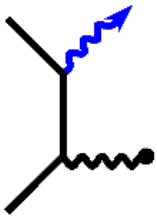
# Handbuch für die Simplify Technologies GUI-Bibliothek

Version 3.0.0



Simplify Technologies GmbH
Steinbühlstraße 15
35578 Wetzlar

$Rev: 6373 $

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

# Inhaltsverzeichnis

# 1 Einleitung

Grafikfähige LCD und Touch–Displays werden in *embedded sytems* zunehmend eingesetzt, um dem Benutzer eine ergonomische Bedienung zu ermöglichen.
Der Vorteil eines Touch–Glases liegt in der Möglichkeit, ergonomische, einfach zu bedienende Benutzeroberflächen zu schaffen, die bei Änderungen der Anwendung einfach durch Software angepaßt werden können, ohne daß aufwendige und kostspielige Hardwareänderungen nötig sind.

Die Simplify Technologies GUI–Bibliothek ist eine Sammlung von aufeinander abgestimmten Unterprogrammen und Funktionen für die Realisierung grafischer Benutzeroberflächen. Sie erlaubt Ihnen, die Vorteile grafikfähiger Displays zu nutzen, ohne daß Sie die aufwendigen Ansteuerungsroutinen selber realisieren müssen. Insbesondere bei der Verwendung von Touch–Displays bedeutet dies eine wesentliche Vereinfachung und Beschleunigung Ihrer Entwicklung.

Die Dokumentation zur Simplify Technologies GUI–Bibliothek besteht aus drei Teilen:

1. Das Ihnen hier vorliegende Handbuch beschreibt die Programmierung und die Funktionen der Simplify Technologies GUI–Bibliothek aus der Sicht des Anwendungsprogrammierers sowie die Anpassung der Bibliothek an Ihre Anforderungen und die Zielhardware.

2. Das “Quickstart Tutorial” soll Ihnen anhand von Beispielen den Einstieg in die Programmierung mit der Simplify Technologies GUI–Bibliothek erleichtern.

3. Die Beschreibung des Treibers für das LCD–Display bzw. den LCD–Controller geht auf Besonderheiten ein, die für den jeweiligen Treiber zu beachten sind.

In Kapitel 2 des Handbuches werden die grundlegenden Konzepte für die Programmierung der grafischen Benutzerschnittstelle (GUI: **g**raphical **u**ser **i**nterface) vorgestellt.
Kapitel 3 behandelt die Funktionsmodule der GUI–Bibliothek und die von ihnen zur Verfügung gestellten Programmiermöglichkeiten im einzelnen.
Kapitel 4 beschreibt zusätzlich Funktionsmodule, die die im vorigen Kapitel vorgestellten Funktionen intensiv nutzen, um die GUI–Bibliothek zu erweitern.
Fragen zur Anpassung der Bibliothek und zur Verwendung auf der entsprechenden Zielhardware werden in Kapitel 5 erläutert.
Kapitel 6 beschreibt den Einsatz von Farbe für farbfähige LCD–Displays mit der Farbversion der Bibliothek.

Ein ausführliches Programmierbeispiel zur Verdeutlichung und als Ausgangsmodell für Ihre eigenen Anwendungen wird in Kapitel 7 beschrieben. Es ergänzt die Beschreibungen der vorangehenden Kapitel und zeigt wichtige Details der Programmierung.
Kapitel 8 gibt Hinweise über Sicherheitsfragen, die beim Einsatz der Bibliothek zu beachten sind.
Die Funktionsreferenz befindet sich in einem separaten Dokument auf der CD.
In Kapitel 10 finden Sie Antworten auf häufig auftretende Fragen.

Sollte dieses Handbuch Fragen offen lassen, werden wir uns bemühen, diese zügig und direkt zu beantworten. Wir wünschen Ihnen viel Spaß und Erfolg bei der Entwicklung Ihrer Anwendung!

# 2 Konzept zur Programmierung der GUI–Bibliothek

Um möglichst hardwareunabhängige Anwendungen zu ermöglichen, werden die Ein- und Ausgabegeräte der Benutzerschnittstelle über ein Zwei–Schichten-Modell angesprochen. Für jede Schicht werden die jeweils geeigneten Treiberfunktionen zur Verfügung gestellt. Die hier verwendeten Begriffe sind *Channel* für geräteunabhängige Funktionen und *Device* für die Schicht geräteabhängiger Funktionen (siehe auch Abb. 1 auf Seite 7):

- Für den Anwendungsprogrammierer ist es in der Regel ausreichend, die hardwareunabhängigen Channels zu verwenden, z. B. den Display–Channel. Diese Channels sind die logischen Repräsentanten der physikalischen Geräte (und auch abstrakter, wie z. B. einer Tastatur, die sowohl physikalisch als auch nur auf einem Touch–Display vorhanden sein kann).

  Der Display–Channel ist so z. B. die abstrakte Repräsentation für grafische Ausgabegeräte wie ein LCD–Display oder Bildschirme.

  Die Channel–Treiber stellen hardwareunabhängig die Funktionen bereit, die für ein Ein- oder Ausgabegerät grundsätzlich zur Verfügung stehen (z. B. die Möglichkeit, auf einem Display–Channel eine Linie zu zeichnen). Soweit möglich sollten die Geräte also über ihre Channel–Treiber angesprochen werden, damit bei einer Änderung der Hardware keine Software–Änderungen notwendig sind.

- Damit die für einen Channel ausgeführten Funktionen Folgen auf realen Geräten haben, wird jeder Channel (eventuell nach Umlenkung über weitere Channels) mit einem Device, dem eigentlichen physikalischen Gerät, verbunden. Die Verbindungsfunktion zwischen Channel- und Device–Treiber realisiert die Umsetzung der hardwareunabhängigen Channel–Funktionen auf das jeweilige Device (z. B. auf ein LCD–Display). Spezielle Device–Funktionen sind dem Anwendungsprogrammierer zugänglich, um hardwareabhängige Funktionen, die nicht generalisiert über die Channel–Funktionen realisiert sind, bereitzustellen. Ein Beispiel hierfür ist die Beleuchtung eines LCD–Displays, die natürlich für eine Anwendung angesteuert werden muß, die aber so hardwarespezifisch ist, daß sie nicht in den allgemeineren Display–Channel aufgenommen ist.

  Im Programmtext enden die Namen der Strukturen, die physikalische Devices darstellen, auf `_device`, z.B. `t_lcd_device`. Sie bezeichnen einen gerätespezifischen Hardwaretreiber. Eine Aufzählung der vorgesehenen Devices befindet sich in der Datei “system.h” bei der Definition von *connection types*

(das sind mögliche Verbindungsarten für die Channels). Die Namen, die auf `_channel` enden, bezeichnen logische Channels.

Für die Funktion der Channels müssen ihre Verbindungspartner (meist Devices, manchmal auch weitere Channels) initialisiert und verfügbar sein, bevor die Verbindung hergestellt wird und Funktionen auf dem Channel angefordert werden. Das ist ziemlich unkompliziert. In den meisten Fällen ist es vor der Benutzung lediglich erforderlich, die naheliegende Verbindung herzustellen, dann kann der Channel benutzt werden.
**Beispiel:** Initialisieren und Verbinden des Display–Channels mit dem LCD–Device (weitere Informationen hierzu finden sich auch im Anwendungsbeispiel in Kapitel 7, in der Funktionsreferenz und in der Datei "lcd.h"):

1. Initialisieren des LCD–Devices, um dieses verfügbar zu machen:

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

   Das LCD–Display kann einen größeren virtuellen Bildschirm verwalten, der hier gewählt werden kann. Abhängig vom LCD–Treiber kann auch eine rotierte Orientierung des LCD–Displays realisiert werden. Der Name des LCD–Displays kann frei (hier als `system_lcd`) gewählt werden.

2. Öffnen des Display–Channels:

   ```
   disp_open( &new_display_pointer);
   ```

   `new_display_pointer` erhält dann einen Zeiger auf die Struktur, die den neuen Display–Channel charakterisiert.

3. Den Display–Channel zum *aktuellen* Display machen:

   ```
   disp_make_current( &new_display_pointer);
   ```

4. Verbinden des Display–Channels mit dem LCD_Device

   ```
   disp_connect( system_lcd);
   ```

Dieses Vorgehen ermöglicht eine recht hardwareunabhängige Programmierung, so daß mit späteren Erweiterungen und Modifikationen die wesentlichen Teile einer Anwendung ungeändert übernommen werden können. Außerdem bietet sich die Möglichkeit der Umlenkung von Ein- und Ausgabe, z. B. in Dateien oder über Schnittstellen auf einen PC.
Bemerkung: Die Devices (die hardwareabhängigen Peripheriegeräte) sind bereits als Struktur im System vorhanden. In der Anwendung werden sie daher über Zeiger angesprochen, die durch die Initialisierung den korrekten Wert zugewiesen bekommen. Channels werden direkt in der Anwendung alloziert; auch bei ihnen wird normalerweise nur ihre Adresse an Funktionen übergeben.

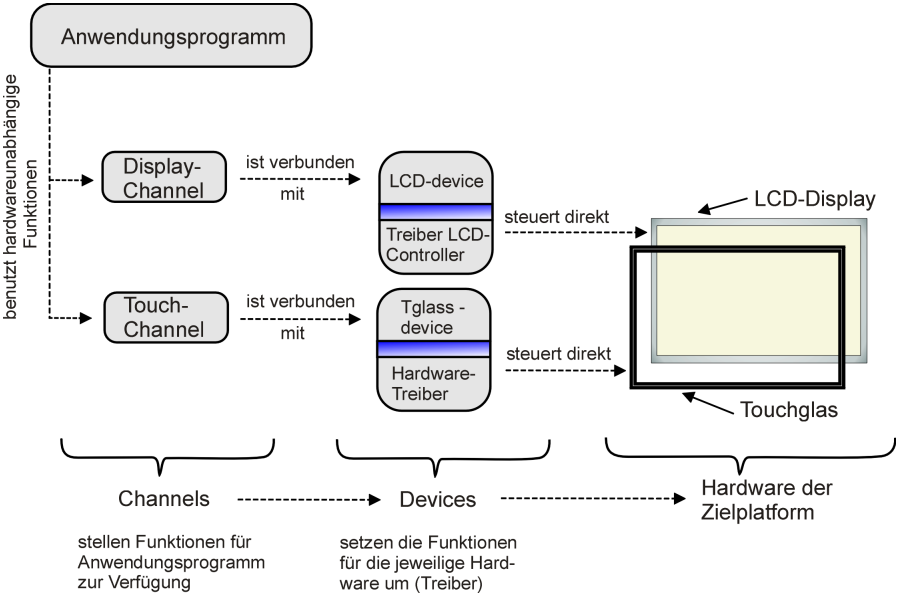



Abbildung 1: Konzept des Programmiermodells

Um weitgehend unabhängig von implementationsabhängigen Datentypen verschiedener Compiler zu sein, werden portable Datentypen in “portab.h” vereinbart und dann verwendet.

Als Zeichensätze werden Windows Pixel–Fonts (Version 2 und 3) verwendet. Dadurch ist es einfach, bei Bedarf andere als die drei vorhandenen Standardzeichensätze einzusetzen.

Außerdem können Fonts im SSF–Format verwendet werden. Dies ist ein binäres Format, welches speziell für die GUI–Bibliothek entwickelt wurde und Unicode–Zeichen unterstützt. SSF–Fonts enthalten nur die Zeichen, die wirklich von der Anwendung benötigt werden und sind deshalb speicherplatzsparend (näheres zu SSF–Fonts siehe 3.4 auf Seite 15).

Für die Bedienelemente der Touch–Version der Bibliothek, Buttons und Touch–Tastaturen, wird das Aussehen mit sogenannten *Styles* bestimmt. Diese werden verwendet, um die äußere Erscheinung der Bedienelemente zu beschreiben. Sie werden anschließend beim Erzeugen eines Bedienelementes übergeben, um so dessen Aussehen festzulegen. Diese externe Behandlung des Aussehens der Elemente hat folgende Vorteile:

- Man muß nicht für jeden Button etc. explizit alle Eigenschaften neu angeben, sondern kann auf einen einmal angelegten Stil verweisen.
- Beim Modifizieren des Designs genügt es, an einer Stelle den Stil zu ändern (den man z.B. als Standard–Stil verwendet) und alle Objekte, die diesen Stil verwenden, bekommen das neue Aussehen.

# 3 Module der GUI–Bibliothek und Programmierung

Hier werden die einzelnen Funktionsmodule der GUI–Bibliothek beschrieben, um eine allgemeine Übersicht über die Funktionsweise und Leistungsfähigkeit dieser Module sowie der verfügbaren Funktionen zu geben. Detaillierte Informationen über die einzelnen Funktionen und ihre Verwendung erhalten Sie in der Funktionsreferenz sowie durch das Programmierbeispiel in Kapitel 7 auf Seite 67.

## 3.1 Funktionsmodule und Header–Dateien

Für jeden Channel und für jedes Device existiert ein eigenes Funktionsmodul. Die für das Modul verfügbaren Funktionen und Definitionen werden in einer dazugehörigen Header–Datei deklariert (für den Compiler) und zusätzlich zum Handbuch beschrieben (diese Header–Dateien bieten ebenfalls Detailinformationen zu den einzelnen Funktionen).
Die einzelnen Module bestehen jeweils aus einer C–Datei mit den Funktionsimplementationen und zwei Header–Dateien. Die Header–Datei "<Modulname>.h" deklariert die Strukturen der einzelnen Channels und Devices, die Funktionen, die auch bei der Anwendungsprogrammierung benutzt werden, und die Definition von Symbolen für die Anwendungsprogrammierung.
Die Header–Datei mit Unterstrich "<Modulname>_.h" enthält die Deklarationen für interne Funktionen der Bibliothek (also solche, die bei der Anwendungsprogrammierung nicht direkt verwendet werden).
Die Funktionen der Device–Module greifen direkt auf die Treiber der jeweils verwendeten Hardware zu. Die Header–Dateien der Module für die Anwendungsprogrammierung sind im einzelnen:

**display.h:** Include–File mit den Funktionen und Strukturen der Display_Channels (hardwareunabhängig). Normalerweise verbunden mit einem LCD_Device.

**lcd.h:** Include–File für die spezifischen Funktionen des LCD–Displays (Device_type: `t_lcd_device`).

**touch.h:** Include–File für Funktionen des Touch–Channels. Diese ermöglichen Positionseingaben auf Displays, z.B. durch Touch–Glas oder Maus. Touch–Channels sind normalerweise mit dem Hardware–Touchglas der Zielhardware verbunden (also mit dem entsprechenden `t_tglass_device`). Touch–Channels weisen darüberhinaus noch eine Verbindung zu dem ihren Koordinaten zugeordneten Display–Channel auf (auf dem typischerweise die Folgen der Eingaben auf dem Touchglas dargestellt werden).

**tglass.h:** Include–File für die (hardwareabhängigen) Funktionen des verwendeten Touch–Glases (Device–Typ: `t_tglass_device`).

**button.h:** Funktionen zur Realisierung von *Buttons* (Bedienknöpfe für das Anwendungsprogramm).

**keyboard.h:** Include–File mit der Beschreibung der Keyboard–Channels (das sind allgemeine Tastaturen). Keyboard–Channels werden mit Tastatur–Devices, z.B. einem `t_keyb_device` verbunden.

**t_keyb.h:** Include–File für eine auf dem Touch–Glas realisierte und frei definierbare Touch–Tastatur (diese Tastaturen werden als Device (`t_keyb_device`) behandelt, da sie an die Geometrie des jeweiligen Touch–Displays gebunden sind.

**sound.h:** Include–File für die Funktionen des Sound–Channels. Dieser ist auf der Zielhardware meistens mit einem Piezo–Device verbunden.

**piezo.h:** Include–File für das Piezo–Device; das ist ein in der Zielhardware eingebauter Piezo–Lautsprecher. (Im von "piezo.c " aufgerufenen Hardwaretreiber kann natürlich auch ein anderer Schallerzeuger angesprochen werden.)

**system.h:** Include–File mit Definitionen für die Channels und Devices. Außerdem Definitionen der Zeichensätze und ihrer Funktionen sowie der Funktionen für den System–Timer.

Die Funktionsnamen für die Programmierung der GUI beginnen mit dem Namen des jeweiligen Moduls, z.B. mit `disp_...()` für Funktionen für Display–Channels. Beispiel: `disp_set_pixel()`.

Zusätzlich zu den Modulen sind noch folgende wichtige Dateien vorhanden:

**struktur.h:** Innerhalb der Bibliothek intern verwendete abstrakte Datenstrukturen (doppelt verkettete Listen und FIFO–Speicher).

**portab.h:** Definition portabler Datentypen.

**memory_wrapper.h:** Erlaubt zwischen der Standardspeicherverwaltung ("stdlib.h") und einer eigenen, einfachen Speicherverwaltung (siehe "simple_memory_wrapper.h") umzuschalten. Alle Dateien der GUI–Bibliothek benutzen die Speicherverwaltungsmakros aus dieser Header–Datei.

**simple_memory_wrapper.h:** Eigene dynamische Speicherverwaltung zum Ersetzen von `malloc()` und `free()`.

**errors.h:** Definition der Fehlercodes für die Bibliotheksfunktionen.

**text_gui.h:** Datei zur sprachabhängigen Definition von Zeichenketten. Mit einem passenden Define kann so zwischen Texten verschiedener Sprachen ausgewählt werden, ohne überall den Quelltext modifizieren zu müssen.

**config_gui.h:** Konfigurationsdatei zur Anpassung der GUI–Bibliothek.

**config_hardware_gui.h:** Hier werden Variablen definiert, durch die die Hardwareumgebung für das User–Interface des Systems beschrieben wird (siehe auch Kapitel 5 auf Seite 38).

## 3.2 Eigenschaften von Funktionen und Variablen

Um möglichst hardwareunabhängig programmieren zu können, wurden die verschiedenen implementationsabhängigen Integer–Typen von ANSI–C nicht verwendet. An ihre Stelle treten die in der Datei "portab.h" definierten Typen, wie z.B. `int16`, `uint8` (die erste ist eine vorzeichenbehaftete Integer–Typ mit einer Länge von 16 Bit, die zweite ein Integer ohne Vorzeichen mit 8 Bit). In der Datei "portab.h" sind außerdem weitere Typen von Variablen vereinbart, wie z.B. `t_color`, `t_font` und `err_code`. Je nach Farbtiefe des Displays ist der Typ `t_color` verschieden groß (1 Byte bei monochrom und 8 Bit Farbe, 4 Bytes bei 24 Bit Farbe).
Der Typ `err_code` bezeichnet den Rückgabetyp der Funktionen. Alle Funktionen, bei denen Fehler auftreten können, geben einen Error–Code vom Typ `err_code` zurück, an dem festgestellt werden kann, ob die Funktion erfolgreich durchgeführt werden konnte oder welche Art von Fehler vorliegt. Die Definitionen der Error–Codes erfolgt in der Datei "errors.h". Insbesondere bedeutet der Rückgabewert `ERR_OK`, daß kein Fehler aufgetreten ist. Bei Funktionen, die Werte zurückgeben sollen, wird dies über Adreßzeiger in der Parameterliste der jeweiligen Funktion realisiert (dies wird in der Funktionsreferenz an Beispielen bei den entsprechenden Funktionen verdeutlicht). Bei Funktionen, bei denen kein Fehler auftreten kann, wird ein Resultat direkt zurückgegeben.

## 3.3 System–Funktionen

Die System–Funktionen erledigen Aufgaben, die für das ganze System zentral abgewickelt werden. Dies sind Funktionen, die die Systemzeit betreffen sowie die Verwaltung der Schriftarten (*Fonts*).
Der Systemtimer (sofern vorhanden) muß zu Beginn aktiviert werden. Dies und auch die Initialisierung der Liste der verfügbaren Fonts geschieht mit der Funktion `system_init()`. Der Systemtimer besteht im regelmäßigen Aufruf der internen Bibliotheksfunktion `system_process_timed_functions()`. Dies geschieht am besten über einen Timer–Interrupt. Details hierzu finden Sie in Kapitel 5 auf Seite 38.

Die interne Funktion `system_process_timed_functions()` sorgt für die regelmäßige Abarbeitung verschiedener Routinen, so z.B. für die regelmäßige Abfrage des Touch–Glases. Außerdem wird die Zeit seit dem Einschalten des Gerätes gezählt. Diese Systemzeit ist auch für die Anwendungsprogrammierung verfügbar, auch um z.B. definierte Verzögerungen zu ermöglichen.
Die Systemzeit wird durch zwei 32–Bit–Zähler definiert. Der erste zählt die sogenannten *ticks* (z.B. 1/100 s) der ersten 24 Stunden seit dem Einschalten, der zweite enthält die Vielfachen ganzer Tage (*days*) seit dem Einschalten.

**Übersicht System–Funktionen:**

**system_init:** Initialisiert bestimmte Variablen des Systems. Startet den Systemtimer, initialisiert die Font–Liste und die dynamische Speicherverwaltung (sofern anwendbar).
**system_font_init:** Initialisiert die Daten eines Fonts.
**system_font_close:** Schließt den Font und gibt den verwendeten Speicher frei.
**system_get_number_of_fonts:** Liest die Anzahl der im System momentan verfügbaren Zeichensätze.
**system_get_font_pointer:** Holt den Zeiger auf einen mit Namen spezifizierten Font.
**system_get_font_ascent:** Holt die Anzahl der Pixel vom oberen Ende des Fonts bis zu seiner ”Baseline”.
**system_get_text_ascent:** Holt die Anzahl der Pixel vom oberen Ende des Texts bis zu seiner ”Baseline”. Hierbei wird der benutzte Font und der Textstil berücksichtigt.
**system_get_font_pixelheight:** Holt die Gesamthöhe des Fonts (in Pixel).
**system_get_text_pixelheight:** Holt die Gesamthöhe des Fonts (in Pixel). Hierbei wird der benutzte Font und der Textstil berücksichtigt.
**system_get_font_avg_width:** Holt die durchschnittliche Breite der Zeichen des Fonts.
**system_get_font_max_width:** Holt die Breite des breitesten Zeichens eines Fonts.
**system_get_font_name:** Setzt den Zeiger auf den String mit dem Namen eines Fonts.
**system_get_charwidth:** Holt die Breite eines bestimmten Zeichens eines Fonts (in Pixel).
**system_get_textwidth:** Holt die Breite eines Textstrings in einem bestimmten Font (in Pixel).
**system_wait_ticks:** Wartet einen bestimmten Zeitraum mit der Weiterausführung des Programms.
**system_wait_until:** Wartet bis zu einem bestimmten Zeitpunkt mit der Weiterausführung des Programms.
**system_get_ticks:** Liest den Wert des Systemtimers aus.
**system_interval_elapsed:** Testet, ob eine vorgegebene Anzahl von Ticks seit einer bestimmten Zeit schon vergangen sind.

## 3.4 Display–Channel

Die Funktionen für die Ansteuerung von Displays sind unabhängig davon, welches Gerät (Device) gerade mit dem Display (–Channel) verbunden ist. Alle Pixel–Koordinaten, die in den Funktionen verwendet werden, sind die dieses *virtuellen* Ausgabegerätes (nämlich des Display–Channels).
Die Koordinaten werden von der linken unteren Ecke an gezählt, d.h. die linke untere Ecke hat die Koordinaten x=0, y=0.
Bitte beachten Sie, daß aus Gründen der Ausführungsgeschwindigkeit auf ein *Clipping* verzichtet wird. Das bedeutet, daß Grafikbefehle, die über den Rand des vereinbarten virtuellen Displays herausführen würden, abhängig von der Funktion nicht in jedem Fall korrekt bis zum Rand, sondern eventuell gar nicht oder mit Artefakten ausgeführt werden.

Viele Funktionen werden relativ zum aktuellen *Grafikcursor* ausgeführt. Dieser bezeichnet kein auf dem Display erscheinendes Cursor–Symbol, sondern eine *aktuelle Position* zum Zeichnen auf dem Display. Verschiedene Funktionen verändern den aktuellen Grafikcursor (d.h. also die Position, zu der die folgenden Funktionen relativ arbeiten). Durch diese Arbeitsweise relativ zu einer aktuellen Position kann leicht bezüglich einer vorher eingestellten Startposition gearbeitet werden, so daß, wenn das erwünscht ist, schnell eine Veränderung der Position der gesamten Ausgabe auf dem Display erfolgen kann, indem nur die Startposition geändert wird.
Die Funktionen wirken jeweils auf dem *aktuellen* Display (es ist möglich, mehrere Display–Channels anzulegen, einer davon muß als der aktuelle benannt werden).
Die Wirkung einer Grafik–Funktion ist außerdem vom *Zeichenmodus* abhängig. Dieser bestimmt, wie das zu zeichnende Element mit dem bereits vorhandenen Hintergrund kombiniert wird. Neben dem Standardmodus `SET`, bei dem der Hintergrund einfach ersetzt wird, sind die logischen Verknüpfungen `OR`, `EXOR` und `AND` verfügbar.

Die Funktionen für Display–Channel können in folgende Kategorien unterteilt werden:

**Funktionen zur Verwaltung von Displays:** Um Display–Channels und ihre Funktionen zu benutzen, müssen diese zuerst geöffnet und mit einem physikalischen Ausgabegerät verbunden werden.

**Funktionen für einfache grafische Befehle:** Die meisten Funktionen werden relativ zur aktuellen Cursorposition ausgeführt. Das Setzen des Cursors (einer gedachten aktuellen Position auf dem Display) selbst kann absolut oder relativ zu der vorherigen Position geschehen. Die elementaren Funktionen für das Setzen und Lesen von Pixeln sind ebenfalls sowohl mit Absolutkoordinaten als auch relativ zur Cursorposition vorhanden. Durch diesen Ansatz

können grafische Strukturen durch einfaches Repositionieren des Cursors an einer anderen Stelle angezeigt werden, ohne nochmals zahlreiche Koordinaten berechnen zu müssen.

Funktionen für das Zeichnen von Linien ermöglichen, unterschiedliche Liniendicken und Linienstile (z.B. durchgezogen oder gestrichelt) zu verwenden. Für die Liniendicke gilt folgendes: Die Linien werden um die ursprünglich verwendete Ausgangskoordinate verbreitert, wenn die Liniendicke größer als 1 gewählt wird. Bei geraden Werten für die Liniendicke wird zunächst in Richtung positiver x– bzw. y–Koordinaten verbreitert. Beispiel: Für eine horizontale Linie mit Liniendicke 2 wird die Linie auf der ursprünglichen y–Koordinate sowie auf der Koordinate y+1 (oder x+1 für Linien mit Steigung größer als 1) ausgeführt. Erhöht man die Liniendicke auf 3, so erfolgt das Zeichnen der Linie nun auf den Koordinaten y-1, y, y+1 (bzw. bei x-1, x, x+1, wenn die Steigung größer 1 ist). Mit Liniendicke = 4 kommt zusätzlich y+2 (oder x+2) hinzu, mit Liniendicke = 5 wird auch bei y-2 (oder x-2) gezeichnet, usw.

Der Linienstil (`t_linestyle`) wird durch einen 16–Bit–Wert definiert. In diesem repräsentiert ein gesetztes Bit einen (in der aktuellen Farbe `current_color`) gesetzten Punkt in der Linie und ein gelöschtes Bit keinen gesetzten Punkt (dort wird die aktuelle Hintergrundfarbe gesetzt). Das Linienmuster beginnt mit dem höchstwertigen Bit und wiederholt sich, sofern das zu zeichnende Linienelement länger als 16 Pixel ist.

**Funktionen für die Bearbeitung von rechteckigen Bereichen:** Um Flächen auf dem Display zu bearbeiten, stehen Füllstile und rechteckige Blöcke zur Verfügung, die ein– und ausgelesen werden können. Außerdem können Bilder im verbreiteten BMP–Format dargestellt werden. Füllstile werden verwendet, um rechteckige Flächen auszufüllen. *Pixelblöcke* sind Bereiche im RAM, die die Bildinformation rechteckiger Bereiche des Displays enthalten. Die Pixel dieser Bereiche können vom Display in den Speicher ausgelesen oder vom Speicher ins Display geschrieben werden. Dadurch lassen sich beliebige rechteckige Displaybereiche hin und her kopieren.

Ein Pixelblock wird über einen Satz Parameter definiert. Dieser beinhaltet im wesentlichen die horizontale und vertikale Ausdehnung des Pixelblocks sowie die vorliegende Farbtiefe. Diese Parameter können mit den Funktionen `disp_set_par_pixelblock()` bzw. `disp_get_par_pixelblock()` gesetzt bzw. ausgelesen werden. Anschließend können Displaydaten in den Pixelblock geholt (`disp_fetch_pixelblock()`) bzw. auf das Display herausgeschrieben (`disp_output_pixelblock()`) werden.

Es stehen Funktionen zum Zeichenen von Rahmen zur Verfügung, wie sie bei Buttons oder anderen Touch–Objekten (vergleiche Schieberegler und Menüs

in Kapitel 4 auf Seite 27) benutzt werden. Die Rahmen können abgerundete Ecken haben und schattiert sein. Hierbei wird das Aussehen der Rahmen wiederum über einen *Style* definiert.

Weitere Details der Füllstile, Pixelblöcke und Rahmen werden bei den jeweiligen Funktionen in der Funktionsreferenz beschrieben.

**Funktionen für die Textausgabe:** Text läßt sich mit verschiedenen (auch proportionalen) Zeichensätzen darstellen. Er kann außerdem mit üblichen Attributen wie “unterstrichen” , “fett” , “kursiv” und “hell” versehen werden. Für Farbdisplays ab einer Farbtiefe von 8 Bit kann der Text kann auch mit Antialiasing ausgegeben werden, um ein glatteres Aussehen zu erhalten. Dabei wird der Text in der halben Größe des verwendeten Zeichensatzes dargestellt. Bei 24-Bit-Farbtiefe wird Text mit dem Attribut “hell” durch eine geeignete Farbe dargestellt und nicht mehr gerastert. Die vertikale Ausgabe von Text funktioniert nur mit den Attributen “hell” und “invertiert” .

(Zu den Attributen siehe auch die Beschreibung der Funktion `disp_set_textstyle()`) in der Funktionsreferenz.

Die Textfunktionen verwenden für ASCII–Texte Fonts im Windows 2.0 und 3.0 Format (Dateiendung FNT), so daß es leicht möglich ist, weitere Fonts in das System zu übernehmen. Die Fonts selbst sind nicht Teil des jeweiligen Display–Channels, sondern werden zentral vom System verwaltet.

Die Darstellung von UTF8 kodierten Texten (z.B. für asiatische Schriftarten) wird auf die folgende Art und Weise realisiert: Es gibt ein Windows–Tool (*SimplifyCharacterCompiler*) welches einen sogenannten *SSF–Font* erzeugt, der beliebig viele Zeichen enthalten kann und nicht auf 256 beschränkt ist wie das FNT–Format. Hierzu gibt man diesem Tool den gewünschten Truetype–Font vor und einen Text, der alle benötigten Zeichen mindestens einmal in der UTF8–Kodierung enthält. Intern werden daraus mehrere FNT–Fonts mit jeweils 256 Zeichen erstellt und eine Übersetzungstabelle die jedem Unicode–Zeichen eine FNT–Font–Nummer und eine Zeichennummer zuordnet. Die FNT–Fonts und die Tabelle werden zu einem SSF–Font zusammengefügt. Dieser SSF–Font kann dann als Binär– oder h–File für die GUI–Bibliothek abgespeichert werden. Die Font–Befehle (aus “system.h/c” bzw. “display.h/c” ) der GUI–Bibliothek arbeiten mit den SSF–Fonts sowie den herkömmlichen FNT–Fonts zusammen. Mit dem Befehl `disp_text()` kann man dann UTF8 kodierte Strings ausgeben und die entsprechenden Zeichen erscheinen auf dem Display, vorausgesetzt der mit `disp_set_font()` eingestellte SSF–Fonts enthält diese Zeichen dann auch.

**Wechselwirkung mit Funktionen des Touch–Channels:** Zu beachten ist, daß mit den Befehlen des Display–Channels auch die Darstellung von Buttons

und Tastaturen auf dem Display unzulässig verändert werden kann (z.B. werden durch einen Aufruf von `disp_clear()`, solange noch Buttons oder Tastaturen aktiv sind, diese zwar auf dem Display verschwinden, die Touch–Funktionalität bleibt jedoch erhalten). Entfernen Sie daher in solchen Fällen vorher die entsprechenden Touch–Elemente vom Display.

**Übersicht Display–Funktionen:**

**disp_open:** Holt den Zeiger auf den neu zu erzeugenden Display-Channel (wenn erfolgreich, sonst NULL). Für das Display werden Standardeinstellungen gemacht .
**disp_connect:** Verbindet das aktuelle Display mit einem anderen Kanal oder physikalischen Ausgabegerät.
**disp_disconnect:** Trennt das aktuelle Display vom Channel oder Device, mit dem es vorher verbunden war.
**disp_make_current:** Macht den angegebenen Display–Channel zum aktuellen Display–Channel ("current_display"), auf dem dann die weiteren Display-Funktionen ausgeführt werden.
**disp_get_current:** Holt den Zeiger auf den im Moment aktuellen Display–Channel. (Funktion wird also nur benötigt, wenn man mehrere Display–Channels benutzt, und abfragen möchte, welcher der aktuelle ist).
**disp_close:** Schließt und deaktiviert das aktuelle Display.
**disp_update:** Aktualisiert das aktuelle Display ("current_display"). Sinnvoll nur bei gepufferten Displays (wie Drucker oder Displays, bei denen die Grafikausgabe zuerst in einem Speicherbereich zwischengespeichert wird).
**disp_save_properties:** Speichert die wichtigsten Einstellungen des aktuellen Displays in eine entsprechende Datenstruktur.
**disp_restore_properties:** Setzt die Display–Eigenschaften entsprechend einer Struktur vom Typ "display_properties".
**disp_reserve_bmp_colors:** Reserviert eine bestimmte Anzahl an Farben aus der Farbpalette für die Benutzung von BMP-Bildern.
**disp_set_palette_entry:** Erlaubt das Setzen eines Eintrags in der Farbpalette.
**disp_clear:** Löscht das Display mit der aktuellen Hintergrundfarbe.
**disp_set_cursor_position:** Setzt die Cursorposition in absoluten Koordinaten.
**disp_get_cursor_position:** Liest die aktuelle Cursorposition.
**disp_move_cursor:** Bewegt Cursorposition relativ zur aktuellen Cursorposition.
**disp_set_pixel:** Setzt ein Pixel in der aktuellen Farbe auf der aktuellen Cursor-Position.
**disp_set_pixel_abs:** Setzt ein Pixel in der aktuellen Farbe an der angegebenen absoluten Position.
**disp_get_pixel:** Liest die Farbe des Pixels an der aktuellen Cursorposition.
**disp_get_pixel_abs:** Liest die Farbe des Pixels an einer absoluten Position. Cursorposition wird dorthin gesetzt.

**disp_line:** Zeichnet eine Linie mit den momentan gültigen Linienattributen (Linienmuster und Dicke) von und relativ zur aktuellen Cursorposition.

**disp_line_abs:** Zeichnet eine Linie mit den momentan gültigen Linienattributen (Linienmuster und Dicke) von einer absolut angegebenen Position zu einer anderen absolut angegebenen Position.

**disp_hline:** Zeichnet eine horizontale Linie mit den momentan gültigen Linienattributen (Linienmuster und Dicke) von und relativ zur aktuellen Cursorposition.

**disp_vline:** Zeichnet eine vertikale Linie mit den momentan gültigen Linienattributen (Linienmuster und Dicke) von und relativ zur aktuellen Cursorposition.

**disp_rectangle:** Zeichnet ein Rechteck relativ zur aktuellen Cursorposition (mit allen Linienattributen). Die aktuelle Cursorposition wird nicht verändert.

**disp_colored_rectangle:** Zeichnet ein mit der aktuellen Farbe gefülltes Rechteck relativ zur aktuellen Cursorposition.

**disp_filled_rectangle:** Zeichnet ein mit einem Muster ausgefülltes Rechteck relativ zur aktuellen Cursorposition (ohne Rahmen).

**disp_colored_triangle:** Zeichnet ein mit der aktuellen Farbe gefülltes Dreieck.

**disp_circle:** Zeichnet einen Kreis mit der aktuellen Cursorposition als Kreismittelpunkt.

**disp_colored_circle:** Zeichnet einen mit der aktuellen Farbe ausgefüllten Kreis relativ zur aktuellen Cursorposition.

**disp_arc:** Zeichnet ein Kreissegment mit der aktuellen Cursorposition als Kreismittelpunkt zwischen einem Startwinkel und einem Endwinkel.

**disp_get_color:** Liest die aktuelle Farbe, die im aktuellen Display-Channel verwendet wird.

**disp_set_color:** Setzt die Farbe, die im aktuellen Display–Channel verwendet werden soll.

**disp_get_background_color:** Liest die Hintergrundfarbe zum Zeichnen auf dem aktuellen Display–Channel.

**disp_set_background_color:** Setzt die Hintergrundfarbe zum Zeichnen auf dem aktuellen Display-Channel.

**disp_get_linestyle:** Liest den aktuell verwendeten Linien–Stil des aktuellen Display–Channels.

**disp_set_linestyle:** Setzt den neuen Linienstil (dies ist ein 16-Bit-Wort für das für Linien–Funktionen verwendete Linien-Muster) des aktuellen Display–Channels.

**disp_get_linewidth:** Liest die aktuell verwendete Liniendicke des aktuellen Display–Channels.

**disp_set_linewidth:** Setzt die neue Liniendicke des aktuellen Display–Channels.

**disp_get_drawmode:** Liest den aktuell verwendeten Zeichen–Modus des aktuellen Display–Channels.

**disp_set_drawmode:** Setzt einen neuen Zeichen–Modus des aktuellen Displays.

**disp_get_fillstyle:** Liest das aktuell verwendete Füllmuster (“fillstyle”) des aktuellen Display–Channels.

**disp_set_fillstyle:** Setzt einen neuen Füllstil (fillstyle) des aktuellen Display–Channels.

**disp_bmp:** Gibt ein Bilde im BMP-Format auf dem aktuellen Display aus.

**disp_bmp_alpha:** Gibt ein Bilde im BMP-Format auf dem aktuellen Display aus, wobei eine Farbe als transparent deklariert wird.

**disp_pixelblock_init:** Initialisiert eine Pixelblock-Struktur und holt den entsprechenden Speicher.

**disp_pixelblock_free:** Gibt den Speicher eines Pixelblocks wieder frei.

**disp_pixelblock_get:** Kopiert einen Pixelblock aus dem Display an der aktuellen Cursorposition (die dabei nicht geändert wird) in den Speicher.

**disp_pixelblock_set:** Gibt einen Pixelblock auf dem Display an der aktuellen Cursorposition (die dabei nicht geändert wird) aus.

**disp_invert_area:** Invertiert ein rechteckiges Gebiet auf dem Display (relativ zur aktuellen Cursorposition, die dabei nicht geändert wird).

**disp_color_change_area:** Ändert eine Farbe innerhalb eines rechteckigen Bereichs.

**disp_framestyle_init:** Holt Speicher für einen Rahmenstil und initialisiert diesen.

**disp_copy_area:** Befehl zum kopieren rechteckiger Bereiche auf dem Display.

**disp_framestyle_clone:** Kopiert einen Rahmenstil und holt Speicher für die Kopie.

**disp_framestyle_set_colors:** Setzt die Farbe für den Rahmenstil.

**disp_framestyle_set_dimensions:** Setzt die Rahmenbreite und den Radius für abgerundete Ecken.

**disp_framestyle_set_linestyle:** Setzt den Linienstil eines Rahmenstils.

**disp_framestyle_set_flags:** Setzt die Konfigurations–Flags eines Rahmenstils.

**disp_framestyle_free:** Gibt den Speicher eines Rahmenstils wieder frei.

**disp_frame:** Zeichnet einen Rahmen z.B. für Buttons oder andere Touch–Objekte.

**disp_delete_frame:** Überschreibt den Rahmen mit der aktuellen Farbe.

**disp_printf:** Stellt die Funktionalität der in C bekannten Bibliotheksfunktion printf() zur Verfügung.

**disp_text:** Schreibt einen Text an die aktuelle Cursor-Position.

**disp_formatted_text:** Wie disp_text(), jedoch darf der String das *newline*–Zeichen (Hex 0A) enthalten. An diesen Stellen wird dann umgebrochen.

**disp_get_textstyle:** Liest die auf dem aktuellen Display momentan gültigen Text-Attribute.

**disp_set_textstyle:** Setzt neue Text-Attribute für das aktuelle Display.

**disp_get_font:** Liest den Zeiger auf den Zeichensatz für die Textausgabe auf dem aktuellen Display.

**disp_set_font:** Setzt einen neuen Zeichensatz für die Textausgabe auf dem aktuellen Display.

## 3.5 LCD–Device

Das Modul *LCD–Device* stellt Funktionen für das in der Zielhardware eingebaute LCD–Display zur Verfügung. Eine Besonderheit vieler LCD–Display bzw. LCD–Controller ist, daß der Grafikspeicher für eine wesentlich größere Anzeigenfläche ausreicht, als direkt auf dem LCD sichtbar ist. Dies ermöglicht, auf einer wesentlich größeren Anzeigenfläche zu arbeiten (*virtuelles Display*), von der jeweils ein bestimmter Ausschnitt (*Viewport*) auf dem LCD angezeigt wird. Dies kann besonders für grafische Darstellungen von Karten oder komplexen Objekten sehr vorteilhaft sein. Eine Funktion zum *Scrollen* des dargestellten Bereiches steht zur Verfügung.
Wenn das LCD-Display mit einem Display–Channel verbunden ist, realisiert es automatisch alle Funktionen des Display–Channels, so daß eine weitgehend hardwareunabhängige Grafikprogrammierung möglich ist.
Daneben hat das LCD–Display noch einige Sonderfunktionen, die nicht für alle Arten von Displays anwendbar sind, wie z.B. eine optionale Beleuchtung und die Einstellung des Kontrastes. Zusätzlich läßt sich das LCD–Display softwaregesteuert ein– und ausschalten, um Strom zu sparen, wenn es nicht benötigt wird.

**Übersicht LCD–Device–Funktionen:**

**lcd_init:** Initialisiert das LCD–Display und gibt den Zeiger auf die interne LCD–Device–Struktur zurück.
**lcd_close:** Schließt das LCD-Display und ruft lcd_sleep() auf.
**lcd_get_status:** Liest den Status des LCD–Displays.
**lcd_get_size:** Liest die Größe des virtuellen Bildschirms des LCD–Displays.
**lcd_get_color_depth:** Liest die Farbtiefe (in Bit) des LCDs.
**lcd_set_viewport:** Setzt den sichtbaren Bereich des LCD-Displays auf eine bestimmte Position auf dem virtuellen Bildschirm (von z. B. 600×800 Pixeln).
**lcd_orientation:** Dreht das Display in 90 Grad Schritten.
**lcd_light:** Steuert die Beleuchtung des LCD.
**lcd_get_light:** Holt die aktuelle Einstellung der Helligkeit der LCD–Hintergrundbeleuchtung.
**lcd_contrast:** Stellt den Kontrastwert des LCDs ein.
**lcd_get_contrast:** Gibt den aktuell für das LCD eingestellten Kontrastwert zurück.
**lcd_sleep:** Schaltet das LCD–Display aus, um Energie zu sparen.
**lcd_wakeup:** Schaltet das LCD–Display ein.

## 3.6 Touch–Channel

Der *Touch–Channel* dient als hardwareunabhängiger Eingabekanal für positionsabhängige berührungsempfindliche Eingabegeräte, wie z.B. Computermäuse, mit denen durch *Klicken* Objekte auf einer bestimmten Position angewählt werden, oder

einem Touch–Glas. Der Touch–Channel erlaubt die Abfrage der Positionen und Systemzeiten dieser Berührungen (sowohl die Werte für den Beginn des Kontakts als auch die für das Ende des Kontakts). Damit lassen sich vielfältige Funktionen wie *Doppelklick* oder *Ziehen von Objekten* softwaremäßig realisieren.

**Übersicht Touch–Channel–Funktionen:**

**touch_open:** Initialisiert den Touch-Channel.
**touch_close:** Schließt den Touch–Channel.
**touch_connect:** Verbindet den Touch–Channel mit dem Eingabegerät (z.B. Touchglas).
**touch_disconnect:** Trennt den Touch–Channel von seinem Eingabegerät (z.B. Touch–Glas).
**touch_update:** Aktualisiert den Touch–Channel (evt. vorhandene Puffer werden geschrieben).
**touch_get_contact_end:** Liest die Endkoordinaten des letzten Kontaktes auf dem Touch-Channel.
**touch_get_contact_start:** Liest die Anfangskoordinaten des letzten Kontaktes auf dem Touch-Channel.
**touch_process_objects:** Testet alle Touch–Objekte mit der aktuellen Touch–Position und dem entsprechenden Zeitstempel.

## 3.7 Buttons

Ein oft benötigtes Bedienungselement auf grafischen Benutzeroberflächen sind *Buttons*, also umrandete Schaltflächen, die mit einer Beschriftung oder einer Grafik versehen sind und die wie Schalter benutzt werden.
Es stehen verschiedene Varianten von Buttons zur Verfügung. Durch entsprechende Wahl der Parameter lassen sich unterschiedliche Größen, Umrandungen und Beschriftungen (Text oder Bilder) auswählen. Das äußere Aussehen der Buttons wird über einen Stil definiert. Außerdem lassen sich die Buttons mit unterschiedlichem Schaltverhalten ausstatten: Ein Button kann wie ein elektrischer Taster permanent aktiviert sein, solange er gedrückt wird (`FAST_BUTTON`). Er kann nur während des Loslassens aktiviert sein (`STANDARD_BUTTON`). Oder er kann zwei Zustände haben (gedrückt, nicht gedrückt), zwischen denen durch Berühren hin- und hergeschaltet wird (`HOLD_BUTTON`). In den beiden letzten Fällen wird der Schaltvorgang nicht bereits durch die Berührung ausgelöst, sondern durch das Loslassen über der Buttonfläche. Der Bediener des Gerätes kann so nach einer versehentlichen Berührung eines Buttons seine Auslösung verhindern, indem er den Kontakt mit dem Touch–Channel erst außerhalb der aktiven Button–Schaltfläche löst.
Bitte beachten Sie bei der Verwendung von Buttons folgendes:

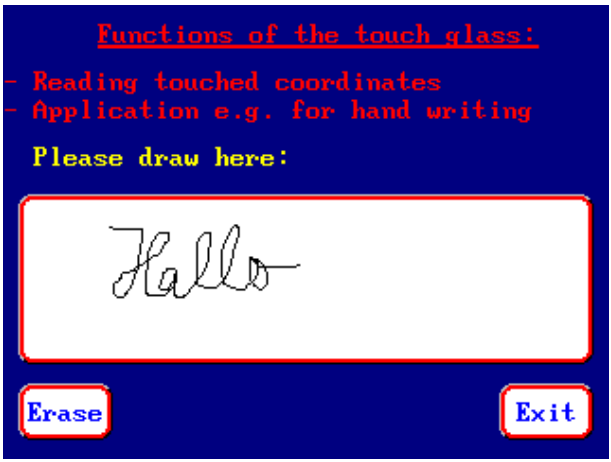



Abbildung 2: Beispiel für einfache Touch-Funktionen

1. Bei der Darstellung von Buttons wird zwischen dem gedrückten und dem nicht gedrückten Zustand unterschieden. Beide können ein völlig unterschiedliches Aussehen in Bezug auf die Art des Rahmens oder die Farben haben.

2. Die Größe des Buttons muß ausreichen, um den Rahmen und den Button–Text oder –Bild aufnehmen zu können. Vergrößert man die Rahmendicke so *wächst* dieser nach Innen, so daß u.U. kein Platz mehr für den Button–Inhalt bleibt.

3. Buttons können auch durch Benutzung des Textstils `TXT_VERTICAL` vertikal beschriftet werden.

**Übersicht Button–Funktionen:**

**button_style_init:** Holt Speicher für einen Button–Stil und initialisiert diesen. Button–Stile werden zur Definition des Aussehens von Buttons eingesetzt. So kann das Aussehen (der Stil) vieler Buttons an einer Stelle definiert werden. Diese Funktion muß aufgerufen werden, bevor der neue Stil mit weiteren Funktionsaufrufen eingerichtet wird.
**button_style_clone:** Kopiert einen Button–Stil und holt Speicher für den neuen Stil.
**button_style_free:** Gibt den Speicher, der von einem Button–Stil belegt wird, wieder frei.
**button_style_set_font:** Setzt die Schriftart eines Button–Stils.
**button_style_set_linefeed:** Setzt die Zeilenhöhe für den Button–Stil.

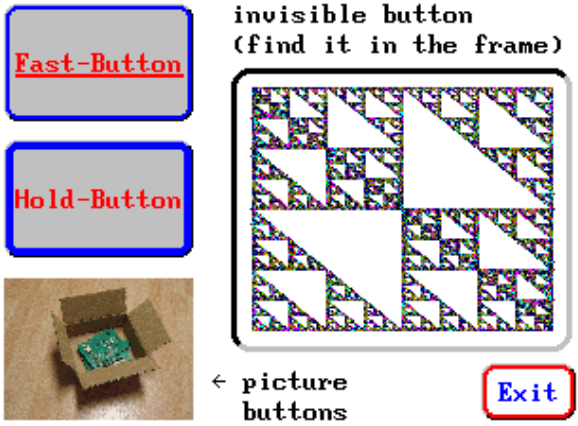



Abbildung 3: Beispiel für verschiedene Buttons

**button_style_set_alpha:** Setzt die Farbe des Alphakanals für Button-Bilder mit Transparents.
**button_style_set_framestyle:** Setzt den Rahmenstil eines Button–Stils.
**button_make_combined_label:** Erzeugt eine Button-Beschriftung, die sich aus einem Hintergrundbild und einem Text zusammensetzt.
**button_free_combined_label:** Gibt den Speicher einer kombinierten Button-Beschriftung wieder frei.
**button_define:** Definiert einen neuen Button.
**button_undefine:** Löscht einen Button und gibt den Speicher wieder frei.
**button_press_hold_button:** Versetzt einen Hold–Button in den gedrückten Zustand.
**button_lift_hold_button:** Versetzt einen Hold–Button in den nicht gedrückten Zustand.
**button_ghost_button:** Deaktiviert einen Button auf dem Touch–Channel und stellt die Beschriftung ”geghostet” dar.
**button_unghost_button:** Aktiviert einen Button auf dem Touch–Channel wieder, der vorher mit button_ghost_button() deaktiviert wurde. Der Text des Buttons wird normal dargestellt. Der Button wird auf dem Touch–Channel dargestellt und kann angewählt werden.
**button_activate:** Aktiviert einen bereits definierten Button auf einem Touch–Channel und stellt ihn auf dem zugehörigen Display–Channel dar.
**button_deactivate:** Deaktiviert einen Button auf einem Touch-Channel und löscht ihn vom zugehörigen Display-Channel.
**button_get_status:** Liest den Status eines Buttons.

**button_get_hold_button_status:** Holt den Status eines Hold-Buttons.
**button_get_position:** Liest die Position eines Buttons.
**button_set_position:** Setzt die Position eines Buttons.
**button_get_label:** Setzt Zeiger auf die beiden Beschriftungen eines Buttons.
**button_change_label:** Ändert die Beschriftung (Text oder Grafik) eines Buttons.
**button_set_style:** Setzt den Stil / das Aussehen eines Buttons auf die Eigenschaften eines angegebenen Button–Stils.

## 3.8 Tglass–Device

Das Modul Tglass–Device stellt Funktionen zur Verfügung, die spezifisch für das in der Zielhardware eingebaute Touch–Glas anwendbar sind. Für die Kalibrierung eines Touch–Glases steht die Funktion `tglass_calibrationsequence()` zur Verfügung. Eventuell kann nach einmaliger Kalibrierung und Bestimmung der Parameter des Touch–Glases auf weitere Kalibrierungen verzichtet werden. Der Ausgabetext für die Kalibrierfunktion ist in der Datei “text_gui.h ” für die Sprachen Deutsch und Englisch enthalten. Weitere Details über die Kalibrierung eines Touch–Glases finden sich in Abschnitt 5.5.1 auf Seite 54 und im “Quickstart Tutorial”.

**Übersicht Tglass–Device–Funktionen:**

**tglass_init:** Initialisiert und aktiviert das Tglass–Device.
**tglass_close:** Schließt das Tglass–Device.
**tglass_set_calib_parameters:** Überschreibt die aktuellen Kalibrierungsparameter des Touch–Glases mit neuen Werten.
**tglass_get_calib_parameters:** Liest die aktuellen Kalibrierungsparameter des Touch–Glases.
**tglass_calibration_sequence:** Kalibriert das Tglass–Device und somit das Touch–Glas.

## 3.9 Keyboard–Channel

In vielen Applikationen sind Texteingaben erforderlich. Um verschiedene Eingabemöglichkeiten für Texte in einen hardwareunabhängigen Rahmen zu stellen, sind hier *Keyboard–Channel* implementiert. Diese werden auf der Zielhardware mit Touch–Tastaturen auf dem Touch–Display verbunden (es ist aber auch denkbar, externe Tastaturen zu verwenden, ohne daß die Programmierung mit Hilfe der Keyboard–Channel–Funktionen geändert werden müßte). Der Keyboard–Channel verfügt über einen Tastatur–Puffer, aus dem die ASCII–Zeichen ausgelesen werden können.

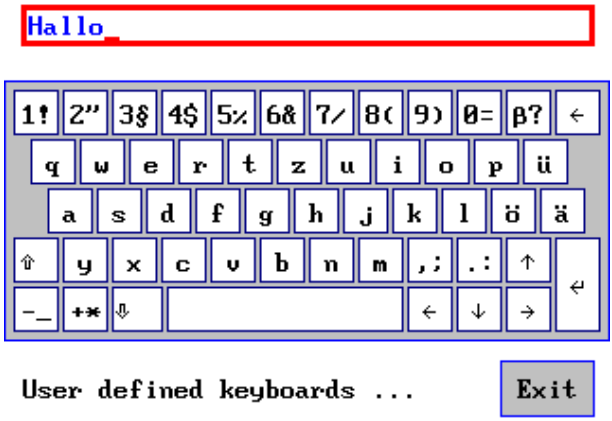



Abbildung 4: Beispiel für Tastatur

**Übersicht Keyboard–Channel–Funktionen:**

**keyboard_open:** Richtet einen Keyboard–Channel ein.
**keyboard_connect:** Verbindet einen Keyboard–Channel mit einem Eingabegerät.
**keyboard_disconnect:** Trennt einen Keyboard–Channel von einem Eingabegerät.
**keyboard_update:** Aktualisiert den Keyboard–Channel (evt. vorhandene Puffer werden geschrieben).
**keyboard_close:** Schließt einen Keyboard–Channel.
**keyboard_getchar:** Liest ASCII–Zeichen aus dem Keyboard–Puffer.
**keyboard_get_status:** Liest den Status des Keyboard–Puffers des Keyboard-Channels.

## 3.10 T_keyb–Device

Das Touch–Keyboard–Device `t_keyb_device` ist eine Touch–Tastatur, die auf dem Touch–Display (also einem Touch–Channel) dargestellt wird. Obwohl das `t_keyb_device` damit ausschließlich aus Software besteht, die letztlich auf vorhandene Hardware zugreift, wird es hier als Device (also wie eine Hardware–Komponente) behandelt. Im Normalfall wird nämlich ein Keyboard–Channel auf einer Zielhardware mit Touch–Glas nicht mit einer externen Hardware–Tastatur verbunden, sondern eben mit einem `t_keyb_device` (als wäre es ein Stück Hardware). Durch die flexiblen Möglichkeiten der Funktionen des `t_keyb_device` können leicht verschiedene Tastaturen für die unterschiedlichen Anwendungen konfiguriert werden, z.B. alphanumerische Tastaturen, Ziffernblöcke und andere anwendungs-

spezifische Tastatur–Anordnungen. Das äußere Aussehen einer Touch–Tastatur wird über einen Stil definiert.
Wie bei *großen* Tastaturen steht auch hier eine Repeat–Funktion für längere Tastendrücke zur Verfügung, ebenso die Möglichkeit, verschiedene Zeichen auszuwählen durch sogenannte *Modifiers*: `KB_SHIFT`, `KB_ALT`, `KB_CTRL`, `KB_CAPS_LOCK`. Bei Bedarf läßt sich solch eine Touch–Tastatur auch auf dem Display verschieben.

**Übersicht T_keyb–Device–Funktionen:**

**t_keyb_style_init:** Initialisiert einen Touch–Keyboard–Stil und holt Speicher dafür.
**t_keyb_style_free:** Gibt den Speicher eines Tastaturstils wieder frei.
**t_keyb_style_set_font:** Setzt den Font eines Touch–Keyboard–Stils.
**t_keyb_style_set_changeflag:** Setzt das Change–Flag, wodurch das Drücken auf eine Umschalttaste (z.B. Shift oder Alt) zu einer Änderung der Tastenbeschriftungen führt.
**t_keyb_style_set_colors:** Setzt die Farben eines Touch–Keyboard–Stils.
**t_keyb_style_set_select_colors:** Setzt die zusätzlichen Farben eines Touch–Keyboard–Stils für den ausgewählten Zustand der Tasten.
**t_keyb_style_set_border_width:** Setzt Rahmenbreite des Tastaturstils.
**t_keyb_make:** Initialisiert ein Touch-Keyboard.
**t_keyb_free:** Gibt den Speicher einer Tastatur–Struktur frei.
**t_keyb_open:** Öffnet ein Touch-Keyboard auf einem Touch–Channel.
**t_keyb_close:** Schließt ein Touch–Keyboard.
**t_keyb_process:** Erledigt die Abfrage eines Touch–Keyboards.
**t_keyb_get_position:** Liest die Position des Touch–Keyboards.
**t_keyb_set_position:** Bewegt das Touch–Keyboard auf eine neuen Position.

## 3.11 Sound–Channel

Für den alltäglichen Einsatz wird oft eine akustische Signalisierung benötigt, sei es, um eine Rückmeldung für eine Berührung auf dem Touch–Display zu bekommen (*Klick*) oder z.B. um die Aufmerksamkeit des Benutzers zu erlangen. Hier sind einfache hardwareunabhängige Sound–Funktionen verfügbar, die die Ausgabe von Tönen erlauben.

**Übersicht Sound–Channel–Funktionen:**

**sound_open:** Initialisiert einen Sound-Channel.
**sound_connect:** Verbindet einen Sound–Channel mit einem anderen Channel oder einem Ausgabegerät (Lautsprecher o.ä.).
**sound_disconnect:** Löst die Verbindung eines Sound–Channels zu einem anderen Channel oder Device (z.B. Lautsprecher).

**sound_close:** Schließt den Sound–Channel.
**sound_update:** Aktualisiert den Sound–Channel (evt. vorhandene Puffer werden geschrieben).
**sound_set_volume:** Stellt den neuen Lautstärkewert für einen Sound–Channel ein.
**sound_get_volume:** Liest die aktuelle Lautstärkeeinstellung eines Sound–Channels.
**sound_make:** Erzeugt Ton auf einem Sound–Channel.
**sound_on:** Schaltet einen Ton auf einem Sound–Channel ein.
**sound_off:** Schaltet einen Ton auf einem Sound–Channel aus.
**sound_click:** Erzeugt einen Tastaturklick.

## 3.12 Piezo–Device

In der Zielhardware erfolgt die Ausgabe von Tönen oft über einen kompakten Piezo–Lautsprecher. Dieser wird nach seiner Initialisierung über einen Sound–Channel angesprochen.

**Übersicht Piezo–Device–Funktionen:**

**piezo_init:** Initialisiert den internen Piezo–Lautsprecher.
**piezo_close:** Piezo–Lautsprecher (Piezo–Device) wird geschlossen (und ist nicht mehr verfügbar).

# 4 Zusätzliche Module (*AddOns*)

Bei den *AddOns* handelt es sich um zusätzliche Module, mit denen die Grundfunktionen der GUI–Bibliothek erweitert werden. Sie befinden sich, ebenso wie die Beipiele, in den dafür vorgesehenen Verzeichnissen auf der CD der GUI–Bibliothek (siehe“Inhalt.txt”).

## 4.1 Sieben–Segment–Anzeige

Das Modul “seven_segments.h/.c” stellt Funktionen zur Darstellung von Sieben–Segment–Zahlen (mit Dezimalpunkt und “E”) zur Verfügung. Hierbei sind die Anzeigen frei skalierbar und auf eine schnelle Ausgabe hin optimiert. Die darzustellenden Zahlen sind als String zu übergeben; alle nicht zulässigen Zeichen in diesem String werden ignoriert. Ein detailliertes Beispiel befindet sich auf der CD (siehe hierzu auch die Funktionsreferenz).

**Übersicht Funktionen der Sieben–Segment–Anzeige:**

**seg7_init:** Initialisiert eine Info–Datenstruktur für die Sieben–Segment–Anzeige und holt Speicher dafür.
**seg7_free:** Gibt den Speicher frei, der durch die Info-Struktur der 7-Segment-Anzeige verbraucht wurde.
**seg7_set_colors:** Setzt die Farben der Zahlen.
**seg7_get_width:** Holt die Breite der gesamten Sieben–Segment–Anzeige in Pixel.
**seg7_define:** Stellt eine Sieben–Segement–Anzeige auf dem Display dar.
**seg7_update:** Erneuert den Inhalt der Sieben–Segment–Anzeige.

## 4.2 Eingabezeile

Dieses Modul “inputline.h/.c” unterstützt den Benutzer bei der Programmierung von Eingabezeilen, wie sie z.B. mit Touch–Tastaturen häufig zum Einsatz kommen. Die zur Verfügung gestellten Funktionen (siehe Funktionsreferenz) übernehmen die grafische Repräsentation der Eingabezeile, wie z.B. bewegen des Textcursors, scrollen des eingegebenen Strings, löschen/einfügen von Zeichen usw.. Der String in der Zeile darf größer sein als die Zeile selber und wird bei Bedarf gescrollt. Die Eingabezeile benutzt die Standardrahmen aus “display.h/.c”. Ein detailliertes Beispiel befindet sich auf CD.

**Übersicht Inputline–Funktionen:**

**inputline_style_init:** Richtet die Struktur des Eingabezeilenstils ein und holt entsprechend Speicher.


Abbildung 5: Beispiel für Sieben-Segmen-Anzeige

**inputline_style_free:** Gibt den Speicher eines Eingabezeilenstils wieder frei.
**inputline_style_set_frame:** Setzt die Eigenschaften des Rahmens eines Eingabezeilenstils.
**inputline_style_set_colors:** Setzt die Farben für den Eingabezeilenstil.
**inputline_style_set_font:** Setzt den Zeichensatz und den Textstil für einen Eingabezeilenstil.
**inputline_define:** Definiert eine neue Eingabezeile, indem zunächst Speicher geholt und die Datenstruktur initialisiert wird. Anschließend wird die Eingabezeile auf dem Display dargestellt.
**inputline_undefine:** Löscht eine Eingabezeile vom Display und gibt den belegten Speicher frei.
**inputline_draw:** Zeichnet die Eingabezeile.
**inputline_blink:** Lässt die Eingabezeile blinken.
**inputline_set_cursor:** Setzt den Cursor auf ein neues Zeichen in der Eingabezeile.
**inputline_move_cursor_right:** Verschiebt den Cursor um ein Zeichen nach rechts.
**inputline_move_cursor_left:** Verschiebt den Cursor um ein Zeichen nach links.
**inputline_move_cursor_left_border:** Bewegt den Cursor zum linken Rand der Eingabezeile.
**inputline_move_cursor_right_border:** Bewegt den Cursor zum rechten Rand der Eingabezeile.
**inputline_delete_char:** Löscht ein Zeichen in der Zeichenkette an der aktuellen Cursorposition.
**inputline_delete_string:** Löscht den kompletten Inhalt der Eingabezeile.

**inputline_insert_char:** Fügt ein Zeichen in die Eingabezeile an der aktuellen Cursorposition ein.
**inputline_insert_string:** Fügt einen String in die Eingabezeile an der aktuellen Cursorposition ein.
**inputline_set_string:** Setzt einen neuen Text in der Eingabezeile.
**inputline_get_cursor_position:** Holt die aktuelle Cursorposition.
**inputline_get_char:** Holt das Zeichen im String an der angegebenen Cursorposition.
**inputline_read_string:** Nur-Lesezugriff auf den Textspeicher der Eingabezeile. Hierzu wird der Textspeicher kopiert.

## 4.3 Fortschrittsbalken

Die Dateien "bar_indicator.h/.c" stellen Funktionen zur Darstellung und Kontrolle von Fortschrittsbalken zur Verfügung. Die Balken werden von den Standardrahmen aus "display.h/.c" umgeben und können sowohl horizontal als auch vertikal ausgerichtet sein. Der Start– und Endwert sind frei wählbar. Hierbei besteht die Möglichkeit, den aktuellen Wert zusammen mit einer physikalischen Einheit (z.B. Temperatur) innerhalb des Balkens anzuzeigen. Außerdem kann ab einem bestimmten Schwellwert der Balken die Farbe ändern, um z.B. eine Gefahrensituation zu signalisieren. Ein ausführliches Programmbeispiel befindet sich auf der CD (siehe auch die Funktionsreferenz).

**Übersicht Bar_Indicator–Funktionen:**

**barind_style_init:** Holt Speicher und initialisiert einen Stil für die Fortschrittsanzeige.
**barind_style_clone:** Holt Speicher für einen neuen Bar–Indicator–Stil und füllt diesen mit den Werten aus einem schon vorhandenen Stil.
**barind_style_free:** Gibt den Speicher eines Bar–Indicator–Stils wieder frei.
**barind_style_set_frame:** Setzt den Rahmenstil für den Stil des Fortschrittbalkens.
**barind_style_set_font:** Setzt Schriftart, Textstil und Farbe für die Textmarkierung des Fortschrittbalkens.
**barind_style_set_bar_appearance:** Setzt das Aussehen des Fortschrittbalkens (Farben und Zeichenmuster).
**barind_define:** Erzeugt einen Forschrittsbalken.
**barind_undefine:** Deaktiviert einen Fortschrittsbalken und gibt den belegten Speicher wieder frei.
**barind_activate:** Zeigt den Fortschrittsbalken auf dem Display an.
**barind_deactivate:** Löscht einen Fortschrittsbalken vom Display.
**barind_set:** Setzt den Fortschrittsbalken auf einen neuen Wert.

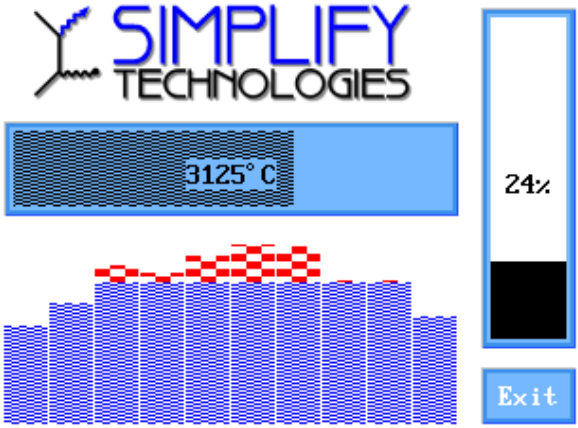



Abbildung 6: Beispiel für Fortschrittsbalken

## 4.4 Schieberegler

Das Modul “slider.h/.c” enthält Funktionen zur Realisierung von Schiebereglern (siehe Funktionsreferenz). Durch Berühren des Touch–Glases kann der Schieberegler ähnlich wie bei anderen Benutzeroberflächen verschoben werden. Der dabei zurückgelieferte Wert liegt zwischen den vorher definierten Grenzen, die wiederum unabhängig von der physikalischen Größe des Reglers sind. Der Schieberegler benutzt die Standardrahmen aus “display.h/.c”. Bei dem Schieberegler handelt es sich um ein *touchable object*, welches durch die selben Funktionen wie die Buttons abgefragt werden kann (siehe `touch_process_objects()` in der Funktionsreferenz). Eine einfache und einheitliche Methode zum Abfragen von *touchable objects* wird in Abschnitt 4.7 vorgestellt. Ein dokumentiertes Beispiel zur Verwendung von Schiebereglern befindet sich auf der CD.

**Übersicht Slider–Funktionen:**

**slider_style_init:** Initialisiert eine Datenstruktur für einen Schieberegler und belegt den Speicher.
**slider_style_clone:** Kopiert einen Schiebereglerstil und legt Speicher für die Kopie an.
**slider_style_free:** Gibt den Speicher eines Schiebereglerstils wieder frei.
**slider_style_set_framestyle:** Setzt den Rahmenstil für den Stil des Schiebereglers.
**slider_style_set_colors:** Setzt die Farben eines Schiebereglerstils.

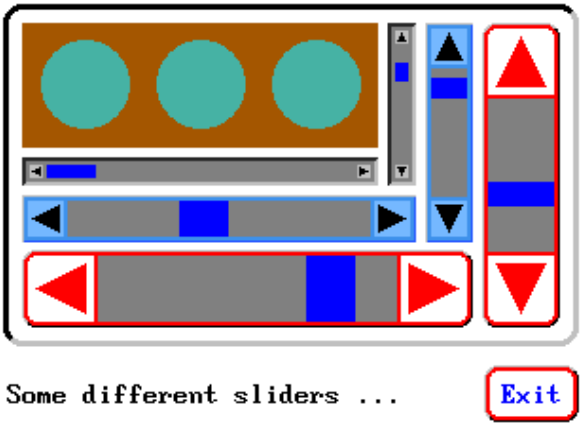



Abbildung 7: Beispiel für Schieberegler

**slider_define:** Definiert einen neuen Schieberegler, indem Speicher für die Datenstruktur belegt wird und diese dann initialisiert wird.

**slider_set_position:** Setzt den Schieber des Schiebereglers auf eine neue Position.

**slider_get_position:** Holt die Position des Schiebers des Schiebereglers.

**slider_redefine:** Setzt einen neuen Bereich für den Schieberegler. Setzt die Größe und die Position des Schiebers neu.

**slider_undefine:** Löscht einen Schieberegler und den von ihm belegten Speicher.

**slider_activate:** Aktiviert einen Schieberegler und stellt ihn auf dem Display dar.

**slider_deactivate:** Deaktiviert einen Schieberegler und löscht ihn vom Display.

## 4.5 Auswahllisten

Das Modul “selection_list.h/.c” enthält Funktionen zur Realisierung von Auswahhlisten (siehe Referenz in der Funktionsreferenz). Durch Touch–Bedienung kann die Liste nach oben und unten gescrollt werden, dabei wählt der Benutzer einzelne oder mehrere Einräge aus. Jeder Eintrag kann aus einem Piktogramm, einem links– und einem rechtsbündigen Text bestehen. Es gibt auch eine Variante der Auswahlliste, die nur zur Darstellung von Informationen dient.
Ein dokumentiertes Beispiel zur Verwendung von Auswahllisten befindet sich auf der CD.


Abbildung 8: Beispiel für Auswahlliste

**Übersicht Selection–List–Funktionen:**

**sl_item_style_init:** Initialisiert eine Stil für die Einträge einer Auswahlliste.

**sl_item_style_clone:** Holt Speicher für einen neuen Stil für die Einträge der Auswahlliste, wobei der neue Stil die Kopie eines alten ist.

**sl_item_style_free:** Gibt den Speicher eines Stils für die Einträge der Auswahlliste wieder frei.

**sl_item_style_set_font:** Setzt den Zeichensatz, den Textstil und die Farbe für einen Texteintrag der Auswahlliste.

**sl_item_style_set_colors:** Setzt die verbleibende(n) Farbe(n) des Stils der Einträge der Auswahlliste.

**sl_style_init:** Initialisiert den Stil der Auswahlliste.

**sl_style_clone:** Kopiert einen Stil der Auswahlliste und legt Speicher für die Kopie an.

**sl_style_free:** Gibt den Speicher eines Auswahllistenstils wieder frei.

**sl_style_set_slider:** Setzt die Parameter des Schiebereglers, der in der Auswahlliste benutzt wird.

**sl_style_set_gap:** Setzt den Abstand zwischen den Icons und dem Text eines Eintrags und zwischen dem Listeneintrag und dem Schieberegler.

**selection_list_define:** Definiert eine neue Auswahlliste.

**selection_list_undefine:** Löscht eine Auswahlliste und gibt den benutzten Speicher frei.

**selection_list_activate:** Aktiviert eine Auswahlliste und stellt sie auf dem Bildschirm dar.
**selection_list_deactivate:** Deaktiviert eine Auswahlliste und löscht diese vom Bildschirm.
**selection_list_add_item:** Fügt einen Eintrag zu einer Auswahlliste hinzu.
**selection_list_remove_item:** Entfernt einen Eintrag aus der Auswahlliste.
**selection_list_count_items:** Zählt die Einträge in der Auswahlliste.
**selection_list_set_position:** Setzt die Auswahlliste auf einen bestimmten Eintrag.
**selection_list_scroll_up:** Scrollt die Auswahlliste um eine Zeile nach oben.
**selection_list_scroll_down:** Scrollt die Auswahlliste um eine Zeile nach unten.
**selection_list_item_set_status:** Setzt den Status eines Eintrags der Auswahlliste.
**selection_list_get_last_event_item_status:** Holt den Status des Eintrags, der den letzten Event ausgelöst hat.
**selection_list_get_select_item:** Holt den Index und den Status des zuletzt selektierten/deselektierten Eintrags.
**selection_list_get_focus_item:** Holt den Index und den Status des zuletzt fokusierten Eintrags.

## 4.6 Menüs

Es gibt drei Unterarten von Menüs:

- *Popup*–Menüs: Ein Menü, welches programmgesteuert auf dem Display erscheint und nach Auswahl eines Eintrags wieder verschwindet (siehe “popup_menu.h/.c”). Das Menü wird ebenfalls wieder ausgeblendet, wenn ein Touch–Kontakt außerhalb des Menüs erfolgt.
- *Single*–Menüs: Eine Kombination von einem Popup–Menü mit einem Button. Der Button dient dazu, das Popup–Menü erscheinen zu lassen. Dieses verschwindet, sobald ein Menüeintrag ausgewählt wurde bzw. der Aktivierungs–Button wieder gedrückt wurde (siehe “menu.h/.c”). Auch hier wird das Menü durch einen Touch–Kontakt außerhalb ebenfalls ausgeblendet.
- Menü–*Bars*: Eine Kombination von einigen Single–Menüs, wobei die Aktivierungs–Buttons alle nebeneinander in einer Zeile liegen und die einzelnen Menüs direkt unter oder über dem Aktivierungs–Button aufklappen (siehe “menu.h/.c”).

Die Menüs benutzen die Standardrahmen aus “display.h/.c”. Bei den Menüs handelt es sich um *touchable objects*, welche durch die selben Funktionen wie die Buttons abgefragt werden können (siehe `touch_process_objects()` in der Funktionsreferenz).
Passende Beispiele befinden sich auf der CD.

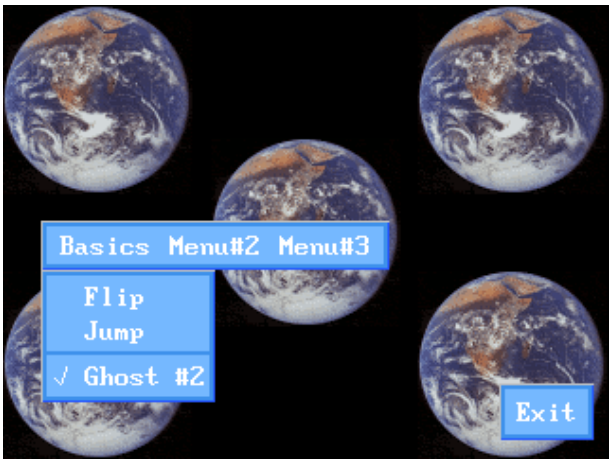



Abbildung 9: Beispiel für Menü

**Übersicht Popup–Menü–Funktionen:**

**popup_style_init:** Holt Speicher für eine Menüstildatenstruktur und initialisiert diese.
**popup_style_free:** Gibt den Speicher eines Menüstils wieder frei.
**popup_style_set_frame:** Setzt den Rahmenstil des Menüstils.
**popup_style_set_separator_height:** Setzt die Höhe der Linie (*Separator*), die zwischen zwei Menüeinträgen gesetzt werden kann.
**popup_style_set_font:** Setzt den Zeichensatz und das Aussehen des Textes für einen Menüstil.
**popup_define:** Legt ein Popup–Menü im Speicher an und initialisiert es.
**popup_add_entry:** Fügt einen neuen Eintrag zu dem Popup–Menü hinzu. Das Menü muß dazu deaktiviert werden.
**popup_remove_entry:** Entfernt einen Eintrag aus dem Popup–Menü. Das Menü muß dazu deaktiviert werden.
**popup_activate:** Aktiviert das Popup–Menü und zeigt es an.
**popup_deactivate:** Löscht das Menü von der Anzeige.
**popup_get_dimensions:** Holt die Größe des Popup–Menüs.
**popup_entry_was_pressed:** Überprüft, ob ein bestimmter Eintrag eines Popup–Menüs ausgewählt wurde.
**popup_set_flags:** Setzt die Flags des Popup–Menüs.
**popup_get_flags:** Holt die Flags des Popup–Menüs.
**popup_set_entry flags:** Setzt die Flags für einen Menüeintrag.

**popup_get_entry flags:** Holt die Flags eines Menüeintrag.
**popup_undefine:** Löscht ein Popup–Menü vom Display und gibt den belegten Speicher wieder frei.

**Übersicht Menü–Funktionen:**

**menu_define_single:** Initialisiert ein Single–Menü und holt den benötigten Speicher.
**menu_undefine_single:** Löscht das Single–Menü vom Display und gibt den belegten Speicher wieder frei.
**menu_activate_single:** Zeigt den Aktivierungs–Button an und fügt das Single–Menü in die Liste der Touch–Objekte ein.
**menu_deactivate_single:** Löscht ein Single–Menü von der Anzeige.
**menu_single_entry_was_pressed:** Überprüft, ob ein bestimmter Eintrag eines Single–Menüs ausgewählt wurde.
**menu_define_bar:** Initialisiert eine Menüzeile und holt Speicher für die Datenstrukturen.
**menu_add_bar_popup:** Fügt ein Popup–Menü zu einer Menüzeile hinzu.
**menu_activate_bar:** Zeigt die Menüzeile an.
**menu_deactivate_bar:** Entfernt die Menüzeile von der Anzeige.
**menu_undefine_bar:** Die Menüzeile wird vom Display entfernt und der belegte Speicher wird freigegeben.
**menu_bar_entry_was_pressed:** Überprüft, ob ein bestimmter Eintrag der Menüzeile gedrückt wurde.
**menu_bar_set_flags:** Setzt die Flags der Menüzeile.
**menu_bar_get_flags:** Holt die Flags einer Menüzeil.

## 4.7 Event–Verarbeitung

Ziel des Moduls “event_handling.h/.c” ist es, die Abfrage von *touchable objects*, also GUI–Elemente, die durch eine Touch–Eingabe bedient werden, einheitlich abzufragen und zu bearbeiten (siehe Funktionsreferenz). Hierzu wird beim Starten der Event–Verarbeitung eine Liste angelegt (*event queue*), in der die einzelnen Ereignisse, die durch die Touch–Objekte ausgelöst wurden, in der Reihenfolge ihres Auftretens abgespeichert werden. Die Auswerteroutine holt dann die einzelnen Ereignisse zur Weiterverarbeitung dort ab. Bei der Gelegenheit wird ein Ereignis auch gerade wieder aus der Liste gelöscht. Werden die Events schneller erzeugt als abgearbeitet, so ist irgendwann die Kapazität der Liste erschöpft und weitere Ereignisse werden verworfen. Man sollte also die Größe der Event–Queue bei der Initialisierung entsprechend wählen und regelmäßig Ereignisse abholen. Z.Z. werden die folgenden *touchable objects* für von der Event–Verarbeitung unterstützt: Buttons, Tastaturen, Schieberegler, Menüs und Auswahllisten. Eine Demonstration der Ereignis–

Verarbeitung befindet sich auf der CD.

**Übersicht Event–Verarbeitungs–Funktionen:**

**events_activate:** Aktiviert die Ereignisverarbeitung für Touch–Objekte wie z.B. Buttons.
**events_is_active:** Überprüft, ob die Ereignisverarbeitung aktiviert ist.
**events_deactivate:** Deaktiviert die Ereignisverarbeitung.
**events_collect:** Sammelt neue Ereignisse ein. Hierzu wird die Liste der Touch–Objekte abgearbeitet, da diese Objekte Ereignisse generieren können.
**events_append_event:** Fügt ein neues Ereignis in die Warteschlange ein.
**events_get_next_event:** Liest ein Ereignis aus der Ereigniswarteschlange und löscht es dort auch.

## 4.8 Sonstiges

### 4.8.1 Clipping

Die Funktionen aus “clip.h/.c” testen, ob die Linien ein vorgegebenes Rechteck scheiden und passen die Anfangs– und Endpunkte so an, daß diese auf dem Rand dieses Rechtecks liegen (siehe Funktionsreferenz und das Beispiel auf der CD).

**Übersicht Clipping–Funktionen:**

**clip_point:** Testet, ob einen Punkt in einem rechteckigen Bereich liegt oder nicht.
**clip_line:** Eine Linie wird auf eine Schnittmenge mit einem rechteckigen Bereich getestet. Die Punkte der Linie werden u.U. so angepaßt, daß sie auf dem Rand des rechteckigen Bereichs liegen.
**clip_text:** Zeichnet einen Text innerhalb eines rechteckigen Bereichs, der vom Text nicht überschritten wird.

### 4.8.2 Zufallszahlen

Die Funktionen zur Erzeugung von Pseudozufallszahlen (vgl. Funktionseferenz ) liegen in den Dateien “random.h/.c”. Der Zufallszahlengenerator muß vor der Benutzung mit einer beliebigen Zahl (*Keim*) gestartet werden, bevor er benutzt werden kann. Die erzeugten Zufallszahlen sind statistisch ziemlich gut verteilt. Ein Beispiel befindet sich auf der CD.

**Übersicht Zufallszahlen–Funktionen:**

**rnd_init:** Initialisiert den Zufallszahlengenerator, der vor der Benutzung von Zufallszahlen gestartet werden muß.

**rnd_32:** Gibt eine 32bit Zufallszahl zurück.
**rnd_number:** Gibt eine Zufallszahl zwischen 0 und (`max`-1) zurück. `max` muß größer oder gleich 1 sein.

### 4.8.3 String–Funktionen

Zur Zeit ist nur eine Funktion zur Umwandlung von Integer–Zahlen in Strings implementiert (vgl. Funktionsreferenz und in den Dateien “string_addon.h/.c”).

**Übersicht String–Funktionen:**

**int_to_str:** Konvertiert eine Zahl in eine Zeichenkette und hängt zusätzlich einen String an, der z.B. als physikalische Einheit benutzt werden kann.

# 5 Anpassung der Bibliothek an die Zielhardware

Dieses Kapitel beschreibt die Anpassung und Konfiguration der Simplify Technologies GUI–Bibliothek für Ihre Anwendung und Zielhardware sowie die Schnittstellen für die Hardware–Treiber.

Um die Simplify Technologies GUI–Bibliothek auf einer bestimmten Zielhardware einzusetzen, müssen einige Definitionen erfolgen, die die Eigenschaften der Zielhardware beschreiben. Diese Definitionen werden in entsprechenden Header–Dateien vorgenommen. Ein paar (wenige) zusätzliche Funktionen werden unter Umständen benötigt, um eine Anpassung an Ihre Hardware zu realisieren.

## 5.1 Anforderungen für den Einsatz der GUI–Bibliothek

Die Simplify Technologies GUI–Bibliothek erlaubt die Programmierung von grafischen Benutzerschnittstellen in ANSI–C. Dadurch kann sie auf sehr unterschiedlichen Hardware–Plattformen eingesetzt werden.

Die Ansteuerung einzelner Hardwarekomponenten, z.B. LCD–Displays, erfordert darüber hinaus einige hardwarespezifische Treiberfunktionen, die weiter unten beschrieben werden. Der Speicherbedarf für die Bibliothek hängt sowohl von dem verwendeten Prozessor und der Entwicklungsumgebung ab als auch von der Anzahl der Funktionen, die Sie für Ihre Anwendung benötigen.
Beispiel: Für einen Hitachi H8/300H–Microcontroller und einen GNU–C–Compiler benötigt eine sehr kleine Anwendung (“Hello World”) inklusive der Funktionen für die Fonts und dem kleinen Font “Mono6x8” unter 20 kByte ROM und deutlich unter 1 kByte RAM. Darin sind bereits Funktionen aus Laufzeit–Bibliotheken von C enthalten, die typischerweise sowieso zu Ihrem Anwendungsprogramm hinzugelinkt werden. Typische Ausführungszeiten für den Hitachi H8/300H–Microcontroller mit 14.7456 MHZ: Ein $320 \times 240$ Pixel großes Display kann in etwa einer knappen Sekunde mit Schriften verschiedener Größe und Attributen beschriftet werden.

Abhängig von den benötigten Funktionen werden unterschiedliche Anforderungen an die Systemumgebung gestellt:
Für die Funktionen zum Ansteuern des LCD–Displays sind keine weiteren Voraussetzungen zu erfüllen, wenn nicht ein automatisch blinkender Sprite–Cursor gewünscht wird. Die Verwaltung der Fonts kann dynamisch geschehen, wenn Ihr System eine dynamische Speicherverwaltung besitzt (also die Verwendung des `malloc()`–Befehls erlaubt), um auch zur Laufzeit noch zusätzliche Fonts hinzunehmen und verwenden zu können. Ansonsten können die Fonts auch ohne `malloc()` verwendet

werden.
Wenn Sie die Möglichkeit wahrnehmen möchten, einen Sprite–Cursor automatisch blinken zu lassen, wird ein Systemtimer benötigt, über den die Handhabung periodischer Funktionen realisiert wird. Dies benötigt dann auch dynamisch allozierten Speicher (`malloc()`).
Die Verwendung eines Touch–Glases erfordert zwingend einen Systemtimer, denn bei der Auswertung einer Berührung muß natürlich zwischen *vorher* und *nachher* unterschieden werden. Außerdem wird hier die dynamische Speicherverwaltung benötigt.
Die Funktionen für Tonausgabe (z.B. als Tastaturklick für Touch–Tastaturen) benötigen einen geeigneten Lautsprecher mit Treiber.

Sollten die Anforderungen für den Einsatz der Bibliothek, obwohl sie bewußt niedrig gehalten wurden, dennoch für Ihren Anwendungsfall problematisch sein, helfen wir Ihnen gerne weiter.

## 5.2 Übersicht und Kurzdarstellung der Konfiguration

Um möglichst schnell mit der Simplify Technologies GUI–Bibliothek arbeiten zu können, empfehlen wir Ihnen folgende Vorgehensweise:

1. Kopieren Sie die Dateien der GUI–Bibliothek und der Hardwaretreiber auf Ihre Festplatte (an einen beliebigen Ort, den Sie für Ihr Projekt als geeignet empfinden). Das Entpacken und Installieren der Dateien wird in der Datei “INSTALL.TXT” beschrieben.

2. Fügen Sie in Ihrer Entwicklungsumgebung sämtliche Dateien der Bibliothek zu Ihrem aktuellen Projekt hinzu. Dieses Hinzufügen von Dateien (bzw. die Einstellung der Pfade für die Source–Dateien) ist in der Anleitung Ihrer Entwicklungsumgebung beschrieben.

3. Passen Sie die Datei “portab.h” an die Gegebenheiten Ihrer Entwicklungsumgebung an: Da der ANSI–C–Standard keine festen Längen für Integer–Variablen hat, sondern nur Mindestlängen vorschreibt, wurden in der gesamten Bibliothek nur die Integer–Datentypen verwendet, die in “portab.h” als `uint8`, `int8`, `uint16` usw. definiert sind (die Zahl gibt hier an, wieviele Bits für die einzelnen Variablen zur Verfügung stehen). Der Typ `t_color` (siehe “display_colors.h”) muß der Farbtiefe der Hardware angepasst werden (siehe auch die Datei “config_gui.h”). Bei monochrom und 8 Bit Farbtiefe sollte `t_color` = `uint8` sein, bei größeren Farbtiefen ist `t_color` = `uint32` zu wählen. Welche Variablengrößen Ihr Compiler verwendet, entnehmen Sie bitte der Datei “limits.h”, die mit dem Compiler mitgeliefert wurde. Ändern Sie dann entsprechend die Definitionen in “portab.h”.

4. Stellen Sie sicher, daß die hardwarespezifischen Treiberfunktionen für Ihr Projekt verfügbar sind. Die erforderlichen Funktionen, auch solche für die Einbindung der Bibliothek in Ihr Gesamtsystem, sind ausführlich in Abschnitt 5.4 beschrieben.
5. Nehmen Sie die Konfiguration der Bibliothek in den Dateien "config_gui.h" und "config_hardware_gui.h" vor. Dadurch wird die Bibliothek den Gegebenheiten Ihres Systems angepaßt, und es werden nur die notwendigen Funktionen mitcompiliert. Detaillierte Informationen hierzu finden Sie in Abschnitt 5.5 auf Seite 53.
6. Nun können Sie die Funktionalität der GUI–Bibliothek in Ihren Anwendungsprogrammen nutzen. Beispiele für Anwendungen finden Sie in Kapitel 7 auf Seite 67 (ein allgemeines, relativ umfangreiches Beispiel) und in eventuell der GUI–Bibliothek beigefügten Treiber–Modulen sowie im "Quickstart Tutorial" und in den Beispielen zu den *AddOns*.

Ein schrittweises Vorgehen bei der Konfiguration von einfachen hin zu komplexeren Anwendungen wird exemplarisch im "Quickstart Tutorial" beschrieben.

## 5.3 Definitionen für die verwendete Entwicklungsumgebung

Die Bibliotheken sind in ANSI–C geschrieben. Der ANSI–Standard definiert jedoch nur Mindestgrößen für Integer–Variablen. Um hier keine Probleme bei der Verwendung unterschiedlicher Compiler zu bekommen, werden innerhalb der Bibliothek nur Integervariablen mit fest definierter Länge verwendet. Diese sind in der Headerdatei "portab.h" definiert. Diese Definitionen sind dort einmalig entsprechend dem verwendeten Compiler einzustellen.
Die GUI–Bibliothek benötigt keine Fließkomma–Arithmetik.

## 5.4 Treiber für Hardware und Systemumgebung

Hier werden die Funktionen vorgestellt, die als Hardwaretreiber direkt auf die jeweils vorhandenen Hardwarekomponenten zugreifen. Diese Funktionen müssen daher entsprechend angepaßt werden. Ihre Deklaration wird jeweils in "<*Modulname*>_.h" vorgenommen, von dem Modul, aus dem sie angesprochen werden. Die Funktionen selbst befinden sich bei mitgelieferten Treibern in eigenen C–Dateien. Zusätzlich werden ein paar Funktionen benötigt, die die Einbindung der Bibliothek in das Gesamtsystem ermöglichen.

Abhängig von der benötigten Funktionalität und den Systemeigenschaften werden die im folgenden beschriebenen Funktionen benötigt.

### 5.4.1 Funktionen zur Realisierung und Verwendung eines Systemtimers

**Wenn nur die einfache Ausgabe auf LCD–Displays gewünscht wird, ist es nicht notwendig, die folgenden Funktionen bereitzustellen; sie werden nur für den automatisch blinkenden Sprite–Cursor und die Touch–Glas–Funktionalität benötigt.**

Ein **Systemtimer** wird benötigt, wenn Sie entweder ein Touch–Glas, das regelmäßig ausgelesen werden soll oder automatisch blinkende Sprite–Cursor einsetzen möchten. Dieser Systemtimer ist eine regelmäßig ausgeführte Funktion (die wir hier einmal `SYSTEMTIMERFUNC` nennen), die ihrerseits die bibliotheksinterne Funktion `system_process_timed_functions()` aufruft. Der Systemtimer wird zweckmäßigerweise über einen Timer–Interrupt ausgelöst. Die GUI–Bibliothek arbeitet dann über die Funktion `system_process_timed_functions()` automatisch die für blinkende Sprites und das Touch–Glas nötigen Funktionen ab (die zusätzliche Systembelastung dadurch ist gering).
Der Systemtimer muß zusätzlich eine *Systemzeit* zur Verfügung stellen, die die Anzahl der Aufrufe von `system_process_timed_functions()` in den beiden globalen Zählervariablen **TIMERDATA** und **DAYDATA** (beide vom Typ `uint32`) folgendermaßen dokumentiert:
`TIMERDATA` ist durch den Systemtimer einfach zu inkrementieren bis zur Dauer von 24 Stunden. Dann ist `TIMERDATA` auf 0 zurückzusetzen und `DAYDATA` um eins zu inkrementieren. Diese Variablen werden auch von den Funktionen `system_wait_until()` und `system_wait_ticks()` ausgewertet, die Sie in Ihrer Anwendung verwenden können. Zum Systemstart werden die beiden Zählervariablen von der Funktion `system_init()` auf 0 gesetzt. Abbildung 10 auf der nächsten Seite zeigt ein Struktogramm der Systemtimerfunktion.

Die Funktion `system_init()` ruft auch eine Funktion **os_systemtimer_start()** auf, die von Ihnen bereitgestellt werden muß. Diese Funktion ist so zu implementieren, daß sie den Systemtimer (d.h. meist den entsprechenden Timer–Interrupt) startet. Nach diesen Vorarbeiten erfolgt dann der Start automatisch mit dem Aufruf von `system_init()` am Beginn des Anwendungsprogrammes.
Damit `TIMERDATA` nicht versehentlich vor dem planmäßigen Übertrag nach `DAYDATA` überläuft, darf der Systemtimer nicht mit einer höheren Frequenz als 49710 Hz betrieben werden.

Wenn das Aufrufen von `system_process_timed_functions()` asynchron zum normalen Programmablauf realisiert wird (wie bei der Verwendung eines Timer–Interrupts für den Systemtimer), muß dem folgendermaßen Rechnung getragen wer-

`SYSTEMTIMERFUNC ---` Zu implementierende Interrupt–Routine des Systemtimers

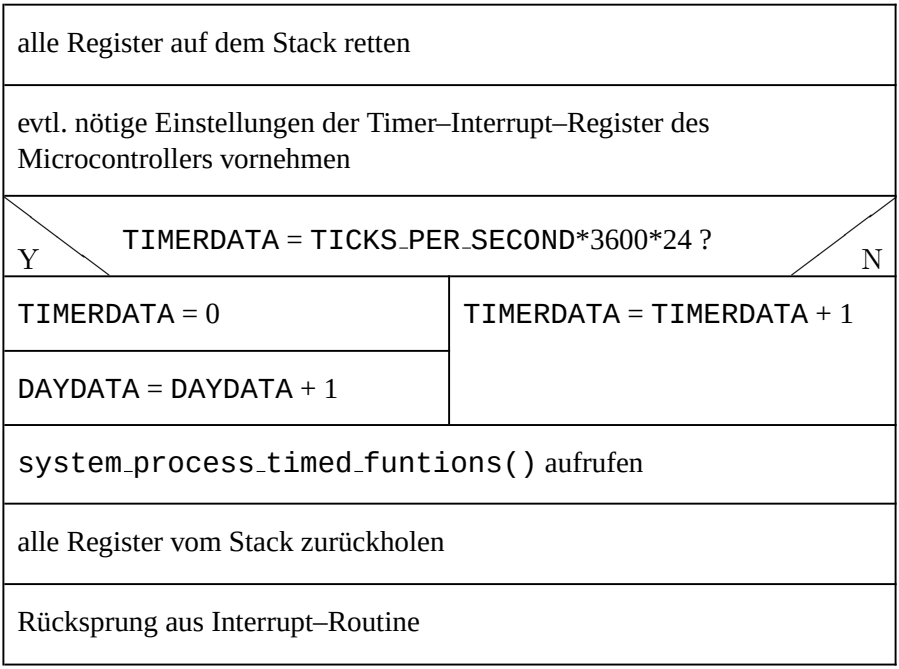



Abbildung 10: Struktogramm der Timer–Interrupt–Routine

den:
Um eine Kontrolle über den Zugriff auf einzelne Strukturen oder Hardwarekomponenten zu haben, werden für blinkende Sprites und Touch–Gläser einfache Semaphoren (im folgenden *Flags* genannt) benötigt. Dies sind 8–Bit–Variablen, die zwei Zustände kennen, nämlich `FLAG_FREE` und `FLAG_BLOCKED` (definiert in "os_gui.h"). Die für die GUI–Bibliothek benötigten Flags sind dort bereits definiert und müssen nicht weiter angepaßt werden. Es ist jedoch erforderlich, folgende Funktionen bereitzustellen, die diese Flags im jeweiligen System sicher, d. h. ohne von anderen (Interrupt-) Funktionen unterbrochen zu werden, setzen und zurücksetzen können:

**err_code os_flag_block( uint8 *flag)**

Diese Funktion muß den Timer–Interrupt sperren (durch Aufruf von

os_disable_timer_interrupt() (s.u.)). Anschließend überprüft sie das angegebene Flag und setzt es auf FLAG_BLOCKED, wenn es vorher FLAG_FREE war. Danach wird der gesperrte Interrupt wieder freigegeben (durch Aufruf von os_enable_timer_interrupt() (s.u.)).

Rückgabewerte: ERR_OK, wenn das Flag noch frei war und erfolgreich gesperrt werden konnte; ERR_FLG, wenn es bereits gesperrt war.

**err_code os_flag_free( uint8 *flag)**

Diese Funktion setzt das Flag auf FLAG_FREE und gibt somit die gesperrte Ressource wieder frei. Das Auslesen des Flags selbst darf nicht durch andere Interrupts gestört werden, so daß auch hier eventuell die betreffenden Interrupts vorher deaktiviert und nach dem Lesen des Flags wieder aktiviert werden sollten.

os_flag_free wird und darf nur benutzt werden, wenn die aufrufende Routine das Flag auch vorher für sich gesperrt hat (also insofern das *Zugriffsrecht* am Flag hat).

Rückgabewert der Funktion: ERR_OK.

Die folgenden Funktionen schalten den Timer–Interrupt ein bzw. aus. Sie werden zum sicheren Verwalten der Semaphoren und für interne Verwaltungsarbeiten der Funktion system_process_timed_functions() benötigt. Implementieren Sie diese Funktionen passend für Ihre Hardware / Ihren Microcontroller:

**void os_disable_timer_interrupt( void)**

Sperrt den Systemtimer–Interrupt.

**void os_enable_timer_interrupt( void)**

Gibt den Systemtimer–Interrupt wieder frei.

Um Wartezeiten eventuell für andere Aufgaben nutzen zu können, gibt es die folgende Funktion:

**void os_process( void)**

Diese Funktion wird während der system_wait–Funktionen aufgerufen, um dem System zu ermöglichen während der Wartezeit andere Aufgaben zu erfüllen. Wenn dies nicht nötig ist, können Sie in der Datei “config_gui.h” die Definition von _OS_USED weglassen und auf die Implementierung dieser Funktion als Funktionsrumpf verzichten.

Die Deklarationen dieser Funktionen und der Konstanten, die von diesen Funktionen verwendet werden, befinden sich in der Datei “os_gui.h”. Die Implementierung der Funktionen sollte in Assembler erfolgen, um eine vollständige Kontrolle des Interrupt–Handlings zu gewährleisten.

`os_flag_block()` --- Zu implementierende Funktion, die Timer–Interrupt–Semaphoren setzt

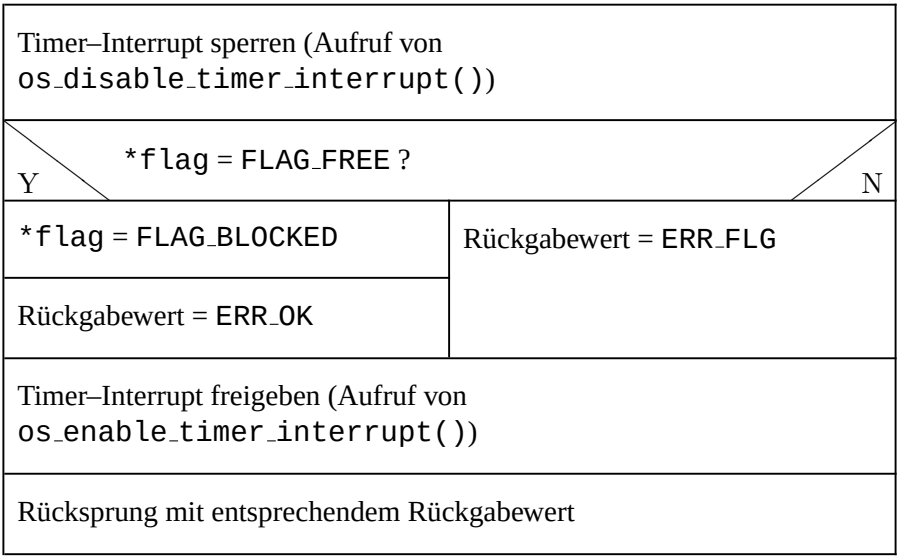



Abbildung 11: Struktogramm von `os_flag_block( uint8 *flag)`

### 5.4.2 Treiber für LCD–Display

LCD–Displays werden üblicherweise von LCD–Controllern angesteuert, die im Adreßraum des Microcontrollers liegen. Diese Adressen der Display–Controller müssen dem Treiber für das LCD–Display bekanntgegeben werden. **Der Treiber mit den hier beschriebenen Funktionen kann fertig mit der Bibliothek erworben werden. Details zu diesem Treiber entnehmen Sie bitte der zum Treiber gehörenden Dokumentation**. Alternativ läßt sich ein Treiber auch über die hier offengelegte Schnittstelle selber programmieren (Deklarationen in “lcd_.h” ):

Die Größe der Typen `t_color` und `t_color_intrep` werden in der Datei “display_colors.h” in Abhängigkeit von der Farbtiefe definiert. Hier bei entspricht `t_color_intrep` der internen Representation eines Pixels, wie sie der jeweilige Display–Controller verwendet (z.B. ist bei einer Farbtiefe von 8 Bit `t_color_intrep` = `uint8`).

Die Endungen der Rendering–Funktionen entsprechen den jeweiligen Orientierungen des Display–Inhalts. `_0` steht für die normale Ausrichtung ($\Leftrightarrow$ Drehung um $0°$). `_180` wird bei einer Drehung des Display–Inhalts um $180°$ benutzt. Alle anderen Orientierungen werden aus den hier beschriebenen Funktionen in einer höheren Abstraktionsebene abgeleitet. Funktionen mit den Endungen `_180` sind nicht notwendig, wenn der LCD–Controller die Display–Rotation direkt in Hardware ausführen kann.

Je nach Farbtiefe können sich die Aufrufe der Treiberfunktionen unterscheiden. Werden keine Angaben gemacht so ist die Funktion bei allen Farbtiefen bereitzustellen.

**void c_lcd_init( uint16 virtual_x, uint16 virtual_y)**

Initialisiert das LCD–Display und schaltet es ein. Wenn das Display die Verwendung virtueller Anzeigenbereiche ermöglicht, wird es als ein virtuelles Display mit den Abmessungen `virtual_x`, `virtual_y` konfiguriert.

**void c_disp_on( void)**

Schaltet das Display ein.

**void c_disp_off( void)**

Schaltet das Display aus.

**void c_lcd_sleep( void)**

Schaltet LCD–Display in den Sleep–Mode (soweit vorhanden), um Energie zu sparen.

**void c_lcd_wakeup( void)**

Aktiviert LCD–Display, um es aus dem Sleep–Mode aufzuwecken.

**void c_lcd_set_viewport( uint16 new_origin_x, uint16 new_origin_y)**

Für Displays mit virtuellem Display–Bereich: Setzt den Ursprung des physikalischen LCD–Displays (die linke untere Ecke des Displays) auf die Koordinaten `new_origin_x`, `new_origin_y` des gesamten virtuellen Displaybereichs.

**void c_lcd_light( uint8 brightness)**

Schaltet die Helligkeit der Hintergrundbeleuchtung des Displays auf den Wert `brightness`.

**void c_lcd_contrast( uint8 new_contrast)**

Stellt den Kontrast des LCD–Displays auf einen neuen Wert ein.

**void c_lcd_bitblt( void)**

Wenn die Grafikdaten nicht direkt in den LCD–Controller übertragen werden, sondern die Grafikdarstellung zunächst in einem RAM–Buffer des Microcontrollers aufgebaut wird, ist es nötig, diese in den LCD–Controller zu übertragen. Die Funktion `c_lcd_bitblt` kopiert den RAM–Puffer in den Display–Speicher des LCD–Controllers. Sie ist nur notwendig, wenn eine solche gepufferte Ausgabe vorsehen ist.

**boolean c_lcd_controller_rotate_0( void)**

Gibt `BOOL_TRUE` zurück, wenn der LCD–Controller die $0°$ Drehung des Display–Inhalts direkt unterstützt. Die Funktion wird benutzt, um von einer beliebigen Orientierung zurück auf die Normalstellung zu drehen.

**boolean c_lcd_controller_rotate_90( void)**

Gibt `BOOL_TRUE` zurück, wenn der LCD–Controller die $90°$ Drehung des Display–Inhalts direkt unterstützt.

**boolean c_lcd_controller_rotate_180( void)**

Gibt `BOOL_TRUE` zurück, wenn der LCD–Controller die $180°$ Drehung des Display–Inhalts direkt unterstützt.

**boolean c_lcd_controller_rotate_270( void)**

Gibt `BOOL_TRUE` zurück, wenn der LCD–Controller die $270°$ Drehung des Display–Inhalts direkt unterstützt.

*Aufruf bei 1 Bit Farbtiefe:*
**void c_lcd_clear( void)**
*Aufruf bei 8/16/24 Bit Farbtiefe:*
**void c_lcd_clear( const t_color_intrep background_color)**

Löscht das LCD–Display. Bei den Farbtreibern kann hierbei die Löschfarbe `background_color` angegeben werden.

*Aufruf bei 1 Bit Farbtiefe:*
**void c_set_pixel_0( uint16 x, uint16 y)**
*Aufruf bei 8/16/24 Bit Farbtiefe:*
**void c_set_pixel_0( uint16 x, uint16 y, const t_color_intrep current_color)**

Setzt einen Pixel an den Koordinaten `x`, `y` mit der Farbe `current_color`.

*Aufruf bei 1 Bit Farbtiefe:*
**void c_clear_pixel_0( uint16 x, uint16 y)**

Löscht einen Pixel an den Koordinaten `x`, `y`. Diese Funktion wird nur im Monochromtreiber benötigt.

*Aufruf bei 1 Bit Farbtiefe:*
**void c_invert_pixel_0( uint16 x, uint16 y)**

*Aufruf bei 8/16/24 Bit Farbtiefe:*
**void c_invert_pixel_0( uint16 x, uint16 y, const t_color_intrep current_color)**

Invertiert einen Pixel an den Koordinaten `x`, `y`.

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

Liest die Farbe eines Pixels an den Koordinaten `x`, `y` in die Variable `pixel_color` ein.

*Aufruf bei 1 Bit Farbtiefe:*
**void c_lcd_hline_0( const uint16 x1, const uint16 x2, const uint16 y, const uint16 line_style, const uint8 draw_mode)**
*Aufruf bei 8/16/24 Bit Farbtiefe:*
**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)**

Zeichnet eine Linie von `x1` nach `x2` in der Höhe `y`. Dabei wird der angegebene `line_style` und `draw_mode` verwendet. Im Falle der Farbtreiber werden die Farben `current_color` und `background_color` verwendet, um den Linienstil zu realisieren.

*Aufruf bei 8/16/24 Bit Farbtiefe:*
**void c_lcd_solid_hline_0( const uint16 x1, const uint16 x2, const uint16 y, const t_color_intrep current_color)**

Zeichnet eine durchgehende Linie von `x1` nach `x2` in der Höhe `y`. Diese Funktion wird nur für die Farbtreiber benötigt.

*Aufruf bei 1 Bit Farbtiefe:*
**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)**
*Aufruf bei 8/16/24 Bit Farbtiefe:*
**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)**

Auffüllen eines rechteckigen Bereiches (linke untere Ecke bei den Koordinaten `x`, `y`, Abmessungen `delta_x`, `delta_y`) mit dem Füllmuster `fill_style` im Zeichenmodus `draw_mode`. `fill_style` ist hier ein Pointer auf ein Array mit sechzehn 16–Bit–Worten, bei denen ein gesetztes Bit einem gesetzten Pixel auf dem Display entspricht. Dabei wird das erste 16–Bit–Wort für die unterste Zeile des zu füllenden Rechteck–Bereiches verwendet (und entsprechend wiederholt, wenn `delta_x` groß genug ist). Das nächste 16–Bit–Wort bestimmt das Aussehen der nächsten Zeile usw. . Im Falle der Farbtrei-

ber werden die Farben `current_color` und `background_color` verwendet, um den Füllstil zu realisieren.

*Aufruf bei 8/16/24 Bit Farbtiefe:*

**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)**

Auffüllen eines rechteckigen Bereiches (linke untere Ecke bei den Koordinaten `x`, `y`, Abmessungen `delta_x`, `delta_y`) mit der Farbe `current_color` (unabhängig vom Zeichenmodus). Diese Funktion wird nur für die Farbtreiber benötigt.

*Aufruf bei 8/16/24 Bit Farbtiefe:*

**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)**

Diese Funktion ändert alle Pixel einer bestimmten Farbe in einem rechteckigen Bereich von `src_color` nach `dest_color`.

*Aufruf bei 1 Bit Farbtiefe:*

**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)**

*Aufruf bei 8/16/24 Bit Farbtiefe:*

**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)**

Zeichnet eine horizontale Linie von `x` nach `endx` in der Höhe `y` mit dem Bitmuster an der Adresse `line_data`. Von dort werden fortlaufende Speicherzellen ausgelesen, bis genügend Bits (1 bedeutet: Pixel gesetzt) für die Linie gelesen sind (startend mit dem höchstwertigen Bit der 1. Adresse für das linke Pixel der horizontalen Linie). Für gesetzte Bits werden Pixel mit der `current_color` gesetzt, ansonsten mit der `background_color`. Der Zeichenmodus ist `draw_mode`; das `flag` bestimmt, ob das Linienmuster invertiert werden soll (bei `flag = 0xff`).

*Aufruf bei 1 Bit Farbtiefe:*

**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)**

*Aufruf bei 8/16/24 Bit Farbtiefe:*

**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)**

Diese Funktion entspricht `c_lcd_patternline_0` und wird beim Zeichnen vertikaler Texte verwendet.

*Aufruf bei 8/16/24 Bit Farbtiefe:*
**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)**

Diese Funktion wird benötigt, wenn Texte mit Antialiasing ausgegeben werden müssen. Sie funktioniert ähnlich wie `c_lcd_patternline_0`, allerdings werden hier immer jeweils zwei Pixel aus zwei übereinander liegeneden Zeilen `line_1st` und `line_2st` (insgesamt vier Pixel) zu einem mittleren Farbwert verrechnet und dann ausgegeben.

*Aufruf bei 8/16/24 Bit Farbtiefe:*
**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)**

Diese Funktion entspricht `c_lcd_antialias_patternline_0` und wird beim Zeichnen vertikaler Texte mit Antialiasing verwendet.

*Aufruf bei 8/16/24 Bit Farbtiefe:*
**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)**

Zeichnet eine horizontale Linie von `x` nach `endx` in der Höhe `y` mit den Farbdaten, die in einem Speicherbereich liegen, auf dessen Anfang der Pointer `line_data` zeigt. Dabei wird der übergebene `draw_mode` berücksichtigt. Diese Funktion wird nur für die Farbtreiber benötigt.

*Aufruf bei 8 Bit Farbtiefe:*
**void c_lcd_set_palette_entry( const uint8 entry, const uint8 red, const uint8 green, const uint8 blue)**

Setzt in der Farbpalette des verwendeten LCD–Controllers einen Farbeintrag. Für den Index `entry` werden drei aufeinanderfolgende Bytes (je eines für Rot, Grün, Blau) geschrieben. Diese Funktion wird nur für den 8 Bit Farbtreiber benötigt.

*Aufruf bei 8 Bit Farbtiefe:*
**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)**
*Aufruf bei 16/24 Bit Farbtiefe:*
**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);**

Zeichnet eine Linie von `x` nach `endx` auf der Höhe `y.` Die Farben der Pixel stehen in einem Speicherbereich, auf den der Pointer `line_data`

zeigt. Der angegebene `draw_mode` wird benutzt. Zum Index der Farben (8 Bit Fall) wird die Anzahl der Systemfarben addiert als Offset zu den BMP–Farbindizes. Wenn `alpha = BOOL_TRUE` ist, dann werden die Pixel mit der Farbe `transparent` nicht gezeichnet. Diese Funktion wird nur für die Farbtreiber benötigt.

*Aufruf bei 16/24 Bit Farbtiefe:*

**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)**

Zeichnet eine Linie von `x` nach `endx` auf der Höhe `y.` Die Farben der Pixel stehen in einem Datenbereich `line_data`, sie entsprechen den Indizes einer vorher festgelegten Farbtabelle. Wenn `alpha = BOOL_TRUE` ist, dann werden die Pixel mit der Farbe `transparent` nicht gezeichnet. Diese Funktion wird nur für die 16/24 Bit Farbtreiber benötigt.

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

Kopiert einen rechteckigen Bereich, der sich von x = `orig_x`, y = `orig_y` um `dx` in x-Richtung und `dy` in y-Richtung erstreckt über eine Distanz von horizontal `distance_x` und vertikal `distance_y`. Zur Zeit ist diese Funktion nur für Farbtreiber implementiert.

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

Liest eine horizontale Linie von `x` nach `endx` in der Höhe `y` und schreibt die Pixel in den Speicher ab der durch den Pointer `line_data` gegebenen Adresse. Die Pixel werden in der Reihenfolge abgelegt, in der sie in `c_lcd_patternline` geschrieben werden.

*Aufruf bei 16/24 Bit Farbtiefe:*

**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)**

Entspricht der Funktion `c_lcd_output_line_0` für eine um 180° gedrehte Darstellung.

*Aufruf bei 16/24 Bit Farbtiefe:*

**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)**

Entspricht der Funktion `c_lcd_output_bmp_line_0` für eine um 180° gedrehte Darstellung.

*Aufruf bei 16/24 Bit Farbtiefe:*
**void c_lcd_scan_line_180( const uint16 x, const uint16 endx, const uint16 y, t_color_intrep *line_data)**

Entspricht der Funktion `c_lcd_scan_line_0` für eine um $180°$ gedrehte Darstellung.

### Funktionen zum Schalten von Kontrast- und Displayspannung:

Die beiden folgenden Funktionen stellen keinen Bestandteil des Treibers da, sondern werden von `c_disp_on` bzw. `c_disp_off` aufgerufen, um die Kontrastspannung ein- und auszuschalten. Wenn nötig, sollten sie entsprechend Ihrer Hardware implementiert werden (Details hierzu finden sich in der Dokumentation zu den Treibern):

**void c_lcd_bias_on_wait( void)**

Diese Funktion soll folgende Aktionen ausführen:

1. Warten der für das verwendete Display spezifizierten Zeit zwischen dem vorher durch den Treiber veranlaßten Einschalten der Displaysignale des LCD–Controllers und dem Einschalten der Spannungen.
2. Aktivierung der Kontrastspannung bzw. Displayspannung soweit anwendbar (TFT–Displaymodule z.B. benötigen in der Regel keine externe Kontrastspannung).

**void c_lcd_bias_off_wait( void)**

Diese Funktion soll folgende Aktionen ausführen:

1. Deaktivieren der Kontrastspannung bzw. Displayspannung soweit anwendbar.
2. Warten der für das verwendete Display spezifizierten Zeit vom Ausschalten und Entladen der Spannungsquellen bis zu dem Zeitpunkt an dem der Treiber anschließend die Displaysignale des LCD–Controllers ausschalten kann.

**void c_lcd_init_lcdsignals_present(void)**

Diese Funktion ist zu implementieren, wenn zusätzliche Initialisierungsschritte benötigt werden *nachdem* die Steuersignale des LCD–Controllers am Display anliegen. Bitte definieren Sie `__LCD_INIT_LCDSIGNALS_PRESENT` damit diese Funktion automatisch nach dem Einschalten der LCD–Controller–Signale aufgerufen wird,.

### 5.4.3 Treiber für Touch–Glas

Bei der Verwendung eines Touch–Glases für die Benutzereingaben müssen zwei elementare Funktionen bereitgestellt werden, die abhängig von der von Ihnen verwendeten Hardware sind. Dies ist einmal eine Funktion zur Initialisierung Ihrer Touch–Hardware und eine Funktion zum Auslesen des Touch–Glases in frei wählbaren Einheiten (z.B. den aus einem A/D–Wandler ausgelesenen Werten). Beide sind in "tglass_.h" deklariert:

**void c_tglass_ground_state( void)**

Diese Funktion initialisiert die Touch–Glas–Hardware und versetzt sie in den Ruhezustand vor einer Auslesesequenz (d.h. z.B. für Folien–Touch–Gläser eine entsprechende Einstellung der Ruhepotentiale an den Folien).

**void c_tglass_get( uint16 *ptr)**

Auslesen der Koordinatenwerte für das Touch–Glas. Diese entsprechen hier noch nicht den kalibrierten Koordinaten, sondern den *rohen* Spannungswerten z.B. eines A/D–Wandlers, die für x- bzw. y–Koordinaten gewonnen werden (die Umwandlung in Pixel–Koordinaten wird später automatisch und nur bei Bedarf vorgenommen). Die gewonnenen x- und y–Werte müssen dann in ein Array aus zwei 16 Bit Werten abgespeichert werden, auf das der Pointer `ptr` zeigt. Dabei ist zuerst der y–Wert, dahinter der x–Wert abzuspeichern !!

Damit die Berechnung der Kalibrierkoeffizienten mittels Integer–Arithmetik korrekt funktioniert, gibt es eine weitere Anforderung an die von dieser Funktion gelieferten Werte: Sie müssen eine Auflösung haben, die rein numerisch mindestens doppelt so hoch ist, wie die Auflösung des unter dem Touchglas befindlichen Displays. Wenn Ihre Hardware eine reale Auflösung liefert, die niedriger ist, müssen die Werte entsprechend *hochskaliert* werden.

Wenn das Touch–Glas nicht gedrückt ist, wird der Wert `_AD_TOUCH_NOT_PRESSED` nach `*ptr` gespeichert. Liegen ungültige Koordinaten vor, so wird die Konstante `_AD_TOUCH_NOT_VALID` an `*ptr` abgelegt. Beide Konstanten sind in "config_hardware_gui.h" als 16 Bit Werte definiert, die ansonsten von der Hardware nicht zurückgeliefert werden.

Da diese Funktion typischerweise periodisch über den Systemtimer–Interrupt abgearbeitet wird, sollte hier auf eine niedrige Laufzeit geachtet werden.

Bemerkung: Bei Touch–Displays mit leitenden Touch–Folien spielt die

RC–Zeit dieser Folienanordnung eine wichtige Rolle. Um zuverlässige Koordinatenwerte zu erhalten, muß sich die Spannung an den Folien stabilisiert haben. Die Funktion `tglass_read()` (in "tglass.c"), die `c_tglass_get()` aufruft, sorgt dafür, daß nur Berührungen, die bereits seit einem Intervall des Systemtimers bestehen als gültig ausgewertet werden. Es steht somit die Zeit eines Intervalls des Systemtimers für die *Aufladung* der Folien auf das stabile Endpotential zur Verfügung.

### 5.4.4 Treiber für Lautsprecher

Für die Ansteuerung des Lautsprechers wird nur eine Funktion (definiert in "piezo_.h") benötigt:

**void c_sound( uint16 frequency, uint16 duration, uint8 volume)**

Ausgabe eines Tones mit der Frequenz `frequency`, der Dauer `duration` und der Lautstärke `volume`.

Die Ausgabe von Tönen ist eigentlich nicht typischer Bestandteil einer grafischen Benutzeroberfläche. Sie wird hier aber dennoch vorgesehen, damit man bei der Verwendung von Touch–Tastaturen und Buttons einen *Tastenklick* als Rückmeldung für den Benutzer verwenden kann (siehe auch die entsprechenden Funktionen für Touch–Tastaturen und Buttons).
In der Bibliothek wird davon ausgegangen, daß es sich bei der Hardware (dem **Device**) für die Tonausgabe um einen Piezo–Lautsprecher handelt (d. h. die Bezeichnungen und Dateinamen enthalten "piezo"). Es können natürlich in diesem Rahmen auch andere Tonausgabegeräte verwendet werden.

## 5.5 Konfiguration der GUI–Bibliothek

Unter Konfiguration wird hier die Spezifikation der Hardware des User–Interfaces und die Auswahl der benötigten Module und Funktionen verstanden. In den Dateien "config_gui.h" und "config_hardware_gui.h" werden die Einstellungen über Definitionen vorgenommen. Für die Konfiguration müssen Sie also den Sourcecode der Funktionen der GUI–Bibliothek nicht modifizieren.

In "config_hardware_gui.h" werden Variablen definiert, durch die die Hardwareumgebung für das User–Interface des Systems beschrieben wird (z.B. die Auflösung des Displays). Variablen für Hardwarekomponenten, die Sie nicht verwenden (z.B. Touch–Glas), können Sie hier unberücksichtigt lassen.

Um für Ihre Systeme keinen unnötigen Code einzubinden, steht hierfür die Konfigurationsdatei "config_gui.h" zur Verfügung, in der über `#define` eingestellt werden kann, welche Module benötigt werden und welche Umgebungsvoraussetzungen gegeben sind.
Die Datei "config_gui.h" enthält zwei Sektionen: In Sektion 1 werden Einstellungen zur Softwareumgebung vorgenommen. Hier können Sie auch auswählen, welche Komponenten der Bibliothek in Ihrem Zielsystem verwendet werden sollen. Durch die bedingte Compilierung auch innerhalb einzelner Funktionen wird so verhindert, daß der Code unnötig Speicher belegt.

Die Konsistenz der Einstellungen in "config_gui.h" wird in der Datei "config_gui_error_check.h" überprüft.

**Beachten Sie bitte, daß die nicht ausgewählten Komponenten auch nicht für die Programmierung der Applikation zur Verfügung stehen. Aufrufe solcher Funktionen würden daher zu Compiler–Fehlern führen, ohne daß dies einen Defekt in der GUI–Bibliothek bedeutet. Nach einer Änderung der Konfiguration ist deshalb auch oft eine Neukompilierung aller Dateien, die die Datei "config_gui.h" einbinden, notwendig.**

In der darauffolgenden Sektion 2 wird Ihre Auswahl ausgewertet. Sollten Sie Einstellungen gewählt haben, die nicht miteinander verträglich sind, werden Sie durch Fehlermeldungen beim Compilieren darauf aufmerksam gemacht (so ist es z. B. nicht sinnvoll möglich, Touch–Tastaturen zu haben, ohne ein Touch–Glas mit den Touch–Funktionen zu benutzen).
Im folgenden werden die Einstellungen in "config_hardware_gui.h" und "config_gui.h" beschrieben.

### 5.5.1 Einstellungen zur Hardware in "config_hardware_gui.h"

**System– und Display–Einstellungen:** Am Anfang der Datei werden zunächst die Eigenschaften des Systems und des Displays festgelegt:

**_SYS_LITTLE_ENDIAN** muß bei Systemen mit Intel–Byte–Order gesetzt werden,

**_LCD_PRESENT** = 1 bei vorhandenem LCD–Display (0 sonst),

**_LCD_LIGHT** = 2 bei vorhandener LCD–Beleuchtung (0 sonst),

**_LCD_LIGHT_ADJUST** = 4, wenn die LCD–Beleuchtungsstärke einstellbar ist (0 sonst),

**_LCD_CONTRAST_ADJUST** = 8, wenn der LCD–Kontrast einstellbar ist (0 sonst),

**_LCD_INVERTED** = 16, wenn man die komplette Darstellung auf dem LCD–Display invertiert wünscht, z.B. bei LCDs, die hardwaremäßig invertiert darstellen (0 sonst).

**_LCD_PHYS_SIZE_X** = Anzahl der Pixel horizontal,

**_LCD_PHYS_SIZE_Y** = Anzahl der Pixel vertikal.

Wenn Sie den Bereich sowie Standardwerte für Kontrast und Beleuchtungsstärke des LCD–Displays berücksichtigen möchten (wenn also auch entsprechende Hardware vorhanden ist), sind folgende Definitionen zu machen:

**_CONTRAST_DEFAULT_DATA** = Kontrastwert (1–255),

**_CONTRAST_LOWER_LIMIT** = Untergrenze für die Kontrasteinstellung (1–255),

**_CONTRAST_UPPER_LIMIT** = Obergrenze für die Kontrasteinstellung (1–255),

**_BRIGHTNESS_DEFAULT_DATA** = Helligkeitswert der Beleuchtung (1–255),

**_BRIGHTNESS_LOWER_LIMIT** = Untergrenze für die Helligkeitseinstellung (1–255),

**_BRIGHTNESS_UPPER_LIMIT** = Obergrenze für die Helligkeitseinstellung (1–255).

**_AUTOMATIC_CONTRAST_TEMP_COMP** = Bedingte Kompilierung: Die Ansteuerelektronik führt eine Temperaturkompensation bei der Kontrastspannung durch.

**_AUTOMATIC_BRIGHTNESS_TEMP_COMP** = Bedingte Kompilierung: Die Ansteuerelektronik führt eine Temperaturkompensation bei der Hintergrundbeleuchtung durch.

**Touch–Einstellungen:** Für das Touch–Glas werden die folgenden Einstellungen benötigt:

**_TGLASS_PRESENT** = 1 bei vorhandenem Touch–Glas (0 sonst),

**_AD_TOUCH_NOT_VALID** = Liefert die Touch–Hardware einen ungültigen Wert, so sollte die Treiberfunktion `_AD_TOUCH_NOT_VALID` zurückgegeben (hier muß ein Wert gewählt werden, der ansonsten nicht vom Hardwaretreiber geliefert werden kann, z.B. 8192).

**_AD_TOUCH_NOT_PRESSED** = Wert, der von der Treiberfunktion zurückgegeben wird, um den *nicht–gedrückt*–Zustand anzuzeigen (hier muß ein Wert gewählt werden, der nicht vom Hardwaretreiber geliefert werden kann, z.B. 4096).

**_TT_DIV und _TT_0_5:** Diese Werte werden zum Berechnen der Touchkoordinaten benötigt und sollten auf _TT_DIV $= 16384$ und _TT_0_5 $=$ _TT_DIV$/2$ gesetzt werden.

**_TOUCH_NORMAL und _TOUCH_ROTATED:** Orientierung des Touch–Glases relativ zum LCD–Display: Wenn die Orientierung der hardwaremäßigen Anschlüsse des Touch–Glases mit den Koordinatenachsen des LCD–Displays übereinstimmt, wenn dieses in der Orientierung `LCD_NORMAL` (ohne Rotation, diese kann nur für bestimmte Treiber bzw. LCD–Controller erfolgen) betrieben wird, setzen Sie bitte `_TOUCH_NORMAL` (dann zeigen also x- und y–Achse von Touch–Glas und LCD–Display in dieselbe Richtung). Diese Einstellung ist für die meisten Aufbauten die richtige.

Wenn der Anschluß des Touch–Glases so realisiert ist, daß dessen Koordinatensystem senkrecht zu dem des LCD–Displays steht, setzen Sie bitte `_TOUCH_ROTATED`.

Es darf selbstverständlich nur eine der beiden Definitionen gesetzt werden.

**_TGLASS_A_DEFAULT_DATA bis _TGLASS_D_DEFAULT_DATA:** Die Kalibrierkonstanten des Touchglases, `_TGLASS_A_DEFAULT_DATA` bis `_TGLASS_D_DEFAULT_DATA`, hängen von der Hardware ab und müssen experimentell (z.B. durch Verwendung der Kalibrierfunktion `tglass_calibration_sequence`) bestimmt werden (siehe auch das entsprechende Beispiel im “Quickstart Tutorial”). Sie sind die Konstanten für die Berechnung der Koordinaten aus den (Spannungs-) Werten, die vom Hardwaretreiber geliefert werden. Es wird ein linearer Zusammenhang der Form $x = A * U_x + B$ zwischen diesen Werten (z.B. Spannungen vom AD–Wandler $U_x$, $U_y$) und den Koordinaten angenommen.

Voraussetzung der Kalibrierung mittels der Funktion `tglass_calibration_sequence` ist, daß das LCD–Display nicht softwaremäßig rotiert wurde, sondern in der Orientierung `LCD_NORMAL` betrieben wird. Die Kalibrierkoeffizienten werden auch verwendet, wenn das Display später gedreht wird; die Koordinaten werden dann automatisch umgerechnet, so daß für den Anwendungsprogrammierer hier keine zusätzliche Arbeit anfällt und auch ein einzelner Satz Kalibrierkoeffizienten für alle Orientierungen ausreichend ist.

**KALIB_X1 bis KALIB_Y4:** Wenn die Kalibrierfunktion `tglass_calibration_sequence` verwendet werden soll, müssen die gewünschten Koordinaten `KALIB_X1` bis `KALIB_Y4` für die Positionen der Fadenkreuze definiert werden. Sinnvoll sind typischerweise Positionen etwas innerhalb der vier Ecken des LCD–Displays. Die Kalibrierfunktion präsentiert die vier Fadenkreuze auf dem LCD–Display und fordert den Anwen-

der auf, diese zu berühren, damit die Kalibrierkonstanten berechnet werden können.

Die Kalibrierroutine gibt bei ihrem Aufruf noch einen Erklärungstext auf dem Display aus. Die Position des Textes wurde auf ein 320×240 Pixel Display ausgelegt und muß bei kleineren Anzeigen im Quellcode der GUI–Bibliothek geändert werden (Dateien “tglass.c” und “text_gui.h”).

**Einstellungen für einen Lautsprecher:** Um einen Lautsprecher (z.B. für Alarmtöne oder *Tastaturklicks*) anzusteuern, werden folgende Definitionen benötigt:

**_PIEZO_PRESENT** = 1 bei vorhandenem Lautsprecher (0 sonst).

**_PIEZO_VOLUME_ADJUST** = 2, wenn die Lautstärke einstellbar ist (0 sonst).

Wenn Sie den Bereich sowie Standardwerte für die Lautstärke des Lautsprechers berücksichtigen möchten (wenn also auch entsprechende Hardware vorhanden ist), sind folgende Definitionen zu machen:

**_VOLUME_DEFAULT_DATA** = Standardeinstellung für die Lautstärke (0–255).

**_VOLUME_LOWER_LIMIT** = Untergrenze für die Lautstärkeeinstellung (0–255).

**_VOLUME_UPPER_LIMIT** = Obergrenze für die Lautstärkeeinstellung (0–255).

Natürlich ist es nicht erforderlich, unbedingt einen Piezo–Lautsprecher als Schallwandler zu verwenden. Lediglich die Hardware–Treiberfunktionen (beschrieben in Kapitel 5.4.4 auf Seite 53) kommen mit dem eigentlichen Lautsprecher *in Kontakt* und müssen entsprechend darauf abgestimmt sein.

### 5.5.2 Einstellungen zur Software–Umgebung in “config_gui.h”

Zunächst muß bestimmt werden, welche Systemumgebung der Bibliothek zur Verfügung steht. Dies beinhaltet Systemtimer und dynamische Speicherverwaltung sowie einige einfache Funktionen. Diese Funktionen sind leicht bereitzustellen und werden detailliert in Kapitel 5.4 auf Seite 40 beschrieben.

#### Die Definitionen für den Systemtimer sind:

**_SYSTEM_TIMER_PRESENT:** Definiert, wenn ein Systemtimer zur Verfügung steht. Dies ist eine Funktion, die folgende Anforderungen erfüllen muß (siehe auch Abschnitt 5.4 auf Seite 40):

- Die periodisch aufzurufenden Funktionen für das Touch–Glas und den blinkenden Sprite–Cursor werden vom Systemtimer über den Aufruf der Funktion `system_process_timed_functions()` aufgerufen.

- Die Zeit wird in zwei globalen 32–Bit–Variablen `TIMERDATA` (Anzahl der Systemticks nach Einschalten des Gerätes bis zu einem Tag) und `DAYDATA` (Anzahl der vollen 24–Stunden–Perioden seit Einschalten des Gerätes) bereitgestellt.
- Dieser Systemtimer sollte für die Verwendung von Touch–Gläsern in der Größenordnung von 100 mal pro Sekunde ausgelöst werden.

**_DISABLE_TIMER_INTERRUPT_NEEDED :** Muß definiert werden, wenn die Adresspointer–Zuweisungsoperationen in den Funktionen `system_timed_function_on()` und `system_timed_function_off()` in der Datei "system.c" nicht in einem Maschinenbefehl abgearbeitet werden können. Dann wird vorher der Timer–Interrupt gesperrt und anschließend freigegeben, so daß die Abarbeitung der Liste durch den Timer immer konsistente Datenstrukturen vorfindet. Dies wird nur benötigt, wenn auch `_SYSTEM_TIMER_PRESENT` definiert ist. Im Zweifel sollte diese Definition vorgenommen werden.

**_TICKS_PER_SECOND:** Anzahl der Systemtimer–Auslösungen pro Sekunde. Wenn z.B. alle 1/100 s der Systemtimer ausgelöst wird, ist der Wert = 100. Über diese Definition wird auch `SEC` automatisch definiert, so daß Sie `SEC` in den Funktionen `system_wait_ticks()` und `system_wait_until()` verwenden können.

Beispiel: `system_wait_ticks( 10*SEC);`

**_SOFTWARE_DELAY_NUMBER:** Wenn kein Systemtimer benötigt wird, arbeitet die Funktion `system_wait_ticks()` über eine Verzögerungsschleife. Wie oft sie durchlaufen werden muß, um das gewünschte Intervall zwischen zwei Systemtimeraufrufen abzuwarten, ist hier zu definieren. Beispiel: Für einen Hitachi H8/300H mit 14.7456 MHZ liegt der korrekte Wert bei etwa 370000.

Die Möglichkeit zur dynamischen Speicherverwaltung ist Bestandteil von ANSI–C (`malloc()`). Dennoch kann es für kleinere Systeme günstig sein, auf diese zu verzichten, wenn man den Verwaltungsaufwand (Verwaltungsdaten im möglicherweise knappen RAM) sparen möchte. In diesem Fall kann die mitgelieferte Speicherverwaltung benutzt werden, die allerings nicht so flexibel und performant wie die Standardversion ist ("simple_memory_wrapper.h/.c").

### Definitionen zur Speicherverwaltung:

**_MALLOC_AVAILABLE:** Definieren Sie `_MALLOC_AVAILABLE`, wenn die dynamische Speicherverwaltung der "stdlib.h" verfügbar gemacht wird.

**_STORAGE_ALLOCATOR_HEAP_SIZE** Wenn Sie die Speicherverwaltung der "stdlib.h" nicht möchten, definieren Sie bitte _STORAGE_ALLOCATOR_HEAP_SIZE als die Anzahl der Bytes, die für die dynamische Speicherverwaltung zur Verfügung stehen. Zur Verwaltung des Speichers für die Speicherverwaltung wird eine Liste der freien Bereiche in diesem Speicher angelegt.

**_MAX_FREE_LIST_ENTRIES:** Diese Defintion ist in Zusammenhang mit der vorherigen nötig und gibt die Anzahl der Einträge der oben genannten Liste vor.

### Weitere Display- und Optimierungsdefinitionen:

**_BITS_PER_PIXEL:** Farbtiefe: 1 bit, 8 bit, 16 bit or 24 bit, abhängig vom verwendeten Treiber (und der verwendet Display–Hardware).

**_MAX_BYTES_PER_LINE:** Gibt die Anzahl der Bytes vor, die eine einzelne Zeile Ihres virtuellen Displays benötigt. Bei monochromen Displays ist dies z.B.:
INT $\left(\frac{1}{8}\,(\text{Anzahl der Pixel des virtuellen Displays in x–Richtung}) + 7\right)$

**_STATIC_RENDERING_BUFFER_WANTED:** Bentuzen Sie diese Definition, wenn der Pufferspeicher für Zeichenoperationen nicht auf dem Heap liegen soll. Diese Option ist nicht mit virtuellen Displays kompatibel.

**_OS_USED:** Um Wartezeiten zu nutzen, die ansonsten in der Funktion system_wait_ticks() in einer Warteschleife verbraucht werden, kann die Definition _OS_USED verwendet werden. Dann wird während der Funktion system_wait_ticks() die von Ihnen bereitgestellte Funktion os_process() aufgerufen. Dann könnte z.B. ein Betriebssystem in der Zwischenzeit weitere Aufgaben wahrnehmen (daher der Name os_process()).

**_MAKE_CHECKS:** Der Code der Bibliothek enthält die Überprüfung der korrekten Verbindungen zwischen den Channels (z.B. Display–Channel) und den damit verbundenen Devices (z.B. LCD–Device). Bei stabilen und getesteten Systemen mit fester Anwendung sollten Fehler hier sowieso nicht mehr auftreten und auf diese Überprüfung kann dann verzichtet werden, um Ausführungszeit und Speicher zu sparen. Die Überprüfung wird mit der Definition von _MAKE_CHECKS eingeschaltet.

**_ALLOW_OVERHEAD:** Für zukünftige Erweiterungen existieren in der GUI–Bibliothek bereits einige Funktions–Hüllen und Variablen, die (noch) keine direkte Verwendung erfahren. Dieser Code kann also ohne weiteres entfernt werden, um unnötigen Speicherplatzbedarf zu reduzieren. Definieren Sie _ALLOW_OVERHEAD, wenn Sie diesen Code dennoch einbinden möchten.

Für die Verwaltung verschiedener Default–Werte (z.B. Display–Kontrast) können die entsprechenden Daten in die jeweiligen Strukturen eingefügt werden (dies ist in der Regel in kleinen Systemen nicht nötig). Wenn Sie dies möchten, definieren Sie:

**_CONTRAST_DEFAULT_HANDLING_WANTED** für Kontrast–Default–Werte beim LCD,

**_BACKLIGHT_DEFAULT_HANDLING_WANTED** für Default–Werte bei der LCD–Beleuchtung,

**_VOLUME_DEFAULT_HANDLING_WANTED** für Default–Werte der Lautsprecherlautstärke,

**_TOUCHPARAMETER_DEFAULT_HANDLING_WANTED** für Default–Werte der Touch–Glas–Kalibrierungskonstanten.

**Konfiguration des benötigten Funktionsumfangs:** Die folgenden Definitionen ermöglichen, den benötigten Funktionsumfang der GUI–Bibliothek festzulegen. Definieren Sie:

**_USE_LCD_DEVICE,** wenn Sie das LCD–Device benutzen möchten (default).

**_USE_PAGEPRINT_DEVICE,** wenn Sie das Pageprint–Device (für Druckerausgabe) benutzen möchten (zukünftige Option),

**_CHOOSE_PRINTERDRIVER_RUNTIME,** wenn es möglich sein soll, den Treiber für das Pageprint–Device zur Laufzeit umzuschalten,

**_PAGEPRINT_COLOR,** wenn das Pageprint–Device für Farbausgabe konfiguriert werden soll (zukünftige Option),

**_GENERAL_LINES_WANTED,** wenn Sie schräge Linien benötigen,

**_BMP_WANTED,** wenn BMP–Bilder Verwendung finden,

**_TRIANGLES_WANTED,** wenn Dreiecke gezeichnet werden sollen,

**_CIRCLES_WANTED,** wenn Kreise und Kreisbögen gebraucht werden,

**_DYNAMIC_FONTS,** wenn die Zeichensätze dynamisch vom System verwaltet werden sollen. Wenn `_DYNAMIC_FONTS` nicht definiert ist, stehen die Funktionen `system_get_number_of_fonts()` und `system_get_font_pointer()` nicht zur Verfügung,

**_TEXT_ANTIALIAS** wenn Kantenglättung bei Texten erwünscht ist,

**_UNICODE_WANTED** wenn Unicode–Unterstüzung bei Fonts aktiviert werden soll,

**_VIRTUAL_DISPLAY_WANTED,** wenn Sie die Funktionalität zum Scrollen des LCD–Bildschirms über ein größeres virtuelles Display benötigen. Ob diese Funktionen sinnvoll sind, hängt auch von Ihrer Hardware und dem LCD–Treiber ab, da ein geeigneter LCD–Controller mit ausreichendem Bildschirmspeicher benötigt wird.

**_BUFFERED_DISPLAY,** wenn die Grafikdaten nicht direkt in den LCD–Controller übertragen werden, sondern die Grafikdarstellung zunächst in einem RAM–Puffer des Microcontrollers aufgebaut wird. Die Funktion `disp_update()` steht Ihnen dann für die Anwendungsprogrammierung zur Verfügung, um die Grafikdaten in den LCD–Controller zu übertragen. Bitte achten Sie darauf, daß Sie diese Funktion dann auch immer anwenden, wenn Ihre Anwendung es erfordert, ansonsten würden die Grafik–Informationen zwar im internen Speicher aufgebaut, aber nicht auf dem Display angezeigt.

**_DISPLAY_ROTATION_90_WANTED,** wenn das Display senkrecht zur normalen Orientierung betrieben (also um 90 Grad gedreht) werden soll. Dieses Feature wird nur für bestimmte LCD–Controller oder Treiber unterstützt.

**_DISPLAY_ROTATION_180_WANTED,** wenn das Display um 180 Grad gedreht werden soll. Dieses Feature wird nur für bestimmte LCD–Controller oder Treiber unterstützt.

**_DISPLAY_ROTATION_270_WANTED,** wenn das Display um 270 Grad gedreht werden soll. Dieses Feature wird nur für bestimmte LCD–Controller oder Treiber unterstützt.

**_DISP_PRINTF_WANTED,** um die Textausgabefunktion `disp_printf()` zu verwenden, die analog der aus C bekannten `printf()`–Funktion funktioniert. Da diese Funktion einige Konversionsfunktionen der C Bibliothek benötigt, kann die Größe Ihres Programmes deutlich ansteigen, wenn diese Funktionen nicht bereits benutzt werden.

**_TOUCH_WANTED,** wenn die Funktionen zur Benutzung eines Touch–Glases verwendet werden,

**_TGLASS_CALIBRATION_WANTED,** wenn die Funktion `tglass_calibration_sequence()` zur Kalibrierung des Touch–Glases benötigt wird,

**_BUTTONS_WANTED,** für Bedienoberflächen mit Touch–Buttons,

**_TOUCH_KEYBOARDS_WANTED,** wenn die Eingabemöglichkeit über konfigurierbare Touch–Tastaturen gebraucht wird.

**_TOUCH_ADDONS_WANTED,** wenn Sie zusätzliche Bedienelemente für Touchgläser benutzen möchten (zukünftige Erweiterung).

**_TOUCH_SELECTION_LISTS_WANTED,** wenn Auswahlisten in den Code aufgenommen werden sollen,

**_ENCODER_SUPPORT_WANTED,** wenn die unterstützung eines externen Encoders gewünscht wird,

**_SOUND_WANTED,** wenn Sie die Funktionen für Lautsprecher mit in den Code übernehmen wollen,

**_SOUND_VOLUME_ADJUST_WANTED,** wenn Ihre Hardware erlaubt, die Lautstärke einzustellen.

**Bitte beachten Sie bei der Konfiguration:** Durch die bedingte Compilierung werden Teile von Strukturen und Funktionen ausgeblendet, um Ressourcen zu sparen. Wenn man diesen ausgeblendeten Code dennoch für andere Zwecke nutzen möchte (z.B. die verketteten Listen aus "struktur.c"), muß man die entsprechenden Compilerswitches bzw. den Code ändern. Beim Verwenden von Funktionen, die bei der Konfiguration ausgeschlossen wurden, tritt sonst ein Compilerfehler auf. Überprüfen Sie dann die Konfiguration, um sicherzustellen, daß die ausgewählte Funktion auch in den Code übernommen wird. Dies gilt analog, wenn Sie Modifikationen oder Erweiterungen an der GUI–Bibliothek vornehmen (so wirkt `__DONT_COMPILE_OVERHEAD` auf alle Funktionen, die momentan nur als Hülle vorhanden sind. Nach dem Ausfüllen dieser Hülle muß dort dann der Switch für die bedingte Compilierung geändert werden, um die Funktion verfügbar zu machen).

## 5.6 Optimierungsmöglichkeiten

Um nicht unnötig Speicherplatz für Module und Rechenzeit für Codesequenzen zu verbrauchen, die auf der Zielhardware gar nicht benötigt werden, können über `#define`–Direktiven in der Datei "config_gui.h" nur die jeweils notwendigen Module compiliert werden. Dies gilt z.B. für die Module "touch.h / tglass.h" für die Funktionen des Touch–Glases und für die Module "keyboard.h" und "t_keyb.h" (Tastaturen auf Touch–Displays). Auch die Module "sound.h" bzw. "pieso.h" sind nicht in jedem Fall nötig. Sie werden verwendet zur optionalen Ausgabe eines Bestätigungstones (*Klick*) bei der Auslösung von Buttons oder Tastaturen. Die Definitionen blenden nicht nur einzelne Funktionen nach Bedarf ein oder aus, sondern optimieren den Code auch innerhalb der jeweiligen Funktion.

Beachten Sie bitte, daß die nicht ausgewählten Komponenten auch nicht für die Programmierung der Applikation zur Verfügung stehen.

Bei knappen Speicher–Resourcen ist es außerdem empfehlenswert, die compilierten Dateien der Simplify Technologies GUI–Bibliothek zu Bibliotheken im Sinne der jeweils verwendeten Entwicklungsumgebung zu machen, so daß der Linker der Entwicklungsumgebung automatisch nur benötigte Funktionen in den Code übernimmt. Da die Simplify Technologies GUI–Bibliothek im Sourcecode geliefert wird, kann darüber hinaus auch Code, der in den einzelnen Modulen nicht benötigt wird, direkt

aus dem Sourcecode entfernt oder modifiziert werden. Wir empfehlen Ihnen, für Änderungen einen Editor mit *Syntax–Highlighting* zu verwenden.

# 6 Farbe und Graustufen (optional)

In diesem Kapitel finden Sie die Beschreibung der Konzepte, die für farbige oder für Graustufendisplays zur Anwendung kommen (auch unterschiedliche Graustufen sind in diesem Zusammenhang *Farben*).

## 6.1 Einschränkungen bei einer Farbtiefe von 8 Bit

### 6.1.1 Farbmodelle und deren Handhabung in der Bibliothek

In der Simplify Technologies GUI–Bibliothek wird Farbe über den Typ `t_color` verwendet. Dieser ist in der Datei "display_colors.h" definiert und wird bei einer Farbtiefe von 8 Bit als `uint8` definiert. Es stehen somit 256 Farben zur Verfügung. Viele LCD–Controller arbeiten mit einem Paletten–Modell für Farben, d.h. die Farben selber werden in einer Index–Tabelle nach den Farbkomponenten (Rot, Grün, Blau) definiert, anschließend aber über den Index dieser Tabelle angesprochen. Diese begrenzte Palette führt zu Einschränkungen in der Benutzung der Farben. Bei einer 256–Farben–Palette können z.B. Fotos in Echtfarbdarstellung nicht dargestellt werden.
Um eine ansprechende Darstellung einer Fotografie zu erzielen, ist es nötig, den begrenzten Farbraum auf dieses Bild hin anzupassen. Schwierig wird dies, wenn zusätzlich noch andere Fotos oder andersfarbige Elemente gleichzeitig auf dem Display dargestellt werden sollen. Es wäre insbesondere störend oder inakzeptabel, wenn die vorher sorgfältig entworfenen Bedienelemente durch eine Änderung der Farbpalette ihre Farben ändern würden. Daher wird in der Bibliothek wie folgt verfahren:

- Es werden zunächst feste *Systemfarben* definiert (siehe "display_colors.h"), die unabhängig von dargestellten Bildern zur Verfügung stehen und sich nicht ändern. Dies sind die 16 Standardfarben der VGA–Palette: `BLACK`, `MAROON`, `GREEN`, `OLIVE`, `NAVY`, `PURPLE`, `TEAL`, `GRAY`, `SILVER`, `RED`, `LIME`, `YELLOW`, `BLUE`, `FUCHSIA`, `AQUA`, `WHITE`. Diese können bei Bedarf leicht an abweichende Anforderungen angepaßt werden (`disp_set_palette_entry()`). Diese Farben können in den Funktionen der Bibliothek auch so mit ihrem Namen angesprochen werden.
- Abhängig von der Anzahl der auf dem Gerät darstellbaren Farben können die über die 16 Standardfarben hinausgehenden Farben zur Darstellung von Bildern genutzt werden. Bei verfügbaren 256 Farben sind so 240 für die Darstellung von Bildern verfügbar. Es ist aber möglich die Anzahl der Farben, die für BMP–Bilder reserviert werden mit dem Befehl `disp_reserve_bmp_colors()` einzustellen, so daß z.B. für Bilder 224 Farben benutzt werden und die restlichen 32 unbeeinflußt bleiben.

### 6.1.2 Handhabung von farbigen BMP–Bildern

Die zur Zeit unterstützten Formate für Bilder sind unkomprimierte Monochrom–BMP–Bilder sowie 16– und 256–Farben–BMP–Bilder (unkomprimiert bzw. RLE–komprimiert). Beim Anzeigen eines farbigen BMP–Bildes werden die Farben des Bildes von der Bibliothek in der Palette des LCD–Controllers so geladen, daß die Paletteneinträge, die nicht für die BMP-Bilder reserviert sind, auch nicht geändert werden. Um eine korrekte Bilddarstellung zu erzielen, ist auf folgendes zu achten:

- Das Bild muß auf die zur Verfügung stehende Anzahl Farben reduziert werden. Dies kann mit einem beliebigen Grafikprogramm durchgeführt werden.
- Wenn mehrere Bilder gleichzeitig angezeigt werden sollen, ist es vorteilhaft, wenn diese alle die selbe Palette haben, so daß nicht so viele Einträge in der Farbtabelle des LCD–Controllers verbraucht werden. Auch dies kann mit einem Grafikprogramm[1] sichergestellt werden. Vorgeschlagene Arbeitsschritte sind:

  1. Kopieren Sie alle Bilder in ein großes neues Bild.
  2. Reduzieren Sie die Farben dieses neuen Bildes auf die zulässige Anzahl.
  3. Separieren Sie die ursprünglichen Bilder aus dem Gesamtbild. Die einzelnen Bilder verwenden nun denselben Farbraum (Farbpalette) und können gleichzeitig ohne Farbstörungen auf dem LCD angezeigt werden.

Wird ein neues Bild geladen, so wird zunächst versucht, die Palette des Bildes in den schon belegten Einträgen der LCD–Controller–Farbtabelle zu finden und diese dann mitzubenutzen. Gibt es keine Überdeckung, so wird die Palette des Bildes in den noch nicht benutzten Farbtabelleneinträgen abgelegt. Sind dort keine Plätze mehr frei, so werden aus dem Bereich der für BMP–Bilder reservierten Farben die ersten Farbtabelleneinträge überschrieben.

Ein Applikationsbeispiel für die Verwendung von Farbe finden Sie sich auf der CD.

#### Ausrichten der BMP–Daten auf Langwortgrenzen

[1] Prinzipiell wäre es auch möglich gewesen, auf die externen Schritte zur Bearbeitung der Bilder zu verzichten und die Umrechnung der Bilder innerhalb der Funktionen der Bibliothek vorzunehmen. Da dies jedoch einen erheblichen Mehraufwand in Rechenzeit und Speicherbedarf bedeuten würde, wurde darauf verzichtet und die externe Bildbearbeitung vorgesehen.

Damit die Bilddaten optimal bearbeitet werden können, müssen diese im Speicher auf Langwortgrenzen liegen. Der Typ BMP–Bilder wird deshalb folgendermaßen definiert (vgl. “display.h”):

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

Es ist ausserdem wichtig, daß die Reihenfolge der einzelenen Datenbytes *big endian* entspricht.
Meist kann man es leicht einrichten, ein BMP–Bild in ein Byte–Array zu konvertieren. Durch den folgenden Trick läßt sich eine Ausrichtung an 32 Bit–Worten erzwingen:

```
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, ...
}};
```

Die Benutzung dieser Tabelle lautet dann:

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

## 6.2 Farbtiefe von 24 Bit *(Truecolor)*

Der Typ `t_color` enthählt in einem 32 bit Wort die Farbinformationen der Rot-, Grün- und Blaukomponente. Es wird empfohlen das vordefinierte Makro DISP_RGB( r, g, b) aus “display_colors.h” zu benutzen, um einen Farbwert zu erzeugen. Mit den Makros `DISP_R_RGB( rgb)`, `DISP_G_RGB( rgb)` und `DISP_B_RGB( rgb)` können die einzelnen Grundfarben aus einem Farbwert extrahiert werden.
BMP–Bilder mit einer Farbtiefe von 24 Bit werden nun unterstützt. Meist reicht es jedoch Bilder mit 256 Farben zu verwenden, die nun ohne Einschränkungen, gleichzeitig dargestellt werden können und darüberhinaus Speicherplatz sparen.
Der Textstil `TXT_LIGHT` wird im 24 Bit–Modus durch eine passende Farbe realisiert und nicht mehr durch Rasterung der Schrift erreicht.

# 7 Ausführliches Programmierbeispiel

Im folgenden wird zunächst ein Programmbeispiel für monochrome Displays besprochen, das die wesentlichen Aspekte der Programmierung der Benutzerschnittstelle illustriert. Anschließend zeigt ein weiteres Beispiel die Verwendung von Farbe für die optionale Farbversion der Bibliothek. Beide Beispiele können als Ausgangspunkt für eigene Programme dienen. Die Sourcen finden Sie auf der CD der Bibliothek in den dafür vorgesehenen Ordnern (siehe “Inhalt.txt”).

## 7.1 Beispiel Monochrom

Bitte beachten Sie, daß für dieses Beispiel alle Module der GUI–Bibliothek verwendet werden, um die vorhandenen Möglichkeiten zu demonstrieren. Bei der Konfiguration der Bibliothek müssen daher alle diese Module mit eingebunden werden. Außerdem wird für das Beispiel die dynamische Verwaltung der Fonts und der Systemtimer verwendet. Dies erfordert als Initialisierung den Aufruf von `system_init()` direkt nach dem Eintreten in die Anwendungsfunktion `main_application()`. Durch Aufruf von `system_init()` wird der Systemtimer gestartet und die Liste der Fonts initialisiert. Einzelne Fonts werden anschließend beim System angemeldet:

```
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
```

Kapitel 5 auf Seite 38 beschreibt die Funktion `os_systemtimer_start()`, die Sie für die korrekte Ausführung von `system_init()` bereitstellen müssen. Diese Funktion ist abhängig von Ihrer Hardware und stellt sicher, daß die interne Funktion `system_process_timed_functions()` regelmäßig aufgerufen wird.
Einfachere Beispiele, die nur einen Teil der Möglichkeiten der Bibliothek nutzen, finden Sie im “Quickstart Tutorial”, das Ihnen den Einstieg in die Programmierung mit der Simplify Technologies GUI–Bibliothek erleichtern soll.

### 7.1.1 Bestandteile des Programmierbeispiels

**Hinweis:** Das hier besprochenen Beispiel enthält die Anwendung in der Funktion `main_application()`. Bitte rufen Sie diese Funktion aus Ihrer `main()`–Funktion auf. Sie können `main_application()` auch in `main()` umbenennen, wenn Sie in Ihrem System von `main()` keine weiteren Aufgaben erledigen lassen müssen. (Für Ihren ANSI–C–Compiler ist es notwendig, daß Sie eine Funktion `main()` in Ihrem Projekt definieren, denn dort wird mit der Ausführung des Programmes begonnen.)
Das Programm beginnt wie die meisten C–Programme mit dem Einlesen der benötigten Header–Dateien und der Definition von globalen Variablen. Dies sind insbesondere die in allen Programmteilen und Unterroutinen verwendeten Channels und Devices sowie die Buttons zur Steuerung des Programmes durch den Benutzer. Anschließend folgt die Implementation der **Anwendungsfunktion** `main_application()`: Zunächst werden die in der Applikation verwendeten Devices initialisiert und mit den Channels verbunden, die eine hardwareunabhängige Programmierung gewährleisten. Dies sind: LCD–Device / Display–Channel, Piezo–Device / Sound–Channel, Tglass–Device / Touch–Channel. Diese Initialisierungs- und Verbindungs–Sequenz ist typisch für alle Anwendungen und wird am einfachsten direkt nach dem Start ausgeführt.
Anschließend erfolgen weitere Initialisierungen, die sich auf Objekte beziehen, die speziell in der vorliegenden Anwendung benutzt werden sollen: Hier werden zunächst die Zeiger auf die System–Fonts erfragt, damit diese später verwendet werden können. Anschließend werden die Buttons und deren Rahmen bzw. Aussehen definiert, die das Auswahlmenü des Beispielprogrammes darstellen und ein Scrollen auf dem virtuellen Display erlauben.

Nachdem die notwendigen Initialisierungen erfolgt sind, wird das Menü aktiviert und das Programm tritt in seine Hauptschleife (die im Falle dieses Beispiels niemals endet). Die Hauptschleife besteht aus der Abfrage der Auswahl–Buttons für die einzelnen Demos. Wenn eine bestimmte Demo ausgewählt wird, wird das Menü über die Unterroutine `deactivate_menu()` deaktiviert und anschließend die gewünschte Demofunktion aufgerufen. Nach der Rückkehr aus der Demo wird durch die Unterroutine `activate_menu()` das Menü wieder aktiviert.
In der Hauptschleife wird zusätzlich das Scrollen über den virtuellen Bild-

schirm ermöglicht, so daß alle Buttons und Demos erreichbar sind, die sonst nicht auf das LCD–Display passen würden.

Nach der Hauptfunktion `main_application()` folgen nun die Unterfunktionen für die einzelnen Demos:

**Die Linien–Demo:** `lines_demo()`**:** Diese Funktion demonstriert die Möglichkeiten der verschiedenen Linienfunktionen. Linien werden in verschiedenen Stilen und Dicken angezeigt. Anschließend wird eine kleine einfache Animation mit Linien erzeugt. Der Programmtext zeigt außerdem Möglichkeiten, die Position des Zeichencursors absolut oder relativ zu seiner vorherigen Position zu verändern.

**Demonstration von Kreisen, Bögen und Rechtecken:** `circles_demo()`**:** In dieser Funktion werden die Funktionen zur Erzeugung von einfachen geometrischen Figuren demonstriert. Neben Kreisen und Bögen (die in Vielfachen von Achtel–Kreisen realisiert sind) können auch Rechtecke mit unterschiedlichem Rand und unterschiedlicher Füllung angezeigt werden.

**Anzeige von BMP–Bildern:** `pictures_demo()`**:** Zur Demonstration der Anzeige von Bildern wird eine Sequenz von schwarz/weißen BMP–Bildern gezeigt. In diesem Beispiel werden BMP–Bilddateien (nachdem sie in ASCII–Headerdateien umgewandelt wurden) als Header–Dateien eingelesen. Genauso ist es möglich, mit der Funktion `bmp_text()` BMP–Daten, die sich anderweitig bereits im Speicher befinden, anzuzeigen.

**Demonstration anderer Grafikfunktionen:** `other_demo()`**:** Hier befindet sich die Demonstration der unterschiedlichen Verknüpfungsmöglichkeiten der Grafikfunktionen mit dem bereits vorhandenen Hintergrund sowie die Demonstration der Sprite–Cursor. Diese erlauben, ein kleines Piktogramm über das Grafik–Display zu bewegen, ohne auf den bereits vorliegenden Hintergrund zu achten (das Sichern und Zurückspeichern des Hintergrundes erfolgt automatisch). Als Beispiel werden zwei verschiedene Cursor–Formen gezeigt, ebenso die Möglichkeit, den Cursor blinken zu lassen, was für die Verwendung als Text–Cursor vorteilhaft ist (dies ist auch in der Demonstration der Tastaturen zu sehen).

**Ausgabe von Text:** `text_demo()`: Diese Funktion erzeugt verschiedene Text–Ausgaben. Diese können mit verschiedenen Zeichensätzen und mit verschiedenen Attributen erfolgen.

**Buttons zur Bedienung durch den Benutzer:** `buttons_demo()`: Hier werden verschiedene Buttons definiert. Nachdem das Aussehen der Buttons und der Rahmen als *Styles* definiert und die Buttons mit der Funktion `button_activate_button()` aktiviert sind, tritt die Funktion in eine Schleife ein, in der das Drücken dieser Buttons mittels der Funktion `touch_process_objects()` ausgewertet wird. Abhängig von der Art des gewählten Buttons werden dann Eigenschaften der Buttons wie Position oder Beschriftung geändert. Der Druck auf den `menu_button` beendet die Schleife und diese Demo.

**Demonstration der Funktionen des Touch–Channels:** `touch_demo()`: In dieser Funktion wird ein Bereich auf dem Touch–Display zum Zeichnen realisiert (man könnte dort z.B. Unterschriften aufnehmen). Nach der Aktivierung der beiden für diese Demo benötigten Buttons wird nach dem Abfangen eventuell noch vorher aufgetretener Kontakte mit dem Touch–Channel eine `do`–Schleife aufgerufen, in der sowohl die Aktivierung der Buttons als auch das Zeichnen auf der Zeichenfläche realisiert ist.

Dazu wird zuerst getestet, ob ein vorhandener Kontakt innerhalb der vorgesehenen Zeichenfläche liegt. Wenn dies der Fall ist, wird eine Linie von der vorherigen Position zur jetzt aktuellen Berührungsposition gezogen. Ist die Berührung außerhalb der Zeichenfläche, werden die Buttons abgearbeitet (mit `touch_process_objects()`). Ist einer der Buttons aktiviert worden, so wird die entsprechende Aktion ausgelöst (Löschen der Zeichenfläche bzw. Verlassen der Demo).

**Texteingabe durch Touch–Tastaturen:** `keyboard_demo`: Diese Demonstration zeigt die Verwendung einer der in Aussehen und Form sehr flexibel definierbaren Touch–Tastaturen. Zunächst wird ein Touch–Keyboard–Device und ein Keyboard–Channel definiert. Anschließend wird ein Stil definiert, der Attribute der Tastatur festlegt. Schließlich wird das Touch–Keyboard mit der Funktion `t_keyb_make()` erzeugt und auf dem Display dargestellt.

Anschließend wird es (es stellt ein Device dar) mit dem Keyboard–Channel verbunden. Nun wird ein Button zum Verlassen der Demo aktiviert.
Die folgende `do`–Schleife erledigt die ganze Arbeit (bis zum Verlassen durch Drücken auf den `menu_button`: In ihr wird zunächst getestet, ob ein Kontakt mit dem Touch–Display innerhalb des Bereichs der Tastatur lag, die in diesem Fall mit `t_keyb_process()` den Kontakt abarbeitet. Das Resultat aus dem Drücken einer Taste auf dem Keyboard wird daraufhin getestet, ob einer der Cursor–Tasten gedrückt wurde (die Zahlen in den `case`–Blöcken bezeichnen die Nummern der Tasten in der Definition der Tastatur (im Header–File, das von "app.c" eingelesen wird). Unter Zuhilfenahme der weiter unten definierten Funktionen `insert_char()` und `remove_char()` wird so der Text–String gemäß der Benutzereingaben in der Variable `string` zusammengesetzt.
Änderungen des Strings, insbesondere durch Löschen oder Hinzufügen von Zeichen, werden zur Aktualisierung wieder auf dem Display ausgegeben.

Bitte beachten Sie auch die der Demonstration beigelegten Beispiele für weitere Keyboards, mit denen Sie experimentieren können (diese sind als Header–Dateien beigefügt und enthalten weitere alphanumerische Tastaturen und Ziffernblöcke).

Auf die Funktionen für die einzelnen Demos folgen weitere einfache Unterroutinen, wie die Anzeige des Info–Fensters (Funktion `info()`), die Funktionen zum Aktivieren und Deaktivieren des Haupt–Menüs (`activate_menu()` und `deactivate_menu()`) und die Warte–Funktion, die am Ende der einzelnen Demos aufgerufen wird (`wait_for_menu_button()`).

## 7.2 Beispiel Farbe

Dieses Beispiel stellt eine ähnliche, aber farbige Applikation, wie das Beispiel für monochrome Darstellung dar. Die dort gegebenen Hinweise gelten entsprechend.

# 8 Software-Sicherheit

## 8.1 Fakten

Software ist komplex, und es ist nur möglich, eine begrenzte Anzahl der möglichen Zustände einer Software zu testen. Bereits bei fertigen Programmen, die mit unterschiedlichen Parametern und Randbedingungen laufen, ergeben sich normalerweise sehr viele mögliche Abläufe. Daher kann nicht davon ausgegangen werden, daß eine Software fehlerfrei ist.
Bei der Verwendung der GUI–Bibliothek kommen noch weitere Tatsachen hinzu, die die Komplexität und damit das Risiko erhöhen, daß Fehler auftreten:

1. Für die Bibliothek kann über die Konfiguration (und die dann erfolgende bedingte Compilierung) eine Vielzahl von Varianten generiert werden.

2. Die Bibliothek wird auf sehr unterschiedlichen Hardware-Plattformen eingesetzt.

3. Die Bibliothek wird nicht alleine auf einem System eingesetzt, sondern als Komponente für eine *beliebige* Anwendung, mit der sie wechselwirkt.

4. Die Erzeugung des ausführbaren Codes kann mit einer Vielzahl von Entwicklungswerkzeugen erfolgen (die ihrerseits komplexe Softwaresysteme sind).

Software ist nur eine Komponente in einem gesamten System. Eine stabil laufende und zuverlässige Hardware ist Voraussetzung für einen sicheren Betrieb.
Simplify Technologies ist bemüht, möglichst hochqualitative Software bereitzustellen. Dennoch ist auch bei der GUI–Bibliothek mit dem Auftreten von Fehlern zu rechnen. Sollte Ihnen ein Fehler in der Bibliothek auffallen, teilen Sie uns dies bitte mit, damit er behoben werden kann.

## 8.2 Vorsichtsmaßnahmen

Die Erstellung zuverlässiger Software kann hier nicht in Kürze adäquat abgehandelt werden. Bitte informieren Sie sich in der zu diesem Thema zahlreich

verfügbaren Literatur [2], der Dokumentation Ihrer Entwicklungswerkzeuge sowie in Fachliteratur für Ihre Anwendung und in eventuell dafür geltenden Vorschriften.

Ein defensiver Programmierstil kann wesentlich zur Zuverlässigkeit Ihrer Anwendung beitragen. Dazu gehört u.a. das Überprüfen von Parametern von Funktionen auf Überschreitung eines erlaubten Wertebereiches und das Testen der Rückgabewerte aufgerufener Funktionen, um aufgetretene Fehler abfangen zu können.

Die fertiggestellte Anwendung sollte in jedem Fall sorgfältig getestet werden. Unter Umständen kann es angezeigt sein, einen von der Software unabhängigen *Watchdog* einzusetzen, der die Integrität des Systems und der Anwendung überprüft, und der das System in einen sicheren Zustand überführt, wenn diese Überprüfung fehlschlägt.

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

# 9 Funktionsreferenz der Bibliothekfunktionen

Eine Übersicht und die Beschreibung der Funktionen finden Sie in diesem Dokument in der Beschreibung der Funktionsmodule im Kapitel 3.

**Die detaillierte Funktionsreferenz zur Programmierung der GUI-Bibliothek befindet sich in der separaten Funktionsreferenz der Bibliothek, die Sie auf der CD der Bibliothek finden.** Für jede der über 300 Funktionen wird dort neben dem Funktionsprototypen und einer Erklärung der Funktionsparameter ein Beispiel für einen typischen Funktionsaufruf angegeben, wie im folgenden Beispiel:

## *disp_line:*

Zeichnet eine Linie mit den momentan gültigen Linienattributen (Linienmuster und Dicke) von und relativ zur aktuellen Cursorposition. Anschließend wird der Endpunkt der Linie zur neuen Cursorposition. Annahme: Aktuelle Cursorposition ist bei x0, y0. Zieht eine Linie zur Position x0 + delta_x - 1, y0 + delta_y - 1 (die Länge der Linie in x- bzw. y-Richtung ist also delta_x und delta_y).

### Funktionsprototyp:

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

### Parameter:

**delta_x, delta_y:** Relative Distanz des Endpunktes von der aktuellen Cursorposition. Negative Werte sind möglich.

### Rückgabewerte:

**ERR_OK:** Erfolg.

**ERR_CNA:** Es ist kein aktuelles Display vorhanden, oder dieses ist nicht verbunden.

**ERR_DDE:** Bei Fehlern des mit dem Display verbundenen Devices.

### Beispiel des Funktionsaufrufs:

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

# 10 Häufige Fragen und Troubleshooting

Hier finden Sie Antworten auf häufig gestellte Fragen und Hilfe bei häufig auftretenden Problemen.

1. **Ich bekomme Fehlermeldungen vom Compiler in der Art "Datei xy not found" bzw. vom Linker, der bestimmte Symbole nicht findet:**

   Der Compiler war nicht in der Lage eine benötigte Datei zu finden. Stellen Sie sicher, daß Sie alle benötigten Dateien ihrer Entwicklungsumgebung bekanntmachen. Eine einfache Lösung wäre auch, alle benötigten Dateien in ein einziges Verzeichnis zu kopieren. Hierbei bitte auch die Dateien des LCD-Treibers nicht vergessen.

   Eine andere Möglichkeit ist, daß die Konfiguration in config_gui.h so vorgenommen worden ist, daß gewisse benötigte Code-Teile nicht mitcompiliert werden. Stellen Sie die config_gui.h so ein, daß alle Compiler-Switches korrekt für Ihre Anwendung gesetzt sind.

2. **Warum treten Compiler-Warnings der Art "Variable x declared but not used" auf ?**

   Die Bibliothek enthält recht allgemein angelegte Strukturen, damit das Programmier-Interface nicht geändert werden muß, wenn Erweiterungen vorgenommen werden.

3. **Alles funktioniert, aber ein bestimmtes Grafikelement oder Text wird nicht dargestellt:**

   Überprüfen Sie, ob die Größe und Positionierung so ist, daß das Element vollständig auf dem virtuellen Display dargestellt werden kann.

4. **Ein Text müßte auf das Display passen; er wird aber dennoch nicht angezeigt:**

   Es ist notwendig, dem aktuellen Display einen Font zur Verwendung anzugeben (mit der Funktion disp_set_font).

5. **Könnte man nicht noch das Feature xy in der Bibliothek haben ?**

   Antwort: Vermutlich ja. Sie können entsprechende Änderungen selber im Quelltext vornehmen. Wenn es sich um eine Eigenschaft von

großem allgemeinen Interesse handelt, können auch wir so eine Erweiterung vornehmen.

6. **Wie kann ich andere Zeichensätze bekommen ?**

   Die GUI-Bibliothek verwendet Zeichensätze im *.FNT-Format. Mit einem geeigneten Fonteditor können Sie die beiliegenden Fonts modifizieren bzw. eigene Fonts erstellen. Die (binäre) *.FNT-Datei konvertieren Sie dann mit dem Bin2H-Utility, das der Bibliothek beiliegt in eine Headerdatei, die Sie dann wie die beiliegenden Zeichensätze verwenden können.

Sollten wichtige Fragen offen geblieben sein, können Sie diese auch direkt an uns stellen.

# A Beiliegende Fonts

Für die beiliegenden Zeichensätze wurden die europäischen Zeichen nach ISO-8859-1 zugrundegelegt. Für manche Zwecke ist es günstig, noch zusätzliche Pfeile zur Verfügung zu haben, z. B. für Tastaturen. Solche Pfeile wurden zusätzlich unter den Hex-Codes 0x1C - 0x1F abgelegt.

Die Zeichensätze stehen in folgenden drei Größen zur Verfügung:

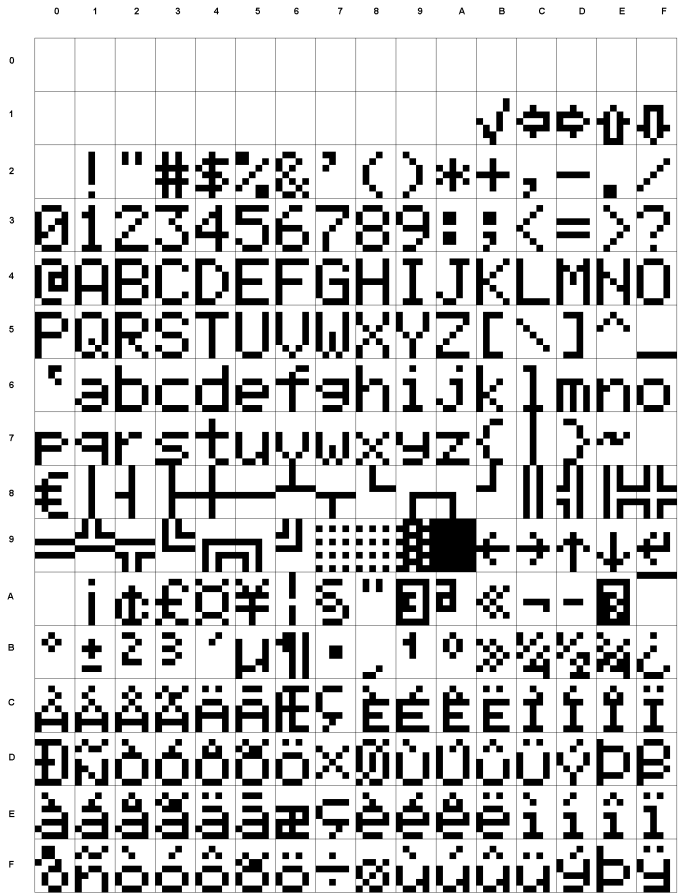



Abbildung 12: Font Mono6x8

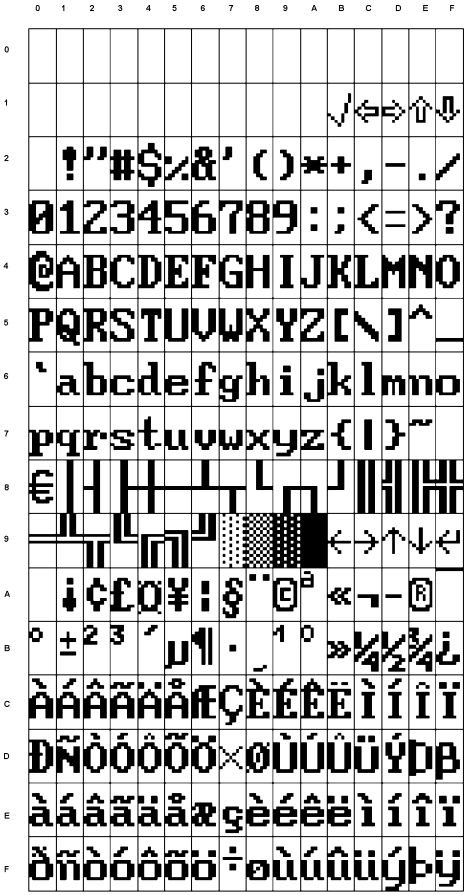


Abbildung 13: Font Mono8x16

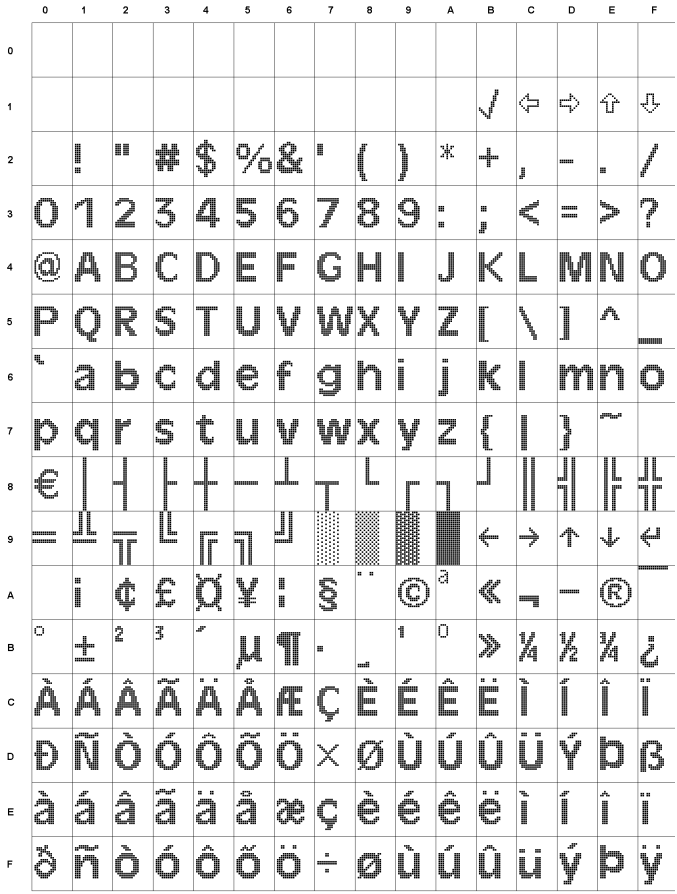


Abbildung 14: Proportionalfont Prop14

# B Lizenz

Allgemeine Softwarelizenz der Simplify Technologies GmbH

Die Firma Simplify Technologies GmbH, Steinbühlstr. 15, D-35578 Wetzlar (im folgenden Lizenzgeber genannt) gewährt dem Lizenznehmer Rechte zur Benutzung der vorliegenden Software, im folgenden "Software" genannt, gemäß den folgenden Bedingungen. Durch Benutzung, Beschaffung oder Verarbeitung der Software werden diese Bedingungen anerkannt.

Definitionen: Die Software kann in folgender Form vorliegen:

"Ausführbare Programme" bezeichnet Software, die direkt auf einer dafür vorgesehenen Zielhardware ausgeführt werden kann, eventuell unter Zuhilfenahme einer Laufzeitumgebung, oder die direkt als Teil eines ausführbaren Programmes (Laufzeit-Bibliothek) genutzt wird. Hierzu zählt auch Software, die direkt auf vom Lizenzgeber gelieferten Systemen implementiert ist ("embedded software").

"Sourcecode" bezeichnet Software, in der Form, in der sie für den Menschen lesbar sind und bearbeitet werden können. Typischerweise ist Sourcecode nicht direkt ausführbar, sondern wird erst durch geeignete Übersetzung oder Interpretation zu ausführbarer Software, d.h. zu ausführbaren Programmen, oder Teilen davon.

"Testversion" bezeichnet Software, die dem Lizenznehmer zu Testzwecken zur Verfügung gestellt wird.

"Beispiel-Software" bezeichnet Software, die als Beispiel für weitere Entwicklungen des Lizenznehmers im Zusammenhang mit Produkten des Lizenzgebers und zur Anleitung dienen soll.

"Käuflich erworbene Software" bezeichnet Software, für die die Lizenz separat von anderen Produkten und Leistungen des Lizenznehmers gegen Zahlung eines vereinbarten Preises zu erwerben ist, oder die zusammen mit und als Bestandteil eines vom Lizenzgeber käuflich erworbenen Produktes geliefert wird.

1. Urheberrecht und Leistungsumfang
a) Der Lizenzgeber hält als Inhaber und Verfügungsberechtigter das nach §§ 69 a ff. UrhG und nach internationalen Vereinbarungen geschützte Softwarerecht. Der Lizenznehmer erkennt den vorstehend genannten Schutz ausdrücklich an.
b) Für kostenfrei bereitgestellte Software und Testversionen gilt: Die Software wird zur Verfügung gestellt, wie sie ist ("as is"). Eigenschaften werden nicht zugesichert.
c) Für käuflich erworbene Software gilt: Der Lizenznehmer erhält alle Unterlagen und Sachen zur Durchführung dieses Vertrages sowie eine angemessene Dokumentation der Software. Die Lieferung erfolgt auf einem Datenträger oder per Datenfernübertragung, die Dokumentation je nach Lieferumfang auch in gedruckter Form.
d) Das Eigentum sowie die Inhaberschaft an geistigen Eigentumsrechten jeder Art sowie an Know-how behält sich der Lizenzgeber vor. Jede Weitergabe an Dritte, soweit nicht nachfolgend gestattet, ist nicht erlaubt.

2. Lizensierte Rechte
Dem Lizenznehmer werden folgende Rechte eingeräumt:
a) Der Lizenznehmer erwirbt ein nicht ausschließliches Nutzungsrecht an der Software. Dieses ist zeitlich unbegrenzt, außer im Fall von "Testversionen", wo es zeitlich auf 30 Tage nach der Beschaffung der Software beschränkt ist.
b) Die Software wird nicht lizensiert für folgende sicherheitskritische Anwendungen: Anwendungen in der Luft-, Raumfahrt-, der Militärtechnik und der Nukleartechnik, Anwendungen mit ionisierender Strahlung, Laser- oder Maserstrahlen, Anwendungen zur Beeinflussung der Bewegung von Fahrzeugen, Anwendungen in der Verkehrssicherheitstechnik (z. B. Airbags, Anti-Blockiersysteme), Anwendungen in Lebenserhaltungssystemen, insbesondere in der Medizintechnik, Anwendungen, bei denen im Versagensfall gefährliche Substanzen freigesetzt werden.
c) Für käuflich erworbenen Sourcecode gilt:
  c1) Der Lizenznehmer darf die Software auf Einzelplatzrechnern oder in Netzwerken innerhalb und am Ort seiner Organisation an einem Standort installieren und mit bis zu 5 Nutzern benutzen.

c2) Die Software oder Teile davon dürfen vom Lizenznehmer mit beliebigen Entwicklungswerkzeugen in eine ausführbare Form übersetzt und so in dessen Produkte übernommen werden und als fester Bestandteil dieser Produkte zusammen mit diesen veräußert werden, sofern es sich nicht um Entwicklungswerkzeuge für Display-Anwendungen Dritter handelt und die Software nicht Bestandteil eines Entwicklungswerkzeuges wird.
c3) Der Lizenznehmer ist berechtigt, die Software im Sourcecode zu modifizieren. Die Rechte auch der modifizierten Versionen liegen beim Lizenzgeber, auch ansonsten unterliegen die modifizierten Versionen diesem Lizenzvertrag. Über beabsichtigte Veränderungen und Verbesserungen hat der Lizenznehmer den Lizenzgeber zu informieren und diesem rechtzeitig vor der Produktfertigstellung ein kostenloses Probeexemplar zukommen zu lassen.
c4) Eine Weitergabe der Software im Quelltext und in linkfähigen Objektformaten an Dritte ist nicht gestattet, auch nicht in Teilen.

d) Für "Beispiel-Software" gilt:
d1) Der Lizenznehmer darf die Software auf Einzelplatzrechnern oder in Netzwerken innerhalb und am Ort seiner Organisation an einem Standort installieren und mit bis zu 5 Nutzern der Software an einem Standort ausschließlich im Zusammenhang mit Produkten des Lizenzgebers benutzen.
d2) Soweit die Beispiel-Software aus Sourcecode besteht, darf sie oder Teile davon vom Lizenznehmer mit beliebigen Entwicklungswerkzeugen in eine ausführbare Form übersetzt und so in dessen Produkte übernommen werden und als fester Bestandteil dieser Produkte zusammen mit diesen veräußert werden, sofern es sich nicht um Entwicklungswerkzeuge für Display-Anwendungen Dritter handelt, die Software nicht Bestandteil eines Entwicklungswerkzeuges wird und die Software in Zusammenhang mit Produkten des Lizenzgebers eingesetzt wird. Wenn die Software eine Testversion ist, besteht das Recht, diese zu verkaufen, auch im Zusammenhang mit Produkten des Lizenzgebers, nicht.
d3) Soweit die Beispiel-Software aus Sourcecode besteht, ist der Lizenznehmer berechtigt, die Software im Quellcode zu modifizieren. Die Rechte auch der modifizierten Versionen liegen bei dem Lizenzgeber, auch ansonsten unterliegen die modifizierten Versionen diesem Lizenzvertrag. Über beabsichtigte Veränderungen und Verbesserungen hat der Lizenznehmer den Lizenzgeber zu informieren und diesem rechtzeitig vor der Produktfertigstellung ein kostenloses Probeexemplar zukommen zu lassen.

e) Für "Ausführbare Programme" gilt:
e1) Sofern es sich bei der Software nicht um Beispielprogramme handelt gilt: Der Lizenznehmer darf die Software auf einem Einzelplatzrechner und am Ort seiner Organisation an einem Standort installieren und nutzen. Außerdem darf der Lizenznehmer bis zu zwei Sicherheitskopien, ausschließlich zur Archivierung, anlegen.
e2) Bezüglich der Rückentwicklung und Decompilierung (Reverse Engineering) gilt das Gesetz über Urheberrecht und verwandte Schutzrechte.
e3) Wenn die Software mit dem .NET-Framework der Firma Microsoft erstellt wurde, gilt bezüglich der von Microsoft bereitgestellten Komponenten der entsprechende, in diesem Fall der Software beigelegte "Endbenutzer-Lizenzvertrag für Microsoft- Software". Die von Microsoft bereitgestellten Komponenten dürfen vom Lizenznehmer, der die Software mit den vom Lizenzgeber hergestellten Produkten weitervertreibt, ausschließlich in Verbindung mit und als Teil der vertragsgegenständlichen Software weitervertrieben werden, wobei die Bestimmungen des "Endbenutzer-Lizenzvertrag für Microsoft-Software" einzuhalten sind.

3. Vergütung
Für "Käuflich erworbene Software" gilt: Für die lizensierten Rechte ist ein Entgelt gemäß Rechnungsstellung zu entrichten. Es gelten die jeweils aktuellen Preislisten bzw. Angebote. Die Lizenz gilt erst zu dem Zeitpunkt als erteilt, in dem die vollständige Bezahlung erfolgt ist.

4. Einschränkungen
a) Sämtliche Schutz-, Urheber- und geistigen Eigentumsrechte liegen beim Lizenzgeber oder dessen Lieferanten, insbesondere die für Programme, Software, Texte, Bilder, Animationen, Audiodaten. Alle nicht ausdrücklich in dieser Lizenz gewährten Rechte bleiben dem Lizenzgeber bzw. dessen Lieferanten vorbehalten.
b) Der Lizenzgeber behält sich zukünftige Änderungen an der Software vor.

5. Beendigung der Lizenz und Geheimhaltung
a) Der Lizenznehmer kann diese Softwarelizenz jederzeit durch vollständiges Löschen der Software von seinem Computer bzw. Netzwerk

beenden.
b) Die Lizenz endet automatisch, wenn der Lizenznehmer gegen die Bestimmungen der Lizenz verstößt. In einem solchen Fall ist der Lizenznehmer verpflichtet, sämtliche Kopien der Software mit allen ihren Komponenten zu vernichten und vom Lizenzgeber erhaltene Produktunterlagen, Dokumentationen und schriftlich fixiertes Know-how an den Lizenzgeber zurückzugeben. Ein Zurückbehaltungsrecht steht dem Lizenznehmer nicht zu.
c) Für "käuflich erworbene Software" und "Beispiel-Software" gilt: Mit Beendigung der Lizenz erlischt auch das Recht, Produkte, die diese Software oder Teile davon in ausführbarer Form enthalten, zu veräußern.
d) Die Vertragsparteien verpflichten sich, alle in diesem Vertragsverhältnis erlangten Informationen über den Vertragspartner unbefristet geheim zu halten. Das gilt neben den Kenntnissen über die Produkt- und Geschäftspolitik sowie Vertriebswege besonders für alle Informationen, die als vertraulich bezeichnet werden oder als Betriebs- und Geschäftsgeheimnisse erkennbar sind.
e) Die Vertragspartner verpflichten eingeschaltete Dritte, ebenfalls diese Pflichten zu erfüllen.

6. Beschränkte Gewährleistung
a) Der Lizenzgeber haftet nicht dafür, daß die lizensierte Software bestimmte Leistungsergebnisse herbeiführt. Das gilt auch für die Gebrauchsfähigkeit des Softwarepakets zu dem vereinbarten oder einem anderen Zweck. Das Risiko der wirtschaftlichen Verwertbarkeit liegt beim Lizenznehmer.
b) Für "käuflich erworbene Software" gilt:
  b1) Der Lizenzgeber leistet Gewähr dafür, daß die Software im wesentlichen den in der Dokumentation beschriebenen Funktionen entspricht und daß die Software frei von Mängeln ist, die den Wert oder die Gebrauchsfähigkeit der Software zu dem vertraglich vorausgesetzten Zweck aufheben oder mindern. Unerhebliche Abweichungen oder Minderungen bleiben außer Betracht. Die vorstehende Gewährleistung bezieht sich nicht auf Mängel, die auf Veränderungen der Software durch den Lizenznehmer zurückzuführen sind.
  b2) Rügt der Lizenznehmer einen oder mehrere Mängel, so ist der Lizenzgeber dazu berechtigt und verpflichtet, den/die Mängel auf seine Kosten zu beseitigen oder gleichwertigen Ersatz zu liefern.
  b3) Scheitern Nachbesserungsversuche, so hat der Lizenznehmer das Recht, nach seiner Wahl entweder das vereinbarte Entgelt herabzusetzen oder vom Vertrag zurückzutreten.
c) Für nicht käuflich erworbene Software gilt:
Es werden ausdrücklich keine Eigenschaften der Software zugesichert. Auch die Beschreibung in der Dokumentation ist keine zugesicherte Eigenschaft. Der Lizenzgeber übernimmt keine Gewähr dafür, daß die Software fehlerfrei ist, oder den Anforderungen des Lizenznehmer, auch in Zusammenhang mit anderen vom Lizenznehmer verwendeten Programmen, genügt.
d) Für Beispiel-Software gilt:
Es werden ausdrücklich keine Eigenschaften der Software zugesichert. Auch die Beschreibung in der Dokumentation ist keine zugesicherte Eigenschaft. Der Lizenzgeber übernimmt keine Gewähr dafür, daß die Software fehlerfrei ist, oder den Anforderungen des Lizenznehmers, auch in Zusammenhang mit anderen vom Lizenznehmer verwendeten Programmen, genügt.
e) Die Gewährleistungsfrist beträgt 12 Monate bei gewerblichen Kunden und juristischen Personen und 24 Monate bei Verbrauchern. Die Verjährung beginnt mit der Ablieferung der vertragsgegenständlichen Software.

7. Beschränkte Haftung und Freistellung
a) Bei Software als einem naturgemäß komplexen Produkt kann nicht davon ausgegangen werden, daß sie fehlerfrei ist. Die Software ist nicht für die Verwendung in Produkten oder auf eine Art und Weise geeignet, die bei fehlerhafter Funktion Schäden verursachen könnte. Dies gilt insbesondere für die Verwendung der Software im Bereich sicherheitskritischer Anwendungen wie z. B. der Medizintechnik, der Luft-, Raumfahrt- und Verkehrstechnik, der Nukleartechnik und Militärtechnik. Der Lizenznehmer erkennt dies durch die Benutzung der Software an. Für die Folgen aus der Benutzung der Software ist der Lizenznehmer selbst verantwortlich. Der Lizenznehmer ist verpflichtet, die Eignung der Software und die korrekte Funktion für die jeweiligen Anwendungszwecke selbst zu verifizieren und sicherzustellen.
b) Eine Haftung ist, soweit nicht der Lizenzgeber oder seine Erfüllungsgehilfen vorsätzlich oder grob fahrlässig gehandelt hat, dem Grunde nach ausgeschlossen. Schadenersatzansprüche des Kunden, gleich aus welchem Rechtsgrund, insbesondere

aus Vertragsverletzung, aus der Verletzung von Pflichten bei Vertragsverhandlungen und aus unerlaubter Handlung sind ausgeschlossen. Dies gilt nicht, soweit z. B. nach dem Produkthaftungsgesetz oder in den Fällen des Vorsatzes, der groben Fahrlässigkeit, des Fehlens zugesicherter Eigenschaften oder der Verletzung wesentlicher Vertragspflichten, sowie bei dem Lizengeber zurechenbaren Körper- und Gesundheitsschäden oder bei Verlust des Lebens zwingend gehaftet wird.
c) Insbesondere wird die Haftung auch für folgende Schäden ausgeschlossen: Der Lizenzgeber haftet nicht für Datenverluste. Der Lizenznehmer weiß, dass er zu regelmäßiger Datensicherung verpflichtet ist. Ausgeschlossen wird auch die Haftung für entgangenen Gewinn, Betriebsunterbrechung, Verlust geschäftlicher Informationen oder irgendeinem anderen Vermögensschaden aus der Benutzung der Software oder aus der Tatsache, daß sie nicht benutzt werden kann. Ebenso ist die Haftung für unmittelbare oder mittelbare Schäden aus der Benutzung der Software bzw. der Unmöglichkeit, die Software zu benutzen, ausgeschlossen. Die Haftung für unvorhersehbare, untypische Schäden, sowie für Folgeschäden ist ebenfalls ausgeschlossen. Dies gilt auch dann, wenn der Lizenzgeber auf die Möglichkeit solcher Schäden hingewiesen worden ist. Eine Änderung der Beweislast zum Nachteil des Kunden ist mit dieser Regelung nicht verbunden. Der Lizenzgeber empfiehlt dem Lizenznehmer, seine Datenbestände regelmäßig zu sichern, und die Ergebnisse seiner Arbeit zu überprüfen.
d) Unter keinen Umständen übersteigt der Haftungsbetrag die für die Software bezahlte Lizenzgebühr.
e) Die Haftungsbeschränkung gilt auch für Mitarbeiter, Vertreter, Erfüllungsgehilfen und Lieferanten des Lizenzgebers.
f) Der Lizenznehmer stellt den Lizenzgeber von Ansprüchen Dritter aus Produkthaftung frei.
g) Der Lizenznehmer haftet für alle Angaben und Behauptungen, die er bei Vertrieb und Werbung aufstellt.

8. Schutzrechte Dritter
a) Der Lizenzgeber geht ausschließlich für den Bereich der Bundesrepublik Deutschland davon aus, daß der vertragsgemäße Gebrauch der Software keine Schutzrechte Dritter beeinträchtigt. Beeinträchtigt er dennoch die Schutzrechte Dritter, haftet der Lizenzgeber gegenüber diesen Dritten für den Bereich der Bundesrepublik Deutschland. Für eine Verletzung von Schutzrechten Dritter außerhalb der Bundesrepublik Deutschland übernimmt der Lizenzgeber keine Gewährleistung. Die Überprüfung, ob die Software außerhalb der Bundesrepublik Deutschland eingesetzt werden darf, obliegt dem Lizenznehmer.
b) Der Lizenznehmer benachrichtigt den Lizenzgeber unverzüglich, wenn Dritte Schutzrechtsverletzungen geltend machen. Der Lizenzgeber trägt die Kosten für rechtliche Auseinandersetzungen um Schutzrechte für den Bereich der Bundesrepublik Deutschland. Der Lizenzgeber entscheidet über die rechtlichen Abwehrmaßnahmen sowie bei Vergleichsverhandlungen. Kosten für rechtliche Auseinandersetzungen für den Bereich außerhalb der Bundesrepublik Deutschland trägt der Lizenznehmer.
c) Beeinträchtigt eine vertragsgemäße Nutzung die Schutzrechte Dritter, im Bereich der Bundesrepublik Deutschland, hat der Lizenzgeber, unter Berücksichtigung der besonderen Umstände des Lizenznehmers, die Wahl, ob er die Lizenz erwirbt, die Software ändert, sie - eventuell teilweise - austauscht, oder die Lizenz beendet.
d) Räumt der Lizenzgeber nicht die Rechte Dritter für den Bereich der Bundesrepublik Deutschland aus, berechtigt das den Lizenznehmer zum Rücktritt. Bei Zahlungen jeder Art zum Beispiel als Schadenersatz erhält der Lizenznehmer dann einen Anteil von der Lizenzgebühr nach § 3 des Vertrages, wenn der Rechtsinhaber ihn in der Ausübung seiner Lizenz verletzte.
e) Für nicht käuflich erworbene Software entsteht die Haftung des Lizenzgebers aufgrund von Schutzrechten Dritter entsprechend den Voraussetzungen des Vertrages nur im Fall grober Fahrlässigkeit oder bei Vorsatz.

9. Schlussbestimmungen
a) Zu einer Abtretung seiner Rechte aus diesem Vertrag bedarf der Lizenznehmer der schriftlichen Einwilligung des Lizenzgebers.
b) Eine Aufrechnung gegen die Forderung nach Lizenzgebühr kann der Lizenznehmer nur mit anerkannten oder rechtskräftig festgestellten Forderungen erklären.
c) Wenn in dieser Lizenz Regelungen fehlen, gelten diesbezüglich ergänzend zu dieser Lizenz die Allgemeinen Geschäftsbedingungen (AGB) des Lizenzgebers. Ansonsten enthält der Vertrag enthält alle getroffenen Vereinbarungen. Weitere schriftliche oder mündliche Nebenabreden bestehen nicht. Änderungen und Ergänzungen bedürfen der Schriftform.

d) Die Rechtsunwirksamkeit einer Bestimmung berührt die Rechtswirksamkeit der anderen Vertragsteile nicht. Die Vertragsparteien verpflichten sich, eine unwirksame Bestimmung durch eine wirksame Regelung zu ersetzen, die ihr im wirtschaftlichen Ergebnis am nächsten kommt und dem Vertragszweck am besten entspricht.
e) Erfüllungsort ist der Sitz des Lizenzgebers.
f) Gerichtsstand für alle Streitigkeiten, die sich aus oder im Zusammenhang mit dieser Lizenz ergeben, ist, soweit gemäß § 38 ZPO vereinbar, der Sitz des Lizenzgebers. Es gilt ausschließlich deutsches Recht, UN-Kaufrecht ist ausgeschlossen.

Stand: 16.05.2007