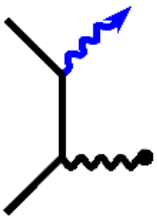
Quickstart Tutorial

for the Simplify Technologies GUI library



Simplify Technologies GmbH
Steinbuehlstrasse 15
D-35578 Wetzlar
Germany

Simplify Technologies GmbH
Steinbuehlstrasse 15
D-35578 Wetzlar, Germany
Tel.: (+49) (0)6441-210390
FAX.: (+49) (0)6441-210399
Internet: www.simplify-technologies.de
Email: hansjuergen.dreuth@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, 2001, 2002, 2003, 2004, 2005, 2006.

Contents

The purpose of this tutorial is to describe the necessary steps to use the GUI–library. Because of simplicity, we first show how to realize a classical “Hello World”–program. This program just writes some text on the LC–display. Later on, this example will be extended to use some of the graphic functions of the library, so that you get an impression of the possibilities of the GUI–library.
If there is the need to input data from a touch glass, then your hardware must support a system timer. The necessary configurations will be described and the calibration and the use of the touch glass will be explained.
All examples in this tutorial can be found on the CD of GUI–library in the repective directories (see the file “Content.txt”).

1 Basics

In order to quickly get first results with the Simplify–Technologies GUI–library, we recommend the following approach:

1. Copy all files of the GUI–library and the hardware drivers to your harddisk (choose any place in the directory path). The installation is explained in the file “INSTALL.TXT” .

2. Add all files of the library to your development environment (in the following paragraph for instance this would be the project “Hello World”). The manual of your development environment describes how to add files.

3. Modify the file “portab.h”, so that it will fit to your development environment: Due to the fact that the ANSI–C standard does not define a precise length of integer types, the GUI–library only makes use of the integer types, which are defined in “portab.h” (`uint8`, `int8`, `uint16` etc.). The numbers behind the specific integer types determine how many bits are available. The size of the variables your compiler uses, are defined in the file “limits.h”, which comes with your compiler. Change the corresponding definitions in “portab.h”.

4. The GUI–library configuration files “config_gui.h ”and “config_hardware_gui.h”have to be modified according to your needs, too. Some hints to this subject are given in the following sections of this tutorial.

2 Hello World

In order to use your LCD, you need an appropriate driver for the build–in LCD–controller. Insted of purchasing one, it is also possible to write the driver yourself (see the interface documentation in the manual).

2.1 Configuration of the GUI–library

To configure the library, you only have to modify the files “config_hardware_gui.h” and “config_gui.h”. It is therefore not necessary to manipulate the source code of some functions of the GUI–library.
The definitions in “config_hardware_gui.h” are used to describe the hardware environment of your system (e.g. the resolution of the display). Place into comments the definitions for the hardware components you do not need (e.g. touch glass).

In the configuration file “config_gui.h” you can choose which parts of the code are linked to your program. This is done by `#define` statements.
The file “config_gui.h” has two main parts: Part 1 is used for the configuration of the software environment. Here you can decide which modules of the library are necessary for your target system. The code size will be significantly reduced, because of conditional compilation even inside single functions.

Please keep in mind, that the components of the library not chosen are not available to your application. If you call the corresponding functions, you will get compiler errors, which does not constitute a defect of the GUI library. We therefore recommend to recompile all files, which include “config_gui.h”, after a modification of the configuration.

In the following second part, your selection will be evaluated. If you have chosen some incompatible options, a compiler error will notify you (e.g. it does not make sense to use a touch keyboard without a touch glass device).

You need only a few settings, to display text or some simple graphics.

2.1.1 config_hardware_gui.h

The LCD–hardware is basically controlled by options for the resolutions `_LCD_PHYS_SIZE_X`, `_LCD_PHYS_SIZE_Y` and it has to be activated with `_LCD_PRESENT = 1`. Enter the according values of your display at `_LCD_PHYS_SIZE_X` and `_LCD_PHYS_SIZE_Y`. Other hardware functions, like the touch glass, are switched off. The file “config_hardware_gui.h” should look as follows:

```
/*************************************************************************/
/* config_hardware_gui.h                                                 */
/* Defines hardware and options                    .                     */
/*                                                                       */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de          */
/*************************************************************************/


#ifndef __CONFIG_HARDWARE_GUI_H
#define __CONFIG_HARDWARE_GUI_H


/*****************************/
/* defines for the lcd display */
```

```
/*******************************/
#define _LCD_PRESENT         1 /* makes display available */
#define _LCD_LIGHT           0
#define _LCD_LIGHT_ADJUST    0
#define _LCD_CONTRAST_ADJUST 0
#define _LCD_INVERTED        0

#define _LCD_PHYS_SIZE_X   320 /* width in pixels */
#define _LCD_PHYS_SIZE_Y   240 /* height in pixels */


/*******************************/
/* other functions are shut off */
/*******************************/
#define _TGLASS_PRESENT      0 /* no touch glass */
#define _PIEZO_PRESENT       0 /* no piezo loudspeaker */


#endif /* __CONFIG_HARDWARE_GUI_H */
```

The original file of the distribution has much more comments and therefore is larger.

2.1.2 config_gui.h

Like mentioned earlier, the file "config_gui.h" consists of two parts, whereas the user only should modify part one. You can find a precise description of all the options within the manual and in the comments of the file. For the first example the following `#define` statements in the first part of "config_gui.h" are required. **Every not mentioned option must be places into comments and the second part of the file (internal definitions) must be untouched.**

```
/*****************************************************************************/
/* config_gui.h                                                              */
/* Defines configuration options for the GUI Library     .                   */
/*                                                                           */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de              */
/*****************************************************************************/


#ifndef __CONFIG_GUI_H
#define __CONFIG_GUI_H


/* read definition of hardware features used by the GUI */
#include "config_hardware_gui.h"


/*###########################################################################*/
/* Defines for customizing the GUI library                                   */
/*                                                                           */
/* Conditional compilation is used to tailor the code so that only the       */
/* modules and options needed are included into the binary thus optimizing   */
/* memory usage and execution speed.                                         */
/*                                                                           */
/* Please carefully read the following customizing options and activate the  */
/* defines if you want to activate the respective feature                    */
/*###########################################################################*/


/*****************************************************************************/
/* general customization                                                     */
/*****************************************************************************/

/* define here how many single system ticks occur during one second. Please note
that the maximum value allowed is 49700 */
```

```
#define _TICKS_PER_SECOND 100


/* the following define does not need to be changed. It allows you to call the
function system_wait_ticks like this: system_wait_ticks(3*SEC); */

#define SEC _TICKS_PER_SECOND


/* if you do not define _SYSTEM_TIMER_PRESENT the only timing function available
will be system_wait_ticks, which will just be waiting in a loop for a while.
In that case if you want to use the system_wait_ticks function define the
following calibration constant to a value that timing by software delay is
allright (for a Hitachi H8/300H with 14.7456 MHz a value of 370000 is ok): */

#define _SOFTWARE_DELAY_NUMBER 370000


/* Define the following, if your system features dynamic memory
allocation as required by ANSI-C. */

/*
#define _MALLOC_AVAILABLE
*/


/* If you do not provide dynamic memory allocation, a simple memory
allocator is provided, which can be enhanced if neccessary.
This memory allocater uses a static heap, which size can be configured with
_STORAGE_ALLOCATER_HEAP_SIZE (in bytes).
A list of free memory areas is needed as well, to organize the heap. This
list has a maximum number of entries (_MAX_FREE_LIST_ENTRIES). */

#define _STORAGE_ALLOCATOR_HEAP_SIZE 5000
#define _MAX_FREE_LIST_ENTRIES       30


/* Defines how many bytes a single display line on a virtual display may need.
This should be defined always. */

#define _MAX_BYTES_PER_LINE 320


/* if you are sure that connection of devices to channels is done correctly in
your application and no bogous pointers would mess up the system, you may omit
the checks for verifying these connections. In embedded systems with fixed
application code you eventually will reach this stage anyway ;-)
Note that parameter checking is not ommited */

#define _MAKE_CHECKS


/*****************************************************************************/
/* customizing the display output                                            */
/*****************************************************************************/

/* define, if you want to use the lcd device (i. e. a lcd display) */

#define _USE_LCD_DEVICE


/* define, if you want to use the sprite functions */

#define _SPRITE_WANTED


/* please define how many bits per pixel will be used to represent the color or
```

```
   grayshade of one LCD pixel (in monochrome: 1, in 256 colors: 8) */

#define _BITS_PER_PIXEL 1


/* if you want to include the code for the functions for lines, define: */

#define _GENERAL_LINES_WANTED


/* if you want to include the code for black and white bmp pictures define: */

#define _BMP_WANTED


/* if you want to include the code for the functions for circles and arcs,
define: */

#define _CIRCLES_WANTED


/*############################################################################*/
/*############################################################################*/
/*  Internal definitions based on what was defined above,                     */
/*     !!! NO CHANGES REQUIRED BELOW THIS LINE !!!                            */
/*############################################################################*/
/*############################################################################*/


...


/*############################################################################*/
#endif /* __CONFIG_GUI_H */
```

2.2 The program

A simple program which displays a little bit of text (the traditional *Hello World*) serves as a starting point. The project of this program can be found on the CD.

```
/*****************************************************************************/
/* main.c      "Hello World"                                                 */
/*                                                                           */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de              */
/*                                                                           */
/* last change: 7 Mar 2006                                                   */
/*****************************************************************************/

#include "main.h"               /* includes for this file                    */
#include "platform_periphery.h" /* includes for special features of the      */
                                /* hardware                                  */

#include "portab.h"      /* includes for operating system functions and defs */
#include "errors.h"      /* error codes                                      */
#include "system.h"      /* includes for the "kernel"                        */
#include "lcd.h"         /* includes lcd-driver                              */
#include "display.h"     /* includes for graphical functions                 */
#include "mono8x16.h"    /* includes of the font data                        */


int main( void)
{
   lcd_device      *system_lcd;  /* pointer to lcd driver                    */
   display_channel  display;     /* data structure for the drawing functions */

   platform_periphery_init();    /* init special periphery of your hardware  */
```

```
   system_init();               /* initialize system timer (not yet) and    */
                                /* font list                                */

   /* define a screen with your systems resolution                          */
   lcd_init( _LCD_PHYS_SIZE_X, _LCD_PHYS_SIZE_Y, LCD_NORMAL, &system_lcd);

   disp_open( &display);        /* generate the display and connect it to   */
   disp_make_current( &display); /* lcd-driver, which controlls the         */
   disp_connect( system_lcd);   /* hardware                                 */

   disp_set_font( __Mono8x16);          /* desired font for text output on  */
   disp_set_drawmode( DRAW_SET);        /* the display                      */
   disp_set_textstyle( TXT_NORMAL);

   disp_set_cursor_position( 100, 120); /* print some text at the defined   */
   disp_text( "Hello World!");          /* position                         */
   disp_set_cursor_position( 100, 100);
   disp_text( "Hello User!");

   /* cleanup would shut off the lcd and also the printed text               */
   /*
   disp_disconnect();
   disp_close( &display);
   lcd_close();
   */

   while (1) {}; /* do something ... */

   return (ERR_OK);
}/* main */
```

Let's look at the different sections of the source code. First you have to initialize the hardware and the fonts, of cause. In this case only the system font is used:

```
   platform_periphery_init();   /* init special periphery of your hardware  */
   system_init();               /* initialize system timer (not yet) and    */
                                /* font list                                */
```

As mentioned in the introduction, the GUI–library uses a two layer system to access the hardware components. The hardware is directly controlled by a layer (*device*) which is very close the hardware. The application software has access to the hardware through a second layer (*channels*). The advantage is, that a channel can be handled easier and it is possible to connect this channel with another device (e.i. a display channel can be used together with a printer or a LCD). The output on the different devices is then comparable. The following lines initialize a display channel and connect it to the LCD device driver:

```
   /* define a screen with your systems resolution                          */
   lcd_init( _LCD_PHYS_SIZE_X, _LCD_PHYS_SIZE_Y, LCD_NORMAL, &system_lcd);

   disp_open( &display);        /* generate the display and connect it to   */
   disp_make_current( &display); /* lcd-driver, which controlls the         */
   disp_connect( system_lcd);   /* hardware                                 */
```

Afterwards a font (system font) and it's display options are chosen. In this case DRAW_SET and TXT_NORMAL lead to "normal" text in the recent color on the display.

```
   disp_set_font( __Mono8x16);          /* desired font for text output on  */
   disp_set_drawmode( DRAW_SET);        /* the display                      */
   disp_set_textstyle( TXT_NORMAL);
```

The text is drawn on the display after the cursor is positioned. (this and the following demo programs try to assume nothing special in respect to the hardware, but for simplicity it is easier to assume a fixed display resolution of 320×240 pixels. Please keep this in mind, if you see some absolute positioning.).

```
disp_set_cursor_position( 100, 120); /* print some text at the defined    */
disp_text( "Hello World!");          /* position                         */
disp_set_cursor_position( 100, 100);
disp_text( "Hello User!");
```

In general, the origin of the coordinate system the GUI–library uses, lies at the bottom, left corner of the display and the coordinates increase if you are going right and up. At the end of the application there usually is some cleanup necessary. These three commands were commented in the example, to prevent the LCD from being erased.

```
disp_disconnect();
disp_close( &display);
lcd_close();
```

Your system can do something else now.

```
while (1) {}; /* do something ... */
```

2.2.1 Lines and circles

The last program shall now be modified to show some lines and circles instead of the text. You find the project of this program on the CD. If you look at the source code of “main”, you will notice by comparing with “Hello World”, that only the following lines are added.

```
/* draw some lines                                                        */
disp_set_color( BLACK);
disp_set_drawmode( DRAW_EXOR);
for (i = 0; i < _LCD_PHYS_SIZE_X; i++)
{
   disp_set_cursor_position( 0, 0);
   disp_line( i, _LCD_PHYS_SIZE_Y - 1);
}/* for */
for (i = 0; i < _LCD_PHYS_SIZE_Y; i++)
{
   disp_set_cursor_position( 0, 0);
   disp_line( _LCD_PHYS_SIZE_X - 1, i);
}/* for */
/* draw some circles                                                      */
disp_set_color( WHITE);
disp_set_drawmode( DRAW_SET);
disp_set_cursor_position( _LCD_PHYS_SIZE_X/2, _LCD_PHYS_SIZE_Y/2);
for (i = 1; i <= 100; i = i + 5)
{
   disp_circle( i);
   disp_circle( i + 1);
   disp_circle( i + 2);
}/* for */
```

`disp_set_color()` is used to adjust the drawing color (default is BLACK at the start of your application). Morover a new drawing mode is introduced: EXOR. After that every pixel which is drawn will be connected with the existing pixel through the exclusive–or operation. To draw a line, you have to position the graphic cursor first `disp_set_cursor_position()` and then use the `disp_line()` command to finish the line. Drawing a circle works similar: first the cursor must be positioned at the center, then `disp_circle()` draws the circle with the desired radius.

2.2.2 Bitmaps and sprites

The “Hello World”–program is now again extended to display a bitmap on LCD. Additionally a pointer (sprite) shall move over the display. The program for that project can be found on the CD.

It is very simple to display a picture in the bmp–format. You place the cursor at the desired lower, left corner of the picture and draw the picture with the command `disp_bmp()`. You only need a pointer to the data of the picture. In this example the picture was first converted to hex–numbers, which were placed into an array. This array is then included to the program with the file “simplifylogo.h ”.
The following lines are needed:

```
disp_set_cursor_position( 55, 80); /* set the bitmap of the logo          */
disp_bmp( simplify_logo);          /* on the LCD                          */
```

A sprite is a small bitmap picture with a resolution of 16×16 pixels. Sprites can be moved on the display without changing the actual contents of the screen. In the example the sprite data is placed in an array. This array contains both the sprite data and mask information (also 16×16 pixels). This mask determines the pixel, which have are to be displayed and which are not.

```
/* pattern for sprite cursor with its respective mask pattern */
fillstyle arrow_sprite[32] =
{
   0x0600, 0x0900, 0x0900, 0x9200, 0xD200, 0xA400, 0x8400, 0x8780,
   0x8100, 0x8200, 0x8400, 0x8800, 0x9000, 0xA000, 0xC000, 0x8000,
   0x0600, 0x0F00, 0x0F00, 0x9E00, 0xDE00, 0xEC00, 0xFC00, 0xFF80,
   0xFF00, 0xFE00, 0xFC00, 0xF800, 0xF000, 0xE000, 0xC000, 0x8000
};
```

The command `disp_set_sprite_pattern()` takes a pointer to the sprite data and the coordinates of the ”hot spot”. During drawing operations the sprite is positioned according to it’s hot spot. Before turning on the sprite with `disp_sprite_on()`, the drawing position should be set with `disp_set_sprite_position()`. The command `disp_sprite_on()` is also useful to set the drawing mode of the sprite. The sprite is turned off with `disp_sprite_off()`).

```
disp_set_sprite_pattern( arrow_sprite, 0, 15); /* activate sprite         */
disp_set_sprite_position( 16, _LCD_PHYS_SIZE_Y/2);
disp_sprite_on( DRAW_EXOR);
```

The following lines of source code cause the sprite to move back and forth across the display. `system_wait_ticks()` is here only used to slow down the movement. Do not forget to adjust this function in the file “config_gui.h” (see section 2.1.2).
Instead of `system_wait_ticks()` you can of course use any other delay loop.

```
system_wait_ticks( 100);
while (1)
{
   for (i = 16; i < _LCD_PHYS_SIZE_X - 16; i++)
   {
      disp_set_sprite_position( i, _LCD_PHYS_SIZE_Y/2);
      system_wait_ticks( 10);
   }/* for */
   for (i = _LCD_PHYS_SIZE_X - 16; i > 16; i--)
   {
      disp_set_sprite_position( i, _LCD_PHYS_SIZE_Y/2);
      system_wait_ticks( 10);
   }/* for */
}/* while */
```

3 System timer

It often is usefull to have a blinking cursor. The GUI–library can provide this with the help of sprites, if your hardware has an additional system timer. A system timer is also needed to read out a touch display.

3.1 Driver functions for the system timer

Some hardware dependent functions are necessary to use the system timer. The most important is an interrupt routine, which is called by the timer `TICKS_PER_SECOND` times per second. This routine must increment the variables `TIMERDATA` and `DAYDATA` (see figure 1). `TIMERDATA` and `DAYDATA` must be 32 bit variables, which are available to the GUI library, if the library needs this time information. This interrupt function must be implemented to match your microcontroller. In order to avoid an overflow of `TIMERDATA` before the carry `DAYDATA` takes place, the maximum frequency of the system timer is 49710 Hz (i.e. `TICKS_PER_SECOND` must be smaller than 49710, too). Normally it is sufficient to have 50 to 100 timer interrupts per second.
The function `system_init()` (from “system.c”) sets the two variables `TIMERDATA` and `DAYDATA` to zero and starts the system timer by calling `os_systemtimer_start()`. The function `os_systemtimer_start()` must be implemented in such a way, that the timer interrupt for the system timer is started correctly.

SYSTEMTIMERFUNC — Necessary interrupt routine of the system timer

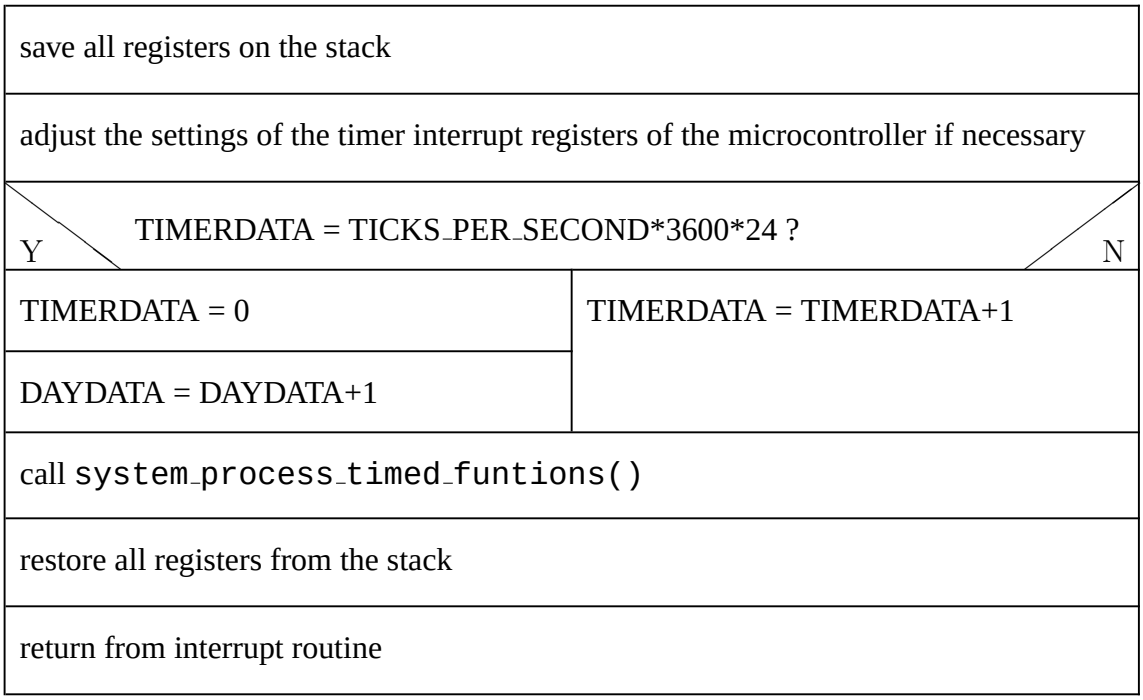


Figure 1: Structogram of the timer interrupt routine

os_flag_block() — Functions which set the semaphores of the timer interrupt

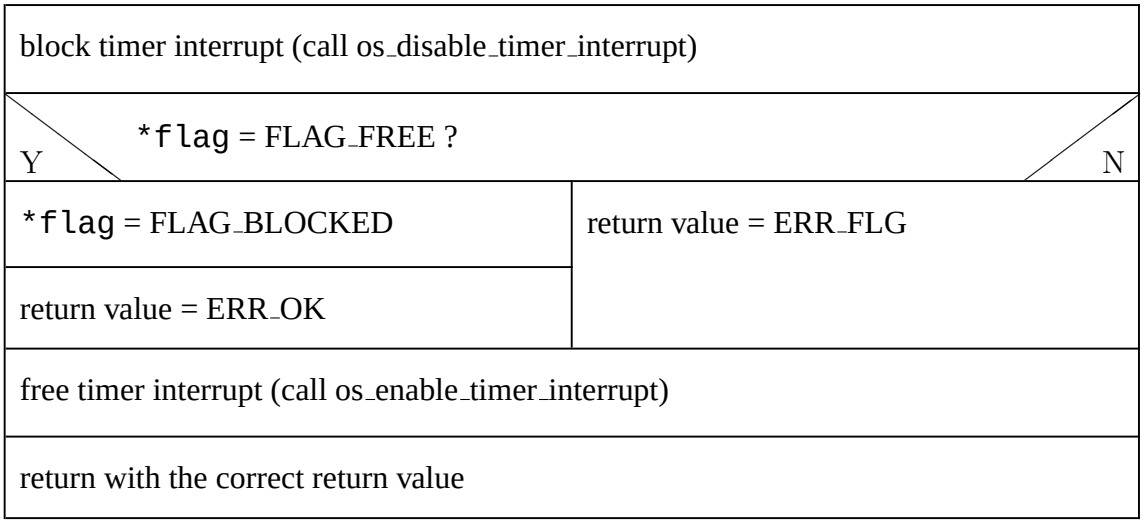


Figure 2: Structogram of `os_flag_block(uint8 *flag)`

At the end of the timer interrupt routine, you need to call the function `system_process_timed_functions()` of the GUI–library. This procedure calls other functions, which need to run at fixed intervals by the use of a pointer list. One job of these functions is the controll of periodical events, e.g. the blinking of the cursor, as mentioned before.
During runtime it might be possible, that the list of *"timed"*–functions changes.
To avoid problems according to these changes, there are two functions which allow to block and free the timer interrupt: `os_flag_block()` and `os_flag_free()`. Therefore these functions use two simple *8–bit semaphores*, which can have the values `FLAG_BLOCKED` and `FLAG_FREE` (both defined in "os_gui.h"). As you see in the description of `os_flag_block()`, the system timer is not really blocked for a longer period of time, but only the flag is set. The critical functions in the GUI–library consider this flag to prevent conflicts with the *timed*–functions.
We recommend to implement the functions `os_flag_block()` and `os_flag_free()` according to the following description.

err_code os_flag_block(uint8 *flag): This function disables the timer interrupt (see figure 2) by calling `os_disable_timer_interrupt()` (see below). After checking the flag, it will be set to `FLAG_BLOCKED`, if it had the value `FLAG_FREE` before. After that, the interrupt is enabled again by calling `os_enable_timer_interrupt()` (see below). Return value: `ERR_OK`, if the flag was free and it also was possible to block it, `ERR_FLG`, if the flag was already blocked.

err_code os_flag_free(uint8 *flag): This function sets the flag to `FLAG_FREE` and thus frees the blocked resources. It is not allowed, that the read out of the flag is disturbed by other interrupts. If that is possible in your system, then you must also disable these interrupts before reading the flag and enable them afterwards.

`os_flag_free()` is and must only be used, if the calling function has blocked

the flag for its own use (i.e. this function has the *right* to modify the flag). Return value of the function: `ERR_OK`.

The following functions must allow to switch on and off the timer interrupt. They are used for save handling the semaphores and also for the internal function `system_process_timed_functions()`.

void os_disable_timer_interrupt(void): Disables the system timer interrupt.

void os_enable_timer_interrupt(void): Enables the system timer interrupt.

In order to use the time in delay loops for other tasks usefully, the following function is needed.

void os_process(void): This function is called inside of the `system_wait` functions, so that the system can handle other tasks during waiting. The declaration and implementation of this function could be done in “main.h” and “main.c”, respectively.

If you do not need this feature, it can be switched off in the file “config_gui.h” by omitting the definition `_OS_USED` (see next section) and the implementation of the function.

The declaration of these functions and of constants, which are used by them, can be found in the file “os_gui.h”. The implementation of these function should be done in assembler language, in order to have full control over the interrupt handling.

3.2 Configuration of the GUI–library

Additionally, activate the lines of source code (the `#define` statements) mentioned in the extract of the file “config_gui.h”(the comments are shortened here). The option `__DISABLE_TIMER_INTERRUPT_NEEDED` is needed with most microcontrollers, because often it is not possible to manipuliate the pointer list of the timed–functions with one assembler directive. The last option `_SPRITE_BLINK_WANTED` makes sure, that the timer function, which controlls the blinking of the sprite, is added to the list of timed functions.

```
/*****************************************************************************/
/* general customization                                                     */
/*****************************************************************************/

/* Using the library only for displaying information on a lcd display
is not time dependent and therefore does not require you to implement a
ßystem time". However a ßystem time" is required if you want to use the
following functionallity which inherently must distinguish different times:
- Touch glass functions (and therefore also buttons and touch keyboards).
- Automatical blinking of sprite cursors
...
*/

#define _SYSTEM_TIMER_PRESENT

/* if _SYSTEM_TIMER_PRESENT is defined functions for reading touch glasses and
blinking sprite cursors may be (dependend on the configuration) automatically
```

```
inserted into the system timer handler.
The system timer must find valid data at each node of the list which handles the
functions to be called periodically when the
timer interrupt occurs (otherwise the system crashes). Thus the process of
inserting or removing the pointer to a function to be triggered by the system
timer interrupt must happen so that writing this information cannot be
corrupted by system timer processing the unfinnished list of timed functions
To ensure this there are 2 ways:
1. If the processor can write the pointers in one machine instruction nothing
   special is required as the system timer interrupt routine will always find
   consistent data
2. If (mostly on smaller controllers) the pointer cannot be written in one
   machine instruction, writing it could be interrupted resulting in
   inconsistent data. To avoid this it then is required to disable the timer
   interrupt during this operation. This is realized by calling the functions
   os_disable_timer_interrupt and os_enable_timer_interrupt, which are included
   into the code of the functions system_timed_function_on and
   system_timed_function_off (in the file system.c) if
   __DISABLE_TIMER_INTERRUPT_NEEDED is defined.
NOTE: This consideration is only relevant if system timer functions are
needed */

#define __DISABLE_TIMER_INTERRUPT_NEEDED


/* Define the following, if your system features dynamic memory
allocation as required by ANSI-C. */

#define _MALLOC_AVAILABLE


/* If you do not provide dynamic memory allocation, a simple memory
allocator is provided, which can be enhanced if neccessary.
This memory allocater uses a static heap, which size can be configured with
_STORAGE_ALLOCATER_HEAP_SIZE (in bytes).
A list of free memory areas is needed as well, to organize the heap. This
list has a maximum number of entries (_MAX_FREE_LIST_ENTRIES). */

#define _STORAGE_ALLOCATOR_HEAP_SIZE 5000
#define _MAX_FREE_LIST_ENTRIES       30


/* if an operating system is used and wants to gain control during wait
functions (in this case it is called by os_process, which you must provide),
define: */

/*
#define _OS_USED
*/


/*****************************************************************************/
/* customizing the display output                                            */
/*****************************************************************************/

/* define, if you want to have an automatically blinking sprite cursor */

#define _SPRITE_BLINK_WANTED
```

3.3 Example with a blinking sprite

With a correct configuration file “config_gui.h” you only have to add the following line of source code to the example program of section 2.2, to make the sprite blink.

```
   disp_set_sprite_blink_interval( 50);           /* now blinking          */
```

Altogether the source code looks as follows. The sprite is not moved any more, because the blinking then can easier be seen. Please note that the system timer is started with `system_init()`. This example can be found on the CD.

```
/*****************************************************************************/
/* main.c      bitmap and blinking sprite demo                               */
/*                                                                           */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de              */
/*                                                                           */
/* last change: 7 Mar 2006                                                   */
/*****************************************************************************/

#include "main.h"              /* includes for this file                     */
#include "platform_periphery.h" /* includes for special features of the      */
                               /* hardware                                   */

#include "portab.h"       /* includes for OS functions and defs              */
#include "errors.h"       /* error codes                                     */
#include "system.h"       /* includes for the "kernel"                       */
#include "lcd.h"          /* includes lcd-driver                             */
#include "display.h"      /* includes for graphical functions                */
#include "simplifylogo.h" /* data for the logo                               */

/* pattern for sprite cursor with its respective mask pattern                */
fillstyle arrow_sprite[32] =
{
   0x0600, 0x0900, 0x0900, 0x9200, 0xD200, 0xA400, 0x8400, 0x8780,
   0x8100, 0x8200, 0x8400, 0x8800, 0x9000, 0xA000, 0xC000, 0x8000,
   0x0600, 0x0F00, 0x0F00, 0x9E00, 0xDE00, 0xEC00, 0xFC00, 0xFF80,
   0xFF00, 0xFE00, 0xFC00, 0xF800, 0xF000, 0xE000, 0xC000, 0x8000
};


int main( void)
{
   lcd_device      *system_lcd; /* pointer to lcd driver                     */
   display_channel  display;    /* data structure for the drawing functions  */

   platform_periphery_init();   /* init special periphery of your hardware   */
   system_init();               /* initialize system timer and font list     */

   /* define a screen with your systems resolution                           */
   lcd_init( _LCD_PHYS_SIZE_X, _LCD_PHYS_SIZE_Y, LCD_NORMAL, &system_lcd);

   disp_open( &display);         /* generate the display and connect it to   */
   disp_make_current( &display); /* lcd-driver, which controlls the          */
   disp_connect( system_lcd);    /* hardware                                 */

   disp_set_cursor_position( 55, 80); /* set the bitmap of the logo          */
   disp_bmp( simplify_logo);          /* on the LCD                          */

   disp_set_sprite_pattern( arrow_sprite, 0, 15); /* activate sprite         */
   disp_set_sprite_position( 50, 50);
   disp_sprite_on( DRAW_EXOR);

   disp_set_sprite_blink_interval( 50);           /* now blinking            */

   while (1)
   {
      system_wait_ticks( 10);
   }/* while */

   /* disp_sprite_off(); */
   /* other clean up ... but never reached */

   return (ERR_OK);
}/* main */
```

4 Touch glass

Very often LCDs are enhanced with a touch glass, so that the user can interact with the system using the display. The GUI–library supports this function, however, your system must provide a timer and the modifications in the previous chapter 3 must be done.

4.1 Driver functions for the touch glass

In order to use a touch glass, your system has to provide two functions, which depend on your hardware. First, you need a function, which initializes the touch hardware and second, there must be a function, which reads out the touch glass in arbitrary units (e.g. the data of the A/D–converters in some units). Both function are declared in the file "tglass_.h".

void c_tglass_ground_state(void): This function initializes the touch glass hardware and sets it into ground state before a readout sequence (e.g. with foil touch glasses a certain potential is applied to the foils).

void c_tglass_get(uint16 *ptr): This function reads the coordinates of the touch glass. Here, provide uncalibrated data, e. g. the *raw* values of the A/D–converters, which represent the position, (some voltage values for the x/y coordinates). The conversion into pixel units is performed automatically and only if needed. The coordinates obtained must be written into an array of two 16 bit values, on which a pointer points (**first** the y–coordinate, then the x–coordinate!).

For correct calculation of the calibration coefficients with integer arithmetic, the precision of this values must be at least equal to the resolution of the display under the touch glass. Moreover there must be a linear relation between the touch glass data and the geometrical position.

If your hardware provides a real resolution, which is lower than the resolution of the display, the values need to be scaled up.

If the touch glass is not pressed or the coordinates are not valid, then the values `_AD_TOUCH_NOT_PRESSED` or `_AD_TOUCH_NOT_VALID` should be returned as a 16 bit value (to be defined in the file "config_hardware_gui.h" , see section 4.2). This also means, that the hardware should normally not return this values in other cases.

Normally, this function is called periodically by the timer interrupt and should therfore be optimized in terms of runtime.

Remark: At touch displays with an resistive touch foil, the RC time of this setup is important. In order to get reliable coordinates, the voltage at the foils must be sufficiently stabilized. The function `tglass_read()` (in "tglass.c"), which calls `c_tglass_get()`, takes into account, that only contacts are to be considered valid, which last at least one a period of one interval of the system timer. Thus, the foils can reach a stable potential in the interval of one tick of the system timer.

4.2 Configuration of the GUI–library

4.2.1 config_hardware_gui.h

You must activate the following #define–statements in the file “config_hardware_gui.h”, **in addition**.

```
/*****************************************************************************/
/* defines for touch glass                                                   */
/*****************************************************************************/
#define _TGLASS_PRESENT        1    /* set to 1 if touch glass is present */
#define _AD_TOUCH_NOT_VALID   8192  /* definition for an A/D converter result
                                       considered illegal */
#define _AD_TOUCH_NOT_PRESSED 4096  /* definition for touch glass not pressed */

/* define how the orientation of the x- and y- axis of the physical interface of
the touch glass is aligned with the coordinate system of the lcd display when
this is used in LCD_NORMAL orientation. */
#define _TOUCH_NORMAL
/* #define _TOUCH_ROTATED */

/* now for the Sheng display */
#define _TGLASS_A_DEFAULT_DATA    23937 /* Parameters for calibration */
#define _TGLASS_B_DEFAULT_DATA      -29 /* and conversion to pixel */
#define _TGLASS_C_DEFAULT_DATA    18322 /* coordinates */
#define _TGLASS_D_DEFAULT_DATA      -27
/* These parameters are strongly dependend on the hardware and touch glas
driver implementation.
If _TGLASS_PRESENT is not set 1 the whole tglass software module tglass.c
is omitted.
Then the following modules which depend on the existence of a touch glas are
also omitted: touch.c (the module for the touch channels), t_keyb.c (the
module for the touch keyboard devices) and keyboard.c (the module for the
keyboard channels) */

/* If you want to use the calibration function for the touch glas you need to
1. define the following coordinates as positions for crosses to appear on
   the display for calibration so they will fit on the display.
2. Adjust the function tglass_calibrate in tglass.c so the text output fits
   your display. Text strings for the output can be adjusted in text_gui.h */
#define KALIB_X1    32  /* Coordinates for the crosses marking the calibrion */
#define KALIB_Y1    224 /* points for the touch glas calibration */
#define KALIB_X2    288
#define KALIB_Y2    16
#define KALIB_X3    32
#define KALIB_Y3    16
#define KALIB_X4    288
#define KALIB_Y4    224
```

Touch support is activated with _TGLASS_PRESENT. The constants _AD_TOUCH_NOT_PRESSED and _AD_TOUCH_NOT_VALID were already discussed in section 4.1. Please set #define _TOUCH_NORMAL, if the orientation of the display connectors (hardware) aligns to the coordinate–axises of the LCD while this is being operated in ”LCD_NORMAL” mode. If the coordinate system is rotated at 90 degrees, because of the mechanical setup, then define _TOUCH_ROTATED. Only one of this options is allowed, of course. The following four definitions _TGLASS_A_DEFAULT_DATA bis _TGLASS_D_DEFAULT_DATA depend on your hardware and are determined with a calibration procedure, which is described in the next section 4.3. These constants are used to transform the ADC values into pixel coordinates. If you want to use the calibration function of the GUI–library, you have to define four pairs of pixel coordinates KALIB_X1 to KALIB_Y4. Normally, you have

to adjust these coordinates to the resolution of your display hardware. These points are where the touch readings for calibration are taken.

4.2.2 config_gui.h

The file "config_gui.h" must be slightly modified, too and some **additional** `#define` statements must be activated.
Beside the mandatory `_TOUCH_WANTED`, you can activate the option `_TGLASS_CALIBRATION_WANTED`, if the GUI–library should include some code for the calibration of the touch hardware. After the calibration store the obtained values in the file "config_hardware_gui.h", then these values will be used automatically in the future.
The option `_BUTTONS_WANTED` and `_TOUCH_KEYBOARDS_WANTED`, which control GUI elements like *buttons* and *touch keyboards*, can be safely ignored for this example.

```
/*****************************************************************************/
/* customizing touch glass functions                                         */
/*****************************************************************************/

/* if you want touch functionallity which is also the prerequisite for buttons
and touch keyboards activate the following define: */

#define _TOUCH_WANTED


/* if you want to include code for calibrating the touch glass activate the
following define. Please note, that you probably need to customize the function
tglass_calibrate in tglass.c to nicely fit your touch display */

#define _TGLASS_CALIBRATION_WANTED


/* define the following if you want to make use of touch buttons */

/*
#define _BUTTONS_WANTED
*/


/* define the following if you want to use touch keyboards */

/*
#define _TOUCH_KEYBOARDS_WANTED
*/


/* define the following if you want to use menus, sliders etc. */

/*
#define _TOUCH_ADDONS_WANTED
*/
```

4.3 Calibration of the touch glass

The calibration constants of the touch glass, **_TGLASS_A_DEFAULT_DATA - _TGLASS_-_DEFAULT_DATA** depend on the hardware and must be determined experimentally (e.g. with the calibration function `tglass_calibration_sequence()` in the file "tglass.h/c"). These constants are used to transform the voltage values of the hardware driver to the corresponding display coordinates. We assume a linear relationship of the form $x = A *$

$U_x + B$ between these values (e.g. the voltages of the ADC U_x, U_y) and the coordinates x,y.
In order to use the function `tglass_calibration_sequence()` for calibration, the LCD may not be rotated, but should be operated in the orientation `LCD_NORMAL`.
The calibration function `tglass_calibration_sequence()` shows four crosshairs on the display and requests the user to touch this points. The calibration constants can be calculated aftwerwards. To ensure a correct touch calibration, two more crosshairs are displayed, which have to be touched. If you want to use this function then you must define the necessary coordinates **KALIB_X1** to **KALIB_Y4** as mentioned earlier. Reasonable positions are near the four corners of the display.
The following program shows, how the calibration function works. It can be found on the CD. It is important, that the display channel used corresponds to the LCD, which is below the touch glass. Again, the two layer concept with drivers and channels is used to control the touch glass. Here the hardware layer is related to the `tglass_device` and is then connected with a `touch_channel` (see source code). Thereby it is possible to change the connection of the touch channel during runtime, lets say for example with an external mouse or an joystick.
Analogously to the channels and devices in section 2.2, the setup of the touch screen is done as follows:

```
tglass_init( 1, &system_tglas);            /* init touch glas hardware   */
touch_open( &touch_glas, &display, NULL);  /* open touch channel         */
touch_connect( &touch_glas, system_tglas); /* connect channel & hardware */
```

as well as the cleanup afterwards

```
/* cleanup touch and tglass */
touch_disconnect( &touch_glas);
touch_close( &touch_glas);
tglass_close();
```

The first parameter of `tglass_init()` sets the number of system ticks, which shall lie between to queries of the touch glass. The procedure `touch_open()` moreover allows to provide for an acoustical feedback (this is not used here).
The calibration routine first shows a short explanation on the display. This text must be shown on a display with a resolution of at least 320×240 pixels. If your display is smaller, then you should modify the (files “tglass.c” and “text_gui.h”accordingly.
After `tglass_calibrate()` is called, the program shows on the display the calibration coefficients calculated. Use a breakpoint here with your debugger or use a delay loop to read the calibration values. The readings obtained shall then be stored in the constants _TGLASS_A_DEFAULT_DATA to _TGLASS_D_DEFAULT_DATA in the file “config_hardware_gui.h”. At the end of the example the *devices* and *channels* used are closed.

```
/*************************************************************************/
/* main.c      calibration of the touch glass                            */
/*                                                                       */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de          */
/*                                                                       */
/* last change: 7 Mar 2006                                               */
/*************************************************************************/

#include "main.h"              /* includes for this file */
```

```
#include "platform_periphery.h" /* includes for special features of the
                                   hardware */

#include "portab.h"      /* includes for operating system functions and defs */
#include "errors.h"      /* error codes */
#include "system.h"      /* includes for the "kernel" */
#include "lcd.h"         /* includes lcd-driver */
#include "display.h"     /* includes for graphical functions */
#include "tglass.h"      /* includes tglass-driver */
#include "touch.h"       /* includes for touch functions */
#include "text_gui.h"    /* defines for calibration explanation */
#include "mono6x8.h"     /* data of a font */

#include <stdio.h>       /* includes standard I/O from ANSI C */


int main( void)
{
   lcd_device      *system_lcd;       /* pointer to lcd driver */
   display_channel display;           /* data structure for the drawing
                                         functions */
   tglass_device   *system_tglas;     /* pointer to tglass driver */
   touch_channel   touch_glas;        /* data structure for the touch
                                         functions */
   const font      *Mono6x8;          /* actual font */
   char            outputstring[50];  /* string for text output */
   uint8           calib_states;      /* current state ID of the calibration
                                         state machine */
   tglass_calib_statevars calib_vars; /* calibration state machine variables */
   err_code        err;               /* error result for the calibration
                                         process */

   platform_periphery_init();   /* init special periphery of your hardware */
   system_init();               /* initialize system timer and font list */

   /* define a screen with your systems resolution */
   lcd_init( _LCD_PHYS_SIZE_X, _LCD_PHYS_SIZE_Y, LCD_NORMAL, &system_lcd);
   disp_open( &display);         /* generate the display and connect it to */
   disp_make_current( &display); /* lcd-driver, which controlls the */
   disp_connect( system_lcd);    /* hardware */

   tglass_init( 1, &system_tglas);            /* init touch glas hardware */
   touch_open( &touch_glas, &display, NULL);  /* open touch channel */
   touch_connect( &touch_glas, system_tglas); /* connect channel & hardware */

   system_font_init( __Mono6x8);
   system_get_font_pointer("Mono6x8", &Mono6x8);
   disp_set_font( Mono6x8);   /* desired font for text output */

   /* calibration with text from "text_gui.h" */
   calib_states = 0;
   do
   {
      err = tglass_calibration_sequence(
         TEXT_CALIB, TEXT_CALIB_VERIFY, TEXT_CALIB_FAILED,
         10, 10, 80, &calib_vars, &calib_states);
   } while ( ERR_OK_FALSE == err);

   disp_clear();
   disp_set_cursor_position( 10, 40);
   if (ERR_OK == err)
   {
      sprintf( outputstring, "%s%i", "TGLASS_A = ", system_tglas->a);
      disp_text( outputstring);
      disp_set_cursor_position( 10, 30);
      sprintf( outputstring, "%s%i", "TGLASS_B = ", system_tglas->b);
      disp_text( outputstring);
      disp_set_cursor_position( 10, 20);
```

```
      sprintf( outputstring, "%s%i", "TGLASS_C = ", system_tglas->c);
      disp_text( outputstring);
      disp_set_cursor_position( 10, 10);
      sprintf( outputstring, "%s%i", "TGLASS_D = ", system_tglas->d);
      disp_text( outputstring);
   }/* if */
   else
   {
      disp_text( "Fatal error. No valid calibration parameters.");
   }/* else */

   /*
   INSERT BREAKPOINT HERE TO READOUT CALIBRATION DATA !!! OR USE
   ENDLESS WHILE LOOP TO STOP.
   */

   while (1) {}

   /* cleanup touch and tglass */
   touch_disconnect( &touch_glas);
   touch_close( &touch_glas);
   tglass_close();

   /* cleanup display and lcd */
   disp_disconnect();
   disp_close( &display);
   lcd_close();

   return( ERR_OK);
}/* main */
```

4.4 Touch Demo

The sprite example of section 2.2.2 is now modified, so that the position of the pointer is controlled by the touch glass. At the sprite position, a small square is drawn additionally. The source code of this project can be found on the CD.

```
/*****************************************************************************/
/* main.c      touch demo                                                    */
/*                                                                           */
/* (C) SIMPLIFY TECHNOLOGIES GmbH, www.simplify-technologies.de              */
/*                                                                           */
/* last change: 7 Mar 2006                                                   */
/*****************************************************************************/

#include "main.h"              /* includes for this file                     */
#include "platform_periphery.h" /* includes for special features of the      */
                               /* hardware                                   */

#include "portab.h"       /* includes for OS functions and defs             */
#include "errors.h"       /* error codes                                    */
#include "system.h"       /* includes for the "kernel"                      */
#include "lcd.h"          /* includes lcd-driver                            */
#include "display.h"      /* includes for graphical functions               */
#include "simplifylogo.h" /* data for the logo                              */
#include "tglass.h"       /* includes tglass-driver                         */
#include "touch.h"        /* includes for touch functions                   */


/* pattern for sprite cursors with their respective mask pattern */
fillstyle arrow_sprite[32] =
{
   0x0600, 0x0900, 0x0900, 0x9200, 0xD200, 0xA400, 0x8400, 0x8780,
   0x8100, 0x8200, 0x8400, 0x8800, 0x9000, 0xA000, 0xC000, 0x8000,
   0x0600, 0x0F00, 0x0F00, 0x9E00, 0xDE00, 0xEC00, 0xFC00, 0xFF80,
```

```
   0xFF00, 0xFE00, 0xFC00, 0xF800, 0xF000, 0xE000, 0xC000, 0x8000
};


int main( void)
{
   lcd_device      *system_lcd;   /* pointer to lcd driver                   */
   display_channel display;       /* data structure for the drawing funcs.   */
   tglass_device   *system_tglas; /* pointer to tglass driver                */
   touch_channel   touch_glas;    /* data structure for the touch funcs.     */

   int16 x, y;                    /* cursor position                         */
   uint32 ticks_end, days_end;    /* time stamp of touch contact             */


   platform_periphery_init();   /* init special periphery of your hardware   */
   system_init();               /* initialize system timer and font list     */

   /* define a screen with your systems resolution                           */
   lcd_init( _LCD_PHYS_SIZE_X, _LCD_PHYS_SIZE_Y, LCD_NORMAL, &system_lcd);

   disp_open( &display);         /* generate the display and connect it to   */
   disp_make_current( &display); /* lcd-driver, which controlls the          */
   disp_connect( system_lcd);    /* hardware                                 */

   tglass_init( 1, &system_tglas);            /* init touch glas hardware    */
   touch_open( &touch_glas, &display, NULL);  /* open touch channel          */
   touch_connect( &touch_glas, system_tglas); /* connect channel & hardware  */

   disp_set_cursor_position( 55, 80); /* set the bitmap of the logo          */
   disp_bmp( simplify_logo);          /* on the LCD                          */

   disp_set_sprite_pattern( arrow_sprite, 0, 15); /* activate sprite         */
   disp_set_sprite_position( _LCD_PHYS_SIZE_X/2, 50);
   disp_sprite_on( DRAW_EXOR);

   system_wait_ticks( 100);
   while (1)
   {
      /* read out touch glass */
      touch_get_contact_end( &touch_glas, &x, &y, &ticks_end, &days_end);

      /* set sprite at this position, if inside LCD */
      if ((x >= 0) && (x < _LCD_PHYS_SIZE_X) &&
         (y >= 0) && (y < _LCD_PHYS_SIZE_Y))
      {
         disp_set_cursor_position( x, y);
         disp_rectangle( 3, 3);
         disp_set_sprite_position( x, y);
      }/* if */
   }/* while */

   /* clean up is not necessary, because never reached */
   /*
   disp_sprite_off();

   touch_disconnect( &touch_glas);
   touch_close( &touch_glas);
   tglass_close();

   disp_disconnect();
   disp_close( &display);
   lcd_close();
   */

   return (ERR_OK);
}/* main */
```

The touch glass is initialized the same way as in the previous section. The position of the bitmap and the sprite are the same as in the example of section 2.2.2. Only the function for reading out the touch position `touch_get_contact_end()` is introduced here. This function returns the coordinates and the time, at which the last contact on the touch channel took place.

5 Miscellaneous

5.1 Speaker support

Electronic devices which make use of touch display technology, can be enhanced with an acoustic feedback. Thus handling is easier and there are fewer mistakes by the user because of the additional feedback. The GUI–library provides some functions for acoustic feedback (also see the function `touch_open()` in section 4.3.

5.1.1 Configuration

Activate the following `#define` statements in the file “config_hardware_gui.h”, in order to use sound support.

```
/*****************************************************************************/
/* defines for piezo loudspeaker                                             */
/*****************************************************************************/
#define _PIEZO_PRESENT         1    /* set 1 if piezo loudspeaker is present */
#define _PIEZO_VOLUME_ADJUST   2    /* set 2 if volume can be adjusted */
#define _VOLUME_DEFAULT_DATA 128    /* volume is considered to be an */
#define _VOLUME_LOWER_LIMIT    0    /* 8-bit value */
#define _VOLUME_UPPER_LIMIT  255
/* These parameters determine which functions are needed to access the piezo
loudspeaker. Functions and data for adjusting the volume are omitted if
_PIEZO_VOLUME_ADJUST is not set to 2.
If _PIEZO_PRESENT is not set to 1 the whole piezo.c module (and the depending
sound channel module sound.c) will be ommited. */
```

The file “config_gui.h”also has some switches, which need to be activated.

```
/*****************************************************************************/
/* customizing sound functions                                               */
/*****************************************************************************/

/* if you want to use sound devices and channels (e. g. for button or keyboard
click) define: */

#define _SOUND_WANTED


/* if the volume of your sound device can be adjusted define: */

#define _SOUND_VOLUME_ADJUST_WANTED
```

5.1.2 Driver functions for the loudspeaker

The sound hardware is controlled by the function `c_sound()` in the file “piezodriver.h/c”, which therfore must be adapted to your hardware.

void c_sound(uint16 frequency, unit16 duration, uint8 volume): Output of a sound with the frequency `frequency` and the duration `duration` in units of system ticks and the volume `volume`.

5.1.3 The use of the speaker

Analogously to the other devices like LCD and touch glass, a sound device is connected to a sound channel:

```
piezo_device  *piezo;               /* piezo loudspeaker    */
sound_channel loudspeaker;

piezo_init( &piezo);                /* init piezo and sound */
sound_open( &loudspeaker);
sound_connect( &loudspeaker, piezo);
```

Afterwards you can generate some noise with the command `sound_make()`.

```
sound_make( &loudspeaker, TONE_D*3, 14);
sound_make( &loudspeaker, TONE_FIS*3, 14);
sound_make( &loudspeaker, TONE_A*3, 20);
```

At last, the cleanup is done with the following functions:

```
sound_disconnect( &loudspeaker);
sound_close( &loudspeaker);
piezo_close();
```

In order to achieve a better usability, it is a good idea to play a short sound whenever the user presses a button or strikes a key. It is possible to generate *touch-clicks* in the following way. First you have to initialize the sound system like in the example above, now you replace the command `touch_open()`:

```
touch_open( &touch_glas, &display, &loudspeaker);
```

The complete example can be found on the CD.

5.2 How to continue

The Simplify Technologie GUI-Library gives you further options to design the user interface of your applications, for instance touch buttons and touch keyboards. You can find several detailed examples within the manual of the library and on the CD of the GUI–library, which demonstrate all these features.