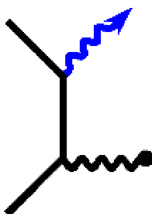
Handbook for the Simplify Technologies GUI Library

Version 3.0.0



Simplify Technologies GmbH
Steinbühlstrasse 15
D-35578 Wetzlar
Germany

$Rev: 6373 $

Simplify Technologies GmbH
Steinbühlstrasse 15
D-35578 Wetzlar
Germany
Tel.: (+49) (0)6441-210390
FAX.: (+49) (0)6441-210399
Internet: www.simplify-technologies.de

The use of general descriptive names, registered names, trademarks, etc. in this handbook does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
Use of a term should not be regarded as affecting the validity of any registered trademark, trademark, or service mark.
All trademarks, registered trademarks or service marks are the property of their respective owners.

All rights reserved. Simplify Technologies GmbH, 2000 – 2011.

Contents

1 Introduction 3

2 Concept of programming with the GUI library 5

3 Modules of the GUI library and programming 8
- 3.1 Function modules and header files . . . 8
- 3.2 Properties of functions and variables . . . 10
- 3.3 System functions . . . 10
- 3.4 Display channel . . . 11
- 3.5 LCD–Device . . . 16
- 3.6 Touch–Channel . . . 17
- 3.7 Buttons . . . 18
- 3.8 Tglass–Device . . . 20
- 3.9 Keyboard–Channel . . . 20
- 3.10 T_keyb device . . . 21
- 3.11 Sound–Channel . . . 22
- 3.12 Piezo–Device . . . 23

4 Additional modules (*AddOns*) 24
- 4.1 Seven segment display . . . 24
- 4.2 Inputline . . . 24
- 4.3 Progress bar (*bar indicator*) . . . 26
- 4.4 Slider . . . 26
- 4.5 Selection list . . . 27
- 4.6 Menus . . . 29
- 4.7 Event handling . . . 31
- 4.8 Miscellaneous . . . 32
  - 4.8.1 Clipping . . . 32
  - 4.8.2 Random numbers . . . 32
  - 4.8.3 String functions . . . 33

5 Adaption of the library to the target hardware 34
- 5.1 Requirements for the use of the GUI library . . . 34
- 5.2 Survey and short description of the configuration . . . 35
- 5.3 Definitions for the programming tool chain used . . . 36
- 5.4 Driver for hardware and system environment . . . 36
  - 5.4.1 Functions for the realization of a system timer . . . 36
  - 5.4.2 Driver for LCD display . . . 40
  - 5.4.3 Driver for touch glass . . . 47
  - 5.4.4 Driver for the loudspeaker . . . 48

5.5 Configuration of the GUI library 48
5.5.1 Adjustments for the hardware in “config_hardware_gui.h” . 49
5.5.2 Adjustments for the software environment in “config_gui.h” 52
5.6 Optimization 56

6 Colors and shades of gray (optional) 58
6.1 Restrictions using a color depth of 8 bit 58
6.1.1 Color models and use within the library 58
6.1.2 Handling of color BMP images 59
6.2 Color depth of 24 bit *(truecolor)* 60

7 Comprehensive programming example 61
7.1 Monochrome example 61
7.1.1 Components of the programming example 61
7.2 Color programming example 65

8 Software safety 66
8.1 Facts 66
8.2 Precautions 66

9 Function reference of the library functions 68

10 Frequently asked questions (FAQ) and troubleshooting 70

A Fonts 72

B License 75

1 Introduction

Graphical LCD and touch displays are increasingly used in *embedded systems* to allow for an ergonomic use of these devices.
The advantage of a touch glass lies in the possibility to create ergonomic and easy to use user interfaces, which can be adapted by a software modification only and thus avoid extensive and costly hardware modifications.

The Simplify Technologies GUI Library is a collection of subroutines working closely together to realize a graphical user interface. It allows you to benefit from the advantages of graphical displays without bothering with the details of complex interfacing routines. Especially when using a touch display this leads to a considerable simplification and acceleration of your development process.

The documentation of the Simplify Technologies GUI Library consists of three parts:

1. This Handbook describes the programming and the functions of the Simplify Technologies GUI Library from the point of view of the application programmer and the process of adapting the library to the requirements of your target hardware.

2. The "Quickstart Tutorial" features examples and a *step–by–step* description of how to start with the product to get you programming with the library as quickly as possible.

3. The description of the drivers of the LCD display and the LCD controller points out specific properties and features of the respective driver which need to be taken into account.

In chapter 2 the handbook describes the basic concepts for programming the graphical user interface (GUI).
Chapter 3 deals with the function modules of the GUI library and with their programming features.
Chapter 4 describes some additional function modules, which make heavy use of the core GUI library features presented in the previous chapter.
Questions of how to adapt the library to the needs of your target hardware are dealt with in chapter 5.
Chapter 6 describes the use of color for color displays with the color version of the library.
A extensive programming example is given in chapter 7 to illustrate details of programming with the library and to be used as a starting point for your own applications.
Chapter 8 gives some advice on what to consider when using the library in embedded systems to obtain a safe system.

The function reference is provided in a separate Document on the CD.
In chapter 10 you find answers to frequently asked questions (FAQ)

If this handbook leaves certain questions open, please feel free to ask us. We will try to give you the answers needed as fast as possible.
We wish you a lot of fun and success while developing your application!

2 Concept of programming with the GUI library

To allow for an application programmed as hardware independently as possible, the input and output of the user interface is realized in a two layer model. For each layer appropriate functions are provided. The terms used here are *Channel* for hardware independent functions and *Device* for the layer of hardware dependend functions (also see fig. 1 on page 7):

- For the application programmer it mostly suffices to use the hardware independent channels e.g. the *display channel*. These channels are logical representatives of the physical devices (and also of more abstract objects like a keyboard which can be a physical one or can also only exist on a touch display).

  The display channel for example is an abstract representation for graphical output devices like a LCD display oder a CRT.

  The channel driver provide hardware independent functions which are generally available for the input or output devices (e.g. for drawing a line on a display channel). The devices therefore shall be accessed using their respective channel as far as possible to avoid the necessity of modifications in the case the underlying hardware changes..

- In order for the functions executed for a channel to have some consequences on a real device, each channel is connected with a *device channel* representing the physical device. The connection function between channel and device driver realizes the interfacing of the hardware independent channel functions to the device currently in use (e.g. a LCD display). Specific device functions are accessible for the application programmer to use hardware dependend functions which cannot be realized in a generic manner with the channel functions. An example is the backlight of a LCD display which of course needs to be accessed from an application, but on the other hand is a hardware specific feature which should not be included into the abstract display channel functions.

  In the source code the names of the structures which resemble physical devices end with `_device`, for instance `t_lcd_device`. These denote a hardware driver, A listing of the devices available can be found in the file system.h at the definition of *connection types* (these are the possible ways of connections of the channels). The names ending with *_channel* denote logical channels.

For the function of the channels it is necessary for their connection partners (the devices, perhaps also further channels) to be initialized and available before the con-

nection is made and functions of the channel are executed. This is fairly simple. Mostly one only realizes the obvious connection and then the channel can be used.
Example: Initialization and connection of the display channel with the LCD device (further information is also provided in the application example (chapter 7), in the function reference and in the file "lcd.h"):

1. Initialize the LCD_Devices, to make it available:

   `lcd_init( virtual_x, virtual_y, orientation, &system_lcd);`

   (the LCD display can handle a larger virtual screen, which can be specified here. Depending on the lcd driver a rotated orientation of the display may also be available. The name of the LCD displays can be arbitrarily chosen (here chosen as `system_lcd`).

2. Open the display channel:

   `disp_open( &new_display_pointer);`

   (`new_display_pointer` then contains a pointer tho the structure which represents the new display channel).

3. Make the display channel the *current* display:

   `disp_make_current( &new_display_pointer);`

4. Connect the display channel with the LCD–device

   `disp_connect( system_lcd);`

This approach allows a fairly hardware independent programming style. Therefore further extensions and modifications can be realized without changing much of the application code. In addition it is also possible to change the data flow e.g. from an input device into files or into communication interfaces.
Remark: The devices, (the hardware dependend peripherals) are already present in the system as structures. In an application they are accessed by pointers, which get their correct value during the initialization. channels are allocated directly within the application and are normally also passed to functions by address.
In order to be independent from implementation dependend data types of different C–compilers only portable data types defined in the file "portab.h" are used.
The library uses Windows pixel fonts (versions 2 and 3), making it easy to obtain different fonts in addition to the three fonts which come with the library. Moreover SSF fonts can be used. This is a binary font format, which was developed for the use with the GUI–Library in order to support unicode characters. SSF fonts only hold the characters needed by an application, which leads to a very low memory footprint (see 3.4 on page 13 for details on SSF fonts).

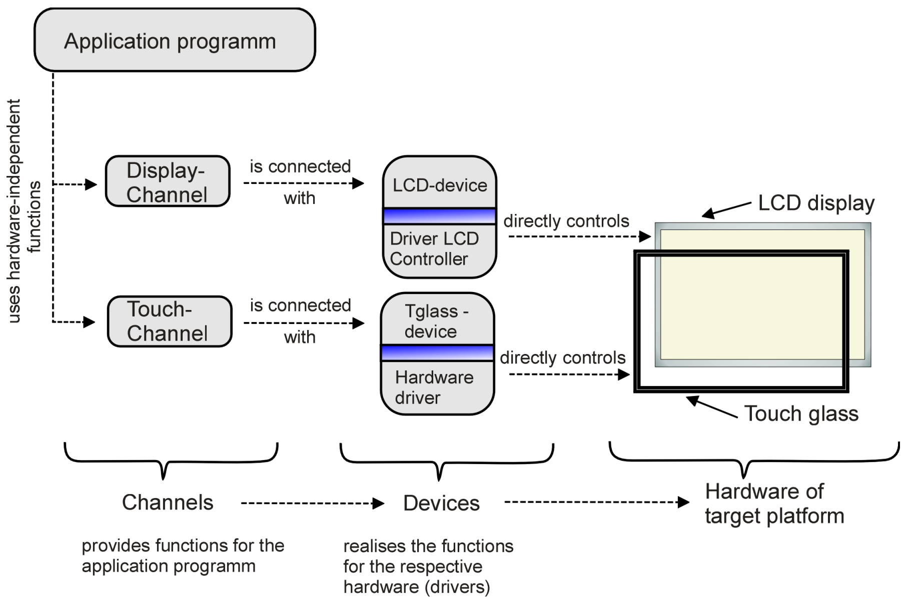


Figure 1: Concept of the programming model

For the user interface elements of the *touch version* of the library the appearance is defined by so-called *styles*. These are used to describe the look of an element and are passed to the respective functions when the user interface element is generated. This external treatment of the appearance of the elements has the following advantages:

- It is not necessary to define the properties for each similar button etc. once more but instead one can refer to a style once created.
- When the design of the application is to be changed the respective style can be modified at one place (e.g. a style used as a *standard style*) and all objects referring to that style get that modification.

3 Modules of the GUI library and programming

Here the modules of the GUI library are outlined in order to give an overall view of the function and possibilities of these modules and the available functions as well as their use. Detailled information about each function and how to use it is obtained from the function reference, and from the programming example in chapter 7 on page 61.

3.1 Function modules and header files

For each channel and for each device there is a function module. The functions available for the module and definitions are declared in the respective header file (for the compiler) and are commented additionally to the handbook (these header files also provide detailled information for the functions).
Each of the modules consist of a C–file with the implementation of the functions and two header files. The header file "<Modul name>.h" declares the structures of the channels and devices, the functions which are used for programming the application and the definition of Symbols for the application program.
The header files with the underscore "<Modulname>_.h" contain the declarations of internal functions of the library (functions which are not directly used by the application programmer).
The function of the device modules directly access the driver of the hardware used.
The header files of the modules for programming the application are:

display.h: Include File with the functions and structures of the display–channels (hardware independent). Normally connected with a LCD–device.

lcd.h: Include file for the specific functions of the LCD–Displays (device type: `lcd_device`)

touch.h: Include file for functions of touch channels. These allow to input positions from the display, e.g. by using a touch glass or a mouse. Touch channels normally are connected with the touch glass of the target hardware (thus with the corresponding `t_tglass_device`). Touch channels also have a connection with the display corresponding to their coordinates (the display on which typically the results of the input to the touch glass is shown).

tglass.h: Include file for the (hardware independent) functions of the touch glass used (device type: `t_tglass_device`).

button.h: Contains functions for realizing *Buttons*.

keyboard.h: Include file with the description of *keyboard channels* (generic keyboards). Keyboard channels are connected with keyboard devices, e.g. with a `t_keyb_Device`.

t_keyb.h: Include file for a touch keyboard realized on a touch glass. (These keyboards are treated as devices even they can be defined very in a very flexible manner as they are bound to the geometry of the touch display used).

sound.h: Include file for the functions of the *sound channel*. This channel usually is connected with a loudspeaker (e.g. a *piezo device*) on the target hardware.

piezo.h: Include file for the piezo device; this is a piezo loudspeaker provided by the target hardware. (The hardware driver called from within "piezo.c" of course could access another type of loudspeaker as well.)

system.h: Include file with definitions for the channels, devices. Additionally definitions of the fonts and corresponding functions, and the functions for the system timer.

The function names for programming the graphical user interface start with the name of the corresponding module, e.g. with `disp_...()` for functions for the display channel. Example: `disp_set_pixel()`.

In addition to these modules there also are the following important files:

struktur.h: Abstract data structures used within the library (double linked lists and FIFO memory)

portab.h: Definition of portable data types.

memory_wrapper.h: The user can switch between the standard memory allocator from "stdlib.h " and simple implementation from Simplify Technologies (see below). All files of the GUI library use the memory management macros of this header file.

simple_memory_wrapper.h: Simple memory allocator to replace `malloc()` and `free()`.

errors.h: Definition of error codes for the library functions.

text_gui.h: File for language dependend definitions of stings. With an appropriate define texts of different language can be chosen without modifying the source code itself.

config_gui.h: Configuration file for the adaption of the GUI library.

config_hardware_gui.h: Here definitions are made which describe the hardware environment of the user interface of the system. (see chapter 5 on page 34, too).

3.2 Properties of functions and variables

In order to support a programming style as hardware independent as possible the various integer types of ANSI–C which are implementation dependend have not been used. Instead we use types as defined in the file "portab.h" as for example `int16`, `uint8` (the first is a signed integer type of length 16 bit, the second is an unsigned integer with 8 bit). In "portab.h" additionally there are further type definitions like e.g. `t_color`, `t_font` and `err_code`. Depending on the colordepth of the display hardware the type `t_color` varies in size (1 byte in the case on 1 bit or 8 bit colordepth, 4 bytes if a colordepth of 24 bit is used).
The type `err_code` is used for the return value of the functions. All functions, which can produce errors, return an error code of type `err_code` which allows to test if the function was executed successfully or what kind of error occurred. The definition of the error codes is given in the file "errors.h". The return value `ERR_OK` shows that no error happened and the function was executed successfully. Functions supposed to return other values do so by using address pointers in their parameter list (this is also shown in the function reference with examples for respective functions). If a function does not deliver an error, the result is returned directly.

3.3 System functions

The system functions perform tasks which are central for the whole system. These are the functions for the system time and the administration of the fonts.
The system timer (if one is present) needs to be activated first. This and the initialization of the list of available fonts happens within the function *system_init()*. The system timer consists of periodic execution of the library function *system_process_timed_-functions()*. This is most conveniently achieved by using a timer interrupt. Details of the system timer can be found in chapter 5 on page 34.
The internal function *system_process_timed_functions()* provides for the periodic execution of various routines, e.g. for the periodic reading of the touch glass. Additionally the time elapsed since the start is counted. The system time is also available for use within the application, e.g. for realizing well defined delays.
The system time is realized by two 32–bit counters. The first counts the so–called *ticks* (e.g. 1/100 s) of the first 24 hours since the start, the second one increments the number of *days* since the timer has been started.

Survey of the system functions:

system_init: Initializes certain variables from the system. Also starts the system-timer, initializes the dynamic storage allocation and font list if applicable.
system_font_init: Initialize the data of a font.
system_font_close: Close a font and release the memory used.

system_get_number_of_fonts: Read the number of the currently available fonts.
system_get_font_pointer: Get the pointer to a font specified by it's name.
system_get_font_ascent: Get the number of pixels from the upper end of a font to it's "baseline".
system_get_text_ascent: Get the number of pixels from the upper end of a text string to it's "baseline". The used font and text style is taken into account.
system_get_font_pixelheight: Get the total height of a font (in pixel).
system_get_text_pixelheight: Get height of the font in pixel. The used font and text style is taken into account.
system_get_font_avg_width: Read the average width of the characters in the font.
system_get_font_max_width: Get the width of the broadest character of a font.
system_get_font_name: Get a pointer to the name of a font.
system_get_charwidth: Read the width of a specified character of a font (in pixel).
system_get_textwidth: Read the width of a text string in a certain font (in pixel).
system_wait_ticks: Wait a certain period of time before resuming program execution.
system_wait_until: Wait until a certain Time.
system_get_ticks: Read values of the system timer.
system_interval_elapsed: Checks if a given number of ticks are elapsed since a reference time.

3.4 Display channel

The functions for accessing the display are independent of the actual kind of device which is currently connected with the display channel. All pixel coordinates which are used by the functions are those of a *virtual* output device (which is the display channel).
The coordinates of the display are counted from the lower left corner, which means this lower left corner has the coordinates x=0, y=0.
Please note that a *clipping* is not provided for reasons of execution speed. This means that graphic instructions which would exceed the borders of the specified virtual display are not or not correctly executed, dependend on the graphic function.

Many functions are executed relatively to the current *graphics cursor*. This graphics cursor is not a cursor symbol appearing on the display but denotes a *current position* for drawing on the display. Many functions change the current graphics cursor (which means the position relative from which following functions work). The approach to work relatively to a current cursor position allows to easily work relatively to a start position chosen in advance and to easy change the position of the whole output by changing only this start position.

The functions influence only the *current* display (it is possible to have more display channels, one of which must be nominated to be the current display).
The effect of a graphical function is also dependend from *draw mode* The draw mode determines how the element to be drawn is combined with already existing graphics. Besides the standard mode `SET`, which simply replaces the background, there are available the logical combinations `OR`, `EXOR` and `AND`.

The functions for the display channel can be divided into the following categories:

Functions for the administration of displays: In order to use display channels and their functions the former must be opened first and be connected with a physical output device.

Functions for graphical primitives: Most of these functions are executed relatively to the current cursor position. Setting the cursor position itself can be performed absolutely or relatively to the former position. The basic functions for setting and reading pixels are available in a version which works with the current cursor position as well as in a version which works on absolute coordinates. This approach allows to show graphical structures on another coordinate by changing the position of the cursor instead of recalculating many coordinates.

Functions for drawing lines allow different line thickness and styles (e.g. solid or dashed). Choosing the line thickness works as follows: The lines are made broader relatively to the start coordinates if the line thickness is chosen to be larger than 1. For even values the line is drawn at the original coordinate and at the coordinate y+1 (or x+1 for lines with slope larger than 1). If the line thickness is increased to 3 then the line is now drawn at coordinates y-1, y, y+1 (or x-1, x, x+1 if the slope is larger than 1). For a line thickness of 4 it is additionally drawn at y+2 (or x+2), for line thickness = 5 it is drawn additionally at y-2 (or x-2) etc.

The line style (`t_linestyle`) is defined by a 16-bit value. A bit set in this value represents a pixel of the line set in the `current_color`; a bit cleared ensures that the background at this position is drawn with the current background color. The line pattern starts with the most significant bit and is repeated if the line element to be drawn is longer than 16 pixel.

Functions for working on rectangular areas: In order to work on areas on the display fill styles and *pixel blocks* are provided. Additionally, pictures in the widely used BMP format can be drawn. Fill styles are used to fill rectangular areas. Pixel blocks are memory cells in RAM which contain the display information of rectangular areas of the display. The pixels of these areas can be read from the display and transferred into the memory or vice versa. This allows to copy display information back and forth.

A pixel block is defined by a set of parameters. These mainly include the horizontal and vertical size of the pixel block as well as the color depth. The parameters can be set and read by the functions `disp_set_par_pixelblock()` and `disp_get_par_pixelblock()`, respectively. Then display data can be fetched into the pixel block (`disp_fetch_pixelblock()`) or written to the display (`disp_output_pixelblock()`).

Moreover, there are functions provided to draw frames, as used with buttons and other touchable objects (refere to sliders or menus in chapter 4 on page 24). The frames can be shaded and they can have round edges. The appearance of the frames is defined by a special *framestyle*.

More details of fill styles, pixel blocks and frames are given within the description of the respective functions in the function reference.

Functions for text output: Text can be displayed with various (monospaced or proportional) fonts. It can be modified with the usual attributes as "underlined" , "fat" , "italic" and "light" . For color displays with 8 bit color depth or more text can be output with anti–aliasing in order to obtain a smoother appearance. The text ist then output half the size of the original font size of the font chosen. If the attribute "light" is used with the color depth of 24 bit, the text is shown in an appropriate color and not rastered. The vertical output of text works only together with the attributes "light" and "inverted" .

For a description of the attributes also refer to the description of the function `disp_set_textstyle()` in the reference in the function reference.

The text functions use fonts in Windows 2.0 and 3.0 format (file extension FNT), to make it easy to get more fonts for the system. The FNT format is used mainly for ASCII-strings. The fonts are not part of the display channel but are administered centrally by the system.

The representation of UTF8 coded text–strings (e.g. for asian fonts) is realised in the following way: With our Windows tool ((*SimplifyCharacterCompiler*)) the user generates a so called SSF font, which is not restricted to 256 characters as FNT fonts. A SSF font contains any number of characters. For this a prefered Truetype font and a text file, which contains the used characters at least one time, are given to the *CharacterCompiler*. Internally, this data is used to assemble several FNT fonts, each with 256 characters, and a translation table, which assigns a FNT-font number and a character number to each unicode charcater used. The FNT fonts and the table are embedded in a SSF font, which can be saved as a header file for C or as binary data. The font functions (from "system.h/c" and "display.h/c") work together with both font formats. With the function `disp_text()` it is possible to print UTF8 coded strings on the display as long as the used SSF font (made as current font with the function `disp_set_font()`) contains the necessary characters.

Interaction with the functions of the touch channel: Please note that with the functions of the display channel the buttons and keyboards shown on the display can also be disturbed (e.g. the call to `disp_clear()` will optically erase active buttons from the display but their touch functionality will remain). Therefore before acting this way remove the respective touch element from the display in advance.

Survey of the display functions:

disp_open: Get the pointer of the new display channel to be created (if sucessful, otherwise NULL). For the display standard parameters are set.
disp_connect: Connects the current display channel to another channel or physical device.
disp_disconnect: Disconnects the current display channel from previous channel or device it was connected to.
disp_make_current: Make display characterized by a pointer the current display on which further display functions work.
disp_get_current: Fetch the pointer of the current display channel. (This function is only needed if there are more than one display channel used and one is interested in what the current display is).
disp_close: Closes and deaktivates the current display channel.
disp_update: Actualize current display channel (only needed for buffered displays like printers or displays where the graphics information is first processed in another memory area).
disp_save_properties: Stores the most important properties of the current display into an appropriate data structure.
disp_restore_properties: Sets the properties of the current display according to a data structure from the type "display_properties".
disp_reserve_bmp_colors: Reserve some colors of the color palette for use with bmp-pictures.
disp_set_palette_entry: Sets an entry in the color table.
disp_clear: Clears the display with the current background color.
disp_set_cursor_position: Sets the cursor position in absolute coordinates.
disp_get_cursor_position: Read the current cursor position.
disp_move_cursor: Move cursor relative to it's current position.
disp_set_pixel: Set pixel (in current color) at current cursor position.
disp_set_pixel_abs: Set pixel in current color at an absolute position.
disp_get_pixel: Read pixel color at current cursor position.
disp_get_pixel_abs: Read pixel color on an absolute position.
disp_line: Draw line from and relatively to the current cursor position.
disp_line_abs: Draw a line with the current attributes (line pattern and width) from a position given in absolute coordinates to another absolute position.

disp_hline: Draw horizontal line from and relatively to the current cursor position.
disp_vline: Draw vertical line from and relatively to the current cursor.
disp_rectangle: Draw rectangle relatively to the cursor postion.
disp_colored_rectangle: Draws a rectangle relatively to the current cursor position, which is filled with the current color.
disp_filled_rectangle: Draws a filled rectangle relative to the current cursor postion. Fillpattern determines how to fill, there is no border.
disp_colored_triangle: Draws a triangle at the given coordinates, which is filled with the current color.
disp_circle: Draw circle with the cursor position as the center and the specified radius.
disp_colored_circle: Draw a circle filled with the current color with the cursor position as the center and the specified radius.
disp_arc: Draw arc with the cursor position as center and the specified radius as the center between a start angle and an end angle.
disp_get_color: Read current color used in the current display channel.
disp_set_color: Set new color to be used on the current display.
disp_get_background_color: Read the current background color used in the current display channel.
disp_set_background_color: Set background color to be used in the current display channel.
disp_get_linestyle: Get current line style used on current display.
disp_set_linestyle: Set current line style (this is a 16 bit word for the line pattern used by line functions on the current display).
disp_get_linewidth: Get current line width used on current display.
disp_set_linewidth: Set new line width to be used on the current display.
disp_get_drawmode: Get current draw mode used on current display.
disp_set_drawmode: Set draw mode for current display.
disp_get_fillstyle: Read the currently used fillstyle of the current display.
disp_set_fillstyle: Set a new fillstyle for the current display channel.
disp_bmp: Displays a bmp picture at the current cursor position.
disp_bmp_alpha: Displays a bmp picture at the current cursor position. One color of the picture is declared transparent.
disp_pixelblock_init: Initialize a pixelblock data structure and gets the memory for it.
disp_pixelblock_free: Frees the memory used by the pixelblock.
disp_pixelblock_get: Copies a pixelblock from display at current cursor position to memory. Current cursor position is not changed.
disp_pixelblock_set: Output a pixelblock on the current display at the current cursor position. Current cursor position is not changed.

disp_invert_area: Inverts a rectangular area relatively to the current cursor position (which is not changed).
disp_color_change_area: Changes the color in a rectangle area.
disp_framestyle_init: Initialize a frame style data structure and gets the memory for it.
disp_copy_area: Command for copying rectangular areas on the display.
disp_framestyle_clone: Gets the memory for a framestyle and copies an existing one into it.
disp_framestyle_set_colors: Sets the color for the frame style.
disp_framestyle_set_dimensions: Set the frame width and the edge radius.
disp_framestyle_set_linestyle: Set the linestyle for the frame.
disp_framestyle_set_flags: Set the property flags for the frame style.
disp_framestyle_free: Frees the memory used by the frame style.
disp_frame: Draws a frame e.g. for buttons or other touchable objects.
disp_delete_frame: Overwrites a frame with the current color.
disp_printf: Provides the functionallity of the well-known C library function printf()
disp_text: Displays a text at the current cursor position according to the current text attributes (see also disp_set_textstyle).
disp_formatted_text: Like disp_text() but here the string can contain ”newline” (Hex 0A)” where then a new line is used.
disp_get_textstyle: Read the text attributes of the current display.
disp_set_textstyle: Set new text style for the current display.
disp_get_font: Get font pointer for text output on the current display.
disp_set_font: Sets a new font to be used for the text output on the current display.

3.5 LCD–Device

The module *LCD device* provides functions for the LCD display used by the target hardware. A feature of many LCD displays and LCD controllers respectively is that the display memory is larger than what is needed for the physical display area. This allows to work on a larger display area (*virtual display*) from which only a part (*viewport*) is shown on the LCD. This can come in handy for certain graphical applications like graphical maps or complex objects. A function for *scrolling* the part shown on the LCD is available, too.
If the LCD display is connected to a display channel it automatically realizes all functions of the display channel. This allows for a mostly hardware independent programming of graphical objects.
Additionally the LCD display has some extra functions which are not applicable for all kinds of displays, e.g. accessing an optional backlight of the display or adjusting the contrast. The LCD can also be switched on or off in order to save power while it

is not needed.

Survey of the LCD device functions:

lcd_init: Initialize the LCD display and return a pointer to the internal LCD device structure.
lcd_close: Closes the LCD display and calls lcd_sleep().
lcd_get_status: Get the status of the LCD device.
lcd_get_size: Get the size of lcd virtual screen.
lcd_get_color_depth: Read the color depth of the LCD.
lcd_set_viewport: Define origin of the visible display area (= physical display, "viewport") in coordinates of the virtual display.
lcd_orientation: Turns the display in steps of 90 degree.
lcd_light: Set the brightness of LCD backlight.
lcd_get_light: Get current brightness value of LCD backlight.
lcd_contrast: Set the new contrast for the lcd display.
lcd_get_contrast: Get the current contrast value set for the lcd display.
lcd_sleep: Switch the LCD display off to conserve power.
lcd_wakeup: Switch the LCD display on.

3.6 Touch–Channel

The *touch channel* is a hardware independent input channel for position dependend and touch sensitive input devices, e.g. for a computer mouse which allows to select objects by *clicking* on a certain position, or for a touch glass. The touch channel allows obtaining the coordinates and the system time of contacts (for the beginning as well as for the end of the contact). This allows for a broad range of functions like *double click* or *drag and drop* to be realized by software.

Survey of the touch channel functions:

touch_open: Initializes the touch channel.
touch_close: Close the touch channel.
touch_connect: Connect the touch channel with an input device (e.g. a tglass device).
touch_disconnect: Disconnects the touch channel form it's input device.
touch_update: Updates the touch channel (possible buffers will be flushed).
touch_get_contact_end: Read the end coordinates of the last contact on the touch channel
touch_get_contact_start: Read the start coordinates of the last contact on the touch channel.
touch_process_objects: Tests all the touchable objects against the actual touch position and timestamp.

Figure 2: Example for simple touch functions

3.7 Buttons

An element often needed within graphical user interfaces is a *button*. This is defined as a touch sensitive area with a border and a label of text or graphics which can be used as a switch.
There are different versions of buttons available. By choosing the parameters accordingly they can be realized in different sizes, with different borders and labels (text or graphics). Their appearance is defined by a *style*. Additionally they can exhibit different switching behavior: One button can be activated permanently while pressed (`FAST_BUTTON`), another is only *activated* when it is released (`STANDARD_BUTTON`), or it can lock into one of two states, *pressed* and *not pressed* (`HOLD_BUTTON`). In the two latter cases the activation of the button is triggered not by touching it, but by releasing it above the button area. The user of a device thus can *cancel* an invalid touch by just releasing the touch screen outside of the button area.

Please note the following aspects when using buttons:

1. If a button is drawn on the display, there are two different states of appearance: the pressed state and the non pressed state. Both of them can have a complete different look in respect of the colors or the frame style.

2. The size of the button must be big enough to hold the frame together with the button content (text or picture). For instance, if you increase the width of the frame, there is maybe not enough space left for the button text.

3. A button can be labeled with vertical text by using the text style `TXT_VERTICAL`.

Survey of the button functions:

button_style_init: Gets memory for a button style and initializes it. Button styles are used for the definition of the appearance of buttons. They thus allow to define the look of many buttons (the ”style”) at one place. This function must be called before a new style is adjusted with further function calls.
button_style_clone: Gets memory for a new button style which is copy of an old one.
button_style_free: Releases the memory used by the button style.
button_style_set_font: Set a font and text style for a button style.
button_style_set_linefeed: Set text linefeed for button style.
button_style_set_alpha: Sets the alpha color for picture buttons with transparency.
button_style_set_framestyle: Set the frame style for the button style.
button_make_combined_label: Generates a combined button label, which consists of a background image and a text label.
button_free_combined_label: Frees the memory used by the combined label.
button_define: Defines a new button.
button_undefine: Deletes a button and releases the used memory.
button_press_hold_button: Set hold button in the pressed state.
button_lift_hold_button: Set hold button in the lifted state.
button_ghost_button: Deactivates a button on the touch channel and displays the text lable ”ghosted”.
button_unghost_button: Reactivates a button on the touch channel which has been ”ghosted” previously. The *normal* label is displayed, i. e. the text is shown with the text style TXT_NORMAL.
button_activate: Activates a button defined before on a touch channel and displays it on the associated display channel.
button_deactivate: Deactivates a button on a touch channel and deletes it from the corresponding display channel.
button_get_status: Read the status of a button.
button_get_hold_button_status: Get the status of an hold button.
button_get_position: Read position of a button.
button_set_position: Set position of a button.
button_get_label: Read the two labels of a button.
button_change_label: Change the label (text or graphics) of a button.
button_set_style: Set the style/appearance of a button to the new properties of a given buttonstyle.

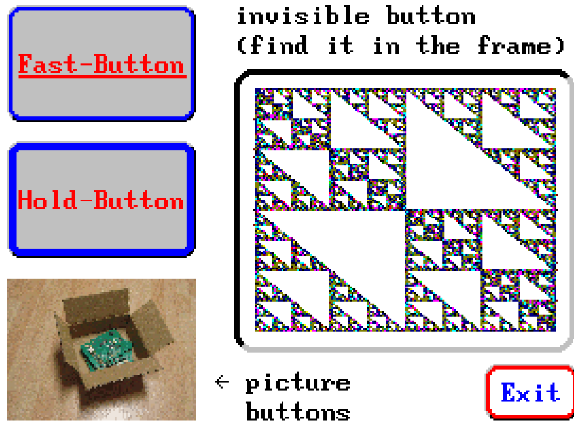


Figure 3: Example for various buttons

3.8 Tglass–Device

The module *tglass device* provides functions which are specific for a touch glass assembled into the hardware of a system. For calibrating the touch glass the function `tglass_calibrationsequence()` is provided. Sometimes after once calibrating the touch glass further calibration is not needed. The text used by the calibrating function is provided in the file "text_gui.h" in English and German language. More details on how to calibrate a touch glass can be found in section 5.5.1 on page 49 and in the "Quickstart Tutorial".

Survey of the Tglass device functions:

tglass_init: Initialize and activate the tglass device.
tglass_close: Close the tglass device.
tglass_set_calib_parameters: Overwrites the current calibration parameters of the touch glass with new values.
tglass_get_calib_parameters: Read the current calibration parameters of the touch glass.
tglass_calibration_sequence: Calibrate the tglass–device and therefor the touch glass.

3.9 Keyboard–Channel

Often text input is needed for IT applications. In order to provide various possibilities for text input in a hardware independent manner *keyboard channels* are im-

plemented. These are connected on the target hardware with touch keyboards on the touch display (one could connect them to external hardware keyboards as well without changing the application which uses the keyboard channel functions. The keyboard channel features a keyboard buffer from which the ASCII characters are read.

Survey of the keyboard channel functions:

keyboard_open: Initializes a keyboard channel.
keyboard_connect: Connect keyboard channel with an input device.
keyboard_disconnect: Disconnects a keyboard channel from an input device.
keyboard_update: Updates the keyboard channel (possible buffers will be flushed).
keyboard_close: Closes a keyboard channel.
keyboard_getchar: Read ASCII character from the keyboard buffer.
keyboard_get_status: Read the status of the keyboard buffer of the keyboard channel.

3.10 T_keyb device

The touch keyboard device (`t_keyb_device`)is a touch keyboard which is displayed on a touch display (more exactly: a touch channel). Even though the `t_keyb_device` only consists of software it finally accesses the hardware provided and is thus denoted a device here (and treated as if it was a hardware component). Normally a keyboard channel is not connected to an external keyboard on a hardware featuring a touch display but rather with this `t_keyb_device` (as if it was a piece of hardware).
The flexible possibilities of the functions of the `tkeyb_device` allow for easy configuration of various touch keyboards for different applications. e.g. alphanumeric keyboards, number blocks and other application specific keyboards. The appearance of a touch keyboard is defined by a *style*.
Like with *large* hardware keyboards repeat functionality for the automatic repetition of key contacts is provided, as also is the possibility to choose between different characters by means of selecting a *mofifier key*: `SHIFT`, `ALT`, `CTRL`, `CAPS_LOCK`.
If needed by the application it is also easy to move the touch keyboard on the display.

Survey of the T_keyb device functions:

t_keyb_style_init: Initializes a touch keyboard style and gets memory for it.
t_keyb_style_free: Frees the memory of the keyboard style structure.
t_keyb_style_set_font: Set the font for a touch keyboard style.
t_keyb_style_set_changeflag: Sets the change flag for changing of the keylabels if a special modifier key is pressed (e.g. shift).

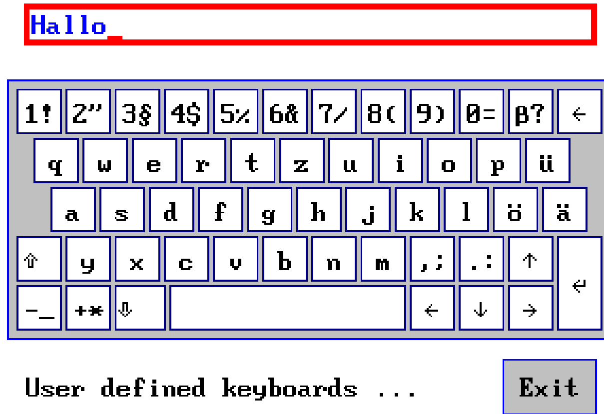


Figure 4: Example for keyboard

t_keyb_style_set_colors: Set the colors for a touch keyboard style.
t_keyb_style_set_select_colors: Set the additional colors for a touch keyboard style for the selected state of the keys.
t_keyb_style_set_border_width: Set the border width of the keyboard style.
t_keyb_make: Initialize a touch keyboard.
t_keyb_free: Frees the memory of the keyboard.
t_keyb_open: Opens a touch keyboard on a touch channel
t_keyb_close: Close a touch keyboard.
t_keyb_process: This function performs the operation of a keyboard.
t_keyb_get_position: Read the position of a touch keyboard.
t_keyb_set_position: Move touch keyboard to a new position.

3.11 Sound–Channel

For all–day use often an acoustic signal is needed, e.g. for providing feedback to the user for a touch on the touch display (*klick*), or to otherwise attract the attention of the user. Simple hardware independent sound functions are provided which allow the output of sound.

Survey of the sound channel functions:

sound_open: Open sound channel.

sound_connect: Connect a sound channel with another channel or an output device (loudspeaker etc.).
sound_disconnect: Disconnect the connection between a sound channel and another channel or device (loudspeaker etc.).
sound_close: Close sound channel.
sound_update: Flush buffers if applicable in the chain to the output device.
sound_set_volume: Set a new value for sound volume of a sound channel.
sound_get_volume: Read current volume setting of a sound channel.
sound_make: Generate sound on a sound channel.
sound_on: A sound is switched on on the sound channel.
sound_off: A sound is switched off on the sound channel.
sound_click: Generate a "keyboard click".

3.12 Piezo–Device

In a target hardware often the output of sound is realized with a compact piezo loudspeaker. This is (after initialization, of course) is accessed from a sound channel.

Survey of the piezo device functions:

piezo_init: Initializes the internal piezo loudspeaker.
piezo_close: Closes the piezo loudspeaker (piezo device), which then is no longer available.

4 Additional modules (*AddOns*)

The *AddOns* are additional modules, which enhance the core functionality of GUI library. The corresponding source files and the demos are in the respective directories on the CD of the GUI library as discriped in the file “Content.txt”.

4.1 Seven segment display

The module “seven_segments.h/.c” provides functions for displaying seven segment numbers (with decimal numbers and “E”). The displayed numbers can be scaled to any size and are optimized for fast output to allow rapid changes of them. The function, which draws the number, needs a string as input. All invalid characters in the string are ignored during drawing of the number.
A detailed example can be found on the CD (q.v. the function reference).

Survey of functions for the seven segment display:

seg7_init: Initialize a segment display info structure and gets memory for it.
seg7_free: Releases the memory used by the info structure of the segment display.
seg7_set_colors: Sets the color of the digits.
seg7_get_width: Gets the width of the whole segment display in pixels.
seg7_define: Draws the initial segment display on the screen.
seg7_update: Updates the contents of a segment display.

4.2 Inputline

This module “inputline.h/.c” supports the user with the programming of inputlines, as they are used together with touch keyboards, for instance. The provided functions (see the function reference) are responsiple for the graphical representation of the inputline. They take care of the insertion/erasure of characters, the scrolling of entered string, the movement of the text cursor and so on. The string can be bigger as the inputline and will be scrolled if necessary. The inputline uses the standard frames as they are defined in “display.h/.c”. A detailed example can be found on the CD.

Survey of inputline functions:

inputline_style_init: Initialize an inputline style data structure and get memory for it.
inputline_style_free: Releases the memory of an inputline style.
inputline_style_set_frame: Sets properties of the border of an inputline.
inputline_style_set_colors: Sets the colors of an inputline style.
inputline_style_set_font: Sets the font and the text style of an inputline style.

Figure 5: Example for seven-segment display

inputline_define: Defines a new inputline by reserving some memory and initializing the data structure. Moreover the inputline is shown on the display.
inputline_undefine: Deletes an inputline and the used memory. The inputline is earsed from the display.
inputline_draw: Draws the inputline.
inputline_blink: Blink the inputline.
inputline_set_cursor: Sets the cursor to a new position (to the next character according to a pixel position, which is given).
inputline_move_cursor_right: Moves the cursor one character to the right.
inputline_move_cursor_left: Moves the cursor one character to the left.
inputline_move_cursor_left_border: Moves the cursor to the left border of the input line.
inputline_move_cursor_right_border: Moves the cursor to the right border of the input line.
inputline_delete_char: Deletes one character of the text-string at the current cursor position.
inputline_delete_string: Deletes the complete string from the inputline.
inputline_insert_char: Inserts a character at the current cursor position.
inputline_insert_string: Inserts a string at the current cursor position.
inputline_set_string: Sets the text of the inputline to a specific string.
inputline_get_cursor_position: Gets the current cursor position.
inputline_get_char: Get a character at a given current cursor position.
inputline_read_string: Read only access to the text of the inputline by copying it.

4.3 Progress bar (*bar indicator*)

The files “bar_indicator.h/.c” provide functions for displaying and controlling of progress bars or bar indicators, as they called in the GUI library. The bars are surrounded by standard frames as they are defined in the files “display.h/.c” . The bar indicators can be vertical or horizontal orientated and their start and end value can be chosen arbitrarily. The user has the possibility to display the actual value together with a physical unit (like temperature e.g.) inside the bar. Moreover, the indicator bar can change its color, if the value exceeds a certain threshold. This feature can be used to indicate a dangerous condition, for instance.
An extensive example can be found on the CD (q.v. the function reference).

Survey of bar_indicator functions:

barind_style_init: Initializes a bar indicator style and gets memory for it.
barind_style_clone: Gets memory for a new bar indicator style, which is a copy of an old one.
barind_style_free: Frees the memory used by the bar indicator style.
barind_style_set_frame: Set the frame style for the bar indicator style.
barind_style_set_font: Set font and text style and color for a text label of the bar indicator.
barind_style_set_bar_appearance: Sets the appearance of the bar indicator (colors and patterns).
barind_define: Creates a bar indicator with some defaults and some user defined main values.
barind_undefine: Undefine specified bar indicator (also free memory used).
barind_activate: Displays the bar indicator on the display.
barind_deactivate: Erases the indicator bar from the display.
barind_set: Sets the bar indicator to a new value.

4.4 Slider

The module “slider.h/.c” has functions to make sliders (see function reference). the slider behaves similar to other graphical user interfaces and can be moved by pressing the touch–glass. The returned value of such an event, is between the previously defined limits, which in turn do not depend on the physical size of the slider.
The slider features the standard frames from “display.h/.c”. The slider belongs to the group of *thouchable objects*, which can be controlled by the same functions as buttons for example (see `touch_process_objects()` in the function reference). A simple means of controlling the *touchable objects* is presented in section 4.7. A documented example of the usage of the slider functions can be found on the CD.

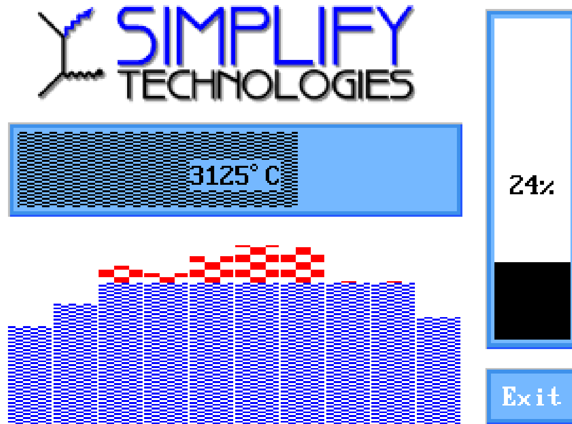


Figure 6: Example for bargraphs

Survey of slider functions:

slider_style_init: Initialize a slider style data structure and get the memory for it.
slider_style_clone: Gets memory for a new slider style, which is a copy of an old one.
slider_style_free: Release the memory of a slider style.
slider_style_set_framestyle: Sets frame style of a slider style.
slider_style_set_colors: Sets the colors of a slider style.
slider_define: Defines a new slider by reserving some memory and initializing the data structure.
slider_set_position: Set the slider to a new position.
slider_get_position: Gets position of a slider.
slider_redefine: Set a new slider range, size and position.
slider_undefine: Deletes a slider and the used memory.
slider_activate: Activates a slider and shows it on the display.
slider_deactivate: Deactivates a slider and erases it from the display.

4.5 Selection list

The module “selection_list.h/.c” has functions to make selection lists (see reference in the function reference). The list can be scrolled using the touch–glass and single or multiple entries of the list can be selected. Every entry consists of an icon as well as a left adjusted text and a right adjusted text. Moreover there is a variant of the

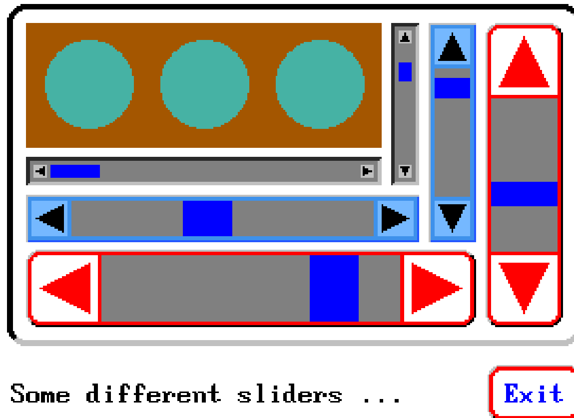


Figure 7: Example for sliders

selection list, which only displays some data. A documented example of the usage of the selection list functions can be found on the CD.

Survey of selection list functions:

sl_item_style_init: Initialize a style for the items of a selection list and get the memory for it.
sl_item_style_clone: Gets memory for a new style of the items of a selection list, which is a copy of an old one.
sl_item_style_free: Release the memory of a style of the items of a selection list.
sl_item_style_set_font: Set font, textstyle and color for a text label of an item style.
sl_item_style_set_colors: Sets the remaining color(s) of the item style.
sl_style_init: Initialize a style for the selection list and get the memory for it.
sl_style_clone: Gets memory for a new style of the selection list, which is a copy of an old one.
sl_style_free: Release the memory of a style of the selection list.
sl_style_set_slider: Sets the parameters of the slider, used in the selection list.
sl_style_set_gap: Sets the gap between icon and text of an item and the gap between the items and the slider.
selection_list_define: Defines a new selection list.
selection_list_undefine: Deletes a selection list and the used memory.
selection_list_activate: Activates a selection list and shows it on the display.
selection_list_deactivate: Deactivates a selection list and erases it from the display.

Figure 8: Example for selection list

selection_list_add_item: Adds an item to the selection list.
selection_list_remove_item: Removes an item from the selection list.
selection_list_count_items: Counts the number of items of the selection list.
selection_list_set_position: Sets or scrolls the selection list to a new item position.
selection_list_scroll_up: Scrolls the list up by one line.
selection_list_scroll_down: Scrolls the list down by one line.
selection_list_item_set_status: Sets the status of one item.
selection_list_get_last_event_item_status: Gets the status of the item, which caused the last event.
selection_list_get_select_item: Gets the index and the status of the last selected/deselected item.
selection_list_get_focus_item: Gets the index and the status of the last focused item.

4.6 Menus

There are three kinds of menus:

- *Popup*–Menu: This menu can be displayed programmatical and vanishes after the selection of one menu–entry or a touch contact outside of the menu (see “popup_menu.h/.c”).
- *Single*–Menu: This menu is a combination between a popup–menu and a button for the activation of the menu. The menu disappears after the selection of

an entry, touching the activation–button or a touch contact beside the menu (see “menu.h/.c”).

- Menu–*Bar*: A combination of several single–menus, whereas all the activation–buttons are lined up in one bar. If a button is selected the corresponding menu appears directly above or under the activation–button (see “menu.h/.c”).

The menus utilize the standard frames as defined in “display.h/.c”. The menus are *touchable objects* as well, which can be processed by the same functions as the buttons or the sliders (see `touch_process_objects()` in the function reference). Applicable examples are on the CD.

Survey of popup menu functions:

popup_style_init: Gets the memory for a menu style data structure and initializes it.
popup_style_free: Releases the memory of a menu style.
popup_style_set_frame: Sets the frame style for the popup style.
popup_style_set_separator_height: Sets height of the line (separator) between, which can be set between two menu entries.
popup_style_set_font: Sets the font and the appearance of the text of a menu style.
popup_define: Initializes the popup menu and gets memory for it.
popup_add_entry: Adds a new entry to a popup menu, the menu must be hidden.
popup_remove_entry: Erases a menu entry from a popup menu. The menu must be hidden.
popup_activate: Activates and displays the popup menu.
popup_deactivate: Erases menu from the display.
popup_get_dimensions: Gets the size of the popup menu.
popup_entry_was_pressed: Tests whether an entry of a popup menu was pressed.
popup_set_flags: Sets the flags of the popup menu.
popup_get_flags: Gets the flags of the popup menu.
popup_set_entry flags: Sets the flags of an popup menu entry.
popup_get_entry flags: Gets the flags of an popup menu entry.
popup_undefine: Erase the popup menu from the screen and deallocates the data structures.

Survey of menu functions:

menu_define_single: Initializes the single menu and gets memory for it.
menu_undefine_single: Erases the single menu from the display and deallocates the data structures.
menu_activate_single: Displays the activation button of the single menu and inserts the single menu in the list of touchable objects.
menu_deactivate_single: Erases a single menu from the display.

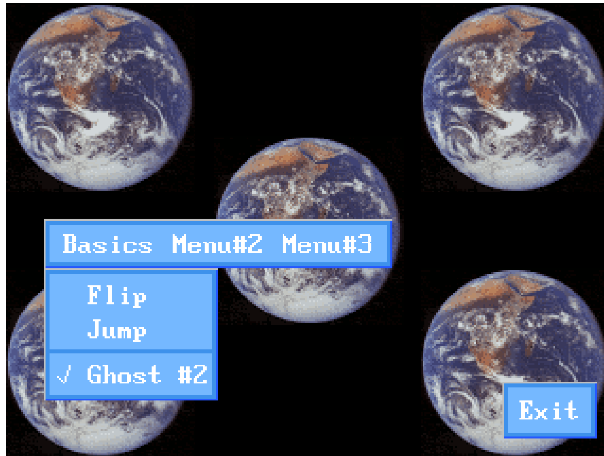


Figure 9: Example for menu

menu_single_entry_was_pressed: Tests whether an entry of a single menu was pressed.
menu_define_bar: Initializes the menu bar and gets memory for it.
menu_add_bar_popup: Add a new popup menu, which will form one of the single menus of the menu bar.
menu_activate_bar: Show the menu bar.
menu_deactivate_bar: Removes the menu bar from the display.
menu_undefine_bar: Removes the menu bar from the display and deallocates the data structures.
menu_bar_entry_was_pressed: Tests whether an entry of a menu bar was pressed.
menu_bar_set_flags: Sets the flags of the menu bar.
menu_bar_get_flags: Gets the flags of the menu bar.

4.7 Event handling

The function of the module "event_handling.h/.c" aim for simplifying the processing of touchable objects (GUI elements, which are controlled by the user, who presses a touch–glass) (see the function reference). This is done by initializing an event queue upon startup of the event handling. This queue holds all touch–event, which where triggered by the user. The events in the queue are evaluated one after another in their order of appearance. They are erased from the queue directly afer evaluation to clear up the occupied memory. However, there is the possibility, that more events appeare, than can be processed within a certain time. In such a case some events have to be dropped. It is advicable to choose the size of the event queue depending on the

application. Up to now, the following touchable objects are supported by the event handling: buttons, keyboards, sliders, menus and selection lists.
A simple application to demonstrate the event handling can be found on the CD.

Survey of event handling functions:

events_activate: Activates the event handling for touchable objects like buttons.
events_is_active: Tests whether the event handling is activated.
events_deactivate: Deactivate the event handling.
events_collect: Collects new events by processing the list of touchable objects, which can generate events.
events_append_event: Inserts a new event into the event queue.
events_get_next_event: Read one event from the queue and delete it.

4.8 Miscellaneous

4.8.1 Clipping

The functions in "clip.h/.c" are testing, whether a line intersects a given rectangle. Whereupon the start and end points of the line are adjusted, so that they are lying on the border of the rectangle (see function reference and the example on the CD).

Survey of clipping functions:

clip_point: Tests a point against the clipping area.
clip_line: A line is clipped against the clipping area and the points of the line are adjusted according to the boundaries.
clip_text: Draws a text in a clipping rectangle.

4.8.2 Random numbers

The function, which generate (pseudo) random numbers (cp. function reference) can be found in the files "random.h/.c". Befor using the random numbers the generator must be started with any numeric *seed*. An example can be found on the CD

Survey of functions for random numbers:

rnd_init: Initializes the random generator, which has to be started before using random numbers.
rnd_32: Returns a 32bit random number.
rnd_number: Returns a random number between 0 and (`max`-1), `max` must be greater or equal 1.

4.8.3 String functions

Right now, there is only one function for the conversion of integer numbers to strings implemented here (compare the function reference and the files "string_addon.h/.c").

Survey of string functions:

int_to_str: Converts a number into a string and adds a unit-string.

5 Adaption of the library to the target hardware

This chapter describes the adaption and configuration of the Simplify Technologies GUI library to your application and target hardware as well as the function interfaces for the hardware drivers.

In order to use the Simplify Technologies GUI library on a specific target hardware it is necessary to provide certain definitions which describe the target hardware. These definitions are made in the respective header files. Under some circumstances a few additional functions need to be provided to realize complete adaption to your hardware.

5.1 Requirements for the use of the GUI library

The Simplify Technologies GUI library allows programming of graphical user interfaces in ANSI–C. Thus it can be used on very different hardware architectures.

Interfacing of the hardware components, e.g. LCD displays in addition requires hardware dependent driver functions which are described below. The amount of memory needed for the library depends on the kind of microprocessor used as well as the tool chain used and of course on the amount of functionality needed for your application. Example: For a Hitachi H8/300H microcontroller and a GNU–C–Compiler a very small application ("Hello World") needs less than 20 kByte ROM and far less than 1 kByte RAM, including functions for the fonts and the small font "Mono6x8". The memory footprint also includes runtime libraries from the C toolchain which typically are linked to your application code anyway. Typical execution times for the Hitachi H8/300H microcontroller with 14.7456 MHZ: A 320×240 Pixel LCD display can be written with text strings using fonts of various sizes and attributes in less than a second.

Depending on the functions needed different requirements are to be met by the system environment:
For the functions for the LCD display there are no further requirements if an automatically blinking sprite cursor in not needed. Administration of the fonts can be done dynamically if your system provides dynamic memory allocation (the possibility to use the `malloc()` function), in order to add fonts during run time. Otherwise the fonts can also be used without the need for `malloc()`.
If you want to use an automatically blinking sprite cursor a system timer is needed, which ensures the periodic execution of certain functions. This also needs dynamic memory allocation (`malloc()`).

The use of a touch glass necessarily needs a system timer because evaluating a contact on a touch glass involves to distinguish between *before* and *after* the contact. Dynamic memory allocation is then needed, too.
The functions for sound output (e.g. the use of a keyboard click with touch keyboards) need a loudspeaker with a corresponding driver.

If the requirements for using the library even though they were held low pose a problem for your kind of application we will be glad to help.

5.2 Survey and short description of the configuration

In order to be able to work with the Simplify Technologies GUI library as quickly as possible, we recommend the following approach:

1. Copy the files of the GUI library and the hardware drivers to an arbitrary place on your hard disk (to any place you feel appropriate for your project). Unpacking and installation of the files is described in the file “INSTALL.TXT” which comes with the library.

2. Add all files of the library to your project. How to do this or how to adjust the path to the files is described in the documentation of your programming tool chain.

3. Adapt the file “portab.h” to the properties of your tool chain: Because the ANSI–C standard does not feature fixed length integer variables but rather requires some minimum length, the library only uses the integer types defined in “portab.h” as `uint8`, *int8*, `uint16` etc. (the number denotes the number of bits needed for the variable). The type `t_color` (siehe “display_colors.h”) must be adapted to the color depth of the used harware (see the file “config_gui.h” too). For monochrom and 8 bit color depth choose `t_color = uint8`, for higher color depths `t_color = uint32` should be used. What the size of the integer types of your compiler is can be found in the file “limits.h”, which is included with your tool chain. Please change the definitions in “portab.h” accordingly.

4. Be sure to make available all hardware dependend driver functions for your project. The functions needed (and also those needed for integrating the library into your system) are described in detail in chapter 5.4 on the next page.

5. Configure the library in the files “config_gui.h” and “config_hardware_gui.h” . This adjusts the library to the properties of your system and also optimizes what functions are compiled. More detailed information for this configuration is found in section 5.5 on page 48.

6. Now you are ready to utilize the functionality of the GUI library in your application programs. Examples can be found in chapter 7 on page 61 (a generic example showing most of the features) and in driver modules, the "Quickstart Tutorial" and in the *AddOn*–expamles coming with the library.

A step–by–step example for the configuration starting from simple to more complex applications is provided in the "Quickstart Tutorial".

5.3 Definitions for the programming tool chain used

The library is written in ANSI–C. The ANSI–Standard only defines minimum sizes for integer variables. In order to avoid problems with the use of different compilers the library only uses integer variables with fixed length. These are defined in the file "portab.h". These definitions need to be adjusted once to the compiler used.
The GUI library does not need floating point arithmetics.

5.4 Driver for hardware and system environment

Here the functions are introduced which directly access your hardware as hardware drivers. These functions thus must be adapted to the hardware. Their declaration is made in the files of type "<*modulname*>_.h" , from the module which accesses them For drivers provided with the library the functions themselves reside in C files. Additionally a few functions are needed to integrate the library into the system used.

Depending on the functionality needed and the system properties the following functions are needed.

5.4.1 Functions for the realization of a system timer

If only simple LCD display output is what you need, it is not necessary to provide the following functions. They are only needed for automatically blinking cursors and if you need the functionality of a touch glass.

A **system timer** is required, if you either have a touch screen which needs to be read periodically or if you want to use automatically blinking sprite cursors. The system timer is a function which is periodically executed on a regular basis (which we want to call `SYSTEMTIMERFUNC` here) and which then calls the library function `system_process_timed_functions()`. The system timer can be triggered by a timer interrupt for example. The GUI library then automatically realizes the blinking sprites and the functions needed for the touch glass by means provided with the function `system_process_timed_functions()`.

The system timer also is required to provide a *system time* which counts the number of calls to `system_process_timed_functions()` in two global counter variables **TIMERDATA** and **DAYDATA** (both are of the type `uint32`) in the following way:
`TIMERDATA` needs to be incremented by the system timer until a period of 24 hours is over. Then `TIMERDATA` must be reset to 0 and `DAYDATA` has to be incremented by 1. These variables are also evaluated by the functions `system_wait_until()` and `system_wait_ticks()`, which can be used by your application. When the system starts both variables are set to 0 by the function `system_init()`. Figure 10 shows a structogram of the system timer function.

`SYSTEMTIMERFUNC ---` Interrupt function of the system timer to be implemented

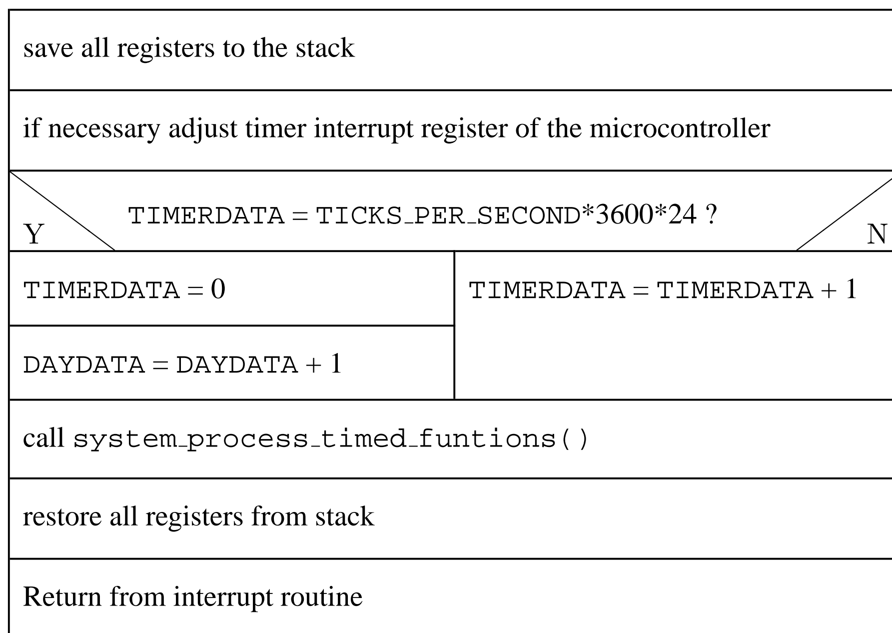


Figure 10: Structogram of the timer interrupt routine

The funktion `system_init()` additionally calls the function **os_systemtimer_start()**, which you must provide. This function needs to be implemented such that it starts

the system timer (this mostly means to start the timer interrupt). The call to this function is automatically executed from `system_init()` at the beginning of the application program.
To prevent `TIMERDATA` to overflow before the carry to `DAYDATA` occurs, the system timer may not run with a frequency higher than 49710 Hz.

If the call to `system_process_timed_functions()` happens asynchronously to the normal flow of the application program (as with the use of a timer interrupt for the system timer) the following has to be taken into account:
In order to control access to certain structures or hardware components for blinking sprites and touch glasses simple semaphores (which we now call *flags*) are needed. These flags are 8–bit variables which have two states. They can be `FLAG_FREE` or `FLAG_BLOCKED` (as defined in "os_gui.h"). The flags needed for the GUI library are already defined there and do not need to be adapted further. It nevertheless is necessary to provice the following functions to ensure that the flags can be set and reset safely, which means without being disturbed by other (interrupt-) functions:

err_code os_flag_block(uint8 *flag)

This function must block the timer interrupt (by calling of `os_disable_timer_interrupt()` (see below)). Then the specified flag is checked and set to `FLAG_BLOCKED`, if it has been `FLAG_FREE` in the beginning. Then the blocked interrupt is released again (by calling `os_enable_timer_interrupt()` (see below)).

Return values: `ERR_OK`, if the flag has been free and could be blocked successfully, `ERR_FLG`, if it was already blocked.

err_code os_flag_free(uint8 *flag)

This function sets the flag to `FLAG_FREE`, and thus releases the blocked ressource. Reading the flag itself may not be disturbed by other interrupts. Thus such interrupts need to be disabled in advance and to be activated after reading the flag.

`os_flag_free()` is and may only be used if the calling function had blocked the flag for it's use and therefore has the *right* to access the flag.

Return value of the function: `ERR_OK`.

The following functions switch the timer interrupt on or off respectively. They are needed for the secure handling of the semaphores and for internal purposes of the function `system_process_timed_functions()`. Please implement these functions according to your hardware / microcontroller:

void os_disable_timer_interrupt(void)

`os_flag_block() ---` Function which sets the timer interrupt flags (to be implemented)

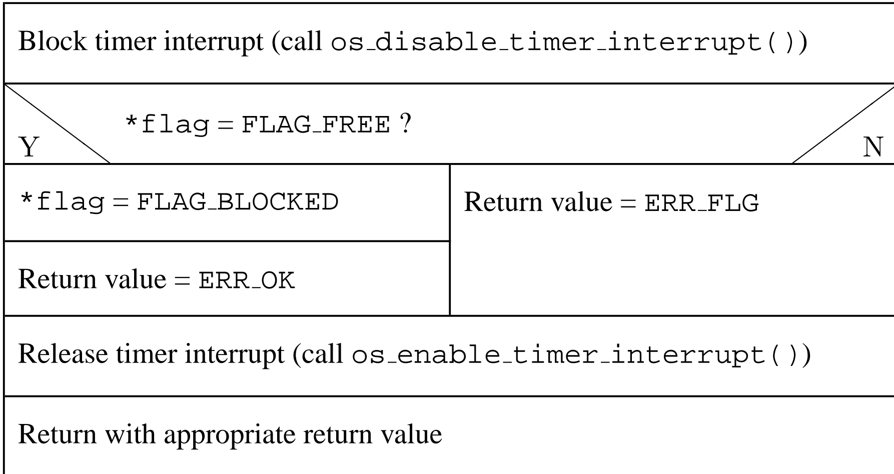


Figure 11: Structogram of `os_flag_block( uint8 *flag)`

Disable the system timer interrupt.

void os_enable_timer_interrupt(void)
Enable the system timer interrupt.

To be able to use waiting periods for other purposes the following function call is provided:

void os_process(void)
This function is called from within the `system_wait`–functions, to allow your system to fullfill additional tasks while waiting. If this is not necessary you can omit the definition of `_OS_USED` in the file “config_gui.h” and choose not to implement the function `os_process()` as a dummy.

The declaration of these functions and of constants which are used by these functions are made in the file “os_gui.h”. Implementation of these function is recommended to be in assembler to have full control of the interrupt handling.

5.4.2 Driver for LCD display

LCD displays usually are driven by dedicated LCD controllers which are mapped into the address space of the microcontroller. These addresses of the LCD controller need to be available to the driver of the LCD display. **The driver with the functions described in the following can be obtained as a component for the library. Details of the driver are given in the driver documentation**. Alternatively the driver can be programmed by using the following software and function interface (declarations in "lcd_.h"):

The size of the types `t_color` and `t_color_intrep` depend on the color depth of the used hardware. They are defined in the file "display_colors.h". The type `t_color_intrep` is used for the internal representation of one pixel, e.g. with a color depth of 8 bit `t_color_intrep` = `uint8`).

The ending of the rendering functions are in accordance to the orientation of the display content. `_0` is used for the normal orientation (⇔ rotation of 0°), at the rotation of the display content by `180`° the ending `_180` is appended. All other orientations are build by described function and can be found at an higher level of abstraction. The functions with the ending `_180` are not necessary if the LCD controller supports hardware rotation of the display content.

Depending on the color depth the call of the driver functions might be different. If no details are made the driver function has to be provided for all color depths.

void c_lcd_init(uint16 virtual_x, uint16 virtual_y)
: Initializes the LCD display and switches it on. If the display supports the use of virtual screens a virtual screen is configured with the size of `virtual_x` and `virtual_y`.

void c_disp_on(void)
: Switches on the LCD display.

void c_disp_off(void)
: Switches off the LCD display.

void c_lcd_sleep(void)
: Switches the LCD display into sleep mode (if available) in order to save energy.

void c_lcd_wakeup(void)
: Activates LCD display in order to wake it up.

void c_lcd_set_viewport(uint16 new_origin_x, uint16 new_origin_y)

For displays with virtual display area: Set the origin of the physical LCD display (the left lower corner of the display) to the coordinates `new_origin_x`, `new_origin_y` of the virtual display area.

void c_lcd_light(uint8 brightness)

Switches the brightness of the background light of the display to the value `brightness`.

void c_lcd_contrast(uint8 new_contrast)

Set the contrast value of the LCD display to the value `new_contrast`.

void c_lcd_bitblt(void)

If the graphics data are not directly transferred into the LCD controller, but the graphics information is stored into a RAM buffer first, it is necessary to copy the data into the LCD controller. The function `c_lcd_bitblt` copies the RAM buffer into the display memory of the LCD controller. It is only needed if such a buffered output is wanted.

boolean c_lcd_controller_rotate_0(void)

Returns `BOOL_TRUE` if the LCD controller supports the rotation by 0° of the display content. This function is used to rotate the back from any orientation to the normal position.

boolean c_lcd_controller_rotate_90(void)

Returns `BOOL_TRUE` if the LCD controller supports the rotation by 90° of the display content.

boolean c_lcd_controller_rotate_180(void)

Returns `BOOL_TRUE` if the LCD controller supports the rotation by 180° of the display content.

boolean c_lcd_controller_rotate_270(void)

Returns `BOOL_TRUE` if the LCD controller supports the rotation by 270° of the display content.

Function call at 1 bit color depth:
void c_lcd_clear(void)
Function call at 8/16/24 bit color depth:
void c_lcd_clear(const t_color_intrep background_color)

Erases the LCD display. In the case of the color drivers the the color `background_color` is committed.

Function call at 1 bit color depth:
void c_set_pixel_0(uint16 x, uint16 y)

Function call at 8/16/24 bit color depth:
void c_set_pixel_0(uint16 x, uint16 y, const t_color_intrep current_color)

Sets a pixel at coordinates `x`, `y` with color `current_color`.

Function call at 1 bit color depth:
void c_clear_pixel_0(uint16 x, uint16 y)

Deletes a pixel at coordinates `x`, `y`.

Function call at 1 bit color depth:
void c_invert_pixel_0(uint16 x, uint16 y)
Function call at 8/16/24 bit color depth:
void c_invert_pixel_0(uint16 x, uint16 y, const t_color_intrep current_color)

Inverts a pixel at coordinates `x`, `y`.

void c_get_pixel_0(uint16 x, uint16 y, color *pixel_color)

Reads the color of a pixel at coordinates `x`, `y` into the variable `pixel_color`.

Function call at 1 bit color depth:
void c_lcd_hline_0(const uint16 x1, const uint16 x2, const uint16 y, const uint16 line_style, const uint8 draw_mode)
Function call at 8/16/24 bit color depth:
void c_lcd_hline_0(const uint16 x1, const uint16 x2, const uint16 y, const uint16 line_style, const t_color_intrep current_color, const t_color_intrep background_color, const uint8 draw_mode)

Draws a line from `x1` to `x2` in the height `y` using the appropriate `line_style` and `draw_mode`. In the case of the color drivers, the colors `current_color` and `background_color` are used to realize the line style.

Function call at 8/16/24 bit color depth:
void c_lcd_solid_hline_0(const uint16 x1, const uint16 x2, const uint16 y, const t_color_intrep current_color)

Draws a line from `x1` to `x2` in the height `y` with the color `current_color`. This function is only needed for color drivers.

Function call at 1 bit color depth:
void c_lcd_filled_rectangle_0(const uint16 x, const uint16 y, const uint16 delta_x, const uint16 delta_y, const t_fillstyle *fill_style, const uint8 draw_mode)
Function call at 8/16/24 bit color depth:
void c_lcd_filled_rectangle_0(const uint16 x, const uint16 y, const uint16 delta_x, const uint16 delta_y, const t_fillstyle *fill_style, const t_color_intrep current_color, const t_color_intrep background_color, const uint8 draw_mode)

Fills a rectangular area (left lower corner at the coordinates x, y, size in delta_x, delta_y) with the fill style fill_style in draw mode draw_mode. fill_style here is a pointer to an array with sixteen 16–bit words where a bit set represents a pixel set on the display. The first 16–bit word is used for the lowest line of the rectangular area to be filled (and is repeated accordingly if delta_x is large enough). The next 16–bit word determines the look of the next line etc. In the case of the color drivers, the colors current_color and background_color are used to realize the fill style.

Function call at 8/16/24 bit color depth:
void c_lcd_colored_rectangle_0(const uint16 x, const uint16 y, const uint16 delta_x, const uint16 delta_y, const t_color_intrep current_color)

Fills a rectangular area (left lower corner at the coordinates x, y, size in delta_x, delta_y) with the color current_color (independent of the draw mode). This function is only needed for color drivers.

Function call at 8/16/24 bit color depth:
void c_lcd_color_change_area_0(const uint16 x, const uint16 y, const uint16 dx, const uint16 dy, const t_color_intrep src_color, const t_color_intrep dest_color)

This function changes the color of all pixels in a rectangular area from src_color to dest_color.

Function call at 1 bit color depth:
void c_lcd_patternline_0(const int16 x, const int16 endx, const int16 y, const uint8 *line_data, const uint8 draw_mode, const uint8 flag)
Function call at 8/16/24 bit color depth:
void c_lcd_patternline_0(const int16 x, const int16 endx, const int16 y, const uint8 *line_data, const t_color_intrep current_color, const t_color_intrep background_color, const uint8 draw_mode, const uint8 flag)

Draws a horizontal line form x to endx in the height y with the pattern which is pointed to by the pointer line_data. Starting there consecutive memory locations are read until sufficient bits (1 means: Pixel set) are read for the line (starting with the most significant bit of the first address for the left pixel of the horizontal line).

The drawing mode is draw_mode; the 8–bit integer flag determines if the line pattern is to be inverted. This happens if flag = 0xff.

Function call at 1 bit color depth:
void c_lcd_vertical_patternline_0(const int16 y, const int16 endy, const int16 x, const uint8 *line_data, const uint8 draw_mode, const uint8 flag)
Function call at 8/16/24 bit color depth:

void c_lcd_vertical_patternline_0(const int16 y, const int16 endy, const int16 x, const uint8 *line_data, const t_color_intrep current_color, const t_color_intrep background_color, const uint8 draw_mode, const uint8 flag)

This function corresponds to `c_lcd_patternline_0` and is used if vertical texts are drawn.

Function call at 8/16/24 bit color depth:
void c_lcd_antialias_patternline_0(const int16 x, const int16 endx, const int16 y, const uint8 *line_1st, const uint8 *line_2nd, const uint8 draw_mode)

This function is necessary for antialiased text rendering. It works similar to `c_lcd_patternline_0`, however `c_lcd_antialias_patternline_0` combines two pixels from two superposed lines `line_1st` and `line_2st` to form an average color for every antialiased pixel.

Function call at 8/16/24 bit color depth:
void c_lcd_vertical_antialias_patternline_0(const int16 y, const int16 endy, const int16 x, const uint8 *line_1st, const uint8 *line_2nd, const uint8 draw_mode)

This function corresponds to `c_lcd_antialias_patternline_0` and is used if vertical antialiased texts are drawn.

Function call at 8/16/24 bit color depth:
void c_lcd_output_line_0(const int16 x, const int16 endx, const int16 y, const t_color_intrep *line_data, const uint8 draw_mode)

Draws a horizontal line from `x` to `endx` in the height `y` with the color data residing in the memory area to which the pointer line_data points. The `draw_mode` given is used. This function is realized only for color drivers.

Function call at 8 bit color depth:
void c_lcd_set_palette_entry(const uint8 entry, const uint8 red, const uint8 green, const uint8 blue)

Set a color entry into the color palette of the LCD controller used. For the index `entry` three consecutive bytes are written (one each for red, green, blue). This function is realized only for the 8 bit color driver.

Function call at 8 bit color depth:
void c_lcd_output_bmp_line_0(const int16 x, const int16 endx, const int16 y, const uint8
***line_data, const uint8 start_index, const t_color transparent, const boolean alpha)**
Function call at 16/24 bit color depth:
void c_lcd_output_bmp_line_0(const int16 x, const int16 endx, const int16 y, const

t_color_intrep *line_data, const t_color_intrep transparent, const boolean alpha);

Draws a line from x to endx at the height y. The colors of the pixel are assumed to be in a memory area to which the pointed to by the pointer line_data. The draw_mode given is used. The number of the system colors is added to the index of the bmp colors. If the flag alpha = BOOL_TRUE, the pixels with the color transparent are not drawn. This function is realized only for color drivers.

Function call at 16/24 bit color depth:
void c_lcd_output_8toRGB_bmp_line_0(const int16 x, const int16 endx, const int16 y, const uint8 *line_data, const t_color_intrep transparent, const boolean alpha)

Draws a line from x to endx at the height y. The color indices (of a predefined color map) of the pixel are assumed to be in the data area line_data. If the flag alpha = BOOL_TRUE, the pixels with the color transparent are not drawn. This function is realized only for 16/24 bit color drivers.

void c_lcd_copy_area_0(const uint16 orig_x, const uint16 orig_y, uint16 dx, uint16 dy, int16 distance_x, int16 distance_y)

Copies a rectangular area which extends from x = orig_x, y = orig_y dx in x-direction and dy in y-dirction over a distance of distance_x horizontally and distance_y vertically. Currently only implemented for color displays.

void c_lcd_scan_line_0(const uint16 x, const uint16 endx, const uint16 y, t_color_intrep *line_data)

Reads a horizontal line from x to endx in the height y and writes the pixel to memory beginning at the location pointed to by the pointer line_data. The pixels are stored in a way, so that they can be written directly by c_lcd_patternline.

Function call at 16/24 bit color depth:
void c_lcd_output_line_180(const int16 x, const int16 endx, const int16 y, const t_color_intrep *line_data, const uint8 draw_mode)

Corresponds to the function c_lcd_output_line_0 for a display rotation of 180°.

Function call at 16/24 bit color depth:
void c_lcd_output_bmp_line_180(const int16 x, const int16 endx, const int16 y, const
t_color_intrep *line_data, const t_color_intrep transparent, const boolean alpha)

Corresponds to the function c_lcd_output_bmp_line_0 for a display rotation of 180°.

Function call at 16/24 bit color depth:
void c_lcd_scan_line_180(const uint16 x, const uint16 endx, const uint16 y, t_color_intrep *line_data)

Corresponds to the function c_lcd_scan_line_0 for a display rotation of 180°.

Functions for switching contrast and display voltage and for further display initialization: The following two functions are not a part of the driver but are called by the driver functions c_disp_on and c_disp_off respectively to switch on and off the bias voltage. If necessary they should be implemented according to the properties of your hardware (details can be found in the documentation of the LCD drivers).

void c_lcd_bias_on_wait(void)

This function should perform the following tasks:

1. Wait the time specified for the display used between the activation of the display signals (which then already has been done by the driver) and switching on the voltage suppliex.
2. Activation of bias and display voltage if applicable (e.g., TFT display modules typically do not need an external bias voltage).

void c_lcd_bias_off_wait(void)

This function should perform the following tasks:

1. Switch off the bias voltage or display voltage as applicable.
2. Wait for the period of time specified for the display used between switching off and discharging the voltage supply until one is allowed to remove the display signals.

void c_lcd_init_lcdsignals_present(void)

This functions needs to be implemented if additional initialization steps are required *after* the signals from the LCD controller are present. To have this function called after the LCD controller signals got switched on, please define __LCD_INIT_LCDSIGNALS_PRESENT.

5.4.3 Driver for touch glass

When using a touch glass for user input two elementary functions need to be provided which are dependend on your hardware. First, this is the function for initializing of your touch hardware. Second, a function for reading the touch glass in arbitrary units (e.g. the values read from an A/D converter). Both functions are declared in "tglass_.h":

void c_tglass_ground_state(void)

This function initializes the touch glass hardware and puts the touch glass in idle state before a read sequence is performed (this means e.g. for 4–wire touch glasses an appropriate adjustment of the idle state potentials at the touch foils).

void c_tglass_get(uint16 *ptr)

Reading the coordinate pairs of the touch glass. These do not need to be calibrated values but *raw* voltage data e.g. from an A/D converter will do (conversion into calibrated pixel coordinates is performed later automatically and only if needed by the library functions)

The x- and y–values read then must be stored in an array of two 16 bit words to which the pointer ptr points. Here first the y value is to be stored then behind it the x–value !!

To make sure that the calculation of the calibration coefficients can be correctly performed using integer arithmetics another requirement must be fullfilled by the values delivered by this function: They must have a resolution which (numerically) is at least double as large as the resolution of the display underlying the touch glass. If your hardware delivers values smaller than that you need to just *scale* them appropriately.

If the touch glass is not pressed the value `_AD_TOUCH_NOT_PRESSED` is stored to `*ptr`. If invalid touch coordinates were received, the value `_AD_TOUCH_NOT_VALID` is stored to `*ptr`. Both constants are defined in "config_hardware_gui.h" as a 16 bit word which is not normally delivered by the hardware.

Because this function typically is executed periodically by the system timer interrupt one should make sure that the execution time is small (enough).

Remark: When using touch displays with conducting touch foils the RC time constant needs to be taken into account. In order to obtain reliable coordinate values the voltage at the foils needs to be settled sufficiently. The function `tglass_read()` (in "tglass.c"), which calls

c_tglass_get(), makes sure that only contacts which already are established since one interval of the system timer are considered valid. Therefore the time of one interval of the system timer is available to allow settling of the potential of the foils to their stable end potential.

5.4.4 Driver for the loudspeaker

For interfacing the loudspeaker only one function (defined in "piezo_.h") is needed:

void c_sound(uint16 frequency, uint16 duration, uint8 volume)
Output of a tone of the specified frequency, duration and volume.

The output of sound normally is not a typical part of a graphical user interface. Nevertheless the frame is provided here in order to be able to have a *keyboard click* for touch keyboards and buttons to provide feedback for the user. (see the respective functions for touch keyboards and buttons).
In the library it is assumed that the hardware (the *device*) for the output of sound is a piezo loudspeaker (this means the names of structures and files contain "piezo"), but certainly all other sound hardware could be used here as well.

5.5 Configuration of the GUI library

In this context *configuration* means the specification of the hardware of the user interface and the selection of the modules and functions needed. The according adjustments are made within the files "config_gui.h" and "config_hardware_gui.h". The configuration process does not involve modifying the source code of the functions of the GUI library.

In "config_hardware_gui.h" variables are defined which describe the hardware environment for the user interface of the system (e.g. the physical resolution of the display). Variables of components which you intend not to use (e.g. a touch glass) can be left unchanged.

In order to avoid including unnecessary code the configuration file "config_gui.h" is provided. Here the modules needed and the properties of the environment are adjusted by using #define.
The file "config_gui.h" contains two sections: In section 1 adjustments for the software environment are made. Here you can choose which components of the library shall be used in your target system. By means of conditional compilation within single functions it is avoided to include unnecessary code.

The consistency of the configuration in “config_gui.h” is ensured by rules defined in the file “config_gui_error_check.h”.

Please note that components not chosen can therefore not be used for programming the application. Calls to such functions thus lead to compiler errors which does not constitute a defect of the library. After changing the configuration it is often necessary to recompile all source files which include “config_gui.h” .

The following section 2 checks what you have chosen. If incompatible selections were made you are notified by an error message which is issued during compilation (e.g. it does not make sense to have touch keyboards without using a touch glass with the touch functions).

These are the adjustments to be made in “config_hardware_gui.h” and “config_gui.h”:

5.5.1 Adjustments for the hardware in “config_hardware_gui.h”

Adjustments for the system an the display: At the beginning of the file the properties of the system and the display are set:

_SYS_LITTLE_ENDIAN must set at systems with Intel–Byte–Order,

_LCD_PRESENT = 1 when a LCD display is present (0 otherwise),

_LCD_LIGHT = 2 if a LCD backlight is present (0 otherwise),

_LCD_LIGHT_ADJUST = 4 if the backlight brightness can be adjusted,

_LCD_CONTRAST_ADJUST = 8 if the LCD display contrast can be adjusted (0 otherwise),

_LCD_INVERTED = 16 if the graphics information on the LCD display shall be inverted completely, e.g. if the LCD does itself invert (0 otherwise).

_LCD_PHYS_SIZE_X = Number of pixel horizontally,

_LCD_PHYS_SIZE_Y = Number of pixel vertically.

If the range and default values as well are to be considered for both, contrast and brightness of the backlight of the LCD display (thus if such hardware features are available), the following definitions can be made:

_CONTRAST_DEFAULT_DATA = default contrast value (1–255),

_CONTRAST_LOWER_LIMIT = lower boundary for contrast value (1–255),

_CONTRAST_UPPER_LIMIT = upper boundary for contrast value (1–255),

_BRIGHTNESS_DEFAULT_DATA = default brightness value for backlight (1–255),

_BRIGHTNESS_LOWER_LIMIT = lower boundary for brightness (1–255),

_BRIGHTNESS_UPPER_LIMIT = upper boundary for brightness (1–255).

_AUTOMATIC_CONTRAST_TEMP_COMP = compile switch, if the hardware performs a temperature compensation of the contrast voltage.

_AUTOMATIC_BRIGHTNESS_TEMP_COMP = compile switch, if the hardware performs a temperature compensation of the LCD brightness.

Adjustments for the touch glass: For the touch glass the following is needed:

_TGLASS_PRESENT = 1 if the touch glass is present (0 otherwise),

_AD_TOUCH_NOT_VALID = value to be returned from the touch driver function to represent an invalid touch contact (a value must be chosen which can not be delivered by the hardware driver, e.g. 8192).

_AD_TOUCH_NOT_PRESSED = value to be returned from the touch driver function to represent the *not pressed* state (a value must be chosen which can not be delivered by the hardware driver, e.g. 4096).

_TT_DIV und _TT_0_5: These constants are used to calculate the contact position on the touch and should be set to _TT_DIV $= 16384$ and _TT_0_5 $=$ _TT_DIV$/2$.

_TOUCH_NORMAL and _TOUCH_ROTATED: Orientation of the touch glass relative to the LCD display: If the orientation of the hardware connection of the touch glass aligns with the coordinate axis of the LCD display during operation in `LCD_NORMAL` orientation (without rotation, which can be used only for certain drivers or LCD controllers) please set `_TOUCH_NORMAL` (then both the x- and y–axis of touch glass and LCD display point into the same direction). This is the correct choice for most cases.

If the connection of the touch glass is realized in a way that it's coordinate system is perpendicular to the coordinate system of the LCD display, please set `_TOUCH_ROTATED`.

Of course only one of these definitions may be set.

_TGLASS_A_DEFAULT_DATA to _TGLASS_D_DEFAULT_DATA: The calibration constants of the touch glass `_TGLASS_A_DEFAULT_DATA` to `_TGLASS_D_DEFAULT_DATA` depend on the hardware and must be determined experimentally (e.g. by using the calibration function `tglass_calibration_sequence`, also see the respective example in the "Quickstart Tutorial").

These are the constants for calculating the coordinates from the (voltage-) values, which are delivered by the hardware driver. A linear relation of the form $x = A * U_x + B$ between these values (e.g. voltages of an A/D converter U_x, U_y) and the coordinates is assumed.

Prerequisite for the calibration with the function `tglass_calibration_sequence` is that the LCD display is not rotated by software but run in the orientation `LCD_NORMAL)`. The calibration coefficients are used even if the display is rotated later; the coordinates are then automatically transformed accordingly. This avoids additional work for the application programmer and also a single set of calibration coefficients is sufficient.

KALIB_X1 to KALIB_Y4: The calibration function `tglass_calibration_sequence` uses the coordinates for the crosses need to be defined `KALIB_X1` to `KALIB_Y4`. Positions within the four edges of the LCD display are appropriate.

The calibration function `tglass_calibration_sequence` shows these four crosses on the LCD display and requests the user to touch those in order to calculate the calibration coefficients.

Moreover the calibration function shows an explaining text on the display. The position of the text is predefined for a 320×240 pixel display and needs to be adjusted for smaller displays in the source code of the library (files “tglass.c” and “text_gui.h”).

Adjustments for a loudspeaker: In order to interface a loudspeaker (e.g. for alert sound or *keyboard clicks*) the following definitions are needed:

_PIEZO_PRESENT = 1 if loudspeaker is present (0 otherwise).

_PIEZO_VOLUME_ADJUST = 2, if volume can be adjusted (0 otherwise).

If you want to account for a range and default values for the volume of the loudspeaker (provided that the hardware has got this feature), please define:

_VOLUME_DEFAULT_DATA = default value for volume (0–255).

_VOLUME_LOWER_LIMIT = lower boundary for volume (0–255).

_VOLUME_UPPER_LIMIT = upper boundary for volume (0–255).

It is certainly not necessary to use a piezo loudspeaker for realizing sound output. Only the hardware driver functions (described in chapter 5.4.4 on page 48) are *in contact* with the real loudspeaker and need to be designed accordingly.

5.5.2 Adjustments for the software environment in "config_gui.h"

First, the system environment to be used by the library must be defined. This includes the system timer and dynamic memory allocation as well as some simple functions. These functions are easily provided as described in detail in chapter 5.4 on page 36.

The definitions for the system timer are:

_SYSTEM_TIMER_PRESENT: Defines if a system timer is available. This is a function which fullfills the following requirements (see section 5.4 on page 36 too):

- The functions to be called periodically for the touch glass and blinking sprite cursors are called from the system timer by calling the function `system_process_timed_functions()`,
- The time is to be stored in two global 32–bit variables `TIMERDATA` (Number of system ticks after starting) and `DAYDATA` (Number of the full 24–hour periods since the start).
- This system timer in connection with touch glasses should be triggered in the order of 100 times per second.

_DISABLE_TIMER_INTERRUPT_NEEDED : Defined if an address pointer assignment in the functions `system_timed_function_on()` and `system_timed_function_off()` in the file system.c cannot be executed in one *atomic* machine instruction. Then the timer interrupt is disabled before assigning a value and activated afterwards to make sure the list needed by the timer is in a consistent state always. This is only needed if `_SYSTEM_TIMER_PRESENT` is defined. If in doubt this definition should be activated.

_TICKS_PER_SECOND: Number of how often the system timer is triggered per second. If e.g. each 1/100 s the system timer is triggered this value is equal to 100. This definition automatically leads to the definition of `SEC`, which allows the use of `SEC` in the functions `system_wait_ticks()` and `system_wait_until()`.

Example: `system_wait_ticks( 10*SEC);`

_SOFTWARE_DELAY_NUMBER: If no system timer is needed the function `system_wait_ticks()` works with a delay loop. How often this loop must be completed to wait a period like the one between two system timer calls should be defined here. Example: For a Hitachi H8/300H with 14.7456 MHZ the correct value is somewhere at 370000.

The possibility for dynamic memory allocation is part of ANSI–C (`malloc()`). However for small systems it can be advantageous to avoid this functions if one needs to save the overhead (e.g. in the case of very limited RAM). In this case the the included memory allocator ("simple_memory_wrapper.h/.c") can be used, with the cost of flexiblility and performance in comparison to the C standard version.

Definitions for the memory management:

_MALLOC_AVAILABLE: Please define `_MALLOC_AVAILABLE` if dynamic memory allocation of the C standard library ("stdlib.h")is made available.

_STORAGE_ALLOCATOR_HEAP_SIZE If you do not want to use the memory allocator of the "stdlib.h", please define `_STORAGE_ALLOCATOR_HEAP_SIZE` as the number of bytes which are needed for the heap memory of the allocator. The included memory allocator makes use of a list to handle the free memory areas in the heap.

_MAX_FREE_LIST_ENTRIES: Define `_MAX_FREE_LIST_ENTRIES` as the numbers of entries in the previously mentioned list.

More display definitions and for optimizing:

_BITS_PER_PIXEL: Colo depth 1 bit, 8 bit, 16 bit or 24 bit, depending on the used driver (and the display hardware).

_MAX_BYTES_PER_LINE Define `_MAX_BYTES_PER_LINE` as the number of bytes which are needed for a single line of your virtual display. For a monochrome display this is:

$\texttt{INT}\left(\frac{1}{8}\,\text{(number of pixels of the virtual displays in x direction)} + 7\right)$

_STATIC_RENDERING_BUFFER_WANTED: Use this define, if your don't want to place the memory buffer for rendering operations into the heap. If this option is used no virtual displays are allowed.

_OS_USED To use waiting times for other purposes which would otherwise be *spoiled* in the function system_wait_ticks in a delay loop, the definition `_OS_USED` can be used. Then during the function `system_wait_ticks()` a function `os_process()` which you must then provide is called. Then e.g. an operating system could in between perform further tasks (thus the name `os_process()`).

_MAKE_CHECKS The code of the library contains checking of correct connections between the channels (e.g. the display channel) and the connected devices (e.g. LCD device). When a system is stable and tested with a certain application errors should not occur here anymore and this check can be omitted in order to save space and execution time.

If you want to activate this check, please define: `_MAKE_CHECKS`.

_ALLOW_OVERHEAD For future extensions there are some dummy functions and variables within the GUI library already which are not yet used. This code can easily omitted to reduce memory usage. Define `_ALLOW_OVERHEAD`, if you nevertheless want to include this code.

For the administration of different default values (e.g. display contrast) these data can be included within the appropriate structures (this is usually not necessary for small systems). If you want that, please define:

_CONTRAST_DEFAULT_HANDLING_WANTED for contrast default values for the LCD,

_BACKLIGHT_DEFAULT_HANDLING_WANTED for brightness default values for the LCD backlight,

_VOLUME_DEFAULT_HANDLING_WANTED for default values for the loudspeaker volume,

_TOUCHPARAMETER_DEFAULT_HANDLING_WANTED for default values of the touch glass calibration coefficients.

Configuration of the functionality needed: The following defines allow to adjust the amount of functionality of the GUI library. Please define:

_USE_LCD_DEVICE, if you want to use the LCD device (default).

_USE_PAGEPRINT_DEVICE, if you want to use the Pageprint device (for printing, future extension),

_CHOOSE_PRINTERDRIVER_RUNTIME, if it should be possible to switch the driver for the pageprint device during runtime,

_PAGEPRINT_COLOR, if the pageprint device shall be configured for color output (future extension),

_GENERAL_LINES_WANTED, if you need arbitrary lines,

_BMP_WANTED, if BMP bitmap pictures are used,

_TRIANGLES_WANTED, if triangles are needed,

_CIRCLES_WANTED, if circles and arcs are needed,

_DYNAMIC_FONTS, if the fonts are to be administered dynamically by the system (in this case dynamic memory allocation is also needed). If `_DYNAMIC_FONTS` is not defined, the functions `system_get_number_of_fonts()` and `system_get_font_pointer()` are not available,

_TEXT_ANTIALIAS if antialising of text is needed,

_UNICODE_WANTED if unicode support of fonts is needed,

_VIRTUAL_DISPLAY_WANTED, if the functionality to scroll the LCD screen across a larger virtual display is needed. If this functionality can be provided also depends on your hardware and the LCD driver, because an appropriate LCD controller and sufficient display memory is needed.

_BUFFERED_DISPLAY, if graphics data is not to be directely written into the LCD controller but first shall be written into a RAM buffer of the microcontroller. The function `disp_update()` is then available for your application to transfer the graphics data into the LCD controller. Please be sure to apply this function whenever necessary to update the graphics for your application. Otherwise the graphics information is only generated in internal memory but not shown on the display.

_DISPLAY_ROTATION_90_WANTED, if the display shall be operated perpendicular to the normal orientation (also is rotated by 90 degrees). This feature is only available with supporting LCD controllers or drivers.

_DISPLAY_ROTATION_180_WANTED, if the display is rotated by 180 degrees. This feature is only available with supporting LCD controllers or drivers.

_DISPLAY_ROTATION_270_WANTED, if the display is rotated by 270 degrees. This feature is only available with supporting LCD controllers or drivers.

_DISP_PRINTF_WANTED, to use the text output function `disp_printf()` which resembles the C function `prinf()`. As this function needs many conversion functions from the C libraries the size of your code may significantly increase if none of these functions are already used by your application.

_TOUCH_WANTED if the functions for the use of a touch glass are used,

_TGLASS_CALIBRATION_WANTED if the function `tglass_calibration_sequence()` is needed for calibration of the touch glass,

_BUTTONS_WANTED for user interfaces with touch buttons,

_TOUCH_KEYBOARDS_WANTED if input is to be made using configurable touch keyboards.

_TOUCH_ADDONS_WANTED if you want to use additional control elements for touch screens (this is used for future extensions).

_TOUCH_SELECTION_LISTS_WANTED, if you want to use selection lists (touch addons need to be activated too),

_ENCODER_SUPPORT_WANTED, if you want to readout an external encoder,

_SOUND_WANTED, if the function for loudspeakers shall be included into the code,

_SOUND_VOLUME_ADJUST_WANTED, if your hardware provides for the adjustment of sound volume.

Please note while performing the configuration: By conditional compilation parts of structures and functions are removed to save system ressources. If the omited code is to be used anyway for other purposes (e.g. the double linked lists from "struktur.c"), the respective compile switches or the code needs to be modified. When using functions which were excluded during configuration compiler errors will be issued. Please check the configuration in this case to make sure that these functions are included into the code. This also holds if modifications or extensions to the library are made (e.g. `__DONT_COMPILE_OVERHEAD` works on all functions which are present only as dummies at the moment). After modifying these dummy functions the switch must there be changed to make the function available.

5.6 Optimization

In order to save memory and execution time for code sequences which are not needed for the target system the file "config_gui.h" provides switches for only compiling the modules needed. This holds for the modules "touch.h / tglass.h" for the functions of the touch glass and for the modules "keyboard.h" and "t_keyb.h" (keyboards on touch displays). Additionally the modules "sound.h" and "pieso.h" respectively are not needed in all cases. They are used for providing an optional feedback (*click*) for buttons and keyboards The definitions do not only omit single functions according to the application but also provide for some optimization within the functions.

Please remember that components not chosen in the configuration are not available for programming the application.

If memory is small it also pays to convert the compiled files of the Simplify Technologies GUI library to libraries in terms of the tool chain which helps the linker to only include functions really used in the actual application

Because the Simplify Technologies GUI library is delivered in source code it is also possible to remove or modify parts of the code not needed in the respective modules. We recommend using an editor with *syntax highlighting* to help you work on the code.

6 Colors and shades of gray (optional)

In this chapter the concepts for using color or gray scale displays are described (different shades of gray are considered *colors* in this context). These properties are not available with the monochrome version of the library.

6.1 Restrictions using a color depth of 8 bit

6.1.1 Color models and use within the library

Within the Simplify Technologies GUI library, color is used by employing the data type `color`. This data type is defined in the file "display_colors.h". Currently `color` is defined as `uint8` because for LCD displays for embedded systems normally 8 bit per pixel (256 colors) suffice. Many LCD controllers work with a palette model for the colors. This means that the colors themselves are defined in an index table by their color components (red, green, blue) and are then referred to by this index of the table. The limited palette leads to restrictions when using color. For a 256 color palette for example real color photos cannot be displayed directly.

In order to obtain an acceptable appearance of a photograph it is necessary to adapt the limited color space to the image. This poses problems if other photos or other elements with different colors need to be displayed at the same time. Especially it would be disturbing or unacceptable if the carefully designed user interface elements change their color due to a change within the color palette. Therefore the approach used by the library is as follows:

- There are fixed *system colors* (siehe "display_colors.h") which are available independently from possibly shown images and which do not change. These are the 16 standard colors of the VGA palette: `BLACK, MAROON, GREEN, OLIVE, NAVY, PURPLE, TEAL, GRAY, SILVER, RED, LIME, YELLOW, BLUE, FUCHSIA, AQUA, WHITE`. If required these can easily be adapted to another standard color set (`disp_set_palette_entry()`). These colors also can be referred to in the functions of the library by their name.

- Depending on the number of the colors which can be shown on the device the colors which are available in excess or the standard colors can be used to display images. E.g. if there the number of colors available is 256, 240 colors can be used to display images. However, it is possible to change the number of colors which are reserved for BMP images with the function `disp_reserve_bmp_colors()`. For instance, there may be 224 colors reserved for BMP pictures and 32 colors will be unchanged if some pictures are displayed.

6.1.2 Handling of color BMP images

At the moment the supported image formats are uncompressed monochrome BMP pictures, 16– and 256–color BMP images (uncompressed or RLE compressed). Monochrome images are always output as black/white images. When BMP images with colors are to be displayed their colors are loaded to the color palette in a way that the 16 standard colors are not affected. In order to obtain a correct display of images please use the following procedure:

- The image must be reduced to the number of colors available. This can be realized with an arbitrary graphics program.
- If more than one images are to be displayed at one time, it is an advantage if they have the same palette of colors. In this way they don't need so much entries in the color table of the LCD controller. This too can be achieved with the use of a conventional graphics program[1]. Suggested steps are:
    1. Copy all images in a new large image.
    2. Reduce the colors of this new image to the allowable maximum number.
    3. Separate the original images from the large image. The single images now use the same color palette and can be displayed on the LCD without artefacts.

If a new picture is loaded, the algorithm tries to find the color palette of the picture in the already used entries of the color table of the LCD controller, The found colors can now be use as well from the new picture. If they don't match, the palette of the new picture uses the not occupied entries in the color table. Because of their limited number, there might be no free entries. In such a case the first colors of the reserved area in the colortable are erased and used for the new picture.

An application example for the use of color can be found on the CD.

Alignment of the BMP–data on longword boundaries

The picture data must be aligned to 32 bit word boundaries to increase the performance of the drawing algorithm. The type for BMP pictures therfore is defined as follows (compare to "display.h"):

[1] In principle it would have been possible to omit the external steps of image processing and to calculate what's necessary within the functions of the library. Because this would have resulted in larger processing time and memory requirements the external processing of images was considered the better option.

```
typedef uint32 *t_bmp_ptr; /*!< pointer to picture data in bmp-format,
                                the bytes of the image must be stored
                                in big endian order */
```

Moreover it is important, that *big endianess* is used for oder of the data bytes. Mostly, it is simple to convert a BMP picture to a byte array. With the following trick the alignement to 32 bit words can be enforced:

```
typedef union { uint8 c[7810]; uint32 align; } t_my_bmp;
const t_my_bmp my_bmp =
{{
0x42,0x4d,0x82,0x1e,0x00,0x00,0x00,0x00,0x00,0x00,0x36,0x01,0x00,0x00,
0x28,0x00,0x00,0x00,0x64,0x00,0x00,0x00,0x4b,0x00,0x00,0x00,0x01,0x00,
0x08,0x00,0x00,0x00,...
}};
```

This array can be now used to display the picture:

```
disp_bmp( (t_bmp_ptr)(&my_bmp));
```

6.2 Color depth of 24 bit *(truecolor)*

The color information of the red, green and blue component is stored in a 32 bit long word of the type `t_color`. It is recommended to use the predefined macros DISP_RGB(r, g, b) from “display_colors.h” to fill a color value. The macros `DISP_R_RGB( rgb)`, `DISP_G_RGB( rgb)` and `DISP_B_RGB( rgb)` are used to extract an elementary color out of a color value.
BMP pictures with a color depth of 24 bit are supported. In most cases it is sufficient to use images with 256 colors, which can be displayed together without restrictions and need less memory.
The textstyle `TXT_LIGHT` is realised with an appropriate color and not by rasterization of the printed text string.

7 Comprehensive programming example

First, a programming example for monochrome displays is given which shows most of the aspects of programming a GUI interface using the GUI library. Second, another example is shown explaining the use of color within the library. Both examples can also be used as a starting point for other applications, the sources can be found on the CD of the library in respective folders as described in the file “Content.txt”.

7.1 Monochrome example

Please note, that for this example all modules of the GUI library are used to get a comprehensive demonstration. When configuring the system all those modules need to be included. Additionally the dynamic allocation of memory and the system timer is used. This requires the initialization by calling `system_init()` directly after entering the application function `main_application()`. By calling `system_init()` the system timer is started and the list of fonts is initialized. The single fonts are then registered to the system:

```
system_init(); /* Initialize system timer and font list */

#ifndef __DONT_COMPILE_DYNAMIC_FONTS
system_font_init( __Mono8x16); /* initialize first font, this */
system_font_init( __Mono6x8);  /* becomes the system default font */
system_font_init( __Prop14);
#endif
```

Chapter 5 on page 34 describes the function `os_systemtimer_start()`, which needs to be available for the correct execution of `system_init()`. This function depends on your hardware. It calls the internal function `system_process_timed_functions()` periodically.
More simple examples, which only demonstrate a part of the possibilities of the library can be found in the “Quickstart Tutorial”, which helps you getting started with the Simplify Technologies GUI library.

7.1.1 Components of the programming example

The example printed here contains the application within the function `main_application()`. Please call this function from within your `main()`

function. You could also rename `main_application()` into `main()` if in your system other tasks do not need to be performed. (For your ANSI–C compiler it is necessary to define a function `main()` in your project because this is where executing the program begins.) The program starts, like most C programs with reading the required header files and the definition of global variables. Especially, these are the channels and devices as well as the buttons for controlling the flow of the program by the user. Next, the implementation of the function `main_application()` follows:

Application function `main_application()`: First, the devices used are initialized and connected with the respective channels to allow a hardware independent programming style. These are: LCD device / display channel, piezo device / sound channel, Tglass device / touch channel. This initialization and connection sequence is typical for all applications an is most comfortably done directly after the start of the program.
Then, further initializations happen which deal with objects which are to be used by this application specificly: First, there is the enquiry of the pointers of the system fonts, so these can be used later on. Then, the buttons with their frames are defined which are used by the selection menu of the example and allow scrolling of the virtual display.

After initializing the menu is activated and the program enters the main loop (which in this case never ends). The main loop consists of checking the selection buttons for the demos. If a certain demo is selected, the menu is deactivated by calling `deactivate_menu()` and then the demo function wanted is called. After returning from the demo function the subroutine `activate_menu()` again activates the selection menu.
In the main loop scrolling of the virtual display is allowed so all buttons and demos which would otherwise not be visible on the LCD display can be selected.

After the main function `main_application()` the subroutines for each demo follow:

The line demo `lines_demo()`: This function demonstrates the possibility of the various line functions. Lines are displayed in various styles and

thickness. Then a small animation with lines is shown. the program text also demonstrates the possibility to either position the graphics cursor absolutely or relatively to the former position.

Demonstration of circles arcs and rectangles: `circles_demo()`**:** In this function the generation of simple geometric objects is demonstrated. Besides circles and arcs (which can be made in multiples of 1/8 circles) rectangles can be displayed with variations in boarder and fill style.

Display of BMP pictures: `pictures_demo()`**:** To demonstrate how pictures can be shown a sequence of monochrome BMP pictures is displayed. In this example the BMP pictures are, after being converted into ASCII header files, included as C header files. It would be possible as well to display BMP files which are already present in the system otherwise with bmp_text.

Demonstration of other graphical functions: `other_demo()`**:** Here the demonstrations of various combinations of the graphical functions with the background can be found as can the demo of the sprite cursor. This allows to have a small pictogram moving across the graphic display without worrying about the background (saving and restoring the background is done automatically). As an example two different types of sprites are shown and also the possibility to make the sprite blink (as might be wanted for a text cursor, which also can be seen in the demo for keyboards).

Output of text: `text_demo()`**:** This function generates various text strings. These can be output with different fonts and various styles.

Buttons for the user: `buttons_demo()`**:** Here different buttons with their frames are defined. After the look of the buttons have been defined as *styles* and the buttons are activated with the function `button_activate_button()` the function enters a loop which evaluates if buttons are pressed with the function `touch_process_objects()`. Depending on the kind of button selected the properties of the buttons like label or position is changed. Pressing the `menu_button` ends both the loop and this demo.

Demonstration of the functions of the touch channel: `touch_demo()`**:** In this function an area of the touch display is assigned to be a drawing area (e. g. for capture of signatures). After activating the buttons used for this demo it is checked if a previous contact with the touch channel happened. Then a do loop is entered which both realizes the use the buttons and drawing on the drawing area as well.
To achieve this first, it is tested if a present contact happened within the boundary of the drawing area. If this is the case a line is drawn from the previous position to the current position of the contact. I the contact happened outside the drawing area the buttons are processed (with `touch_process_objects()`). If one of the buttons has been activated, the corresponding action is performed (erasing the drawing area or leaving the demo).

Text input with touch keyboards: `keyboard_demo()`**:** This demonstration shows how to use keyboard which can be configured in a very flexible way in terms of functionality and look. First a touch keyboard device and a keyboard channel is defined. After defining a style which determines the appearance of the touch keybard the touch keyboard is displayed with the function `t_keyb_make()`. Then, being considered a *device* it is connected with the keyboard channel.
Now a button for returning from the demo is activated.
The following do loop performs the whole job (until it is left by pressing the `menu_button`: First it is tested, if a contact with the touch display happened within the area of the keyboard, which in this case would be processed with `t_keyb_process()`. The result of pressing a key on the keyboard then is tested if one of the cursor keys has been pressed (the numbers in the case statement are the numbers of the keys in the definition of the keyboard in the header file (which was included into "app.c"). By using the auxiliary functions `insert_char()` and `remove_char()` (defined below) the text string is composed according to the users input in the variable `string`.
Changes of the string, especially those by deleting or adding characters are then output to the display.

Please note that there are further keyboard examples for you to experiment with included in the library (those are header files and contain more alphanumeric or number block keyboards).

The file ends with simple auxiliary functions like the display of the info window (function `info()`), the function for activation and deactivation of the main menu (`activate_menu()` and `deactivate_menu()`) and the wait function which is called at the end of some demos, `wait_for_menu_button()`.

7.2 Color programming example

This example is an application comparable to the monochrome one but using color from the color version of the library. The hints given there basically apply here, too.

8 Software safety

8.1 Facts

Software is complex and it is only possible to test a limited number of the possible states of a software. Even programs ready to run which run with varying parameters and environmental conditions exhibit a very large number of possible run sequences. Therefore one cannot assume that a software is free of defects.
When using the GUI library there are additional facts which increase complexity thus also increasing the risk for defects to occur.

1. Configuring the library (and then compiling it conditionally) generates a large variety of possible variants of the software.

2. The library code is used on very different hardware environments.

3. The library is not run solely on a system but as a component together with an *arbitrary* application to interact with.

4. The generation of executable code can be performed with a large number of different development tool chains (which themselves are complex software systems).

Software is also only a component in a complete system. A reliable hardware is required for a safe operation.
Simplify Technologies strives to deliver high quality software. Nevertheless one needs to be aware of the possibility of the occurrence of faults within the library. If a fault occurs within the library, please let us know for fixing the bug.

8.2 Precautions

Programming reliable software is not a subject which can be dealt with adequately here in a short way. Please obtain information about this subject from the large selection of literature [2], the documentation of your development

[2] e. g. Hoang Pham: Software Reliability, Springer 2000, ISBN 981-3083-84-0, or John D. Musa: Software Reliability, McGraw-Hill 1999, ISBN 0-07-913271-5

tool chain, special literature for your application and regulations eventually applying to your application.
A defensive style of programming can considerably increase the reliability of your application. This involves (among other issues) checking the parameters of functions not to exceed their allowed range and testing the return value of functions in order to detect errors which may have occurred.
The completed application must be tested carefully. Additionally it may be advisable to use a ”watchdog device” which is independent from the software and checks the integrity of the system and the application and, if the verification fails, puts the system in a secure state.

9 Function reference of the library functions

An overview and a brief description of the GUI library functions can be found in section 3 of this document.

A detailled function reference of the GUI library is provided in a separate function reference manual which is provided on the CD of the library. For each of the more than 300 functions the function reference provides the function prototype and an explanation as well as the description of the function parameters and an example of a typical function call. This is done as in the following example:

disp_line:

Draw line from and relatively to the current cursor position. The current line attributes and width are used. The endpoint of the line then becomes the new current cursor position. Assumption: Current cursor position is at x0, y0: Draws a line from ther to the position x0 + delta_x-1, y0 + delta_y-1 (the length of the line in x and y direction respectively is thus delta_x and delta_y).

Function prototype:

```
err_code disp_line(
    const int16 delta_x,
    const int16 delta_y);
```

Parameters:

delta_x, delta_y: relative distance of the endpoint of the line to the current cursor position. Negative values are allowed.

Return values:

ERR_OK: success.

ERR_CNA: there is no current display or it is not connected.

ERR_PAR: the end points would be at negative coordinates or line length is 0.

ERR_xxx: error codes of the device connected to the display.

Example for function call:

```
disp_line(
    delta_x,
    delta_y);
```

10 Frequently asked questions (FAQ) and troubleshooting

Here you find answers to questions frequently asked and help for problems frequently encountered.

1. **I get error messages from the compiler like ”file xy not found” or from the linker, which does not find certain symbols:**

   The compiler could not find a file needed. Please make sure to let your development tool chain know where all the files needed are. A simple solution is to copy all files needed in a single directory. Please do not forget the files for your LCD driver.

   Another possible cause can be, that the configuration in config_gui.h has been made in a way that certain parts of the code needed is not compiled. Please make sure that all the compiler switches are set correctly for your application in config_gui.h.

2. **Why do I get compiler warnings of the type ”Variable x declared but not used” ?**

   The library contains generic structures so that the programming interface does not need to be changed in case of further extensions .

3. **Everything seems to be ok but a certain graphical element or text respectively is not displayed:**

   Please make sure if size and position allow the element to be displayed on the virtual display completely.

4. **A text should fit on the display; however it is not shown anyway:**

   You must assign a font for the current display to be the current font (using the function disp_set_font).

5. **Is it possible to in addition get the feature xy for the library ?**

   Answer: Probably yes. You can make modifications yourself within the source code. If a property of general interest is concerned, we might do this extension. too.

6. **How can I get more / other fonts ?**

   The GUI library uses fonts in *.FNT format. Using an appropriate font editor allows you to modify the fonts included with the library or to generate your own fonts. The binary *.FNT file can then be converted into a header file with the utility bin2h which comes with the library. This header file can then be used the same way like the header file of the standard fonts.

If there are important questions yet to be answered, feel free to ask Simplify Technologies directly.

A Fonts

For the fonts included the european character set according to ISO-8859-1 was chosen. For some purposes it is advantageous to have additional arrows, e. g. for keyboards. Such arrows were additionally stored under the hex codes 0x1C - 0x1F.

The character set is available in three sizes:

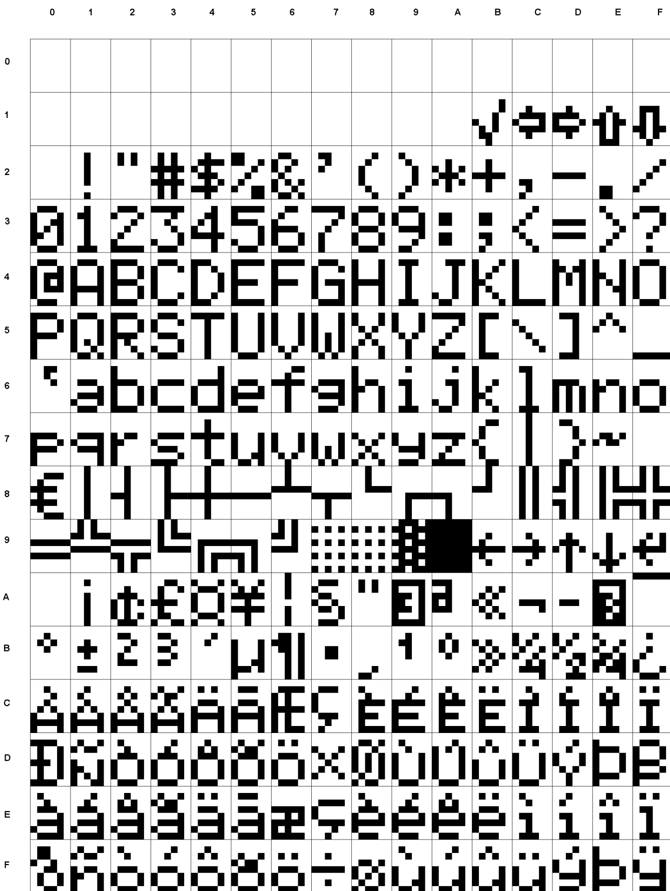


Figure 12: Font Mono6x8

Figure 13: Font Mono8x16

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1												√	⇦	⇨	⇧	⇩
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8	€	│	┤	├	┼	─	┴	┬	└	┌	┐	┘	║	╣	╠	╬
9	═	╩	╦	╚	╔	╗	╝	░	▒	▓	█	←	→	↑	↓	↵
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure 14: Proportionalfont Prop14

B License

General Software License

Simplify Technologies GmbH, Steinbuehlstr. 15, D-35578 Wetzlar, Germany (hereinafter referred to as the "licensor") grants the licensee the right to use the respective software (hereinafter referred to as the "software") in accordance with the terms and conditions set out below. By using, acquiring or embedding the software into a system, the licensee agrees to the terms and the conditions of this license.

Definitions: The software can be:

"Executable programs" denotes software, which can be directly executed on a respective target hardware, possibly using a run-time environment, or which is used directly as a part of an executable program (run-time librariy). This includes the software which is directly implemented in the systems of the licensor ("embedded software").

"Source code" denotes software in the form where it can be read and processed by a human. Typically source code cannot be executed directly but needs appropriate compilation or interpretation to become executable software (which means executable programs or parts of executable programs.

"Test version" denotes software, which is provided to licensee for testing purposes.

"Example software" denotes software, which is provided to licensee for example and instruction purposes to help licensee to start development with respect to the products of the licensor.

"Purchased software" denotes software, for which the license is obtained seperately from other products and services of the licensor in return for the payment of an agreed price, or which is provided together and as a part of a product sold by the licensor.

1. Intellectual property rights and scope of license
a) The licensor is the owner of all intellectual property rights and all other rights over the software as set out in sections 69 a ss. of the German Copyright Law (UrhG) and international treaty provisions, and the sole holder of the right of disposition of the software. The protection thereby granted to the licensor is hereby expressly accepted by the licensee.
b) For software provided without cost the following provisions apply:
The software is provided "as-is". There is no guarantee of any properties of the software.
c) For purchased software the following provisions apply:
The licensee shall be given all documentation and items required for realization of this contract and adequate documentation of the software.
The software and documentation get deliverd on a data carrier or via remote data transmission, printed documentation may or may not be provided.
d) The licensor retains all rights not expressly granted to the licensee, in particular all rights of ownership of all intellectual property rights in the software, the know-how and the user documentation. The licensee may not transfer the software to any third party unless expressly permitted under the terms of this license.

2. Rights granted
Licensee is hereby granted the following rights:
a) Licensee is granted a non-exclusive, right to use the software.
This right is non-time-limited, except for the case of "test versions" where it is time-limited to 30 day commencing from the point of time of delivery of the software licensed.
b) The software may not be used for the following security-related applications: applications in aeronautics, space, military and in nuclear technology, applications dealing with ionizing radiation, lasers or maser radiation, applications to influence the movement of vehicles, applications in traffic security systems (e.g. airbags, break control systems), applications in life-support systems, especially those in medical applications, applications in which dangerous substances would or might be released into the environment in the event of failure.

c) For purchased source code the following provisions apply:
c1) Licensee may use the software on a single computer or within a network comprising up to 5 users at the location of his organization.
c2) The software or parts of the software may be translated by licensee using any development tool into an executable form and included in licensee's products in this executable form as long as the software thus employed is a fixed component of those products and is sold together with the products as long as the products do not constitute a development tool for display applications of third parties and the software does not become a component of a development tool.
c3) Licensee is entitled to modify the source code of the software. The licensor shall retain all rights over any such modified source code. Modified source code shall be considered as software covered by this license.
Licensee is obliged to inform licensor of any intention to modify or improve the software. A sample copy of all modifications made is to be sent to the licensor free of charge before product completion.
c4) Transfer to any third party of the software or any part of the software in source code or in a linkable object format is hereby expressly prohibited.
d) Für example software the following provisions apply:
d1) Licensee may use the software on a single computer or within a network comprising up to 5 users at the location of his organization, exclusively in conjunction with the products of the licensor.
d2) If the example software consists of source code, the example software or parts of it may be translated by licensee using any development tool into an executable form and included in licensee's products in this executable form as long as the software thus employed is a fixed component of those products and is sold together with the products as long as the products do not constitute a development tool for display applications of third parties and the software does not become a component of a development tool and provided that the example software is used in conjunction with licensors products. If the software is a test version, there is no right to sell the software even in conjuction with the licensors products.
d3) If the example software consists of source code, Licensee is entitled to modify the source code of the software. The licensor shall retain all rights over any such modified source code. Modified source code shall be considered as software covered by this license. Licensee is obliged to inform licensor of any intention to modify or improve the software. A sample copy of all modifications made is to be sent to the licensor free of charge before product completion.
e) For executable programs the following provisions apply:
e1) If the software is not example software the following applies: Licensee may install use the software on a single computer at the location of his organization. Additionally, licensee may hold up to two backup copies, exclusively for archiving purposes.
e2) Regarding decompilation and reverse engineering, the German Copyright Law (UrhG) and other intellectual property rights apply.
e3) If the software was produced with the .NET framework from Microsoft, the components supplied by Microsoft are governed by the respective "end user license for Microsoft Software", which in this case is also included with the software. The distribution of the components supplied by Microsoft must also comply with the "end user license for Microsoft Software" and may only take place in conjunction with the software governed by this license.

3. License fee
For purchased source code the following provisions apply:
The rights granted by this license are subject to a fee to be paid by licensee in accordance with the relevant invoice issued. License fees are set out in licensor's price lists as are issued from time to time or are based on quotations made on an individual basis. The license is only granted when full payment of the license fee is received by the licensor.

4. Restrictions
a) All intellectual property rights in the software and user documentation are owned by the licensor or the suppliers of the licensor, especially the programs, software, texts, pictures, animations, audio data. All rights not expressly granted under this license shall remain with the licensor or the licensor's suppliers.
b) Licensor retains the right to make modifications to the software at any time.

5. End of license and confidentiality
a) Licensee is entitled at any time to end this software license agreement by

completely and finally deleting the software from his computer and/or network and all other systems on which the software is kept.
b) This license shall automatically end, if licensee does not comply with the terms and conditions of the license. In such case, the licensee is obliged to destroy all copies of the software and to return all materials accompanying the product, documentation and know-how which was given in written form to the licensor. Licensee has no right to retain any such materials.
c) For "purchased software" and "example software" the right to sell products which contain the software or parts of the software in executable form shall end as soon as this license ends.
d) Both the licensor and the licensee are obliged to keep secret all information about the other party which was obtained from or became known through executing this license. This includes knowledge about the product and business policies as well as distribution methods, especially all information which was expressly declared confidential or which is by nature classifiable as technical or trade secrets.
e) In the event of collaboration with or involvement of third parties, licensor and licensee hereby undertake to bind such third parties to the obligations set out above.

6. Limited warranty
a) The licensor does not accept any responsibility and shall not be held liable for the results obtained through use of the software, nor does licensor warrant that any particular results be achieved by the software. This also holds for the suitability or usability of the software package for the intended purpose or any other purpose. Economically purposeful usability of the software is at the sole risk of the licensee.
b) For purchased software the following provisions apply:
  b1) Licensor warrants that the software substantially conforms to the applicable documentation and that it is free from major defects which would restrict its usability for the purposes licensed. Minor defects are not considered to restrict the ability of the software to be used for the licensed purpose. This warranty does not include or relate to modifications made by the licensee.
  b2) If licensee complains about one or more defects, the licensor is entitled to remove these defects at his cost or to deliver an adequate substitute.
  b3) Should attempts at rectification of defects not succeed, the licensee shall be entitled, at his discretion, to either pay a reduced license fee or to withdraw from the contract.
c) For software not purchased the following provisions apply: There is no warranted quality or properties of the software. Also the description in the documentation does not warrant certain quality or properties of the software. Licensor does not warrant that the software is free from major defects which would restrict its usability for the purposes licensed.
d) For example software the following provisions apply: There is no warranted quality or properties of the software. Also the description in the documentation does not warrant certain quality or properties of the software. Licensor does not warrant that the software is free from major defects which would restrict its usability for the purposes licensed.
e) The warranty period is 12 month for commercial customers and juristic persons and 24 month for consumers / end-users, commencing from the point of time of delivery of the software licensed.

7. Limitation of liability and indemnity obligation
a) Software is inherently complex and may not be completely free of errors. The software is for this reason not suitable for use in products or in a way that might result in damages in the event of software faults. This is especially the case if the software were to be used in safety-related applications such as in medical applications, aerospace and space applications, traffic technology, nuclear and military technology. This is expressly acknowledged by licensee when using the software. The results and possible damages resulting from the use of the software are the sole responsibility of the licensee. The licensee is obliged to verify and secure the fitness of the software for any particular purpose to which it is put.
b) The licensor and the suppliers and agents of the licensor shall not be held liable unless they act intentionally or with gross negligence. Liability claims for whatever legal basis, especially breach of an obligation other than by delay or impossibility, breach of duty during contract negotiations and tort are excluded. This does not hold if liability is mandatory, e. g. according to the german product liability law, in the case of intention or gross negligence,

absence of warrented quality, violation of substantial contractual obligations, death or personal injury accountable to the licensor.
c) Liability is in particular excluded for the following damages: Licensor is not liable for loss of data. Licensee agrees to ensure that sufficient backups are made and appropriate data security measures are undertaken. Expressly excluded is the liability for loss of profit, interruption of business, goodwill, loss of business information or other property damages resulting from the use of the software or the fact that it cannot be used. Liability is also expressly excluded for incidental, untypical or consequential damages. This shall hold even if licensor has been advised of the possibility of obtaining such damages. A shift in the burden of proof for the disadvantage of the customer is not obtained with this regulation. Licensor hereby advises licensee to regularly check the results of licensee’s work and to secure data on a regular basis.
d) In no case shall licensor’s liability for damages exceed the amount paid by licensee for the software out of which such claim arose.
e) The limitation of liability shall also extend to all employees, representatives, agents, and suppliers of the licensor.
f) Licensee shall indemnify and release licensor from and against and defend from any claim, suit or proceedings relating to product liability.
g) Licensee shall be liable for all statements and claims made by licensee for distribution and marketing purposes.

8. Intellectual property rights of third parties
a) Licensor believes that for the Federal Republic of Germany the use of the software according to the license does not affect the intellectual property rights of third parties. If the use of the software nevertheless affects the intellectual property rights of third parties within the Federal Republic of Germany, licensor is liable against such third parties. Licensor does not warrant that the software is free of intellectual property rights of third parties outside the Federal Republic of Germany. Licensee is obliged to ensure that the software can be legally used outside the Federal Republic of Germany.
b) Licensee shall inform licensor immediately if third parties claim that intellectual property rights have been infringed. Licensor shall pay the cost of any legal action arising from claims in relation to an infringement of intellectual property rights within the Federal Republic of Germany, and shall manage and control the defence or settlement of any such claims. Licensee shall pay the costs of any legal action for cases outside the Federal Republic of Germany
c) If the use of the software according to this license affects the intellectual property rights of third parties within the Federal Republic of Germany, the licensor, at his sole discretion but with consideration to the licensee’s situation, shall decide whether to obtain a license, change the software or replace the software fully or partially, or ends the license.
d) If licensor does not settle the issue of intellectual property rights of third parties within the Federal Republic of Germany, licensee is entitled to withdraw from the license agreement. In the event that the holder of property rights prohibits licensee from exercising the license, payments of all kind made (e.g. compensation payments) shall entitle the licensee to claim or reclaim part of the licensing fee paid under clause 3 above of the license.
e) For software not purchased licensor can only be held liable due to infringement of intellectual property rights of third parties according to the provisions of this license in the case of gross negligence or intent.

9. Miscellaneous
a) Transfer by the licensee of the rights granted under this license is only permissible if the prior written consent of the licensor to the transfer has been obtained.
b) Set-off against the obligation to pay the license fee can only be made by the licensee if he has accounts receivable from the licensor that have been accepted or have been legally validated.
c) If certain issues are not covered in this agreement the licensors general terms of sale shall apply as far as necessary for resolving the respective issue. Otherwise this license contains all contractual agreements made between the parties. No other oral or written agreements have been made. Modifications and additions to this contract shall only be valid if they have been made in writing.
d) In the event that any of the above provisions are held to be in violation of applicable law, void, or unenforceable in any jurisdiction, both parties agree to replace such provision(s) with a valid agreement that will - as far as is

possible - achieve or lead to the same economic results as those intended, and that best complies with the overall intent of this contract.
e) Place of performance is the place of business of the licensor.
f) Place of jurisdiction for all issues arising from this license is, as far as is allowed by sec. 38 of the German Civil Code (ZPO), the seat of the licensor. This Agreement is solely governed by the laws of the Federal Republic of Germany. Application of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded.

Version Date: 16.05.2007