  set_task_prio main, 1 ' reset task priority for usual running
  call bAdministrateWindows( blReturn )
end


'-----------------------------------------------------------------------------
' wInitGeneral:
'-----------------------------------------------------------------------------
' initialize fonts and general elements
'-----------------------------------------------------------------------------
' RETURN VALUES:
'       wpvElementId    IN:  first free identifier for these elements
'                       OUT: next free identifier for further elements
'       bpvReturn       error code: 0=OK, <0=ERROR
'-----------------------------------------------------------------------------
```

```
sub wInitGeneral( var word wpvElementId; var byte bpvReturn )
  call bTglCreateFont( &
  FID_KeybView, &                ' identifier of font
  "Valencia", 18, "normal", &    ' name, size, type of font
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif

  call bTglCreateFontParams( &
  FID_KeybKey, &                 ' 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
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif

  call bTglCreateFontParams( &
  FID_Text, &                    ' 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", "word", &               ' overlay, wrap mode
  bpvReturn )                    ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
end


'----------------------------------------------------------------------------
' wInitKeyboard:
'----------------------------------------------------------------------------
' PARAMETERS:
'       wpWindowId      identifier for window
' RETURN VALUES:
'       wpvElementId    IN:  first free identifier for these elements
'                       OUT: next free identifier for further elements
'       bpvReturn       error code: 0=OK, <0=ERROR
'----------------------------------------------------------------------------
sub wInitKeyboard( word wpWindowId; var word wpvElementId &
  ;                var byte bpvReturn )
  byte blFrame                   ' frame thickness of keys
  word wlDistance                ' space between keys
  word wlX,wlY                   ' top left edge of top left key
  word wlWidth,wlHeight          ' key size
  byte blKeysRow,blKeysCol       ' number of keys in row, column
  word wlHeightLabel             ' height of label for user input
  string slKeyCodes$(80)         ' key codes and layout codes for keyboard
  '***********************************
  ' create your own keyboard style here
  ' mind the limited size of the lcd
  ' when choosing the values!
  '***********************************
```

```
  blFrame       = 4
  wlDistance    = 4
  wlWidth       = 60
  wlHeight      = 40
  blKeysRow     = 4
  blKeysCol     = 4
  wlHeightLabel = 36 ' this value is determined by the height of the label
  '                     for the user input!
  '' 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 &
  -              (blKeysRow*wlWidth  + (blKeysRow-1)*wlDistance))/2
  wlY         = wlHeightLabel + (LCD_HEIGHT - wlHeightLabel &
  -              (blKeysCol*wlHeight + (blKeysCol-1)*wlDistance))/2
  slKeyCodes$ = "123"+chr$(_DEL)+chr$(_NEXT_ROW) &      ' keys row 1
  +              "456"+chr$(_OK) +chr$(_NEXT_ROW) &      ' keys row 2
  +              "789"           +chr$(_NEXT_ROW) &      ' keys row 3
  +              "*0#"                                   ' keys row 4
  call wTglInitKeyboardSelfmade( wpWindowId, FID_KeybView, &
  FID_KeybKey, blFrame, wlDistance, wlX,wlY, wlWidth,wlHeight, &
  slKeyCodes$, wgEID_KeybView, wpvElementId, bpvReturn )
end


'---------------------------------------------------------------------------
' wInitText:
'---------------------------------------------------------------------------
' PARAMETERS:
'       wpWindowId      identifier for window
' RETURN VALUES:
'       wpvElementId    IN:  first free identifier for these elements
'                       OUT: next free identifier for further elements
'       bpvReturn       error code: 0=OK, <0=ERROR
'---------------------------------------------------------------------------
#define KEY_Text        0
word wgEID_Text
sub wInitText( word wpWindowId; var word wpvElementId; var byte bpvReturn )
  wgEID_Text = wpvElementId
  call bTglCreateTextButtonWnd( &
  260, 180, &                     ' width, height of element
  DFLT_Text, &                    ' text in element
  FID_Text, 5, &                  ' text, font id, frame thickness
  TGL_KEY_ATTR_AUTOREPEAT_OFF, &' key attributes: auto repeat,beep,no switch
  wpvElementId, wpWindowId, &     ' identifier of text button, window
  30, 30, &                       ' x, y coordinate on lcd
  KEY_Text, &                     ' keycode
  bpvReturn )                     ' return code (0: OK exit  >0: error exit)
  if bpvReturn <> TGL_MSG_OK then
    return
  endif
  wpvElementId = wpvElementId + 1
end
'---------------------------------------------------------------------------
' wExecText:
'---------------------------------------------------------------------------
' Edit text
'---------------------------------------------------------------------------
' 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 wExecText( var word wpvWindowId; var byte bpvReturn )
  byte ever, blKeycode
  word wlIbuFill
  for ever = 0 to 0 step 0
    call bTglGetKeycode( blKeycode, wlIbuFill )
    if 0 < wlIbuFill then
      switchi blKeycode

      case KEY_Text:
        sgInput$ = REQUEST_Input
        set_task_prio main, 4 ' speed up task for keyboard
        call sTglGetKeyboardInput( WID_Keyboard, wgEID_KeybView, &
        MAX_LEN_Text, sgInput$, bpvReturn )
        set_task_prio main, 1 ' reset task prio fur usual running
        if bpvReturn <> TGL_MSG_OK then
          return
        endif
        if 0 < len(sgInput$) then
          call bTglSetText( wgEID_Text, sgInput$, bpvReturn )
          if bpvReturn <> TGL_MSG_OK then
            return
          endif
        endif
        return

      endswitch ' blKeycode
    endif ' 0 < wlIbuFil
  next ' for ever
end


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

  wlWindowId = WID_Text
  for ever=0 to 0 step 0
    call bTglShowWindow( wlWindowId, bpvReturn )
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
    switchi wlWindowId

    case WID_Text:
      call wExecText( wlWindowId, bpvReturn )

    default:
      bpvReturn = 0FFh
```

```
      return
    endswitch
    if bpvReturn <> TGL_MSG_OK then
      return
    endif
  next
end
```

sTglGetKeyboardInput

call sTglGetKeyboardInput(WIDunshifted, EIDkeybTxt, 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	
WIDunshifted	-	●	-	-	-	identifier of the first keyboard window
EIDkeybTxt	-	●	-	-	-	identifier of the label for the user input
MaxLenInput	-	●	-	-	-	maximal input length
						Return Values:
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.

Sample program:
See sample in wTglInitKeyboard

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
						Return Values:
Input$	-	-	-	●	-	IN info text for the user OUT user input
Timeout	-	-	●	-	-	IN: stop keyboard input after this idle time 0 = no timeout OUT: >0 normal exit -1 timeout exit
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.

sTglEditText

call sTglEditText(WindowId, EID_View, MaxLenInput, Text$, Timeout, Keycode, & Result)

Function: Edit passed text and returns last keycode.

Parameters:

	B	W	L	S	F	
WindowId	-	●	-	-	-	identifier of the first keyboard window
EID_View	-	●	-	-	-	identifier of the label for the user input
MaxLenInput	-	●	-	-	-	maximal input length
						Return Values:
Text$	-	-	-	●	-	IN former text
						OUT modified text
Timeout	-	-	●	-	-	IN: stop keyboard input after this idle time
						0 = no timeout
						OUT: >0 normal exit
						-1 timeout exit
Keycode	●	-	-	-	-	code of last pressed key
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 pass a small and narrow font width the initialization of the keyboard, e.g VALENCIA_8_NORMAL.

Sample program:
See sample in wTglInitKeyboard

sTglGetKeybPassword

call sTglGetKeybPassword(WndIdUnshifted, ElemIdKeybTxt, MaxLenInput, Input$, Result)

Function: Shows keyboard in an own window, informs the user about the wanted input, displays “*” for each character and return the password.

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
						Return Values:
Input$	-	-	-	●	-	IN info text for the user
						OUT password
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

sTglGetKeybPasswordTimeout

call sTglGetKeybPasswordTimeout(WndIdUnshifted, ElemIdKeybTxt, MaxLenInput, Input$, Timeout, Result)

Function: Shows keyboard in an own window, informs the user about the wanted input, displays "*" for each character and return the password. 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
						Return Values:
Input$	-	-	-	●	-	IN info text for the user OUT password
Timeout	-	-	●	-	-	IN: stop keyboard input after this idle time 0 = no timeout OUT: >0 normal exit -1 timeout exit
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

sTglGetKeybParams

call sTglGetKeybParams(WndIdUnshifted, ElemIdKeybTxt, MaxLenInput, Mode, Input$, Timeout, Keycode, Result)

Function: Shows keyboard in an own window assembling all keyboard functionalities which are described above. The last pressed keyboard key will be additionally returned.

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
Mode	●	-	-	-	-	TGL_KEYB_MODE_EDIT edit passed text TGL_KEYB_MODE_GREET show greetings first TGL_KEYB_MODE_PWD invisible input "***"
						Return Values:
Input$	-	-	-	●	-	IN info text for the user OUT password
Timeout	-	-	●	-	-	IN: stop keyboard input after this idle time 0 = no timeout OUT: >0 normal exit -1 timeout exit
Keycode	●	-	-	-	-	last pressed key
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 pass a small and narrow font with 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.

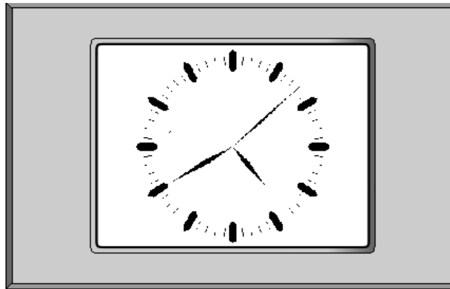


figure 91: Example for RTC application

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*

bTglInitRtc

call bTglInitRtc(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
						Return Values:
FontId	●	-	-	-	-	IN: current identifier of fonts OUT: next free identifier of fonts
ElementId	-	●	-	-	-	IN: current identifier of elements OUT: next free identifier of element
WindowId	-	●	-	-	-	IN: current identifier of windows identifier of window for calling the application OUT: next free identifier of window
ElemIdRtc	-	●	-	-	-	identifier of element for calling the application
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

There must be an existing font for a correct initialization of the keyboard.
Call bTglCreateFont or alternatively pass parameters as follows with the creation of the font:

alignment horizontal	"left"
alignment vertical	"top"
spacing type	"prop"
spacing blank	0
spacing char	SPACING_CHAR_DEFAULT
spacing vertical	0
overlay mode	"imm"
wrap mode	"char"

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	3	22 total 19 text buttons 3 labels	2

TGL_RTC_STYLE_1:

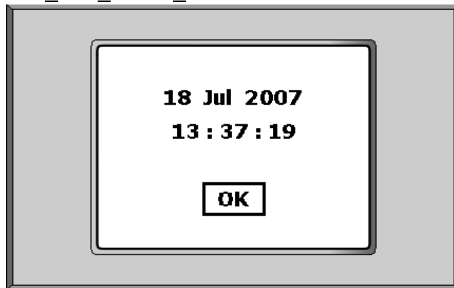


figure 92: TGL_RTC_STYLE_1 date and time

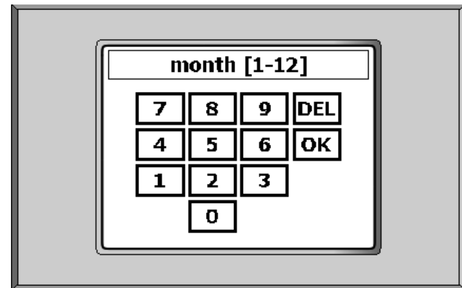


figure 93: TGL_RTC_STYLE_1 keyboard

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
						Return Values:
Result	●	-	-	-	-	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 wlElementId       ' current identifier for creation of elements
  word wlWindowId        ' current identifier for creation of windows
  byte blFontId          ' current identifier for creation of fonts
```

```
'**********************************
' RTC vars
'**********************************
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

'**********************************
' device drivers
'**********************************
#include TGL_DEVICE_DRIVERS_TP1000.INC
install_device #RTC, "RTC1.TD2"

'**********************************
' INITIALIZATION
'**********************************
call bTglInit( blReturn )
wlElementId = 0
wlWindowId  = 0
blFontId    = 0
set_len$( slInput$, 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 Graphic

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 94: 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.

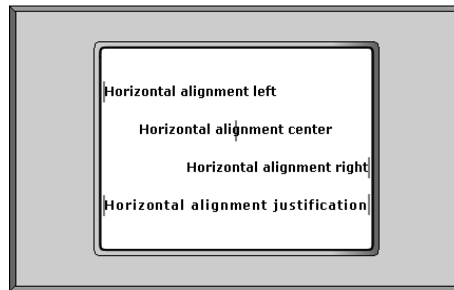


figure 95: Horizontal alignment

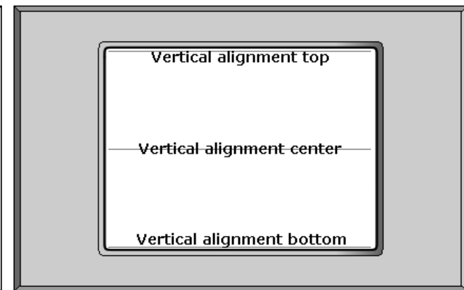


figure 96: Vertical alignment

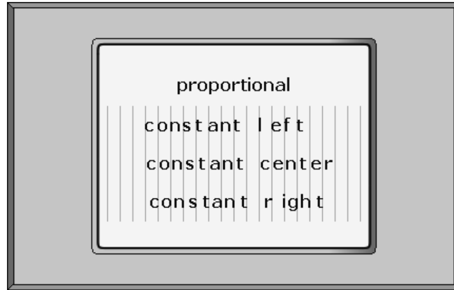


figure 97: Character spacing

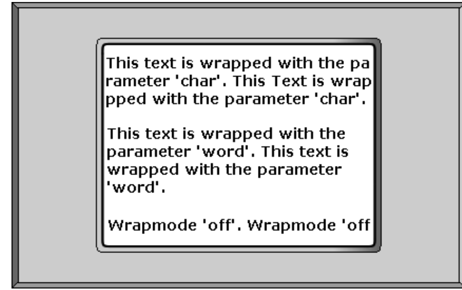


figure 98: Text wrapping

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*
- *sTglBuildText*
- *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 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	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 normal, bold	8,10,11,12,14,18,21 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 *%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\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 flattering 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 (propor-tional 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
ATLANTA_18_ITALIC	29	27	2	-4	-12

font	line height	max. width	default character spacing (propor-tional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
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
ISTANBUL_8_NORMAL	14	11	1	-2	-5

font	line height	max. width	default character spacing (propor-tional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
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
TOKIO_12_BOLD	21	13	1	0	-2

font	line height	max. width	default character spacing (propor-tional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
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
VALENCIA_12_NORMAL	18	21	1	-7	-9

font	line height	max. width	default character spacing (propor-tional spacing type)	default character spacing (constant spacing type)	default character spacing digits (constant spacing type)
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	26	2	-5	-10
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 99: 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.

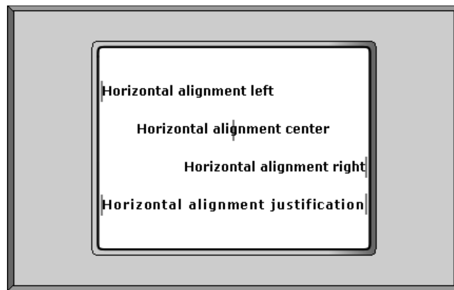


figure 100: Horizontal alignment

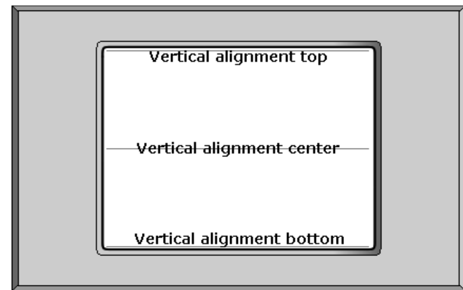


figure 101: Vertical alignment

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.

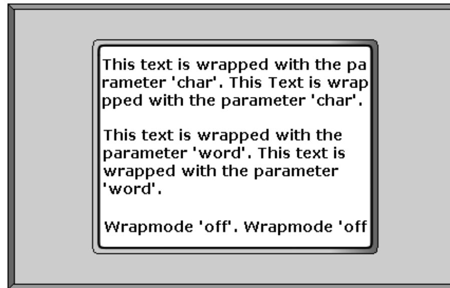


figure 102: Text wrapping

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
BLANK	0020	<20h>
!	0021	<21h>
"	0022	<22h>
#	0023	<23h>
$	0024	<24h>
%	0025	<25h>
&	0026	<26h>
'	0027	<27h>
(	0028	<28h>
)	0029	<29h>
*	002a	<2Ah>
+	002b	<2Bh>
,	002c	<2Ch>
-	002d	<2Dh>
.	002e	<2Eh>
/	002f	<2Fh>
0	0030	<30h>
1	0031	<31h>
2	0032	<32h>

char	Uni code	Tiger BASIC code
3	0033	<33h>
4	0034	<34h>
5	0035	<35h>
6	0036	<36h>
7	0037	<37h>
8	0038	<38h>
9	0039	<39h>
:	003a	<3Ah>
;	003b	<3Bh>
<	003c	<3Ch>
=	003d	<3Dh>
>	003e	<3Eh>
?	003f	<3Fh>
@	0040	<40h>
A	0041	<41h>
B	0042	<42h>
C	0043	<43h>
D	0044	<44h>
E	0045	<45h>
F	0046	<46h>

char	Uni code	Tiger BASIC code
G	0047	<47h>
H	0048	<48h>
I	0049	<49h>
J	004a	<4Ah>
K	004b	<4Bh>
L	004c	<4Ch>
M	004d	<4Dh>
N	004e	<4Eh>
O	004f	<4Fh>
P	0050	<50h>
Q	0051	<51h>
R	0052	<52h>
S	0053	<53h>
T	0054	<54h>
U	0055	<55h>
V	0056	<56h>
W	0057	<57h>
X	0058	<58h>
Y	0059	<59h>
Z	005a	<5Ah>

char	Uni code	Tiger BASIC code
[	005b	<5Bh>
\	005c	<5Ch>
]	005d	<5Dh>
^	005e	<5Eh>
_	005f	<5Fh>
`	0060	<60h>
a	0061	<61h>
b	0062	<62h>
c	0063	<63h>
d	0064	<64h>
e	0065	<65h>
f	0066	<66h>
g	0067	<67h>
h	0068	<68h>
i	0069	<69h>
j	006a	<6Ah>
k	006b	<6Bh>
l	006c	<6Ch>
m	006d	<6Dh>
n	006e	<6Eh>
o	006f	<6Fh>
p	0070	<70h>
q	0071	<71h>
r	0072	<72h>
s	0073	<73h>
t	0074	<74h>
u	0075	<75h>
v	0076	<76h>
w	0077	<77h>
x	0078	<78h>
y	0079	<79h>
z	007a	<7Ah>
{	007b	<7Bh>
		007c	<7Ch>
}	007d	<7Dh>
~	007e	<7Eh>
_DEL	007f	<7Fh>
€	20ac	<80h>
free	free	<81h>
free	free	<82h>
free	free	<83h>
"	0022	<84h>
free	free	<85h>
free	free	<86h>
free	free	<87h>
ˆ	02c6	<88h>
‰	2030	<89h>
Š	0160	<8Ah>
free	free	<8Bh>
Œ	0152	<8Ch>
free	free	<8Dh>
Ž	017d	<8Eh>
free	free	<8Fh>
free	free	<90h>
free	free	<91h>
free	free	<92h>
"	0022	<93h>
Ő	0150	<94h>
Ű	0170	<95h>
ő	0151	<96h>
ű	0171	<97h>
˜	02dc	<98h>
™	2122	<99h>
š	0161	<9Ah>
˝	02dd	<9Bh>
œ	0153	<9Ch>
free	free	<9Dh>
ž	017e	<9Eh>
free	free	<9Fh>
free	free	<A0h>
¡	00a1	<A1h>
¢	00a2	<A2h>
£	00a3	<A3h>
free	free	<A4h>
¥	00a5	<A5h>
free	free	<A6h>
§	00a7	<A7h>
¨	00a8	<A8h>
©	00a9	<A9h>
free	free	<AAh>
free	free	<ABh>
free	free	<ACh>
Ÿ	0178	<ADh>
®	00ae	<AEh>
free	free	<AFh>
°	00b0	<B0h>
±	00b1	<B1h>
free	free	<B2h>
free	free	<B3h>
´	00b4	<B4h>
µ	00b5	<B5h>
˙	02d9	<B6h>
˘	02d8	<B7h>

char	Uni code	Tiger BASIC code
¸	00b8	<B8h>
Ğ	011e	<B9h>
ğ	011f	<BAh>
Ş	015e	<BBh>
ş	015f	<BCh>
İ	0130	<BDh>
ı	0131	<BEh>
¿	00bf	<BFh>
À	00c0	<C0h>
Á	00c1	<C1h>
Â	00c2	<C2h>
Ã	00c3	<C3h>
Ä	00c4	<C4h>
Å	00c5	<C5h>
Æ	00c6	<C6h>
Ç	00c7	<C7h>
È	00c8	<C8h>
É	00c9	<C9h>
Ê	00ca	<CAh>
Ë	00cb	<CBh>
Ì	00cc	<CCh>
Í	00cd	<CDh>
Î	00ce	<CEh>
Ï	00cf	<CFh>
Ð	00d0	<D0h>
Ñ	00d1	<D1h>
Ò	00d2	<D2h>
Ó	00d3	<D3h>
Ô	00d4	<D4h>
Õ	00d5	<D5h>
Ö	00d6	<D6h>
×	00d7	<D7h>
Ø	00d8	<D8h>
Ù	00d9	<D9h>
Ú	00da	<DAh>
Û	00db	<DBh>
Ü	00dc	<DCh>
Ý	00dd	<DDh>
Þ	00de	<DEh>
ß	00df	<DFh>
à	00e0	<E0h>
á	00e1	<E1h>
â	00e2	<E2h>
ã	00e3	<E3h>
ä	00e4	<E4h>
å	00e5	<E5h>
æ	00e6	<E6h>
ç	00e7	<E7h>
è	00e8	<E8h>
é	00e9	<E9h>
ê	00ea	<EAh>
ë	00eb	<EBh>
ì	00ec	<ECh>
í	00ed	<EDh>
î	00ee	<EEh>
ï	00ef	<EFh>
ð	00f0	<F0h>
ñ	00f1	<F1h>
ò	00f2	<F2h>
ó	00f3	<F3h>
ô	00f4	<F4h>
õ	00f5	<F5h>
ö	00f6	<F6h>
÷	00f7	<F7h>
ø	00f8	<F8h>
ù	00f9	<F9h>
ú	00fa	<FAh>
û	00fb	<FBh>
ü	00fc	<FCh>
ý	00fd	<FDh>
þ	00fe	<FEh>
ÿ	00ff	<FFh>

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>	Tabulator: Next character position is next multiple of 64 pixels from the left element border
<0Dh>	Carriage Return + Line Feed
<1Ch>	Font Select: Next byte is a new identifier of a font, not a regular character. Font must have been created before.

The following sample program will show this graphic on the display.

figure 103: Change font in one text

```
'-----------------------------------------------------------------------------
' FONT_ChangeFontInText.TIG
'-----------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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
  '*****************************************
  '                                                txt
  call sTglBuildText("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,& ' wDst,hDst, x,y, width,height
  FALSE,FONT_ID_0,TGL_ROTATE_0, sgLcd$, blReturn ) ' inv,FID,rot, dst, 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
%ProgramData%\Wilke Technology\Tiger Basic 5.4\Libraries\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 bTglInit( blReturn )
```

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
'------------------------------------------------------------------------------
#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 bTglInit( blReturn )
  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 sTglBuildText ( slText$,&' text
  LCD_WIDTH,LCD_HEIGHT, &        ' format width, height of destination
  80,60, &                       ' x,y coordinates
  160,120, &                     ' width, height of text graphic
  FALSE,0,TGL_ROTATE_0, &        ' inverting flag, frame thickness
  sgLcd$, blReturn )             ' dst, return value (0=Ok, >0=Error)

  '***********************************
  ' SHOW TEXT GRAPHIC
  '***********************************
  call vTglShowUserGraphic( sgLcd$ )
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 bold 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	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 normal, bold	8,10,11,12,14,18,21 24,36,48

default Parameters	
horizontal alignment	left
vertical alignement	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
						Return Values:
Result	●	-	-	-	-	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

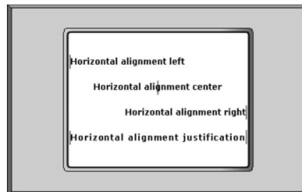


figure 104: Horizontal alignment

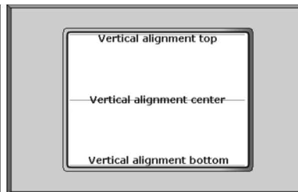


figure 105: Vertical alignment

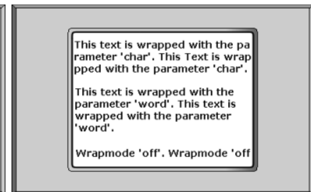


figure 90: Text wrapping

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 flattering 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:

	B	W	L	S	F	
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 flattering 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
						Return Values:
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

sTglBuildText, sTglBuildTextF

```
call sTglBuildText (  Dst$,        Text$, WDst,HDst, X,Y, Width,Height, InvertFlag, &
                      FontId, bpRotate, Result )
call sTglBuildText F(Addr, Len,  Text$, WDst,HDst, X,Y, Width,Height, InvertFlag, &
                      FontId, bpRotate, Result )
```

Function: Builds a text graphic and places it in a graphic area. Optionally the text graphic can be inverted and rotated.

Parameters:

	B	W	L	S	F	
Text$	-	-	-	●	-	this text will be put into graphic
Addr, Len	-	-	●	-	-	first address and length of text in flash
WDst	-	-	●	-	-	width of destination area (multiple of 8)
HDst	-	-	●	-	-	height of destination area
X, Y	-	-	●	-	-	position of top left corner of graphic box
Width, Height	-	-	●	-	-	size of graphic box
InvertFlag	●	-	-	-	-	TRUE = show graphic box inverted FALSE = show graphic box normal
FontId	●	-	-	-	-	unique identifier of this font
Rotate	●	-	-	-	-	rotation to the right: TGL_ROTATE_0, TGL_ROTATE_90 TGL_ROTATE_180, TGL_ROTATE_270
						Return Values:
Dst$	-	-	-	●	-	graphic area with formatted graphic text
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok ›0 error

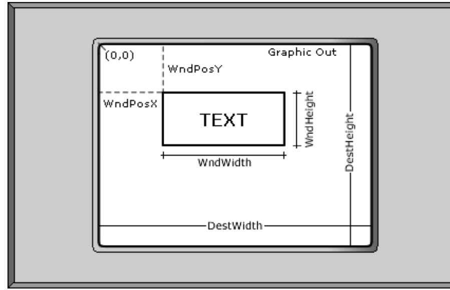


figure 106: Placing a text graphic on LCD

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 (32/8)*100 = 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 determinated in the file *TigerGraphicLibraryConf.INC*. The default value is 512 characters. You can increase this value if you have enough RAM.

The subroutine *lTglCalcTextToWindow* returns the fitting chars in the graphic box.

lTglCalcTextToWindow, lTglCalcTextToWindowF

```
call lTglCalcTextToWindow(   NumCharsWnd, NumLinesWnd, MaxNumLines, &
                             Text$,       FontId, WndWidth, WndHeight, Result )
call lTglCalcTextToWindowF(  NumCharsWnd, NumLinesWnd, MaxNumLines, &
                             Addr, Len,   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
Addr, Len	-	-	●	-	-	first address and length of text in flash
FontId	●	-	-	-	-	unique identifier of this font
WndWidth	-	-	●	-	-	width of graphic box
WndHeight	-	-	●	-	-	height of graphic box
						Return Values:
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

lTglGetLineHeight

call lTglGetLineHeight(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	-	-	●	-	-	**Return Value:** 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 107: Line height

For details see table in section *Specific Parameters of Graphic Fonts*

lTglCalcTextGraphicWidth

call lTglCalcTextGraphicWidth(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
						Return Value:
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, vTglShowUserGraphicF*
  vTglShowUserGraphicParams, vTglShowUserGraphicFParams
- *vTglHideUserGraphic*
- *sTglGetWindowGraphic*
- *vTglPutWindowGraphic vTglPutWindowGraphicF*
- *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!

A full LCD output request a string length of LCD_SIZE!

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.

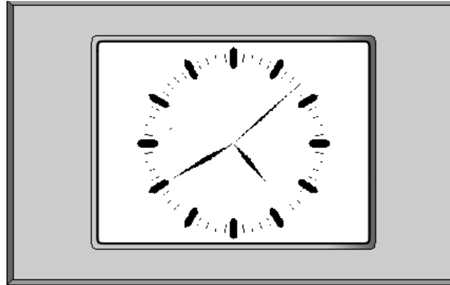


figure 108: Self made graphic analog clock

sample program:

```
'------------------------------------------------------------------------------
` TGL_USER_GRAPHIC_analog_clock.TIG
'------------------------------------------------------------------------------
#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 wlIbuFill        ' filling of touch panel input buffer
  byte blKeycode

  '' TGL general
  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
  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, "RTC1.TD2"

'***********************************
' INITIALIZATION
'***********************************
'' variables
call bTglInit( blReturn )
wlElementId = 0
wlWindowId  = 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 = wlWindowId
call bTglInitRtc( TGL_RTC_STYLE_1, blFONT_ID_rtc, wlElementId, &
wlWindowId, 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 = wlWindowId
```

```
  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
  wlElementId, wlWindowId, &       ' identifier of element, window
  0, 0, &                          ' x, y coordinate on LCD
  0h, &                            ' keycode
  blReturn )                       ' return code (0: OK exit  >0: error exit)

  '***************************************
  ' start program
  '***************************************
  call bTglShowWindow( wlWND_ID_clock, blReturn )
  for ever = 0 to 0 step 0

    '' set new time
    get #TP, #0, #UFCI_IBU_FILL, 0, wlIbuFill
    if 0 < wlIbuFill then
      call bTglSetRtc( wlWND_ID_rtc, wlELEM_ID_rtc, blReturn )
      call bTglShowWindow( wlWND_ID_clock, blReturn )
    endif

    '' update time on LCD secondly
    get #rtc, #3, 0, slDate$ ' h,m,s,d,M,yy,D
    if slDateOld$ <> slDate$ then
      slDateOld$ = slDate$
      call sUpdateClock( slDate$ )
    endif

    release_task
  next  ' end of endless loop
end


'----------------------------------------------------------------------------
' update time on LCD
'----------------------------------------------------------------------------
sub sUpdateClock( string spDate$ )
  byte blReturn          ' return value of tgl subroutines
  string slDate$(8)
  byte blSec, blMin, blHour              ' time
  long llRotHour,llRotMin,llRotSec       ' angles

  ' extract time
  blSec  = nfroms( spDate$, 2, TGL_BYTE )
  blMin  = nfroms( spDate$, 1, TGL_BYTE )
  blHour = nfroms( spDate$, 0, TGL_BYTE )

  ' calculate angles for sec, min and hour
  llRotSec  = blSec *600  + 18000                 ' blSec*(36000/60) + 18000
  llRotMin  = blMin *600  + 18000                 ' blMin*(36000/60) + 18000
  llRotHour = blHour*3000 + 18000 + blMin*50      ' blHour*(36000/12)+ 18000
  '                                               ' + (blMin*(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, vTglShowUserGraphicF

```
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, 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	Return Values:
WndGraphic	-	-	-	●	-	LCD content of the current window
Result	●	-	-	-	-	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, 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 graphic operations, e.g. drawing a graph of measurands or visualizing values statically by charts or dynamically by gauges. These subroutines are working independently from elements. That means that these subroutines will draw graphics in strings which have to be declared by the user.

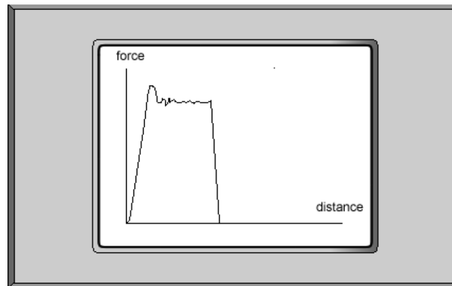


figure 109: Graph

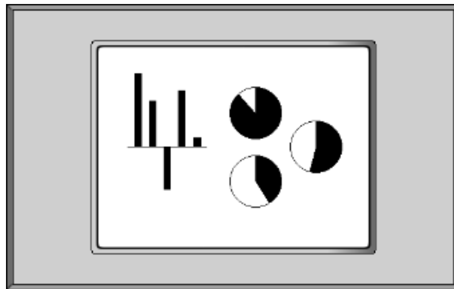


figure 110: Charts

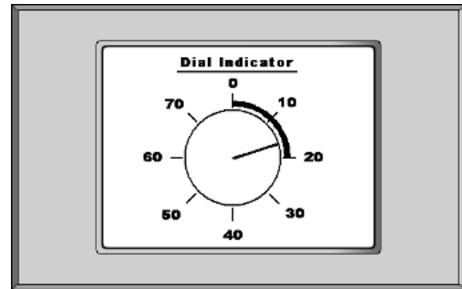


figure 111: Gauges

For showing these graphics on the LCD you can use the subroutines in chapter *User Graphic*.

For the use of graphs, charts and gauges with elements please see the chapters *General Subroutines* and *Gauges*.

Available Subroutines:

- *sTglDrawRectangle*
- *sTglDrawFrame*
- *sTglDrawBar*
- *sTglDrawCircle*

- *wTglCalcCircleParams*
- *sTglDrawPie*
- *sTglUpdatePie*
- *sTglDrawHand*
- *sTglDrawGraph*
- *sTglDrawAxes*
- *sTglClearGraph*
- *sTglRotate, sTglRotateF*
- *sTglRotateToDst, sTglRotateToDstL*
- *sTglRotateFToDstOffs*
- *sTglDrawLine*
- *sTglDrawNextLine*
- *sTglGraphicMove*
- *sTglGraphicErase*
- *sTglGraphicInvert*

sTglDrawRectangle

call sTglDrawRectangle (WDst,HDst, Width,Height, X,Y, Frame, Dst$)

Function: Draws a rectangle with the attributes size, position and line thickness.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the rectangle
X,Y	-	●	-	-	-	position of the rectangle in the destination
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

sTglDrawFrame

call sTglDrawFrame (WDst,HDst, Width,Height, Frame, Dst$)

Function: Draws a frame in the left top edge of a graphical area.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the rectangle
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

!

The destination area will be completely overwritten!

This subroutine is faster than *sTglDrawRectangle*.

sTglDrawBar

call sTglDrawBar(WDst,HDst, Width,Height, X,Y, Value, Min,Max, Base, Frame, Dst$)

Function: Draws a bar with the attributes size, position and line thickness. The value determines the percentaged filling of the bar relating to the limits beginning from the base.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the bar
X,Y	-	●	-	-	-	position of the bar in the destination
Value	-	-	●	-	-	actual bar value
Min,Max	-	-	●	-	-	bar value limits
Base	-	●	-	-	-	location of base of bar TGL_GA_BASE_LEFT TGL_GA_BASE_RIGHT TGL_GA_BASE_BOTTOM TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

sTglDraw Circle

call sTglDrawCircle(WDst,HDst, Radius, X,Y, Value, Frame, Dst$)

Function: Draws a circle with the attributes radius, position and line thickness.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Radius	-	●	-	-	-	radius of circle
X,Y	-	●	-	-	-	position of the centre in the destination
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

The size of the circle graphic is always odd (2*radius+1 in square).

wTglCalcCircleParams

call wTglCalcCircleParams (Width,Height, Frame, Radius, Diameter, DFmt)

Function: Calculates circle radius, diameter and its next multiple of 8 from passed size and line thickness.

Parameters:

	B	W	L	S	F	
Width,Height	-	●	-	-	-	size of the circle area
Frame	●	-	-	-	-	line thickness
						Return Values:
Radius	-	●	-	-	-	radius of circle
Diameter, DFmt	-	●	-	-	-	diameter, its next multiple of 8

The circle diameter is always odd (2*radius+1 in square).

sTglDrawPie

call sTglDrawPie(WDst,HDst, Width,Height, X,Y, Value, Min,Max, Base, Frame, Tmp$, Dst$)

Function: Updates pie with radius at center position occasionally with frame circle.
The value determines the percentaged filling of the pie relating to the limits beginning from the base.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the pie
X,Y	-	●	-	-	-	position of the pie in the destination
Value	-	-	●	-	-	actual pie value
Min,Max	-	-	●	-	-	pie value limits
Base	-	●	-	-	-	location of base of pie TGL_GA_BASE_LEFT TGL_GA_BASE_RIGHT TGL_GA_BASE_BOTTOM TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness
						Return Values:
Tmp$	-	-	-	●	-	temporary string for graphical operations
Dst$	-	-	-	●	-	container for graphical area

The subroutine needs temporary string in minimum size of pie for graphical operations.

The size of the pie graphic is always odd (2*radius+1 in square).

sTglUpdatePie

call sTglUpdatePie(WDst,HDst, Width,Height, X,Y, Old,New, Min,Max, Base, Frame, Tmp$, Dst$)

Function: Updates pie with radius at center position occasionally with frame circle.
The value determines the percentaged filling of the pie relating to the limits beginning from the base.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the pie
X,Y	-	●	-	-	-	position of the pie in the destination
Old,New	-	-	●	-	-	former and actual pie value
Min,Max	-	-	●	-	-	pie value limits
Base	-	●	-	-	-	location of base of pie TGL_GA_BASE_LEFT TGL_GA_BASE_RIGHT TGL_GA_BASE_BOTTOM TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness
						Return Values:
Tmp$	-	-	-	●	-	graphical working string
Dst$	-	-	-	●	-	container for graphical area

The subroutine needs temporary string in minimum size of pie for working.
The size of the pie graphic is always odd (2*radius+1 in square).

sTglDrawHand

call sTglDrawHand(Width,Height, Value, Min,Max, Base, Frame, Dst$)

Function: Draws a cyclic gauge's hand.

Parameters:

	B	W	L	S	F	
Width,Height	-	●	-	-	-	size of the hands circle
Value	-	-	●	-	-	actual hand's value
Min,Max	-	-	●	-	-	hand value limits
Base	-	●	-	-	-	location of base of hand
						TGL_GA_BASE_LEFT
						TGL_GA_BASE_RIGHT
						TGL_GA_BASE_BOTTOM
						TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

The size of the gauge graphic is always odd (2*radius+1 in square).

The container *Dst$* will be erased first and set to the optimal length fitting to the gauge.

sTglUpdateHand

call sTglUpdateHand(WDst,HDst, Width,Height, X,Y, Old,New, Min,Max, Base, Frame, Tmp$, Dst$)

Function: Updates a gauge's hand.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the gauge
X,Y	-	●	-	-	-	position of the pie in the destination
Old,New	-	-	●	-	-	former and actual gauge value
Min,Max	-	-	●	-	-	gauge value limits
Base	-	●	-	-	-	location of base of pie TGL_GA_BASE_LEFT TGL_GA_BASE_RIGHT TGL_GA_BASE_BOTTOM TGL_GA_BASE_TOP
Frame	●	-	-	-	-	line thickness
						Return Values:
Tmp$	-	-	-	●	-	graphical working string
Dst$	-	-	-	●	-	container for graphical area

The subroutine needs temporary string in minimum gauge size for working.
The size of the gauge graphic is always odd (2*radius+1 in square).

sTglDrawListbox

call sTglDrawListbox(Width,Height, WBut, Frame, Dst$)

Function: Draws an empty listbox.

Parameters:

	B	W	L	S	F	
Width,Height	-	●	-	-	-	size of the listbox
WBut	-	●	-	-	-	width of up and down buttons
Frame	●	-	-	-	-	line thickness
						Return Values:
Dst$	-	-	-	●	-	container for graphical area

The arrow size is related to the button width.

The container *Dst$* will be erased first and set to the optimal length fitting to the listbox.

sTglUpdateScope

call sTglUpdateScope(WDst,HDst, Width,Height, X,Y, Data$, Min,Max, Step, Tmp$, Dst$)

Function: Move graph of oscilloscope.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst must be a multiple of 8!
Width,Height	-	●	-	-	-	size of the area for the oszillograph
X,Y	-	●	-	-	-	position of the scope's area in the destination
Data$	-	-	-	●	-	new and old value of graph, LSB first
Min,Max	-	-	-	-	●	scope value limits
Step	-	●	-	-	-	horizontal pixel translation of graph for these values
Direction	●	-	-	-	-	moving direction related to unrotatedgraph TGL_DIR_RIGHT, TGL_DIR_LEFT TGL_DIR_BOTTOM, TGL_DIR_TOP
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Tmp$	-	-	-	●	-	temporary sting of same size of Dst$ for internal use
Dst$	-	-	-	●	-	graphical data code

The values can be easily written in Data$ by calling the functions ntos$, rtos$ or pushn.

sTglDrawGraph

```
call sTglDrawGraph(WDst,HDst, Width,Height, X,Y, Data$, Size, &
                   FirstVal, Num, NumTotal, LimInf,LimSup, Axis, Rotation, Dst$ )
```

Function: Draws a scaled graph in a rectangular area of a destination area.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst 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
Data$	-	-	-	●	-	buffer containing numerical values of the same type, LSB first
Size	●	-	-	-	-	variable type of values in Data$ 1=byte, 2=word, 4=long, 8=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 1 invalid value, must be at least 2 or 0
NumTotal	-	●	-	-	-	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	-	-	-	-	●	limits for y scaling lpLimInf=lpLimSup automatically y scaling by the graph's height

	B	W	L	S	F	
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
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270 **Return Values:**
Dst$	-	-	-	●	-	graphical data code

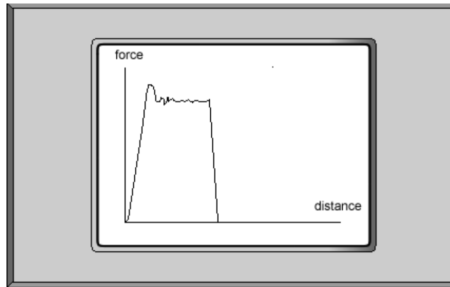


figure 112: Graph

For drawing a graph in an element "graphic", please call the subroutine *bTglShowGraph* described in the chapter *General Subroutines.*

For a correct functionality Graph$ must have been initialized with a fitting length. For a graphical area of e.g. 320x240 pixels the string request a minimal length of (320/8)*240 = 9600 bytes. For a “white” graphical area initialize the sting with e.g. fill$(“00”%, 9600)

If the axes should be drawn alone without any value, pass an empty string *Data$.* Alternatively call *sTglDrawAxes.*

sTglDrawAxes

call sTglDrawAxes(WDst,HDst, Width,Height, X,Y, Axis, Rotation, Dst$, Result)

Function: Draws axes for a graph in a rectangular area of a destination area.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst 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
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
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Dst$	-	-	-	●	-	graphical data code
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

sTglClearGraph

call sTglClearGraph(WDst,HDst, Width,Height, X,Y, Axis, Rotation, Dst$, Result)

Function: Clear a graph and redraw its axes.

Parameters:

	B	W	L	S	F	
WDst,HDst	-	●	-	-	-	format width, height of the destination WDst 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
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
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Dst$	-	-	-	●	-	graphical data code
Result	●	-	-	-	-	error code, for details see table of error codes 0 ok >0 error

sTglRotate, sTglRotateF

```
call sTglRotate(    Src$,   Width,Height, Rotation, Dst$)
call sTglRotateF(   Addr,   Width,Height, Rotation, Dst$)
```

Function: Rotates a bitmap passed as string resp. as flash address.

Parameters:

	B	W	L	S	F	
Src$	-	-	-	●	-	Container for unrotated source graphic
Addr	-	-	●	-	-	flash address of unrotated source graphic
Width,Height	-	●	-	-	-	size of the unrotated source graphic
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Dst$	-	-	-	●	-	rotated graphical data code

Width and height are fitting to those in the destination string.

sTglRotateToDst, sTglRotateToDstL

call sTglRotateToDst(Wsrc, Wdst,Hdst, X,Y, Width,Height, bpCopyMode, &
Rotation, CopyMode, Tmp$, Src$, Dst$)
call sTglRotateToDstL(Wsrc, Wdst,Hdst, X,Y, Width,Height, bpCopyMode, &
Rotation,CopyMode, Tmp$, Src$, Dst$)

Function: Rotate and copy passed bitmap into destination string.
sTglRotateToDst: parameters 1-7: word
sTglRotateToDstL: parameters 1-7: long

Parameters:

	B	W	L	S	F	
WSrc	-	●	●	-	-	format width of bitmap (a multiple of 8)
WDst,HDst	-	●	●	-	-	format size of destination area
X,Y	-	●	●	-	-	coordinates of left top bitmao edge
Width,Height	-	●	●	-	-	size of the unrotated source graphic
CopyMode	●	-	-	-	-	
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Tmp$	-	-	-	●	-	temporary string
Src$	-	-	-	●	-	Container for unrotated source graphic
Dst$	-	-	-	●	-	rotated graphical data code

Width and height are fitting to those in the destination string.

Source string must be variable and will be modified for 180°+270°!

sTglRotateFToDst

call sTglRotateFToDst(BmpAddr, Wsrc, WDst,HDst, X,Y, Width,Height, & bpCopyMode, Rotation, CopyMode, Tmp$, Dst$)

Function: Rotate and copy bitmap from flash to destination string.

Parameters:

	B	W	L	S	F	
BmpAddr	-	-	●	-	-	flash address of bitmap
WSrc	-	●	-	-	-	format width of bitmap (a multiple of 8)
WDst,HDst	-	●	-	-	-	format size of destination area
X,Y	-	●	-	-	-	coordinates of left top bitmao edge
Width,Height	-	●	-	-	-	size of the unrotated source graphic
CopyMode	●	-	-	-	-	
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Tmp$	-	-	-	●	-	temporary string
Dst$	-	-	-	●	-	rotated graphical data code

Width and height are fitting to those in the destination string.

Source string must be variable and will be modified for 180°+270°!

sTglRotateFToDstOffs

```
call sTglRotateFToDstOffs( BmpAddr, Wsrc, X0,Y0, W0,H0, X,Y, Width,Height,  &
                           bpCopyMode,  Rotation,  CopyMode, Tmp$, Dst$)
```

Function: Rotate and copy a part of a bitmap from flash to destination area.

Parameters:

	B	W	L	S	F	
BmpAddr	-	-	●	-	-	flash address of bitmap
WSrc	-	●	-	-	-	format width of bitmap (a multiple of 8)
X0,Y0	-	●	-	-	-	offset position in bitmap
W0,H0	-	●	-	-	-	size of whole bitmap
X,Y	-	●	-	-	-	coordinates of left top bitmao edge
Width,Height	-	●	-	-	-	size of the unrotated source graphic
CopyMode	●	-	-	-	-	
Rotation	●	-	-	-	-	Rotation TGL_ROTATE_0 TGL_ROTATE_90 TGL_ROTATE_180 TGL_ROTATE_270
						Return Values:
Tmp$	-	-	-	●	-	temporary string
Dst$	-	-	-	●	-	rotated graphical data code

Coordinates, Width and height are fitting to those in the destination string.

Destination area is sized as LCD_WIDTHxLCD_HEIGHT

sTglGraphicMove

```
call sTglGraphicMove(   Wsrc,Hsrc Width,Height, X,Y,  Step, Mode, Direction, &
                        Flag, Tmp$, Src$)
```

Function: Move a rectangular graphic in a graphical area, occasionally with erasing the graphic at the old position

Parameters:

	B	W	L	S	F	
Wsrc,Hsrc	-	●	-	-	-	format size of source area
Width,Height	-	●	-	-	-	size of the graphic
X,Y	-	●	-	-	-	position of the graphic
Step	●	-	-	-	-	translation width
Mode	●	-	-	-	-	overlay mode
Direction	●	-	-	-	-	translation direction
Flag	●	-	-	-	-	TRUE: wipeout graphic at old position FALSE: do not erase anything
						Return Values:
Tmp$	-	-	-	●	-	temporary string, min format size of graphic
Src$	-	-	-	●	-	container of graphical Area

sTglGraphicErase

call sTglGraphicErase(Wsrc,Hsrc Width,Height, X,Y, Src$)

Function: Erase a rectangular graphic from a graphical area.

Parameters:

	B	W	L	S	F	
Wsrc,Hsrc	-	●	-	-	-	format size of source area
Width,Height	-	●	-	-	-	size of the graphic
X,Y	-	●	-	-	-	position of the graphic
						Return Values:
Src$	-	-	-	●	-	graphical data code

sTglGraphicInvert

call sTglGraphicInvert(Wsrc,Hsrc Width,Height, X,Y, Src$)

Function: Invert a rectangular graphic in a graphical area.

Parameters:

	B	W	L	S	F	
Wsrc,Hsrc	-	●	-	-	-	format size of source area
Width,Height	-	●	-	-	-	size of the graphic
X,Y	-	●	-	-	-	position of the graphic
						Return Values:
Src$	-	-	-	●	-	graphical data code

Eeprom

On the TP1000 is a 64k eeprom which can be easily used for variable program settings.

The pins and ports are initialized with the inclusion of the TP1000 device drivers *TGL_DEVICE_DRIVERS_TP1000.INC*.

For a definition of the i²c pins see *TigerGraphicLibraryConf.INC*.

The calibration parameters for the touchpanel are already saved on this eeprom. The addresses can be set in the configuration file *TigerGraphicLibraryConf.INC*. Default addresses for these parameters are the highest addresses from 65509 to 65535.

sReadStringFromEeprom

call sReadStringFromEeprom(Addr, Length, Data$)

Function: Reads from an address a number of bytes from the eeprom.

Parameters:

	B	W	L	S	F	
Addr	-	-	-●	-	-	eeprom address
Length	-	●	-	-	-	number of bytes
						Return Values:
Data$	-	-	-	●	-	data bytes

vWriteStringToEeprom

```
call vWriteStringToEeprom( Data$, Addr )
```

Function: Writes a string to the eeprom.

Parameters:

	B	W	L	S	F	
Data$	-	-	-	●	-	data bytes
Addr	-	-●	-	-	-	eeprom address

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, wlIbuFill ' get buffer length
if 0 < wlIbuFill then                     ' check length of input buffer
  get #TP, #0, 0, slInput$                ' 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 => 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.

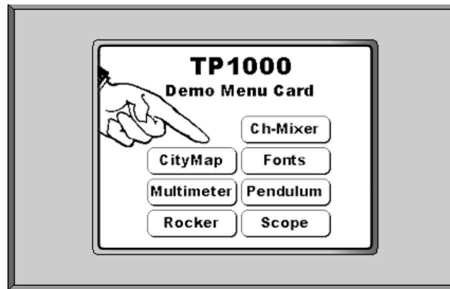


figure 113: Application menu

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 bTglInit( blReturn )
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 grahic 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 1=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_INVALID	You have to reserve more space for elements in file *TigerGraphicLibraryConf.INC*. Increase the value of *TGL_MAX_NUM_ELEMENTS* or use smaller identifier, if available.
2	TGL_ERR_WINDOW_INVALID	The identifier of the window is too large. If you want to use more windows, please increase the value of *TGL_MAX_NUM_WINDOWS* in *TigerGraphicLibraryConf.INC*
3	TGL_ERR_WINDOW_STR_LEN	Reserved RAM for elements in windows is full. Increase the value of *TGL_WINDOW_ATTRIBUTES_LEN* in *TigerGraphicLibraryConf.INC*
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_ELEMENTS	You have to reserve more space for specified elements in file *TigerGraphicLibraryConf.INC*. Increase the value of maximum number of specified element. E.g. increase *TGL_MAX_NUM_BUTTONS*, if specified element is a button.
7	TGL_ERR_ALREADY_PLACED	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_FULL	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_ATTRIBUTE	Attribute does not exist
10	TGL_ERR_INVALID_ATTRIBUTE_VALUE	Value for attribute is not allowed
11	TGL_ERR_INVALID_POSITION	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_PARAMETER	Selected option is not available
15	TGL_MSG_NO_CHANGE	Element has not changed (no inversion ...)
16	TGL_ERR_TEXT_TOO_LONG	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_OVERFLOW	Not enough reserved RAM for text. Please increase *TGL_MAX_MEM_TEXTS_LEN* in TigerGraphicLibraryConf.INC
23	TGL_ERR_INVALID_LINK	This link is not allowed. Please read documentation for *bTglLink*
24	TGL_ERR_LCD_ROTATION	Only one definition of TGL_LCD_ROTATION may be activated in TigerGraphicLibraryConf.INC
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.

No.	Name	Description
26	free	
29	free	
30	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 *TGL_KEY_ATTR_SWITCH*
31	TGL_ERR_LCD_CPY_STR_TO_SMALL	Passed parameter for copy of window graphic is smaller than *LCD_SIZE*
40	TGL_ERR_TYPE_INVALID	not existing type or invalid type for this subroutine
41	TGL_ERR_LIST_OVERFLOW	invalid index of list
42	TGL_ERR_NO_FLASH_ADDR	text resp. bitmap has no flash address
43	TGL_ERR_BASE_INVALID	invalid base value for this type resp. this element
44	TGL_ERR_SLIDER_TYPE_INVALID	nor X nor Y slider
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 TGL_MAX_NUM_BUTTONS_IN_WINDOW in TigerGraphicLibraryConf.TIG
61	TGL_ERR_MISSING_INTERNAL_KEYCODE	Fatal error. Please call support
62	free	
63	free	
81	TGL_ERR_FONT_TASKS_OVERFLOW	*sCreateTxtGraphic* is called in too many tasks. Please increase *TGL_MAX_NUM_TXT_GRAPHIC_STR* in *TigerGraphicLibraryConf.INC*
82	TGL_ERR_FONT_GRAPHIC_OVERFLOW	Text graphic does not fit in the internal string. Occasionally increase *TGL_TXT_GRAPHIC_LEN* in *TigerGraphicLibraryConf.INC*

No.	Name	Description
83	TGL_ERR_FONT_INVALID	Index of format string is too high. Please increase *TGL_MAX_NUM_FONTS* in *TigerGraphicLibraryConf.INC*
84	TGL_ERR_FONT_NAME_INVALID	Invalid font name choice. Please see documentation of *bTglCreateFont.*
85	TGL_ERR_FONT_TYPE_INVALID	Invalid font type choice. Please see documentation of *bTglCreateFont.*
86	TGL_ERR_FONT_ALIGN_V_INVALID	Invalid vertical alignment parameter. Please see documentation of *bTglCreateFontParams.*
87	TGL_ERR_FONT_ALIGN_H_INVALID	Invalid horizontal alignment parameter. Please see documentation of *bTglCreateFontParams.*
88	TGL_ERR_FONT_SPC_TYPE_INVALID	Invalid spacing type parameter
89	TGL_ERR_FONT_OVERLAY_INVALID	Invalid overlay mode. Please see documentation of *bTglCreateFontParams.*
90	TGL_ERR_FONT_WRAP_INVALID	Invalid wrapping option. Please see documentation of *bTglCreateFontParams.*
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_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_VAR_OVERFLOW	Internal variable overflow. Increment TGL_TXT_GRAPHIC_TXT_LEN in TigerGraphicLibraryConf.INC
97	TGL_ERR_TEXT_GRA_VAR_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	

No.	Name	Description
110	TGL_ERR_INVALID_STYLE	Style is not available for this subroutine. Please see description of subroutine.
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	TGL_ERR_AXES_INVALID	Invalid value for axes
150	TGL_NO_ALT_GRA	No alternative graphic available for this element. Please call *vTglLink* 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 - a special shift key for changing the keys - change of the size for the user input - up sized single keys Additionally this example gives you an example how to organize your code for a graphical user interface
TGL_KEYBOARD_STYLES.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 attibutes 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_bTglUpdateScope.TIG	general	Draws oscillograph
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

Name	Type	Description
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 roteted graphics
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 an 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 roteted labels
TGL_LISTBOX_createWnd.TIG	listbox	Create and place a listbox in a window
TGL_LISTBOX_set_get_index.TIG	listbox	Set and get incices of listboxes
TGL_lTglCalcTextToWindow.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_axis	general	Shows the types for axes of graphs
TGL_SHOW_GRAPH_clear	general	Demonstrates how to erase a graph from LCD

Name	Type	Description
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 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_createWnd.TIG	slider	Demonstrates how to create and place a slider in one go
TGL_SLIDER_Y_dockWnd.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

Name	Type	Description
TGL_STANDBY.TIG	general	Demonstrates how to use the stand-by function og 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 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.

Name	Type	Description
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
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 appartment 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

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 sTglShowButton
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
	- TP1000_DMX_Demo.TIG - Doorbells.TIG - TERMINAL.tig bug fixes new features - bigger fonts, - new font family Stockholm - new functionality font change in text
1.06	bug fixes - bTglSetAttribute - TGL_ATTR_INVERT
1.07	bug fixes - bTglGetSliderValue - bTglSetAttribute - TGL_ATTR_INVERT enhancements - gauges - blink - standby
1.08	bug fixes - bTglShowText for textbuttons - textbutton with text stored in flash memory - bTglSetSliderLimits - selfmade keyboard with changing of keys - selfmade keyboard with wide buttons applications - styled sliders
1.09	bug fixes - btglShowText for textbuttons - textbutton with text stored in flash memory - blinking of textbuttons and labels enhancements - set inversion state of elements TGL_ATTR_INV_STATE
1.10	enhancements - sTglGetKeybInputTimeout - bTglCreateFont tolerant parameters resp. " _-" - lTglGetFontHeight - bTglSetMargins/bTglGetMargins

Version of	Description / Changes
	- bTglGetTouch - bTglGetPushButtonState bug fixes - cursor off - text position keyboard view - create text elements check font
1.11	bug fixes - bTglGetButtonState/bTglSetButtonState - switches: non following identifiers - blink text elements enhancements - delete text elements
1.12	enhancements - new element listbox - bTglCreateFont/bTglCreateFontParams/bTglSetFont - new error message TGL_ERR_FONT_NOT_INCLUDED - default spacing for constant spacing of letters and digits bug fixes - bTglSetText/bTglShow text refresh on LCD - key beep after vTglHideWindow - alignments in bitmaps helsinki 52,56,60
1.13	enhancements - rtc style 1 -> setting of day of week bug fixes - update bars base right,top,bottom
1.14	enhancements - fully multitasking ability - new error codes for internal string overflow using graphic fonts - step by step lessons for the programming of applications using the Tiger Graphic Library bug fixes - line height with font select - line wrapping by cr - missing initializations in the keyboard and checking subroutines
1.15	enhancements - bTglSetLimits

Version of	Description / Changes
	- init push buttons, switches, background masks - tutorial for programming applications bugfixes - gauges: min‹›0 / max ‹ min - sTblStrR ‹ 0 - stop blinking of elements in not shown windows - rotate 180°+270° - graph limits
2.00	enhancements - initialization subroutines for push textbuttons, switch textbuttons, sliders - internal accellerations - hidden password input - new bitmap keyboard in hexstyle - new icons and element bitmaps - examples easier to copy - LCD orientations: landscape, portrait, upside-down - LCD grid and cordinate definitions bugfixes - blink - separators in lists for listboxes
2.01	bug fixes - bTglShowText update height - sub name: wTglGetNumTouchedElementsFlag -› wTglGetTouchedElementsFlag - reinversion push button
2.02	enhancements - bTglShowDetail, bTglShowDetailF - bTglSetKeyAttributes, bTglGetKeyAttributes - sTglBuildTextF - lTglCalcTextToWindowF - TGL_BASIC_TEMPLATE
2.03	- Telephone number changed