

emWin 8051

**Graphic Library with
Graphical User Interface
for 8051 processors**

Version 4.00

Manual Rev. 0



**www.segger.com
solutions for embedded software**

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER SYSTEME GmbH (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of the manufacturer. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2008 SEGGER Microcontroller GmbH & Co. KG, Hilden / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available.

For registration, please provide the following:

- Company name and address
- Your name
- Your job title
- Your email address and telephone number
- Name and version of the product

Please send this information to: register@segger.com

Contact address

SEGGER Microcontroller GmbH & Co. KG
 In den Weiden 11
 D-40721 Hilden
 Germany
 Tel. +49 2103-2878-0
 Fax. +49 2103-2878-28
 E-mail: support@segger.com
 Internet: <http://www.segger.com>

Manual versions

This manual describes the latest software version. If any error occurs, please inform us and we will try to assist you as soon as possible.

For further information on topics or routines not yet specified, please contact us.

Print date: 6/22/09

Manual version	Date	By	Explanation
4.00R0	090618	AS	Initial version



SEGGER Microcontroller Systeme GmbH develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development-time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash microcontrollers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:

<http://www.segger.com>

United States Office:

<http://www.segger-us.com>

EMBEDDED SOFTWARE (Middleware)



emWin

Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.



embOS

Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.



emFile

File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.



USB-Stack

USB device stack

A USB stack designed to work on any embedded system with a USB client controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher

Flash programmer

Flash Programming tool primarily for microcontrollers.

J-Link

JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace

JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.



Table of Contents

1	Introduction to emWin 8051	11
1.1	Purpose of this document	12
1.2	Assumptions	12
1.3	Differences between emWin and emWin 8051	12
1.4	Requirements.....	12
1.4.1	Target system (hardware)	12
1.4.2	Development environment (compiler).....	12
1.5	Features of emWin 8051.....	13
1.6	Samples and demos.....	14
1.7	How to use this manual	14
1.8	Typographic conventions for syntax	14
1.9	Screen and coordinates	14
1.10	How to connect the LCD to the microcontroller	15
1.11	Data types.....	16
2	Getting Started.....	17
2.1	Recommended directory structure.....	18
2.1.1	Subdirectories.....	18
2.1.2	Include directories	18
2.2	Adding emWin to the target program	18
2.3	Creating a library.....	19
2.3.1	Adapting the library batch files to a different system	19
2.4	"C" files to include in the project.....	21
2.5	Configuring emWin	21
2.6	Initializing emWin.....	22
2.7	Using emWin with target hardware	22
2.8	The "Hello world" sample program	23
3	Simulator.....	25
3.1	Using the simulator.....	26
3.1.1	Using the simulator in the trial emWin version	26
3.1.1.1	Directory structure.....	26
3.1.1.2	Visual C++ workspace.....	26
3.1.1.3	Compiling the demo program	26
3.1.1.4	Compiling the samples	27
3.1.2	Using the simulator with the emWin source	28
3.1.2.1	Directory structure.....	28
3.1.2.2	Visual C++ workspace.....	28
3.1.2.3	Compiling the application.....	28
3.2	Device simulation	30
3.2.1	Device simulator API.....	30
3.2.2	Hardkey simulation	33
3.2.2.1	Hardkey simulator API.....	34
3.3	Integrating the emWin simulation into an existing simulation.....	37
3.3.1	Directory structure.....	37
3.3.2	Using the simulation library.....	37
3.3.2.1	Modifying WinMain.....	37
3.3.2.2	Sample application	37
3.3.3	Integration into the embOS Simulation	39
3.3.3.1	WinMain	39

3.3.3.2	Target program (main)	39
3.3.4	GUI simulation API	41
4	Viewer	45
4.1	Using the viewer	46
4.1.1	Using the simulator and the viewer	46
4.1.2	Using the viewer with virtual pages	47
4.1.3	Always on top	47
4.1.4	Open further windows of the display output	47
4.1.5	Zooming	47
4.1.6	Copy the output to the clipboard	48
4.1.7	Using the viewer with multiple displays	48
4.1.8	Using the viewer with multiple layers	49
5	Displaying Text	51
5.1	Basic routines	52
5.2	Text API	52
5.3	Routines to display text	53
5.4	Selecting text drawing modes	58
5.5	Selecting text alignment	60
5.6	Setting the current text position	61
5.7	Retrieving the current text position	62
5.8	Routines to clear a window or parts of it	62
6	Displaying Values	65
6.1	Value API	66
6.2	Displaying decimal values	66
6.3	Displaying floating-point values	70
6.4	Displaying binary values	72
6.5	Displaying hexadecimal values	73
6.6	Version of emWin	74
7	2-D Graphic Library	75
7.1	Graphic API	76
7.2	Drawing modes	77
7.3	Query current client rectangle	78
7.4	Pen size	79
7.5	Basic drawing routines	79
7.6	Drawing bitmaps	83
7.7	Drawing lines	85
7.8	Drawing polygons	89
7.9	Drawing circles	93
7.10	Drawing ellipses	94
7.11	Drawing arcs	95
7.12	Drawing graphs	96
7.13	Drawing pie charts	97
7.14	Saving and restoring the GUI-context	98
7.15	Clipping	98
8	Fonts	101
8.1	Introduction	102
8.2	Font types	102
8.3	Font formats	102
8.3.1	'C' file format	102
8.4	Declaring custom fonts	102
8.5	Selection of a font	103
8.6	Font API	103
8.7	'C' file related font functions	103
8.8	Common font-related functions	104

8.9	Character sets.....	107
8.9.1	ASCII.....	107
8.9.2	ISO 8859-1 Western Latin character set	107
8.9.3	Unicode.....	109
8.10	Font converter	109
8.10.1	Adding fonts	110
8.11	Standard fonts	110
8.11.1	Font identifier naming convention	110
8.11.2	Font file naming convention	111
8.11.3	Measurement, ROM-size and character set of fonts	112
8.11.4	Proportional fonts	113
8.11.4.1	Overview.....	113
8.11.4.2	Measurement, ROM size and used files	114
8.11.4.3	Characters.....	115
8.11.5	Monospaced fonts.....	123
8.11.5.1	Overview.....	123
8.11.5.2	Measurement, ROM size and used files	123
8.11.5.3	Characters.....	124
8.11.6	Digit fonts (proportional)	128
8.11.6.1	Overview.....	128
8.11.6.2	Measurement, ROM size and used files	128
8.11.6.3	Characters.....	128
8.11.7	Digit fonts (monospaced).....	130
8.11.7.1	Overview.....	130
8.11.7.2	Measurement, ROM size and used files	130
8.11.7.3	Characters.....	130
9	Bitmap Converter	133
9.1	What it does	134
9.2	Loading a bitmap.....	134
9.2.1	Supported file formats.....	134
9.2.2	Loading from a file.....	134
9.2.3	Using the clipboard	134
9.3	Generating "C" files from bitmaps	135
9.3.1	Supported bitmap formats	135
9.3.2	Palette information	135
9.3.3	Transparency	136
9.3.4	Alpha blending	136
9.3.5	Selecting the best format.....	137
9.3.6	Saving the file.....	138
9.4	Color conversion.....	139
9.5	Compressed bitmaps.....	140
9.6	Using a custom palette	140
9.6.1	Saving a palette file	141
9.6.2	Palette file format.....	141
9.6.3	Palette files for fixed palette modes.....	141
9.6.4	Converting a bitmap	141
9.7	Command line usage.....	142
9.7.1	Format for commands	142
9.7.2	Valid command line options.....	142
9.8	Example of a converted bitmap	144
10	Colors.....	147
10.1	Predefined colors	148
10.2	The color bar test routine	148
10.3	Fixed palette modes.....	149
10.4	Default fixed palette modes	150
10.5	Detailed fixed palette mode description	151
10.6	Custom palette modes.....	160
10.7	Modifying the color lookup table at run time	160

10.8	Color API	160
10.9	Basic color functions	161
10.10	Index & color conversion	163
10.11	Lookup table (LUT) group	164
11	Execution Model: Single Task / Multitask	167
11.1	Supported execution models	168
11.2	Single task system (superloop)	168
11.2.1	Description	168
11.2.2	Superloop example (without emWin)	168
11.2.3	Advantages	168
11.2.4	Disadvantages	168
11.2.5	Using emWin	168
11.2.6	Superloop example (with emWin)	169
11.3	Multitask system: one task calling emWin	169
11.3.1	Description	169
11.3.2	Advantages	169
11.3.3	Disadvantages	169
11.3.4	Using emWin	169
11.4	Multitask system: multiple tasks calling emWin	170
11.4.1	Description	170
11.4.2	Advantages	170
11.4.3	Disadvantages	170
11.4.4	Using emWin	170
11.4.5	Recommendations	170
11.4.6	Example	170
11.5	GUI configuration macros for multitasking support	171
11.6	Kernel interface routine API	172
12	Virtual screen / Virtual pages	177
12.1	Introduction	178
12.2	Requirements	178
12.3	Configuration	179
12.3.1	Sample configuration	179
12.4	Samples	180
12.4.1	Basic sample	180
12.5	Virtual screen API	181
13	Keyboard Input	185
13.1	Description	186
13.1.1	Driver layer API	186
13.1.2	Application layer API	187
14	Foreign Language Support	189
14.1	Unicode	190
14.1.1	UTF-8 encoding	190
14.1.2	Unicode characters	190
14.1.3	UTF-8 strings	191
14.1.3.1	Using U2C.exe to convert UTF-8 text into "C"-code	191
14.1.4	Unicode API	192
14.1.4.1	UTF-8 functions	192
14.1.4.2	Double byte functions	194
15	Display drivers	199
15.1	Available drivers and supported display controllers	200
15.2	CPU / Display controller interface	202
15.2.1	Full bus interface	202
15.2.2	Simple bus interface	203
15.2.3	4 pin SPI interface	204

15.2.4	3 pin SPI interface	204
15.2.5	I2C bus interface	205
15.2.6	Non readable displays	205
15.3	Detailed display driver descriptions	205
15.3.1	LCDLin driver	205
15.3.1.1	LCDLin driver (32/16/8 bit access).....	206
15.3.1.2	LCDLin driver (32 bit access).....	207
15.3.1.3	LCDLin driver (8 and 16 bit access).....	212
15.3.2	LCD667XX driver	216
15.3.3	LCDTemplate driver	217
15.3.4	LCDNull driver.....	218
15.4	LCD layer and display driver API	218
15.4.1	Display driver API.....	218
15.4.2	Driver routines.....	220
15.4.2.1	Init & display control group.....	220
15.4.2.2	Drawing group	220
15.4.2.3	"Get" group	222
15.4.2.4	Lookup table (LUT) group	223
15.4.2.5	Miscellaneous group.....	223
15.4.3	Callback routines.....	224
15.4.4	LCD layer routines	224
15.4.4.1	"Get" group	224
16	Timing and Execution-Related Functions	229
16.1	Timing and execution API	230
17	Low-Level Configuration (LCDConf.h).....	231
17.1	Available configuration macros	232
17.2	General (required) configuration.....	234
17.3	Initialisation of the controller.....	235
17.4	Display orientation.....	236
17.5	Color configuration	238
17.6	Simple bus interface configuration	239
17.6.1	Macros used by a simple bus interface.....	239
17.6.2	Example of memory mapped interface.....	240
17.6.3	Sample routines for connection to I/O pins.....	241
17.7	3 pin SPI configuration	242
17.7.1	Macros used by a 3 pin SPI interface	242
17.7.2	Sample routines for connection to I/O pins.....	242
17.8	4 pin SPI configuration	243
17.8.1	Macros used by a 4 pin SPI interface	243
17.8.2	Sample routines for connection to I/O pins.....	243
17.9	I2C bus interface configuration	245
17.9.1	Macros used by a I2C bus interface	245
17.9.2	Sample routines for connection to I/O pins.....	246
17.10	Full bus interface configuration	247
17.10.1	Macros used by a full bus interface	247
17.10.2	Configuration example	249
17.11	Virtual display support.....	251
17.12	LCD controller configuration: COM/SEG lines	252
17.13	Configuring multiple display controllers.....	253
17.13.1	Macros used by the distribution layer	253
17.13.2	Hardware access	253
17.13.3	COM/SEG line configuration	253
17.13.4	Configuration example	254
17.14	COM/SEG lookup tables.....	255
17.15	Miscellaneous.....	256
18	High-Level Configuration (GUIConf.h).....	259
18.1	General notes	260

18.2	How to configure the GUI	260
18.2.1	Sample configuration	260
18.3	Available GUI configuration macros.....	260
18.3.1	GUI_MEMCPY.....	261
18.3.2	GUI_MEMSET.....	261
18.3.3	GUI_TRIAL_VERSION	261
18.4	Runtime configuration.....	262
18.4.1	Memory requirements.....	262
18.4.2	Available GUI configuration routines	263
18.5	Runtime configuration.....	264
18.6	GUI_X routine reference.....	264
18.6.1	Init routines	265
18.6.2	Timing routines.....	265
18.6.3	Kernel interface routines	266
18.7	Debugging	266
18.8	Dynamic memory	267
18.9	Special considerations for certain Compilers/CPUs	269
18.9.1	AVR with IAR-Compiler	269
18.9.2	8051 Keil compiler and other 8-bit CPU compilers.....	269
19	Performance and Resource Usage.....	271
19.1	Memory requirements.....	272
19.1.1	Memory requirements of the GUI components.....	272
19.1.2	Stack requirements	272
20	Support	273
20.1	Problems with tool chain (compiler, linker)	274
20.1.1	Compiler crash.....	274
20.1.2	Compiler warnings.....	274
20.1.3	Linker problems	274
20.2	Problems with hardware/driver	275
20.3	Problems with API functions.....	275
20.4	Problems with the performance	275
20.5	Contacting support.....	276
20.6	FAQ's	276

Chapter 1

Introduction to emWin 8051

The following chapter introduces emWin 8051 and gives basic information about its purpose and usage.

1.1 Purpose of this document

This guide describes how to install, configure and use the emWin graphical user interface for embedded applications. It also explains the internal structure of the software.

1.2 Assumptions

This guide assumes that you already possess a solid knowledge of the "C" programming language. If you feel that your knowledge of "C" is not sufficient, we recommend *The "C" Programming Language* by Kernighan and Richie, which describes the programming standard and, in newer editions, also covers the ANSI "C" standard. Knowledge of assembly programming is not required.

1.3 Differences between emWin and emWin 8051

emWin 8051 is designed to develop software for 8051 based CPUs with the Keil 8051 compiler. This version does not include features like the window manager, the widget library and memory devices which are supported by emWin (full version).

If you would like to develop your software for other target systems using emWin, don't hesitate to contact us or visit our website www.segger.com, where you can find the latest emWin documentation and sample applications.

1.4 Requirements

A target system is not required in order to develop software with emWin; most of the software can be developed using the simulator. However, the final purpose is usually to be able to run the software on a target system.

1.4.1 Target system (hardware)

Your target system must:

- Have a 8051 CPU
- Have a minimum of RAM and ROM
- Have a full graphic LCD (any type and any resolution)

The memory requirements vary depending on which parts of the software are used and how efficient your target compiler is. It is therefore not possible to specify precise values, but the following apply to typical systems.

Small systems (no window manager)

- RAM: 100 bytes
- Stack: 600 bytes
- ROM: 10-25 kb (depending on the functionality used)

Note that ROM requirements will increase if your application uses many fonts. All values are rough estimates and cannot be guaranteed.

1.4.2 Development environment (compiler)

emWin 8051 is designed to develop software only for 8051 based CPUs with the Keil 8051 compiler.

1.5 Features of emWin 8051

emWin is designed to provide an efficient, processor- and LCD controller-independent graphical user interface for any application that operates with a graphical LCD. It is compatible with single-task and multitask environments, with a proprietary operating system or with any commercial RTOS. emWin is shipped as "C" source or library code. It may be adapted to any size physical and virtual display with any LCD controller and CPU. Its features include the following:

General

- Any (monochrome, grayscale or color) LCD with any controller supported (if the right driver is available).
- May work without LCD controller on smaller displays.
- Any interface supported using configuration macros.
- Display-size configurable.
- Characters and bitmaps may be written at any point on the LCD, not just on even-numbered byte addresses.
- Routines are optimized for both size and speed.
- Compile time switches allow for different optimizations.
- For slower LCD controllers, LCD can be cached in memory, reducing access to a minimum and resulting in very high speed.
- Clear structure.
- Virtual display support; the virtual display can be larger than the actual display.

Graphic library

- Bitmaps of different color depths supported.
- Bitmap converter available.
- Absolutely no floating-point usage.
- Fast line/point drawing (without floating-point usage).
- Very fast drawing of circles/polygons.
- Different drawing modes.

Fonts

- A variety of different fonts are shipped with the basic software: 4*6, 6*8, 6*9, 8*8, 8*9, 8*16, 8*17, 8*18, 24*32, and proportional fonts with pixel-heights of 8, 10, 13, 16.
- New fonts can be defined and simply linked in.
- Only the fonts used by the application are actually linked to the resulting executable, resulting in minimum ROM usage.
- Fonts are fully scalable, separately in X and Y.
- Font converter available; any font available on your host system (i.e. Microsoft Windows) can be converted.

String/value output routines

- Routines to show values in decimal, binary, hexadecimal, any font.
- Routines to edit values in decimal, binary, hexadecimal, any font.

Touch-screen & mouse support

- For window objects such as the button widget, emWin offers touch-screen and mouse support.

PC tools

- Simulation plus viewer.
- Bitmap converter.
- Font converter.

1.6 Samples and demos

To give you a better idea of what emWin can do, we have different demos available as "ready-to-use" simulation executables under `Sample\EXE`. The source of the sample programs is located in the folder `Sample`. The folder `Sample\GUIDemo` contains an application program showing most of emWin's features. All samples are also available at www.segger.com.

1.7 How to use this manual

This manual explains how to install, configure and use emWin. It describes the internal structure of the software and all the functions that emWin offers (the Application Program Interface, or API). Before actually using emWin, you should read or at least glance through this manual in order to become familiar with the software. The following steps are then recommended:

- Copy the emWin files to your computer.
- Go through Chapter 2: "Getting Started".
- Use the simulator in order to become more familiar with what the software can do (refer to Chapter 3: "Simulator").
- Expand your program using the rest of the manual for reference.

1.8 Typographic conventions for syntax

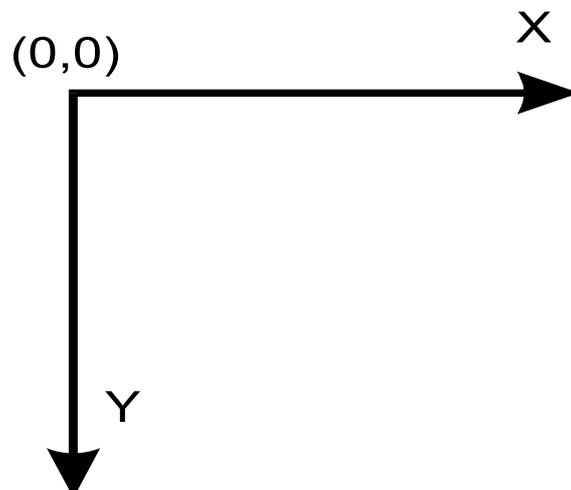
This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
Keyword	Text that you enter at the command-prompt or that appears on the display (i.e. system functions, file- or pathnames).
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
New Sample	Sample code that has been added to an existing program example.
Warning	Important cautions or reminders.

1.9 Screen and coordinates

The screen consists of many dots that can be controlled individually. These dots are called pixels. Most of the text and drawing functions that emWin offers in its API to the user program can write or draw on any specified pixel.

The horizontal scale is called the X-axis, whereas the vertical scale is called the Y-axis. Coordinates are denoted as a pair consisting of an X- and a Y-value (X, Y). The X-coordinate is always first in routines that require X and Y coordinates. The upper left corner of the display (or a window) has per default the coordinates (0,0). Positive



X-values are always to the right; positive Y-values are always down. The above graph illustrates the coordinate system and directions of the X- and Y- axes. All coordinates passed to an API function are always specified in pixels.

1.10 How to connect the LCD to the microcontroller

emWin handles all access to the LCD. Virtually any LCD controller can be supported, independently of how it is accessed. For details, please refer to Chapter 17: "Low-Level Configuration". Also, please get in contact with us if your LCD controller is not supported. We are currently writing drivers for all LCD controllers available on the market and may already have a proven driver for the LCD controller that you intend to use. It is usually very simple to write the routines (or macros) used to access the LCD in your application. SEGGER Microcontroller Systeme GmbH offers the service of making these customizations for you, if necessary with your target hardware.

It does not really matter how the LCD is connected to the system as long as it is somehow accessible by software, which may be accomplished in a variety of ways. Most of these interfaces are supported by a driver which is supplied in source code form. This driver does not normally require modifications, but is configured for your hardware by making changes in the file `LCDConf.h`. Details about how to customize a driver to your hardware as necessary are explained in Chapter 15: "LCD Drivers". The most common ways to access the LCD are described as follows. If you simply want to understand how to use emWin, you may skip this section.

LCD with memory-mapped LCD controller:

The LCD controller is connected directly to the data bus of the system, which means the controller can be accessed just like a RAM. This is a very efficient way of accessing the LCD controller and is most recommended. The LCD addresses are defined to the segment `LCDSEG`, and in order to be able to access the LCD the linker/locator simply needs to be told where to locate this segment. The location must be identical to the access address in physical address space. Drivers are available for this type of interface and for different LCD controllers.

LCD with LCD controller connected to port / buffer

For slower LCD controllers used on fast processors, the use of port-lines may be the only solution. This method of accessing the LCD has the disadvantage of being somewhat slower than direct bus-interface but, particularly with a cache that minimizes the accesses to the LCD, the LCD update is not slowed down significantly. All that needs to be done is to define routines or macros which set or read the hardware ports/buffers that the LCD is connected to. This type of interface is also supported by different drivers for the different LCD controllers.

Proprietary solutions: LCD without LCD controller

The LCD can also be connected without an LCD controller. In this case, the LCD data is usually supplied directly by the controller via a 4- or 8-bit shift register. These proprietary hardware solutions have the advantage of being inexpensive, but the disadvantage of using up much of the available computation time. Depending on the CPU, this can be anything between 20 and almost 100 percent; with slower CPUs, it is really not possible at all. This type of interface does not require a specific LCD driver because emWin simply places all the display data into the LCD cache. You yourself must write the hardware-dependent portion that periodically transfers the data in the cache memory to your LCD.

Sample code for transferring the video image into the display is available in both "C" and optimized assembler for M16C and M16C/80.

1.11 Data types

Since "C" does not provide data types of fixed lengths which are identical on all platforms, emWin uses, in most cases, its own data types as shown in the table below:

Data type	Definition	Explanation
I8	signed char	8-bit signed value
U8	unsigned char	8-bit unsigned value
I16	signed short	16-bit signed value
U16	unsigned short	16-bit unsigned value
I32	signed long	32-bit signed value
U32	unsigned long	32-bit unsigned value
I16P	signed short	16-bit (or more) signed value
U16P	unsigned short	16-bit (or more) unsigned value

For most 16/32-bit controllers, the settings will work fine. However, if you have similar defines in other sections of your program, you might want to change or relocate them. A recommended place is in the file `Global.h`.

Chapter 2

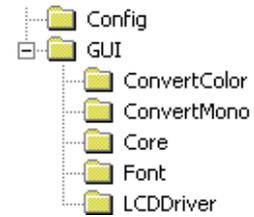
Getting Started

The following chapter provides an overview of the basic procedures for setting up and configuring emWin on your target system. It also includes a simple program example.

If you find yourself unsure about certain areas, keep in mind that most topics are treated in greater detail in later chapters. You will most likely need to refer to other parts of the manual before you begin more complicated programming.

2.1 Recommended directory structure

We recommend keeping emWin separate from your application files. It is good practice to keep all the program files (including the header files) together in the GUI subdirectories of your project's root directory. The directory structure should be similar to the one pictured on the right. This practice has the advantage of being very easy to update to newer versions of emWin by simply replacing the GUI\ directories. Your application files can be stored anywhere.



2.1.1 Subdirectories

The following table shows the contents of all GUI subdirectories:

Directory	Contents
Config	Configuration files
GUI\ConvertMono	Color conversion routines used for grayscale displays *
GUI\ConvertColor	Color conversion routines used for color displays *
GUI\Core	emWin core files
GUI\Font	Font files
GUI\LCDDriver	LCD driver

(* = optional)

2.1.2 Include directories

You should make sure that the include path contains the following directories (the order of inclusion is of no importance):

- Config
- GUI\Core

Warning: Always make sure that you have only one version of each file!

It is frequently a major problem when updating to a new version of emWin if you have old files included and therefore mix different versions. If you keep emWin in the directories as suggested (and only in these), this type of problem cannot occur. When updating to a newer version, you should be able to keep your configuration files and leave them unchanged. For safety reasons, we recommend backing (or at least renaming) the GUI\ directories prior to updating.

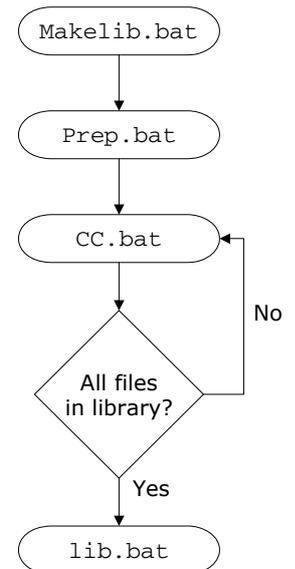
2.2 Adding emWin to the target program

You basically have a choice between including only the source files that you are actually going to use in your project, which will then be compiled and linked, or creating a library and linking the library file. If your tool chain supports "smart" linking (linking in only the modules that are referenced and not those that are not referenced), there is no real need to create a library at all, since only the functions and data structures which are required will be linked. If your tool chain does not support "smart" linking, a library makes sense, because otherwise everything will be linked in and the program size will be excessively large. For some CPUs, we have sample projects available to help you get started.

2.3 Creating a library

Building a library from the sources is a simple procedure. The first step is to copy the batch files (located under `Sample\Makelib`) into your root directory. Then, make any necessary changes. There are a total of four batch files which need to be copied, described in the table below. The main file, `Makelib.bat`, will be the same for all systems and requires no changes. To build a library for your target system, you will normally need to make slight modifications to the other three smaller files. Finally, start the file `Makelib.bat` to create the library. The batch files assume that your `GUI` and `Config` subdirectories are set up as recommended.

The procedure for creating a library is illustrated in the flow chart to the right. The `Makelib.bat` file first calls `Prep.bat` to prepare the environment for the tool chain. Then it calls `CC.bat` for every file to be included in the library. It does this as many times as necessary. `CC.bat` adds each object file to a list that will be used by `lib.bat`. When all files to be added to the library have been listed, `Makelib.bat` then calls `lib.bat`, which uses a librarian to put the listed object files into the actual library. Of course you are free to create libraries an other way.



File	Explanation
Makelib.bat	Main batch file. No modification required.
Prep.bat	Called by Makelib.bat to prepare environment for the tool chain to be used,
CC.bat	Called by Makelib.bat for every file to be added to the library; creates a list of these object files which will then be used in the next step by the librarian in the lib.bat file.
lib.bat	Called by Makelib.bat to put the object files listed by CC.bat into a library.

The files as shipped assume that a Microsoft compiler is installed in its default location. If all batch files are copied to the root directory (directly above `GUI`) and no changes are made at all, a simulation library will be generated for the `emWin` simulation. In order to create a target library, however, it will be necessary to modify `Prep.bat`, `CC.bat`, and `lib.bat`.

2.3.1 Adapting the library batch files to a different system

The following will show how to adapt the files by a sample adaptation for a Mitsubishi M32C CPU.

Adapting Prep.bat

`Prep.bat` is called at the beginning of `Makelib.bat`. As described above its job is to set the environment variables for the used tools and the environment variable `PATH`, so that the batch files can call the tools without specifying an absolute path. Assuming the compiler is installed in the folder `C:\MTOOL` the file `Prep.bat` could look as follows:

```

@ECHO OFF
SET TOOLPATH=C:\MTOOL

REM *****
REM   Set the variable PATH to be able to call the tools

SET PATH=%TOOLPATH%\BIN;%TOOLPATH%\LIB308;%PATH%

REM *****
REM   Set the tool internal used variables

SET BIN308=%TOOLPATH%\BIN
SET INC308=%TOOLPATH%\INC308
  
```

```
SET LIB308=%TOOLPATH%\LIB308
SET TMP308=%TOOLPATH%\TMP
```

Adapting CC.bat

The job of `CC.bat` is to compile the passed source file and adding the file name of the object file to a link list. When starting `MakeLib.bat` it creates the following subdirectories relative to its position:

Directory	Contents
Lib	This folder should contain the library file after the build process.
Temp\Output	Should contain all the compiler output and the link list file. Will be deleted after the build process.
Temp\Source	<code>MakeLib.bat</code> uses this folder to copy all source and header files used for the build process. Will be deleted after the build process.

The object file should be created (or moved) to `Temp\Output`. This makes sure all the output will be deleted after the build process. Also the link list should be located in the output folder. The following shows a sample for the Mitsubishi compiler:

```
@ECHO OFF
GOTO START
REM *****
REM   Explanation of the used compiler options:

-silent : Suppresses the copyright message display at startup
-M82    : Generates object code for M32C/80 Series (Remove this switch
         for M16C80 targets)
-c      : Creates a relocatable file (extension .r30) and ends processing
-I      : Specifies the directory containing the file(s) specified in #include
-dir    : Specifies the destination directory
-OS     : Maximum optimization of speed followed by ROM size
-fFRAM  : Changes the default attribute of RAM data to far
-fETI   : Performs operation after extending char-type data to the int type
         (Extended according to ANSI standards)

:START

REM *****
REM   Compile the passed source file with the Mitsubishi NC308 compiler

NC308 -silent -M82 -c -IInc -dir Temp\Output -OS -fFRAM -fETI Temp\Source\%1.c

REM *****
REM   Pause if any problem occurs

IF ERRORLEVEL 1 PAUSE

REM *****
REM   Add the file name of the object file to the link list

ECHO Temp\Output\%1.R30>>Temp\Output\Lib.dat
```

Adapting Lib.bat

After all source files have been compiled `Lib.bat` will be called from `MakeLib.bat`. The job is to create a library file using the link list created by `CC.bat`. The destination folder of the library file should be the `Lib` folder created by `MakeLib.bat`. The following shows a sample for the Mitsubishi librarian:

```
@ECHO OFF
GOTO START
REM *****
REM   Explanation of the used options:

-C : Creates new library file
@  : Specifies command file

:START

REM *****
REM   Create the first part of the linker command file

ECHO -C Lib\GUI>Temp\Output\PARA.DAT
```

```

REM *****
REM   Merge the first part with the link list to the linker command file

COPY Temp\Output\PARA.DAT+Temp\Output\Lib.dat Temp\Output\LINK.DAT

REM *****
REM   Call the Mitsubishi librarian

LB308 @Temp\Output\LINK.DAT

REM *****
REM   Pause if any problem occurs

IF ERRORLEVEL 1 PAUSE

```

2.4 "C" files to include in the project

Generally speaking, you need to include the core "C" files of emWin, the LCD driver, all font files you plan to use and any optional modules you have ordered with emWin:

- All "C" files of the folder `GUI\Core`
- The fonts you plan to use (located in `GUI\Font`)
- LCD driver: All "C" files of the folder `GUI\LCDDriver` and `Config\LCDConf.c`.
- All "C" files of the folder `GUI\JPEG` (only if you need JPEG support)

Additional software packages

If you plan to use additional, optional modules you must also include their "C" files:

- Gray scale converting functions: all "C" files located in `GUI\ConvertMono`
- Color conversion functions: all "C" files located in `GUI\ConvertColor`

Target specifics

- `GUI_X.c`. A sample file is available as `Sample\GUI_X`. This file contains the hardware-dependent part of emWin and should be modified as described in Chapter 18: "High-Level Configuration".
- For port/buffer-accessed LCDs, interface routines must be defined. Samples of the required routines are available under `Samples\LCD_X` or on our website in the download area.

Be sure that you include `GUI.h` in all of your source files that access emWin.

2.5 Configuring emWin

The `Config` folder should contain the configuration files matching your order. The file `LCDConf.h` contains all the definitions necessary to adapt emWin to your LCD. The file `LCDConf.c` contains the initialization routine of the display controller, which is the main task when configuring emWin. For details, please see Chapter 17: "Low-Level Configuration".

If emWin is not configured correctly, because you did not select the right display resolution or chose the wrong LCD controller, it will probably not display anything at all or display something that does not resemble what you expected. So take care to tailor `LCDConf.h` and `LCDConf.c` to your needs.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, where 0 means deactivated and 1 means activated (actually anything other than 0 would work, but using 1 makes it easier to read a `config` file). These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macro.

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant. Typical examples are in the configuration of the resolution of an LCD.

Selection switches "S"

Selection switches are used to select one out of multiple options where only one of those options can be selected. A typical example might be the selection of the type of LCD controller used, where the number selected denotes which source code (in which LCD driver) is used to generate object code.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, in which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module (such as the access to an LCD) which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

2.6 Initializing emWin

The routine `GUI_Init()` initializes the LCD and the internal data structures of emWin, and must be called before any other emWin function. This is done by placing the following line into the init sequence of your program:

```
GUI_Init();
```

If this call is left out, the entire graphics system will not be initialized and will therefore not be ready. For more details about the initialization process please refer to Chapter 18: "High-Level Configuration".

2.7 Using emWin with target hardware

The following is just a basic outline of the general steps that should be taken when starting to program with emWin. All steps are explained further in subsequent chapters.

Step 1: Customizing emWin

The first step is usually to customize emWin by modifying the header file `LCDCConf.h` and the display controller initialization in `LCDCConf.c`.

Step 2: Defining access addresses or access routines

For memory-mapped LCDs, the access addresses of the LCD simply need to be defined in `LCDCConf.h`. For port/buffer-accessed LCDs, interface routines must be defined. Samples of the required routines are available under `Samples\LCD_X` or on our website in the download area.

Step 3: Compiling, linking and testing the sample code

emWin comes with sample code for both single- and multitask environments. Compile, link and test these little sample programs until you feel comfortable doing so.

Step 4: Modifying the sample program

Make simple modifications to the sample programs. Add additional commands such as displaying text in different sizes on the display, showing lines and so on.

Step 5: In multitask applications: adapt to your OS (if necessary)

If multiple tasks should be able to access the display simultaneously, the macros `GUI_MAXTASK` and `GUI_OS` come into play, as well as the file `GUITask.c`. For details and sample adaptations, please refer to Chapter 18: "High-Level Configuration".

Step 6: Write your own application using emWin

By now you should have a clearer understanding of how to use emWin. Think about how to structure the program your application requires and use emWin by calling the appropriate routines. Consult the reference chapters later in this manual, as they discuss the specific emWin functions and configuration macros that are available.

2.8 The "Hello world" sample program

A "Hello world" program has been used as a starting point for "C" programming since the early days, because it is essentially the smallest program that can be written. A "Hello world" program with emWin, called `HELLO.c`, is shown below and is available as `BASIC>HelloWorld.c` in the sample shipped with emWin.

The whole purpose of the program is to write "Hello world" in the upper left corner of the display. In order to be able to do this, the hardware of the application, the LCD and the GUI must first be initialized. emWin is initialized by a call to `GUI_Init()` at the start of the program, as described previously. In this example, we assume that the hardware of your application is already initialized.

The "Hello world" program looks as follows:

```

/*****
*                               SEGGGER MICROCONTROLLER SYSTEME GmbH                               *
*                               Solutions for real time microcontroller applications                 *
*                               emWin sample code                                             *
*                               ****                                                              *
*****
-----
File       : BASIC>HelloWorld.c
Purpose    : Simple demo drawing "Hello world"
-----
*/

#include "GUI.H"

/*****
*                               main                                                            *
*                               ****                                                              *
*****
*/

void MainTask(void) {
/*
ToDo: Make sure hardware is initialized first!!
*/
    GUI_Init();
    GUI_DispString("Hello world!");
    while(1);
}

```

Adding functionality to the "Hello world" program

Our little program has not been doing too much so far. We can now extend the functionality a bit: after displaying "Hello world", we would like the program to start counting on the display in order to be able to estimate how fast outputs to the LCD can be made. We can simply add a bit of code to the loop at the end of the main program, which is essentially a call to the function that displays a value in decimal form. The example is available as `BASIC>Hello1.c` in the sample folder.

```

/*****
*                               SEGGGER MICROCONTROLLER SYSTEME GmbH                               *
*                               Solutions for real time microcontroller applications                 *
*                               emWin sample code                                             *
*                               ****                                                              *
*****
-----
File       : BASIC>Hello1.c

```

```
Purpose      : Simple demo drawing "Hello world"
-----
*/

#include "GUI.H"

/*****
 *
 *          main
 *
 *****/

void MainTask(void) {
    int i=0;
    /*
    ToDo:  Make sure hardware is initialized first!!
    */
    GUI_Init();
    GUI_DispString("Hello world!");
    while(1) {
        GUI_DispDecAt( i++, 20,20,4);
        if (i>9999) i=0;
    }
}
```

Chapter 3

Simulator

The PC simulation of emWin allows you to compile the same "C" source on your Windows PC using a native (typically Microsoft) compiler and create an executable for your own application. Doing so allows the following:

- Design of the user interface on your PC (no need for hardware!).
- Debugging of your user interface program.
- Creation of demos of your application, which can be used to discuss the user interface.

The resulting executable can be easily sent via email.



3.1 Using the simulator

The emWin simulator uses Microsoft Visual C++ (version 6.00 or higher) and the integrated development environment (IDE) which comes with it. You will see a simulation of your LCD on your PC screen, which will have the same resolution in X and Y and can display the exact same colors as your LCD once it has been properly configured. The entire graphic library API and window manager API of the simulation are identical to those on your target system; all functions will behave in the very same way as on the target hardware since the simulation uses the same "C" source code as the target system. The difference lies only in the lower level of the software: the LCD driver. Instead of using the actual LCD driver, the PC simulation uses a simulation driver which writes into a bitmap. The bitmap is then displayed on your screen using a second thread of the simulation. This second thread is invisible to the application; it behaves just as if the LCD routines were writing directly to the display.

3.1.1 Using the simulator in the trial emWin version

The trial version of emWin contains a full library which allows you to evaluate all available features of emWin. It also includes the emWin viewer (used for debugging applications), as well as demo versions of the font converter and the bitmap converter. Keep in mind that, being a trial version, you will not be able to change any configuration settings or view the source code, but you will still be able to become familiar with what emWin can do.

3.1.1.1 Directory structure

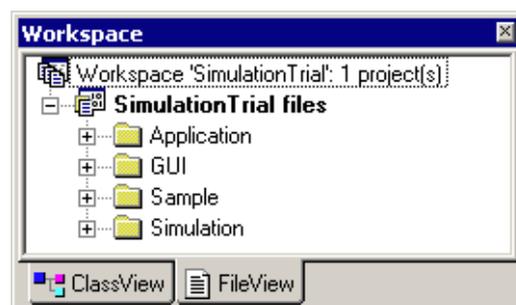
The directory structure of the simulator in the trial version will appear as pictured to the right. The table below explains the contents of the folders:

Directory	Contents
Application	Source of the demo program.
Config	configuration files used to build the library. Do not make any changes to these files!
Exe	Ready-to-use demo program.
GUI	Library files and include files needed to use the library.
Sample	Simulation samples and their sources.
Simulation	Files needed for the simulation.
Tool	The emWin viewer, a demo version of the bitmap converter and a demo version of the font converter.



3.1.1.2 Visual C++ workspace

The root directory shown above includes the Microsoft Visual C++ workspace (SimulationTrial.dsw) and project file (SimulationTrial.dsp). Double-click the workspace file to open the Microsoft IDE. The directory structure of the Visual C++ workspace will look like the one shown to the right.



3.1.1.3 Compiling the demo program

The source files for the demo program are located in the Application directory as a ready-to-go simulation, meaning that you need only to rebuild and start it. Please note that to rebuild the executable, you will need to have Microsoft Visual C++ (version 6.00 or later) installed.

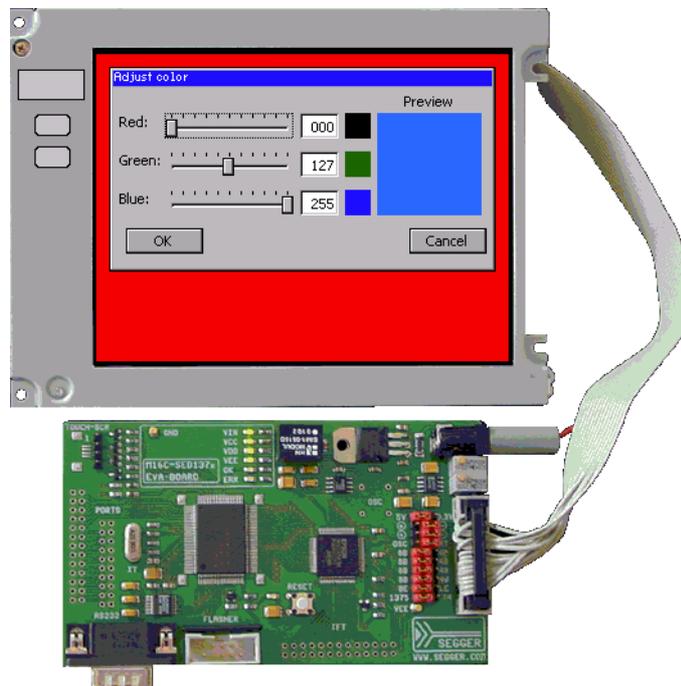
- Step 1: Open the Visual C++ workspace by double-clicking on `Simulation-Trial.dsw`.
- Step 2: Rebuild the project by choosing `Build/Rebuild All` from the menu (or by pressing F7).
- Step 3: Start the simulation by choosing `Build/Start Debug/Go` from the menu (or by pressing F5).

The demo project will begin to run and may be exited at any time by right-clicking on it and selecting `Exit`.

3.1.1.4 Compiling the samples

The `sample` directory contains ready-to-go samples that demonstrate different features of emWin and provide examples of some of their typical uses. In order to build any of these executables, their "C" source must be 'activated' in the project. This is easily done with the following procedure:

- Step 1: Exclude the `Application` folder from the build process by right-clicking the `Application` folder of the workspace and selecting 'Settings\General\Exclude from build'.
- Step 2: Open the sample folder of the workspace by double-clicking on it. Include the sample which should be used by right-clicking on it and deselecting 'Settings\General\Exclude from build'. The screenshot below shows the sample `DIALOG_SliderColor.c`.
- Step 3: Rebuild the sample by choosing `Build/Rebuild All` from the menu (or by pressing F7).
- Step 4: Start the simulation by choosing `Build/Start Debug/Go` from the menu (or by pressing F5). The result of the sample selected above is pictured below:

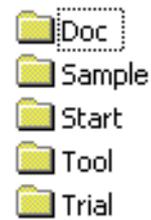


3.1.2 Using the simulator with the emWin source

3.1.2.1 Directory structure

The root directory of the simulator can be anywhere on your PC, e.g. `C:\work\emWinSim`. The directory structure will appear as shown to the right. This structure is very similar to that which we recommend for your target application (see Chapter: "Getting Started" for more information).

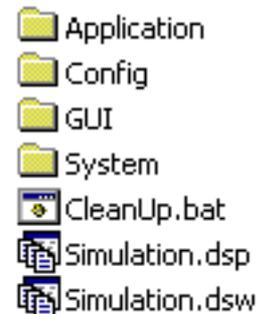
The following table shows the contents of the folders:



Directory	Contents
Doc	Contains emWin-Documentation.
Sample	Code samples, described later in this documentation.
Start	All you need to create a new project with emWin
Tool	Tools shipped with emWin
Trial	Complete trial version

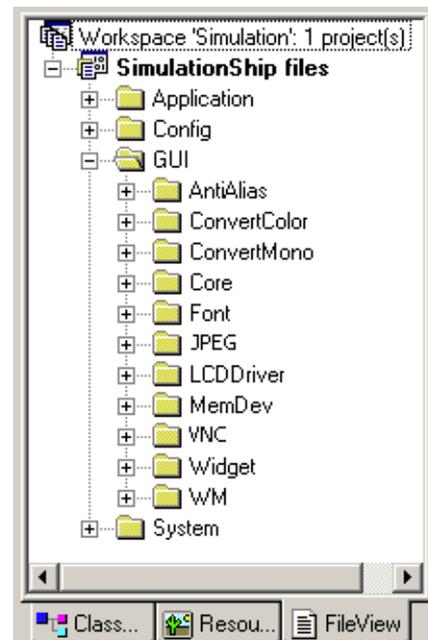
If you want to start a new project you should make a copy of the Start-folder. It contains all you need for a new project. The sub-directories containing emWin program files are in the `Start\GUI` folder and should contain the exact same files as the directories of the same names which you are using for your target (cross) compiler. You should not make any changes to the `GUI` subdirectories, as this would make updating to a newer version of emWin more difficult.

The `Start\Config` directory contains configuration files which need to be modified in order to reflect your target hardware settings (mainly LCD-size and colors which can be displayed).



3.1.2.2 Visual C++ workspace

The root directory shown above includes the Microsoft Visual C++ workspace (`Simulation.dsw`) and project files (`Simulation.dsp`). The workspace allows you to modify an application program and debug it before compiling it on your target system. The directory structure of the Visual C++ workspace will appear similar to that shown to the right. Here, the `GUI` folder is open to display the emWin subdirectories. Please note that your `GUI` directory may not look exactly like the one pictured, depending on which additional features of emWin you have. The folders `Core`, `Font` and `LCDDriver` are part of the basic emWin package and will always appear in the workspace directory.



3.1.2.3 Compiling the application

The demo simulation contains one or more application "C" files (located in the `Application` directory), which can be modified. You may also add files to or remove files from the project. Typically you would want to at least change the bitmap to your own company logo or image of choice. You should then rebuild the program within the Visual C++ workspace in order to test/debug it. Once you have reached a point where you are satisfied with

the result and want to use the program in your application, you should be able to compile these same files on your target system and get the same result on the target display. The general procedure for using the simulator would be as follows:

- Step 1: Open the Visual C++ workspace by double-clicking on `Simulation.dsw`.
- Step 2: Compile the project by choosing `Build/Rebuild All` from the menu (or by pressing F7).
- Step 3: Run the simulation by choosing `Build/Start Debug/Go` from the menu (or by pressing F5).
- Step 4: Replace the bitmap with your own logo or image.
- Step 5: Make further modifications to the application program as you wish, by editing the source code or adding/deleting files.
- Step 6: Compile and run the application program within Visual C++ to test the results. Continue to modify and debug as needed.
- Step 7: Compile and run the application program on your target system.

3.2 Device simulation

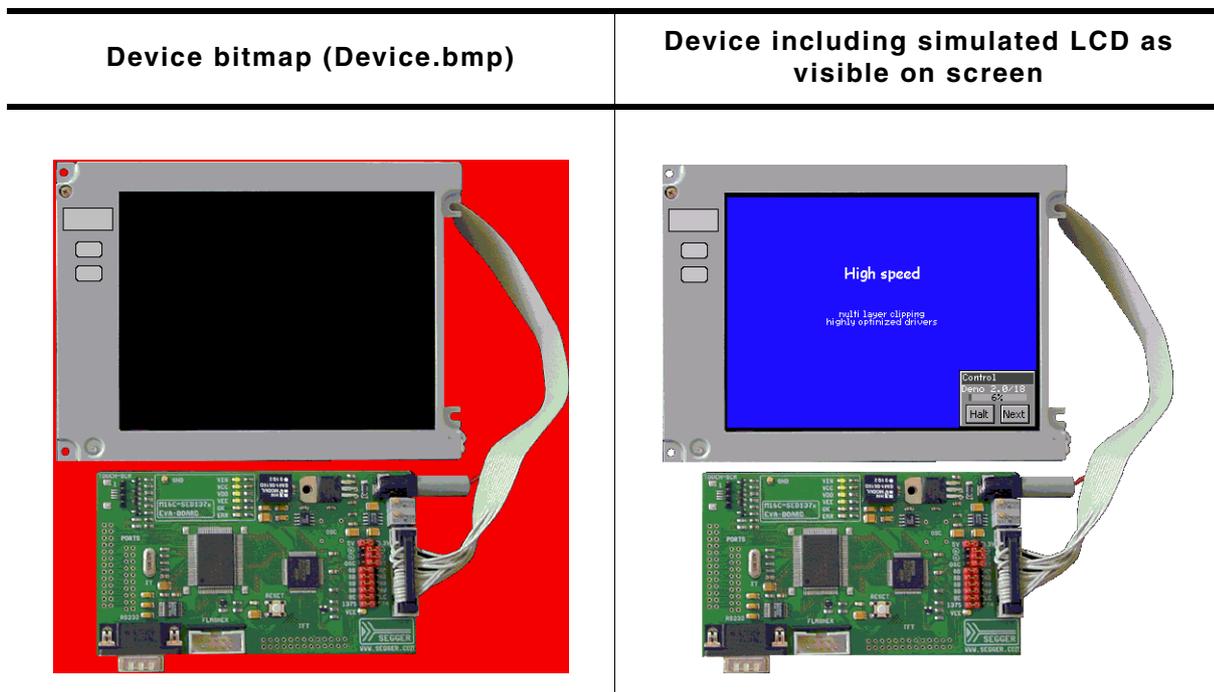
The simulator can show the simulated LCD in a bitmap of your choice, typically your target device. The bitmap can be dragged over the screen and may, in certain applications, be used to simulate the behavior of the entire target device.

In order to simulate the appearance of the device, a bitmap is required. This bitmap is usually a photo (top view) of the device, and must be named `Device.bmp`. It may be a separate file (in the same directory as the executable), or it may be included as a resource in the application by including the following line in the resource file (extension `.rc`):

```
145 BITMAP DISCARDABLE "Device.bmp"
```

For more information, please refer to the Win32 documentation.

The size of the bitmap should be such that the size of the area in which the LCD will be shown equals the resolution of the simulated LCD. This is best seen in the following example:



The red area is automatically made transparent. The transparent areas do not have to be rectangular; they can have an arbitrary shape (up to a certain complexity which is limited by your operating system, but is normally sufficient). Bright red (`0xFF0000`) is the default color for transparent areas, mainly because it is not usually contained in most bitmaps. To use a bitmap with bright red, the default transparency color may be changed with the function `SIM_SetTransColor()`.

3.2.1 Device simulator API

All of the device simulator API functions must be called in the setup phase. The calls should ideally be done from within the routine `SIM_X_Init()`, which is located in the file `SIM_X.c`. The example below calls `SIM_SetLCDPos()` in the setup:

```
#include <windows.h>
#include <stdio.h>

#include "SIM.h"

void SIM_X_Init() {
    SIM_SetLCDPos(0,0);    // Define the position of the LCD in the bitmap
}
```

The table below lists the available device-simulation-related routines in alphabetical order within their respective categories. Detailed descriptions of the routines follow:

Routine	Explanation
<code>SIM_GUI_SetLCDColorBlack()</code>	Set the color to be used as black (color monochrome displays).
<code>SIM_GUI_SetLCDColorWhite()</code>	Set the color to be used as white (color monochrome displays).
<code>SIM_GUI_SetLCDPos()</code>	Set the position for the simulated LCD within the target device bitmap.
<code>SIM_GUI_SetMag()</code>	Set magnification factors for X and/or Y axis.
<code>SIM_GUI_SetTransColor()</code>	Set the color to be used for transparent areas (default: 0xFF0000).

`SIM_GUI_SetLCDColorBlack()`, `SIM_GUI_SetLCDColorWhite()`

Description

Set the colors to be used as black or white, respectively, on color monochrome displays.

Prototypes

```
int SIM_GUI_SetLCDColorBlack(int DisplayIndex, int Color);
int SIM_GUI_SetLCDColorWhite(int DisplayIndex, int Color);
```

Parameter	Meaning
<code>DisplayIndex</code>	Reserved for future use; must be 0.
<code>Color</code>	RGB value of the color.

Add. information

These functions can be used to simulate the true background color of your display. The default color values are black and white, or 0x000000 and 0xFFFFFFFF.

Example using default settings

```
void SIM_X_Init() {
    SIM_GUI_SetLCDPos(14, 84); // Define the position of the LCD
                               // in the bitmap
    SIM_GUI_SetLCDColorBlack (0, 0x000000); // Define the color used as black
    SIM_GUI_SetLCDColorWhite (0, 0xFFFFFFFF); // Define the color used as white
} // (used for colored monochrome displays)
```



Example using yellow instead of white

```
void SIM_X_Init() {
    SIM_GUI_SetLCDPos(14, 84); // Define the position of the LCD
                               // in the bitmap
    SIM_GUI_SetLCDColorBlack (0, 0x000000); // Define the color used as black
    SIM_GUI_SetLCDColorWhite (0, 0x00FFFF); // Define the color used as white
} // (used for colored monochrome displays)
```



SIM_GUI_SetLCDPos()

Description

Sets the position for the simulated LCD within the target device bitmap.

Prototype

```
void SIM_GUI_SetLCDPos(int x, int y);
```

Parameter	Meaning
x	X-position of the upper left corner for the simulated LCD (in pixels).
y	Y-position of the upper left corner for the simulated LCD (in pixels).

Add. information

The X- and Y-positions are relative to the target device bitmap, therefore position (0,0) refers to the upper left corner (origin) of the bitmap and not your actual LCD. Only the origin of the simulated screen needs to be specified; the resolution of your display should already be reflected in the configuration files in the `Config` directory. The use of this function enables the use of the bitmaps `Device.bmp` and `Device1.bmp`. Please note that the values need to be ≥ 0 for enabling the use of the bitmaps. If the use of the device bitmaps should be disabled, omit the call of this function in `SIM_X_Init()`.

SIM_GUI_SetMag()

Description

Sets magnification factors for X and/or Y axis.

Prototype

```
void SIM_GUI_SetMag(int MagX, int MagY);
```

Parameter	Meaning
MagX	Magnification factor for X axis.
MagY	Magnification factor for Y axis.

Add. information

Per default the simulation uses one pixel on the PC for each pixel of the simulated display. The use of this function makes sense for small displays. If using a device bitmap together with a magnification > 1 the device bitmap needs to be adapted to the magnification. The device bitmap is not magnified automatically.

SIM_GUI_SetTransparentColor()

Description

Sets the color to be used for transparent areas of device or hardkey bitmaps.

Prototype

```
I32 SIM_GUI_SetTransparentColor(I32 Color);
```

Parameter	Meaning
Color	RGB value of the color in the format 00000000RRRRRRRRGGGGGGGGBBBBBBBB.

Add. information

The default setting for transparency is bright red (0xFF0000). You would typically only need to change this setting if your bitmap contains the same shade of red.

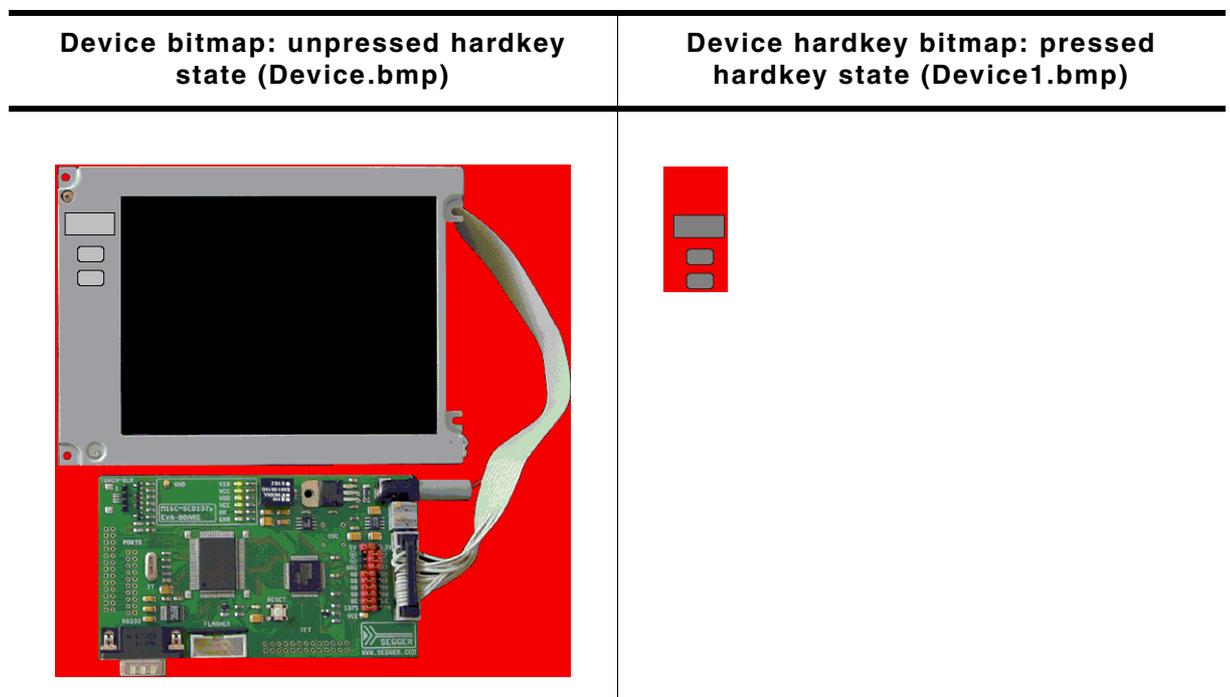
3.2.2 Hardkey simulation

Hardkeys may also be simulated as part of the device, and may be selected with the mouse pointer. The idea is to be able to distinguish whether a key or button on the simulated device is pressed or unpressed. A hardkey is considered "pressed" as long as the mouse button is held down; releasing the mouse button or moving the pointer off of the hardkey "unpresses" the key. A toggle behavior between pressed and unpressed may also be specified with the routine `SIM_HARDKEY_SetMode()`.

In order to simulate hardkeys, you need a second bitmap of the device which is transparent except for the keys themselves (in their pressed state). This bitmap can again be in a separate file in the directory, or included as a resource in the executable. The filename needs to be `Device1.bmp`, and the following lines would typically be included in the resource file (extension `.rc`):

```
145 BITMAP DISCARDABLE "Device.bmp"
146 BITMAP DISCARDABLE "Device1.bmp"
```

Hardkeys may be any shape, as long as they are exactly the same size in pixels in both `Device.bmp` and `Device1.bmp`. The following example illustrates this:



When a key is "pressed" with the mouse, the corresponding section of the hardkey bitmap (`Device1.bmp`) will overlay the device bitmap in order to display the key in its pressed state.

The keys may be polled periodically to determine if their states (pressed/unpressed) have changed and whether they need to be updated. Alternatively, a callback routine may be set to trigger a particular action to be carried out when the state of a hardkey changes.

3.2.2.1 Hardkey simulator API

The hardkey simulation functions are part of the standard simulation program shipped with emWin. If using a user defined emWin simulation these functions may not be available. The table below lists the available hardkey-simulation-related routines in alphabetical order within their respective categories. Detailed descriptions of the routines follow:

Routine	Explanation
SIM_HARDKEY_GetNum()	Return the number of available hardkeys.
SIM_HARDKEY_GetState()	Return the state of a specified hardkey (0: unpressed, 1: pressed).
SIM_HARDKEY_SetCallback()	Set a callback routine to be executed when the state of a specified hardkey changes.
SIM_HARDKEY_SetMode()	Set the behavior for a specified hardkey (default = 0: no toggle).
SIM_HARDKEY_SetState()	Set the state for a specified hardkey (0: unpressed, 1: pressed).

SIM_HARDKEY_GetNum()

Description

Returns the number of available hardkeys.

Prototype

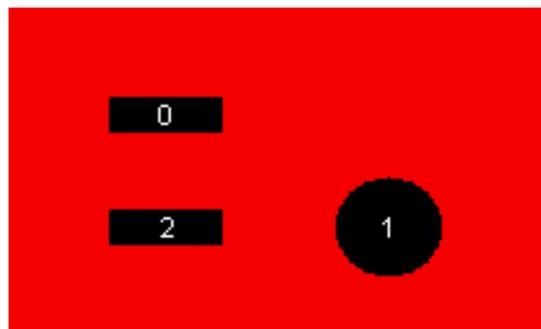
```
int SIM_HARDKEY_GetNum(void);
```

Return value

The number of available hardkeys found in the bitmap.

Add. information

The numbering order for hardkeys is standard reading order (left to right, then top to bottom). The topmost pixel of a hardkey is therefore found first, regardless of its horizontal position. In the bitmap below, for example, the hardkeys are labeled as they would be referenced by the [KeyIndex](#) parameter in other functions:



It is recommended to call this function in order to verify that a bitmap is properly loaded.

SIM_HARDKEY_GetState()

Description

Returns the state of a specified hardkey.

Prototype

```
int SIM_HARDKEY_GetState(unsigned int KeyIndex);
```

Parameter	Meaning
KeyIndex	Index of hardkey (0 = index of first key).

Return value

State of the specified hardkey:

0: unpressed

1: pressed

SIM_HARDKEY_SetCallback()**Description**

Sets a callback routine to be executed when the state of a specified hardkey changes.

Prototype

```
SIM_HARDKEY_CB * SIM_HARDKEY_SetCallback(unsigned int KeyIndex,
                                         SIM_HARDKEY_CB * pfCallback);
```

Parameter	Meaning
KeyIndex	Index of hardkey (0 = index of first key).
pfCallback	Pointer to callback routine.

Return value

Pointer to the previous callback routine.

Add. information

Please note that multi tasking support has to be enabled if GUI functions need to be called within the callback functions. Without multi tasking support only the GUI functions which are allowed to be called within an interrupt routine should be used.

The callback routine must have the following prototype:

Prototype

```
typedef void SIM_HARDKEY_CB(int KeyIndex, int State);
```

Parameter	Meaning
KeyIndex	Index of hardkey (0 = index of first key).
State	State of the specified hardkey (see table below).

Permitted values for parameter State	
0	Unpressed.
1	Pressed.

SIM_HARDKEY_SetMode()**Description**

Sets the behavior for a specified hardkey.

Prototype

```
int SIM_HARDKEY_SetMode(unsigned int KeyIndex, int Mode);
```

Parameter	Meaning
KeyIndex	Index of hardkey (0 = index of first key).
Mode	Behavior mode (see table below).

Permitted values for parameter Mode	
0	Normal behavior (default).
1	Toggle behavior.

Add. information

Normal (default) hardkey behavior means that a key is considered pressed only as long as the mouse button is held down on it. When the mouse is released or moved off of the hardkey, the key is considered unpressed.

With toggle behavior, each click of the mouse toggles the state of a hardkey to pressed or unpressed. That means if you click the mouse on a hardkey and it becomes pressed, it will remain pressed until you click the mouse on it again.

SIM_HARDKEY_SetState()**Description**

Sets the state for a specified hardkey.

Prototype

```
int SIM_HARDKEY_SetState(unsigned int KeyIndex, int State);
```

Parameter	Meaning
KeyIndex	Index of hardkey (0 = index of first key).
State	State of the specified hardkey (see table below).

Permitted values for parameter State	
0	Unpressed.
1	Pressed.

Add. information

This function is only usable when `SIM_HARDKEY_SetMode()` is set to 1 (toggle mode).

3.3 Integrating the emWin simulation into an existing simulation

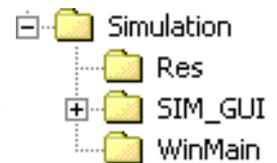
In order to integrate the emWin simulation into an existing simulation, the source code of the simulation is not required. The source code of the simulation is not normally shipped with emWin. It is a separate (optional) software item and is not included in the emWin basic package.

Normally the source code of the emWin simulation is not needed but available as an optional software item. As described earlier in this chapter the basic package and the trial version contains a simulation library. The API functions of this library can be used if for example the emWin simulation should be added to an existing hardware or real time kernel (RTOS) simulation.

To add the emWin simulation to an existing simulation (written in "C" or C++, using the Win32 API), only a few lines of code need to be added.

3.3.1 Directory structure

The subfolder `Simulation` of the `System` folder contains the emWin simulation. The directory structure is shown on the right. The table below explains the contents of the subfolders:



Directory	Contents
Simulation	Simulation source and header files to be used with and without the simulation source code. The folder also contains a ready to use simulation library.
Res	Resource files.
SIM_GUI	GUI simulation source code (optional).
WinMain	Contains the WinMain routine.

3.3.2 Using the simulation library

The following steps will show how to use the simulation library to integrate the emWin simulation into an existing simulation:

- Step 1: Add the simulation library `GUISim.lib` to the project.
- Step 2: Add all GUI files to the project as described in the chapter 2.1.1, "Subdirectories".
- Step 3: Add the include directories to the project as described in the chapter 2.1.2, "Include Directories".
- Step 4: Modify WinMain.

3.3.2.1 Modifying WinMain

Every windows WIN32 program starts with `WinMain()` (contrary to a normal "C" program from the command line, which starts with `main()`). All that needs to be done is to add a few lines of code to this routine.

The following function calls need to be added (normally in this order as show in the following application code sample):

- `SIM_GUI_Init`
- `SIM_GUI_CreateLCDWindow`
- `CreateThread`
- `SIM_GUI_Exit`

3.3.2.2 Sample application

The following application is available under `Sample\WinMain\SampleApp.c` and shows how to integrate the emWin simulation into an existing application:

```

#include <windows.h>
#include "GUI_SIM_Win32.h"
void MainTask(void);

/*****
 *
 *      _Thread
 */
static DWORD __stdcall _Thread(void* Parameter) {
    MainTask();
    return 0;
}

/*****
 *
 *      _WndProcMain
 */
static LRESULT CALLBACK _WndProcMain(HWND hWnd, UINT message,
                                     WPARAM wParam, LPARAM lParam) {
    SIM_GUI_HandleKeyEvents(message, wParam);
    switch (message) {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}

/*****
 *
 *      _RegisterClass
 */
static void _RegisterClass(HINSTANCE hInstance) {
    WNDCLASSEX wcex;
    memset (&wcex, 0, sizeof(wcex));
    wcex.cbSize      = sizeof(WNDCLASSEX);
    wcex.hInstance   = hInstance;
    wcex.style       = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = (WNDPROC)_WndProcMain;
    wcex.hIcon       = 0;
    wcex.hCursor     = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_APPWORKSPACE + 1);
    wcex.lpszMenuName = 0;
    wcex.lpszClassName = "GUIApplication";
    RegisterClassEx(&wcex);
}

/*****
 *
 *      WinMain
 */
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow) {
    DWORD ThreadID;
    MSG Msg;
    HWND hWndMain;
    /* Register window class */
    _RegisterClass(hInstance);
    /* Create main window */
    hWndMain = CreateWindow("GUIApplication", "Application window",
                           WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_VISIBLE,
                           0, 0, 328, 267, NULL, NULL, hInstance, NULL);
    /* Initialize the emWin simulation and create a LCD window */
    SIM_GUI_Init(hInstance, hWndMain, lpCmdLine, "embOS - emWin Simulation");
    SIM_GUI_CreateLCDWindow(hWndMain, 0, 0, 320, 240, 0);
    /* Create a thread which executes the code to be simulated */
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)_Thread, NULL, 0, &ThreadID);
    /* Main message loop */
    while (GetMessage(&Msg, NULL, 0, 0)) {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    SIM_GUI_Exit();
}

```

3.3.3 Integration into the embOS Simulation

3.3.3.1 WinMain

The following code sample shows how to modify the existing WinMain of the embOS simulation in order to integrate the emWin simulation. The red colored lines should be added to WinMain to initialize the emWin simulation, to create a simulation window and to exit the emWin simulation:

```

...
#include "GUI_SIM_Win32.h"
...
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow) {
MSG     Msg;
HACCEL hAccelTable;
HWND    hWndMain;
BITMAP  BmpDevice;
DWORD   ThreadID;
/* Init global data */
_StopHyperThreading();
_hInst = hInstance;
/* Register main window class */
_RegisterClass();
/* Load bitmap */
_hBmpDevice = (HBITMAP)LoadImage(_hInst,
                                (LPCTSTR) IDB_DEVICE,
                                IMAGE_BITMAP, 0, 0, 0);
_hMenuPopup = LoadMenu(_hInst, (LPCSTR) IDC_CONTEXTMENU);
_hMenuPopup = GetSubMenu(_hMenuPopup, 0);
/* Create main window */
GetObject(_hBmpDevice, sizeof(BmpDevice), &BmpDevice);
hWndMain = CreateWindowEx(WS_EX_TOPMOST, _sWindowClass,
                        "embOS Simulation",
                        WS_SYSMENU | WS_CLIPCHILDREN | WS_POPUP | WS_VISIBLE,
                        10, 20, BmpDevice.bmWidth, BmpDevice.bmHeight,
                        NULL, NULL, _hInst, NULL);

if (!hWndMain) {
    return 1; /* Error */
}
/* Init emWin simulation and create window */
SIM_GUI_Init(hInstance, hWndMain, lpCmdLine, "embOS - emWin Simulation");
SIM_GUI_CreateLCDWindow(hWndMain, 80, 50, 128, 64, 0);
/* Show main window */
ShowWindow(hWndMain, nCmdShow);
/* Load accelerator table */
hAccelTable = LoadAccelerators(_hInst, (LPCTSTR) IDC_WINMAIN);
/* application initialization: */
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread, NULL, 0, &ThreadID);
/* main message loop */
if (SIM_Init(hWndMain) == 0) {
    while (GetMessage(&Msg, NULL, 0, 0)) {
        if (!TranslateAccelerator(Msg.hwnd, hAccelTable, &Msg)) {
            TranslateMessage(&Msg);
            DispatchMessage(&Msg);
        }
    }
}
/* Exit emWin simulation */
SIM_GUI_Exit();
return 0;
}

```

3.3.3.2 Target program (main)

The emWin API can be called from one or more target threads. Without RTOS, the WIN32 API function `CreateThread` is normally used to create a target thread which calls the emWin API; within an RTOS simulation, a target task/thread (Created by the simulated RTOS) is used to call the emWin API. In other words: Use `OS_CreateTask` to create a task for the user interface.

Below a modified embOS start application:

```

#include "RTOS.H"
#include "HW_LED.h"
#include "GUI.h"

OS_STACKPTR int Stack0[128], Stack1[128], Stack2[2000]; /* Task stacks */
OS_TASK TCB0, TCB1, TCB2; /* Task-control-blocks */

void Task0(void) {
    while (1) {
        HW_LED_Toggle0();
        OS_Delay(100);
    }
}

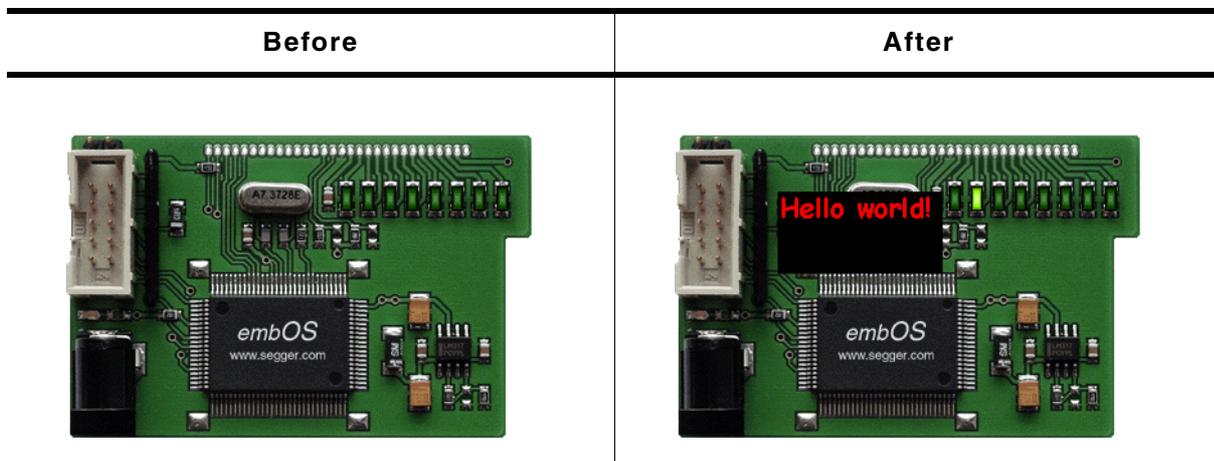
void Task1(void) {
    while (1) {
        HW_LED_Toggle1();
        OS_Delay(500);
    }
}

void MainTask(void) {
    GUI_COLOR aColor[] = {GUI_RED, GUI_YELLOW};
    GUI_Init();
    while (1) {
        int i;
        for (i = 0; i < 2; i++) {
            GUI_Clear();
            GUI_SetColor(aColor[i]);
            GUI_SetFont(&GUI_FontComic24B_ASCII);
            GUI_DispStringAt("Hello world!", 1, 1);
            OS_Delay(200);
        }
    }
}

/*****
 *
 *      main
 *
 *****/
#include <windows.h>
void main(void) {
    OS_IncDI(); /* Initially disable interrupts */
    OS_InitKern(); /* initialize OS */
    OS_InitHW(); /* initialize Hardware for OS */
    /* You need to create at least one task here ! */
    OS_CREATETASK(&TCB0, "HP Task", Task0, 100, Stack0);
    OS_CREATETASK(&TCB1, "LP Task", Task1, 50, Stack1);
    OS_CREATETASK(&TCB2, "GUI Task", MainTask, 80, Stack2);
    OS_Start(); /* Start multitasking */
}

```

The following table shows the simulation before and after integrating the emWin simulation:



3.3.4 GUI simulation API

The table below lists the available routines for user defined simulation programmes in alphabetical order within their respective categories. The functions are only available with the source code of the emWin simulation. Detailed descriptions of the routines follow:

Routine	Explanation
SIM_GUI_CreateLCDInfoWindow()	Creates a window which shows the available colors of the given layer with the given size and position.
SIM_GUI_CreateLCDWindow()	Creates a LCD window with the given size and position.
SIM_GUI_Exit()	Stops the GUI simulation.
SIM_GUI_Init()	Initializes the GUI simulation.
SIM_GUI_SetLCDWindowHook()	Sets a hook function to be called if the LCD window receives a message.

SIM_GUI_CreateLCDInfoWindow()

Description

Creates a window which shows the available colors for the given layer.

Prototype

```
HWND SIM_GUI_CreateLCDInfoWindow(HWND hParent,
                                  int x, int y, int xSize, int ySize
                                  int LayerIndex);
```

Parameter	Meaning
hParent	Handle of the parent window.
x	X position in parent coordinates.
y	Y position in parent coordinates.
xSize	X size in pixel of the new window. Should be 160 if using a color depth between 1 and 8 or 128 if working in high color mode.
ySize	Y size in pixel of the new window. Should be 160 if using a color depth between 1 and 8 or 128 if working in high color mode.
LayerIndex	Index of layer to be shown.

Add. information

The created color window has no frame, no title bar and no buttons.

Example

```
SIM_GUI_CreateLCDInfoWindow(hWnd, 0, 0, 160, 160, 0);
```

Screenshot



SIM_GUI_CreateLCDWindow()

Description

Creates a window which simulates a LCD display with the given size at the given position.

Prototype

```
HWND SIM_GUI_CreateLCDWindow(HWND hParent,
```

```
int x, int y, int xSize, int ySize
int LayerIndex);
```

Parameter	Meaning
hParent	Handle of the parent window.
x	X position in parent coordinates.
y	Y position in parent coordinates.
xSize	X size in pixel of the new window.
ySize	Y size in pixel of the new window.
LayerIndex	Index of layer to be shown.

Add. information

All display output to the given layer will be shown in this window. The size of the window should be the same as configured in `LCDCConf.h`.

The created simulation window has no frame, no title bar and no buttons.

SIM_GUI_Exit()

Description

The function should be called before the simulation returns to the calling process.

Prototype

```
void SIM_GUI_Exit(void);
```

SIM_GUI_Init()

Description

This function initializes the emWin simulation and should be called before any other `SIM_GUI...` function call.

Prototype

```
int SIM_GUI_Init(HINSTANCE hInst, HWND hWndMain,
                char * pCmdLine, const char * sAppName);
```

Parameter	Meaning
hInst	Handle to current instance passed to <code>WinMain</code> .
hWndMain	Handle of the simulations main window.
pCmdLine	Pointer to command line passed to <code>WinMain</code>
sAppName	Pointer to a string that contains the application name.

Add. information

The parameters `hWndMain` and `sAppName` are used if a message box should be displayed.

SIM_GUI_SetLCDWindowHook()

Description

Sets a hook function to be called from the simulation if the LCD window receives a message.

Prototype

```
void SIM_GUI_SetLCDWindowHook(SIM_GUI_tfHook * pfHook);
```

Parameter	Meaning
pfHook	Pointer to hook function.

Prototype of hook function

```
int Hook(HWND hWnd, UINT Message, WPARAM wParam, LPARAM lParam,
        int * pResult);
```

Parameter	Meaning
<code>hWnd</code>	Handle of LCD window.
<code>Message</code>	Message received from the operating system.
<code>wParam</code>	wParam message parameter passed by the system.
<code>lParam</code>	lParam message parameter passed by the system.
<code>pResult</code>	Pointer to an integer which should be used as return code if the message has been processed by the hook function.

Return value

The hook function should return 0 if the message has been processed. In this case the GUI simulation ignores the message.

Chapter 4

Viewer

If you use the simulator when debugging your application, you cannot see the display output when stepping through the source code. The primary purpose of the viewer is to solve this problem. It shows the contents of the simulated display(s) while debugging in the simulation.

The viewer gives you the following additional capabilities:

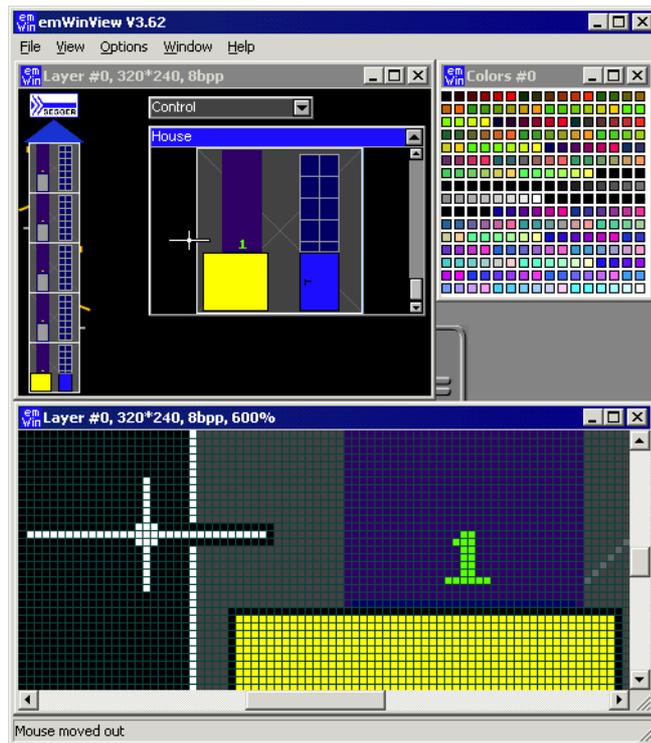
- Multiple windows for each layer
- Watching the whole virtual layer in one window
- Magnification of each layer window
- Composite view if using multiple layers

4.1 Using the viewer

The viewer allows you to:

- Open multiple windows for any layer/display
- Zoom in on any area of a layer/display
- See the contents of the individual layers/displays as well as the composite view in multi-layer configurations
- See the contents of the virtual screen and the visible display when using the virtual screen support.

The screenshot shows the viewer displaying the output of a single layer configuration. The upper left corner shows the simulated display. In the upper right corner is a window, which shows the available colors of the display configuration. At the bottom of the viewer a second display window shows a magnified area of the simulated display. If you start to debug your application, the viewer shows one display window per layer and one color window per layer. In a multi layer configuration, a composite view window will also be visible.



4.1.1 Using the simulator and the viewer

If you use the simulator when debugging your application, you cannot see the display output when stepping through the source code. This is due to a limitation of Win32: If one thread (the one being debugged) is halted, all other threads of the process are also halted. This includes the thread which outputs the simulated display on the screen.

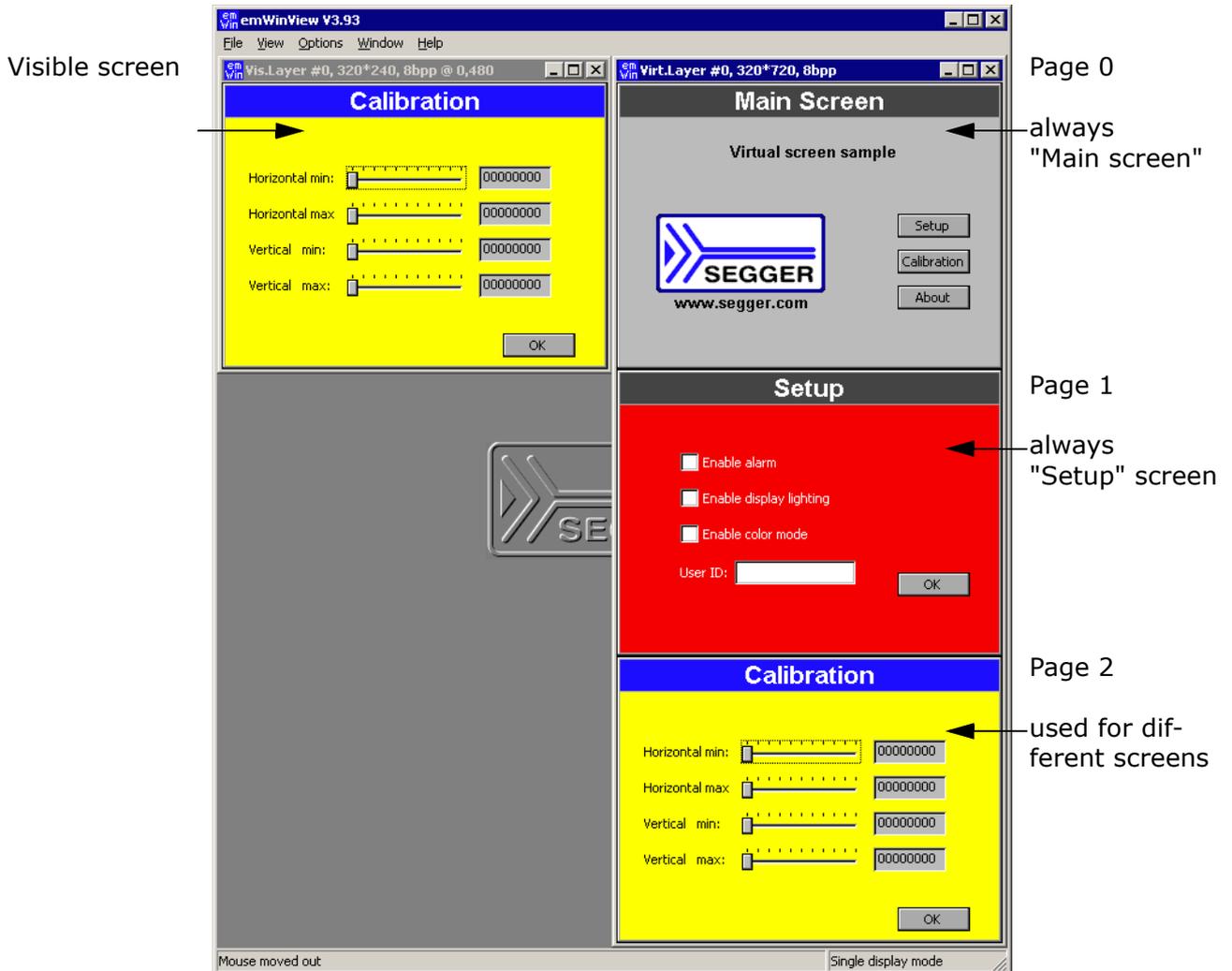
The emWin viewer solves this problem by showing the display window and the color window of your simulation in a separate process. It is your choice if you want to start the viewer before debugging your application or while you are debugging. Our suggestion:

- Step 1: Start the viewer. No display- or color window is shown until the simulation has been started.
- Step 2: Open the Visual C++ workspace.
- Step 3: Compile and run the application program.
- Step 4: Debug the application as described previously.

The advantage is that you can now follow all drawing operations step by step in the LCD window.

4.1.2 Using the viewer with virtual pages

By default the viewer opens one window per layer which shows the visible part of the video RAM, normally the display. If the configured virtual video RAM is larger than the display, the command `View/Virtual Layer/Layer (0...4)` can be used to show the whole video RAM in one window. When using the function `GUI_SetOrg()`, the contents of the visible screen will change, but the virtual layer window remains unchanged:



For more information about virtual screens please refer to chapter 'Virtual Screens'.

4.1.3 Always on top

Per default the viewer window is always on top. You can change this behavior by selecting `Options\Always on top` from the menu.

4.1.4 Open further windows of the display output

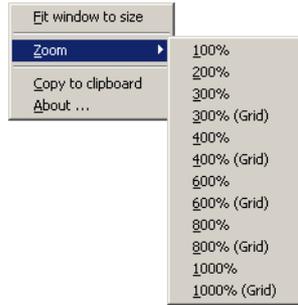
If you want to show a magnified area of the LCD output or the composite view of a multi layer configuration it could be useful to open more than one output window. You can do this by `View/Visible Layer/Layer (1...4)`, `View/Virtual Layer/Layer (1...4)` or `View/Composite`.

4.1.5 Zooming

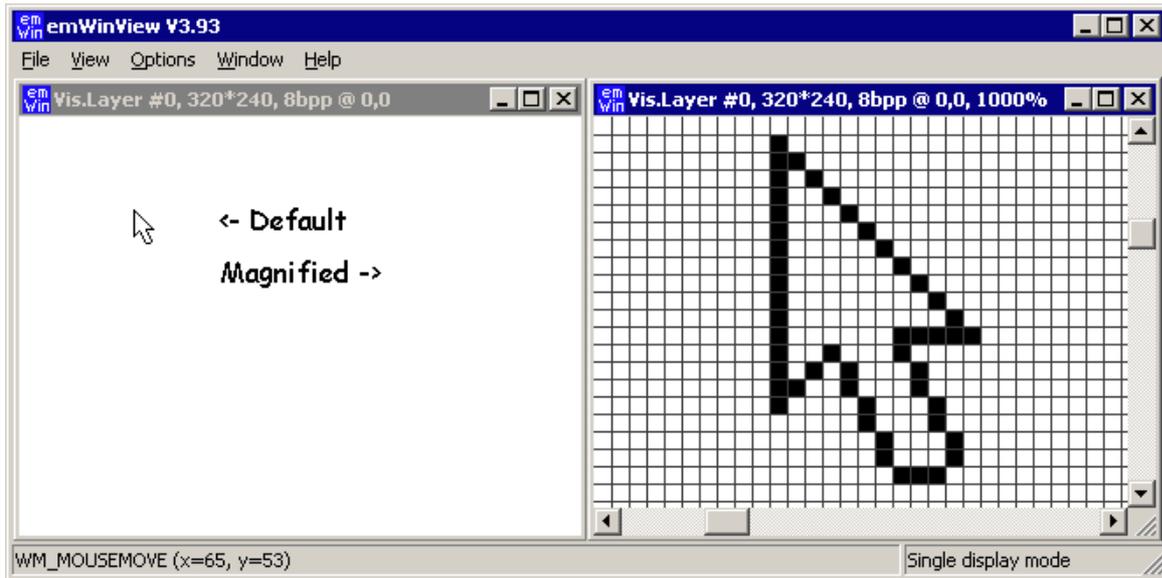
Zooming in or out is easy:

Right-click on a layer or composite window opens the `Zoom` popup menu.

Choose one of the zoom options:



Using the grid



If you magnify the LCD output $\geq 300\%$, you have the choice between showing the output with or without a grid. It is possible to change the color of the grid. This can be done choosing the Menu point `Options/Grid color`.

Adapting the size of the window

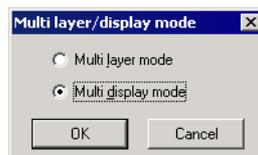
If you want to adapt the size of the window to the magnification choose `Fit window to size` from the first popup menu.

4.1.6 Copy the output to the clipboard

Click onto a LCD window or a composite view with the right mouse key and choose `Copy to clipboard`. Now you can paste the contents of the clipboard for example into the `mspaint` application.

4.1.7 Using the viewer with multiple displays

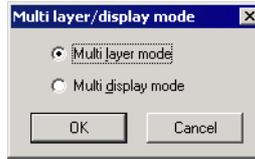
If you are working with multiple displays you should set the viewer into 'Multi display mode' by using the command `Options/Multi layer/display`.



When starting the debugger the viewer will open one display window and one color window for each display:

4.1.8 Using the viewer with multiple layers

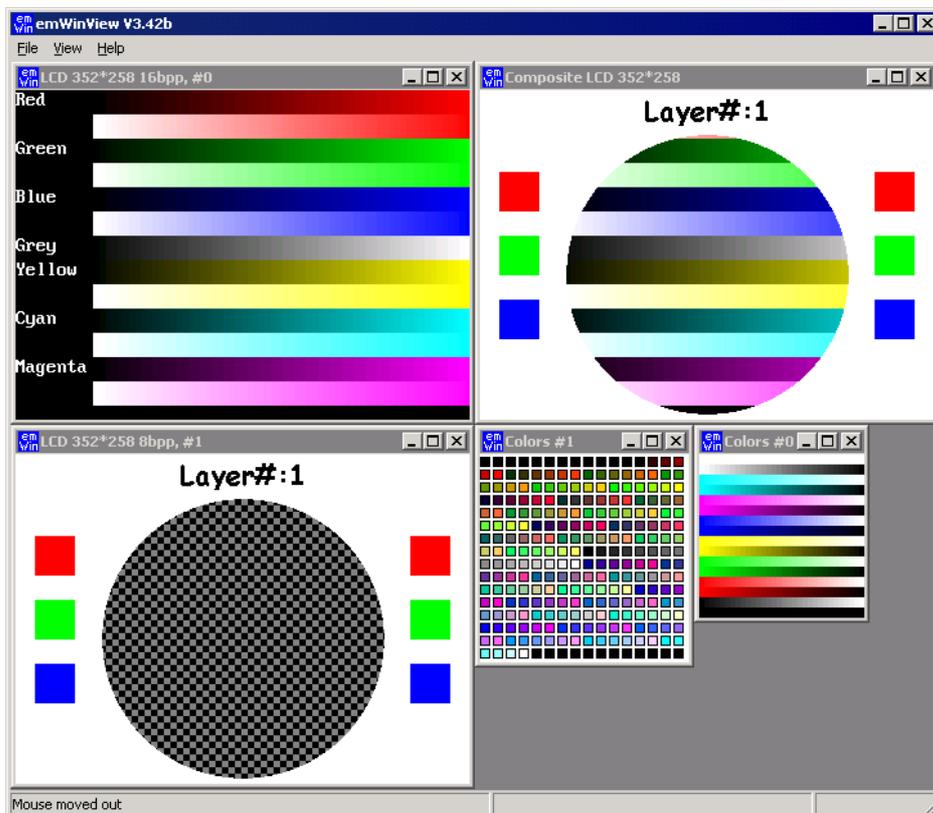
If you are working with multiple layers you should set the viewer into 'Multi layer mode' by using the command `Options/Multi layer/display`.



When starting the debugger the viewer will open one LCD window and one color window for each layer and one composite window for the result.

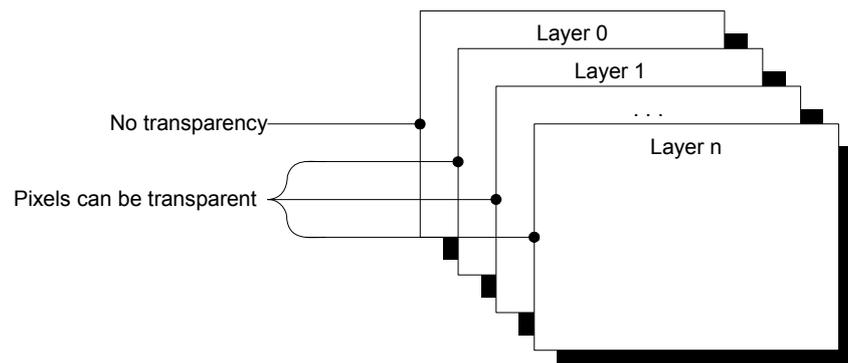
Sample

The sample below shows a screenshot of the viewer with 2 layers. Layer 0 shows color bars with a high color configuration. Layer 1 shows a transparent circle on a white background with colored rectangles. The composite window shows the result which is actually visible on the display



Transparency

The composite window of the viewer shows all layers; layers with higher index are on top of layers with lower index and can have transparent pixels:



Chapter 5

Displaying Text

It is very easy to display text with emWin. Knowledge of only a few routines already allows you to write any text, in any available font, at any point on the display. We first provide a short introduction to displaying text, followed by more detailed explanations of the individual routines that are available.

5.1 Basic routines

In order to display text on the LCD, simply call the routine `GUI_DispString()` with the text you want to display as parameters. For example:

```
GUI_DispString("Hello world!");
```

The above code will display the text "Hello world" at the current text position. However, as you will see, there are routines to display text in a different font or in a certain position. In addition, it is possible to write not only strings but also decimal, hexadecimal and binary values to the display. Even though the graphic displays are usually byte-oriented, the text can be positioned at any pixel of the display, not only at byte positions.

Control characters

Control characters are characters with a character code of less than 32. The control characters are defined as part of ASCII. emWin ignores all control characters except for the following:

Char. Code	ASCII code	"C"	Meaning
10	LF	\n	Line feed. The current text position is changed to the beginning of the next line. Per default, this is: X = 0. Y + =font-distance in pixels (as delivered by <code>GUI_GetFontDistY()</code>).
13	CR	\r	Carriage return. The current text position is changed to the beginning of the current line. Per default, this is: X = 0.

Usage of the control character `LF` can be very convenient in strings. A line feed can be made part of a string so that a string spanning multiple lines can be displayed with a single routine call.

Positioning text at a selected position

This may be done by using the routine `GUI_GotoXY()` as shown in the following example:

```
GUI_GotoXY(10,10); // Set text position (in pixels)
GUI_DispString("Hello world!"); // Show text
```

5.2 Text API

The table below lists the available text-related routines in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
Routines to display text	
GUI_DispChar()	Displays single character at current position.
GUI_DispCharAt()	Displays single character at specified position.
GUI_DispChars()	Displays character a specified number of times.
GUI_DispNextLine()	Moves the cursor to the beginning of the next line.
GUI_DispString()	Displays string at current position.
GUI_DispStringAt()	Displays string at specified position.
GUI_DispStringAtCEOL()	Displays string at specified position, then clear to end of line.
GUI_DispStringHCenterAt()	Displays string centered horizontally at the given position.
GUI_DispStringInRect()	Displays string in specified rectangle.
GUI_DispStringInRectWrap()	Displays string in specified rectangle with optional wrapping.

Routine	Explanation
<code>GUI_DispStringLen()</code>	Display string at current position with specified number of characters.
Selecting text drawing modes	
<code>GUI_GetTextMode()</code>	Returns the current text mode
<code>GUI_SetTextMode()</code>	Sets text drawing mode.
<code>GUI_SetTextStyle()</code>	Sets the text style to be used.
Selecting text alignment	
<code>GUI_GetTextAlign()</code>	Return current text alignment mode.
<code>GUI_SetLBorder()</code>	Set left border after line feed.
<code>GUI_SetTextAlign()</code>	Set text alignment mode.
Setting the current text position	
<code>GUI_GotoX()</code>	Set current X-position.
<code>GUI_GotoXY()</code>	Set current (X,Y) position.
<code>GUI_GotoY()</code>	Set current Y-position.
Retrieving the current text position	
<code>GUI_GetDispPosX()</code>	Return current X-position.
<code>GUI_GetDispPosY()</code>	Return current Y-position.
Routines to clear a window or parts of it	
<code>GUI_Clear()</code>	Clear active window (or entire display if background is the active window).
<code>GUI_DispCEOL()</code>	Clear display from current text position to end of line.

5.3 Routines to display text

GUI_DispChar()

Description

Displays a single character at the current text position in the current window using the current font.

Prototype

```
void GUI_DispChar(U16 c);
```

Parameter	Meaning
<code>c</code>	Character to display.

Add. information

This is the basic routine for displaying a single character. All other display routines (`GUI_DispCharAt()`, `GUI_DispString()`, etc.) call this routine to output the individual characters.

Which characters are available depends on the selected font. If the character is not available in the current font, nothing is displayed.

Example

Shows a capital A on the display:

```
GUI_DispChar('A');
```

Related topics

```
GUI_DispChars(), GUI_DispCharAt()
```

GUI_DispCharAt()

Description

Displays a single character at a specified position in the current window using the current font.

Prototype

```
void GUI_DispCharAt(U16 c, I16P x, I16P y);
```

Parameter	Meaning
<code>c</code>	Character to display.
<code>x</code>	X-position to write to in pixels of the client window.
<code>y</code>	Y-position to write to in pixels of the client window.

Add information

Displays the character with its upper left corner at the specified (X,Y) position. Writes the character using the routine `GUI_DispChar()`. If the character is not available in the current font, nothing is displayed.

Example

Shows a capital A on the display in the upper left corner:

```
GUI_DispCharAt('A', 0, 0);
```

Related topics

```
GUI_DispChar(), GUI_DispChars()
```

GUI_DispChars()

Description

Displays a character a specified number of times at the current text position in the current window using the current font.

Prototype

```
void GUI_DispChars(U16 c, int Cnt);
```

Parameter	Meaning
<code>c</code>	Character to display.
<code>Cnt</code>	Number of repetitions (0 <= Cnt <= 32767).

Add. information

Writes the character using the routine `GUI_DispChar()`. If the character is not available in the current font, nothing is displayed.

Example

Shows the line "*****" on the display:

```
GUI_DispChars('*', 30);
```

Related topics

```
GUI_DispChar(), GUI_DispCharAt()
```

GUI_DispNextLine()

Description

Moves the cursor to the beginning of the next line.

Prototype

```
void GUI_DispNextLine(void);
```

Related topics

`GUI_SetLBorder()`

GUI_DispString()**Description**

Displays the string passed as parameter at the current text position in the current window using the current font.

Prototype

```
void GUI_DispString(const char GUI_FAR *s);
```

Parameter	Meaning
<code>s</code>	String to display.

Add. information

The string can contain the control character `\n`. This control character moves the current text position to the beginning of the next line.

Example

Shows "Hello world" on the display and "Next line" on the next line:

```
GUI_DispString("Hello world");// Disp text
GUI_DispString("\nNext line");// Disp text
```

Related topics

`GUI_DispStringAt()`, `GUI_DispStringAtCEOL()`, `GUI_DispStringLen()`,

GUI_DispStringAt()**Description**

Displays the string passed as parameter at a specified position in the current window using the current font.

Prototype

```
void GUI_DispStringAt(const char GUI_FAR *s, int x, int y);
```

Parameter	Meaning
<code>s</code>	String to display.
<code>x</code>	X-position to write to in pixels of the client window.
<code>y</code>	Y-position to write to in pixels of the client window.

Example

Shows "Position 50,20" at position 50,20 on the display:

```
GUI_DispStringAt("Position 50,20", 50, 20);// Disp text
```

Related topics

`GUI_DispString()`, `GUI_DispStringAtCEOL()`, `GUI_DispStringLen()`,

GUI_DispStringAtCEOL()**Description**

This routine uses the exact same parameters as `GUI_DispStringAt()`. It does the same thing: displays a given string at a specified position. However, after doing so, it clears the remaining part of the line to the end by calling the routine `GUI_DispCEOL()`. This routine can be handy if one string is to overwrite another, and the overwriting string is or may be shorter than the previous one.

GUI_DispStringHCenterAt()

Description

Displays the string passed as parameter horizontally centered at a specified position in the current window using the current font.

Prototype

```
void GUI_DispStringHCenterAt(const char GUI_FAR *s, int x, int y);
```

Parameter	Meaning
<code>s</code>	String to display.
<code>x</code>	X-position to write to in pixels of the client window.
<code>y</code>	Y-position to write to in pixels of the client window.

GUI_DispStringInRect()

Description

Displays the string passed as parameter at a specified position within a specified rectangle, in the current window using the current font.

Prototype

```
void GUI_DispStringInRect(const char GUI_FAR *s, GUI_RECT *pRect, int Align);
```

Parameter	Meaning
<code>s</code>	String to display.
<code>pRect</code>	Rectangle to write to in pixels of the client window.
<code>Align</code>	Alignment flags; "OR" combinable. A flag for horizontal and a flag for vertical alignment should be combined. Available flags are: GUI_TA_TOP, GUI_TA_BOTTOM, GUI_TA_VCENTER for vertical alignment. GUI_TA_LEFT, GUI_TA_RIGHT, GUI_TA_HCENTER for horizontal alignment.

Example

Shows the word "Text" centered horizontally and vertically in the current window:

```
GUI_RECT rClient;
GUI_GetClientRect(&rClient);
GUI_DispStringInRect("Text", &rClient, GUI_TA_HCENTER | GUI_TA_VCENTER);
```

Add. information

If the specified rectangle is too small, the text will be clipped.

Related topics

`GUI_DispString()`, `GUI_DispStringAtCEOL()`, `GUI_DispStringLength()`,

GUI_DispStringInRectWrap()

Description

Displays a string at a specified position within a specified rectangle, in the current window using the current font and (optionally) wraps the text.

Prototype

```
void GUI_DispStringInRectWrap(const char GUI_UNI_PTR * s,
                             GUI_RECT * pRect,
                             int TextAlign,
```

```
GUI_WRAPMODE WrapMode);
```

Parameter	Meaning
<code>s</code>	String to display.
<code>pRect</code>	Rectangle to write to in pixels of the client window.
<code>TextAlign</code>	Alignment flags; "OR" combinable. A flag for horizontal and a flag for vertical alignment should be combined. Available flags are: GUI_TA_TOP, GUI_TA_BOTTOM, GUI_TA_VCENTER for vertical alignment. GUI_TA_LEFT, GUI_TA_RIGHT, GUI_TA_HCENTER for horizontal alignment.
<code>WrapMode</code>	(see table below)

Permitted values for parameter <code>pLCD_Api</code>	
GUI_WRAPMODE_NONE	No wrapping will be performed.
GUI_WRAPMODE_WORD	Text is wrapped word wise.
GUI_WRAPMODE_CHAR	Text is wrapped char wise.

Add. information

If word wrapping should be performed and the given rectangle is too small for a word char wrapping is executed at this word.

Example

Shows a text centered horizontally and vertically in the given rectangle with word wrapping:

```
int i;
char acText[] = "This sample demonstrates text wrapping";
GUI_RECT Rect = {10, 10, 59, 59};
GUI_WRAPMODE aWm[] = {GUI_WRAPMODE_NONE,
                      GUI_WRAPMODE_CHAR,
                      GUI_WRAPMODE_WORD};

GUI_SetTextMode(GUI_TM_TRANS);
for (i = 0; i < 3; i++) {
    GUI_SetColor(GUI_BLUE);
    GUI_FillRectEx(&Rect);
    GUI_SetColor(GUI_WHITE);
    GUI_DispStringInRectWrap(acText, &Rect, GUI_TA_LEFT, aWm[i]);
    Rect.x0 += 60;
    Rect.x1 += 60;
}
}
```

Screenshot of above example



GUI_DispStringLen()

Description

Displays the string passed as parameter with a specified number of characters at the current text position, in the current window using the current font.

Prototype

```
void GUI_DispStringLen(const char GUI_FAR *s, int Len);
```

Parameter	Meaning
<code>s</code>	String to display. Should be a \0 terminated array of 8-bit character. Passing NULL as parameter is permitted.
<code>Len</code>	Number of characters to display.

Add. information

If the string has less characters than specified (is shorter), it is padded with spaces. If the string has more characters than specified (is longer), then only the given number of characters is actually displayed.

This function is especially useful if text messages can be displayed in different languages (and will naturally differ in length), but only a certain number of characters can be displayed.

Related topics

`GUI_DispString()`, `GUI_DispStringAt()`, `GUI_DispStringAtCEOL()`,

5.4 Selecting text drawing modes

Normally, text is written into the selected window at the current text position using the selected font in normal text. Normal text means that the text overwrites whatever is already displayed where the bits set in the character mask are set on the display. In this mode, active bits are written using the foreground color, while inactive bits are written with the background color. However, in some situations it may be desirable to change this default behavior. `emWin` offers four flags for this purpose (one default plus three modifiers), which may be combined:

Normal text

Text can be displayed normally by specifying `GUI_TEXTMODE_NORMAL` or 0.

Reverse text

Text can be displayed in reverse by specifying `GUI_TEXTMODE_REVERSE`. What is usually displayed as white on black will be displayed as black on white.

Transparent text

Transparent text means that the text is written on top of whatever is already visible on the display. The difference is that whatever was previously on the screen can still be seen, whereas with normal text the background is erased.

Text can be displayed transparently by specifying `GUI_TEXTMODE_TRANS`.

XOR text

What usually is drawn white (the actual character) is inverted. The effect is identical to that of the default mode (normal text) if the background is black. If the background is white, the output is identical to reverse text.

If you use colors, an inverted pixel is calculated as follows:

New pixel color = number of colors - actual pixel color - 1.

Transparent reversed text

As with transparent text, it does not overwrite the background, and as with reverse text, the text is displayed in reverse.

Text can be displayed in reverse transparently by specifying `GUI_TEXTMODE_TRANS | GUI_TEXTMODE_REVERSE`.

Example

Displays normal, reverse, transparent, XOR, and transparent reversed text:

```
GUI_SetFont (&GUI_Font8x16);
GUI_SetBkColor (GUI_BLUE);
GUI_Clear ();
GUI_SetPenSize (10);
GUI_SetColor (GUI_RED);
GUI_DrawLine (80, 10, 240, 90);
GUI_DrawLine (80, 90, 240, 10);
GUI_SetBkColor (GUI_BLACK);
GUI_SetColor (GUI_WHITE);
GUI_SetTextMode (GUI_TM_NORMAL);
GUI_DispStringHCenterAt ("GUI_TM_NORMAL" , 160, 10);
GUI_SetTextMode (GUI_TM_REV);
GUI_DispStringHCenterAt ("GUI_TM_REV" , 160, 26);
GUI_SetTextMode (GUI_TM_TRANS);
GUI_DispStringHCenterAt ("GUI_TM_TRANS" , 160, 42);
```

```

GUI_SetTextMode(GUI_TM_XOR);
GUI_DispStringHCenterAt("GUI_TM_XOR" , 160, 58);
GUI_SetTextMode(GUI_TM_TRANS | GUI_TM_REV);
GUI_DispStringHCenterAt("GUI_TM_TRANS | GUI_TM_REV", 160, 74);

```

Screen shot of above example



GUI_GetTextMode()

Description

Returns the currently selected text mode.

Prototype

```
int GUI_GetTextMode(void);
```

Return value

The currently selected text mode.

GUI_SetTextMode()

Description

Sets the text mode to the parameter specified.

Prototype

```
int GUI_SetTextMode(int TextMode);
```

Parameter	Meaning
TextMode	Text mode to set. May be any combination of the TEXTMODE flags.

Permitted values for parameter TextMode (OR-combinable)	
GUI_TEXTMODE_NORMAL	Sets normal text. This is the default setting; the value is identical to 0.
GUI_TEXTMODE_REVERSE	Sets reverse text.
GUI_TEXTMODE_TRANSPARENT	Sets transparent text.
GUI_TEXTMODE_XOR	Text will be inverted on the display.

Return value

The previous selected text mode.

Example

Shows "The value is" at position 0,0 on the display, shows a value in reverse text, then sets the text mode back to normal:

```

int i = 20;
GUI_DispStringAt("The value is", 0, 0);
GUI_SetTextMode(GUI_TEXTMODE_REVERSE);
GUI_DispDec(20, 3);
GUI_SetTextMode(GUI_TEXTMODE_NORMAL);

```

GUI_SetTextStyle()

Description

Sets the text style to the parameter specified.

Prototype

```
char GUI_SetTextStyle(char Style);
```

Parameter	Meaning
Style	Text style to set (see table below).

Permitted values for parameter Style	
GUI_TS_NORMAL	Renders text normal (default).
GUI_TS_UNDERLINE	Renders text underlined.
GUI_TS_STRIKETHRU	Renders text in strikethrough type.
GUI_TS_OVERLINE	Renders text in overline type.

Return value

The previous selected text style.

5.5 Selecting text alignment

GUI_GetTextAlign()

Description

Returns the current text alignment mode.

Prototype

```
int GUI_GetTextAlign(void);
```

GUI_SetLBorder()

Description

Sets the left border for line feeds in the current window.

Prototype

```
void GUI_SetLBorder(int x)
```

Parameter	Meaning
x	New left border (in pixels, 0 is left border).

GUI_SetTextAlign()

Description

Sets the text alignment mode for string output in the current window.

Prototype

```
int GUI_SetTextAlign(int TextAlign);
```

Parameter	Meaning
TextAlign	Text alignment mode to set. May be a combination of a horizontal and a vertical alignment flag.

Permitted values for parameter <code>TextAlign</code> (horizontal and vertical flags are OR-combinable)	
Horizontal alignment	
<code>GUI_TA_LEFT</code>	Align X-position left (default).
<code>GUI_TA_HCENTER</code>	Center X-position.
<code>GUI_TA_RIGHT</code>	Align X-position right.
Vertical alignment	
<code>GUI_TA_TOP</code>	Align Y-position with top of characters (default).
<code>GUI_TA_VCENTER</code>	Center Y-position.
<code>GUI_TA_BOTTOM</code>	Align Y-position with bottom pixel line of font.

Return value

The selected text alignment mode.

Add. information

`GUI_SetTextAlign()` does not affect the character output routines beginning with `GUI_DispChar()`. Please note that the settings made with this function are valid only for one string.

Example

Displays the value 1234 with the center of the text at x=100, y=100:

```
GUI_SetTextAlign(GUI_TA_HCENTER | GUI_TA_VCENTER);
GUI_DispDecAt(1234,100,100,4);
```

5.6 Setting the current text position

Every task has a current text position. This is the position relative to the origin of the window (usually (0,0)) where the next character will be written if a text output routine is called. Initially, this position is (0,0), which is the upper left corner of the current window. There are 3 functions which can be used to set the current text position.

GUI_GotoXY(), GUI_GotoX(), GUI_GotoY()

Description

Set the current text write position.

Prototypes

```
char GUI_GotoXY(int x, int y);
char GUI_GotoX(int x);
char GUI_GotoY(int y);
```

Parameter	Meaning
<code>x</code>	New X-position (in pixels, 0 is left border).
<code>y</code>	New Y-position (in pixels, 0 is top border).

Return value

Usually 0.

If a value $\neq 0$ is returned, then the current text position is outside of the window (to the right or below), so a following write operation can be omitted.

Add. information

`GUI_GotoXY()` sets both the X- and Y-components of the current text position.

`GUI_GotoX()` sets the X-component of the current text position; the Y-component remains unchanged.

`GUI_GotoY()` sets the Y-component of the current text position; the X-component remains unchanged.

Example

Shows "(20,20)" at position 20,20 on the display:

```
GUI_GotoXY(20,20)
GUI_DispString("The value is");
```

5.7 Retrieving the current text position

GUI_GetDispPosX()

Description

Returns the current X-position.

Prototype

```
int GUI_GetDispPosX(void);
```

GUI_GetDispPosY()

Description

Returns the current Y-position.

Prototype

```
int GUI_GetDispPosY(void);
```

5.8 Routines to clear a window or parts of it

GUI_Clear()

Description

Clears the current window.

Prototype

```
void GUI_Clear(void);
```

Add. information

If no window has been defined, the current window is the entire display. In this case, the entire display is cleared.

Example

Shows "Hello world" on the display, waits 1 second and then clears the display:

```
GUI_DispStringAt("Hello world", 0,0); // Disp text
GUI_Delay(1000); // Wait 1 second (not part of emWin)
GUI_Clear(); // Clear screen
```

GUI_DispCEOL()

Description

Clears the current window (or the display) from the current text position to the end of the line using the height of the current font.

Prototype

```
void GUI_DispCEOL(void);
```

Example

Shows "Hello world" on the display, waits 1 second and then displays "Hi" in the same place, replacing the old string:

```
GUI_DispStringAt("Hello world", 0, 0); // Disp text  
Delay (1000);  
GUI_DispStringAt("Hi", 0, 0);  
GUI_DispCEOL();
```


Chapter 6

Displaying Values

The preceding chapter explained how to show strings on the display. Of course you may use strings and the functions of the standard "C" library to display values. However, this can sometimes be a difficult task. It is usually much easier (and much more efficient) to call a routine that displays the value in the form that you want. emWin supports different decimal, hexadecimal and binary outputs. The individual routines are explained in this chapter.

All functions work without the usage of a floating-point library and are optimized for both speed and size. Of course `sprintf` may also be used on any system. Using the routines in this chapter can sometimes simplify things and save both ROM space and execution time.

6.1 Value API

The table below lists the available value-related routines in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
Displaying decimal values	
<code>GUI_DispNet()</code>	Display value in decimal form at current position with specified number of characters.
<code>GUI_DispNetAt()</code>	Display value in decimal form at specified position with specified number of characters.
<code>GUI_DispNetMin()</code>	Display value in decimal form at current position with minimum number of characters.
<code>GUI_DispNetShift()</code>	Display long value in decimal form with decimal point at current position with specified number of characters.
<code>GUI_DispNetSpace()</code>	Display value in decimal form at current position with specified number of characters, replace leading zeros with spaces.
<code>GUI_DispNetDec()</code>	Display value in decimal form at current position with specified number of characters and sign.
<code>GUI_DispNetDecShift()</code>	Display long value in decimal form with decimal point at current position with specified number of characters and sign.
Displaying floating-point values	
<code>GUI_DispNetFloat()</code>	Display floating-point value with specified number of characters.
<code>GUI_DispNetFloatFix()</code>	Display floating-point value with fixed no. of digits to the right of decimal point.
<code>GUI_DispNetFloatMin()</code>	Display floating-point value with minimum number of characters.
<code>GUI_DispNetSFloatFix()</code>	Display floating-point value with fixed no. of digits to the right of decimal point and sign.
<code>GUI_DispNetSFloatMin()</code>	Display floating-point value with minimum number of characters and sign.
Displaying binary values	
<code>GUI_DispNetBin()</code>	Display value in binary form at current position.
<code>GUI_DispNetBinAt()</code>	Display value in binary form at specified position.
Displaying hexadecimal values	
<code>GUI_DispNetHex()</code>	Display value in hexadecimal form at current position.
<code>GUI_DispNetHexAt()</code>	Display value in hexadecimal form at specified position.
Version of emWin	
<code>GUI_GetVersionString()</code>	Return the current version of emWin.

6.2 Displaying decimal values

GUI_DispNet()

Description

Displays a value in decimal form with a specified number of characters at the current text position, in the current window using the current font.

Prototype

```
void GUI_DispNet(I32 v, U8 Len);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum -2147483648 (= -2^{31}). Maximum 2147483647 (= $2^{31} - 1$).
<code>Len</code>	No. of digits to display (max. 10).

Add. information

Leading zeros are not suppressed (are shown as 0).
If the value is negative, a minus sign is shown.

Example

```
// Display time as minutes and seconds
GUI_DispString("Min:");
GUI_DispDec(Min,2);
GUI_DispString(" Sec:");
GUI_DispDec(Sec,2);
```

Related topics

GUI_DispSDec(), GUI_DispDecAt(), GUI_DispDecMin(), GUI_DispDecSpace()

GUI_DispDecAt()**Description**

Displays a value in decimal form with a specified number of characters at a specified position, in the current window using the current font.

Prototype

```
void GUI_DispDecAt(I32 v, I16P x, I16P y, U8 Len);
```

Parameter	Meaning
v	Value to display. Minimum -2147483648 (= -2^{31}). Maximum 2147483647 (= $2^{31} - 1$).
x	X-position to write to in pixels of the client window.
y	Y-position to write to in pixels of the client window.
Len	No. of digits to display (max. 10).

Add. information

Leading zeros are not suppressed.
If the value is negative, a minus sign is shown.

Example

```
// Update seconds in upper right corner
GUI_DispDecAT(Sec, 200, 0, 2);
```

Related topics

GUI_DispDec(), GUI_DispSDec(), GUI_DispDecMin(), GUI_DispDecSpace()

GUI_DispDecMin()**Description**

Displays a value in decimal form at the current text position in the current window using the current font. The length need not be specified; the minimum length will automatically be used.

Prototype

```
void GUI_DispDecMin(I32 v);
```

Parameter	Meaning
v	Value to display. Minimum: -2147483648 (= -2^{31}); maximum 2147483647 (= $2^{31} - 1$). Maximum no. of digits displayed is 10.

Add. information

If values have to be aligned but differ in the number of digits, this function is not a good choice. Try one of the functions that specify the number of digits.

Example

```
// Show result
GUI_DispString("The result is :");
GUI_DispDecMin(Result);
```

Related topics

GUI_DispDec(), GUI_DispDecAt(), GUI_DispSDec(), GUI_DispDecSpace()

GUI_DispDecShift()**Description**

Displays a long value in decimal form with a specified number of characters and with decimal point at the current text position, in the current window using the current font.

Prototype

```
void GUI_DispDecShift(I32 v, U8 Len, U8 Shift);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum: -2147483648 (= -2^{31}); maximum: 2147483647 (= $2^{31} - 1$).
<code>Len</code>	No. of digits to display (max. 10).
<code>Shift</code>	No. of digits to show to right of decimal point.

Add. information

Watch the maximum number of 9 characters (including sign and decimal point).

GUI_DispDecSpace()**Description**

Displays a value in decimal form at the current text position in the current window using the current font. Leading zeros are suppressed (replaced by spaces).

Prototype

```
void DispDecSpace(I32 v, U8 MaxDigits);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum: -2147483648 (= -2^{31}); maximum: 2147483647 (= $2^{31} - 1$).
<code>MaxDigits</code>	No. of digits to display, including leading spaces. Maximum no. of digits displayed is 10 (excluding leading spaces).

Add. information

If values have to be aligned but differ in the number of digits, this function is a good choice.

Example

```
// Show result
GUI_DispString("The result is :");
GUI_DispDecSpace(Result, 200);
```

Related topics

GUI_DispDec(), GUI_DispDecAt(), GUI_DispSDec(), GUI_DispDecMin()

GUI_DispSDec()**Description**

Displays a value in decimal form (with sign) with a specified number of characters at the current text position, in the current window using the current font.

Prototype

```
void GUI_DispsDec(I32 v, U8 Len);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum: -2147483648 (= -2^{31}); maximum: 2147483647 (= $2^{31} - 1$).
<code>Len</code>	No. of digits to display (max. 10).

Add. information

Leading zeros are not suppressed.

This function is similar to `GUI_DispsDec`, but a sign is always shown in front of the value, even if the value is positive.

Related topics

`GUI_DispsDec()`, `GUI_DispsDecAt()`, `GUI_DispsDecMin()`, `GUI_DispsDecSpace()`

GUI_DispsDecShift()

Description

Displays a `long` value in decimal form (with sign) with a specified number of characters and with decimal point at the current text position, in the current window using the current font.

Prototype

```
void GUI_DispsDecShift(I32 v, U8 Len, U8 Shift);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum: -2147483648 (= -2^{31}); maximum: 2147483647 (= $2^{31} - 1$).
<code>Len</code>	No. of digits to display (max. 10).
<code>Shift</code>	No. of digits to show to right of decimal point.

Add. information

A sign is always shown in front of the value.

Watch the maximum number of 9 characters (including sign and decimal point).

Example

```
void DemoDec(void) {
    long l = 12345;
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x8);
    GUI_DispsStringAt("GUI_DispsDecShift:\n",0,0);
    GUI_DispsDecShift(l, 7, 3);
    GUI_SetFont(&GUI_Font6x8);
    GUI_DispsStringAt("Press any key",0,GUI_VYSIZE-8);
    WaitKey();
}
```

Screen shot of above example



6.3 Displaying floating-point values

GUI_DispFloat()

Description

Displays a floating-point value with a specified number of characters at the current text position in the current window using the current font.

Prototype

```
void GUI_DispFloat(float v, char Len);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum 1.2 E-38; maximum 3.4 E38.
<code>Len</code>	No. of digits to display (max. 10).

Add. information

Leading zeros are suppressed. The decimal point counts as one character.
If the value is negative, a minus sign is shown.

Example

```
/*      Shows all features for displaying floating point values      */
void DemoFloat(void) {
    float f = 123.45678;
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x8);
    GUI_DispStringAt("GUI_DispFloat:\n",0,0);
    GUI_DispFloat(f,9);
    GUI_GotoX(100);
    GUI_DispFloat(-f,9);
    GUI_DispStringAt("GUI_DispFloatFix:\n",0,20);
    GUI_DispFloatFix(f,9,2);
    GUI_GotoX(100);
    GUI_DispFloatFix(-f,9,2);
    GUI_DispStringAt("GUI_DispSFloatFix:\n",0,40);
    GUI_DispSFloatFix(f,9,2);
    GUI_GotoX(100);
    GUI_DispSFloatFix(-f,9,2);
    GUI_DispStringAt("GUI_DispFloatMin:\n",0,60);
    GUI_DispFloatMin(f,3);
    GUI_GotoX(100);
    GUI_DispFloatMin(-f,3);
    GUI_DispStringAt("GUI_DispSFloatMin:\n",0,80);
    GUI_DispSFloatMin(f,3);
    GUI_GotoX(100);
    GUI_DispSFloatMin(-f,3);
    GUI_SetFont(&GUI_Font6x8);
    GUI_DispStringAt("Press any key",0,GUI_VYSIZE-8);
    WaitKey();
}
```

Screen shot of above example

```

GUI_DisFloat:
123.45678      -123.4568
GUI_DisFloatFix:
000123.46     -00123.46
GUI_Dis$FloatFix:
+00123.46     -00123.46
GUI_DisFloatMin:
123.457      -123.457
GUI_Dis$FloatMin:
+123.457     -123.457

Press any key

```

GUI_DisFloatFix()

Description

Displays a floating-point value with specified number of total characters and a specified number of characters to the right of the decimal point, at the current text position in the current window using the current font.

Prototype

```
void GUI_DisFloatFix (float v, char Len, char Decs);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum 1.2 E-38; maximum 3.4 E38.
<code>Len</code>	No. of digits to display (max. 10).
<code>Decs</code>	No. of digits to show to right of decimal point.

Add. information

Leading zeros are not suppressed.
If the value is negative, a minus sign is shown.

GUI_DisFloatMin()

Description

Displays a floating-point value with a minimum number of decimals to the right of the decimal point, at the current text position in the current window using the current font.

Prototype

```
void GUI_DisFloatMin(float f, char Fract);
```

Parameter	Meaning
<code>v</code>	Value to display. Minimum 1.2 E-38; maximum 3.4 E38.
<code>Fract</code>	Minimum no. of characters to display.

Add. information

Leading zeros are suppressed.
If the value is negative, a minus sign is shown.
The length need not be specified; the minimum length will automatically be used. If values have to be aligned but differ in the number of digits, this function is not a good choice. Try one of the functions that specify the number of digits.

GUI_DispSFloatFix()

Description

Displays a floating-point value (with sign) with a specified number of total characters and a specified number of characters to the right of the decimal point, in the current window using the current font.

Prototype

```
void GUI_DispSFloatFix(float v, char Len, char Decs);
```

Parameter	Meaning
v	Value to display. Minimum 1.2 E-38; maximum 3.4 E38.
Len	No. of digits to display (max. 10).
Decs	No. of digits to show to right of decimal point.

Add. information

Leading zeros are not suppressed.
A sign is always shown in front of the value.

GUI_DispSFloatMin()

Description

Displays a floating-point value (with sign) with a minimum number of decimals to the right of the decimal point, at the current text position in the current window using the current font.

Prototype

```
void GUI_DispSFloatMin(float f, char Fract);
```

Parameter	Meaning
v	Value to display. Minimum 1.2 E-38; maximum 3.4 E38.
Fract	Minimum no. of digits to display.

Add. information

Leading zeros are suppressed.
A sign is always shown in front of the value.
The length need not be specified; the minimum length will automatically be used. If values have to be aligned but differ in the number of digits, this function is not a good choice. Try one of the functions that specify the number of digits.

6.4 Displaying binary values

GUI_DispBin()

Description

Displays a value in binary form at the current text position in the current window using the current font.

Prototype

```
void GUI_DispBin(U32 v, U8 Len);
```

Parameter	Meaning
v	Value to display, 32-bit.
Len	No. of digits to display (including leading zeros).

Add. information

As with decimal and hexadecimal values, the least significant bit is rightmost.

Example

```
//
// Show binary value 7, result: 000111
//
U32 Input = 0x7;
GUI_DispBin(Input, 6);
```

Related topics

GUI_DispBinAt()

GUI_DispBinAt()**Description**

Displays a value in binary form at a specified position in the current window using the current font.

Prototype

```
void DispBinAt(U32 v, I16P y, I16P x, U8 Len);
```

Parameter	Meaning
v	Value to display, 16-bit.
x	X-position to write to in pixels of the client window.
y	Y-position to write to in pixels of the client window.
Len	No. of digits to display (including leading zeroes).

Add. information

As with decimal and hexadecimal values, the least significant bit is rightmost.

Example

```
//
// Show binary input status
//
GUI_DispBinAt(Input, 0,0, 8);
```

Related topics

GUI_DispBin(), GUI_DispHex()

6.5 Displaying hexadecimal values**GUI_DispHex()****Description**

Displays a value in hexadecimal form at the current text position in the current window using the current font.

Prototype

```
void GUI_DispHex(U32 v, U8 Len);
```

Parameter	Meaning
v	Value to display, 16-bit.
Len	No. of digits to display.

Add. information

As with decimal and binary values, the least significant bit is rightmost.

Example

```
/* Show value of AD-converter */
GUI_DispHex(Input, 4);
```

Related topics

GUI_DispDec(), GUI_DispBin(), GUI_DispHexAt()

GUI_DispHexAt()**Description**

Displays a value in hexadecimal form at a specified position in the current window using the current font.

Prototype

```
void GUI_DispHexAt(U32 v, I16P x, I16P y, U8 Len);
```

Parameter	Meaning
v	Value to display, 16-bit.
x	X-position to write to in pixels of the client window.
y	Y-position to write to in pixels of the client window.
Len	No. of digits to display.

Add. information

As with decimal and binary values, the least significant bit is rightmost.

Example

```
//
// Show value of AD-converter at specified position
//
GUI_DispHexAt(Input, 0, 0, 4);
```

Related topics

GUI_DispDec(), GUI_DispBin(), GUI_DispHex()

6.6 Version of emWin**GUI_GetVersionString()****Description**

Returns a string containing the current version of emWin.

Prototype

```
const char * GUI_GetVersionString(void);
```

Example

```
//
// Displays the current version at the current cursor position
//
GUI_DispString(GUI_GetVersionString());
```

Chapter 7

2-D Graphic Library

emWin contains a complete 2-D graphic library which should be sufficient for most applications. The routines supplied with emWin can be used with or without clipping and are based on fast and efficient algorithms. Currently, only the `DrawArc()` function requires floating-point calculations.

7.1 Graphic API

The table below lists the available graphic-related routines in alphabetical order within their respective categories. Detailed descriptions can be found in the sections that follow.

Routine	Explanation
Drawing modes	
<code>GUI_GetDrawMode()</code>	Returns the current drawing mode.
<code>GUI_SetDrawMode()</code>	Sets the drawing mode.
Pen size	
<code>GUI_GetPenSize()</code>	Returns the current pen size in pixels.
<code>GUI_SetPenSize()</code>	Sets the pen size in pixels.
Query current client rectangle	
<code>GUI_GetClientRect()</code>	Returns the current available drawing area.
Basic drawing routines	
<code>GUI_ClearRect()</code>	Fills a rectangular area with the background color.
<code>GUI_DrawGradientV()</code>	Draws a rectangle filled with a vertical color gradient.
<code>GUI_DrawGradientH()</code>	Draws a rectangle filled with a horizontal color gradient.
<code>GUI_DrawPixel()</code>	Draws a single pixel.
<code>GUI_DrawPoint()</code>	Draws a point.
<code>GUI_DrawRect()</code>	Draws a rectangle.
<code>GUI_DrawRectEx()</code>	Draws a rectangle.
<code>GUI_DrawRoundedRect()</code>	Draws a rectangle with rounded corners.
<code>GUI_FillRect()</code>	Draws a filled rectangle.
<code>GUI_FillRectEx()</code>	Draws a filled rectangle.
<code>GUI_FillRoundedRect()</code>	Draws a filled rectangle with rounded corners.
<code>GUI_InvertRect()</code>	Invert a rectangular area.
Drawing bitmaps	
<code>GUI_DrawBitmap()</code>	Draws a bitmap.
<code>GUI_DrawBitmapEx()</code>	Draws a scaled bitmap.
<code>GUI_DrawBitmapExp()</code>	Draws a bitmap using additional parameters.
<code>GUI_DrawBitmapMag()</code>	Draws a magnified bitmap.
Drawing lines	
<code>GUI_DrawHLine()</code>	Draws a horizontal line.
<code>GUI_DrawLine()</code>	Draws a line from a specified startpoint to a specified endpoint (absolute coordinates).
<code>GUI_DrawLineRel()</code>	Draws a line from the current position to an endpoint specified by X- and Y-distances (relative coordinates).
<code>GUI_DrawLineTo()</code>	Draws a line from the current position to a specified endpoint.
<code>GUI_DrawPolyLine()</code>	Draws a polyline.
<code>GUI_DrawVLine()</code>	Draws a vertical line.
<code>GUI_GetLineStyle()</code>	Returns the current line style.
<code>GUI_MoveRel()</code>	Moves the line pointer relative to its current position.
<code>GUI_MoveTo()</code>	Moves the line pointer to the given position.
<code>GUI_SetLineStyle()</code>	Sets the current line style.
Drawing polygons	
<code>GUI_DrawPolygon()</code>	Draws the outline of a polygon.
<code>GUI_EnlargePolygon()</code>	Enlarges a polygon.
<code>GUI_FillPolygon()</code>	Draws a filled polygon.
<code>GUI_MagnifyPolygon()</code>	Magnifies a polygon.
<code>GUI_RotatePolygon()</code>	Rotates a polygon by a specified angle.
Drawing circles	
<code>GUI_DrawCircle()</code>	Draws the outline of a circle.

Routine	Explanation
<code>GUI_FillCircle()</code>	Draws a filled circle.
Drawing ellipses	
<code>GUI_DrawEllipse()</code>	Draws the outline of an ellipse.
<code>GUI_FillEllipse()</code>	Draws a filled ellipse.
Drawing arcs	
<code>GUI_DrawArc()</code>	Draws an arc.
Drawing a graph	
<code>GUI_DrawGraph()</code>	Draws a graph.
Drawing a pie chart	
<code>GUI_DrawPie()</code>	Draws a circle sector.
Saving and restoring the GUI-context	
<code>GUI_RestoreContext()</code>	Restores the GUI-context.
<code>GUI_SaveContext()</code>	Saves the GUI-context.
Clipping	
<code>GUI_SetClipRect()</code>	Sets the rectangle used for clipping.

7.2 Drawing modes

emWin can draw in NORMAL mode or in XOR mode. The default is NORMAL mode, in which the content of the display is overdrawn by the graphic. In XOR mode, the content of the display is inverted when it is overdrawn.

Restrictions associated with GUI_DRAWMODE_XOR

- XOR mode is only useful when using two displayed colors inside the active window or screen.
- Some drawing functions of emWin do not work precisely with this drawing mode. Generally, this mode works only with a pen size of one pixel. That means before using functions like `GUI_DrawLine()`, `GUI_DrawCircle()`, `GUI_DrawRect()` and so on, you must make sure that the pen size is set to 1 when you are working in XOR mode.
- When drawing bitmaps with a color depth greater than 1 bit per pixel (bpp) this drawing mode takes no effect.
- When using drawing functions such as `GUI_DrawPolyLine()` or multiple calls of `GUI_DrawLineTo()`, the fulcrums are inverted twice. The result is that these pixels remain in the background color.

GUI_GetDrawMode()

Description

Returns the current drawing mode.

Prototype

```
GUI_DRAWMODE GUI_GetDrawMode(void);
```

Return value

The currently selected drawing mode.

Add. information

For details about drawing modes please refer to the function `GUI_SetDrawMode()`.

GUI_SetDrawMode()

Description

Selects the specified drawing mode.

Prototype

```
GUI_DRAWMODE GUI_SetDrawMode(GUI_DRAWMODE mode);
```

Parameter	Meaning
<code>mode</code>	Drawing mode to set. May be a value returned by any routine which sets the drawing mode or one of the constants below.

Permitted values for parameter <code>mode</code>	
<code>GUI_DM_NORMAL</code>	Default: Draws points, lines, areas, bitmaps.
<code>GUI_DM_XOR</code>	Inverts points, lines, areas when overwriting the color of another object on the display.

Return value

The selected drawing mode.

Add. information

In addition to setting the drawing mode, this routine may also be used to restore a drawing mode that has previously been changed.

If using colors, an inverted pixel is calculated as follows:

New pixel color = number of colors - actual pixel color - 1.

Example

```
//
// Showing two circles, the second one XOR-combined with the first:
//
GUI_Clear();
GUI_SetDrawMode(GUI_DRAWMODE_NORMAL);
GUI_FillCircle(120, 64, 40);
GUI_SetDrawMode(GUI_DRAWMODE_XOR);
GUI_FillCircle(140, 84, 40);
```

Screen shot of above example



7.3 Query current client rectangle

GUI_GetClientRect()

Description

The current client rectangle depends on using the window manager or not. If using the window manager the function uses `WM_GetClientRect` to retrieve the client rectangle. If not using the window manager the client rectangle corresponds to the complete LCD display.

Prototype

```
void GUI_GetClientRect(GUI_RECT* pRect);
```

Parameter	Meaning
<code>pRect</code>	Pointer to <code>GUI_RECT</code> -structure to store result.

7.4 Pen size

The pen size determines the thickness of vector drawing operations line `GUI_DrawLine()`, `GUI_DrawCircle()` and so on. Please note that the pen size takes not effect on all drawing functions.

GUI_GetPenSize()

Description

Returns the current pen size.

Prototype

```
U8 GUI_GetPenSize(void);
```

GUI_SetPenSize()

Description

Sets the pen size to be used for further drawing operations.

Prototype

```
U8 GUI_SetPenSize(U8 PenSize);
```

Parameter	Meaning
<code>PenSize</code>	Pen size in pixels to be used.

Return value

Previous pen size.

Add information

The pen size should be ≥ 1 .

7.5 Basic drawing routines

The basic drawing routines allow drawing of individual points, horizontal and vertical lines and shapes at any position on the display. Any available drawing mode can be used. Since these routines are called frequently in most applications, they are optimized for speed as much as possible. For example, the horizontal and vertical line functions do not require the use of single-dot routines.

GUI_ClearRect()

Description

Clears a rectangular area at a specified position in the current window by filling it with the background color.

Prototype

```
void GUI_ClearRect(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
<code>x0</code>	Upper left X-position.
<code>y0</code>	Upper left Y-position.
<code>x1</code>	Lower right X-position.
<code>y1</code>	Lower right Y-position.

Related topics

`GUI_InvertRect()`, `GUI_FillRect()`

GUI_DrawGradientH()

Description

Draws a rectangle filled with a horizontal color gradient.

Prototype

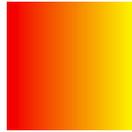
```
void GUI_DrawGradientH(int x0, int y0, int x1, int y1,
                      GUI_COLOR Color0, GUI_COLOR Color1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.
Color0	Color to be drawn on the leftmost side of the rectangle.
Color1	Color to be drawn on the rightmost side of the rectangle.

Example

```
GUI_DrawGradientH(0, 0, 99, 99, 0x0000FF, 0x00FFFF);
```

Screenshot of above example



GUI_DrawGradientV()

Description

Draws a rectangle filled with a vertical color gradient.

Prototype

```
void GUI_DrawGradientV(int x0, int y0, int x1, int y1,
                      GUI_COLOR Color0, GUI_COLOR Color1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.
Color0	Color to be drawn on the leftmost side of the rectangle.
Color1	Color to be drawn on the rightmost side of the rectangle.

Example

```
GUI_DrawGradientV(0, 0, 99, 99, 0x0000FF, 0x00FFFF);
```

Screenshot of above example



GUI_DrawPixel()

Description

Draws a pixel at a specified position in the current window.

Prototype

```
void GUI_DrawPixel(int x, int y);
```

Parameter	Meaning
x	X-position of pixel.
y	Y-position of pixel.

Related topics

[GUI_DrawPoint\(\)](#)

GUI_DrawPoint()**Description**

Draws a point with the current pen size at a specified position in the current window.

Prototype

```
void GUI_DrawPoint(int x, int y);
```

Parameter	Meaning
x	X-position of point.
y	Y-position of point.

Related topics

[GUI_DrawPixel\(\)](#)

GUI_DrawRect()**Description**

Draws a rectangle at a specified position in the current window.

Prototype

```
void GUI_DrawRect(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.

GUI_DrawRectEx()**Description**

Draws a rectangle at a specified position in the current window.

Prototype

```
void GUI_DrawRectEx(const GUI_RECT *pRect);
```

Parameter	Meaning
pRect	Pointer to a GUI_RECT-structure containing the coordinates of the rectangle

GUI_DrawRoundedRect()**Description**

Draws a rectangle at a specified position in the current window with rounded corners.

Prototype

```
void GUI_DrawRoundedRect(int x0, int y0, int x1, int y1, int r);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.
r	Radius to be used for the rounded corners.

GUI_FillRect()**Description**

Draws a filled rectangular area at a specified position in the current window.

Prototype

```
void GUI_FillRect(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.

Add. information

Uses the current drawing mode, which normally means all pixels inside the rectangle are set.

Related topics

GUI_InvertRect(), GUI_ClearRect()

GUI_FillRectEx()**Description**

Draws a filled rectangle at a specified position in the current window.

Prototype

```
void GUI_FillRectEx (const GUI_RECT* pRect);
```

Parameter	Meaning
pRect	Pointer to a GUI_RECT-structure containing the coordinates of the rectangle

GUI_FillRoundedRect()**Description**

Draws a filled rectangle at a specified position in the current window with rounded corners.

Prototype

```
void GUI_FillRoundedRect(int x0, int y0, int x1, int y1, int r);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.

Parameter	Meaning
x1	Lower right X-position.
y1	Lower right Y-position.
r	Radius to be used for the rounded corners.

GUI_InvertRect()

Description

Draws an inverted rectangular area at a specified position in the current window.

Prototype

```
void GUI_InvertRect(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.

Related topics

GUI_FillRect(), GUI_ClearRect()

7.6 Drawing bitmaps

Generally emWin is able to display any bitmap image at any display position. On 16 bit CPUs (sizeof(int) == 2), the size of one bitmap per default is limited to 64Kb. If larger bitmaps should be displayed with a 16 bit CPU please refer to chapter "High-Level Configuration\Available GUI configuration macros".

GUI_DrawBitmap()

Description

Draws a bitmap image at a specified position in the current window.

Prototype

```
void GUI_DrawBitmap(const GUI_BITMAP* pBM, int x, int y);
```

Parameter	Meaning
pBM	Pointer to the bitmap to display.
x	X-position of the upper left corner of the bitmap in the display.
y	Y-position of the upper left corner of the bitmap in the display.

Add. information

The picture data is interpreted as bitstream starting with the most significant bit (msb) of the first byte.

A new line always starts at an even byte address, as the nth line of the bitmap starts at offset n*BytesPerLine. The bitmap can be shown at any point in the client area. Usually, the bitmap converter is used to generate bitmaps. For more information, please refer to Chapter 9: "Bitmap Converter".

Example

```
extern const GUI_BITMAP bmSeggerLogoBlue; /* declare external Bitmap */

void main() {
    GUI_Init();
    GUI_DrawBitmap(&bmSeggerLogoBlue, 45, 20);
}
```

Screen shot of above example



GUI_DrawBitmapExp()

Description

Same function as `GUI_DrawBitmap()`, but with additional parameters.

Prototype

```
void GUI_DrawBitmapExp(int x0, int y0,
                      int XSize, int YSize,
                      int XMul, int YMul,
                      int BitsPerPixel,
                      int BytesPerLine,
                      const U8* pData,
                      const GUI_LOGPALETTE* pPal);
```

Parameter	Meaning
<code>x0</code>	X-position of the upper left corner of the bitmap in the display.
<code>y0</code>	Y-position of the upper left corner of the bitmap in the display.
<code>Xsize</code>	Number of pixels in horizontal direction. Valid range: 1... 255.
<code>Ysize</code>	Number of pixels in vertical direction. Valid range: 1... 255.
<code>XMUL</code>	Scale factor of X-direction.
<code>YMul</code>	Scale factor of Y-direction.
<code>BitsPerPixel</code>	Number of bits per pixel.
<code>BytesPerLine</code>	Number of bytes per line of the image.
<code>pData</code>	Pointer to the actual image, the data that defines what the bitmap looks like.
<code>pPal</code>	Pointer to a <code>GUI_LOGPALETTE</code> structure.

GUI_DrawBitmapEx()

Description

This routine makes it possible to scale and/or to mirror a bitmap on the display.

Prototype

```
void GUI_DrawBitmapEx(const GUI_BITMAP* pBitmap,
                     int x0, int y0,
                     int xCenter, int yCenter,
                     int xMag, int yMag);
```

Parameter	Meaning
<code>pBM</code>	Pointer to the bitmap to display.
<code>x0</code>	X-position of the anchor point in the display.
<code>y0</code>	Y-position of the anchor point in the display.
<code>xCenter</code>	X-position of the anchor point in the bitmap.
<code>yCenter</code>	Y-position of the anchor point in the bitmap.
<code>xMag</code>	Scale factor of X-direction.
<code>yMag</code>	Scale factor of Y-direction.

Add. information

A negative value of the `xMag`-parameter would mirror the bitmap in the X-axis and a negative value of the `yMag`-parameter would mirror the bitmap in the Y-axis. The unit of `xMag`- and `yMag` are thousandth. The position given by the parameter `xCenter` and `yCenter` specifies the pixel of the bitmap which should be displayed at the display at position `x0/y0` independent of scaling or mirroring.

This function can not be used to draw RLE-compressed bitmaps.

GUI_DrawBitmapMag()**Description**

This routine makes it possible to magnify a bitmap on the display.

Prototype

```
void GUI_DrawBitmapMag(const GUI_BITMAP* pBM,
                      int x0, int y0,
                      int XMul, int YMul);
```

Parameter	Meaning
<code>pBM</code>	Pointer to the bitmap to display.
<code>x0</code>	X-position of the upper left corner of the bitmap in the display.
<code>y0</code>	Y-position of the upper left corner of the bitmap in the display.
<code>XMul</code>	Magnification factor of X-direction.
<code>YMul</code>	Magnification factor of Y-direction.

GUI_DrawStreamedBitmap()**Description**

Draws a bitmap from a data bitmap data stream.

Prototype

```
void GUI_DrawStreamedBitmap(const GUI_BITMAP_STREAM* pBMH, int x, int y);
```

Parameter	Meaning
<code>pBMH</code>	Pointer to the data stream.
<code>x</code>	X-position of the upper left corner of the bitmap in the display.
<code>y</code>	Y-position of the upper left corner of the bitmap in the display.

Add. information

You can use the bitmap converter (Chapter 9) to create bitmap data streams. The format of these streams is not the same as the format of a `.bmp` file.

7.7 Drawing lines

The most frequently used drawing routines are those that draw a line from one point to another.

GUI_DrawHLine()**Description**

Draws a horizontal line one pixel thick from a specified starting point to a specified endpoint in the current window.

Prototype

```
void GUI_DrawHLine(int y, int x0, int x1);
```

Parameter	Meaning
y	Y-position.
x0	X-starting position.
x1	X-end position.

Add. information

If $x1 < x0$, nothing will be displayed.

With most LCD controllers, this routine is executed very quickly because multiple pixels can be set at once and no calculations are needed. If it is clear that horizontal lines are to be drawn, this routine executes faster than the `GUI_DrawLine()` routine.

GUI_DrawLine()

Description

Draws a line from a specified starting point to a specified endpoint in the current window (absolute coordinates).

Prototype

```
void GUI_DrawLine(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
x0	X-starting position.
y0	Y-starting position.
x1	X-end position.
y1	Y-end position.

Add. information

If part of the line is not visible because it is not in the current window or because part of the current window is not visible, this is due to clipping.

GUI_DrawLineRel()

Description

Draws a line from the current (X,Y) position to an endpoint specified by X-distance and Y-distance in the current window (relative coordinates).

Prototype

```
void GUI_DrawLineRel(int dx, int dy);
```

Parameter	Meaning
dx	Distance in X-direction to end of line to draw.
dy	Distance in Y-direction end of line to draw.

GUI_DrawLineTo()

Description

Draws a line from the current (X,Y) position to an endpoint specified by X- and Y-coordinates in the current window.

Prototype

```
void GUI_DrawLineTo(int x, int y);
```

Parameter	Meaning
<code>x</code>	X-end position.
<code>y</code>	Y-end position.

GUI_DrawPolyLine()**Description**

Connects a predefined list of points with lines in the current window.

Prototype

```
void GUI_DrawPolyLine(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

Parameter	Meaning
<code>pPoint</code>	Pointer to the polyline to display.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>x</code>	X-position of origin.
<code>y</code>	Y-position of origin.

Add. information

The starting point and endpoint of the polyline need not be identical.

GUI_DrawVLine()**Description**

Draws a vertical line one pixel thick from a specified starting point to a specified endpoint in the current window.

Prototype

```
void GUI_DrawVLine(int x, int y0, int y1);
```

Parameter	Meaning
<code>x</code>	X-position.
<code>y0</code>	Y-starting position.
<code>y1</code>	Y-end position.

Add. information

If `y1 < y0`, nothing will be displayed.

With most LCD controllers, this routine is executed very quickly because multiple pixels can be set at once and no calculations are needed. If it is clear that vertical lines are to be drawn, this routine executes faster than the `GUI_DrawLine()` routine.

GUI_GetLineStyle()**Description**

Returns the current line style used by the function `GUI_DrawLine`.

Prototype

```
U8 GUI_GetLineStyle (void);
```

Return value

Current line style used by the function `GUI_DrawLine`.

GUI_MoveRel()

Description

Moves the current line pointer relative to its current position.

Prototype

```
void GUI_MoveRel(int dx, int dy);
```

Parameter	Meaning
dx	Distance to move in X.
dy	Distance to move in Y.

Related topics

[GUI_DrawLineTo\(\)](#), [GUI_MoveTo\(\)](#)

GUI_MoveTo()

Description

Moves the current line pointer to the given position.

Prototype

```
void GUI_MoveTo(int x, int y);
```

Parameter	Meaning
x	New position in X.
y	New position in Y.

GUI_SetLineStyle()

Description

Sets the current line style used by the function [GUI_DrawLine](#).

Prototype

```
U8 GUI_SetLineStyle(U8 LineStyle);
```

Parameter	Meaning
LineStyle	New line style to be used (see table below).

Permitted values for parameter LineStyle	
GUI_LS_SOLID	Lines would be drawn solid (default).
GUI_LS_DASH	Lines would be drawn dashed.
GUI_LS_DOT	Lines would be drawn dotted.
GUI_LS_DASHDOT	Lines would be drawn alternating with dashes and dots.
GUI_LS_DASHDOTDOT	Lines would be drawn alternating with dashes and double dots.

Return value

Previous line style used by the function [GUI_DrawLine](#).

Add. information

This function sets only the line style used by [GUI_DrawLine](#). The style will be used only with a pen size of 1.

7.8 Drawing polygons

The polygon drawing routines can be helpful when drawing vectorized symbols.

GUI_DrawPolygon()

Description

Draws the outline of a polygon defined by a list of points in the current window.

Prototype

```
void GUI_DrawPolygon(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

Parameter	Meaning
<code>pPoint</code>	Pointer to the polygon to display.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>x</code>	X-position of origin.
<code>y</code>	Y-position of origin.

Add. information

The polyline drawn is automatically closed by connecting the endpoint to the starting point.

GUI_EnlargePolygon()

Description

Enlarges a polygon on all sides by a specified length in pixels.

Prototype

```
void GUI_EnlargePolygon(GUI_POINT* pDest,
                       const GUI_POINT* pSrc,
                       int NumPoints, int Len);
```

Parameter	Meaning
<code>pDest</code>	Pointer to the destination polygon.
<code>pSrc</code>	Pointer to the source polygon.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>Len</code>	Length (in pixels) by which to enlarge the polygon.

Add. information

Make sure the destination array of points is equal to or larger than the source array.

Example

```
#define countof(Array) (sizeof(Array) / sizeof(Array[0]))

const GUI_POINT aPoints[] = {
    { 0, 20},
    { 40, 20},
    { 20, 0}
};

GUI_POINT aEnlargedPoints[countof(aPoints)];

void Sample(void) {
    int i;
    GUI_Clear();
    GUI_SetDrawMode(GUI_DM_XOR);
    GUI_FillPolygon(aPoints, countof(aPoints), 140, 110);
    for (i = 1; i < 10; i++) {
        GUI_EnlargePolygon(aEnlargedPoints, aPoints, countof(aPoints), i * 5);
        GUI_FillPolygon(aEnlargedPoints, countof(aPoints), 140, 110);
    }
}
```

Screen shot of above example



GUI_FillPolygon()

Description

Draws a filled polygon defined by a list of points in the current window.

Prototype

```
void GUI_FillPolygon(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

Parameter	Meaning
<code>pPoint</code>	Pointer to the polygon to display and to fill.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>x</code>	X-position of origin.
<code>y</code>	Y-position of origin.

Add. information

The polyline drawn is automatically closed by connecting the endpoint to the starting point. It is not required that the endpoint touches the outline of the polygon. Rendering a polygon is done by drawing one or more horizontal lines for each y-position of the polygon. Per default the maximum number of points used to draw the horizontal lines for one y-position is 12 (which means 6 lines per y-position). If this value needs to be increased, the macro `GUI_FP_MAXCOUNT` can be used to set the maximum number of points.

Example

```
#define GUI_FP_MAXCOUNT 50
```

GUI_MagnifyPolygon()

Description

Magnifies a polygon by a specified factor.

Prototype

```
void GUI_MagnifyPolygon(GUI_POINT* pDest,
                        const GUI_POINT* pSrc,
                        int NumPoints, int Mag);
```

Parameter	Meaning
<code>pDest</code>	Pointer to the destination polygon.
<code>pSrc</code>	Pointer to the source polygon.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>Mag</code>	Factor used to magnify the polygon.

Add. information

Make sure the destination array of points is equal to or larger than the source array.

Note the difference between enlarging and magnifying a polygon. Whereas setting the parameter `Len` to 1 will enlarge the polygon by one pixel on all sides, setting the parameter `Mag` to 1 will have no effect.

Example

```
#define countof(Array) (sizeof(Array) / sizeof(Array[0]))

const GUI_POINT aPoints[] = {
    { 0, 20},
    { 40, 20},
    { 20, 0}
};

GUI_POINT aMagnifiedPoints[countof(aPoints)];

void Sample(void) {
    int Mag, y = 0, Count = 4;
    GUI_Clear();
    GUI_SetColor(GUI_GREEN);
    for (Mag = 1; Mag <= 4; Mag *= 2, Count /= 2) {
        int i, x = 0;
        GUI_MagnifyPolygon(aMagnifiedPoints, aPoints, countof(aPoints), Mag);
        for (i = Count; i > 0; i--, x += 40 * Mag) {
            GUI_FillPolygon(aMagnifiedPoints, countof(aPoints), x, y);
        }
        y += 20 * Mag;
    }
}
```

Screen shot of above example



GUI_RotatePolygon()

Description

Rotates a polygon by a specified angle.

Prototype

```
void GUI_RotatePolygon(GUI_POINT* pDest,
                      const GUI_POINT* pSrc,
                      int NumPoints,
                      float Angle);
```

Parameter	Meaning
<code>pDest</code>	Pointer to the destination polygon.
<code>pSrc</code>	Pointer to the source polygon.
<code>NumPoints</code>	Number of points specified in the list of points.
<code>Angle</code>	Angle in radian used to rotate the polygon.

Add. information

Make sure the destination array of points is equal to or larger than the source array.

Example

The following example shows how to draw a polygon. It is available as 2DGL_DrawPolygon.c in the samples shipped with emWin.

```

/*****
*          SEGGER MICROCONTROLLER SYSTEME GmbH          *
*          Solutions for real time microcontroller applications *
*
*          emWin sample code                             *
*
*****/

-----
File      : 2DGL_DrawPolygon.c
Purpose   : Example for drawing a polygon
-----
*/

#include "gui.h"

/*****
*
*          The points of the arrow
*
*****/

static const GUI_POINT aPointArrow[] = {
    { 0, -5},
    {-40, -35},
    {-10, -25},
    {-10, -85},
    { 10, -85},
    { 10, -25},
    { 40, -35},
};

/*****
*
*          Draws a polygon
*
*****/

static void DrawPolygon(void) {
    int Cnt =0;
    GUI_SetBkColor(GUI_WHITE);
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_SetColor(0x0);
    GUI_DispStringAt("Polygons of arbitrary shape ", 0, 0);
    GUI_DispStringAt("in any color", 120, 20);
    GUI_SetColor(GUI_BLUE);
    /* Draw filled polygon */
    GUI_FillPolygon (&aPointArrow[0],7,100,100);
}

/*****
*
*          main
*
*****/

void main(void) {
    GUI_Init();
    DrawPolygon();
    while(1)
        GUI_Delay(100);
}

```

Screen shot of above example



7.9 Drawing circles

GUI_DrawCircle()

Description

Draws the outline of a circle of specified dimensions, at a specified position in the current window.

Prototype

```
void GUI_DrawCircle(int x0, int y0, int r);
```

Parameter	Meaning
x0	X-position of the center of the circle in pixels of the client window.
y0	Y-position of the center of the circle in pixels of the client window.
r	Radius of the circle (half the diameter). Minimum: 0 (will result in a point); maximum: 180.

Add. information

This routine cannot handle a radius in excess of 180 because it uses integer calculations that would otherwise produce an overflow. However, for most embedded applications this is not a problem since a circle with diameter 360 is larger than the display anyhow.

Example

```
// Draw concentric circles
void ShowCircles(void) {
    int i;
    for (i=10; i<50; i++)
        GUI_DrawCircle(120,60,i);
}
```

Screen shot of above example



GUI_FillCircle()

Description

Draws a filled circle of specified dimensions at a specified position in the current window.

Prototype

```
void GUI_FillCircle(int x0, int y0, int r);
```

Parameter	Meaning
x0	X-position of the center of the circle in pixels of the client window.
y0	Y-position of the center of the circle in pixels of the client window.
r	Radius of the circle (half the diameter). Minimum: 0 (will result in a point); maximum: 180.

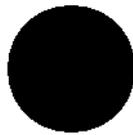
Add. information

This routine cannot handle a radius in excess of 180.

Example

```
GUI_FillCircle(120,60,50);
```

Screen shot of above example



7.10 Drawing ellipses

GUI_DrawEllipse()

Description

Draws the outline of an ellipse of specified dimensions, at a specified position in the current window.

Prototype

```
void GUI_DrawEllipse (int x0, int y0, int rx, int ry);
```

Parameter	Meaning
x0	X-position of the center of the circle in pixels of the client window.
y0	Y-position of the center of the circle in pixels of the client window.
rx	X-radius of the ellipse (half the diameter). Minimum: 0; maximum: 180.
ry	Y-radius of the ellipse (half the diameter). Minimum: 0; maximum: 180.

Add. information

This routine cannot handle rx/ry parameters in excess of 180 because it uses integer calculations that would otherwise produce an overflow.

Example

See the GUI_FillEllipse() example.

GUI_FillEllipse()

Description

Draws a filled ellipse of specified dimensions at a specified position in the current window.

Prototype

```
void GUI_FillEllipse(int x0, int y0, int rx, int ry);
```

Parameter	Meaning
<code>x0</code>	X-position of the center of the circle in pixels of the client window.
<code>y0</code>	Y-position of the center of the circle in pixels of the client window.
<code>rx</code>	X-radius of the ellipse (half the diameter). Minimum: 0; maximum: 180.
<code>ry</code>	Y-radius of the ellipse (half the diameter). Minimum: 0; maximum: 180.

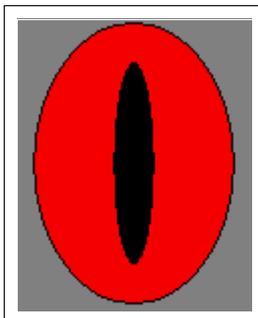
Add. information

This routine cannot handle a rx/ry parameters in excess of 180.

Example

```
/*
 * Demo ellipses
 */
GUI_SetColor(0xff);
GUI_FillEllipse(100, 180, 50, 70);
GUI_SetColor(0x0);
GUI_DrawEllipse(100, 180, 50, 70);
GUI_SetColor(0x000000);
GUI_FillEllipse(100, 180, 10, 50);
```

Screen shot of above example



7.11 Drawing arcs

GUI_DrawArc()

Description

Draws an arc of specified dimensions at a specified position in the current window. An arc is a section of the outline of a circle.

Prototype

```
void GL_DrawArc (int xCenter, int yCenter, int rx, int ry, int a0, int a1);
```

Parameter	Meaning
<code>xCenter</code>	Horizontal position of the center in pixels of the client window.
<code>yCenter</code>	Vertical position of the center in pixels of the client window.
<code>rx</code>	X-radius (pixels).
<code>ry</code>	Y-radius (pixels).
<code>a0</code>	Starting angle (degrees).
<code>a1</code>	Ending angle (degrees).

Limitations

Currently the ry parameter is not used. The rx parameter is used instead.

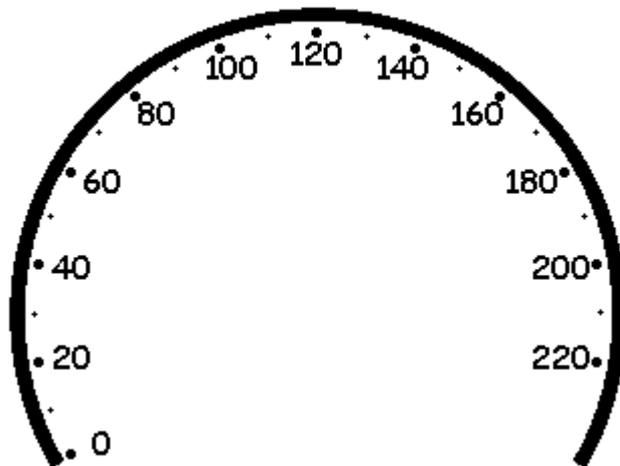
Add. information

GUI_DrawArc() uses the floating-point library. It cannot handle rx/ry parameters in excess of 180 because it uses integer calculations that would otherwise produce an overflow.

Example

```
void DrawArcScale(void) {
    int x0 = 160;
    int y0 = 180;
    int i;
    char ac[4];
    GUI_SetBkColor(GUI_WHITE);
    GUI_Clear();
    GUI_SetPenSize( 5 );
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_SetFont(&GUI_FontComi18B_ASCII);
    GUI_SetColor( GUI_BLACK );
    GUI_DrawArc( x0,y0,150, 150,-30, 210 );
    GUI_Delay(1000);
    for (i=0; i<= 23; i++) {
        float a = (-30+i*10)*3.1415926/180;
        int x = -141*cos(a)+x0;
        int y = -141*sin(a)+y0;
        if (i%2 == 0)
            GUI_SetPenSize( 5 );
        else
            GUI_SetPenSize( 4 );
        GUI_DrawPoint(x,y);
        if (i%2 == 0) {
            x = -123*cos(a)+x0;
            y = -130*sin(a)+y0;
            sprintf(ac, "%d", 10*i);
            GUI_SetTextAlign(GUI_TA_VCENTER);
            GUI_DispStringHCenterAt(ac,x,y);
        }
    }
}
```

Screen shot of above example



7.12 Drawing graphs

GUI_DrawGraph()

Description

Draws a graph at once.

Prototype

```
void GUI_DrawGraph(I16 *paY, int NumPoints, int x0, int y0);
```

Parameter	Meaning
paY	Pointer to an array containing the Y-values of the graph.
NumPoints	Number of Y-values to be displayed.
x0	Starting point in x.
y0	Starting point in y.

Add. information

The function first sets the line-cursor to the position specified with x0, y0 and the first Y-value of the given array. Then it starts drawing lines to x0 + 1, y0 + *(paY + 1), x0 + 2, y0 + *(paY + 2) and so on.

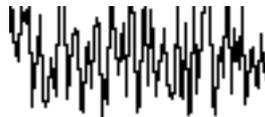
Example

```
#include "GUI.h"
#include <stdlib.h>

I16 aY[100];

void MainTask(void) {
    int i;
    GUI_Init();
    for (i = 0; i < GUI_COUNTOF(aY); i++) {
        aY[i] = rand() % 50;
    }
    GUI_DrawGraph(aY, GUI_COUNTOF(aY), 0, 0);
}
```

Screen shot of above example



7.13 Drawing pie charts

GUI_DrawPie()

Description

Draws a circle sector.

Prototype

```
void GUI_DrawPie(int x0, int y0, int r, int a0, int a1, int Type);
```

Parameter	Meaning
x0	X-position of the center of the circle in pixels of the client window.
y0	Y-position of the center of the circle in pixels of the client window.
r	Radius of the circle (half the diameter).
a0	Starting angle (degrees).
a1	End angle (degrees).
Type	(reserved for future use, should be 0)

Example

```
int i, a0, a1;
const unsigned aValues[] = {100, 135, 190, 240, 340, 360};
const GUI_COLOR aColors[] = { GUI_BLUE, GUI_GREEN, GUI_RED,
                             GUI_CYAN, GUI_MAGENTA, GUI_YELLOW };
for (i = 0; i < GUI_COUNTOF(aValues); i++) {
```

```

a0 = (i == 0) ? 0 : aValues[i - 1];
a1 = aValues[i];
GUI_SetColor(aColors[i]);
GUI_DrawPie(100, 100, 50, a0, a1, 0);
}

```

Screen shot of above example



7.14 Saving and restoring the GUI-context

GUI_RestoreContext()

Description

The function restores the GUI-context.

Prototype

```
void GUI_RestoreContext(const GUI_CONTEXT* pContext);
```

Parameter	Meaning
pContext	Pointer to a GUI_CONTEXT structure containing the new context.

Add. information

The GUI-context contains the current state of the GUI like the text cursor position, a pointer to the current font and so on. Sometimes it could be usefull to save the current state ant to restore it later. For this you can use these functions.

GUI_SaveContext()

Description

The function saves the current GUI-context. (See also GUI_RestoreContext)

Prototype

```
void GUI_SaveContext(GUI_CONTEXT* pContext);
```

Parameter	Meaning
pContext	Pointer to a GUI_CONTEXT structure for saving the current context.

7.15 Clipping

GUI_SetClipRect()

Description

Sets the clipping rectangle used for limiting the output.

Prototype

```
void GUI_SetClipRect(const GUI_RECT* pRect);
```

Parameter	Meaning
pRect	Pointer to the rectangle which should be used for clipping. A NULL pointer should be used to restore the default value.

Add. information

The clipping area is per default limited to the configured (virtual) display size. Under some circumstances it can be useful to use a smaller clipping rectangle, which can be set using this function.

The rectangle referred to should remain unchanged until the function is called again with a NULL pointer.

Sample

The following sample shows how to use the function:

```
GUI_RECT Rect = {10, 10, 100, 100};
GUI_SetClipRect(&Rect);
. /* Use the clipping area ... */
.
GUI_SetClipRect(NULL);
```


Chapter 8

Fonts

This chapter describes the various methods of font support in emWin. The most common fonts are shipped with emWin as 'C' font files. All of them contain the ASCII character set and most of them also the characters of ISO 8859-1. In fact, you will probably find that these fonts are fully sufficient for your application. For detailed information on the individual fonts, please refer to the subchapter "Standard Fonts", which describes all fonts included with emWin and shows all characters as they appear on the display.

emWin is compiled for 8-bit characters, allowing for a maximum of 256 different character codes out of which the first 32 are reserved as control characters. The characters that are available depends on the selected font.

For accessing the full Unicode area of 65536 possible characters emWin supports UTF8 decoding which is described in chapter 14 "Foreign Language Support".

8.1 Introduction

The first way of font support was the possibility to use 'C' files with font definitions containing bitmaps with 1bpp pixel information for each character. This kind of font support was limited to use only the fonts which are compiled with the application.

8.2 Font types

Contrary to emWin, emWin 8051 does not support different font types. The only font type supported is "Proportional bitmap".

Proportional bitmap fonts

Each character of a proportional bitmap font has the same height and its own width. The pixel information is saved with 1bpp.

8.3 Font formats

The following explains the differences between the supported font formats, when to use them and what is required to be able to use them.

8.3.1 'C' file format

This is the most common way of using fonts. When using fonts in form of 'C' files, we recommend compiling all available fonts and linking them as library modules or putting all of the font object files in a library which you can link with your application. This way you can be sure that only the fonts which are needed by your application are actually linked. The font converter (described in a separate manual) may be used to create additional fonts.

When to use

This format should be used if the fonts are known at compile time and if there is enough addressable memory available for the font data.

Requirements

In order to be able to use a font 'C' file in your application, the following requirements must be met:

- The font file is in a form compatible with emWin as "C" file, object file or library.
- The font file is linked with your application.
- The font declaration is contained in the application.

Format description

A font 'C' file contains at first the pixel information of all characters included by the font. It is followed by a character information table with size information about each character. This table is followed by range information structures for each contiguous area of characters contained in the font file, whereas each structure points to the next one. Please note that this method can enlarge a font file a lot if using many separate characters. After the range information structures a `GUI_FONT` structure follows with the main information like type, pixel size and so on of the font.

8.4 Declaring custom fonts

The most recommended way of declaring the prototypes of custom fonts is to put them into an application defined header file. This should be included from each application source file which uses these fonts. It could look like the following sample:

```
#include "GUI.h"

extern GUI_CONST_STORAGE GUI_FONT GUI_FontApp1;
```

```
extern GUI_CONST_STORAGE GUI_FONT GUI_FontApp2;
```

Please note that this kind of declaring prototypes does not work if the fonts should be used with emWin configuration macros like `BUTTON_FONT_DEFAULT` or similar. In this case the fonts need to be declared in the configuration file `GUIConf.h`. The declaration in this case can look like the following sample:

```
typedef struct GUI_FONT GUI_FONT;

extern const GUI_FONT GUI_FontApp1;

#define BUTTON_FONT_DEFAULT &GUI_FontApp1
#define EDIT_FONT_DEFAULT &GUI_FontApp1
```

The `typedef` is required because the structure `GUI_FONT` has not been defined at the early point where `GUIConf.h` is included by emWin.

8.5 Selection of a font

emWin offers different fonts, one of which is always selected. This selection can be changed by calling the function `GUI_SetFont()` or one of the `GUI_XXX_CreateFont()` functions, which select the font to use for all text output to follow for the current task.

If no font has been selected by your application, the default font is used. This default is configured in `GUIConf.h` and can be changed. You should make sure that the default font is one that you are actually using in your application because the default font will be linked with your application and will therefore use up ROM memory.

8.6 Font API

The table below lists the available font-related routines in alphabetical order within their respective categories. Detailed descriptions can be found in the sections that follow.

Routine	Explanation
'C' file related font functions	
GUI_SetFont()	Sets the current font.
Common font-related functions	
GUI_GetCharDistX()	Returns the width in pixels (X-size) of a specified character in the current font.
GUI_GetFont()	Returns a pointer to the currently selected font.
GUI_GetFontDistY()	Returns the Y-spacing of the current font.
GUI_GetFontInfo()	Returns a structure containing font information.
GUI_GetFontSizeY()	Returns the height in pixels (Y-size) of the current font.
GUI_GetStringDistX()	Returns the X-size of a text using the current font.
GUI_GetTextExtend()	Evaluates the size of a text using the current font
GUI_GetYDistOfFont()	Returns the Y-spacing of a particular font.
GUI_GetYSizeOfFont()	Returns the Y-size of a particular font.
GUI_IsInFont()	Evaluates whether a specified character is in a particular font.

8.7 'C' file related font functions

GUI_SetFont()

Description

Sets the font to be used for text output.

Prototype

```
const GUI_FONT * GUI_SetFont(const GUI_FONT * pNewFont);
```

Parameter	Meaning
<code>pFont</code>	Pointer to the font to be selected and used.

Return value

Returns a pointer to the previously selected font so that it may be restored at a later point.

Examples

Displays sample text in 3 different sizes, restoring the former font afterwards:

```
void DispText(void) {
    const GUI_FONT GUI_FLASH* OldFont=GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("This text is 8 by 16 pixels",0,0);
    GUI_SetFont(&GUI_Font6x8);
    GUI_DispStringAt("This text is 6 by 8 pixels",0,20);
    GUI_SetFont(&GUI_Font8);
    GUI_DispStringAt("This text is proportional",0,40);
    GUI_SetFont(OldFont); // Restore font
}
```

Screen shot of above example:

```

This text is 8 by 16 pixels
This text is 6 by 8 pixels
This text is proportional
```

Displays text and value in different fonts:

```
GUI_SetFont(&GUI_Font6x8);
GUI_DispString("The result is: "); // Disp text
GUI_SetFont(&GUI_Font8x8);
GUI_DispDec(42,2); // Disp value
```

Screen shot of above example:

```
The result is: 42
```

8.8 Common font-related functions

GUI_GetFont()

Description

Returns a pointer to the currently selected font.

Prototype

```
const GUI_FONT * GUI_GetFont(void)
```

GUI_GetCharDistX()

Description

Returns the width in pixels (X-size) used to display a specified character in the currently selected font.

Prototype

```
int GUI_GetCharDistX(U16 c);
```

Parameter	Meaning
<code>c</code>	Character to calculate width from.

GUI_GetFontDistY()

Description

Returns the Y-spacing of the currently selected font.

Prototype

```
int GUI_GetFontDistY(void);
```

Add. information

The Y-spacing is the vertical distance in pixels between two adjacent lines of text. The returned value is the `YDist` value of the entry for the currently selected font. The returned value is valid for both proportional and monospaced fonts.

GUI_GetFontInfo()

Description

Calculates a pointer to a `GUI_FONTINFO` structure of a particular font.

Prototype

```
void GUI_GetFontInfo(const GUI_FONT* pFont, GUI_FONTINFO* pfi);
```

Parameter	Meaning
<code>pFont</code>	Pointer to the font.
<code>pfi</code>	Pointer to a <code>GUI_FONTINFO</code> structure.

Add. information

The definition of the `GUI_FONTINFO` structure is as follows:

```
typedef struct {
    U16 Flags;
} GUI_FONTINFO;
```

The member variable flags can take the following values:

```
GUI_FONTINFO_FLAG_PROP
GUI_FONTINFO_FLAG_MONO
GUI_FONTINFO_FLAG_AA
GUI_FONTINFO_FLAG_AA2
GUI_FONTINFO_FLAG_AA4
```

Example

Gets the info of `GUI_Font6x8`. After the calculation, `FontInfo.Flags` contains the flag `GUI_FONTINFO_FLAG_MONO`.

```
GUI_FONTINFO FontInfo;
GUI_GetFontInfo(&GUI_Font6x8, &FontInfo);
```

GUI_GetFontSizeY()

Description

Returns the height in pixels (Y-size) of the currently selected font.

Prototype

```
int GUI_GetFontSizeY(void);
```

Add. information

The returned value is the `YSize` value of the entry for the currently selected font. This value is less than or equal to the Y-spacing returned by the function `GUI_GetFontDistY()`.

The returned value is valid for both proportional and monospaced fonts.

GUI_GetStringDistX()

Description

Returns the X-size used to display a specified string in the currently selected font.

Prototype

```
int GUI_GetStringDistX(const char GUI_FAR *s);
```

Parameter	Meaning
<code>s</code>	Pointer to the string.

GUI_GetTextExtend()

Description

Calculates the size of a given string using the current font.

Prototype

```
void GUI_GetTextExtend(GUI_RECT* pRect, const char* s, int Len);
```

Parameter	Meaning
<code>pRect</code>	Pointer to GUI_RECT-structure to store result.
<code>s</code>	Pointer to the string.
<code>Len</code>	Number of characters of the string.

GUI_GetYDistOfFont()

Description

Returns the Y-spacing of a particular font.

Prototype

```
int GUI_GetYDistOfFont(const GUI_FONT* pFont);
```

Parameter	Meaning
<code>pFont</code>	Pointer to the font.

Add. information

(see GUI_GetFontDistY())

GUI_GetYSizeOfFont()

Description

Returns the Y-size of a particular font.

Prototype

```
int GUI_GetYSizeOfFont(const GUI_FONT* pFont);
```

Parameter	Meaning
<code>pFont</code>	Pointer to the font.

Add. information

(see GUI_GetFontSizeY())

GUI_IsInFont()

Description

Evaluates whether or not a particular font contains a specified character.

Prototype

```
char GUI_IsInFont(const GUI_FONT* pFont, U16 c);
```

Parameter	Meaning
<code>pFont</code>	Pointer to the font.
<code>c</code>	Character to be searched for.

Add. information

If the pointer `pFont` is set to 0, the currently selected font is used.

Example

Evaluates whether the font `GUI_FontD32` contains an "X":

```
if (GUI_IsInFont(&GUI_FontD32, 'X') == 0) {
    GUI_DispString("GUI_FontD32 does not contains 'X'");
}
```

8.9 Character sets

8.9.1 ASCII

emWin supports the full set of ASCII characters. These are the following 96 characters from 32 to 127:

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2x		!		"#	\$	%	&		'()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x		`a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Unfortunately, as ASCII stands for American Standard Code for Information Interchange, it is designed for American needs. It does not include any of the special characters used in European languages, such as Ä, Ö, Ü, á, à, and others. There is no single standard for these "European extensions" of the ASCII set of characters; several different ones exist. The one used on the Internet and by most Windows programs is ISO 8859-1, a superset of the ASCII set of characters.

8.9.2 ISO 8859-1 Western Latin character set

emWin supports the ISO 8859-1, which defines characters as listed below:

Code	Description	Char
160	non-breaking space	
161	inverted exclamation	¡
162	cent sign	¢
163	pound sterling	£
164	general currency sign	₣
165	yen sign	¥
166	broken vertical bar	¦
167	section sign	§
168	umlaut (dieresis)	¨
169	copyright	©
170	feminine ordinal	ª
171	left angle quote, guillemotleft	«

Code	Description	Char
172	not sign	¬
173	soft hyphen	­
174	registered trademark	®
175	macron accent	—
176	degree sign	°
177	plus or minus	±
178	superscript two	²
179	superscript three	³
180	acute accent	´
181	micro sign	μ
182	paragraph sign	¶
183	middle dot	·
184	cedilla	¸
185	superscript one	¹
186	masculine ordinal	º
187	right angle quote, guillemot right	»
188	fraction one-fourth	¼
189	fraction one-half	½
190	fraction three-fourth	¾
191	inverted question mark	¿
192	capital A, grave accent	À
193	capital A, acute accent	Á
194	capital A, circumflex accent	Â
195	capital A, tilde	Ã
196	capital A, dieresis or umlaut mark	Ä
197	capital A, ring	Å
198	capital A, diphthong (ligature)	Æ
199	capital C, cedilla	Ç
200	capital E, grave accent	È
201	capital E, acute accent	É
202	capital E, circumflex accent	Ê
203	capital E, dieresis or umlaut mark	Ë
204	capital I, grave accent	Ì
205	capital I, acute accent	Í
206	capital I, circumflex accent	Î
207	capital I, dieresis or umlaut mark	Ï
208	Eth, Icelandic	Ð
209	N, tilde	Ñ
210	capital O, grave accent	Ò
211	capital O, acute accent	Ó
212	capital O, circumflex accent	Ô
213	capital O, tilde	Õ
214	capital O, dieresis or umlaut mark	Ö
215	multiply sign	×
216	capital O, slash	Ø
217	capital U, grave accent	Ù
218	capital U, acute accent	Ú
219	capital U, circumflex accent	Û
220	capital U, dieresis or umlaut mark	Ü
221	capital Y, acute accent	Ý
222	THORN, Icelandic	þ
223	sharp s, German (s-z ligature)	ß
224	small a, grave accent	à
225	small a, acute accent	á
226	small a, circumflex accent	â
227	small a, tilde	ã
228	small a, dieresis or umlaut mark	ä
229	small a, ring	å
230	small ae diphthong (ligature)	æ

Code	Description	Char
231	cedilla	ç
232	small e, grave accent	è
233	small e, acute accent	é
234	small e, circumflex accent	ê
235	small e, dieresis or umlaut mark	ë
236	small i, grave accent	ì
237	small i, acute accent	í
238	small i, circumflex accent	î
239	small i, dieresis or umlaut mark	ï
240	small eth, Icelandic	ð
241	small n, tilde	ñ
242	small o, grave accent	ò
243	small o, acute accent	ó
244	small o, circumflex accent	ô
245	small o, tilde	õ
246	small o, dieresis or umlaut mark	ö
247	division sign	÷
248	small o, slash	ø
249	small u, grave accent	ù
250	small u, acute accent	ú
251	small u, circumflex accent	û
252	small u, dieresis or umlaut mark	ü
253	small y, acute accent	ý
254	small thorn, Icelandic	þ
255	small y, dieresis or umlaut mark	ÿ

8.9.3 Unicode

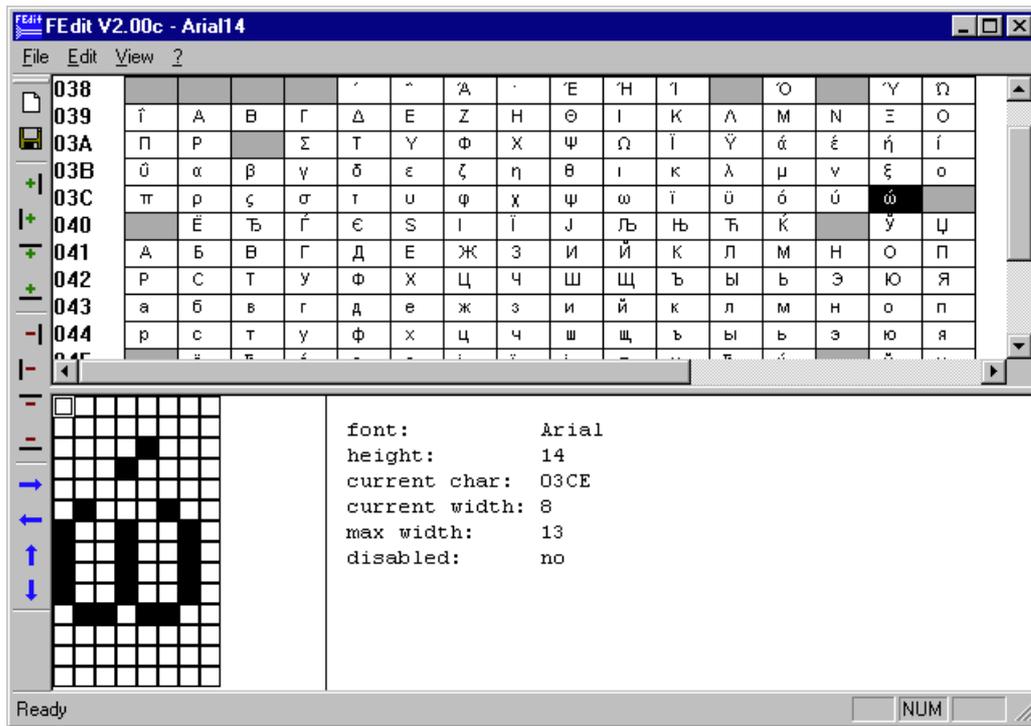
Unicode is the ultimate in character coding. It is an international standard based on ASCII and ISO 8859-1. Contrary to ASCII, UNICODE requires 16-bit characters because all characters have their own code. Currently, more than 30,000 different characters are defined. However, not all of the character images are defined in emWin. It is the responsibility of the user to define these additional characters. Please contact SEGGER Microcontroller Systeme GmbH or your distributor, as we may already have the character set that you need.

8.10 Font converter

Fonts which can be used with emWin must be defined as `GUI_FONT` structures in "C". The structures -- or rather the font data which is referenced by these structures -- can be rather large. It is very time-consuming and inefficient to generate these fonts manually. We therefore recommend using the font converter, which automatically generates "C" files from fonts.

The font converter is a simple Windows program. You need only to load an installed Windows font into the program, edit it if you want or have to, and save it as a "C" file. The "C" file may then be compiled, allowing the font to be shown on your display with emWin on demand.

The character codes 0x00 - 0x1F and 0x80 - 0x9F are disabled by default. The following is a sample screen shot of the font converter with a font loaded:



The font converter is described in a separate documentation which can be obtained by request from SEGGER Microcontroller Systeme GmbH (info@segger.com).

8.10.1 Adding fonts

Once you have created a font file and linked it to the project, declare the linked font as `extern const GUI_FONT`, as shown in the example below:

Example

```
extern const GUI_FONT GUI_FontNew;

int main(void) {
    GUI_Init();
    GUI_Clear();
    GUI_SetFont(&GUI_FontNew);
    GUI_DispString("Hello world\n");
    return 0;
}
```

8.11 Standard fonts

emWin is shipped with a selection of fonts which should cover most of your needs. The standard font package contains monospaced and proportional fonts in different sizes and styles. **Monospaced fonts** are fonts with a fixed character width, in which all characters have the same width in pixels. **Proportional fonts** are fonts in which each character has its own individual pixel-width.

This chapter provides an overview of the emWin standard fonts.

8.11.1 Font identifier naming convention

All standard fonts are named as follows. The elements of the naming convention are then explained in the table:

GUI_Font [<style>] [<width>x] <height> [x<MagX>x<MagY>] [H] [B] [_<characterset>]

Element	Meaning
GUI_Font	Standard prefix for all fonts shipped with emWin.
<style>	Specifies a non-standard font style. Example: Comic style in GUI_FontComic18B_ASCII.
<width>	Width of characters, contained only in monspaced fonts.
<height>	Height of the font in pixels.
<MagX>	Factor of magnification in X, contained only in magnified fonts.
<MagY>	Factor of magnification in Y, contained only in magnified fonts.
H	Abbreviation for "high". Only used if there is more than one font with the same height. It means that the font appears "higher" than other fonts.
B	Abbreviation for "bold". Used in bold fonts.
<characterset>	Specifies the contents of characters: ASCII: Only ASCII characters 0x20-0x7E (0x7F). 1: ASCII characters and European extensions 0xA0 - 0xFF. HK: Hiragana and Katakana. 1HK: ASCII, European extensions, Hiragana and Katakana. D: Digit fonts, character set: +-.0123456789.

Example 1

GUI_Font16_ASCII

Element	Meaning
GUI_Font	Standard font prefix.
16	Height in pixels.
ASCII	Font contains ASCII characters only.

Example 2

GUI_Font8x15B_ASCII

Element	Meaning
GUI_Font	Standard font prefix.
8	Width of characters.
x15	Height in pixels.
B	Bold font.
ASCII	Font contains ASCII characters only.

Example 3

GUI_Font8x16x1x2

Element	Meaning
GUI_Font	Standard font prefix.
8	Width of characters.
x16	Height in pixels.
x1	Magnification factor in X.
x2	Magnification factor in Y.

8.11.2 Font file naming convention

The names for the font files are similar to the names of the fonts themselves. The files are named as follows:

F[<width>]<height>[H][B][<characterset>]

Element	Meaning
F	Standard prefix for all fonts files shipped with emWin.
<width>	Width of characters, contained only in monspaced fonts.
<height>	Height of the font in pixels.
H	Abbreviation for "high". Only used if there is more than one font with the same height. It means that the font appears "higher" than other fonts.
B	Abbreviation for "bold". Used in bold fonts.
<characterset>	Specifies the contents of characters: ASCII: Only ASCII characters 0x20-0x7E (0x7F). 1: ASCII characters and European extensions 0xA0 - 0xFF. HK: Hiragana and Katakana. 1HK: ASCII, European extensions, Hiragana and Katakana. D: Digit fonts.

8.11.3 Measurement, ROM-size and character set of fonts

The following pages describe the standard fonts shipped with emWin. For each font there is a measurement diagram, an overview of all characters included and a table containing the ROM size in bytes and the font files required for use.

The following parameters are used in the measurement diagrams:

Element	Meaning
F	Size of font in Y.
B	Distance of base line from the top of the font.
C	Height of capital characters.
L	Height of lowercase characters.
U	Size of underlength used by letters such as "g", "j" or "y".

8.11.4 Proportional fonts

8.11.4.1 Overview

The following screenshot gives an overview of all available proportional fonts:

GUI_Font8_ASCII	+ABCg
GUI_Font8_1	+ABCg
GUI_Font10S_ASCII	+ABCg
GUI_Font10S_1	+ABCg
GUI_Font10_ASCII	+ABCg
GUI_Font10_1	+ABCg
GUI_Font13_ASCII	+ABCg
GUI_Font13_1	+ABCg
GUI_Font13B_ASCII	+ ABCg
GUI_Font13B_1	+ ABCg
GUI_Font13H_ASCII	+ABCg
GUI_Font13H_1	+ABCg
GUI_Font13HB_ASCII	+ ABCg
GUI_Font13HB_1	+ ABCg
GUI_Font16_ASCII	+ABCg
GUI_Font16_1	+ABCg
GUI_Font16_HK	+あぶエラ
GUI_Font16_1HK	+ABCg
GUI_Font16B_ASCII	+ ABCg
GUI_Font16B_1	+ ABCg
GUI_FontConic18B_ASCII	+ ABCg
GUI_FontConic18B_1	+ ABCg
GUI_Font20_ASCII	+ABCg
GUI_Font20_1	+ABCg
GUI_Font20B_ASCII	+ ABCg
GUI_Font20B_1	+ ABCg
GUI_Font24_ASCII	+ABCg
GUI_Font24_1	+ABCg
GUI_Font24B_ASCII	+ ABCg
GUI_Font24B_1	+ ABCg
GUI_FontConic24B_ASCII	+ ABCg
GUI_FontConic24B_1	+ ABCg
GUI_Font32_ASCII	+ABCg
GUI_Font32_1	+ABCg
GUI_Font32B_ASCII	+ ABCg
GUI_Font32B_1	+ ABCg

8.11.4.2 Measurement, ROM size and used files

The following table shows the measurement, ROM size and used files of the fonts:

Font name	Measurement	ROM size in bytes	Used files
GUI_Font8_ASCII	F: 8, B: 7, C: 7, L: 5, U: 1	1562	F08_ASCII.c
GUI_Font8_1	F: 8, B: 7, C: 7, L: 5, U: 1	1562+ 1586	F08_ASCII.c F08_1.c
GUI_Font10S_ASCII	F: 10, B: 8, C: 6, L: 4, U: 2	1760	F10S_ASCII.c
GUI_Font10S_1	F: 10, B: 8, C: 6, L: 4, U: 2	1760+ 1770	F10_ASCII.c F10_1.c
GUI_Font10_ASCII	F: 10, B: 9, C: 8, L: 6, U: 1	1800	F10_ASCII
GUI_Font10_1	F: 10, B: 9, C: 8, L: 6, U: 1	1800+ 2456	F10_ASCII.c F10_1.c
GUI_Font13_ASCII	F: 13, B: 11, C: 8, L: 6, U: 2	2076	F13_ASCII.c
GUI_Font13_1	F: 13, B: 11, C: 8, L: 6, U: 2	2076+ 2149	F13_ASCII.c F13_1.c
GUI_Font13B_ASCII	F: 13, B: 11, C: 8, L: 6, U: 2	2222	F13B_ASCII.c
GUI_Font13B_1	F: 13, B: 11, C: 8, L: 6, U: 2	2222+ 2216	F13B_ASCII.c F13B_1.c
GUI_Font13H_ASCII	F: 13, B: 11, C: 9, L: 7, U: 2	2232	F13H_ASCII.c
GUI_Font13H_1	F: 13, B: 11, C: 9, L: 7, U: 2	2232+ 2291	F13H_ASCII.c F13H_1.c
GUI_Font13HB_ASCII	F: 13, B: 11, C: 9, L: 7, U: 2	2690	F13HB_ASCII.c
GUI_Font13HB_1	F: 13, B: 11, C: 9, L: 7, U: 2	2690+ 2806	F13HB_ASCII.c F13HB_1.c
GUI_Font16_ASCII	F: 16, B: 13, C: 10, L: 7, U: 3	2714	F16_ASCII.c
GUI_Font16_1	F: 16, B: 13, C: 10, L: 7, U: 3	2714+ 3850	F16_ASCII.c F16_1.c
GUI_Font16_HK	-	6950	F16_HK.c
GUI_Font16_1HK	F: 16, B: 13, C: 10, L: 7, U: 3	120+ 6950+ 2714+ 3850	F16_1HK.c F16_HK.c F16_ASCII.c F16_1.c
GUI_Font16B_ASCII	F: 16, B: 13, C: 10, L: 7, U: 3	2690	F16B_ASCII.c
GUI_Font16B_1	F: 16, B: 13, C: 10, L: 7, U: 3	2690+ 2790	F16B_ASCII.c F16B_1.c
GUI_FontComic18B_ASCII	F: 18, B: 15, C: 12, L: 9, U: 3	3572	FComic18B_ASCII.c
GUI_FontComic18B_1	F: 18, B: 15, C: 12, L: 9, U: 3	3572+ 4334	FComic18B_ASCII.c FComic18B_1.c
GUI_Font20_ASCII	F: 20, B: 16, C: 13, L: 10, U: 4	4044	F20_ASCII.c
GUI_Font20_1	F: 20, B: 16, C: 13, L: 10, U: 4	4044+ 4244	F20_ASCII.c F20_1.c
GUI_Font20B_ASCII	F: 20, B: 16, C: 13, L: 10, U: 4	4164	F20B_ASCII.c
GUI_Font20B_1	F: 20, B: 16, C: 13, L: 10, U: 4	4164+ 4244	F20B_ASCII.c F20B_1.c
GUI_Font24_ASCII	F: 24, B: 20, C: 17, L: 13, U: 4	4786	F24_ASCII.c

Font name	Measurement	ROM size in bytes	Used files
GUI_Font24_1	F: 24, B: 20, C: 17, L: 13, U: 4	4786+ 5022	F24_ASCII.c F24_1.c
GUI_Font24B_ASCII	F: 24, B: 19, C: 15, L: 11, U: 5	4858	F24B_ASCII.c
GUI_Font24B_1	F: 24, B: 19, C: 15, L: 11, U: 5	4858+ 5022	F24B_ASCII.c F24B_1.c
GUI_FontComic24B_ASCII	F: 24, B: 20, C: 17, L: 13, U: 4	6146	FComic24B_ASCII
GUI_FontComic24B_1	F: 24, B: 20, C: 17, L: 13, U: 4	6146+ 5598	FComic24B_ASCII FComic24B_1
GUI_Font32_ASCII	F: 32, B: 26, C: 20, L: 15, U: 6	7234	F32_ASCII.c
GUI_Font32_1	F: 32, B: 26, C: 20, L: 15, U: 6	7234+ 7734	F32_ASCII.c F32_1.c
GUI_Font32B_ASCII	F: 32, B: 25, C: 20, L: 15, U: 7	7842	F32B_ASCII.c
GUI_Font32B_1	F: 32, B: 25, C: 20, L: 15, U: 7	7842+ 8118	F32B_ASCII.c F32B_1.c

8.11.4.3 Characters

The following shows all characters of all proportional standard fonts:

GUI_Font8_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz
Z[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

GUI_Font8_1

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz
Z[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹
º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç
èéêëìíîïðñóôõö÷øùúûüýþÿ

GUI_Font10S_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz
Z[\]^_`abcdefghijklmnopqrstuvwxyz{|}

GUI_Font10S_1

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz
Z[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹
º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç
èéêëìíîïðñóôõö÷øùúûüýþÿ

GUI_Font10_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuv
WXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

GUI_Font10_1

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrstuv
WXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹
º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç
èéêëìíîïðñóôõö÷øùúûüýþÿ

GUI_Font13_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrst
UVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

GUI_Font13_1

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNopqrst
UVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹
º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç
èéêëìíîïðñóôõö÷øùúûüýþÿ

GUI_Font13B_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~

GUI_Font13B_1

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊË
ÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñóôõö÷ø
ùúûüýþÿ

GUI_Font13H_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
}~
```

GUI_Font13H_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇ
ÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääåæçèéêëìíîïðóôõ
ö÷øùúûüýþÿ
```

GUI_Font13HB_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI
JKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq
rstuvwxyz{|}~
```

GUI_Font13HB_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI
JKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq
rstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»
¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞ
ßàáâãääåæçèéêëìíîïðóôõö÷øùúûüýþÿ
```

GUI_Font16_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~
```

GUI_Font16_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊ
ËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääåæçèéêëìíîïðóôõö
÷øùúûüýþÿ
```

GUI_Font16_HK

```
ああいいうええおおかがきぎくぐげげごご
さざしじすずせぜそそただちちっつづてと
どなにぬねのはばひびびふぶふへべへほほ
ぼまみむめもややゆゆよよらりるれろわわゐ
ゑをんァアイイウウエエオオカガキギクグケ
ゲゴゴサザシジスズセゼソゾタダチヂツツヅ
テデトドチニヌネノハババヒビビフブブヘベ
ペホボボマミムメモヤユエユョヨラリルレロ
ワヅヅヅヅヅヅヅヅヅヅヅヅヅ
```

GUI_Font16_1HK

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊË
ËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääåæçèéêëìíîïðñóôõö÷øùúûüýþÿ
あいうえおかがきぎくぐ
けげごさざしじすずせぜそぞただちぢっつ
づてでとどなにぬねのはばびひびびふぶぶへ
べべほぼほまみむめもややゆよよらりるれ
ろわわゐゑをんァアィイウウェエォオカガキ
ギクグケゲコゴサザシジスズセゼソゾタダチ
ヂッツツツテトドナニヌネノハババヒビピフ
ブブヘベペホボボマミムメモヤユユョヨラ
リルレロロワヰヱヰンヴカケ
```

GUI_Font16B_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
xyz{~
```

GUI_Font16B_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
xyz{~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆ
ÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääåæçèéêëìíîï
ðñóôõö÷øùúûüýþÿ
```

GUI_FontComic18B_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCD
EFGHIJKLMNOPQRSTUVWXYZ[\]^_`ab
cdefghijklmnopqrstuvwxyz{|}~€
```

GUI_FontComic18B_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCD
EFGHIJKLMNOPQRSTUVWXYZ[\]^_`ab
cdefghijklmnopqrstuvwxyz{|}~ ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇ
ÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääåæçèéêëìíîïðñóôõö÷øùúûüýþÿ
```

GUI_Font20_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@AB
 CDEFGHIJKLMNOPQRSTUVWXYZ[\]^
 _`abcdefghijklmnopqrstuvwxyz{|}~

GUI_Font20_1

!"#\$%&'()*+,-./0123456789:;<=>?@AB
 CDEFGHIJKLMNOPQRSTUVWXYZ[\]^
 _`abcdefghijklmnopqrstuvwxyz{|}~ ¡¢£
 ¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅ
 ÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßà
 áâãääåæçèéêëìíîïðñòóôõ÷øùúûüýþÿ

GUI_Font20B_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?@A
 BCDEFGHIJKLMNOPQRSTUVWXYZ[
 \]^_`abcdefghijklmnopqrstuvwxyz{|}
 ~

GUI_Font20B_1

!"#\$%&'()*+,-./0123456789:;<=>?@A
 BCDEFGHIJKLMNOPQRSTUVWXYZ[
 \]^_`abcdefghijklmnopqrstuvwxyz{|}
 ~ ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿À
 ÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙ
 ÚÛÜÝÞßàáâãääåæçèéêëìíîïðñòóôõ÷øù
 úûüýþÿ

GUI_Font24_ASCII

!"#\$%&'()*+,-./0123456789:;<=>?
 @ABCDEFGHIJKLMNQRSTU
 VWXYZ[\]^_`abcdefghijklmnopqrst
 uvwxyz{|}~

GUI_Font24_1

!"#\$%&'()*+,-./0123456789:;<=>?
 @ABCDEFGHIJKLMNQRSTU
 VWXYZ[\]^_`abcdefghijklmnopqrst
 uvwxyz{|}~ ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ
 ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎ
 ÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãääå
 æçèéêëìíîïðñòóôõö÷øùúûüýþÿ

GUI_Font24B_ASCII

!"#\$%&'()*+,-./0123456789:;<=>
 ?@ABCDEFGHIJKLMNQRST
 UVWXYZ[\]^_`abcdefghijklmnop
 qrstuvwxyz{|}~

GUI_Font24B_1

!"#\$%&'()*+,-./0123456789:;<=>
 ?@ABCDEFGHIJKLMNQRST
 UVWXYZ[\]^_`abcdefghijklmnop
 qrstuvwxyz{|}~ ¡¢£¤¥¦§¨©ª«¬®¯
 °±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈ
 ÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàá
 âãääåæçèéêëìíîïðñòóôõö÷øùúû
 ýþÿ

GUI_FontComic24B_ASCII

!"#\$%&'()*+,-./0123456789:
 ;<=>?@ABCDEFGHIJKLMN
 OPQRSTUVWXYZ[\]^_`
 abcdefghi
 jklmnopqrstuvwxyz{|}~□

GUI_FontComic24B_1

!"#\$%&'()*+,-./0123456789:
 ;<=>?@ABCDEFGHIJKLMNPO
 QRSTUVWXYZ[\]^_`abcdefghi
 jklmnopqrstuvwxyz{|}~ i¢£¥¦
 §¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾
 ¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓ
 ÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëì
 íîïðñòóôõö÷øùúûüýþÿ

GUI_Font32_ASCII

!"#\$%&'()*+,-./012345678
 9:;<=>?@ABCDEFGHIJK
 LMNOPQRSTUVWXYZ[\]
 ^_`abcdefghijklmnopqrstuv
 wxyz{|}~

GUI_Font32_1

!"#\$%&'()*+,-./012345678
 9:;<=>?@ABCDEFGHIJK
 LMNOPQRSTUVWXYZ[\]
 ^_`abcdefghijklmnopqrstuv
 wxyz{|}~ i¢£¥¦§¨©ª«¬®¯°
 ±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅ
 ÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×
 ØÙÚÛÜÝÞßàáâãäåæçèé
 êëìíîïðñòóôõö÷øùúûüýþÿ

GUI_Font32B_ASCII

!"#\$%&'()*+,-./01234567
 89:;<=>?@ABCDEFGHIJ
 KLMNOPQRSTUVWXYZ[
 \]^_`abcdefghijklmnopq
 rstuvwxyz{|}~

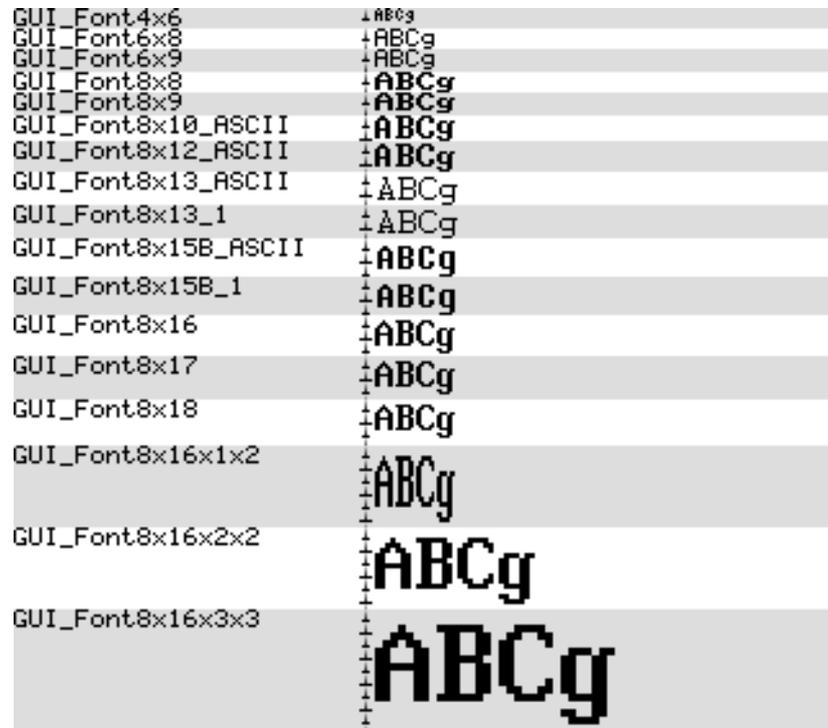
GUI_Font32B_1

!"#\$%&'()*+,-./01234567
 89:;<=>?@ABCDEFGHIJ
 KLMNOPQRSTUVWXYZ[
 \]^_`abcdefghijklmnopq
 rstuvwxyz{|}~ ¡¢£¤¥¦§¨©ª
 «¬®¯°±²³´µ¶·¸¹º»¼½¾¿À
 ÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒ
 ÓÔÕÖ×ØÙÚÛÜÝÞßàáâã
 äåæçèéêëìíîïðñòóôõö÷ø
 ùúûüýþÿ

8.11.5 Monospaced fonts

8.11.5.1 Overview

The following screenshot gives an overview of all available monospaced fonts:



8.11.5.2 Measurement, ROM size and used files

The following table shows the measurement, ROM size and used files of the fonts:

Font name	Measurement	ROM size in bytes	Used files
GUI_Font4x6	F: 6, B: 5, C: 5, L: 4, U: 1	620	F4x6.c
GUI_Font6x8	F: 8, B: 7, C: 7, L: 5, U: 1	1840	F6x8.c
GUI_Font6x9	F: 9, B: 7, C: 7, L: 5, U: 2	1840 (same ROM location as GUI_Font6x8)	F6x8.c
GUI_Font8x8	F: 8, B: 7, C: 7, L: 5, U: 1	1840	F8x8.c
GUI_Font8x9	F: 9, B: 7, C: 7, L: 5, U: 2	1840 (same ROM location as GUI_Font8x8)	F8x8.c
GUI_Font8x10_ASCII	F: 10, B: 9, C: 9, L: 7, U: 1	1770	F8x10_ASCII.c
GUI_Font8x12_ASCII	F: 12, B: 10, C: 9, L: 6, U: 2	1962	F8x12_ASCII.c
GUI_Font8x13_ASCII	F: 13, B: 11, C: 9, L: 6, U: 2	2058	F8x13_ASCII.c
GUI_Font8x13_1	F: 13, B: 11, C: 9, L: 6, U: 2	2058+ 2070	F8x13_ASCII.c F8x13_1.c
GUI_Font8x15B_ASCII	F: 15, B: 12, C: 9, L: 7, U: 3	2250	F8x15_ASCII.c
GUI_Font8x15B_1	F: 15, B: 12, C: 9, L: 7, U: 3	2250+ 2262	F8x15B_ASCII.c F8x15B_1.c
GUI_Font8x16	F: 16, B: 12, C: 10, L: 7, U: 4	3304	F8x16.c

GUI_Font8x12_ASCII

```
!"#$%&'(<)*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz<|>~^
```

GUI_Font8x13_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~|
```

GUI_Font8x13_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~| i ç £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ ·
¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ
```

GUI_Font8x15B_ASCII

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~|
```

GUI_Font8x15B_1

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~| i ç £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ ·
¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß
à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ
```

GUI_Font8x16

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~^ ↔ ↑ ↓ ↵ i ç £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ±
² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù
Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ
```

GUI_Font8x17

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
HIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
pqrstuvwxyz{|}~^ ↔ ↑ ↓ ↵ i ç £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ±
² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù
Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ
```

GUI_Font8x18

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
 IJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
 pqrstuvwxyz{|}~`↔↑↓↙ ↘ ¡¢£¤¥¦§¨ª«¬®¯°±
 ²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙ
 ÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ

GUI_Font8x16x1x2

!"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
 IJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
 pqrstuvwxyz{|}~`↔↑↓↙ ↘ ¡¢£¤¥¦§¨ª«¬®¯°±
 ²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙ
 ÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ

GUI_Font8x16x2x2

!"#\$%&'()*+,-./0123
 456789:;<=>?@ABCDEFGH
 IJKLMNOPQRSTUVWXYZ[\]
 ^_`abcdefghijklmno
 pqrstuvwxyz{|}~`↔↑↓
 ↙ ↘ ¡¢£¤¥¦§¨ª«¬®¯°±
 ²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆ
 ÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙ
 ÚÛÜÝÞßàáâãäåæçèéêëìí
 îïðñòóôõö÷øùúûüýþÿ

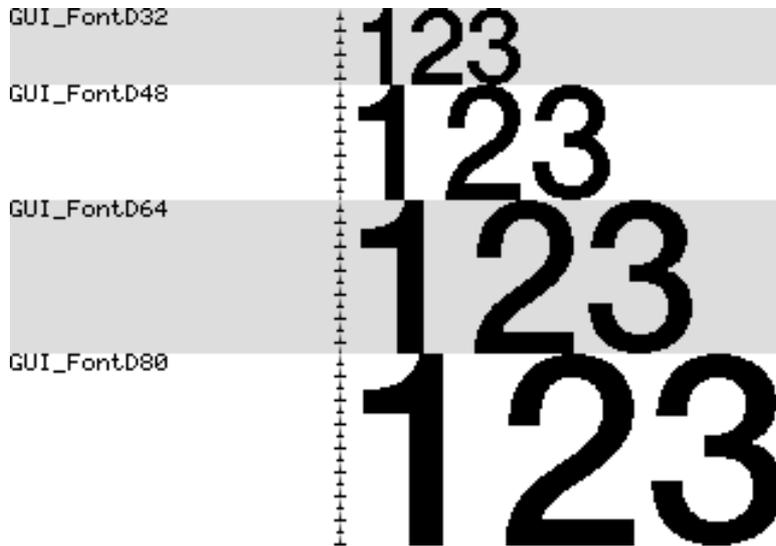
GUI_Font8x16x3x3

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

8.11.6 Digit fonts (proportional)

8.11.6.1 Overview

The following screenshot gives an overview of all available proportional digit fonts:



8.11.6.2 Measurement, ROM size and used files

The following table shows the measurement, ROM size and used files of the fonts:

Font name	Measurement	ROM size in bytes	Used files
GUI_FontD32	F: 32, C: 31	1574	FD32.c
GUI_FontD48	F: 48, C: 47	3512	FD48.c
GUI_FontD64	F: 64, C: 63	5384	FD64.c
GUI_FontD80	F: 80, C: 79	8840	FD80.c

8.11.6.3 Characters

The following shows all characters of all proportional digit fonts:

GUI_FontD32

9: + - . 0 1 2 3 4 5 6 7 8

GUI_FontD48

+ - . 0 1 2 3 4
5 6 7 8 9 :

GUI_FontD64

+ - . 0 1 2
3 4 5 6 7 8
9 :

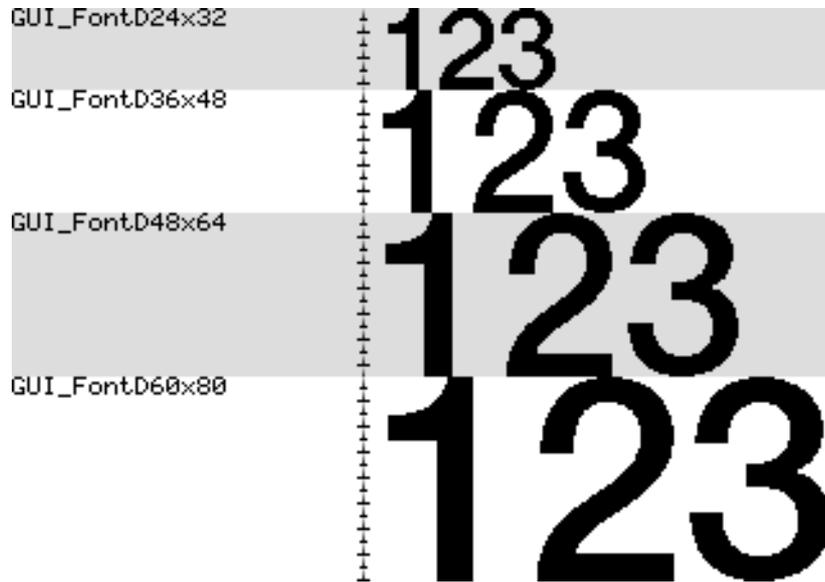
GUI_FontD80

+ - . 0 1
2 3 4 5 6
7 8 9 :

8.11.7 Digit fonts (monospaced)

8.11.7.1 Overview

The following screenshot gives an overview of all available monospaced digit fonts:



8.11.7.2 Measurement, ROM size and used files

The following table shows the measurement, ROM size and used files of the fonts:

Font name	Measurement	ROM size in bytes	Used files
GUI_FontD24x32	F: 32, C: 31	1606	FD24x32.c
GUI_FontD36x48	F: 48, C: 47	3800	FD36x48.c
GUI_FontD48x64	F: 64, C: 63	5960	FD48x60.c
GUI_FontD60x80	F: 80, C: 79	9800	FD60x80.c

8.11.7.3 Characters

The following shows all characters of all monospaced digit fonts:

GUI_FontD24x32

9: +- . 0 1 2 3 4 5 6 7 8

GUI_FontD36x48

+ - . 0 1 2 3
4 5 6 7 8 9 :

GUI_FontD48x64

+ - . 0 1
2 3 4 5 6 7
8 9 :

GUI_FontD60x80

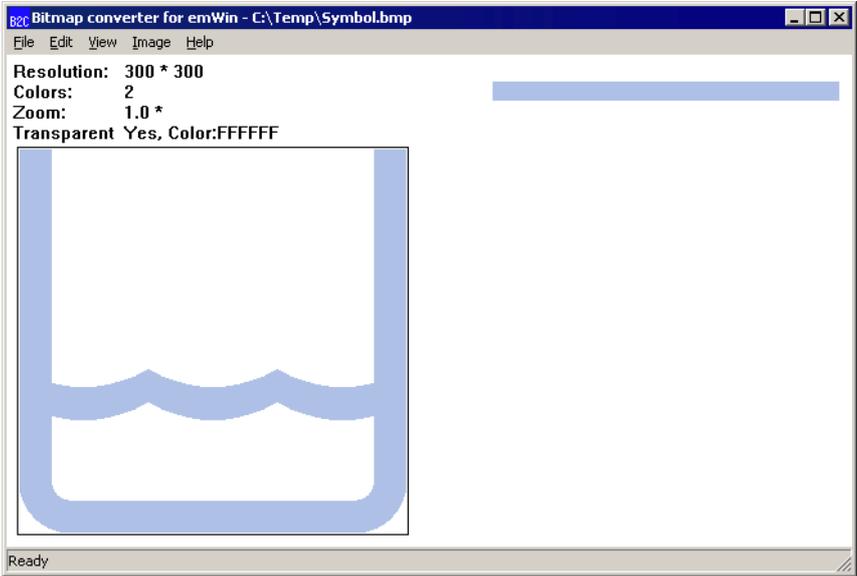
+ - . 0
1 2 3 4 5
6 7 8 9 :

Chapter 9

Bitmap Converter

The bitmap converter is a Windows program which is easy to use. Simply load a bit-
map (in the form of a .bmp or a .gif file) into the application. Convert the color for-
mat if you want or have to, and convert it into a "C" file by saving it in the
appropriate format. The "C" file may then be compiled, allowing the image to be
shown on your display with emWin.

Screenshot of the Bitmap Converter



9.1 What it does

The bitmap converter is primarily intended as a tool to convert bitmaps from a PC format to a "C" file. Bitmaps which can be used with emWin are normally defined as `GUI_BITMAP` structures in "C". The structures -- or rather the picture data which is referenced by these structures -- can be quite large. It is time-consuming and inefficient to generate these bitmaps manually. We therefore recommend using the bitmap converter, which automatically generates "C" files from bitmaps.

It also features color conversion, so that the resulting "C" code is not unnecessarily large. You would typically reduce the number of bits per pixel in order to reduce memory consumption. The bitmap converter displays the converted image.

A number of simple functions can be performed with the bitmap converter, including scaling the size, flipping the bitmap horizontally or vertically, rotating it, and inverting the bitmap indices or colors (these features can be found under the `Image` menu). Any further modifications to an image must be made in a bitmap manipulation program such as Adobe Photoshop or Corel Photopaint. It usually makes the most sense to perform any image modifications in such a program, using the bitmap converter for converting purposes only.

9.2 Loading a bitmap

9.2.1 Supported file formats

The bitmap converter basically supports 2 file formats: Windows bitmap files (*.bmp) and "Graphic Interchange Format" (*.gif):

Windows Bitmap Files

The bitmap converter supports the most common bitmap file formats. Bitmap files of the following formats can be opened by the bitmap converter:

- 1, 4 or 8 bits per pixel (bpp) with palette;
- 16, 24 or 32 bpp without palette (full-color mode, in which each color is assigned an RGB value);
- RLE4 and RLE8.

Trying to read bitmap files of other formats will cause an error message of the bitmap converter.

Graphic Interchange Format

The bitmap converter supports reading of one image per GIF file. If the file for example contains a movie consisting of more than one image, the converter reads only the first image.

Transparency and interlaced GIF images are supported by the converter.

9.2.2 Loading from a file

An image file of one of the supported formats may be opened directly in the bitmap converter by selecting `File/Open`.

9.2.3 Using the clipboard

Any other type of bitmap (i.e. .jpg, .jpeg, .png, .tif) may be opened with another program, copied to the clipboard, and pasted into the bitmap converter. This process will achieve the same effect as loading directly from a file.

9.3 Generating "C" files from bitmaps

The main function of the bitmap converter is to convert PC-formatted bitmaps into "C" files which can be used by emWin. Before doing so, however, it is often desirable to modify the color palette of an image so that the generated "C" file is not excessively large.

The bitmap may be saved as a `.bmp` or a `.gif` file (which can be reloaded and used or loaded into other bitmap manipulation programs) or as a "C" file. A "C" file will serve as an input file for your "C" compiler. It may contain a palette (device-independent bitmap, or DIB) or be saved without (device-dependent bitmap, or DDB). DIBs are recommended, as they will display correctly on any display; a DDB will only display correctly on a display which uses the same palette as the bitmap.

"C" files may be generated as "C with palette", "C without palette", "C with palette, compressed" or "C without palette, compressed". For more information on compressed files, see the section "Compressed bitmaps" as well as the example at the end of the chapter.

9.3.1 Supported bitmap formats

The following table shows the currently available output formats for "C" files:

Format	Color depth	Compression	Transparency	Palette
1 bit per pixel	1bpp	no	yes	yes
2 bits per pixel	2bpp	no	yes	yes
4 bits per pixel	4bpp	no	yes	yes
8 bits per pixel	8bpp	no	yes	yes
Compressed, RLE4	4bpp	yes	yes	yes
Compressed, RLE8	8bpp	yes	yes	yes
High color 555	15bpp	no	no	no
High color 555, red and blue swapped	15bpp	no	no	no
High color 565	16bpp	no	no	no
High color 565, red and blue swapped	16bpp	no	no	no
High color 565, compressed	16bpp	yes	no	no
High color 565, red and blue swapped, compressed	16bpp	yes	no	no
True color 888	24bpp	no	no	no
True color 888, red and blue swapped	24bpp	no	no	no

Warning: Note that emWin 8051 only supports uncompressed Bitmaps with a color depth of 1 to 8 bits per pixel.

9.3.2 Palette information

A bitmap palette is an array of 24 bit RGB color entries. Bitmaps with a color depth from 1 - 8 bpp can be saved with (device independent bitmap, DIB) or without palette information (device dependent bitmap DDB).

Device independent bitmaps (DIB)

The color information is stored in the form of an index into the color array. Before emWin draws a DIB, it converts the 24 bit RGB colors of the bitmap palette into color indices of the hardware palette. The advantage of using DIBs is that they are hardware independent and can be drawn correctly on systems with different color configurations. The disadvantages are the additional ROM requirement for the palette and the slower performance because of the color conversion.

Device dependent bitmaps (DDB)

The pixel information of a DDB is the index of the displays hardware palette. No conversion needs to be done before drawing a DDB. The advantages are less ROM requirement and a better performance. The disadvantage is that these bitmaps can not be displayed correctly on systems with other color configurations.

9.3.3 Transparency

A palette based bitmap can be converted to a transparent bitmap. Transparency means each pixel with index 0 will not produce any output. The command `Image/Transparency` can be used to select the color which should be used for transparency. After selecting the transparent color, the pixel indices of the image will be recalculated, so that the selected color is on position 0 of the bitmap palette. When saving the bitmap file as 'C' file, it will be saved with the transparency attribute.

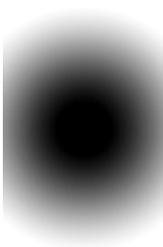
9.3.4 Alpha blending

Note: This feature is not supported by emWin 8051.

Alpha blending is a method of combining an image with the background to create the effect of semi transparency. The alpha value of a pixel determines its transparency. The color of a pixel after drawing the bitmap is a blend of the former color and the color value in the bitmap. In emWin logical colors are handled as 32 bit values. The lower 24 bits are used for the color information and the upper 8 bits are used to manage the alpha value. An alpha value of 0 means the image is opaque and a value of 0xFF means completely transparent. Because the supported file formats (BMP and GIF) do not support alpha blending, the bitmap converter initially has no information about the alpha values. It supports specifying/calculating the alpha values by two ways:

Loading the alpha values from an alpha mask bitmap

This method loads the alpha values from a separate file. Black pixels of the alpha mask file means opaque and white means transparent. The following table shows a sample:

Starting point	Alpha mask	Result
		

The command `File/Load Alpha Mask` can be used for loading an alpha mask.

Creating the alpha values from two bitmaps

This method uses the difference between the pixels of two pictures to calculate the alpha values. The first image should show the item on a black background. The second image should show the same on a white background. The following table shows a sample of how to create the alpha values using the command `File/Create Alpha`:

Starting point	Black background	White background	Result
			

The command `File/Create Alpha` can be used for creating the alpha values.

9.3.5 Selecting the best format

emWin supports various formats for the generated "C" file. It depends on several conditions which will be the 'best' format and there is no general rule to be used. Color depth, compression, palette and transparency affect the drawing performance and/or ROM requirement of the bitmap.

Color depth

In general the lower the color depth the smaller the ROM requirement of the bitmap. Each display driver has been optimized for drawing 1bpp bitmaps (text) and bitmaps with the same color depth as the display.

Compression

The supported RLE compression method has the best effect on bitmaps with many horizontal sequences of equal-colored pixels. Details later in this chapter. The performance is typically slightly slower than drawing uncompressed bitmaps.

Palette

The ROM requirement of a palette is 4 bytes for each color. So a palette of 256 colors uses 1kB. Furthermore emWin needs to convert the colors of the palette before drawing the bitmap. Advantage: Bitmaps are device independent meaning they can be displayed on any display, independent of its color depth and format.

Transparency

The ROM requirement of transparent bitmaps is the same as without transparency. The performance is with transparency slightly slower than without.

High color and true color bitmaps

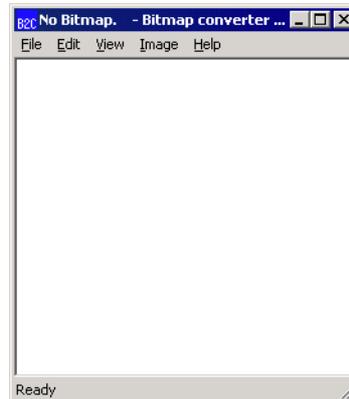
Special consideration is required for bitmaps in these formats. Generally the use of these formats only make sense on displays with a color depth of 15 bits and above. Further it is strongly recommended to save the 'C' files in the exact same format used by the hardware. Please note that using the right format will have a positive effect on the drawing performance. If a high color bitmap for example should be shown on a system with a color depth of 16bpp which has the red and blue components swapped, the best format is 'High color 565, red and blue swapped'. Already a slightly other format has the effect, that each pixel needs color conversion, whereas a bitmap in the right format can be rendered very fast without color conversion. The difference of drawing performance in this case can be factor 10 and more.

9.3.6 Saving the file

The basic procedure for using the bitmap converter is illustrated below:

Step 1: Start the application.

The bitmap converter is opened showing an empty window.

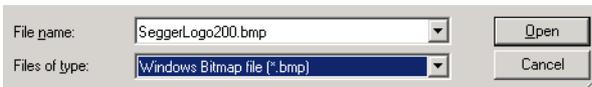


Step 2: Load a bitmap into the bitmap converter.

Choose File/Open.

Locate the document you want to open and click Open (must be a .bmp file).

In this example, the file SeggerLogo200.bmp is chosen.



The bitmap converter displays the loaded bitmap.



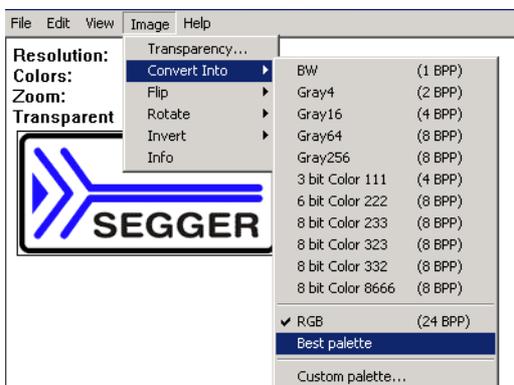
In this example, the loaded bitmap is in full-color mode. It must be converted to a palette format before a "C" file can be generated.

Step 3: Convert the image if necessary.

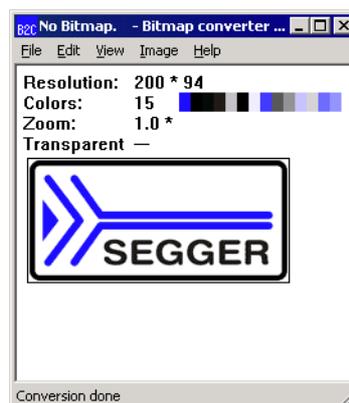
Choose Image/Convert Into.

Select the desired palette.

In this example, the option Best palette is chosen.



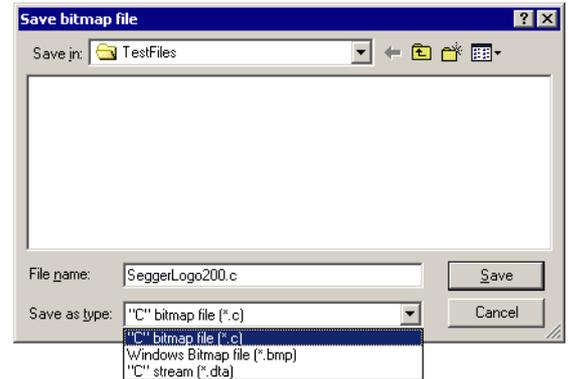
The bitmap converter displays the converted bitmap.



The image is unchanged in terms of appearance, but uses less memory since a palette of only 15 colors is used instead of the full-color mode. These 15 colors are the only ones actually required to display this particular image.

Step 4: Save the bitmap as a "C" file.

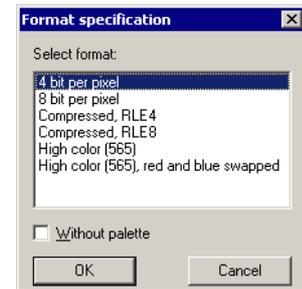
Choose `File/Save As`.
 Select a destination and a name for the "C" file.
 Select the file type. In this example, the file is saved as "C" bitmap file."
 Click `Save`.



Step 5: Specify bitmap format.

If the bitmap should be saved as 'C' file the format should now be specified. Use one of the available formats shown in the dialog. If the bitmap should be saved without palette, activate the check box "Without palette"

The bitmap converter will create a separate file in the specified destination, containing the "C" source code for the bitmap.



9.4 Color conversion

The primary reason for converting the color format of a bitmap is to reduce memory consumption. The most common way of doing this is by using the option `Best palette` as in the above example, which customizes the palette of a particular bitmap to include only the colors which are used in the image. It is especially useful with full-color bitmaps in order to make the palette as small as possible while still fully supporting the image. Once a bitmap file has been opened in the bitmap converter, simply select `Image/Convert Into/Best palette` from the menu.

For certain applications, it may be more efficient to use a fixed color palette, chosen from the menu under `Image/Convert Into`. For example, suppose a bitmap in full-color mode is to be shown on a display which supports only four grayscales. It would be a waste of memory to keep the image in the original format, since it would only appear as four grayscales on the display. The full-color bitmap can be converted into a four-grayscale, 2bpp bitmap for maximum efficiency.

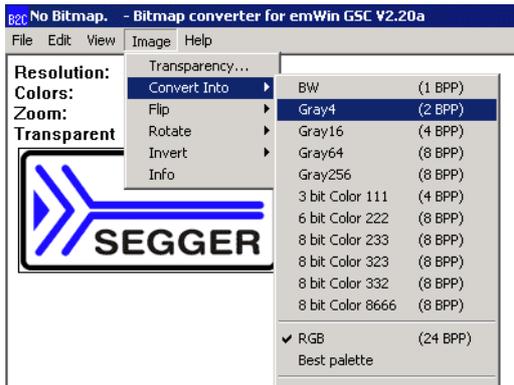
The procedure for conversion would be as follows:

The bitmap converter is opened and the same file is loaded as in steps 1 and 2 of the previous example.

The bitmap converter displays the loaded bitmap.



Choose Image/Convert Into/Gray4.



The bitmap converter displays the converted bitmap.



In this example, the image uses less memory since a palette of only 4 grayscales is used instead of the full-color mode. If the target display supports only 4 grayscales, there is no use in having a higher pixel depth as it would only waste memory.

9.5 Compressed bitmaps

The bitmap converter and emWin support run-length encoding (RLE) compression of bitmaps in the resulting source code files. The RLE compression method works most efficiently if your bitmap contains many horizontal sequences of equal-colored pixels. An efficiently compressed bitmap will save a significant amount of space. However, compression is not recommended for photographic images since they do not normally have sequences of identical pixels. It should also be noted that a compressed image may take slightly longer to display.

If you want to save a bitmap using RLE compression, you can do so by selecting one of the compressed output formats when saving as a "C" file: "C with palette, compressed" or "C without palette, compressed". There are no special functions needed for displaying compressed bitmaps; it works in the same way as displaying uncompressed bitmaps.

Compression ratios

The ratio of compression achieved will vary depending on the bitmap used. The more horizontal uniformity in the image, the better the ratio will be. A higher number of bits per pixel will also result in a higher degree of compression.

In the bitmap used in the previous examples, the total number of pixels in the image is $(200 \times 94) = 18,800$.

Since 2 pixels are stored in 1 byte, the total uncompressed size of the image is $18,800/2 = 9,400$ bytes.

The total compressed size for this particular bitmap is 3,803 bytes for 18,800 pixels (see the example at the end of the chapter).

The ratio of compression can therefore be calculated as $9,400/3,803 = 2.47$.

9.6 Using a custom palette

Converting bitmaps to a custom palette and saving them without palette information can save memory and can increase the performance of bitmap drawing operations.

More efficient memory utilisation

Per default each bitmap contains its own palette. Even the smallest bitmaps can contain a large palette with up to 256 colors. In many cases only a small fraction of the palette is used by the bitmap. If using many of these bitmaps the amount of memory used by the palettes can grow rapidly.

So it can save much ROM if converting the bitmaps used by emWin to the available hardware palette and saving them as (D)evice (D)ependent (B)itmaps without palette information.

Better bitmap drawing performance

Before emWin draws a bitmap, it needs to convert each device independent bitmap palette to the available hardware palette. This is required because the pixel indices of the bitmap file are indices into the device independent bitmap palette and not to the available hardware palette.

Converting the bitmap to a DDB means that color conversion at run time is not required and speeds up the drawing.

9.6.1 Saving a palette file

The bitmap converter can save the palette of the currently loaded bitmap into a palette file which can be used for converting other bitmaps with the command `Image/Convert Into/Custom palette`. This requires that the current file is a palette based file and not a RGB file. To save the palette the command `File/Save palette...` can be used.

9.6.2 Palette file format

Custom palette files are simple files defining the available colors for conversion. They contain the following:

- Header (8 bytes).
- NumColors (U32, 4 bytes).
- 0 (4 bytes).
- U32 Colors[NumColors] (NumColors*4 bytes, type GUI_COLOR).

Total file size is therefore: $16 + (\text{NumColors} * 4)$ bytes. A custom palette file with 8 colors would be $16 + (8 * 4) = 48$ bytes. At this point, a binary editor must be used in order to create such a file.

The maximum number of colors supported is 256; the minimum is 2.

Sample

This sample file would define a palette containing 2 colors -- red and white:

```
0000: 65 6d 57 69 6e 50 61 6c 02 00 00 00 00 00 00 00
0010: ff 00 00 00 ff ff ff 00
```

The 8 headers make up the first eight bytes of the first line. The U32 is stored lsb first (big endian) and represents the next four bytes, followed by the four 0 bytes. Colors are stored 1 byte per color, where the 4th byte is 0 as follows: RRGGBB00. The second line of code defines the two colors used in this sample.

9.6.3 Palette files for fixed palette modes

Using the custom palette feature can even make sense with the most common used fixed palette modes, not only with custom hardware palettes. For the most palette based fixed palette modes a palette file can be found in the folder `Sample\Palette`.

9.6.4 Converting a bitmap

The command `Image/Convert Into/Custom palette` should be used for converting the currently loaded bitmap to a custom palette. The bitmap converter tries to find the nearest color of the palette file for each pixel of the currently loaded bitmap.

9.7 Command line usage

It is also possible to work with the bitmap converter using the command prompt. All conversion functions available in the bitmap converter menu are available as commands, and any number of functions may be performed on a bitmap in one command line.

9.7.1 Format for commands

Commands are entered using the following format:

```
BmpCvt <filename>.bmp <-command>
```

(If more than one command is used, one space is typed between each.)

For example, a bitmap with the name `logo.bmp` is converted into `Best palette` format and saved as a "C" file named `logo.bmp` all at once by entering the following at the command prompt:

```
BmpCvt logo.bmp -convertintobestpalette -saveaslogo,1 -exit
```

Note that while the file to be loaded into the bitmap converter always includes its `.bmp` extension, no file extension is written in the `-saveas` command. An integer is used instead to specify the desired file type. The number 1 in the `-saveas` command above designates "C with palette". The `-exit` command automatically closes the program upon completion. See the table below for more information.

9.7.2 Valid command line options

The following table lists all permitted bitmap converter commands. It can also be viewed at any time by entering `BmpCvt -?` at the command prompt.

Command	Explanation
<code>-convertintobw</code>	Convert to BW.
<code>-convertintogray4</code>	Convert to Gray4.
<code>-convertintogray16</code>	Convert to Gray16.
<code>-convertintogray64</code>	Convert to Gray64.
<code>-convertintogray256</code>	Convert to Gray256.
<code>-convertinto111</code>	Convert to 111.
<code>-convertinto222</code>	Convert to 222.
<code>-convertinto233</code>	Convert to 233.
<code>-convertinto323</code>	Convert to 323.
<code>-convertinto332</code>	Convert to 332.
<code>-convertinto8666</code>	Convert to 8666.
<code>-convertintorgb</code>	Convert to RGB.
<code>-convertintobestpalette</code>	Convert to best palette.
<code>-convertintocustompalette<filename></code>	Convert to a custom palette.
<code><filename></code>	User-specified filename of desired custom palette.
<code>-exit</code>	Terminate PC program automatically.
<code>-fliph</code>	Flip image horizontally.
<code>-flipv</code>	Flip image vertically.
<code>-help</code>	Display this box.
<code>-invertindices</code>	Invert indices.
<code>-rotate90cw</code>	Rotate image by 90 degrees clockwise.
<code>-rotate90cc</code>	Rotate image by 90 degrees counter-clockwise.
<code>-rotatel180</code>	Rotate image by 180 degrees.

Command	Explanation
<code>-saveas<filename>, <type> [, <fmt> [, <noplt>]]</code>	Save file as filename.
<code><filename></code>	User-specified file name including the file extension.
<code><type></code>	Must be an integer from 1 to 6 as follows: 1: "C" with palette (.c file) 2: Windows Bitmap file (.bmp file) 3: "C" stream (.dta file) 4: GIF format (.gif file)
<code><fmt></code>	Specifies the bitmap format (only if type == 1): 1: 1 bit per pixel 2: 2 bits per pixel 4: 4 bits per pixel 5: 8 bits per pixel 6: RLE4 compression 7: RLE8 compression 8: High color 565 9: High color 565, red and blue swapped 10: High color 555 11: High color 555, red and blue swapped 12: RLE16 compression 13: RLE16 compression, red and blue swapped 17: True color 24bpp If this parameter is not given, the bitmap converter uses the following default formats in dependence of the number of colors of the bitmap: Number of colors <= 2: 1 bit per pixel Number of colors <= 4: 2 bits per pixel Number of colors <= 16: 4 bits per pixel Number of colors <= 256: 8 bits per pixel RGB: High color 565
<code><noplt></code>	Saves the bitmap with or without palette (only if type == 1) 0: Save bitmap with palette (default) 1: Save bitmap without palette
<code>-transparency<RGB-Color></code>	Sets the transparent color.
<code><RGB-Color></code>	RGB color which should be used as transparent color.
<code>-?</code>	Display this box.

9.8 Example of a converted bitmap

A typical example for the use of the bitmap converter would be the conversion of your company logo into a "C" bitmap. Take another look at the sample bitmap pictured below:



The bitmap is loaded into the bitmap converter, converted to Best palette, and saved as "C with palette". The resulting "C" source code is displayed below (some data is not shown to conserve space).

Resulting "C" code (generated by bitmap converter)

```

/*****
*           SEGGER MICROCONTROLLER SYSTEME GmbH           *
*       Solutions for real time microcontroller applications   *
*                               www.segger.com                *
*****
*
* C-file generated by
*
*       Bitmap converter for emWin V3.76.
*       Compiled Feb 10 2004, 09:26:47
*       (C) 1998 - 2004 Segger Microcontroller Systeme GmbH
*
*****
*
* Source file: test
* Dimensions:  200 * 94
* NumColors:   15
*
*****
*/

#include "stdlib.h"

#include "GUI.h"

#ifndef GUI_CONST_STORAGE
#define GUI_CONST_STORAGE const
#endif

/* Palette
The following are the entries of the palette table.
Every entry is a 32-bit value (of which 24 bits are actually used)
the lower 8 bits represent the Red component,
the middle 8 bits represent the Green component,
the highest 8 bits (of the 24 bits used) represent the Blue component
as follows:  0xBBGGRR
*/

static GUI_CONST_STORAGE GUI_COLOR ColorsLogo[] = {
    0xFFFFFFFF,0xFF0000,0x000000,0x0F0F0F
    ,0x1C1F23,0xC3C3C3,0x020202,0xFBEBEF
    ,0xFF3B3B,0x5A5B5E,0x909294,0xFFC1C1
    ,0xD0D1D1,0xFF6868,0xFF9393
};

static GUI_CONST_STORAGE GUI_LOGPALETTE PalLogo = {
    15,/* number of entries */
    0, /* No transparency */
    &ColorsLogo[0]
};

static GUI_CONST_STORAGE unsigned char acLogo[] = {
0x00, 0x00, 0xC9, 0x43, ... , 0x00,/* Not all data is shown */
0x00, 0x0A, 0x32, 0x22, ... , 0x00,/* in this example */
0x00, 0x92, 0x22, 0x22, ... , 0x00,
0x0A, 0x22, 0x22, 0x22, ... , 0xA0,
0xC6, 0x22, 0x23, 0x95, ... , 0x6C,
.
.
.
0xC6, 0x22, 0x23, 0xA5, ... , 0x6C,

```

```

0x0A, 0x22, 0x22, 0x22, ... , 0x90,
0x07, 0x92, 0x22, 0x22, ... , 0x70,
0x00, 0x7A, 0x32, 0x22, ... , 0x00,
0x00, 0x00, 0xCA, 0x44, ... , 0x00
};

GUI_CONST_STORAGE GUI_BITMAP bmLogo = {
    200, /* XSize */
    94, /* YSize */
    100, /* BytesPerLine */
    4, /* BitsPerPixel */
    acLogo, /* Pointer to picture data (indices) */
    &PalLogo /* Pointer to palette */
};

/* *** End of file *** */

```

Compressing the file

We can use the same bitmap image to create a compressed "C" file, which is done simply by loading and converting the bitmap as before, and saving it as "C with palette, compressed". The source code is displayed below (some data is not shown to conserve space).

The compressed image size can be seen towards the end of the file as 3,803 bytes for 18,800 pixels.

Resulting compressed "C" code (generated by bitmap converter)

```

/*****
*          SEGGER MICROCONTROLLER SYSTEME GmbH          *
*      Solutions for real time microcontroller applications  *
*                      www.segger.com                      *
*****
*
* C-file generated by
*
*      Bitmap converter for emWin V3.76.
*      Compiled Feb 10 2004, 09:26:47
*      (C) 1998 - 2004 Segger Microcontroller Systeme GmbH
*
*****
*
* Source file: test
* Dimensions: 200 * 94
* NumColors: 15
*
*****
*/

#include "stdlib.h"

#include "GUI.h"

#ifndef GUI_CONST_STORAGE
#define GUI_CONST_STORAGE const
#endif

/* Palette
The following are the entries of the palette table.
Every entry is a 32-bit value (of which 24 bits are actually used)
the lower 8 bits represent the Red component,
the middle 8 bits represent the Green component,
the highest 8 bits (of the 24 bits used) represent the Blue component
as follows: 0xBEGRR
*/

static GUI_CONST_STORAGE GUI_COLOR ColorsLogoCompressed[] = {
    0xFFFFFFFF, 0xFF0000, 0x000000, 0x0F0F0F
    , 0x1C1F23, 0xC3C3C3, 0x020202, 0xFBEBEF
    , 0xFF3B3B, 0x5A5B5E, 0x909294, 0xFFC1C1
    , 0xD0D1D1, 0xFF6868, 0xFF9393
};

static GUI_CONST_STORAGE GUI_LOGPALETTE PalLogoCompressed = {
    15, /* number of entries */
    0, /* No transparency */
    &ColorsLogoCompressed[0]
};

```

```

static GUI_CONST_STORAGE unsigned char acLogoCompressed[] = {
  /* RLE: 004 Pixels @ 000,000*/ 4, 0x00,
  /* ABS: 003 Pixels @ 004,000*/ 0, 3, 0xC9, 0x40,
  /* RLE: 186 Pixels @ 007,000*/ 186, 0x03,
  /* ABS: 003 Pixels @ 193,000*/ 0, 3, 0x49, 0xC0,
  /* RLE: 007 Pixels @ 196,000*/ 7, 0x00,
  .
  .
  .
  /* RLE: 006 Pixels @ 198,092*/ 6, 0x00,
  /* ABS: 004 Pixels @ 004,093*/ 0, 4, 0xCA, 0x44,
  /* RLE: 184 Pixels @ 008,093*/ 184, 0x03,
  /* ABS: 004 Pixels @ 192,093*/ 0, 4, 0x44, 0xA5,
  /* RLE: 004 Pixels @ 196,093*/ 4, 0x00,

  0}; /* 3803 for 18800 pixels */

GUI_CONST_STORAGE GUI_BITMAP bmLogoCompressed = {
  200, /* XSize */
  94, /* YSize */
  100, /* BytesPerLine */
  GUI_COMPRESS_RLE4, /* BitsPerPixel */
  acLogoCompressed, /* Pointer to picture data (indices) */
  &PalLogoCompressed /* Pointer to palette */
  ,GUI_DRAW_RLE4
};

/* *** End of file *** */

```

Chapter 10

Colors

emWin supports black/white, grayscale (monochrome with different intensities) and color displays. The same user program can be used with any display; only the LCD-configuration needs to be changed. The color management tries to find the closest match for any color that should be displayed.

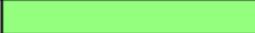
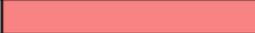
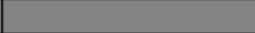
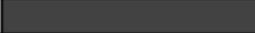
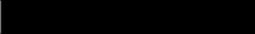
Logical colors are the colors the application deals with. A logical color is always defined as an RGB value. This is a 24-bit value containing 8 bits per color as follows: 0xBBGGRR. Therefore, white would be 0xFFFFFFFF, black would be 0x000000, bright red 0xFF.

Physical colors are the colors which can actually be displayed by the display. They are specified in the same 24-bit RGB format as logical colors. At run-time, logical colors are mapped to physical colors.

For displays with few colors (such as monochrome displays or 8/16-color LCDs), emWin converts them by using an optimized version of the "least-square deviation search". It compares the color to display (the logical color) with all the available colors that the LCD can actually show (the physical colors) and uses the one that the LCD-metric considers closest.

10.1 Predefined colors

In addition to self-defined colors, some standard colors are predefined in emWin, as shown in the following table:

GUI_BLUE		0xFF0000
GUI_GREEN		0x00FF00
GUI_RED		0x0000FF
GUI_CYAN		0xFFFF00
GUI_MAGENTA		0xFF00FF
GUI_YELLOW		0x00FFFF
GUI_LIGHTBLUE		0xFF8080
GUI_LIGHTGREEN		0x80FF80
GUI_LIGHTRED		0x8080FF
GUI_LIGHTCYAN		0xFFFF80
GUI_LIGHTMAGENTA		0xFF80FF
GUI_LIGHTYELLOW		0x80FFFF
GUI_DARKBLUE		0x800000
GUI_DARKGREEN		0x008000
GUI_DARKRED		0x000080
GUI_DARKCYAN		0x808000
GUI_DARKMAGENTA		0x800080
GUI_DARKYELLOW		0x008080
GUI_WHITE		0xFFFFFFFF
GUI_LIGHTGRAY		0xD3D3D3
GUI_GRAY		0x808080
GUI_DARKGRAY		0x404040
GUI_BLACK		0x000000
GUI_BROWN		0x2A2AA5

Example

```
/* Set background color to magenta */
GUI_SetBkColor(GUI_MAGENTA);
GUI_Clear();
```

10.2 The color bar test routine

The color bar sample program is used to show 13 color bars as follows:

Black -> Red, White -> Red, Black -> Green, White -> Green, Black -> Blue, White -> Blue, Black -> White, Black -> Yellow, White -> Yellow, Black -> Cyan, White -> Cyan, Black -> Magenta and White -> Magenta.

This little routine may be used on all displays in any color format. Of course, the results vary depending on the colors that can be displayed; the routine requires a display size of 320*240 in order to show all colors. The routine is used to demonstrate the effect of the different color settings for displays. It may also be used by a test program to verify the functionality of the display, to check available colors and grayscales, as well as to correct color conversion. The screen shots are taken from the windows simulation and will look exactly like the actual output on your display if your settings and hardware are working properly. The routine is available as `COLOR_ShowColorBar.c` in the samples shipped with emWin.

10.3 Fixed palette modes

The following table lists the available fixed palette color modes and the necessary #defines which need to be made in the file `LCDConf.h` in order to select them. Detailed descriptions follow.

LCD_FIXEDPALETTE (Color Mode)	No. available colors	LCD_SWAP_RB	Mask
1	2 (black and white)	x	0x01
2	4 (grayscale)	x	0x03
4	16 (grayscale)	x	0x0F
5	32 (grayscale)	x	0x1F
111	8	0	BGR
111	8	1	RGB
222	64	0	BBGGRR
222	64	1	RRGGBB
233	256	0	BBGGRRRR
233	256	1	RRGGBBBB
323	256	0	BBBGRRRR
323	256	1	RRRGBBBB
332	256	0	BBBGRRRR
332	256	1	RRRGBBBB
44412	4096	0	0000BBBBGGGGRRRR
44412	4096	1	0000RRRRGGGGBBBB
444121	4096	0	BBBBGGGGRRRR0000
44416	4096	0	0BBBB0GGGG0RRRR0
44416	4096	1	0RRRR0GGGG0BBBB0
555	32768	0	0BBBBBGGGGRRRRR
555	32768	1	0RRRRRGGGGBBBBB
556	65536	0	BBBBBGGGGRRRRR
556	65536	1	RRRRRGGGGBBBBB
565	65536	0	BBBBBGGGGRRRRR
565	65536	1	RRRRRGGGGBBBBB
655	65536	0	BBBBBGGGGRRRRR
655	65536	1	RRRRRGGGGBBBBB
666	262144	0	BBBBBGGGGRRRRR
822216	256	x	0xFF

LCD_FIXEDPALETTE (Color Mode)	No. available colors	LCD_SWAP_RB	Mask
84444	240	x	0xFF
8666	232	x	0xFF
86661	233 (232 + transparency)	x	0xFF
888	16777216	0	BBBBBBBBGGGGGGRRRRRR
888	16777216	1	RRRRRRGGGGGGBBBBBB
8888	16777216 + 8 bit alpha blending	0	AAAAAABBBBBBBGGGGGGRRRRRR
8888	16777216 + 8 bit alpha blending	1	AAAAAARRRRRRGGGGGGBBBBBB
-1	x	x	x

10.4 Default fixed palette modes

If no fixed palette mode has been defined in the file `LCDCnf.h`, emWin uses a default value depending on the used color depth. The following table shows the default fixed palette modes depending on the value of `LCD_BITSPERPIXEL`:

LCD_BITSPERPIXEL	Default fixed palette mode
1	1
2	2
4	4
5	5
8	8666
12	44412
15	555
16	565
24	888
32	8888

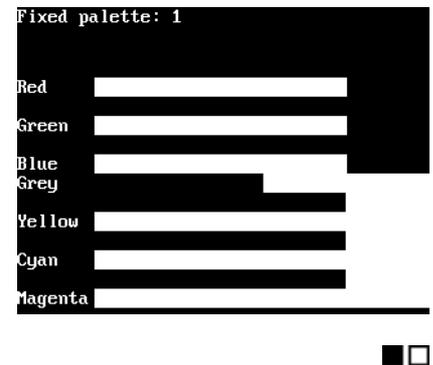
10.5 Detailed fixed palette mode description

The following gives a detailed description of the available colors in each predefined fixed palette mode.

1 mode: 1 bpp (black and white)

Use of this mode is necessary for monochrome displays with 1 bit per pixel.

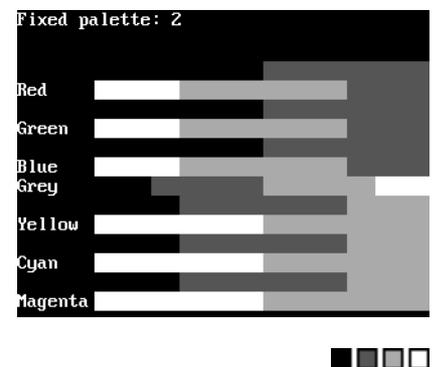
Available colors: 2:



2 mode: 2 bpp (4 grayscales)

Use of this mode is necessary for monochrome displays with 2 bits per pixel.

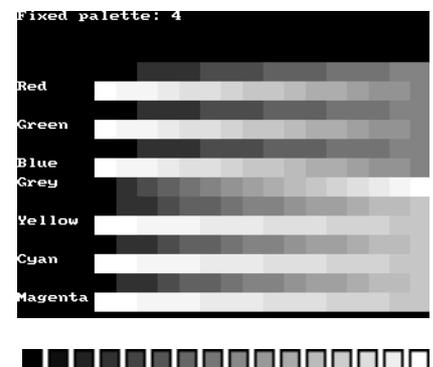
Available colors: $2 \times 2 = 4$:



4 mode: 4 bpp (16 grayscales)

Use of this mode is necessary for monochrome displays with 4 bits per pixel.

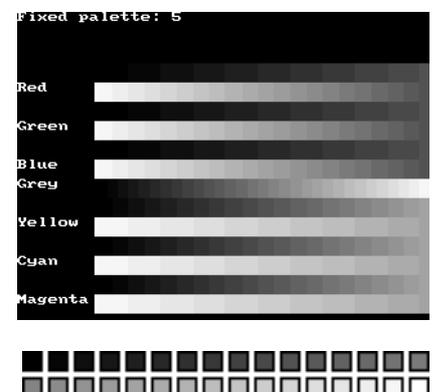
Available colors: $2 \times 2 \times 2 \times 2 = 16$:



5 mode: 5 bpp (32 grayscales)

Use of this mode is necessary for monochrome displays with 5 bits per pixel.

Available colors: $2 \times 2 \times 2 \times 2 \times 2 = 32$:



111 mode: 3 bpp (2 levels per color)

Use this mode if the basic 8 colors are enough, if your hardware supports only one bit per pixel and color or if you do not have sufficient video memory for a higher color depth.

Color mask: BGR



Available colors: $2 \times 2 \times 2 = 8$:

111 mode: 3 bpp (2 levels per color), red and blue swapped

Use this mode if the basic 8 colors are enough, if your hardware supports only one bit per pixel and color or if you do not have sufficient video memory for a higher color depth. The available colors are the same as those in 111 mode.

Color mask: RGB

Available colors: $2 \times 2 \times 2 = 8$:



222 mode: 6 bpp (4 levels per color)

This mode is a good choice if your hardware does not have a palette for every individual color. 2 bits per pixel and color are reserved; usually 1 byte is used to store one pixel.

Color mask: BBGGRR



Available colors: $4 \times 4 \times 4 = 64$:

222 mode: 6 bpp (4 levels per color), red and blue swapped

This mode is a good choice if your hardware does not have a palette for every individual color. 2 bits per pixel and color are reserved; usually 1 byte is used to store one pixel. The available colors are the same as those in 222 mode.

Color mask: RRGGBB

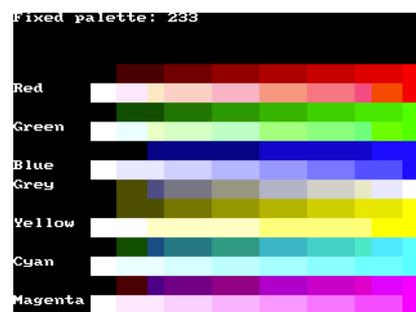
Available colors: $4 \times 4 \times 4 = 64$:



233 mode: 8 bpp

This mode supports 256 colors. 3 bits are used for the red and green components of the color and 2 bits for the blue component. As shown in the picture, the result is 8 grades for green and red and 4 grades for blue. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: BBGGRRR



Available colors: $4 \times 8 \times 8 = 256$:

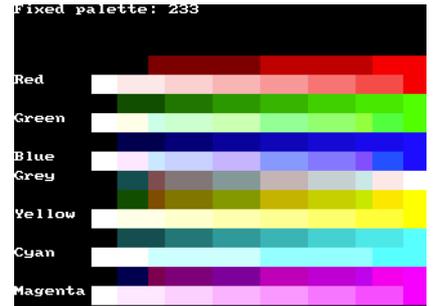


233 mode: 8 bpp, red and blue swapped

This mode supports 256 colors. 3 bits are used for the red and green components of the color and 2 bits for the blue component. The result is 8 grades for green and blue and 4 grades for red. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: RRGG BBBB

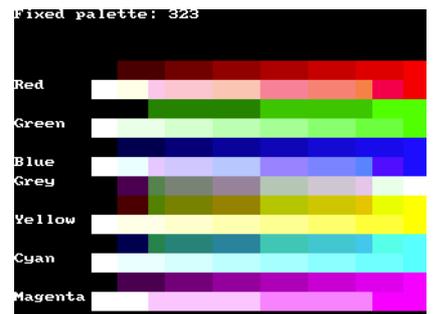
Available colors: $4 \times 8 \times 8 = 256$:



323 mode: 8 bpp

This mode supports 256 colors. 3 bits are used for the red and blue components of the color and 2 bits for the green component. As shown in the picture, the result is 8 grades for blue and red and 4 grades for green. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: BBBG RRRR



Available colors: $8 \times 4 \times 8 = 256$:



323 mode: 8 bpp, red and blue swapped

This mode supports 256 colors. 3 bits are used for the red and blue components of the color and 2 bits for the green component. The available colors are the same as those in 323 mode. The result is 8 grades for red and blue and 4 grades for green. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: RRRGGBBB

Available colors: $8 \times 4 \times 8 = 256$:

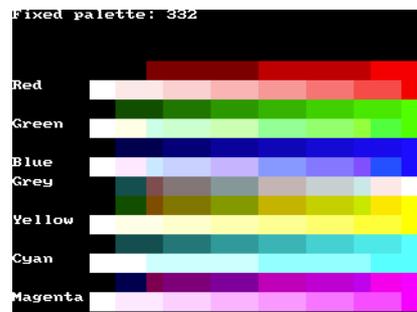


332 mode: 8 bpp

This mode supports 256 colors. 3 bits are used for the blue and green components of the color and 2 bits for the red component. As shown in the picture, the result is 8 grades for green and blue and 4 grades for red. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: BBBGGGRR

Available colors: $8 \times 8 \times 4 = 256$:

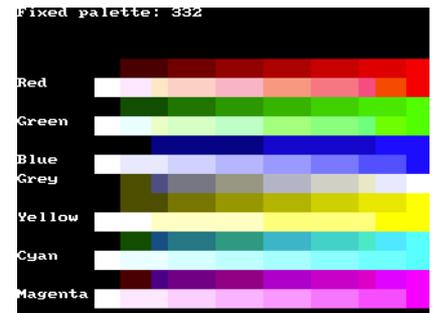


332 mode: 8 bpp, red and blue swapped

This mode supports 256 colors. 3 bits are used for the red and green components of the color and 2 bits for the blue component. The result is 8 grades for red and green and only 4 grades for blue. We discourage the use of this mode because it do not contain real shades of gray.

Color mask: RRRGGGBB

Available colors: $8 \times 8 \times 4 = 256$:

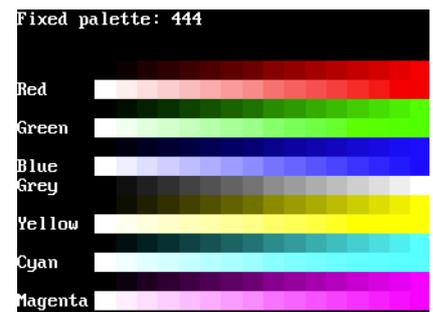


44412 mode:

The red, green and blue components are each 4 bits.

Color mask: 0000BBBBGGGRRR

Available colors: $16 \times 16 \times 16 = 4096$.



44416 mode:

The red, green and blue components are each 4 bits.

One bit between the color components is not used. The available colors are the same as those in 44412 mode.

Color mask: 0BBBB0GGGG0RRR0

Available colors: $16 \times 16 \times 16 = 4096$.

44412 mode: red and blue swapped

The red, green and blue components are each 4 bits. The available colors are the same as those in 44412 mode.

Available colors: $16 \times 16 \times 16 = 4096$.

Color mask: RRRRGGGGBBBB

44416 mode: red and blue swapped

The red, green and blue components are each 4 bits. One bit between the color components is not used. The available colors are the same as those in 44412 mode.

Color mask: 0RRR0GGGG0BBBB0

Available colors: $16 \times 16 \times 16 = 4096$.

444121 mode:

The red, green and blue components are each 4 bits. The lower 4 bits of the color mask are not used. The available colors are the same as those in 44412 mode.

Color mask: BBBBGGGGRRRR0000

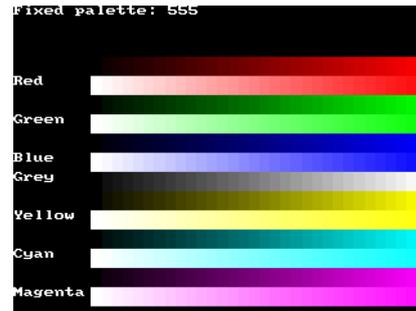
Available colors: $16 \times 16 \times 16 = 4096$.

555 mode: 15 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 15 bpp (such as SED1356 or SED13806). The red, green and blue components are each 5 bits.

Color mask: BBBBGGGGGRRRRR

Available colors: $32 \times 32 \times 32 = 32768$.



555 mode: 15 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 15 bpp. The red, green and blue components are each 5 bits. The available colors are the same as those in 555 mode.

Color mask: RRRRRGGGGGBBBBB

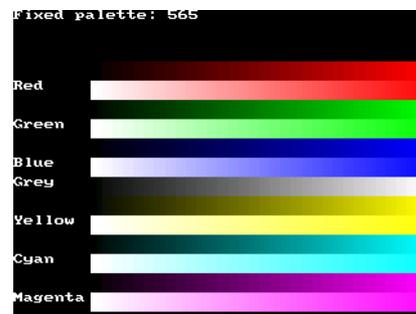
Available colors: $32 \times 32 \times 32 = 32768$.

565 mode: 16 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The red and the blue component is 5 bits and the green component is 6 bit.

Color mask: BBBBGGGGGGRRRRR

Available colors: $32 \times 64 \times 32 = 65536$.



565 mode: 16 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The available colors are the same as those in 565 mode.

Color sequence: RRRRRGGGGGBBBBB

Available colors: $32 \times 64 \times 32 = 65536$.

556 mode: 16 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The blue and the green component is 5 bit and the red component is 6 bit.

Color mask: BBBBGGGGGRRRRR

Available colors: $32 \times 32 \times 64 = 65536$.

556 mode: 16 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The red and the green component is 5 bit and the blue component is 6 bit.

Color mask: RRRRRGGGGGBBBBB

Available colors: $32 \times 32 \times 64 = 65536$.

655 mode: 16 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The red and the green component is 5 bit and the blue component is 6 bit.

Color mask: BBBBGGGGGRRRRR

Available colors: $64 \times 32 \times 32 = 65536$.

655 mode: 16 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 16 bpp. The blue and the green component is 5 bit and the red component is 6 bit.

Color mask: RRRRRRGGGGGBBBBB

Available colors: $64 \times 32 \times 32 = 65536$.

666 mode: 18 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 18 bpp. The red, green and the blue component is 6 bit.

Color mask: BBBBGGGGGRRRRR

Available colors: $64 \times 64 \times 64 = 262144$.

666 mode: 18 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color-depth of 18 bpp. The red, green and the blue component is 6 bit.

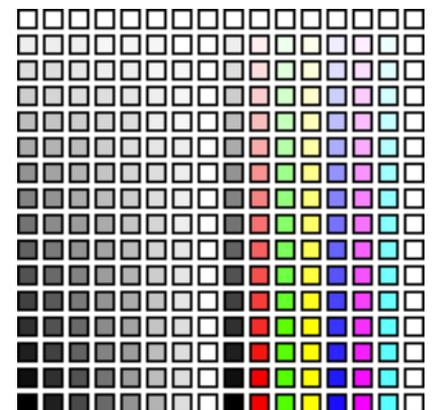
Color mask: RRRRRRGGGGGBBBBB

Available colors: $64 \times 64 \times 64 = 262144$.

822216 mode: 8 bpp, 2 levels per color + 8 grayscales + 16 levels of alpha blending

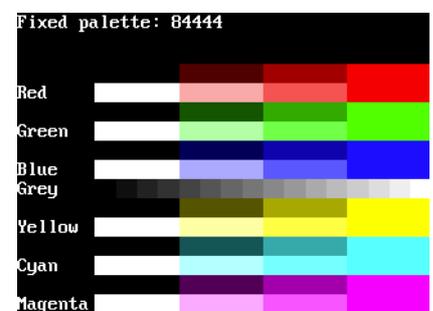
This mode can be used with a programmable color lookup table (LUT), supporting a total of 256 possible colors and alpha blending support. It supports the 8 basic colors, 8 grayscales and 16 levels of alpha blending for each color / grayscale. With other words it can be used if only a few colors are required but more levels of alpha blending.

Available colors: $(2 \times 2 \times 2 + 8) * 16 = 256$



84444 mode: 8 bpp, 4 levels per color + 16 grayscales + 4(3) levels of alpha blending

This mode can be used with a programmable color lookup table (LUT), supporting a total of 256 possible colors and alpha blending support. 4 levels of intensity are available for each color, in addition to 16 grayscales and 4 levels of alpha blending for each color / grayscale. With other words it can be used if only a few levels of alpha blending are required and different shades of colors.



Available colors: $(4 \times 4 \times 4 + 16) * 3 = 240$



8666 mode: 8bpp, 6 levels per color + 16 gray-scales

This mode is most frequently used with a programmable color lookup table (LUT), supporting a total of 256 possible colors using a palette. The screen shot gives an idea of the available colors; this mode contains the best choice for general purpose applications. Six levels of intensity are available for each color, in addition to 16 greyscales.

Available colors: $6 \times 6 \times 6 + 16 = 232$:



86661 mode: 8bpp, 6 levels per color + 16 greyscales + transparency

This mode is most frequently used with multi layer configurations and a programmable color lookup table (LUT), supporting a total of 256 possible colors using a palette. The difference between 8666 and 86661 is, that the first color indices of the 86661 mode are not used. So the color conversion routine GUI_Color2Index does never return 0 which is used for transparency.

Available colors: $6 \times 6 \times 6 + 16 = 232$.



888 mode: 24 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color depth of 24 bpp. The red, green and blue components are each 8 bits.

Color mask: BBBBGGGGRRRRRRR
Available colors: $256 \times 256 \times 256 = 16777216$.



888 mode: 24 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color depth of 24 bpp. The red, green and blue components are each 8 bits.

Color mask: RRRRRRRGGGGGGGBBBBBBB
Available colors: $256 \times 256 \times 256 = 16777216$.

8888 mode: 32 bpp

Use of this mode is necessary for a display controller that supports RGB colors with a color depth of 32 bpp, where the lower 3 bytes are used for the color components and the upper byte is used for alpha blending. The red, green, blue and alpha blending components are each 8 bits.

Color mask: AAAAAAABBBBBBBGGGGGGRRRRRRR
Available colors: $256 \times 256 \times 256 = 16777216$.

8888 mode: 32 bpp, red and blue swapped

Use of this mode is necessary for a display controller that supports RGB colors with a color depth of 32 bpp, where the lower 3 bytes are used for the color components and the upper byte is used for alpha blending. The red, green, blue and alpha blending components are each 8 bits.

Color mask: AAAAAAARRRRRRRGGGGGGBBBBBBB
Available colors: $256 \times 256 \times 256 = 16777216$.

-1 mode: Application defined fixed palette mode

If none of the fixed palette modes matches the need of color conversion this mode makes it possible to use an application defined fixed palette mode. Color conversion (RGB -> Index, Index -> RGB) will be done by calling application defined conversion routines. When setting `LCD_FIXEDPALETTE` to -1, emWin expects the following conversion functions as part of the application program:

```
unsigned LCD_Color2Index_User(LCD_COLOR Color);
LCD_COLOR LCD_Index2Color_User(int Index);
unsigned LCD_GetIndexMask_User(void);
```

The function `LCD_Color2Index_User()` is called by emWin if a RGB value should be converted into an index value for the display controller whereas the function `LCD_Index2Color_User()` is called if an index value should be converted into a RGB value.

`LCD_GetIndexMask_User()` should return a bit mask value, which has each bit set to 1 which is used by the display controller and unused bits should be set to 0. For example the index mask of the 44416 mode is `0BBBB0GGGG0RRRR0`, where 0 stands for unused bits. The bit mask for this mode is `0x7BDE`.

10.6 Custom palette modes

emWin can handle a custom hardware palette. A custom palette simply lists all the available colors in the same order as they are used by the hardware. This means that no matter what colors your LCD controller/display combination is able to display, emWin will be able to simulate them in the PC simulation and handle these colors correctly in your target system. Working with a custom palette requires a color depth ≤ 8 bpp.

In order to define a custom palette, you should do so in the configuration file `LCD-Conf.h`.

Example

The following example (part of `LCDCConf.h`) would define a custom palette with 4 colors, all of which are shades of gray:

```
#define LCD_FIXEDPALETTE 0
#define LCD_PHYSCOLORS 0xffffffff, 0xaaaaaaaa, 0x55555555, 0x00000000
```

10.7 Modifying the color lookup table at run time

The color information at each pixel is stored either in RGB mode (in which the red, green and blue components are kept for each pixel) or in color-index mode (in which a single number called the color index is stored for each pixel). Each color index corresponds to an entry in a lookup table, or color map, that defines a specific set of R, G and B values.

If your LCD controller features a color lookup table (LUT), it is properly initialized by emWin during the initialization phase (`GUI_Init()` \rightarrow `LCD_Init()` \rightarrow `LCD_InitLUT()` \rightarrow `LCD_L0_SetLUTEntry()`). However, it might be desirable (for various reasons) to modify the LUT at run time. Some possible reasons include:

- Color corrections in order to compensate for display problems (non-linearities) or gamma-correction
- Inversion of the display.
- The need to use more colors (at different times) than the hardware can show (at one time).

If you are simply modifying the LUT at run time, the color conversion routines will not be aware of this and will therefore still assume that the LUT is initialized as it was originally.

Using different colors

The default contents of the color table are defined at compile time in the configuration file `GUIConf.h` (`LCD_PHYSCOLORS`). In order to minimize RAM consumption, this data is normally declared `const` and is therefore stored in ROM. In order to be able to modify it, it needs to be stored in RAM. This can be achieved by activation of the configuration switch `LCD_LUT_IN_RAM`. If this is enabled, the API function `GUI_SetLUTColor()` becomes available and can be used to modify the contents of the color table and the LUT at the same time.

A call to `LCD_InitLUT()` will restore the original (default) settings.

10.8 Color API

The following table lists the available color-related functions in alphabetical order within their respective categories. Detailed description of the routines can be found in the sections that follow.

Routine	Explanation
Basic color functions	
<code>GUI_GetBkColor()</code>	Return the current background color.
<code>GUI_GetBkColorIndex()</code>	Return the index of the current background color.

Routine	Explanation
GUI_GetColor()	Return the current foreground color.
GUI_GetColorIndex()	Return the index of the current foreground color.
GUI_SetBkColor()	Set the current background color.
GUI_SetBkColorIndex()	Set the index of the current background color.
GUI_SetColor()	Set the current foreground color.
GUI_SetColorIndex()	Set the index of the current foreground color.
Index & color conversion	
GUI_CalcColorDist()	Returns the difference between 2 colors
GUI_CalcVisColorError()	Returns the difference to the next available color
GUI_Color2Index()	Convert color into color index.
GUI_Color2VisColor()	Returns the nearest available color
GUI_ColorIsAvailable()	Checks if given color is available
GUI_Index2Color()	Convert color index into color.
Lookup table (LUT) group	
GUI_InitLUT()	Initialize the LUT (hardware).
GUI_SetLUTColor()	Set color of a color index (both hardware and color table).
GUI_SetLUTEntry()	Write a value into the LUT (hardware).

10.9 Basic color functions

GUI_GetBkColor()

Description

Returns the current background color.

Prototype

```
GUI_COLOR GUI_GetBkColor(void);
```

Return value

The current background color.

GUI_GetBkColorIndex()

Description

Returns the index of the current background color.

Prototype

```
int GUI_GetBkColorIndex(void);
```

Return value

The current background color index.

GUI_GetColor()

Description

Returns the current foreground color.

Prototype

```
GUI_COLOR GUI_GetColor(void);
```

Return value

The current foreground color.

GUI_GetColorIndex()

Description

Returns the index of the current foreground color.

Prototype

```
int GUI_GetColorIndex(void);
```

Return value

The current foreground color index.

GUI_SetBkColor()

Description

Sets the current background color.

Prototype

```
GUI_COLOR GUI_SetBkColor(GUI_COLOR Color);
```

Parameter	Meaning
Color	Color for background, 24-bit RGB value.

Return value

The selected background color.

GUI_SetBkColorIndex()

Description

Sets the index of the current background color.

Prototype

```
int GUI_SetBkColorIndex(int Index);
```

Parameter	Meaning
Index	Index of the color to be used.

Return value

The selected background color index.

GUI_SetColor()

Description

Sets the current foreground color.

Prototype

```
void GUI_SetColor(GUI_COLOR Color);
```

Parameter	Meaning
Color	Color for foreground, 24-bit RGB value.

Return value

The selected foreground color.

GUI_SetColorIndex()

Description

Sets the index of the current foreground color.

Prototype

```
void GUI_SetColorIndex(int Index);
```

Parameter	Meaning
Index	Index of the color to be used.

Return value

The selected foreground color index.

10.10 Index & color conversion

GUI_CalcColorDist()

Calculates the distance between 2 colors. The distance will be calculated by the sum of the square value from the distances of the red, green and the blue component:

$$\text{Difference} = (\text{Red1} - \text{Red0})^2 + (\text{Green1} - \text{Green0})^2 + (\text{Blue1} - \text{Blue0})^2$$

Prototype

```
U32 GUI_CalcColorDist(GUI_COLOR Color0, GUI_COLOR Color1)
```

Parameter	Meaning
Color0	RGB value of the first color.
Color1	RGB value of the second color.

Return value

The distance as described above.

GUI_CalcVisColorError()

Calculates the distance to the next available color. For details about the calculation please take a look at GUI_CalcColorDist.

Prototype

```
U32 GUI_CalcVisColorError(GUI_COLOR color)
```

Parameter	Meaning
Color	RGB value of the color to be calculated.

Return value

The distance to the next available color.

GUI_Color2Index()

Returns the index of a specified RGB color value.

Prototype

```
int GUI_Color2Index(GUI_COLOR Color)
```

Parameter	Meaning
Color	RGB value of the color to be converted.

Return value

The color index.

GUI_Color2VisColor()

Returns the next available color of the system as an RGB color value.

Prototype

```
GUI_COLOR GUI_Color2VisColor(GUI_COLOR color)
```

Parameter	Meaning
Color	RGB value of the color.

Return value

The RGB color value of the nearest available color.

GUI_ColorIsAvailable()

Checks if the given color is available.

Prototype

```
char GUI_ColorIsAvailable(GUI_COLOR color)
```

Parameter	Meaning
Color	RGB value of the color.

Return value

1 if color is available, 0 if not.

GUI_Index2Color()

Returns the RGB color value of a specified index.

Prototype

```
int GUI_Index2Color(int Index)
```

Parameter	Meaning
Index	Index of the color. to be converted

Return value

The RGB color value.

10.11 Lookup table (LUT) group

These functions are optional and will work only if supported by the LCD controller hardware. A display controller with LUT hardware is required. Please consult the manual for the LCD controller you are using for more information on LUTs.

GUI_InitLUT()**Description**

Initializes the lookup table of the LCD controller(s).

Prototype

```
void LCD_InitLUT(void);
```

Add. information

The lookup table needs to be enabled (by the `LCD_INITCONTROLLER` macro) for this function to have any effect.

GUI_SetLUTColor()**Description**

Modifies a single entry to the color table and the LUT of the LCD controller(s).

Prototype

```
void GUI_SetLUTColor(U8 Pos, GUI_COLOR Color);
```

Parameter	Meaning
<code>Pos</code>	Position within the lookup table. Should be less than the number of colors (e.g. 0-3 for 2 bpp, 0-15 for 4 bpp, 0-255 for 8 bpp).
<code>Color</code>	24-bit RGB value.

Add. information

The closest value possible will be used for the LUT. If a color LUT is to be initialized, all 3 components are used. In monochrome modes the green component is used, but it is still recommended (for better understanding of the program code) to set all 3 colors to the same value (such as `0x555555` or `0xa0a0a0`).

The lookup table needs to be enabled (by the `LCD_INITCONTROLLER` macro) for this function to have any effect. This function is always available, but has an effect only if:

- a) The LUT is used
- b) The color table is located in RAM (`LCD_PHYSCOLORS_IN_RAM`)

GUI_SetLUTEntry()**Description**

Modifies a single entry to the LUT of the LCD controller(s).

Prototype

```
void GUI_SetLUTEntry(U8 Pos, GUI_COLOR Color);
```

Parameter	Meaning
<code>Pos</code>	Position within the lookup table. Should be less than the number of colors (e.g. 0-3 for 2 bpp, 0-15 for 4 bpp, 0-255 for 8 bpp).
<code>Color</code>	24-bit RGB value.

Add. information

The closest value possible will be used for the LUT. If a color LUT is to be initialized, all 3 components are used. In monochrome modes the green component is used, but it is still recommended (for better understanding of the program code) to set all 3 colors to the same value (such as `0x555555` or `0xa0a0a0`).

The lookup table needs to be enabled (by the `LCD_INITCONTROLLER` macro) for this function to have any effect. This function is often used to ensure that the colors actually displayed match the logical colors (linearization).

Example

```
//
// Linearize the palette of a 4-grayscale LCD
//
GUI_SetLUTEntry(0, 0x000000);
GUI_SetLUTEntry(1, 0x777777); // 555555 would be linear
GUI_SetLUTEntry(2, 0xbbbbbb); // aaaaaa would be linear
GUI_SetLUTEntry(3, 0xffffffff);
```


Chapter 11

Execution Model: Single Task / Multitask

emWin has been designed from the beginning to be compatible with different types of environments. It works in single task and in multitask applications, with a proprietary operating system or with any commercial RTOS such as embOS or uC/OS.

11.1 Supported execution models

We have to basically distinguish between 3 different execution models:

Single task system (superloop)

The entire program runs in one superloop. Normally, all software components are periodically called. Interrupts must be used for real time parts of the software since no real time kernel is used.

Multitask system: one task calling emWin

A real time kernel (RTOS) is used, but only one task calls emWin functions. From the graphic software's point of view, it is the same as being used in a single task system.

Multitask system: multiple tasks calling emWin

A real time kernel (RTOS) is used, and multiple tasks call emWin functions. This works without a problem as long as the software is made thread-safe, which is done by enabling multitask support in the configuration and adapting the kernel interface routines. For popular kernels, the kernel interface routines are readily available.

11.2 Single task system (superloop)

11.2.1 Description

The entire program runs in one superloop. Normally, all components of the software are periodically called. No real time kernel is used, so interrupts must be used for real time parts of the software. This type of system is primarily used in smaller systems or if real time behavior is not critical.

11.2.2 Superloop example (without emWin)

```
void main (void) {
    HARDWARE_Init();

    /* Init software components */
    XXX_Init();
    YYY_Init();

    /* Superloop: call all software components regularly */
    while (1) {
        /* Exec all components of the software */
        XXX_Exec();
        YYY_Exec();
    }
}
```

11.2.3 Advantages

No real time kernel is used (-> smaller ROM size, just one stack -> less RAM for stacks), no preemption/synchronization problems.

11.2.4 Disadvantages

The superloop type of program can become hard to maintain if it exceeds a certain program size. Real time behavior is poor, since one software component cannot be interrupted by any other component (only by interrupts). This means that the reaction time of one software component depends on the execution time of all other components in the system.

11.2.5 Using emWin

There are no real restrictions regarding the use of emWin. As always, `GUI_Init()` has to be called before you can use the software. From there on, any API function can be used. If the window manager's callback mechanism is used, then an emWin update function has to be called regularly. This is typically done by calling the

GUI_Exec() from within the superloop. Blocking functions such as GUI_Delay() and GUI_ExecDialog() should not be used in the loop since they would block the other software modules.

The default configuration, which does not support multitasking (#define GUI_OS 0) can be used; kernel interface routines are not required.

11.2.6 Superloop example (with emWin)

```
void main (void) {
    HARDWARE_Init();

    /* Init software components */
    XXX_Init();
    YYY_Init();
    GUI_Init();          /* Init emWin */

    /* Superloop: call all software components regularly */
    while (1) {
        /* Exec all components of the software */
        XXX_Exec();
        YYY_Exec();
        GUI_Exec();      /* Exec emWin for functionality like updating windows */
    }
}
```

11.3 Multitask system: one task calling emWin

11.3.1 Description

A real time kernel (RTOS) is used. The user program is split into different parts, which execute in different tasks and typically have different priorities. Normally the real time critical tasks (which require a certain reaction time) will have the highest priorities. **One single task** is used for the user interface, which calls emWin functions. This task usually has the lowest priority in the system or at least one of the lowest (some statistical tasks or simple idle processing may have even lower priorities).

Interrupts can, but do not have to be used for real time parts of the software.

11.3.2 Advantages

The real time behavior of the system is excellent. The real time behavior of a task is affected only by tasks running at higher priority. This means that changes to a program component running in a low priority task do not affect the real time behavior at all. If the user interface is executed from a low priority task, this means that changes to the user interface do not affect the real time behavior. This kind of system makes it easy to assign different components of the software to different members of the development team, which can work to a high degree independently from each other.

11.3.3 Disadvantages

You need to have a real time kernel (RTOS), which costs money and uses up ROM and RAM (for stacks). In addition, you will have to think about task synchronization and how to transfer information from one task to another.

11.3.4 Using emWin

If the window manager's callback mechanism is used, then an emWin update function (typically GUI_Exec(), GUI_Delay()) has to be called regularly from the task calling emWin. Since emWin is only called by one task, to emWin it is the same as being used in a single task system.

The default configuration, which does not support multitasking (#define GUI_OS 0) can be used; kernel interface routines are not required. You can use any real time kernel, commercial or proprietary.

11.4 Multitask system: multiple tasks calling emWin

11.4.1 Description

A real time kernel (RTOS) is used. The user program is split into different parts, which execute in different tasks with typically different priorities. Normally the real time critical tasks (which require a certain reaction time) will have the highest priorities. **Multiple tasks** are used for the user interface, calling emWin functions. These tasks typically have low priorities in the system, so they do not affect the real time behaviour of the system.

Interrupts can, but do not have to be used for real time parts of the software.

11.4.2 Advantages

The real time behavior of the system is excellent. The real time behavior of a task is affected only by tasks running at higher priority. This means that changes of a program component running in a low priority task do not affect the real time behavior at all. If the user interface is executed from a low priority task, this means that changes on the user interface do not affect the real time behavior. This kind of system makes it easy to assign different components of the software to different members of the development team, which can work to a high degree independently from each other.

11.4.3 Disadvantages

You have to have a real time kernel (RTOS), which costs money and uses up some ROM and RAM (for stacks). In addition, you will have to think about task synchronization and how to transfer information from one task to another.

11.4.4 Using emWin

If the window manager's callback mechanism is used, then an emWin update function (typically `GUI_Exec()`, `GUI_Delay()`) has to be called regularly from one or more tasks calling emWin.

The default configuration, which does not support multitasking (`#define GUI_OS 0`) can **NOT** be used. The configuration needs to enable multitasking support and define a maximum number of tasks from which emWin is called (excerpt from `GUIConf.h`):

```
#define GUI_OS 1           // Enable multitasking support
#define GUI_MAX_TASK 5    // Max. number of tasks that may call emWin
```

Kernel interface routines are required, and need to match the kernel being used. You can use any real time kernel, commercial or proprietary. Both the macros and the routines are discussed in the following chapter sections.

11.4.5 Recommendations

- Call the emWin update functions (i.e. `GUI_Exec()`, `GUI_Delay()`) from just one task. It will help to keep the program structure clear. If you have sufficient RAM in your system, dedicate one task (with the lowest priority) to updating emWin. This task will continuously call `GUI_Exec()` as shown in the example below and will do nothing else.
- Keep your real time tasks (which determine the behavior of your system with respect to I/O, interface, network, etc.) separate from tasks that call emWin. This will help to assure best real time performance.
- If possible, use only one task for your user interface. This helps to keep the program structure simple and simplifies debugging. (However, this is not required and may not be suitable in some systems.)

11.4.6 Example

This excerpt shows the dedicated emWin update task. It is taken from the example `MT_Multitasking`, which is included in the samples shipped with emWin:

```

/*****
*
*          GUI background processing
*
* This task does the background processing.
* The main job is to update invalid windows, but other things such as
* evaluating mouse or touch input may also be done.
*/
void GUI_Task(void) {
    while(1) {
        GUI_Exec();          /* Do the background work ... Update windows etc.) */
        GUI_X_ExecIdle();   /* Nothing left to do for the moment ... Idle processing */
    }
}

```

11.5 GUI configuration macros for multitasking support

The following table shows the configuration macros used for a multitask system with multiple tasks calling emWin:

Type	Macro	Default	Explanation
N	GUI_MAXTASK	4	Define the maximum number of tasks from which emWin is called when multitasking support is enabled (see below).
B	GUI_OS	0	Activate to enable multitasking support.
F	GUI_X_SIGNAL_EVENT()	-	Defines a function that signals an event.
F	GUI_X_WAIT_EVENT()	GUI_X_ExecIdle()	Defines a function that waits for an event.

GUI_MAXTASK

Description

Defines the maximum number of tasks from which emWin is called to access the display.

Type

Numerical value

Add. information

This function is only relevant when GUI_OS is activated.

GUI_OS

Description

Enables multitasking support by activating the module GUI_Task.

Type

Binary switch

0: inactive, multitask support disabled (default)

1: active, multitask support enabled

GUI_X_SIGNAL_EVENT

Description

Defines a function that signals an event.

Type

Function replacement

Add. information

Per default the GUI needs to periodically check for events unless a function is defined which waits and one that triggers an event. This macro defines the function which triggers an event. It makes only sense in combination with `GUI_X_WAIT_EVENT`. The advantage of using the macros `GUI_X_SIGNAL_EVENT` and `GUI_X_WAIT_EVENT` instead of polling is the reduction of CPU load of the waiting task to 0% while it waits for input. If the macro has been defined as recommended and the user gives the system any input (keyboard or pointer input device) the defined function should signal an event.

It is recommended to specify the function `GUI_X_SignalEvent()` for the job.

Sample

```
#define GUI_X_SIGNAL_EVENT() GUI_X_SignalEvent()
```

GUI_X_WAIT_EVENT**Description**

Defines a function which waits for an event.

Type

Function replacement

Add. information

Per default the GUI needs to periodically check for events unless a function is defined which waits and one that triggers an event. This macro defines the function which waits for an event. Makes only sense in combination with `GUI_X_SIGNAL_EVENT`. The advantage of using the macros `GUI_X_SIGNAL_EVENT` and `GUI_X_WAIT_EVENT` instead of polling is the reduction of CPU load of the waiting task to 0% while it waits for input. If the macro has been defined as recommended and the system waits for user input the defined function should wait for an event signaled from the function defined by the macro `GUI_X_SIGNAL_EVENT`.

It is recommended to specify the function `GUI_X_WaitEvent()` for the job.

Sample

```
#define GUI_X_WAIT_EVENT() GUI_X_WaitEvent()
```

11.6 Kernel interface routine API

An RTOS usually offers a mechanism called a resource semaphore, in which a task using a particular resource claims that resource before actually using it. The display is an example of a resource that needs to be protected with a resource semaphore. `emWin` uses the macro `GUI_USE` to call the function `GUI_Use()` before it accesses the display or before it uses a critical internal data structure. In a similar way, it calls `GUI_Unuse()` after accessing the display or using the data structure. This is done in the module `GUI_Task.c`.

`GUI_Task.c` in turn uses the GUI kernel interface routines shown in the table below. These routines are prefixed `GUI_X_` since they are high-level (hardware-dependent) functions. They must be adapted to the real time kernel used in order to make the `emWin` task (or thread) safe. Detailed descriptions of the routines follow, as well as examples of how they are adapted for different kernels.

Routine	Explanation
<code>GUI_X_InitOS()</code>	Initialize the kernel interface module (create a resource semaphore/mutex).
<code>GUI_X_GetTaskId()</code>	Return a unique, 32-bit identifier for the current task/thread.
<code>GUI_X_Lock()</code>	Lock the GUI (block resource semaphore/mutex).

Routine	Explanation
GUI_X_SignalEvent()	Signals an event.
GUI_X_Unlock()	Unlock the GUI (unblock resource semaphore/mutex).
GUI_X_WaitEvent()	Waits for an event.

GUI_X_InitOS()

Description

Creates the resource semaphore or mutex typically used by `GUI_X_Lock()` and `GUI_X_Unlock()`.

Prototype

```
void GUI_X_InitOS(void)
```

GUI_X_GetTaskID()

Description

Returns a unique ID for the current task.

Prototype

```
U32 GUI_X_GetTaskID(void);
```

Return value

ID of the current task as a 32-bit integer.

Add. information

Used with a real-time operating system.

It does not matter which value is returned, as long as it is unique for each task/thread using the emWin API and as long as the value is always the same for each particular thread.

GUI_X_Lock()

Description

Locks the GUI.

Prototype

```
void GUI_X_Lock(void);
```

Add. information

This routine is called by the GUI before it accesses the display or before using a critical internal data structure. It blocks other threads from entering the same critical section using a resource semaphore/mutex until `GUI_X_Unlock()` has been called. When using a real time operating system, you normally have to increment a counting resource semaphore.

GUI_X_SignalEvent()

Description

Signals an event.

Prototype

```
void GUI_X_SignalEvent(void);
```

Add. information

This function is optional, it is used only via the macro `GUI_X_SIGNAL_EVENT()`.

GUI_X_Unlock()

Description

Unlocks the GUI.

Prototype

```
void GUI_X_Unlock(void);
```

Add. information

This routine is called by the GUI after accessing the display or after using a critical internal data structure.

When using a real time operating system, you normally have to decrement a counting resource semaphore.

GUI_X_WaitEvent()

Description

Waits for an event.

Prototype

```
void GUI_X_WaitEvent(void);
```

Add. information

This function is optional, it is used only via the macro `GUI_X_WAIT_EVENT()`.

Examples

Kernel interface routines for embOS

The following example shows an adaption for embOS (excerpt from file `GUI_X_embOS.c` located in the folder `Sample\GUI_X`):

```
#include "RTOS.H"

static OS_TASK* _pGUITask;
static OS_RSEMA _RSEma;

void GUI_X_InitOS(void)    { OS_CreateRSEma(&_RSEma);    }
void GUI_X_Unlock(void)   { OS_Unuse(&_RSEma); }
void GUI_X_Lock(void)     { OS_Use(&_RSEma); }
U32 GUI_X_GetTaskId(void) { return (U32)OS_GetTaskID(); }

void GUI_X_WaitEvent(void) {
    _pGUITask = OS_GetpCurrentTask();
    OS_WaitEvent(1);
}

void GUI_X_SignalEvent(void) {
    if (_pGUITask) {
        OS_SignalEvent(1, _pGUITask);
    }
}
```

Kernel interface routines for uC/OS

The following example shows an adaption for uC/OS (excerpt from file `GUI_X_uCOS.c` located in the folder `Sample\GUI_X`):

```
#include "INCLUDES.H"

static OS_EVENT * pDispSem;
static OS_EVENT * pGUITask;

U32 GUI_X_GetTaskId(void) { return ((U32)(OSTCBCur->OSTCBPrio)); }
void GUI_X_Unlock(void)   { OSSemPost(pDispSem); }
```

```

void GUI_X_InitOS(void)    {
    pDispSem = OSSemCreate(1);
    pGUITask = OSSemCreate(0);
}

void GUI_X_Lock(void)     {
    INT8U err;
    OSSemPend(pDispSem, 0, &err);
}

```

Kernel interface routines for Win32

The following is an excerpt from the Win32 simulation for emWin. (When using the emWin simulation, there is no need to add these routines, as they are already in the library.)

Note: cleanup code has been omitted for clarity.

```

/*****
*
*      emWin - Multitask interface for Win32
*
*****/

    The folling section consisting of 4 routines is used to make
    emWin thread safe with WIN32
*/

static HANDLE hMutex;

void GUI_X_InitOS(void) {
    hMutex = CreateMutex(NULL, 0, "emWinSim - Mutex");
}

unsigned int GUI_X_GetTaskId(void) {
    return GetCurrentThreadId();
}

void GUI_X_Lock(void) {
    WaitForSingleObject(hMutex, INFINITE);
}

void GUI_X_Unlock(void) {
    ReleaseMutex(hMutex);
}

```


Chapter 12

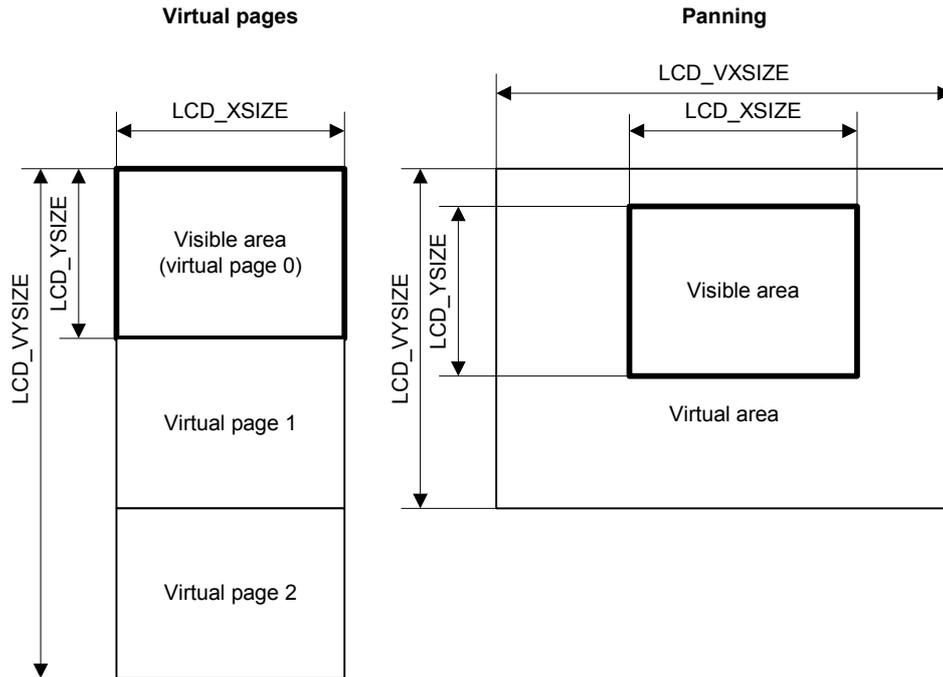
Virtual screen / Virtual pages

A virtual screen means a display area greater than the physical size of the display. It requires additional video memory and allows instantaneous switching between different screens even on slow CPUs. The following chapter shows

- the requirements for using virtual screens,
- how to configure emWin
- and how to take advantage of virtual screens.

If a virtual display area is configured, the visible part of the display can be changed by setting the origin.

12.1 Introduction



The virtual screen support of emWin can be used for panning or for switching between different video pages.

Panning

If the application uses one screen which is larger than the display, the virtual screen API functions can be used to make the desired area visible.

Virtual pages

Virtual pages are a way to use the display RAM as multiple pages. If an application for example needs 3 different screens, each screen can use its own page in the display RAM. In this case, the application can draw the second and the third page before they are used. After that the application can switch very fast between the different pages using the virtual screen API functions of emWin. The only thing the functions have to do is setting the right display start address for showing the desired screen. In this case the virtual Y-size typically is a multiple of the display size in Y.

12.2 Requirements

The virtual screen feature requires hardware with more display RAM than required for a single screen and the ability of the hardware to change the start position of the display output.

Video RAM

The used display controller should support video RAM for the virtual area. For example if the display has a resolution of 320x240 and a color depth of 16 bits per pixel and 2 screens should be supported, the required size of the video RAM can be calculated as follows:

$$\text{Size} = \text{LCD_XSIZE} * \text{LCD_YSIZE} * \text{LCD_BITSPERPIXEL} / 8 * \text{NUM_SCREENS}$$

$$\text{Size} = 320 * 240 * 16 / 8 * 2$$

$$\text{Size} = 307200 \text{ Bytes}$$

Configurable display start position

The used display controller needs a configurable display start position. This means the display driver even has a register for setting the display start address or it has a command to set the upper left display start position.

12.3 Configuration

The virtual screen support configuration should be done in the file `LCDConf.h`. The table below shows all available configuration macros:

Type	Macro	Default	Explanation
F	LCD_SET_ORG	---	Macro used to set the display start position of the upper left corner.
N	LCD_VXSIZE	LCD_XSIZE	Horizontal resolution of virtual display.
N	LCD_VYSIZE	LCD_YSIZE	Vertical resolution of virtual display.

LCD_SET_ORG

Description

This macro is used by the display driver to set the display start position of the upper left corner of the display.

Type

Function replacement.

Prototype

```
#define LCD_SET_ORG(x, y)
```

Parameter	Meaning
<code>x</code>	X position of the visible area.
<code>y</code>	Y position of the visible area.

Example

```
#define LCD_SET_ORG(x, y) SetDisplayOrigin(x, y) /* Function call for setting the
display start position */
```

LCD_VXSIZE, LCD_VYSIZE

Description

The virtual screen size is configured by the macros `LCD_VXSIZE` and `LCD_VYSIZE`. `LCD_VXSIZE` always should be $>$ `LCD_XSIZE` and `LCD_VYSIZE` should be $>$ `LCD_YSIZE`. If a virtual area is configured the clipping area of `emWin` depends on the virtual screen and not on the display size. Drawing operations outside of `LCD_XSIZE` and `LCD_YSIZE` but inside the virtual screen are performed.

Type

Numerical values.

12.3.1 Sample configuration

The following excerpt of the file `LCDConf.h` shows how to configure `emWin` for using a virtual area of 640x480 pixels on a QVGA display with 320x240 pixels:

```
#define LCD_SET_ORG(x, y) SetDisplayOrigin(x, y) /* Function call for setting the
display start position */
#define LCD_XSIZE          320 /* X-resolution of LCD */
#define LCD_YSIZE          240 /* Y-resolution of LCD */
#define LCD_VXSIZE        640 /* Virtual X-resolution */
#define LCD_VYSIZE        480 /* Virtual Y-resolution */
```

12.4 Samples

In the following a few samples are shown to make clear how to use virtual screens with emWin.

12.4.1 Basic sample

The following sample shows how to use a virtual screen of 128x192 and a display of 128x64 for instantaneous switching between 3 different screens.

Configuration

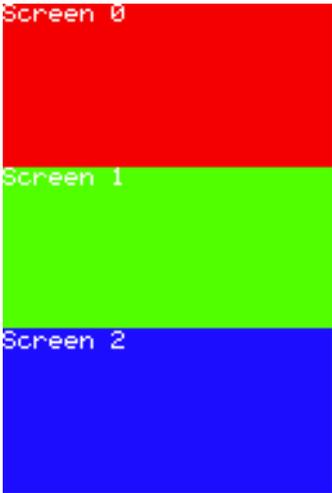
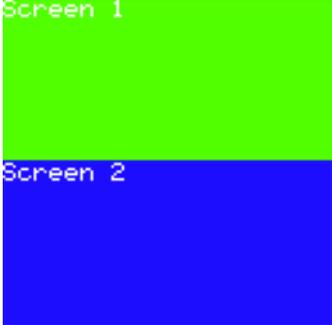
```
#define LCD_XSIZE 128
#define LCD_YSIZE 64
#define LCD_VYSIZE 192
```

Application

```
GUI_SetColor(GUI_RED);
GUI_FillRect(0, 0, 127, 63);
GUI_SetColor(GUI_GREEN);
GUI_FillRect(0, 64, 127, 127);
GUI_SetColor(GUI_BLUE);
GUI_FillRect(0, 127, 127, 191);
GUI_SetColor(GUI_WHITE);
GUI_SetTextMode(GUI_TM_TRANS);
GUI_DispStringAt("Screen 0", 0, 0);
GUI_DispStringAt("Screen 1", 0, 64);
GUI_DispStringAt("Screen 2", 0, 128);
GUI_SetOrg(0, 64); /* Set origin to screen 1 */
GUI_SetOrg(0, 128); /* Set origin to screen 2 */
```

Output

The table below shows the output of the display:

Explanation	Display output	Contents of virtual area
Before executing GUI_SetOrg(0, 240)		
After executing GUI_SetOrg(0, 240)		
After executing GUI_SetOrg(0, 480)		

12.5 Virtual screen API

The following table lists the available routines of the virtual screen support.

Routine	Explanation
GUI_GetOrg()	Returns the display start position.
GUI_SetOrg()	Sets the display start position.

GUI_GetOrg()

Description

Returns the display start position.

Prototype

```
void GUI_GetOrg(int * px, int * py);
```

Parameter	Meaning
px	Pointer to variable of type int to store the X position of the display start position.
py	Pointer to variable of type int to store the Y position of the display start position.

Add. information

The function stores the current display start position into the variables pointed by the given pointers.

GUI_SetOrg()

Description

Sets the display start position.

Prototype

```
void GUI_SetOrg(int x, int y);
```

Parameter	Meaning
x	New X position of the display start position.
y	New Y position of the display start position.

Visible screen

Page 0

always
"Main screen"

Page 1

always
"Setup" screen

Page 2

used for dif-
ferent screens

Chapter 13

Keyboard Input

emWin provides support for any kind of keyboards. Any type of keyboard driver is compatible with emWin.
The software for keyboard input is located in the subdirectory `GUI\Core` and part of the basic package.

13.1 Description

A keyboard input device uses ASCII character coding in order to be able to distinguish between characters. For example, there is only one "A" key on the keyboard, but an uppercase "A" and a lowercase "a" have different ASCII codes (0x41 and 0x61, respectively).

emWin predefined character codes

emWin also defines character codes for other "virtual" keyboard operations. These codes are listed in the table below, and defined in an identifier table in `GUI.h`. A character code in emWin can therefore be any extended ASCII character value or any of the following predefined emWin values.

Predefined virtual key code	Description
<code>GUI_KEY_BACKSPACE</code>	Backspace key.
<code>GUI_KEY_TAB</code>	Tab key.
<code>GUI_KEY_ENTER</code>	Enter/return key.
<code>GUI_KEY_LEFT</code>	Left arrow key.
<code>GUI_KEY_UP</code>	Up arrow key.
<code>GUI_KEY_RIGHT</code>	Right arrow key.
<code>GUI_KEY_DOWN</code>	Down arrow key.
<code>GUI_KEY_HOME</code>	Home key (move to beginning of current line).
<code>GUI_KEY_END</code>	End key (move to end of current line).
<code>GUI_KEY_SHIFT</code>	Shift key.
<code>GUI_KEY_CONTROL</code>	Control key.
<code>GUI_KEY_ESCAPE</code>	Escape key.
<code>GUI_KEY_INSERT</code>	Insert key.
<code>GUI_KEY_DELETE</code>	Delete key.

13.1.1 Driver layer API

The keyboard driver layer handles keyboard messaging functions. These routines notify the window manager when specific keys (or combinations of keys) have been pressed or released.

The table below lists the driver-layer keyboard routines in alphabetical order. Detailed descriptions follow.

Routine	Explanation
<code>GUI_StoreKeyMsg()</code>	Store a message in a specified key.
<code>GUI_SendKeyMsg()</code>	Send a message to a specified key.

GUI_StoreKeyMsg()

Description

Stores the message data (Key, PressedCnt) into the keyboard buffer.

Prototype

```
void GUI_StoreKeyMsg(int Key, int Pressed);
```

Parameter	Meaning
Key	May be any extended ASCII character (between 0x20 and 0xFF) or any predefined emWin character code.
Pressed	Key state (see table below).

Permitted values for parameter Pressed	
1	Pressed state.
0	Released (unpressed) state.

Add. information

This function can be used from an interrupt service routine. The input buffer contains only one character.

GUI_SendKeyMsg()

Description

Sends the keyboard data to the window with the input focus. If no window has the input focus, the function `GUI_StoreKeyMsg()` is called to store the data to the input buffer.

Prototype

```
void GUI_SendKeyMsg(int Key, int Pressed);
```

Parameter	Meaning
Key	May be any extended ASCII character (between 0x20 and 0xFF) or any predefined emWin character code.
Pressed	Key state (see <code>GUI_StoreKeyMsg()</code>).

Add. information

This function should not be called from an interrupt service routine.

13.1.2 Application layer API

The table below lists the application-layer keyboard routines in alphabetical order. Detailed descriptions follow.

Routine	Explanation
GUI_ClearKeyBuffer()	Clear the key buffer.
GUI_GetKey()	Return the contents of the key buffer.
GUI_StoreKey()	Store a key in the buffer.
GUI_WaitKey()	Wait for a key to be pressed.

GUI_ClearKeyBuffer()

Description

Clears the key buffer.

Prototype

```
void GUI_ClearKeyBuffer(void);
```

GUI_GetKey()

Description

Returns the current contents of the key buffer.

Prototype

```
int GUI_GetKey(void);
```

Return value

Codes of characters in key buffer; 0 if no keys in buffer.

GUI_StoreKey()

Description

Stores a key in the buffer.

Prototype

```
void GUI_StoreKey(int Key);
```

Parameter	Meaning
Key	May be any extended ASCII character (between 0x20 and 0xFF) or any predefined emWin character code.

Add. Information

This function is typically called by the driver and not by the application itself.

GUI_WaitKey()

Description

Waits for a key to be pressed.

Prototype

```
int GUI_WaitKey(void);
```

Add. Information

The application is "blocked", meaning it will not return until a key is pressed.

Chapter 14

Foreign Language Support

Text written in a foreign language like Arabic or Chinese contains characters, which are normally not part of the fonts shipped with emWin. This chapter explains the basics like the Unicode standard, which defines all available characters worldwide and the UTF-8 encoding scheme, which is used by emWin to decode text with Unicode characters. It also explains how to enable Arabic language support.

14.1 Unicode

The Unicode standard is a 16-bit character encoding scheme. All of the characters available worldwide are in a single 16-bit character set (which works globally). The Unicode standard is defined by the Unicode consortium.

emWin can display individual characters or strings in Unicode, although it is most common to simply use mixed strings, which can have any number of Unicode sequences within one ASCII string.

14.1.1 UTF-8 encoding

ISO/IEC 10646-1 defines a multi-octet character set called the Universal Character Set (UCS) which encompasses most of the world's writing systems. Multi-octet characters, however, are not compatible with many current applications and protocols, and this has led to the development of a few UCS transformation formats (UTF), each with different characteristics.

UTF-8 has the characteristic of preserving the full ASCII range, providing compatibility with file systems, parsers and other software that rely on ASCII values but are transparent to other values.

In emWin, UTF-8 characters are encoded using sequences of 1 to 3 octets. If the high-order bit is set to 0, the remaining 7 bits being used to encode the character value. In a sequence of n octets, $n > 1$, the initial octet has the n higher-order bits set to 1, followed by a bit set to 0. The remaining bit(s) of that octet contain bits from the value of the character to be encoded. The following octet(s) all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each to contain bits from the character to be encoded.

The following table shows the encoding ranges:

Character range	UTF-8 Octet sequence
0000 - 007F	0xxxxxxx
0080 - 07FF	110xxxxx 10xxxxxx
0800 - FFFF	1110xxxx 10xxxxxx 10xxxxxx

Encoding example

The text "Halöle" contains ASCII characters and European extensions. The following hexdump shows this text as UTF-8 encoded text:

```
48 61 6C C3 B6 6C 65
```

Programming examples

If we want to display a text containing non-ASCII characters, we can do this by manually computing the UTF-8 codes for the non-ASCII characters in the string.

However, if your compiler supports UTF-8 encoding (Sometimes called multi-byte encoding), even non-ASCII characters can be used directly in strings.

```
//
// Example using ASCII encoding:
//
GUI_UC_SetEncodeUTF8();          /* required only once to activate UTF-8*/
GUI_DispString("Hal\xc3\xb6le");

//
// Example using UTF-8 encoding:
//
GUI_UC_SetEncodeUTF8();          /* required only once to activate UTF-8*/
GUI_DispString("Halöle");
```

14.1.2 Unicode characters

The character output routine used by emWin (`GUI_DispChar()`) does always take an unsigned 16-bit value (U16) and has the basic ability to display a character defined by Unicode. It simply requires a font which contains the character you want to display.

14.1.3 UTF-8 strings

This is the most recommended way to display Unicode. You do not have to use special functions to do so. If UTF-8-encoding is enabled each function of emWin which handles with strings decodes the given text as UTF-8 text.

14.1.3.1 Using U2C.exe to convert UTF-8 text into "C"-code

The `Tool` subdirectory of emWin contains the tool `U2C.exe` to convert UTF-8 text to "C"-code. It reads an UTF-8 text file and creates a "C"-file with "C"-strings. The following steps show how to convert a text file into "C"-strings and how to display them with emWin:

Step 1: Creating a UTF-8 text file

Save the text to be converted in UTF-8 format. You can use `Notepad.exe` to do this. Load the text under `Notepad.exe`:

```
Japanese:
1 - エンコーディング
2 - テキスト
3 - サポート
English:
1 - encoding
2 - text
3 - support
```

Choose "File/Save As...". The file dialog should contain a combo box to set the encoding format. Choose "UTF-8" and save the text file.

Step 2: Converting the text file into a "C"-code file

Start `U2C.exe`. After starting the program you need to select the text file to be converted. After selecting the text file the name of the "C"-file should be selected. Output of `U2C.exe`:

```
"Japanese:"
"1 - \xe3\x82\xa8\xe3\x83\xb3\xe3\x82\xb3\xe3\x83\xbc
   \xe3\x83\x87\xe3\x82\xa3\xe3\x83\xb3\xe3\x82\xb0"
"2 - \xe3\x83\x86\xe3\x82\xad\xe3\x82\xb9\xe3\x83\x88"
"3 - \xe3\x82\xb5\xe3\x83\x9d\xe3\x83\xbc\xe3\x83\x88"
"English:"
"1 - encoding"
"2 - text"
"3 - support"
```

Step 3: Using the output in the application code

The following sample shows how to display the UTF-8 text with emWin:

```
#include "GUI.h"

static const char * _apStrings[] = {
    "Japanese:",
    "1 - \xe3\x82\xa8\xe3\x83\xb3\xe3\x82\xb3\xe3\x83\xbc"
    "\xe3\x83\x87\xe3\x82\xa3\xe3\x83\xb3\xe3\x82\xb0",
    "2 - \xe3\x83\x86\xe3\x82\xad\xe3\x82\xb9\xe3\x83\x88",
    "3 - \xe3\x82\xb5\xe3\x83\x9d\xe3\x83\xbc\xe3\x83\x88",
    "English:",
    "1 - encoding",
    "2 - text",
    "3 - support"
};

void MainTask(void) {
    int i;
    GUI_Init();
    GUI_SetFont(&GUI_Font16_1HK);
    GUI_UC_SetEncodeUTF8();
    for (i = 0; i < GUI_COUNTOF(_apStrings); i++) {
        GUI_DispString(_apStrings[i]);
        GUI_DispNextLine();
    }
    while(1) {
        GUI_Delay(500);
    }
}
```

14.1.4 Unicode API

The table below lists the available routines in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
UTF-8 functions	
GUI_UC_ConvertUC2UTF8()	Converts a Unicode string into UTF-8 format.
GUI_UC_ConvertUTF82UC()	Converts a UTF-8 string into Unicode format.
GUI_UC_Encode()	Encodes the given character with the current encoding.
GUI_UC_GetCharCode()	Returns the decoded character.
GUI_UC_GetCharSize()	Returns the number of bytes used to encode the given character.
GUI_UC_SetEncodeNone()	Disables encoding.
GUI_UC_SetEncodeUTF8()	Enables UTF-8 encoding.
Double byte functions	
GUI_UC_DispString()	Displays a double byte string.

14.1.4.1 UTF-8 functions

GUI_UC_ConvertUC2UTF8()

Description

Converts the given double byte Unicode string into UTF-8 format.

Prototype

```
int GUI_UC_ConvertUC2UTF8(const U16 GUI_UNI_PTR * s, int Len,
                          char * pBuffer, int BufferSize);
```

Parameter	Meaning
s	Pointer to Unicode string to be converted.
Len	Number of Unicode characters to be converted.
pBuffer	Pointer to a buffer to write in the result.
BufferSize	Buffer size in bytes.

Return value

The function returns the number of bytes written to the buffer.

Add. Information

UTF-8 encoded characters can use up to 3 bytes. To be on the safe side the recommended buffer size is: Number of Unicode characters * 3.

If the buffer is not big enough for the whole result, the function returns when the buffer is full.

GUI_UC_ConvertUTF82UC()

Description

Converts the given UTF-8 string into Unicode format.

Prototype

```
int GUI_UC_ConvertUTF82UC(const char GUI_UNI_PTR * s, int Len,
```

```
U16 * pBuffer, int BufferSize);
```

Parameter	Meaning
<code>s</code>	Pointer to UTF-8 string to be converted.
<code>Len</code>	Number of UTF-8 characters to be converted.
<code>pBuffer</code>	Pointer to a buffer to write in the result.
<code>BufferSize</code>	Buffer size in words.

Return value

The function returns the number of Unicode characters written to the buffer.

Add. Information

If the buffer is not big enough for the whole result, the function returns when the buffer is full.

GUI_UC_Encode()

Description

This function encodes a given character with the current encoding settings.

Prototype

```
int GUI_UC_Encode(char* s, U16 Char);
```

Parameter	Meaning
<code>s</code>	Pointer to a buffer to store the encoded character.
<code>Char</code>	Character to be encoded.

Return value

The number of bytes stored to the buffer.

Add. Information

The function assumes that the buffer has at least 3 bytes for the result.

GUI_UC_GetCharCode()

Description

This function decodes a character from a given text.

Prototype

```
U16 GUI_UC_GetCharCode(const char* s);
```

Parameter	Meaning
<code>s</code>	Pointer to the text to be encoded.

Return value

The encoded character.

Related topics

`GUI_UC_GetCharSize()`

GUI_UC_GetCharSize()

Description

This function returns the number of bytes used to encode the given character.

Prototype

```
int GUI_UC_GetCharSize(const char* s);
```

Parameter	Meaning
s	Pointer to the text to be encoded.

Return value

Number of bytes used to encode the given character

Add. information

This function is used to determine how much bytes a pointer has to be incremented to point to the next character. The following example shows how to use the function:

```
static void _Display2Characters(const char * pText) {
    int Size;
    U16 Character;
    Size = GUI_UC_GetCharSize(pText);          /* Size to increment pointer */
    Character = GUI_UC_GetCharCode(pText);    /* Get first character code */
    GUI_Dispatch(Character);                  /* Display first character */
    pText += Size;                            /* Increment pointer */
    Character = GUI_UC_GetCharCode(pText);    /* Get next character code */
    GUI_Dispatch(Character);                  /* Display second character */
}
```

GUI_UC_SetEncodeNone()**Description**

Disables character encoding.

Prototype

```
void GUI_UC_SetEncodeNone(void);
```

Add. information

After calling this function each byte of a text will be handled as one character. This is the default behaviour of emWin.

GUI_UC_SetEncodeUTF8()**Description**

Enables UTF-8 encoding.

Prototype

```
void GUI_UC_SetEncodeUTF8(void);
```

Add. information

After calling GUI_UC_SetEncodeUTF8 each string related routine of emWin encodes a given sting in accordance to the UTF-8 transformation.

14.1.4.2 Double byte functions**GUI_UC_DispatchString()****Description**

This function displays the given double byte string.

Prototype

```
void GUI_UC_DispString(const U16 GUI_FAR *s);
```

Parameter	Meaning
s	Pointer to double byte string.

Add. Information

If you need to display double byte strings you should use this function. Each character has to be defined by a 16 bit value.

Chapter 15

Display drivers

A display driver supports a particular family of display controllers (typically LCD controllers) and all displays which are connected to one or more of these controllers. The driver is essentially generic, meaning it can be configured by modifying the configuration file `LCDCConf.h`. The driver itself does not need to be modified. This file contains all configurable options for the driver including how the hardware is accessed and how the controller(s) are connected to the display.

This chapter provides an overview of the display drivers available for emWin. It explains the following in terms of each driver:

- Which LCD controllers can be accessed, as well as supported color depths and types of interfaces.
- Additional RAM requirements.
- Additional functions.
- How to access the hardware.
- Special configuration switches.
- Special requirements for particular LCD controllers.

15.1 Available drivers and supported display controllers

The following table lists the available drivers and which display controllers are supported by each:

Driver	Value for macro LCD_CONTROLLER	Display Controller	Supported bits/pixel
LCDLin	1300 1301 1304 1305 1352 1353 1354 1356 1374 1375 1376 1386 3200 32168	Any display controller with linear video memory in 1, 2, 4, 8 and 16 bits/pixel mode such as: Epson S1D13505 Epson S1D13700 (direct interface) Epson S1D13706 Epson S1D13715 (direct interface) Epson S1D13717 (direct interface) Epson S1D13719 (direct interface) Epson S1D13711 Epson S2D13705 Epson S2D13A05 Solomon SSD1905 Fujitsu MB86276 (Lime) Fujitsu MB86290A (Cremson) Fujitsu MB86291 (Scarlet) Fujitsu MB86292 (Orchid) Fujitsu MB86293 (Coral Q) Fujitsu MB86294 (Coral B) Fujitsu MB86295 (Coral P) Toshiba Capricorn 2 Epson S1D13A03, S1D13A04, S1D13A05 Epson S1D13513 (direct interface) Epson SED1352, S1D13502 Epson SED1353, S1D13503 Epson SED1354, S1D13504 Epson SED1356, S1D13506 Epson SED1374, S1D13704 Epson SED1375, S1D13705 Epson SED1376, S1D13706 Epson SED1386, S1D13806 Any display controller with linear video memory (ARM or MIPS CPUs such as Sharp LH754xx, LH79520, Motorola Dragonball or NEC VR4181A) which should be accessed only in 32 bit mode. Any display controller with linear video memory which should be accessed in 8, 16 or 32 bit mode in dependence of the required operation.	1, 2, 4, 8, 16, 24, 32
LCD667XX	66700 66701 66701 66701 66702 66703 66704 66705 66705 66706 66707 66708 66708 66708 66708 66709 66709 66709 66709 66710 66711 66766 66766 66766 66772 66772 66772 66772 66772 66772 66789	Sharp LR38825 Renesas R63401 Renesas R61509 OriseTech SPFD5420A Solomon SSD1289 Toshiba JBT6K71 Sharp LCY-A06003 Samsung S6D0129 Renesas R61505 MagnaChip D54E4PA7551 Himax HX8312 Ilitek ILI9320 Ilitek ILI9325 OriseTech SPFD5408 LG Electronics LGDP4531 Novatek NT39122 Sitronix ST7628 Sitronix ST7637 Renesas R61516 Novatek NT7573 Epson S1D13743 Hitachi HD66766 Samsung S6D0110A Ilitek ILI9161 Hitachi HD66772 Samsung S6D0117 Sitronix ST7712 Himax HX8301 Ilitek ILI9220 Hitachi HD66789	16

The basic package contains 2 drivers which don't support a specific LCD controller. They can be used as template for a new driver or for measurement purpose:

Driver	Value for macro LCD_CONTROLLER	LCD Controller	Supported bits/pixel
LCDTemplate	-1	Driver template. Can be used as a starting point for writing a new driver. Part of the basic package	-
LCDNull	-2	Empty driver. (Performs no output) Can be used for measurement purpose. Part of the basic package.	-

Selecting a driver

As described in Chapter 17: "Low-Level Configuration", the macro `LCD_CONTROLLER` defines the LCD controller used. A controller is specified by its appropriate value, listed in the table above.

The following sections discuss each of the available drivers individually.

15.2 CPU / Display controller interface

Different display controllers have different CPU interfaces. The most common ones are the following:

- Full bus interface
- Simple bus interface
- 4 pin SPI interface
- 3 pin SPI interface
- I2C bus interface

Below we explain these interfaces and how to configure them. Note that not all config macros are always required. For details about which macros are required please take a look to the driver documentation later in this chapter. The Chapter 17: "Low-Level Configuration", explains the macros itself.

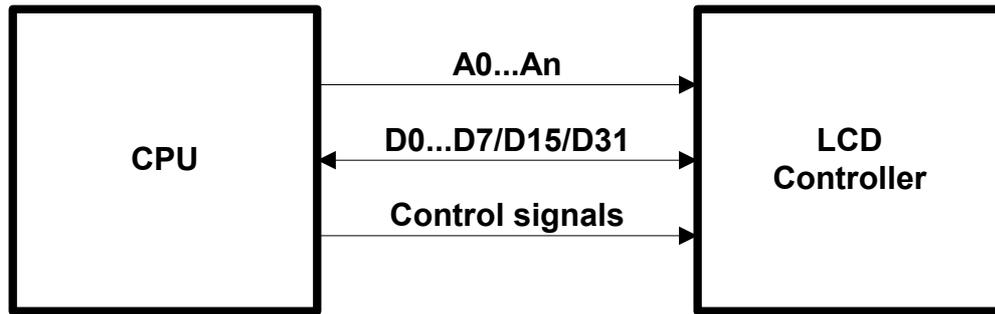
15.2.1 Full bus interface

Some LCD controllers (especially those for displays with higher resolution) require a full-address bus, which means they are connected to at least 14 address bits. In a full bus interface configuration, video memory is directly accessible by the CPU; the full-address bus is connected to the LCD controller.

The only knowledge required when configuring a full bus interface is information about the address range (which will generate a CHIP-SELECT signal for the LCD controller) and whether 8- or 16-bit accesses should be used (bus-width to the LCD controller). In other words, you need to know the following:

- Base address for video memory access
- Base address for register access
- Distance between adjacent video memory locations (usually 1/2/4-byte)
- Distance between adjacent register locations (usually 1/2/4-byte)
- Type of access (8/16/32-bit) for video memory
- Type of access (8/16/32-bit) for registers

Typical block diagram for LCD controllers with full bus interface



Macros used by a full bus interface

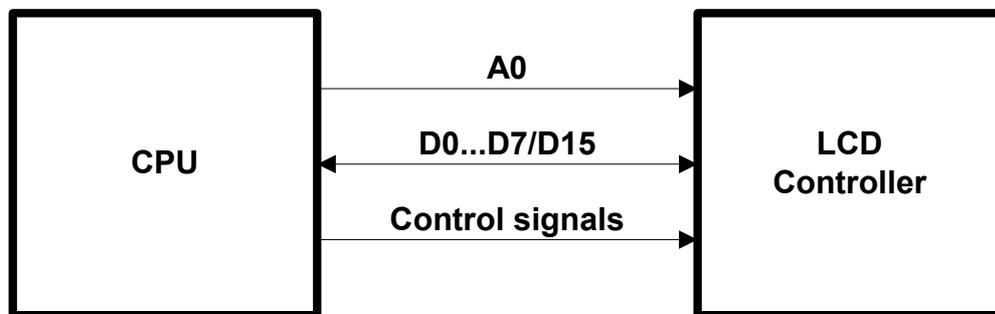
The following table shows the used hardware access macros:

Type	Macro	Explanation
F	LCD_READ_MEM	Reads the video memory of the LCD controller.
F	LCD_WRITE_MEM	Writes data to the video memory of the LCD controller.
F	LCD_READ_REG	Reads the register of the LCD controller.
F	LCD_WRITE_REG	Writes data to a specified register of the LCD controller.

15.2.2 Simple bus interface

Most LCD controllers for smaller displays (usually up to 240*128 or 320*240) use a simple bus interface to connect to the CPU. With a simple bus, only one address bit (usually A0) is connected to the LCD controller. Some of these controllers are very slow, so that the hardware designer may decide to connect it to input/output (I/O) pins instead of the address bus.

Typical block diagram for LCD controllers with simple bus interface



8 (16) data bits, one address bit and 2 or 3 control lines are used to connect the CPU and one LCD controller. Four macros inform the LCD driver how to access each controller used. If the LCD controller(s) is connected directly to the address bus of the CPU, configuration is simple and usually consists of no more than one line per macro. If the LCD controller(s) is connected to I/O pins, the bus interface must be simulated, which takes about 5-10 lines of program per macro (or a function call to a routine which simulates the bus interface). The signal **A0** is also called **C/D** (Command/Data), **D/I** (Data/Instruction) or **RS** (Register select), depending on the display controller.

Macros used by a simple bus interface

The following table shows the used hardware access macros:

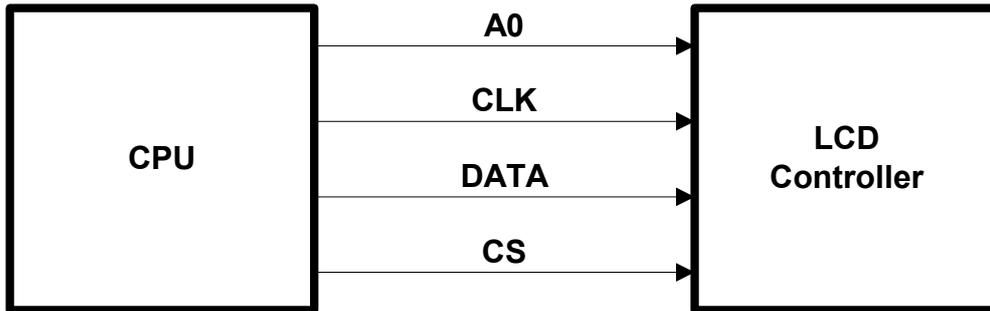
Type	Macro	Explanation
F	LCD_READ_A0	Reads a byte from LCD controller with A0 - line low.
F	LCD_READ_A1	Reads a byte from LCD controller with A0 - line high.

Type	Macro	Explanation
F	LCD_WRITE_A0	Writes a byte to LCD controller with A0 - line low.
F	LCD_WRITE_A1	Writes a byte to LCD controller with A0 - line high.
F	LCD_WRITEM_A1	Writes several bytes to the LCD controller with A0 - line high.

15.2.3 4 pin SPI interface

Using a 4 pin SPI interface is very similar to a simple bus interface. To connect a LCD display using 4 pin SPI interface the lines A0, CLK, DATA, and CS must be connected to the CPU.

Typical block diagram for LCD controllers with 4 pin SPI interface



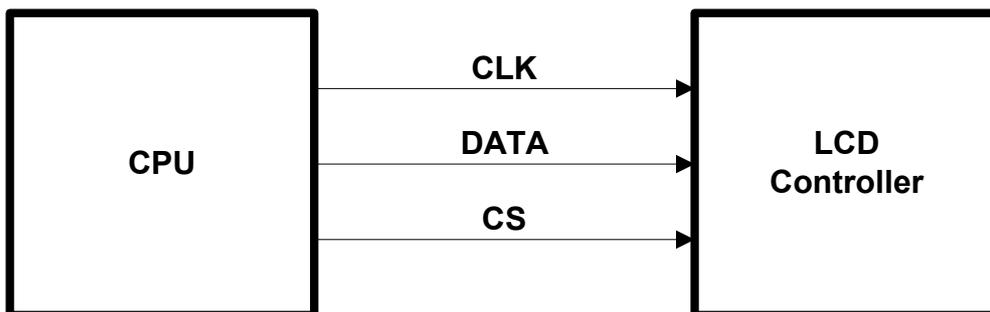
Macros used by a 4 pin SPI interface

The following table shows the used hardware access macros:

Type	Macro	Explanation
F	LCD_WRITE_A0	Writes a byte to LCD controller with A0 (C/D) - line low.
F	LCD_WRITE_A1	Writes a byte to LCD controller with A0 (C/D) - line high.
F	LCD_WRITEM_A1	Writes several bytes to the LCD controller with A0 (C/D) - line high.

15.2.4 3 pin SPI interface

Typical block diagram for LCD controllers with 3 pin SPI interface



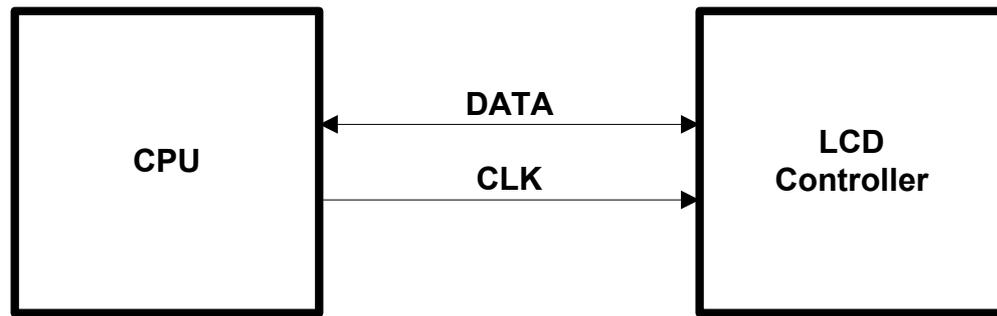
Macros used by a 3 pin SPI interface

The following table shows the used hardware access macros:

Type	Macro	Explanation
F	LCD_WRITE	Writes a byte to LCD controller.
F	LCD_WRITEM	Writes several bytes to the LCD controller.

15.2.5 I2C bus interface

Typical block diagram for LCD controllers with I2C bus interface



Macros used by a 3 pin SPI interface

The following table shows the used hardware access macros:

Type	Macro	Explanation
F	LCD_READ_A0	Reads a status byte from LCD controller.
F	LCD_READ_A1	Reads a data byte from LCD controller.
F	LCD_WRITE_A0	Writes a instruction byte to LCD controller.
F	LCD_WRITE_A1	Writes a data byte to LCD controller.
F	LCD_WRITEM_A1	Writes several data bytes to the LCD controller.

15.2.6 Non readable displays

Some display controllers with a simple bus interface do not support reading back display data. Especially displays which are connected via SPI interface often have this limitation. In this case we recommend using a display data cache. For details how to enable a display data cache please refer to the detailed driver descriptions later in this chapter.

On systems with a very small RAM it is sometimes not possible to use a display data cache. If a display is not readable and a display data cache can not be used some features of emWin will not work. The list below shows these features:

- Cursors and Sprites
- XOR-operations, required for text cursors in EDIT and MULTIEDIT widgets
- Alpha blending
- Antialiasing

This is valid for all drivers where one data unit (8 or 16 bit) represents one pixel. Display drivers, where one data unit represents more than one pixel, can not be used if no display data cache is available and the display is not readable. An example is the LCDPage1bpp driver where one byte represents 8 pixels.

15.3 Detailed display driver descriptions

15.3.1 LCDLin driver

This driver comes with 3 different variants:

LCDLin driver (32/16/8 bit access)

This variant does not use a fixed kind of memory access. In dependence of the operation which should be done it chooses the most economic method of memory access. So this driver should provide the best performance on systems with 32 bit full bus interface and is the most recommended one for internal display controllers.

LCDLin driver (32 bit access)

This variant strictly accesses the video RAM in 32 bit units and is also recommended to be used with internal display controllers. It offers a similar performance as the 32/16/8 variant.

LCDLin driver (16/8 bit access)

The least variant accesses the video memory either in 8 bit or in 16 bit units, dependent on the configuration. Using this variant is recommended for external display controllers with 8- or 16 bit memory access.

15.3.1.1 LCDLin driver (32/16/8 bit access)

This variant of the LCDLin driver accesses the video memory with the most economic method depending on the required operation. If for example an area should be filled the driver uses 32 bit access as far as possible. But if for example only a single pixel should be set in a 16bpp configuration it uses 16 bit access.

This driver can be used with any display controller with linear memory organization (as described below) and full bus interface. Most controllers for bigger displays and higher color depth (typically starting at quarter VGA) comply with this requirement and can therefore be controlled by this driver.

Supported hardware

Controllers

The following table list the supported controllers and their assigned numbers for LCD_CONTROLLER, as well as the level of support:

#	LCD controller	Add. info
32168	Any display controller with linear video memory which should be accessed in 8, 16 or 32 bit mode in dependence of the required operation.	The LUT (color look up table or palette RAM) is not handled by the driver. If a LUT mode is used, the application program is responsible for initialization of the LUT. LUT support can be added by using the macro LCD_SET_LUT_ENTRY.

Bits per pixel

Currently supported color depths are 16 and 32 bpp.

Interfaces

The driver supports any 32 bit full bus interface.

Display data RAM organization

The display RAM organisation is the same as for the 32 bit variant. For details please refer to the previous subchapter 'LCDLin driver (32 bit access)'.

Additional RAM requirements of the driver

None.

Additional driver functions

None.

Hardware configuration

This driver requires a full bus interface for hardware access as described in Chapter 17: "Low-Level Configuration".

Available configuration macros

The following table lists the macros which must be defined for hardware access:

Macro	Explanation
<code>LCD_ENDIAN_BIG</code>	Should be set to 1 for big endian mode, 0 for little endian mode.
<code>LCD_VRAM_ADR</code>	Defines the start address of the video memory.

Additional configuration switches

The following table shows optional configuration switches available for this driver:

Macro	Explanation
<code>LCD_FILL_RECT</code>	Function replacement macro which defines a function to be called by the driver for filling rectangles.
<code>LCD_OFF</code>	Function replacement macro which switches the LCD off.
<code>LCD_ON</code>	Function replacement macro which switches the LCD on.
<code>LCD_SET_LUT_ENTRY</code>	Used to set a single lookup table or palette RAM entry.

LCD_FILL_RECT()

Description

This macro can be used for defining a function which should be called by the display driver for filling rectangles.

Type

Function replacement.

Prototype

```
#define LCD_FILL_RECT(x0, y0, x1, y1, Index)
```

Parameter	Meaning
<code>x0</code>	Leftmost X-position of the rectangle to be filled.
<code>y0</code>	Topmost Y-position of the rectangle to be filled.
<code>x1</code>	Rightmost X-position of the rectangle to be filled.
<code>y1</code>	Bottommost Y-position of the rectangle to be filled.
<code>Index</code>	Color index to be used for filling.

Add. information

If this macro is defined, the driver calls the function defined by this macro instead of using its own filling routine. Using this macro can make sense if for example a BitBLT engine should be used for filling instead of the driver internal filling function. Index values are in the range of 0 - ((1 << LCD_BITS_PER_PIXEL) - 1).

Example

```
void CustomFillRect(int x0, int y0, int x1, int y1, int Index);

#define LCD_FILL_RECT(x0, y0, x1, y1, Index) CustomFillRect(x0, y0, x1, y1, Index)
```

15.3.1.2 LCDLin driver (32 bit access)

Generally display controller with linear video memory can be accessed with the LCD-Lin driver for 8 and 16 bit access and with the LCDLin driver for 32 bit access. If 32 bit access is possible, it is recommended to use the 32 bit driver with the better performance.

This driver can be used with any display controller with linear memory organization (as described below) and full bus interface (32 bit data bus). Most controllers for bigger displays and higher color depth (typically starting at quarter VGA) comply with this requirement and can therefore be controlled by this driver.

Supported hardware

Controllers

The following table list the supported controllers and their assigned numbers for LCD_CONTROLLER, as well as the level of support:

#	LCD controller	Add. info
3200	Any display controller with linear video memory in 4, 8 or 16 bits/pixel mode such as the build in display controller of Sharp LH754XX, ARM or MIPS CPU's.	The LUT (color look up table or palette RAM) is not handled by the driver. If a LUT mode is used, the application program is responsible for initialization of the LUT. LUT support can be added by using the macro LCD_SET_LUT_ENTRY.

Bits per pixel

Supported color depths are 1, 2, 4, 8, 16, 24 and 32 bpp.

Interfaces

The driver supports any 32 bit full bus interface.

Display data RAM organization

Big endian

1 bit / pixel

2 bit / pixel

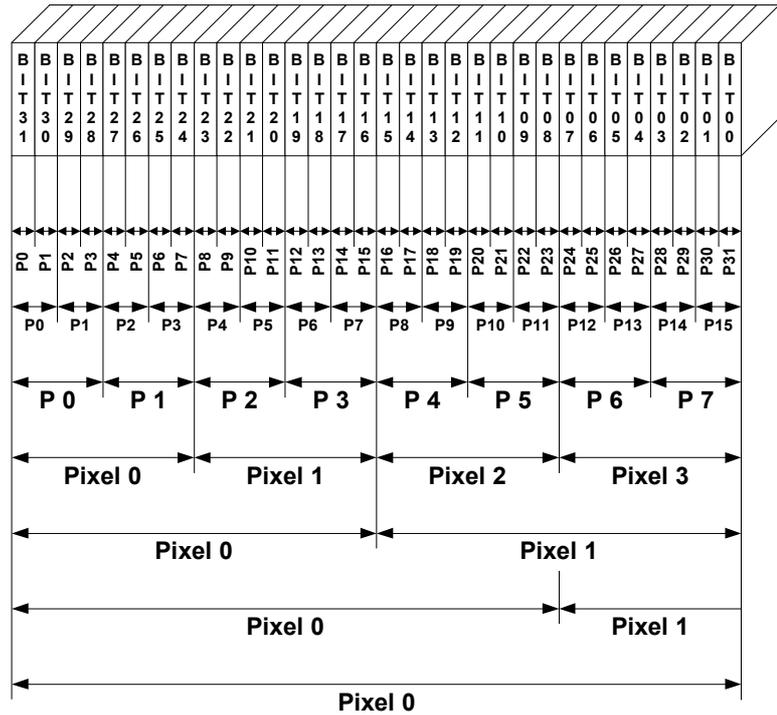
4 bit / pixel

8 bit / pixel

16 bit / pixel

24 bit / pixel

32 bit / pixel



Little endian

1 bit / pixel

2 bit / pixel

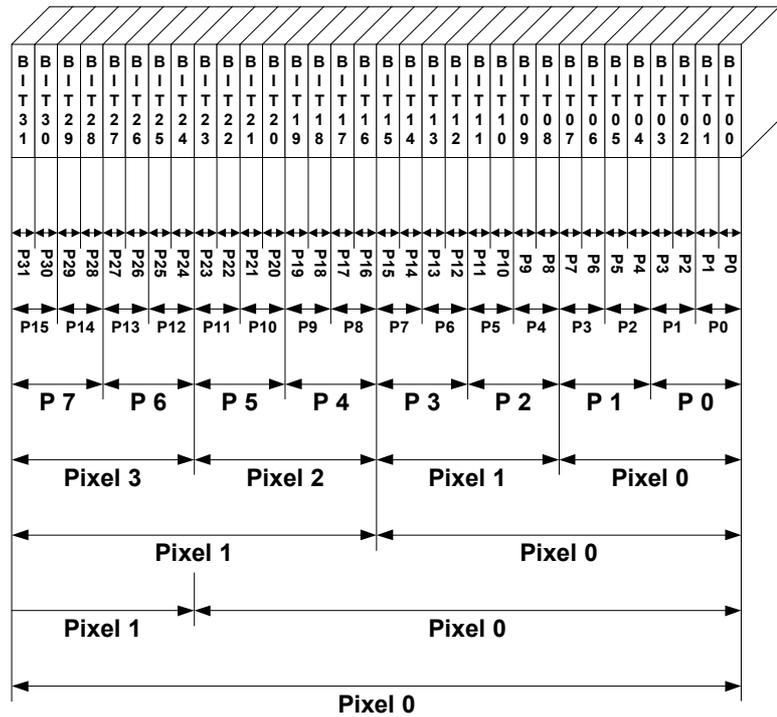
4 bit / pixel

8 bit / pixel

16 bit / pixel

24 bit / pixel

32 bit / pixel



The picture above shows the relation between the display memory and the pixels of the LCD in terms of the color depth and the endian mode.

Little endian video mode

Least significant bits are used and output first. The least significant bits are for the first (left-most) pixel.

Big endian video mode

Most significant bits are used and output first. The most significant bits are for the first (left-most) pixel.

Additional RAM requirements of the driver

None.

Additional driver functions

None.

Hardware configuration

This driver requires a full bus interface for hardware access as described in Chapter 17: "Low-Level Configuration (LCDConf.h)".

Available configuration macros

The following table lists the macros which must be defined for hardware access:

Macro	Explanation
LCD_ENDIAN_BIG	Should be set to 1 for big endian mode, 0 for little endian mode.
LCD_VRAM_ADR	Defines the start address of the video memory.

Available configuraion routines

The following table lists the available runtime configuration routines:

Routine	Explanation
LCD_SetSizeEx()	Changes the size of the visible area.
LCD_SetVRAMAddrEx()	Changes the video RAM start address.
LCD_SetVSizeEx()	Changes the size of the virtual display area.

For more details please refer to 15.4.1 "Display driver API" later in this chapter.

Additional configuration switches

The following table shows optional configuration switches available for this driver:

Macro	Explanation
LCD_FILL_RECT	Function replacement macro which defines a function to be called by the driver for filling rectangles.
LCD_LIN_SWAP	Swaps pixels within a byte.
LCD_OFF	Function replacement macro which switches the LCD off.
LCD_ON	Function replacement macro which switches the LCD on.
LCD_READ_MEM	Read the contents of video memory of controller.
LCD_SET_LUT_ENTRY	Used to set a single lookup table or palette RAM entry.
LCD_WRITE_MEM	Write to video memory of controller.

LCD_FILL_RECT()

Description

This macro can be used for defining a function which should be called by the display driver for filling rectangles.

Type

Function replacement.

Prototype

```
#define LCD_FILL_RECT(x0, y0, x1, y1, Index)
```

Parameter	Meaning
x0	Leftmost X-position of the rectangle to be filled.
y0	Topmost Y-position of the rectangle to be filled.
x1	Rightmost X-position of the rectangle to be filled.
y1	Bottommost Y-position of the rectangle to be filled.
Index	Color index to be used for filling.

Add. information

If this macro is defined, the driver calls the function defined by this macro instead of using its own filling routine. Using this macro can make sense if for example a BitBLT engine should be used for filling instead of the driver internal filling function. Index values are in the range of 0 - ((1 << LCD_BITS_PER_PIXEL) - 1).

Example

```
void CustomFillRect(int x0, int y0, int x1, int y1, int Index);
```

```
#define LCD_FILL_RECT(x0, y0, x1, y1, Index) CustomFillRect(x0, y0, x1, y1, Index)
```

LCD_LIN_SWAP()

Description

This macro enables swapping of pixels within one byte.

Type

Numeric.

Prototype

```
#define LCD_LIN_SWAP
```

Add. information

Sometimes a display driver like the embedded display driver of the Motorola MX1 has a different pixel assignment as the default assignment shown under 'Display data RAM organization'. In this case the macro `LCD_LIN_SWAP` can be used to swap the pixels within one byte after reading from and before writing to the video RAM. If the value of the macro is > 0, pixel swapping is activated. The value of `LCD_LIN_SWAP` defines the swapping mode. The following table shows the supported swapping modes in dependence of the defined value:

Value	Default	After swapping																
1	<table border="1"><tr><td>P7</td><td>P6</td><td>P5</td><td>P4</td><td>P3</td><td>P2</td><td>P1</td><td>P0</td></tr></table>	P7	P6	P5	P4	P3	P2	P1	P0	<table border="1"><tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P5</td><td>P6</td><td>P7</td></tr></table>	P0	P1	P2	P3	P4	P5	P6	P7
P7	P6	P5	P4	P3	P2	P1	P0											
P0	P1	P2	P3	P4	P5	P6	P7											
2	<table border="1"><tr><td>P3</td><td>P2</td><td>P1</td><td>P0</td></tr></table>	P3	P2	P1	P0	<table border="1"><tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td></tr></table>	P0	P1	P2	P3								
P3	P2	P1	P0															
P0	P1	P2	P3															
4	<table border="1"><tr><td>P1</td><td>P0</td></tr></table>	P1	P0	<table border="1"><tr><td>P0</td><td>P1</td></tr></table>	P0	P1												
P1	P0																	
P0	P1																	

Example

```
#define LCD_LIN_SWAP 1
```

LCD_READ_MEM(), LCD_WRITE_MEM()

The default definitions of these macros are:

```
#define LCD_READ_MEM(Off)          (((U32 *)LCD_VRAM_ADR + (U32)Off))
#define LCD_WRITE_MEM(Off, Data)  (((U32 *)LCD_VRAM_ADR + (U32)Off) = Data)
```

These macros normally need not to be defined in the configuration file. It makes only sense to define them, if the memory access should not work as defined like shown above. In this case these macros can be defined in the configuration file `LCDConf.h` instead of the video memory start address.

How to migrate from LCDLin to LCDLin32

The driver for 8 and 16 bit access needs the definition of 2 memory access macros, `LCD_READ_MEM` and `LCD_WRITE_MEM`. The driver for 32 bit access needs the definition of the display RAM memory address. The following sample shows how to define the memory access.

Example

Configuration for 16 bit access:

```
#define LCD_CONTROLLER 1300
#define LCD_READ_MEM(Off)      (((U16*) (0xc00000 + ((U32) (Off)) << 1)))
#define LCD_WRITE_MEM(Off,Data) (((U16*) (0xc00000 + ((U32) (Off)) << 1)) = Data)
```

Configuration for 32 bit access:

```
#define LCD_CONTROLLER 3200
#define LCD_VRAM_ADR 0xc00000
```

15.3.1.3 LCDLin driver (8 and 16 bit access)

Generally display controller with linear video memory can be accessed with the LCDLin driver for 8 and 16 bit access and with the LCDLin driver for 32 bit access. If 32 bit access is possible, it is recommended to use the 32 bit driver with the better performance.

This driver can be used with any LCD Controller with linear memory organization (as described below) and full bus interface (8 or 16 bit data bus). Most controllers for bigger displays and higher color depth (typically starting at quarter VGA) comply with this requirement and can therefore be controlled by this driver.

Supported hardware

Controllers

The following table list the supported controllers and their assigned numbers for `LCD_CONTROLLER`, as well as the level of support:

#	LCD controller	Add. info
1300	Any LCD controller with linear memory and full bus interface, such as: Epson SED1352, S1D13502 Epson SED1353, S1D13503 Epson S1D13700 (direct interface) Solomon SSD1905 Fujitsu MB86290A (Cremson) Fujitsu MB86291 (Scarlet) Fujitsu MB86292 (Orchid) Fujitsu MB86293 (Coral Q) Fujitsu MB86294 (Coral B) Fujitsu MB86295 (Coral P) Microcontrollers with built-in LCD controllers such as Sharp LH79531	The LUT (color look up table) is not handled by the driver. If a LUT mode is used (typically 16 or 256 colors), the application program is responsible for the initialization of the LUT.
1301	Toshiba Capricorn 2	LUT is handled by driver if required All layers can be supported
1304	Epson S1D13A03, S1D13A04, S1D13A05	LUT is handled by driver if required 2 D Engine supported (BitBLT)
1354	Epson SED1354, S1D13504	LUT is handled by driver if required
1356	Epson SED1356, S1D13506	LUT is handled by driver if required. 2 D Engine supported (BitBLT)

#	LCD controller	Add. info
1374	Epson SED1374, S1D13704	LUT is handled by driver if required
1375	Epson SED1375, S1D13705	LUT is handled by driver if required
1376	Epson SED1376, S1D13706	LUT is handled by driver if required
1386	Epson SED1386, S1D13806	LUT is handled by driver if required 2 D Engine supported (BitBLT)

Bits per pixel

Supported color depths are 1, 2, 4, 8 and 16 bpp.

Interfaces

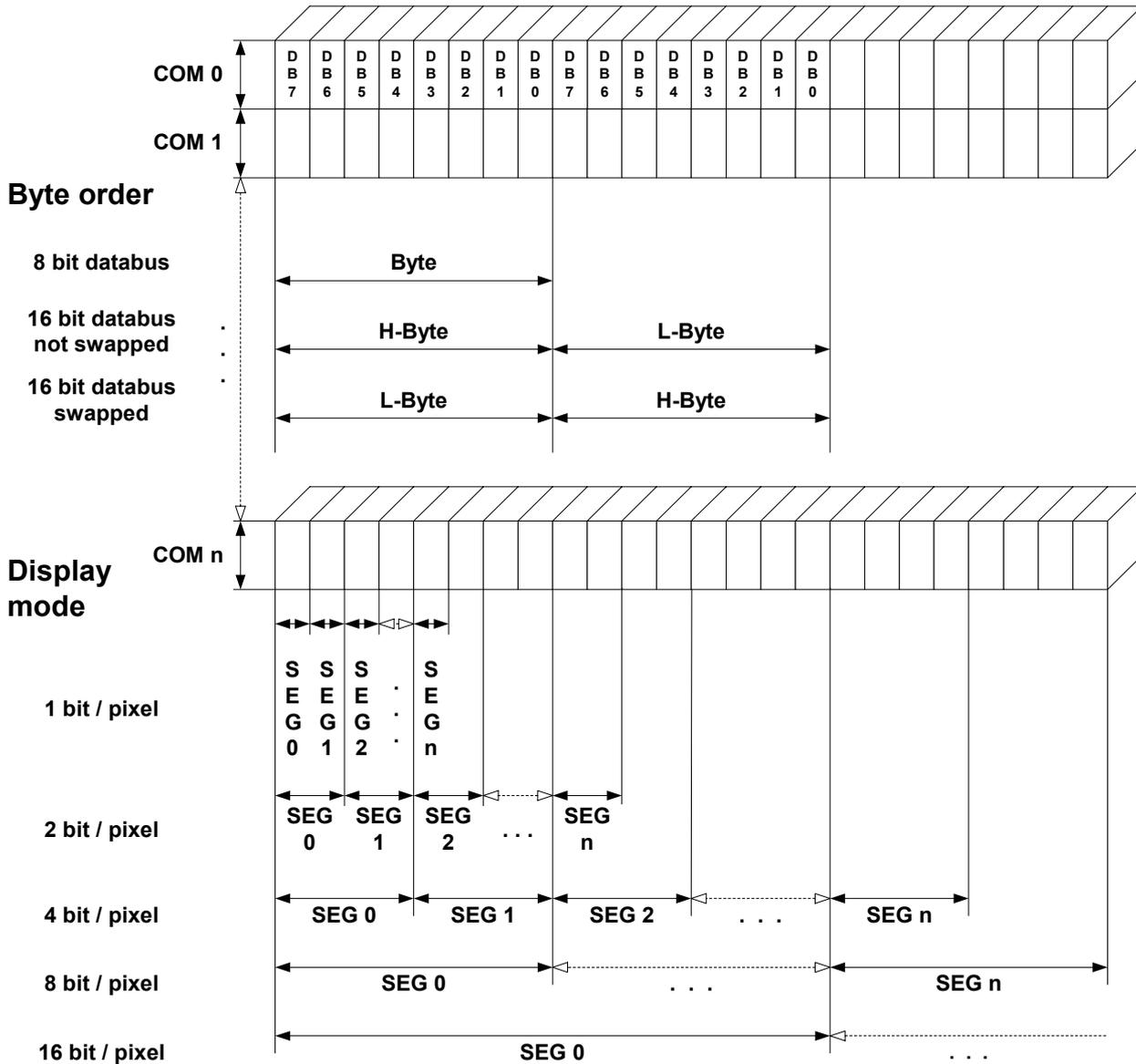
The chips supported by this driver can be interfaced in 8/16-bit parallel (full bus) modes.

The driver supports both interfaces. Please refer to the respective LCD controller manual in order to determine if your chip can be interfaced in 8-bit mode.

Built-in LCD controllers

This driver can also be used with built-in LCD controllers. In this case, either 8 or 16 bit access can be selected. It is typically best to use 8 bit access if the built-in LCD controller operates in an 8-bpp mode and likewise to use 16 bit access if the built-in LCD-controller operates in a 16-bpp mode.

Display data RAM organization



The picture above shows the relation between the display memory and the SEG and COM lines of the LCD in terms of the color depth.

Additional RAM requirements of the driver

None.

Additional driver functions

None.

Hardware configuration

This driver requires a full bus interface for hardware access as described in Chapter 17: "Low-Level Configuration". The following table lists the macros which must be defined for hardware access:

Macro	Explanation
LCD_READ_MEM	Read the contents of video memory of controller.
LCD_READ_REG	Read the contents of a configuration register of controller.
LCD_WRITE_MEM	Write to video memory (display data RAM) of controller.
LCD_WRITE_REG	Write to a configuration register of controller.

Additional configuration switches

The following table shows optional configuration switches available for this driver:

Macro	Explanation
<code>LCD_BUSWIDTH</code>	Select bus-width (8/16) of LCD controller/CPU interface. Default is 16.
<code>LCD_CNF4</code>	Endian mode selection for S1D13A03-A05 controllers. Default is 0.
<code>LCD_ENABLE_MEM_ACCESS</code>	Switch the M/R signal to memory access. Only used for S1D13506 and S1D13806 LCD controllers.
<code>LCD_ENABLE_REG_ACCESS</code>	Switch the M/R signal to register access. Only used for S1D13506 and S1D13806 LCD controllers.
<code>LCD_FILL_RECT</code>	Function replacement macro which defines a function to be called by the driver for filling rectangles. For details please refer to LCDLin-driver (32 bit access).
<code>LCD_ON</code>	Function replacement macro which switches the LCD on
<code>LCD_OFF</code>	Function replacement macro which switches the LCD off
<code>LCD_SET_LUT_ENTRY</code>	Function replacement macro used to set a single lookup table or palette RAM entry.
<code>LCD_SWAP_BYTE_ORDER</code>	Inverts the endian mode (swaps the high and low bytes) between CPU and LCD controller when using a 16-bit bus interface.
<code>LCD_USE_BITBLT</code>	If set to 0, it disables the BitBLT engine. If set to 1 (the default value), the driver will use all available hardware acceleration.

Additional info for S1D13A03, S1D13A04 and S1D13A05

LCD_CNF4

The configuration switch `LCD_CNF4` configures the endian mode selection. If the `CNF4` pin of the controller is configured as high the macro should be 1, if the pin is low it should be 0 (default). To set the endian mode to big endian the following line should be added to `LCDCConf.h`:

```
#define LCD_CNF4 (1) /* Selects the big endian mode */
```

Additional info for S1D13806, S1D13A03, S1D13A04 and S1D13A05

LCD_SWAP_RB

The configuration switch `LCD_SWAP_RB` (swaps the red and blue components) must be activated (set to 1) by inserting the following line into `LCDCConf.h`:

```
#define LCD_SWAP_RB (1) /* Has to be set */
```

LCD_X_InitController()

When writing or modifying the initialization macro, consider the following:

- To initialize the embedded SDRAM, bit 7 of register 20 (SDRAM initialization bit) must be set to 1 (a minimum of 200 µs after reset).
- When the SDRAM initialization bit is set, the actual initialization sequence occurs at the first SDRAM refresh cycle. The initialization sequence requires approximately 16 MCLKs to complete, and memory accesses cannot be made while the initialization is in progress.

For more information, please see the LCD controller documentation.

LCD_READ_REG, LCD_WRITE_REG

In order for the BitBLT engine to work, the data type of the offset must be unsigned long. This is set with the configuration macros `LCD_READ_REG` and `LCD_WRITE_REG` as follows:

```
#define LCD_READ_REG(Off) *((volatile U16*) (0x800000+(((U32) (Off))<<1)))
#define LCD_WRITE_REG(Off,Data) *((volatile U16*) (0x800000+(((U32) (Off))<<1)))=Data
```

15.3.2 LCD667XX driver

Supported hardware

Controllers

This driver works with the following display controllers:

- Epson S1D13743
- Himax HX8301, HX8312
- Hitachi HD66766, HD66789, HD66772
- Ilitek ILI9320, ILI9325, ILI9220, ILI9161
- LG Electronics LGDP4531
- MagnaChip D54E4PA7551
- Novatek NT39122, NT7573
- OriseTech SPFD5408, SPFD5420A
- Renesas R63401, R61509, R61516, R61505
- Samsung S6D0129, S6D0110A, S6D0117
- Sitronix ST7628, ST7712, ST7637
- Sharp LR38825, LCY-A06003
- Solomon SSD1289
- Toshiba JBT6K71

Bits per pixel

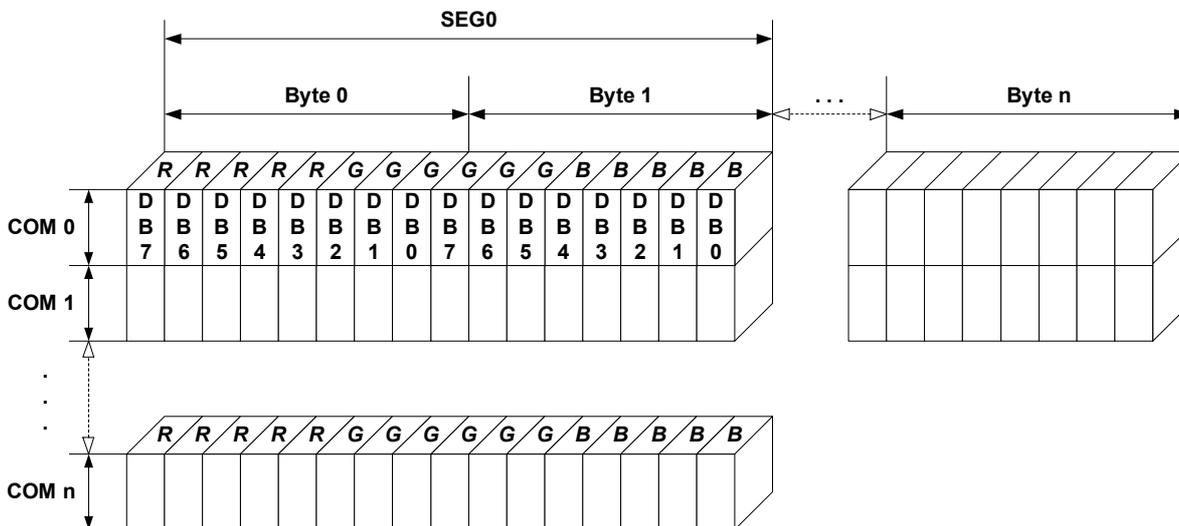
Supported color depth is 16 bpp.

Interfaces

The driver supports 8-bit parallel, 16 bit parallel and 3 pin SPI interface. Default mode is 8-bit parallel.

Display data RAM organization

16 bits per pixel, fixed palette = 565



The picture above shows the relation between the display memory and the SEG and COM lines of the LCD.

Additional RAM requirements

This LCD driver can be used with and without a display data cache, containing a complete copy of the contents of the LCD data RAM. The amount of memory used by the cache is: $LCD_XSIZE \times LCD_YSIZE \times 2$ bytes. Using a cache is only recommended if a lot of drawing operations uses the XOR drawing mode. A cache would avoid reading the display data in this case. Normally the use of a cache is not recommended.

The driver can be used with a write buffer used for drawing multiple pixels of the same color. If multiple pixels of the same color should be drawn the driver first fills the buffer and then executes only one time the macro `LCD_WRITEM_A1` to transfer the data to the display controller. The default buffer size is 500 bytes.

Additional driver functions

None.

Hardware configuration

This driver accesses the hardware with a simple bus interface or with a 3 pin SPI interface. The following table lists the macros which must be defined for hardware access:

Macro	Explanation
<code>LCD_NUM_DUMMY_READS</code>	Number of required dummy reads if a read operation should be executed. The default value is 2. If using a serial interface the display controllers HD66766 and HD66772 need 5 dummy reads. Sharp LR38825 needs 3 dummy reads with a 8-bit bus.
<code>LCD_REG01</code>	This macro is only required if a Himax HX8312A is used. Unfortunately the register 0x01 (Control register 1) contains orientation specific settings as well as common settings. So this macro should contain the contents of this register.
<code>LCD_SERIAL_ID</code>	With a serial interface this macro defines the ID signal of the device ID code. It should be 0 (default) or 1.
<code>LCD_USE_SERIAL_3PIN</code>	Should be set to 1 if the serial interface is used. Default is 0.
<code>LCD_USE_PARALLEL_16</code>	Should be set to 1 if the 16 bit parallel interface is used. Default is 0.
<code>LCD_WRITE_BUFFER_SIZE</code>	Defines the size of the write buffer. Using a write buffer increases the performance of the driver. If multiple pixels should be written with the same color, the driver first fills the buffer and then writes the contents of the buffer with one execution of the macro <code>LCD_WRITEM_A1</code> , instead of multiple macro executions. The default buffer size is 500 bytes.
<code>LCD_WRITE_A0</code>	Write a byte to display controller with RS-line low.
<code>LCD_WRITE_A1</code>	Write a byte to display controller with RS-line high.
<code>LCD_READM_A1</code>	Read multiple bytes (8 bit parallel interface) or multiple words (16 bit parallel interface) from display controller with RS-line high.
<code>LCD_WRITEM_A1</code>	Write multiple bytes (8 bit parallel interface) or multiple words (16 bit parallel interface) to display controller with RS-line high.
<code>LCD_WRITEM_A0</code>	Write multiple bytes (8 bit parallel interface) or multiple words (16 bit parallel interface) to display controller with RS-line low.

The driver initializes the 'Driver Output Mode' and 'Entry Mode' register itself. The user does not need to initialize this registers in `LCD_X_InitController()`.

Additional configuration switches

None.

Special requirements

None.

15.3.3 LCDTemplate driver

This driver is part of the basic package and can be easily adapted to each display controller. It contains the complete functionality needed for a display driver.

Adapting the template driver

To adapt the driver to a currently not supported display controller you only have to adapt the routines `LCD_L0_SetPixelIndex()` and `LCD_L0_GetPixelIndex()`. The upper layers calling this routines makes sure that the given coordinates are in range, so that no check on the parameters needs to be performed.

If a display is not readable the function `LCD_L0_GetPixelIndex()` won't be able to read back the contents of the display data RAM. In this case a display data cache should be implemented in the driver, so that the contents of each pixel is known by the driver. If no data cache is available in this case some functions of emWin will not work right. These are all functions which need to invert pixels. Especially the XOR draw mode and the drawing of text cursors (which also uses the XOR draw mode) will not work right. A simple application which do not use the XOR draw mode will also work without adapting the function `LCD_L0_SetPixelIndex()`.

In a second step, a new driver should be modified to use its own number for activation (`LCD_CONTROLLER`), and optionally be optimized to improve drawing speed.

15.3.4 LCDNull driver

This driver is part of the basic package and can be used for measurement purpose. It contains all API functions of a LCD driver without any function.

Using this driver

Since the driver contains only 'empty' API functions it makes it possible to measure the time difference used for some GUI-operations between using the real hardware driver and this empty driver. The time difference is the time used for the LCD display operation.

15.4 LCD layer and display driver API

emWin requires a driver for the hardware. This chapter explains what an LCD driver for emWin does and what routines it supplies to emWin (the application program interface, or API).

Under most circumstances, you probably do not need to read this chapter, as most calls to the LCD layer of emWin will be done through the GUI layer. In fact, we recommend that you only call LCD functions if there is no GUI equivalent (for example, if you wish to modify the lookup table of the LCD controller directly). The reason for this is that LCD driver functions are not thread-safe, unlike their GUI equivalents. They should therefore not be called directly in multitask environments.

15.4.1 Display driver API

The table below lists the available emWin LCD-related routines in alphabetical order. Detailed descriptions of the routines can be found in the sections that follow.

LCD_L0: Driver routines

Routine	Explanation
Init & display control group	
LCD_L0_Init()	Initialize the display.
LCD_L0_Off()	Switch LCD off.
LCD_L0_On()	Switch LCD on.
Drawing group	
LCD_L0_DrawBitmap()	Universal draw bitmap routine.
LCD_L0_DrawHLine()	Draw a horizontal line.
LCD_L0_DrawPixel()	Draw a pixel in the current foreground color.
LCD_L0_DrawVLine()	Draw a vertical line.
LCD_L0_FillRect()	Fill a rectangular area.
LCD_L0_SetPixelIndex()	Draw a pixel in a specified color.
LCD_L0_XorPixel()	Invert a pixel.
"Get" group	
LCD_L0_GetPixelIndex()	Returns the index of the color of a specific pixel.
"Set" group	
LCD_L0_SetOrg()	Sets the origin of the upper left corner.
Lookup table group	
LCD_L0_SetLUTEntry()	Modify a single entry of LUT.
Misc. group (optional)	
LCD_L0_ControlCache()	Lock/unlock/flush LCD cache.

User defined routines

Routine	Explanation
LCD_X_InitController()	Called by the display driver to get the display controller initialized.

LCD: LCD layer routines

Routine	Explanation
"Get" group	
LCD_GetBitsPerPixel()	Return the number of bits per pixel.
LCD_GetBitsPerPixelEx()	Returns the number of bits per pixel of given layer/display.
LCD_GetFixedPalette()	Return the fixed palette mode.
LCD_GetFixedPaletteEx()	Returns the fixed palette mode of given layer/display.
LCD_GetNumColors()	Return the number of available colors.
LCD_GetNumColorsEx()	Returns the number of available colors of given layer/display.
LCD_GetVXSize()	Return virtual X-size of LCD in pixels.
LCD_GetVXSizeEx()	Returns virtual X-size of given layer/display in pixels.
LCD_GetVYSize()	Return virtual Y-size of LCD in pixels.
LCD_GetVYSizeEx()	Returns virtual Y-size of given layer/display in pixels.
LCD_GetXMag()	Returns the magnification factor in x.
LCD_GetXMagEx()	Returns the magnification factor of given layer/display in x.
LCD_GetXSize()	Return physical X-size of LCD in pixels.
LCD_GetXSizeEx()	Returns physical X-size of given layer/display in pixels.
LCD_GetYMag()	Returns the magnification factor in y.
LCD_GetYMagEx()	Returns the magnification factor of given layer/display in y.
LCD_GetYSize()	Return physical Y-size of LCD in pixels.
LCD_GetYSizeEx()	Returns physical Y-size of given layer/display in pixels.

15.4.2 Driver routines

15.4.2.1 Init & display control group

LCD_L0_Init()

Description

Initializes the LCD using the configuration settings in `LCDConf.h`. This routine is called automatically by `GUI_Init()` if the upper GUI layer is used and therefore should not need to be called manually.

Prototype

```
void LCD_L0_Init (void);
```

LCD_L0_Off(), LCD_L0_On()

Description

Switch the display off or on, respectively.

Prototypes

```
void LCD_L0_Off(void);
void LCD_L0_On(void);
```

Add. information

Use of these routines does not affect the contents of the video memory or other settings. You may therefore safely switch off the display and switch it back on without having to refresh the contents.

15.4.2.2 Drawing group

LCD_L0_DrawBitmap()

Description

Draws a pre-converted bitmap.

Prototype

```
LCD_L0_DrawBitMap(int x0, int y0,
                  int Xsize, int Ysize,
                  int BitsPerPixel,
                  int BytesPerLine,
                  const U8* pData, int Diff,
                  const LCD_PIXELINDEX* pTrans);
```

Parameter	Meaning
<code>x0</code>	Upper left X-position of bitmap to draw.
<code>y0</code>	Upper left Y-position of bitmap to draw.
<code>Xsize</code>	Number of pixels in horizontal direction.
<code>Ysize</code>	Number of pixels in vertical direction.
<code>BitsPerPixel</code>	Number of bits per pixel.
<code>BytesPerLine</code>	Number of bytes per line of the image.
<code>pData</code>	Pointer to the actual image, the data that defines what the bitmap looks like.
<code>Diff</code>	Number of pixels to skip from the left side.

LCD_L0_DrawHLine()

Description

Draws a horizontal line one pixel thick, at a specified position using the current foreground color.

Prototype

```
void LCD_L0_DrawHLine(int x0, int y, int x1);
```

Parameter	Meaning
x0	Start position of line.
y	Y-position of line to draw.
x1	End position of line.

Add. information

With most LCD controllers, this routine executes very quickly because multiple pixels can be set at once and no calculations are needed. If it is clear that horizontal lines are to be drawn, this routine executes faster than the `DrawLine` routine.

LCD_L0_DrawPixel()

Description

Draws one pixel at a specified position using the current foreground color.

Prototype

```
void LCD_L0_DrawPixel(int x, int y);
```

Parameter	Meaning
x	X-position of pixel to draw.
y	Y-position of pixel to draw.

LCD_L0_DrawVLine()

Description

Draws a vertical line one pixel thick, at a specified position using the current foreground color.

Prototype

```
void LCD_L0_DrawVLine(int x , int y0, int y1);
```

Parameter	Meaning
x	X-position of line to draw.
y0	Start position of line.
y1	End position of line.

Add. information

With most LCD-controllers, this routine executes very quickly because multiple pixels can be set at once and no calculations are needed. If it is clear that horizontal lines are to be drawn, this routine executes faster than the `DrawLine` routine.

LCD_L0_FillRect()

Description

Draws a filled rectangle at a specified position using the current foreground color.

Prototype

```
void LCD_L0_FillRect(int x0, int y0, int x1, int y1);
```

Parameter	Meaning
x0	Upper left X-position.
y0	Upper left Y-position.
x1	Lower right X-position.
y1	Lower right Y-position.

LCD_L0_SetPixelIndex()**Description**

Draws one pixel using a specified color

Prototype

```
void LCD_L0_SetPixelIndex(int x, int y, int ColorIndex);
```

Parameter	Meaning
x	X-position of pixel to draw.
y	Y-position of pixel to draw.
ColorIndex	Color to be used.

LCD_L0_XorPixel()**Description**

Inverts one pixel.

Prototype

```
void LCD_L0_XorPixel(int x, int y);
```

Parameter	Meaning
x	X-position of pixel to invert.
y	Y-position of pixel to invert.

15.4.2.3 "Get" group**LCD_L0_GetPixelIndex()****Description**

Returns the RGB color index of a specified pixel.

Prototype

```
int LCD_L0_GetPixelIndex(int x, int y);
```

Parameter	Meaning
x	X-position of pixel.
y	Y-position of pixel.

Return value

The index of the pixel.

Add. information

For further information see Chapter 10: "Colors".

15.4.2.4 Lookup table (LUT) group

LCD_L0_SetLUTEntry()

Description

Modifies a single entry to the LUT of the LCD controller(s).

Prototype

```
void LCD_L0_SetLUTEntry(U8 Pos, LCD_COLOR Color);
```

Parameter	Meaning
<code>Pos</code>	Position within the lookup table. Should be less than the number of colors, e.g. 0-3 for 2bpp, 0-15 for 4bpp, 0-255 for 8bpp.
<code>Color</code>	24-bit RGB value. The closest value possible will be used for the LUT. If a color LUT is to be initialized, all 3 components are used. In monochrome modes the green component is used, but it is still recommended (for better understanding of the program code) to set all 3 colors to the same value (such as 0x555555 or 0xa0a0a0).

15.4.2.5 Miscellaneous group

LCD_L0_ControlCache()

Description

Locks, unlocks or flushes the cache. This routine may be used to set the cache to a locked state, in which all drawing operations on the driver cause changes in the video memory's cache (in CPU RAM), but do not cause any visible output. Unlocking or flushing then causes those changes to be written to the display. This can help to avoid flickering of the display and also accelerate drawing. It does not matter how many different drawing operations are executed; the changes will all be written to the display at once. In order to be able to do this, `LCD_SUPPORT_CACHECONTROL` must be enabled in the configuration file.

Prototype

```
U8 LCD_ControlCache(U8 command);
```

Parameter	Meaning
<code>command</code>	Specify the command to be given to the cache. Use the symbolic values in the table below.

Permitted values for parameter <code>command</code>	
<code>LCD_CC_UNLOCK</code>	Set the default mode: cache is transparent.
<code>LCD_CC_LOCK</code>	Lock the cache, no write operations will be performed until cache is unlocked or flushed.
<code>LCD_CC_FLUSH</code>	Flush the cache, writing all modified data to the video RAM.

Return value

Information on the state of the cache. Ignore.

Add. information

When the cache is locked, the driver maintains a "hitlist" -- a list of bytes which have been modified and need to be written to the display. This hitlist uses 1 bit per byte of video memory.

This is an optional feature which is not supported by all LCD drivers

Example

The code in the following example performs drawing operations on the display which overlap. In order to accelerate the update of the display and to avoid flickering, the cache is locked before and unlocked after these operations.

```

LCD_ControlCache(LCD_CC_LOCK);

GUI_FillCircle(30,30,20);
GUI_SetDrawMode(GUI_DRAWMODE_XOR);
GUI_FillCircle(50,30,10);
GUI_SetTextMode(GUI_TEXTMODE_XOR);
GUI_DispStringAt("Hello world\n",0,0);
GUI_DrawHLine(16, 5,25);
GUI_DrawHLine(18, 5,25);
GUI_DispStringAt("XOR Text",0,20);
GUI_DispStringAt("XOR Text",0,60);

LCD_ControlCache(LCD_CC_UNLOCK);

```

15.4.3 Callback routines

LCD_X_InitController()

Description

The function is called by the display controller. The job of the routine is to get the display controller registers initialized right.

Prototype

```
void LCD_X_InitController(unsigned LayerIndex);
```

Parameter	Meaning
LayerIndex	Index of layer to be initialized.

Add. information

The sample folder contains several samples for a large number of display controllers. This function replaces the macro `LCD_INIT_CONTROLLER` which was used in older versions.

15.4.4 LCD layer routines

15.4.4.1 "Get" group

LCD_GetBitsPerPixel()

Description

Returns the number of bits per pixel.

Prototype

```
int LCD_GetBitsPerPixel(void);
```

Return value

Number of bits per pixel.

LCD_GetBitsPerPixelEx()

Description

Returns the number of bits per pixel.

Prototype

```
int LCD_GetBitsPerPixelEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

Number of bits per pixel.

LCD_GetFixedPalette()**Description**

Returns the fixed palette mode.

Prototype

```
int LCD_GetFixedPalette(void);
```

Return value

The fixed palette mode. See Chapter 10: "Colors" for more information on fixed palette modes.

LCD_GetFixedPaletteEx()**Description**

Returns the fixed palette mode.

Prototype

```
int LCD_GetFixedPaletteEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

The fixed palette mode. See Chapter 10: "Colors" for more information on fixed palette modes.

LCD_GetNumColors()**Description**

Returns the number of currently available colors on the LCD.

Prototype

```
int LCD_GetNumColors(void);
```

Return value

Number of available colors

LCD_GetNumColorsEx()**Description**

Returns the number of currently available colors on the LCD.

Prototype

```
U32 LCD_GetNumColorsEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

Number of available colors.

LCD_GetVXSize(), LCD_GetVYSize()

Description

Returns the virtual X- or Y-size, respectively, of the LCD in pixels. In most cases, the virtual size is equal to the physical size.

Prototype

```
int LCD_GetVXSize(void)
int LCD_GetVYSize(void)
```

Return value

Virtual X/Y-size of the display.

LCD_GetVXSizeEx(), LCD_GetVYSizeEx()

Description

Returns the virtual X- or Y-size, respectively, of the LCD in pixels. In most cases, the virtual size is equal to the physical size.

Prototype

```
int LCD_GetVXSizeEx(int Index);
int LCD_GetVYSizeEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

Virtual X/Y-size of the display.

LCD_GetXMag(), LCD_GetYMag()

Description

Returns the magnification factor in X- or Y-axis, respectively.

Prototype

```
int LCD_GetXMag(int Index);
int LCD_GetYMag(int Index);
```

Return value

Magnification factor in X- or Y-axis.

LCD_GetXMagEx(), LCD_GetYMagEx()

Description

Returns the magnification factor in X- or Y-axis, respectively.

Prototype

```
int LCD_GetXMagEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

Magnification factor in X- or Y-axis.

LCD_GetXSize(), LCD_GetYSize()

Description

Returns the physical X- or Y-size, respectively, of the LCD in pixels.

Prototypes

```
int LCD_GetXSize(void)
int LCD_GetYSize(void)
```

Return value

Physical X/Y-size of the display.

LCD_GetXSizeEx(), LCD_GetYSizeEx()

Description

Returns the physical X- or Y-size, respectively, of the LCD in pixels.

Prototype

```
int LCD_GetXSizeEx(int Index);
int LCD_GetYSizeEx(int Index);
```

Parameter	Meaning
Index	Layer index.

Return value

Physical X/Y-size of the display.

Chapter 16

Timing and Execution-Related Functions

Some widgets, as well as our demonstration code, require time-related functions. The other parts of the emWin graphic library do not require a time base. The demonstration code makes heavy use of the routine `GUI_Delay()`, which delays for a given period of time. A unit of time is referred to as a tick.

16.1 Timing and execution API

The table below lists the available timing- and execution-related routines in alphabetical order. Detailed descriptions of the routines follow.

Routine	Explanation
<code>GUI_Delay()</code>	Delay for a specified period of time.
<code>GUI_GetTime()</code>	Return the current system time.

GUI_Delay()

Description

Delays for a specified period of time.

Prototype

```
void GUI_Delay(int Period);
```

Parameter	Explanation
<code>Period</code>	Period in ticks until function should return.

Add. information

The time unit (tick) is usually milliseconds (depending on `GUI_X_` functions). `GUI_Delay()` only executes idle functions for the given period. If the window manager is used, the delay time is used for the updating of invalid windows (through execution of `WM_Exec()`). This function will call `GUI_X_Delay()`.

GUI_GetTime()

Description

Returns the current system time.

Prototype

```
int GUI_GetTime(void);
```

Return value

The current system time in ticks.

Add. information

This function will call `GUI_X_GetTime()`.

Chapter 17

Low-Level Configuration (LCD-Conf.h)

Before you can use emWin on your target system, you need to configure the software for your application. Configuring means modifying the configuration (header) files which usually reside in the (sub)directory `Config`. We try to keep the configuration as simple as possible, but there are some configuration macros (in the file `LCD-Conf.h`) which you must modify in order for the system to work properly. These include:

- LCD macros, defining the size of the display as well as optional features (such as mirroring, etc.)
- LCD controller macros, defining how to access the controller you are using.

17.1 Available configuration macros

The following table shows the available macros used for low-level configuration:

Type	Macro	Default	Explanation
General (required) configuration			
S	LCD_CONTROLLER	---	Select LCD controller.
N	LCD_BITSPERPIXEL	---	Specify bits per pixel.
S	LCD_FIXEDPALETTE	---	Specify fixed palette mode. Set to 0 for a user-defined color lookup table (then LCD_PHYSCOLORS must be defined).
N	LCD_XSIZE	---	Define horizontal resolution of LCD.
N	LCD_YSIZE	---	Define vertical resolution of LCD.
Initialisation of the controller			
F	LCD_INIT_CONTROLLER() (obsolete)	---	Initialization sequence for the LCD controller(s). Not applicable with all controllers.
Display orientation			
B	LCD_MIRROR_X	0	Activate to mirror X-axis.
B	LCD_MIRROR_Y	0	Activate to mirror Y-axis.
B	LCD_SWAP_XY	0	Activate to swap X- and Y-axes. If set to 0, SEG lines refer to columns and COM lines refer to rows.
Color configuration			
N	LCD_MAX_LOG_COLORS	256	Maximum number of logical colors that the driver can support in a bitmap. Please note that a value >256 makes no sense.
A	LCD_PHYSCOLORS	---	Defines the contents of the color lookup table. Only required if LCD_FIXEDPALETTE is set to 0.
B	LCD_PHYSCOLORS_IN_RAM	0	Only relevant if physical colors are defined. Puts physical colors in RAM, making them modifiable at run time
B	LCD_REVERSE	0	Activate to invert the display at compile time.
B	LCD_REVERS_LUT	0	Activate to initialize the lookup table with inverted colors at run time.
F	LCD_SET_LUT_ENTRY	---	Used to set a single lookup table or palette RAM entry.
B	LCD_SWAP_RB	0	Activate to swap the red and blue components.
Magnifying the LCD			
N	LCD_XMAG<n>	1	Horizontal magnification factor of LCD.
N	LCD_YMAG<n>	1	Vertical magnification factor of LCD.
Simple bus interface configuration			
F	LCD_READ_A0 (Result)	---	Read a byte from LCD controller with A-line low.
F	LCD_READ_A1 (Result)	---	Read a byte from LCD controller with A-line high.
F	LCD_WRITE_A0 (Byte)	---	Write a byte to LCD controller with A-line low.
F	LCD_WRITE_A1 (Byte)	---	Write a byte to LCD controller with A-line high.
F	LCD_WRITE_A1	---	Write multiple bytes to LCD controller with A-line high.
Full bus interface configuration			
F	LCD_READ_MEM (Index)	---	Read the contents of video memory of controller.
F	LCD_READ_REG (Index)	---	Read the contents of a configuration register of controller.
F	LCD_WRITE_MEM (Index, Data)	---	Write to video memory (display data RAM) of controller.

Type	Macro	Default	Explanation
F	LCD_WRITE_REG (Index, Data)	---	Write to a configuration register of controller.
S	LCD_BUSWIDTH	16	Select bus-width (8/16) of LCD controller/CPU interface.
F	LCD_ENABLE_REG_ACCESS	---	Switch the M/R signal to register access. Not applicable with all controllers.
F	LCD_ENABLE_MEM_ACCESS	---	Switch the M/R signal to memory access. Not applicable with all controllers.
B	LCD_SWAP_BYTE_ORDER	0	Activate to invert the endian mode (swap the high and low bytes) between CPU and LCD controller when using a 16-bit bus interface.
Virtual display			
N	LCD_VXSIZE	LCD_XSIZE	Horizontal resolution of virtual display. Not applicable with all drivers.
N	LCD_VYSIZE	LCD_YSIZE	Vertical resolution of virtual display. Not applicable with all drivers.
LCD controller configuration: common/segment lines			
N	LCD_FIRSTSEG<n>	0	LCD controller <n>: first segment line used.
N	LCD_FIRSTCOM<n>	0	LCD controller <n>: first common line used.
COM/SEG lookup tables			
A	LCD_LUT_COM	---	COM lookup table for controller.
A	LCD_LUT_SEG	---	SEG lookup table for controller.
Miscellaneous			
N	LCD_DIST_NUM_CONTROLLERS	1	Number of LCD controllers used.
B	LCD_CACHE	1	Deactivate to disable use of display data cache, which slows down the speed of the driver. Not applicable with all drivers.
B	LCD_USE_BITBLT	1	Deactivate to disable BitBLT engine. If set to 1, the driver will use all available hardware acceleration.
B	LCD_SUPPORT_CACHECONTROL	0	Activate to enable cache control functions of LCD_L0_ControlCache() driver API. Not applicable with all controllers.
N	LCD_TIMERINIT0	---	Timing value used by ISR for displaying pane 0 when using CPU as controller.
N	LCD_TIMERINIT1	---	Timing value used by ISR for displaying pane 1 when using CPU as controller.
F	LCD_ON	---	Function replacement macro which switches the LCD on.
F	LCD_OFF	---	Function replacement macro which switches the LCD off.

How to configure the LCD

We recommend the following procedure:

1. Make a copy of a configuration file of similar configuration. Several configuration samples for your particular LCD controller can be found in the folder `Sample\LCDConf\xxx`, where `xxx` is your LCD driver.
2. Configure the bus interface by defining the appropriate simple bus or full bus macros.
3. Define the size of your LCD (`LCD_XSIZE`, `LCD_YSIZE`).
4. Select the controller used in your system, as well as the appropriate bpp and the palette mode (`LCD_CONTROLLER`, `LCD_BITSPERPIXEL`, `LCD_FIXEDPALETTE`).
5. Configure which common/segment lines are used, if necessary.
6. Test the system.
7. Reverse X/Y if necessary (`LCD_REVERSE`); go back to step 6 in this case.
8. Mirror X/Y if necessary (`LCD_MIRROR_X`, `LCD_MIRROR_Y`); go back to step 6 in this case.
9. Check all the other configuration switches.
10. Erase unused sections of the configuration.

17.2 General (required) configuration

LCD_CONTROLLER

Description

Defines the LCD controller used.

Type

Selection switch

Add. information

The LCD controller used is designated by the appropriate number. Please refer to Chapter 15: "LCD Drivers" for more information about available options.

Example

Specifies an Epson SED1565 controller:

```
#define LCD_CONTROLLER 1565 /* Selects SED 1565 LCD-controller */
```

LCD_BITSPERPIXEL

Description

Specifies the number of bits per pixel.

Type

Numerical value

LCD_FIXEDPALETTE

Description

Specifies the fixed palette mode.

Type

Selection switch

Add. information

Set the value to 0 to use a color lookup table instead of a fixed palette mode. The macro `LCD_PHYSCOLORS` must then be defined.

LCD_XSIZE, LCD_YSIZE

Description

Define the horizontal and vertical resolution (respectively) of the display used.

Type

Numerical values

Add. information

The values are logical sizes; X-direction specifies the direction which is used as the X-direction by all routines of the LCD driver. Usually the X-size equals the number of segments.

17.3 Initialisation of the controller

LCD_INIT_CONTROLLER (obsolete)

Description

This macro is no longer required. The display driver calls the function `LCD_X_InitController()` for initializing the display controller in case of `LCD_INIT_CONTROLLER` is not defined. This is the recommended way because this is more flexible than using a macro within the driver which needs to be defined before compiling the driver. To keep compatibility to older versions the macro can be used. If it is defined the function `LCD_X_InitController()` is not required.

Type

Function replacement

Add. information

It is executed during the `LCD_L0_Init()` routines of the driver. Please consult the data sheet of your controller for information on how to initialize your hardware.

Example

The sample below has been written for and tested with an Epson SED1565 controller using an internal power regulator.

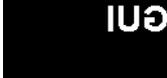
```
#define LCD_INIT_CONTROLLER() \
LCD_WRITE_A0(0xe2); /* Internal reset */ \
LCD_WRITE_A0(0xae); /* Display on/off: off */ \
LCD_WRITE_A0(0xac); /* Power save start: static indicator off */ \
LCD_WRITE_A0(0xa2); /* LCD bias select: 1/9 */ \
LCD_WRITE_A0(0xa0); /* ADC select: normal */ \
LCD_WRITE_A0(0xc0); /* Common output mode: normal */ \
LCD_WRITE_A0(0x27); /* V5 voltage regulator: medium */ \
LCD_WRITE_A0(0x81); /* Enter electronic volume mode */ \
LCD_WRITE_A0(0x13); /* Electronic volume: medium */ \
LCD_WRITE_A0(0xad); /* Power save end: static indicator on */ \
LCD_WRITE_A0(0x03); /* static indicator register set: on (constantly on) */ \
LCD_WRITE_A0(0x2F); /* Power control set: booster, regulator and follower off */ \
LCD_WRITE_A0(0x40); /* Display Start Line */ \
LCD_WRITE_A0(0xB0); /* Display Page Address 0 */ \
LCD_WRITE_A0(0x10); /* Display Column Address MSB */ \
LCD_WRITE_A0(0x00); /* Display Column Address LSB */ \
LCD_WRITE_A0(0xaf); /* Display on/off: on */ \
LCD_WRITE_A0(0xe3); /* NOP */ \
```

17.4 Display orientation

There are 8 possible display orientations; the display can be turned 0°, 90°, 180° or 270° and can also be viewed from top or from bottom. The default orientation is 0° and top view. These 4 * 2 = 8 different display orientations can also be expressed as a combination of 3 binary switches: X-mirror, Y-mirroring and X/Y swapping.

For this purpose, the binary configuration macros listed below can be used with each driver in any combination. If your display orientation is o.k. (Text on the display is readable; i.e. runs from left to right, is not upside-down and not mirrored), none of the configuration macros for display orientation are required. Otherwise, start by swapping X/Y if necessary and the mirror the X / Y axis as required or take a look at the table below which indicates which config switches have to be activated in which case. The orientation is handled as follows: Mirroring in X and Y first, then swapping (if selected).

Please note, that if more than one display orientation should be used at runtime, the multi display / multi layer feature is required.

Display	Orientation macros in LCDConf.h
	No orientation macro required
	Use #define LCD_MIRROR_X 1
	Use #define LCD_MIRROR_Y 1
	Use #define LCD_MIRROR_X 1 #define LCD_MIRROR_Y 1
	Use #define LCD_SWAP_XY 1
	Use #define LCD_SWAP_XY 1 #define LCD_MIRROR_X 1

Display	Orientation macros in LCDConf.h
	Use <pre>#define LCD_SWAP_XY 1 #define LCD_MIRROR_X 1 #define LCD_MIRROR_Y 1</pre>
	Use <pre>#define LCD_SWAP_XY 1 define LCD_MIRROR_Y 1</pre>

Driver optimizations

We can not optimize all drivers for all possible combinations of orientations and other config switches. In general, the default orientation is optimized. If you need to use a driver in an orientation which has not been optimized, please contact us.

LCD_MIRROR_X

Description

Inverts the X-direction (horizontal) of the display.

Type

Binary switch

0: inactive, X not mirrored (default)

1: active, X mirrored

Add. information

If activated: $X \rightarrow LCD_XSIZE-1-X$.

This macro, in combination with `LCD_MIRROR_Y` and `LCD_SWAP_XY`, can be used to support any orientation of the display. Before changing this configuration switch, make sure that `LCD_SWAP_XY` is set as required by your application.

LCD_MIRROR_Y

Description

Inverts the Y-direction (vertical) of the display.

Type

Binary switch

0: inactive, Y not mirrored (default)

1: active, Y mirrored

Add. information

If activated: $Y \rightarrow LCD_YSIZE-1-Y$.

This macro, in combination with `LCD_MIRROR_X` and `LCD_SWAP_XY`, can be used to support any orientation of the display. Before changing this configuration switch, make sure that `LCD_SWAP_XY` is set as required by your application.

LCD_SWAP_XY

Description

Swaps the horizontal and vertical directions (orientation) of the display.

Type

Binary switch

0: inactive, X-Y not swapped (default)

1: active, X-Y swapped

Add. information

If set to 0 (not swapped), SEG lines refer to columns and COM lines refer to rows.

If activated: X -> Y.

When changing this switch, you will also have to swap the X-Y settings for the resolution of the display (using `LCD_XSIZE` and `LCD_YSIZE`).

17.5 Color configuration

LCD_MAX_LOG_COLORS

Description

Defines the maximum number of colors supported by the driver in a bitmap. The maximum number of colors in a palette based bitmap is 256. So a value >256 makes no sense.

Type

Numerical value (default is 256)

Add. information

If you are using a 4-grayscale LCD, it is usually sufficient to set this value to 4. However, in this case remember not to try to display bitmaps with more than 4 colors.

LCD_PHYSCOLORS

Description

Defines the contents of the color lookup table, if one is used.

Type

Alias

Add. information

This macro is only required if `LCD_FIXEDPALETTE` is set to 0. Refer to the color section for more information.

LCD_PHYSCOLORS_IN_RAM

Description

Puts the contents of the physical color table in RAM if enabled.

Type

Binary switch

0: inactive (default)

1: active

LCD_REVERSE

Description

Inverts the display at compile time.

Type

Binary switch

0: inactive, not reversed (default)

1: active, reversed

LCD_SET_LUT_ENTRY**Description**

This macro can be used to set a single LUT entry. If defined the macro will be executed each time the GUI needs to set a LUT entry (typically during the initialisation).

Type

Function replacement

Prototype

```
#define LCD_SET_LUT_ENTRY(Pos, Color)
```

Parameter	Meaning
Pos	Zero based index of LUT entry to be set.
Color	RGB value of color to be set.

LCD_SWAP_RB**Description**

Swaps the red and blue color components.

Type

Binary switch

0: inactive, not swapped (default)

1: active, swapped

17.6 Simple bus interface configuration**17.6.1 Macros used by a simple bus interface**

The following macros are used for LCD controllers with simple bus interface.

LCD_READ_A0**Description**

Reads a byte from LCD controller with A0 (C/D) - line low.

Type

Function replacement

Prototype

```
#define LCD_READ_A0(Result)
```

Parameter	Meaning
Result	Result read. This is not a pointer, but a placeholder for the variable in which the value will be stored.

LCD_READ_A1**Description**

Reads a byte from LCD controller with A0 (C/D) - line high.

Type

Function replacement

Prototype

```
#define LCD_READ_A1(Result)
```

Parameter	Meaning
Result	Result read. This is not a pointer, but a placeholder for the variable in which the value will be stored.

LCD_WRITE_A0**Description**

Writes a byte to LCD controller with A0 (C/D) - line low.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A0(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITE_A1**Description**

Writes a byte to LCD controller with A0 (C/D) - line high.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A1(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITEM_A1**Description**

Writes several bytes to the LCD controller with A0 (C/D) - line high.

Type

Function replacement

Prototype

```
#define LCD_WRITEM_A1(paBytes, NumberOfBytes)
```

Parameter	Meaning
paBytes	Placeholder for the pointer to the first data byte.
NumberOfBytes	Number of data bytes to be written.

17.6.2 Example of memory mapped interface

The following example demonstrates how to access the LCD by a memory mapped interface:

```

void WriteM_A1(char *paBytes, int NummerOfBytes) {
    int i;
    for (i = 0; i < NummerOfBytes; i++) {
        (*(volatile char *)0xc0001) = *(paBytes + i);
    }
}

#define LCD_READ_A1(Result) Result = (*(volatile char *)0xc0000)
#define LCD_READ_A0(Result) Result = (*(volatile char *)0xc0001)
#define LCD_WRITE_A1(Byte) (*(volatile char *)0xc0000) = Byte
#define LCD_WRITE_A0(Byte) (*(volatile char *)0xc0001) = Byte

#define LCD_WRITEM_A1(paBytes, NummerOfBytes) WriteM_A1(paBytes, NummerOfBytes)

```

17.6.3 Sample routines for connection to I/O pins

Several examples can be found in the folder `Sample\LCD_X`:

- Port routines for 6800 interface
- Port routines for 8080 interface
- Simple port routines for a serial interface
- Port routines for a simple I2C bus interface

These samples can be used directly. All you need to do is to define the port access macros listed at the top of each example and to map them in your `LCDConf.h` in a similar manner to that shown below:

```

void LCD_X_Write00(char c);
void LCD_X_Write01(char c);
char LCD_X_Read00(void);
char LCD_X_Read01(void);
#define LCD_WRITE_A1(Byte) LCD_X_Write01(Byte)
#define LCD_WRITE_A0(Byte) LCD_X_Write00(Byte)
#define LCD_READ_A1(Result) Result = LCD_X_Read01()
#define LCD_READ_A0(Result) Result = LCD_X_Read00()

```

Note that not all LCD controllers handle the A0 or C/D bit in the same way. For example, a Toshiba controller requires that this bit be low when accessing data and an Epson SED1565 requires it to be high.

17.7 3 pin SPI configuration

17.7.1 Macros used by a 3 pin SPI interface

The following macros are used for LCD controllers with 4 pin SPI interface.

LCD_WRITE

Description

Writes a byte to the LCD controller.

Type

Function replacement

Prototype

```
#define LCD_WRITE(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITEM

Description

Writes several bytes to the LCD controller.

Type

Function replacement

Prototype

```
#define LCD_WRITEM(paBytes, NumberOfBytes)
```

Parameter	Meaning
paBytes	Placeholder for the pointer to the first data byte.
NumberOfBytes	Number of data bytes to be written.

17.7.2 Sample routines for connection to I/O pins

An example can be found in the folder `Sample\LCD_X`:

- `LCD_X_SERIAL.c`, port routines for a serial interface

This sample can be used directly. All you need to do is to define the port access macros listed at the top of the example and to map them in your `LCDConf.h` in a similar manner to that shown below:

```
void LCD_X_Write(char c);
void LCD_X_WriteM(char * pData, int NumBytes);
#define LCD_WRITE(Byte) LCD_X_Write(Byte)
#define LCD_WRITEM(data, NumBytes) LCD_X_WriteM(data, NumBytes)
```

17.8 4 pin SPI configuration

17.8.1 Macros used by a 4 pin SPI interface

The following macros are used for LCD controllers with 4 pin SPI interface.

LCD_WRITE_A0

Description

Writes a byte to LCD controller with A0 - line low.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A0(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITE_A1

Description

Writes a byte to LCD controller with A0 - line high.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A1(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITEM_A1

Description

Writes several bytes to the LCD controller with A0 - line high.

Type

Function replacement

Prototype

```
#define LCD_WRITEM_A1(paBytes, NumberOfBytes)
```

Parameter	Meaning
paBytes	Placeholder for the pointer to the first data byte.
NumberOfBytes	Number of data bytes to be written.

17.8.2 Sample routines for connection to I/O pins

An example can be found in the folder `Sample\LCD_X`:

- `LCD_X_SERIAL.c`, port routines for a serial interface

This sample can be used directly. All you need to do is to define the port access macros listed at the top of the example and to map them in your `LCDCConf.h` in a similar manner to that shown below:

```
void LCD_X_Write00(char c);
void LCD_X_Write01(char c);
void LCD_X_WriteM01(char * pData, int NumBytes);
#define LCD_WRITE_A0(Byte) LCD_X_Write00(Byte)
#define LCD_WRITE_A1(Byte) LCD_X_Write01(Byte)
#define LCD_WRITEM_A1(data, NumBytes) LCD_X_WriteM01(data, NumBytes)
```

17.9 I2C bus interface configuration

17.9.1 Macros used by a I2C bus interface

The following macros are used for LCD controllers with I2C bus interface.

LCD_READ_A0

Description

Reads a status byte from LCD controller.

Type

Function replacement

Prototype

```
#define LCD_READ_A0(Result)
```

Parameter	Meaning
<code>Result</code>	Result read. This is not a pointer, but a placeholder for the variable in which the value will be stored.

LCD_READ_A1

Description

Reads a data byte from LCD controller.

Type

Function replacement

Prototype

```
#define LCD_READ_A1(Result)
```

Parameter	Meaning
<code>Result</code>	Result read. This is not a pointer, but a placeholder for the variable in which the value will be stored.

LCD_WRITE_A0

Description

Writes a instruction byte to LCD controller.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A0(Byte)
```

Parameter	Meaning
<code>Byte</code>	Byte to write.

LCD_WRITE_A1

Description

Writes a data byte to LCD controller.

Type

Function replacement

Prototype

```
#define LCD_WRITE_A1(Byte)
```

Parameter	Meaning
Byte	Byte to write.

LCD_WRITEM_A1**Description**

Writes several data bytes to the LCD controller.

Type

Function replacement

Prototype

```
#define LCD_WRITEM_A1(paBytes, NumberOfBytes)
```

Parameter	Meaning
paBytes	Placeholder for the pointer to the first data byte.
NumberOfBytes	Number of data bytes to be written.

17.9.2 Sample routines for connection to I/O pins

An example can be found in the folder `Sample\LCD_X`:

- `LCD_X_I2CBUS.c`, port routines for a serial interface

This sample can be used directly. All you need to do is to define the port access macros listed at the top of the example and to map them in your `LCDConf.h` in a similar manner to that shown below:

```
void LCD_X_Write00(char c);
void LCD_X_Write01(char c);
char LCD_X_Read00(void);
char LCD_X_Read01(void);
#define LCD_WRITE_A1(Byte) LCD_X_Write01(Byte)
#define LCD_WRITE_A0(Byte) LCD_X_Write00(Byte)
#define LCD_READ_A1(Result) Result = LCD_X_Read01()
#define LCD_READ_A0(Result) Result = LCD_X_Read00()
```

17.10 Full bus interface configuration

17.10.1 Macros used by a full bus interface

The following macros are used for LCD controllers with a full bus interface.

LCD_READ_MEM

Description

Reads the video memory of the LCD controller.

Type

Function replacement

Prototype

```
#define LCD_READ_MEM(Index)
```

Parameter	Meaning
Index	Index of video memory of controller.

Add. information

This macro defines how to read the video memory of the LCD controller.

In order to configure this switch correctly, you need to know the base address of the video memory, the spacing and if 8/16- or 32-bit accesses are permitted. You should also know the correct syntax for your compiler because this kind of hardware access is not defined in ANSI "C" and is therefore different for different compilers.

LCD_READ_REG

Description

Reads the register of the LCD controller.

Type

Function replacement

Prototype

```
#define LCD_READ_REG(Index)
```

Parameter	Meaning
Index	Index of the register to read.

Add. information

This macro defines how to read the registers of the LCD controller. Usually, the registers are memory-mapped. In this case, the macro can normally be written as a single line.

In order to configure this switch correctly, you need to know the address the registers are mapped to, the spacing and if 8/16- or 32-bit accesses are permitted. You should also know the correct syntax for your compiler because this kind of hardware access is not defined in ANSI "C" and is therefore different for different compilers. However, the syntax shown below works with the majority of them.

Example

If the registers are mapped to a memory area starting at 0xc0000, the spacing is 2 and 16-bit accesses should be used; with most compilers the define should look as follows:

```
#define LCD_READ_REG(Index)          *((U16*)(0xc0000+(Off<<1)))
```

LCD_WRITE_MEM

Description

Writes data to the video memory of the LCD controller.

Type

Function replacement

Prototype

LCD_WRITE_MEM(Index, Data)

Parameter	Meaning
Index	Index of video memory of controller.
Data	Data to write to the register

Add. information

This macro defines how to write to the video memory of the LCD controller.

In order to configure this switch correctly, you need to know the base address of the rvideo memory, the spacing and if 8/16- or 32-bit accesses are permitted, as well as the correct syntax for your compiler.

With 8-bit accesses, a value of 1 indicates byte 1.

With 16-bit accesses, a value of 1 indicates word 1.

LCD_WRITE_REG

Description

Writes data to a specified register of the LCD controller

Type

Function replacement

Prototype

LCD_WRITE_REG(Index, Data)

Parameter	Meaning
Index	Index of the register to write to
Data	Data to write to the register

Add. information

This macro defines how to write to the registers of the LCD controller. If the registers are memory-mapped, the macro can normally be written as a single line.

In order to configure this switch correctly, you need to know the address the registers are mapped to, the spacing and if 8/16- or 32-bit accesses are permitted, as well as the correct syntax for your compiler.

With 8-bit accesses, a value of 1 indicates byte 1.

With 16-bit accesses, a value of 1 indicates word 1.

Example

If the registers are mapped to a memory area starting at 0xc0000, the spacing is 4 and 8-bit access should be used; with most compilers the define should look as follows:

```
#define LCD_WRITE_REG(Index,Data) *((U8volatile *) (0xc0000+(Off<<2)))=data
```

LCD_BUSWIDTH

Description

Defines bus-width of LCD controller/CPU interface (external display access).

Type

Selection switch
 8: 8-bit wide VRAM
 16: 16-bit wide VRAM (default)

Add. information

Since this completely depends on your hardware, you will have to substitute these macros. The Epson SED1352 distinguishes between memory and register access; memory is the video memory of the LCD controller and registers are the 15 configuration registers. The macros define how to access (read/write) VRAM and registers.

LCD_ENABLE_REG_ACCESS**Description**

Enables register access and sets the M/R signal to high.

Type

Function replacement

Prototype

```
#define LCD_ENABLE_REG_ACCESS() MR = 1
```

Add. Information

Only used for Epson SED1356 and SED1386 controllers.
 After using this macro, `LCD_ENABLE_MEM_ACCESS` must also be defined in order to switch back to memory access after accessing the registers.

LCD_ENABLE_MEM_ACCESS**Description**

Switches the M/R signal to memory access. It is executed after register access functions and sets the M/R signal to low.

Type

Function replacement

Prototype

```
#define LCD_ENABLE_MEM_ACCESS() MR = 0
```

Add. information

Only used for Epson SED1356 and SED1386 controllers.

LCD_SWAP_BYTE_ORDER**Description**

Inverts the endian mode (swaps the high and low bytes) between CPU and LCD controller when using a 16-bit bus interface.

Type

Binary switch
 0: inactive, endian modes not swapped (default)
 1: active, endian modes swapped

17.10.2 Configuration example

The example assumes the following:

Base address video memory	0x80000
Base address registers	0xc0000
Access to video RAM	16-bit

Access to register	16-bit
Distance between adjacent video memory locations	2 bytes
Distance between adjacent register locations	2 bytes

```
#define LCD_READ_REG(Index)          * ((U16*) (0xc0000 + (Off << 1)))  
#define LCD_WRITE_REG(Index, data) * ((U16*) (0xc0000 + (Off << 1))) = data  
#define LCD_READ_MEM(Index)        * ((U16*) (0x80000 + (Off << 1)))  
#define LCD_WRITE_MEM(Index, data) * ((U16*) (0x80000 + (Off << 1))) = data
```

17.11 Virtual display support

LCD_VXSIZE, LCD_VYSIZE

Description

Define the horizontal and vertical resolution (respectively) of the virtual display.

Type

Numerical values

Add. information

The values are logical sizes; X-direction specifies the direction which is used as X-direction by all routines of the LCD driver.

If a virtual display is not used, these values should be the same as the values for `LCD_XSIZE`, `LCD_YSIZE` (these are the default settings).

The virtual display feature requires hardware support and is not available with all drivers.

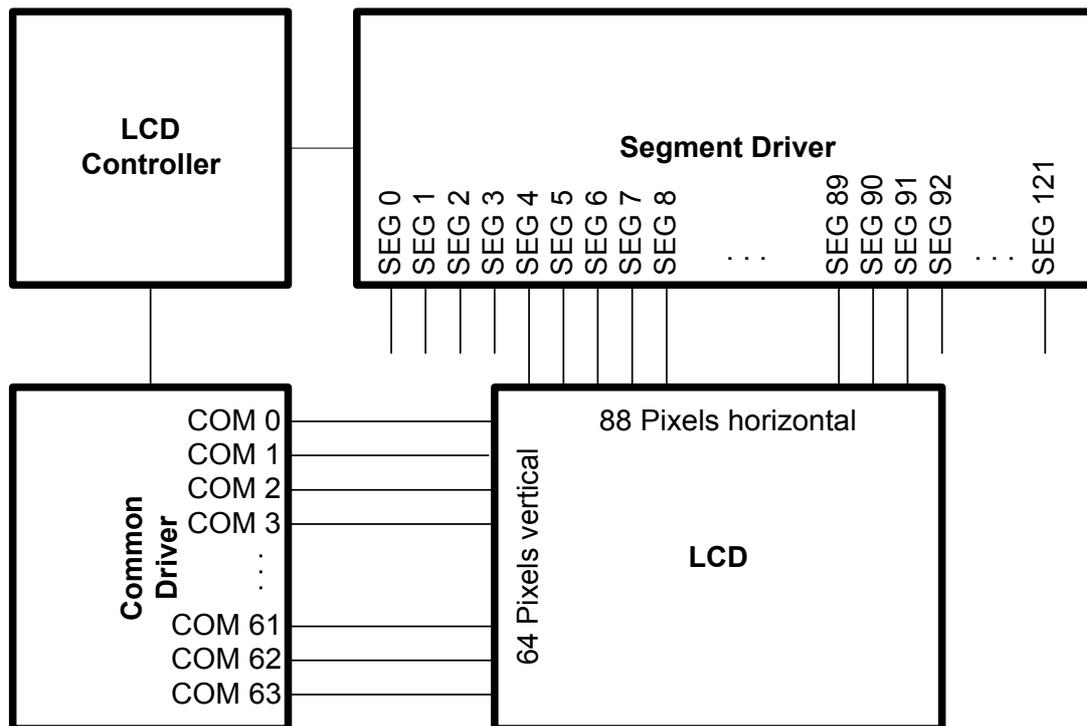
17.12 LCD controller configuration: COM/SEG lines

For most LCDs, the setup of common (COM) and segment (SEG) lines is straightforward and neither special settings for COM/SEG lines nor the configuration macros in this section are required. This section explains how the LCD controller(s) is physically connected to your display. The direction does not matter; it is only assumed that continuous COM and SEG lines are used. If the direction of SEGs or COMs is reversed, use `LCD_MIRROR_X/LCD_MIRROR_Y` to set them in the direction required by your application. If non-continuous COM/SEG lines have been used, you have to modify the driver (putting in a translation table will do) or -- even better -- go back to the hardware (LCD module) designer and ask him/her to start over. The following macros can be used to configure the COM/SEG lines:

Type	Macro	Explanation
N	<code>LCD_FIRSTSEG0</code>	First segment line used.
N	<code>LCD_FIRSTCOM0</code>	First common line used.

Example

The following block diagram shows a single display, controlled by a single display controller, using external COM and SEG drivers. All outputs of the common driver (COM0-COM63) are being used, but only some outputs of the segment driver (SEG4-SEG91). Note that for simplicity the video RAM is not shown in the diagram.



Configuration for the above example

```
#define LCD_FIRSTSEG0    4    /* Contr.0: first segment line used */
#define LCD_FIRSTCOM0   0    /* Contr.0: first com line used */
```

Please also note that the above configuration is identical if the COM or SEG lines are mirrored and even if the LCD is built-in sideways (90° turned, X-Y swapped). The same applies if the COM/SEG drivers are integrated into the LCD controller, as is the case for some controllers designed for small LCDs. A typical example for this type of controller would be the Epson SED15XX series.

17.13 Configuring multiple display controllers

Multiple display controllers (not multiple displays or multiple layers!) are supported for displays connected by a simple bus interface. If one display is controlled by more than one controller, emWin uses a distribution layer for accessing the different controllers. This is typically the case for some black/white (1bpp) displays only.

Please note that only one display with more than one display controller can be used simultaneously.

17.13.1 Macros used by the distribution layer

The following table shows the available macros for configuring the distribution layer:

Type	Macro	Explanation
N	LCD_DIST_NUM_CONTROLLERS	Number of available display controllers
A	LCD_DIST_DRIVER	Name of the display driver to be used
N	LCD_DIST_X0_<n>	LCD controller 1: leftmost (lowest) X-position.
N	LCD_DIST_Y0_<n>	LCD controller 1: topmost (lowest) Y-position.
N	LCD_DIST_X1_<n>	LCD controller 1: rightmost (lowest) X-position.
N	LCD_DIST_Y1_<n>	LCD controller 1: bottommost (lowest) Y-position.

17.13.2 Hardware access

If more than one controller is used for the display, there must be defined access macros for each of them individually, according to the hardware. The macros needed for the additional controllers are very similar to those for the first one. With a direct bus connection, usually only the addresses are different. When I/O pins are used, the sequence for the access is the same except for the CHIP-SELECT signal.

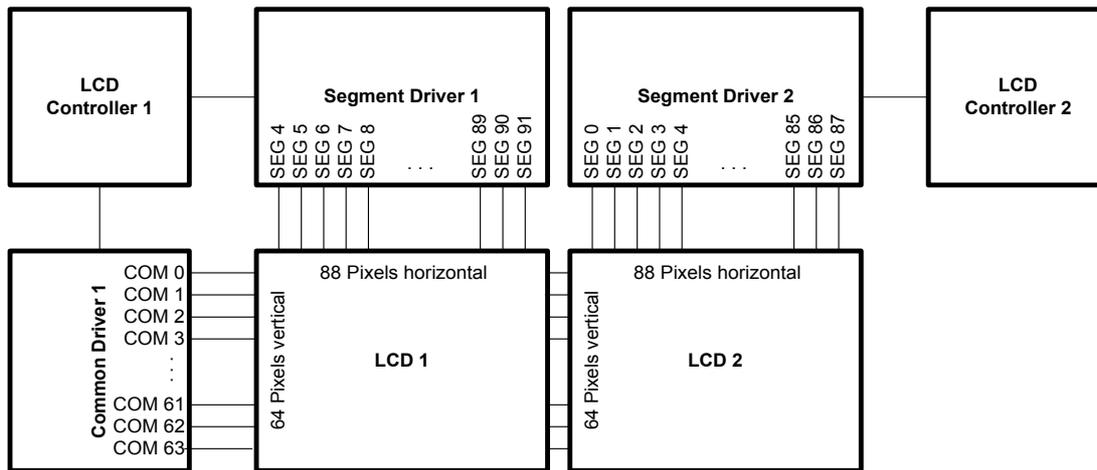
When using more than one controller, add a 'C' and the index of the controller as the postfix for controllers > 0.

Example

```
LCD_WRITE_A1(Byte) /* Write byte to controller 0 with RS line high */
LCD_WRITE_A0(Byte) /* Write byte to controller 0 with RS line low */
LCD_WRITE_A1C1(Byte) /* Write byte to controller 1 with RS line high */
LCD_WRITE_A0C1(Byte) /* Write byte to controller 1 with RS line low */
...
```

17.13.3 COM/SEG line configuration

The configuration switches are identical to the switches for the first controller (controller 0), except the index is 1, 2 or 3 instead of 0. The following diagram shows a hardware configuration using two LCD controllers. The COM lines are driven by the common driver connected to controller 1 and are directly connected to the second LCD. LCD 1 is connected to segment driver 1 using SEG lines 4 to 91. LCD 2 is driven by SEG 0 to SEG 87 of segment driver 2.



17.13.4 Configuration example

The following shows a configuration sample for the above case of 2 display controllers:

```
#define LCD_DIST_DRIVER "LCDPage1bpp.c" /* Real display driver to be used */

#define LCD_XSIZE          176 /* Display size in X */
#define LCD_YSIZE          64  /* Display size in Y */

#define LCD_DIST_NUM_CONTROLLERS 2 /* Number of available controllers */

#define LCD_DIST_X0_0      0 /* Leftmost position of area 0 */
#define LCD_DIST_Y0_0      0 /* Topmost position of area 0 */
#define LCD_DIST_X1_0     87 /* Rightmost position of area 0 */
#define LCD_DIST_Y1_0     63 /* Bottommost position of area 0 */

#define LCD_DIST_X0_1     88 /* Leftmost position of area 1 */
#define LCD_DIST_Y0_1      0 /* Topmost position of area 1 */
#define LCD_DIST_X1_1    175 /* Rightmost position of area 1 */
#define LCD_DIST_Y1_1     63 /* Bottommost position of area 1 */

#define LCD_FIRSTSEGO     4 /* Contr.0: first segment line used */
#define LCD_FIRSTCOM0     0 /* Contr.0: first com line used */
#define LCD_FIRSTSEG1     0 /* Contr.1: first segment line used */
#define LCD_FIRSTCOM1     0 /* Contr.1: first com line used */
```

17.14 COM/SEG lookup tables

When using "chip on glass" technology, it is sometimes very difficult to ensure that the COM and SEG outputs of the controller(s) are connected to the display in a linear fashion. In this case a COM/SEG lookup table may be required in order to inform the driver as to how the COM/SEG lines are connected.

LCD_LUT_COM

Description

Defines a COM lookup table for the controller.

Type

Alias

Example

Let us assume your display contains only 10 COM lines and their connecting order is 0, 1, 2, 6, 5, 4, 3, 7, 8, 9. To configure the LCD driver so that the COM lines are accessed in the correct order, the following macro should be added to your `LCD-Conf.h`:

```
#define LCD_LUT_COM 0, 1, 2, 6, 5, 4, 3, 7, 8, 9
```

If you need to modify the segment order, you should use the macro `LCD_LUT_SEG` in the same manner.

LCD_LUT_SEG

Description

Defines a SEG lookup table for the controller.

Type

Alias

17.15 Miscellaneous

LCD_DIST_NUM_CONTROLLERS

Description

Defines the number of LCD controllers used.

Type

Numerical value (default is 1)

LCD_CACHE

Description

Controls caching of video memory in CPU memory.

Type

Binary switch

0: disabled, no display data cache used

1: enabled, display data cache used (default)

Add. information

This switch is not supported by all LCD drivers.

Using a display data cache (which speeds up access) is recommended if access to the video memory is slow, which is usually the case with larger displays and simple bus interfaces (particularly if port-access or serial interfaces are used). Disabling the cache will slow down the speed of the driver.

LCD_USE_BITBLT

Description

Controls usage of hardware acceleration.

Type

Binary switch

0: disabled, BitBLT engine is not used

1: enabled, BitBLT engine is used (default)

Add. information

Disabling the BitBLT engine will instruct the driver not to use the available hardware acceleration.

LCD_SUPPORT_CACHECONTROL

Description

Switch support for the `LCD_L0_ControlCache()` function of the driver.

Type

Binary switch

0: disabled, `LCD_L0_ControlCache()` may not be used (default)

1: enabled, `LCD_L0_ControlCache()` may be used

Add. information

The API function `LCD_L0_ControlCache()` permits locking, unlocking, or flushing of the cache. Please note that this feature is intended only for some LCD controllers with simple bus interface, for which it is important to access the controller as little as possible in order to maximize speed. For other controllers, this switch has no effect.

LCD_TIMERINIT0

Description

Timing value used by an interrupt service routine for displaying pane 0 of a pixel.

Type

Numerical value

Add. information

This macro is only relevant when no LCD controller is used, since it is then the job of the CPU to update the display in an interrupt service routine.

LCD_TIMERINIT1

Description

Timing value used by an interrupt service routine for displaying pane 1 of a pixel.

Type

Numerical value

Add. information

This macro is only relevant when no LCD controller is used, since it is then the job of the CPU to update the display in an interrupt service routine.

LCD_ON

Description

Switches the LCD on.

Type

Function replacement

LCD_OFF

Description

Switches the LCD off.

Type

Function replacement

Chapter 18

High-Level Configuration (GUIConf.h)

High-level configuration is relatively simple. In the beginning, you can normally use the existing configuration files (for example, those used in the simulation). Only if there is a need to fine-tune the system, or to minimize memory consumption, does the high-level configuration file `GUIConf.h` need to be changed. This file is usually located in the `Config` subdirectory of your project's root directory. Use the file `GUIConf.h` for any high-level configuration.

The second thing to do when using emWin on your hardware is to change the hardware-dependent functions, located in the file `Sample\GUI_X\GUI_X.c`.

18.1 General notes

The configuration options explained in this chapter are the available options for the general library.

18.2 How to configure the GUI

We recommend the following procedure:

1. Make a copy of the original configuration file.
2. Review all configuration switches.
3. Erase unused sections of the configuration.

18.2.1 Sample configuration

The following is a short sample GUI configuration file:

```
#define GUI_WINSUPPORT          (1) /* Use window manager if true (1) */
#define GUI_SUPPORT_TOUCH      (1) /* Support a touch screen */
#define GUI_ALLOC_SIZE        5000 /* Size of dynamic memory */
#define GUI_DEFAULT_FONT      &GUI_Font6x8 /* This font is used as default */
```

18.3 Available GUI configuration macros

The following table shows the available macros used for high-level configuration of emWin

Type	Macro	Default	Explanation
N	GUI_ALLOC_SIZE (obsolete)	0	Defines the size (number of bytes available) for optional dynamic memory. Dynamic memory is required for windows, memory devices, image decompression, antialiasing and alpha blending.
S	GUI_DEBUG_LEVEL	1 (target) 4 (simulation)	Defines the debug level, which determines how many checks (assertions) are performed by emWin and if debug errors, warnings and messages are output. Higher debug levels generate bigger code.
N	GUI_DEFAULT_BKCOLOR	GUI_BLACK	Define the default background color.
N	GUI_DEFAULT_COLOR	GUI_WHITE	Define the default foreground color.
S	GUI_DEFAULT_FONT	&GUI_Font6x8	Define which font is used as default after GUI_Init(). If you do not use the default font, it makes sense to change to a different default, as the default font is referenced by the code and will therefore always be linked.
N	GUI_MAXBLOCKS (obsolete)	---	Defines the number of available memory blocks for the memory management of emWin. The maximum number of blocks depends per default on GUI_ALLOC_SIZE.
N	GUI_MAXTASK	4	Define the maximum number of tasks from which emWin is called to access the display when multitasking support is enabled (see Chapter 11: "Execution Model: Single Task/Multitask").
F	GUI_MEMCPY	---	This macro allows replacement of the memcpy function.
F	GUI_MEMSET	---	Replacement of the memset function of the GUI.

Type	Macro	Default	Explanation
B	GUI_OS	0	Activate to enable multitasking support with multiple tasks calling emWin (see Chapter 11: "Execution Model: Single Task/Multitask").
B	GUI_SUPPORT_BIDI	0	Activates the bidirectional language support required for drawing Arabic and Hebrew text.
B	GUI_SUPPORT_LARGE_BITMAPS	0	If a system with a 16 bit CPU (sizeof(int) == 2) should display bitmaps >64Kb this configuration macro should be set to 1.
B	GUI_SUPPORT_TOUCH	0	Enables optional touch-screen support. 1 enables the default touch screen support. 2 enables compatibility mode to older versions without runtime configuration.
B	GUI_SUPPORT_UNICODE	1	Enables support for Unicode characters embedded in 8-bit strings. Please note: Unicode characters may always be displayed, as character codes are always treated as 16-bit.
B	GUI_TRIAL_VERSION	0	Marks the compiler output as evaluation version.

18.3.1 GUI_MEMCPY

This macro allows replacement of the memcpy function of the GUI. On a lot of systems, memcpy takes up a considerable amount of time because it is not optimized by the compiler manufacturer. emWin contains an alternative memcpy routine, which has been optimized for 32 bit CPUs. On a lot of systems this routine should generate faster code than the default memcpy routine. However, this is still a generic "C"-routine, which in a lot of systems can be replaced by faster code, typically using either a different "C" routine, which is better optimized for the particular CPU or by writing a routine in Assembly language.

To use the optimized emWin routine add the following define to the file GUIConf.h:

```
#define GUI_MEMCPY(pSrc, pDest, NumBytes) GUI__memcpy(pSrc, pDest, NumBytes)
```

18.3.2 GUI_MEMSET

This macro allows replacement of the memset function of the GUI. On a lot of systems, memset takes up a considerable amount of time because it is not optimized by the compiler manufacturer. We have tried to address this by using our own memset() Routine GUI__memset. However, this is still a generic "C"-routine, which in a lot of systems can be replaced by faster code, typically using either a different "C" routine, which is better optimized for the particular CPU, by writing a routine in Assembly language or using the DMA.

If you want to use your own memset replacement routine, add the define to the GUIConf.h file.

18.3.3 GUI_TRIAL_VERSION

This macro can be used to mark the compiler output as an evaluation build. It should be defined if the software is given to a third party for evaluation purpose (typically with evaluation boards).

Note that a special license is required to do this; the most common licenses do not permit redistribution of emWin in source or object code (relinkable) form. Please contact sales@segger.com if you would like to do this.

If `GUI_TRIAL_VERSION` is defined, the following message is shown when calling `GUI_Init()`:

```

This software
contains an eval-
build of emWin.

A license is
required to use
it in a product.

www.segger.com

```

This message is always shown in the upper left corner of the display and is normally visible for 1 second. The timing is implemented by a call `GUI_X_Delay(1000)`. The functionality of emWin is in no way limited if this switch is active.

Sample

```
#define GUI_TRIAL_VERSION 1
```

18.4 Runtime configuration

The illustration shows the initialization process of emWin. From the application only `GUI_Init()` needs to be called. In the following the different configuration routines which are called within the initialization process are explained.

```

GUI_Init()
├── GUI_X_Config()           /* If GUI_ALLOC_SIZE not defined */
│   ├── GUI_ALLOC_AssignMemory() /* Required */
│   └── GUI_ALLOC_SetAvBlockSize() /* Required */
├── GUI_X_Init()
│   ├── GUI_TOUCH_Calibrate() /* Optional */
│   └── GUI_TOUCH_SetOrientation() /* Optional */

```

GUI_X_Config()

It is called at the very first beginning of the initialization process to make sure memory is assigned to emWin. Within this routine `GUI_ALLOC_AssignMemory()` and `GUI_ALLOC_SetAvBlockSize()` must be used to assign a memory block to emWin and set the average memory block size. The functions are explained later in this chapter.

GUI_X_Init()

This function can be used for initializing an optional touch screen using the functions `GUI_TOUCH_Calibrate()` and `GUI_TOUCH_SetOrientation()`. It is called immediately after `GUI_X_Config()`.

LCD_X_InitController()

After the general initialization the callback function `LCD_X_InitController()` is called during the initialization process for putting the display into operation. This function is part of the application and has to make sure that the display controller is initialized right.

18.4.1 Memory requirements

The following gives a rough overview of the memory requirement. The dynamic memory is used by the window manager/widget library (optional), memory devices (optional), antialiasing (optional) and alpha blending. The following gives you an overview of the memory requirements of these modules.

Memory requirement of the window manager

If the window manager is used, approximately 50 bytes are used per application defined window and approximately 100 bytes per widget. Typical applications using the window manager/widgets requires app. 2500 bytes.

Memory requirement of the memory devices

Depending on the color depth used one pixel of a memory device requires 1 or 2 bytes of RAM. Configurations with a color depth between 1 and 8bpp (`LCD_BITSPERPIXEL` between 1 and 8) uses 1 byte of RAM. A color depth >8 and ≤16bpp uses 2 bytes of RAM. If using the memory device module to prevent the

display from flickering when using drawing operations for overlapping items a minimum of 4000 bytes is recommended. If enough RAM is available the allocated size should be sufficient to store the pixels of the largest window plus the memory requirement of the window manager.

Runtime configuration of available memory

The available memory to be used by emWin can also be configured at runtime during initialization. To use runtime configuration the value defined by `GUI_ALLOC_SIZE` needs to be 0. For further details about this please refer to the function `GUI_ALLOC_AssignMemory()`.

Examples

- If the current configuration is 16bpp and the largest window is 320x240 pixels $320 \times 240 \times 2 = 153600$ bytes are useful for the memory device module.
- If the current configuration is 16bpp and the largest window is 320x240 pixels $320 \times 240 = 76800$ bytes are useful for the memory device module.

18.4.2 Available GUI configuration routines

The following table shows the available routines used for high-level configuration of emWin:

Routine	Explanation
<code>GUI_ALLOC_AssignMemory()</code>	Assigns a memory block for the memory management system.
<code>GUI_ALLOC_SetAvBlockSize()</code>	Sets the average size of the memory blocks. The bigger the block size, the less number of memory blocks are available.

GUI_ALLOC_AssignMemory()

Description

The function assigns the one and only memory block to emWin which is used by the internal memory management system. This function should be called typically from `GUI_X_Config()`.

Prototype

```
void GUI_ALLOC_AssignMemory(void * p, U32 NumBytes);
```

Parameter	Meaning
<code>p</code>	Pointer to the memory block which should be used by emWin.
<code>NumBytes</code>	Size of the memory block in bytes.

Add. information

It is required to set the value of `GUI_ALLOC_SIZE` defined in `GUIConf.h` to 0 to enable runtime memory configuration. Please note that not the complete memory block can be used by the application, because a small overhead of the memory is used by the management system itself.

GUI_ALLOC_SetAvBlockSize()

Description

Sets the average block size of the memory blocks allocated by the memory management system. The block size affects the number of maximum available blocks. If for example an application uses some listviews with a large number of entries it makes sense to set the average block size to a small value. On the other hand if an application uses the memory management primarily for a few memory devices or image decompression the average size should be set to a bigger value. The recommended range is between 32 and 1024. The value depends on the application.

This function should be called typically from `GUI_X_Config()`.

Prototype

```
void GUI_ALLOC_SetAvBlockSize(U32 BlockSize);
```

Parameter	Meaning
BlockSize	Average block

Add. information

The average block size is used to calculate the maximum number of available memory blocks:

Max. # of blocks = Size of memory in bytes / (BlockSize + sizeof(BLOCK_STRUCT))

BlockStruct means an internal structure whose size depends on GUI_DEBUG_LEVEL. If it is >0 the size will be 12 bytes, otherwise 8 bytes. Please note that the structure size also depends on the used compiler.

18.5 Runtime configuration

The illustration shows the initialization process of emWin. From the application only GUI_Init() needs to be called. In the following the different configuration routines which are called within the initialization process are explained.

GUI_X_Config()

It is called at the very first beginning of the initialization process to make sure memory is assigned to emWin. Within this routine GUI_ALLOC_AssignMemory() and GUI_ALLOC_SetAvBlockSize() must be used to assign a memory block to emWin and set the average memory block size.

GUI_X_Init()

This function can be used for initializing a touch screen using the functions GUI_TOUCH_Calibrate() and GUI_TOUCH_SetOrientation(). It is called immediately after GUI_X_Config().

LCD_X_InitController()

Typically the initialization macro LCD_INIT_CONTROLLER defined in LCDConf.h contains a function call to LCD_X_InitController() which needs to be part of the application. This routine should make sure that the display controller registers are initialized right. If a runtime configurable display driver is used this routine should also make sure that the display driver is configured right.

```

GUI_Init()
├── GUI_X_Config()           /* If GUI_ALLOC_SIZE not defined */
│   ├── GUI_ALLOC_AssignMemory() /* Required */
│   └── GUI_ALLOC_SetAvBlockSize() /* Required */
└── GUI_X_Init()
    ├── GUI_TOUCH_Calibrate() /* Optional */
    └── GUI_TOUCH_SetOrientation() /* Optional */

```

18.6 GUI_X routine reference

When using emWin on the target hardware, there are several hardware-dependent functions which must exist. When using the simulation, the library already contains them. A sample file can be found under Sample\GUI_X\GUI_X.c. The following table

lists the available hardware-dependent functions in alphabetical order within their respective categories. Detailed description of the routines can be found in the sections that follow.

Routine	Explanation
Init routines	
GUI_X_Config()	Called from <code>GUI_Init()</code> if runtime configuration is required.
GUI_X_Init()	Called from <code>GUI_Init()</code> ; can be used to initialize hardware.
Timing routines	
GUI_X_Delay()	Return after a given period.
GUI_X_ExecIdle()	Called only from non-blocking functions of window manager.
GUI_X_GetTime()	Return the system time in milliseconds.
Kernel interface routines	
GUI_X_InitOS()	Initialize the kernel interface module (create a resource semaphore/mutex).
GUI_X_GetTaskId()	Return a unique, 32-bit identifier for the current task/thread.
GUI_X_Lock()	Lock the GUI (block resource semaphore/mutex).
GUI_X_Unlock()	Unlock the GUI (unblock resource semaphore/mutex).
Debugging	
GUI_X_Log()	Return debug information; required if logging is enabled.

18.6.1 Init routines

GUI_X_Config()

Description

Called from `GUI_Init()` for runtime configuration. Typically used for memory management initialization described above in this chapter. First `GUI_X_Config()` is called and then `GUI_X_Init()`.

Prototype

```
void GUI_X_Config(void);
```

GUI_X_Init()

Description

Called from `GUI_Init()`; can be used to initialize hardware.

Prototype

```
void GUI_X_Init(void);
```

18.6.2 Timing routines

GUI_X_Delay()

Description

Returns after a specified time period in milliseconds.

Prototype

```
void GUI_X_Delay(int Period)
```

Parameter	Meaning
Period	Period in milliseconds.

GUI_X_ExecIdle()

Description

Called only from non-blocking functions of the window manager.

Prototype

```
void GUI_X_ExecIdle(void);
```

Add. information

Called when there are no longer any messages which require processing. In this case the GUI is up to date.

GUI_X_GetTime()

Description

Used by `GUI_GetTime` to return the current system time in milliseconds.

Prototype

```
int GUI_X_GetTime(void)
```

Return value

The current system time in milliseconds, of type integer.

18.6.3 Kernel interface routines

Detailed descriptions for these routines may be found in Chapter 11: "Execution Model: Single Task/Multitask".

18.7 Debugging

GUI_X_ErrorOut(), GUI_X_Warn(), GUI_X_Log()

Description

These routines are called by emWin with debug information in higher debug levels in case a problem (Error) or potential problem is discovered. The routines can be blank; they are not required for the functionality of emWin. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

Fatal errors are output using `GUI_X_ErrorOut()` if `(GUI_DEBUG_LEVEL >= 3)`

Warnings are output using `GUI_X_Warn()` if `(GUI_DEBUG_LEVEL >= 4)`

Messages are output using `GUI_X_Log()` if `(GUI_DEBUG_LEVEL >= 5)`

Prototypes

```
void GUI_X_ErrorOut(const char * s);
```

```
void GUI_X_Warn(const char * s);
```

```
void GUI_X_Log(const char * s);
```

Parameter	Meaning
<code>s</code>	Pointer to the string to be sent.

Add. information

This routine is called by emWin to transmit error messages or warnings, and is required if logging is enabled. The GUI calls this function depending on the configuration macro `GUI_DEBUG_LEVEL`. The following table lists the permitted values for `GUI_DEBUG_LEVEL`:

Value	Symbolic name	Explanation
0	<code>GUI_DEBUG_LEVEL_NOCHECK</code>	No run-time checks are performed.
1	<code>GUI_DEBUG_LEVEL_CHECK_PARA</code>	Parameter checks are performed to avoid crashes. (Default for target system)
2	<code>GUI_DEBUG_LEVEL_CHECK_ALL</code>	Parameter checks and consistency checks are performed.
3	<code>GUI_DEBUG_LEVEL_LOG_ERRORS</code>	Errors are recorded.
4	<code>GUI_DEBUG_LEVEL_LOG_WARNINGS</code>	Errors and warnings are recorded. (Default for PC-simulation)
5	<code>GUI_DEBUG_LEVEL_LOG_ALL</code>	Errors, warnings and messages are recorded.

18.8 Dynamic memory

emWin contains its own memory management system. But it is also possible to use your own memory management system. The following table shows the available macros used for dynamic memory configuration of emWin:

Type	Macro	Explanation
F	<code>GUI_ALLOC_ALLOC(Size)</code>	Used to allocate a memory block, returns a memor handle.
F	<code>GUI_ALLOC_FREE(pMem)</code>	Used to release a memory block.
F	<code>GUI_ALLOC_GETMAXSIZE()</code>	Returns the maximum number of bytes of available memory.
F	<code>GUI_ALLOC_H2P(hMem)</code>	Converts a memory handle to a memory pointer.
A	<code>GUI_HMEM</code>	Type of a memory handle.

GUI_ALLOC_ALLOC

Description

Allocates a memory block and returns a handle to it.

Type

Function replacement.

Prototype

```
#define GUI_ALLOC_ALLOC(Size)
```

Parameter	Meaning
<code>Size</code>	Size of required memory block

Example

```
#define GUI_ALLOC_ALLOC(Size) malloc(Size)
```

GUI_ALLOC_FREE

Description

Releases a memory block.

Type

Function replacement.

Prototype

```
#define GUI_ALLOC_FREE(pMem)
```

Parameter	Meaning
pMem	Pointer to memory block to be released.

Example

```
#define GUI_ALLOC_FREE(pMem) free(pMem)
```

GUI_ALLOC_GETMAXSIZE**Description**

Returns the maximum number of bytes of available memory.

Type

Function replacement.

Prototype

```
#define GUI_ALLOC_GETMAXSIZE()
```

Example

```
#define GUI_ALLOC_GETMAXSIZE() 10000
```

GUI_ALLOC_H2P**Description**

Converts a memory handle to a memory pointer.

Type

Function replacement.

Prototype

```
#define GUI_ALLOC_H2P(hMem)
```

Parameter	Meaning
hMem	Memory handle to be converted to a memory pointer.

Example

```
#define GUI_ALLOC_H2P(hMem) hMem
```

GUI_HMEM**Description**

Defines the type of a memory handle.

Type

Text replacement.

Example

```
#define GUI_HMEM void *
```

Example

The following sample is an excerpt from the GUIConf.h which uses the standard dynamic memory system:

```
#include <malloc.h>
#include <memory.h>

#define GUI_HMEM void *
#define GUI_ALLOC_ALLOC(Size) malloc(Size)
```

```
#define GUI_ALLOC_FREE(pMem)      free(pMem)
#define GUI_ALLOC_H2P(hMem)      hMem
#define GUI_ALLOC_GETMAXSIZE()  10000
```

18.9 Special considerations for certain Compilers/ CPUs

18.9.1 AVR with IAR-Compiler

When using an Atmel AVR CPU and a IAR compiler and you need to put const data in flash ROM you need to add 2 additional configuration macros to GUIConf.h:

Type	Macro	Default	Explanation
A	GUI_UNI_PTR	---	Define "universal pointer".
A	GUI_CONST_STORAGE	const	Define const storage.

GUI_UNI_PTR

Description

Defines a "universal pointer" which can point to RAM and flash ROM. On some systems it can be necessary since a default pointer can access RAM only, not the built-in Flash.

Type

Alias.

Example

```
#define GUI_UNI_PTR __generic
```

GUI_CONST_STORAGE

Description

Defines the const storage. On some systems it can be necessary since otherwise constants are copied into RAM.

Type

Alias.

Example

```
#define GUI_CONST_STORAGE __flash const
```

18.9.2 8051 Keil compiler and other 8-bit CPU compilers

Keils 8051 Compiler (tested in V5 & V6) has limitation as far as function pointers are concerned. The compiler is limited in respect to the number of parameters which can be passed to a function called thru a function pointer (indirect function call). Some other 8-bit compilers (for 6502 type architectures, such as ST7, but possibly also other chips) may also have a similar limitation. The config switch below allows to circumvent most of these limitations by avoiding function calls with multiple parameters.

Type	Macro	Default	Explanation
A	GUI_COMPILER_SUPPORTS_FP	1	Set to 0 if the compiler does not support complex function calls via function pointers.

GUI_COMPILER_SUPPORTS_FP

Description

Used to enable/disable the use of complex function pointers.

Type

Alias.

Example

```
#define GUI_COMPILER_SUPPORTS_FP 0 /* Disable the use of complex function pointers */
```

Add. information

Disabling this config switch will make it possible to use the software even with very limited "C"-compilers for small chips. However, this comes at a price:

The available functionality is limited as well. The following limitations apply in this case:

- Text rotation can not be used
- Compressed bitmaps can not be used
- Higher level software, such as memory devices, window manager & VNC server can not be used
- Antialiasing can not be used
- Some other (smaller) restrictions may apply.

Chapter 19

Performance and Resource Usage

High performance combined with low resource usage has always been a major design consideration. emWin runs on 8/16/32-bit CPUs. Depending on which modules are being used, even single-chip systems with less than 64kb ROM and 2kb RAM can be supported by emWin. The actual performance and resource usage depends on many factors (CPU, compiler, memory model, optimization, configuration, interface to LCD controller, etc.). This chapter contains benchmarks and information about resource usage in typical systems which can be used to obtain sufficient estimates for most target systems.

19.1 Memory requirements

The operation area of emWin varies widely, depending primarily on the application and features used. In the following sections, memory requirements of different modules are listed as well as memory requirement of sample applications. The memory requirements of the GUI components have been measured on a system as follows:

ARM7, IAR Embedded workbench V4.42A, Thumb mode, Size optimization

19.1.1 Memory requirements of the GUI components

The following table shows the memory requirements of the main components of emWin. These values depend a lot on the compiler options, the compiler version and the used CPU. Please note that the listed values are the requirements of the basic functions of each module and that there are several additional functions available which have not been considered in the table:

Component	ROM	RAM	Explanation
Driver	+ 2 - 8 kB	20	The memory requirements of the driver depend on the configured driver and if a data cache is used or not. With a data cache the driver requires more RAM. For details please refer to the driver documentation.
Core	5.2 kB	80 Byte	Memory requirements of a typical 'Hello world' application without using additional software items.
Core / Fonts	(see explanation)	-	For details about the ROM requirements of the standard fonts shipped with emWin please refer to the font chapter.

*1. The listed memory requirements of the widgets contain the basic routines required for creating and drawing the widget. Depending of the specific widget there are several additional functions available which are not listed in the table.

19.1.2 Stack requirements

The basic stack requirement is app. 600 bytes.

Chapter 20

Support

This chapter should help if any problem occurs. This could be a problem with the tool chain, with the hardware, the use of the GUI functions or with the performance and it describes how to contact the emWin support.

20.1 Problems with tool chain (compiler, linker)

The following shows some of the problems that can occur with the use of your tool chain. The chapter shows what to do in case of a problem and how to contact the emWin support.

20.1.1 Compiler crash

You ran into a tool chain (compiler) problem, not a problem of emWin. If one of the tools of your tool chain crashes, you should contact your compiler support:

```
"Tool internal error, please contact support"
```

20.1.2 Compiler warnings

The code of emWin has been tested on different target systems and with different compilers. We spend a lot of time on improving the quality of the code and we do our best to avoid compiler warnings. But the sensitivity of each compiler regarding warnings is different. So we can not avoid compiler warnings for unknown tools.

Warnings you should not see

This kind of warnings should not occur:

```
"Function has no prototype"  
"Incompatible pointer types"  
"Variable used without having been initialized"  
"Illegal redefinition of macro"
```

Warnings you may see

Warnings such as the ones below should be ignored:

```
"Integer conversion, may loose significant bits"  
"Statement not reached"  
"Meaningless statements were deleted during op-timization"  
"Condition is always true/false"  
"Unreachable code"
```

Most compilers offers a way to supress selected warnings.

Warning "Parameter not used"

Depending of the used configuration sometimes not all of the parameters of the functions are used. To avoid compiler warnings regarding this problem you can define the macro `GUI_USE_PARA` in the file `GUIConf.h` like the following sample:

```
#define GUI_USE_PARA(para) para=para;
```

emWin uses this macro wherever necessary to avoid this type of warning.

20.1.3 Linker problems

Undefined externals

If your linker shows the error message "Undefined external symbols..." please check if the following files have been included to the project or library:

- All source files shipped with emWin
- In case of a simple bus interface: One of the hardware routines located in the folder `Sample\LCD_X?` For details about this please take a look to the chapter 'Low-Level Configuration'.
- One of the files located in the folder `Sample\GUI_X?` For details about this please take a look to the chapter 'High-Level Configuration'.

Executable to large

Some linkers are not able to link only the modules/functions referenced by the project. This results is an executable with a lot of unused code. In this case the use of a library would be very helpful. For details about how to build an emWin library please take a look at the chapter 'Getting started'.

20.2 Problems with hardware/driver

If your tools are working fine but your display does not work may one of the following helps to find the problem.

Stack size to low?

Please make sure there have been configured enough stack. We can not estimate exactly how much stack will be used by your configuraton and with your compiler. If you wand to

Initialisation of the display wrong?

If the `LCD_INIT_CONTROLLER` macro is needed by your driver please check, if this macro has been adapted to your needs.

Display interface configured wrong?

When starting to work with emWin and the display does not show something you should use an oscilloscope to measure the pins connected with the display/controller. If there is a problem please check the following:

- If using a simple bus interface: Probably the hardware routines have not been configured correctly. If possible use an emulator and step through these routines.
- If using a full bus interface: Probably the register/memory access have not been configured correctly.

20.3 Problems with API functions

If your tool chain and your hardware works fine but the API functions do not function as documented, please make a small sample as described later under 'Contacting Support'. This allows us to easily reproduce the problem and solve it quickly.

20.4 Problems with the performance

If there is any performance problem with emWin it should be determined, which part of the software causes the problem.

Does the driver causes the problem?

To determine the cause of the problem the first step should be writing a small test routine which executes some testcode and measures the time used to execute this code. Starting point should be the file `ProblemReport.c` described above. To measure the time used by the real hardware driver the shipment of emWin contains the driver `LCDNull.c`. This driver can be used if no output to the hardware should be done. To activate the driver the `LCD_CONTROLLER` macro in `LCDConf.h` as follows:

```
#define LCD_CONTROLLER -2
```

The difference between the used time by the real driver and the `LCDNull` driver shows the execution time spent in the real hardware driver.

Driver not optimized?

If there is a significant difference between the use of the real driver and the `LCDNull` driver the cause of the problem could be a not optimized driver mode. If using one of the following macros: `LCD_MIRROR_X`, `LCD_MIRROR_Y`, `LCD_SWAP_XY` or `LCD_CACHE` the driver may not be optimized for the configured mode. In this case please contact our support, we should be able to optimize the code.

Slow display controller?

Also please take a look to the chapter 'Display drivers'. If using a slow display controller like the Epson SED1335 this chapter may answer the question, why the driver works slow.

20.5 Contacting support

If you need to contact the emWin support, please send the following information to the support:

- A detailed description of the problem may written as comment in the sample code.
- The configuration file `GUIConf.h`.
- The configuration file `LCDConf.h`.
- A sample source file which can be compiled in the simulation without any additional files as described in the following.
- If there are any problems with the tool chain please also send the error message of the compiler/linker.
- If there are any problems with the hardware/driver and a simple bus interface is used please also send the hardware routines including the configuration.

Problem report

The following file can be used as a starting point when creating a problem report. Please also fill in the CPU, the used tool chain and the problem description. It can be found under `Sample\Tutorial\ProblemReport.c`:

```

/*****
*           SEGGER MICROCONTROLLER SYSTEME GmbH           *
*       Solutions for real time microcontroller applications   *
*
*                   emWin problem report                   *
*
*****/

-----
File           : ProblemReport.c
CPU            :
Compiler/Tool chain :
Problem description :
-----
*/

#include "GUI.h"
/* Add further GUI header files here as required. */

/*****
*
*       Static code
*
*****/
* Please insert helper functions here if required.
*/

/*****
*
*       MainTask
*/
void MainTask(void) {
    GUI_Init();
    /*
    * To do: Insert the code here which demonstrates the problem.
    */
    while (1); /* Make sure program does not terminate */
}

```

20.6 FAQ's

- Q: I use a different LCD controller. Can I still use emWin?
A: Yes. The hardware access is done in the driver module and is completely independent of the rest of the GUI. The appropriate driver can be easily written for any controller (memory-mapped or bus-driven). Please get in touch with us.
- Q: Which CPUs can I use emWin 8051 with?
A: emWin 8051 can only be used with an 8051 CPU with a Keil compiler.

Q: Is emWin flexible enough to do what I want to do in my application?

A: emWin should be flexible enough for any application. If for some reason you do not think it is in your case, please contact us.

Q: Does emWin work in a multitask environment?

A: Yes, it has been designed with multitask kernels in mind.

Index

Numerics

- 3 pin SPI, configuration 242
- 4 pin SPI, configuration 243–244

A

- Access addresses, defining 22
- Access routines, defining 22
- Additional software 21
- Alias macro 22
- ANSI 12, 247
- API reference
 - colors 160
 - device simulator 30, 34
 - fonts 103
 - graphics 76
 - kernel interface routines 172
 - LCD driver 219
 - LCD layer 219
 - text 52
 - timing and execution 230
 - values 66
- Application program interface (API) .14, 218
- Arcs, drawing 95–96
- ASCII 52, 101, 107, 109

B

- Best palette option 139, 142, 144
- Binary switch macro 21
- Binary values, displaying 72–73
- Bitmap converter 13, 133–146
 - command line usage 142–144
 - supported input formats 134
 - using for color conversion 139–140
- Bitmaps 133–146
 - color conversion of 139–140
 - device-dependent (DDB) 135
 - device-independent (DIB) 135
 - drawing 83–85
 - full-color mode 139
 - generating "C" files from 133–139
 - manipulating 134
 - RLE compressed 135, 140, 145

- BmpCvt.exe 142–144

C

- "C" compiler 25, 135
- "C" files
 - converting bitmaps into 133–139
 - converting fonts into 109
 - inclusion of in emWin 21
- "C" programming language 12
- Callback routines 33
- Character sets 107–109
- Circles, drawing 93–94
- Clipping 75
- Color bar test routine 148–149
- Color conversion, of bitmaps . 134, 139–140
- Color lookup table (LUT) 160
- Color palettes
 - best palette option 139, 142, 144
 - custom 140–141, 160
 - fixed 139, 149–158
- Colors 147
 - converting 147
 - logical 147
 - physical 147
 - predefined 148
- COM/SEG lines
 - configuration 252–255
 - lookup tables for 255
- Command line usage
 - of bitmap converter 142–144
- Compile time switches 13
- Compiling, with simulator
 - demo program 26, 28
 - for your application 28
 - samples 27
- Config folder 21, 28, 231
- Configuration, of emWin 21
 - high-level 259–270
 - low-level 231–257
- Control characters 52, 101
- Coordinates 14
- Custom palettes
 - defining for hardware 160

- file formats, for color conversion 141
 - for color conversion 140–141
- D**
- Data types 16
 - Decimal values, displaying 66–69
 - Demos 14
 - Device simulation ??- 32
 - Device.bmp 30, 33
 - Device1.bmp 30, 33
 - Device-dependent bitmap (DDB) 135
 - Device-independent bitmap (DIB) 135
 - Directories, inclusion of 18
 - Directory structure
 - for emWin 18
 - for simulator 28
 - for Visual C++ workspace 28
 - Display driver 199
 - Drawing modes 77–78
- E**
- Ellipses, drawing 94–95
 - embOS 167
 - kernel interface routines for 174
 - emWin
 - as trial version 26–27
 - configuration of 21
 - data types used (see Data types)
 - directory structure for 18
 - features of 13
 - in multitask environments 22
 - initialization of 22
 - memory requirements 272
 - updating to newer versions 18
 - Execution model 167
 - supported types 168
- F**
- Fixed color palettes 139
 - Fixed palette modes 149–158
 - Floating-point calculations 75
 - Floating-point values, displaying 70–72
 - Font converter 13, 102, 109
 - Font editor 109
 - Font files
 - linking 102, 110
 - naming convention 111
 - Fonts 13, 101
 - adding 110
 - converting (see Font converter)
 - creating additional 102
 - declaring 102, 110
 - default 103
 - defining 13
 - Digit fonts (monospaced) 130
 - Digit fonts (proportional) 128–129
 - editing 109
 - file naming convention 111–112
 - generating "C" files from 109
 - included with emWin 13, 101
 - monospaced 110, 123–127
 - naming convention 110–111
 - proportional 102, 110, 113–122
 - scaling 13
 - selecting 103–104
 - usage of 102
- G**
- Foreign Language Support 189
 - Full bus interface, configuration ... 247–249
 - Full-color mode, of bitmaps 139
 - Function replacement macro 22
 - Function-level linking 18
- G**
- Graphic library 13, 75, 229
 - Grayscales 139, 147
 - GUI configuration 260–270
 - GUI subdirectories 18, 28
 - GUI_ALLOC_ALLOC 267
 - GUI_ALLOC_AssignMemory 263
 - GUI_ALLOC_FREE 267
 - GUI_ALLOC_GETMAXSIZE 268
 - GUI_ALLOC_H2P 268
 - GUI_ALLOC_SetAvBlockSize 263
 - GUI_ALLOC_SIZE 260
 - GUI_BITMAP structures 134
 - GUI_CalcColorDist 163
 - GUI_CalcVisColorError 163
 - GUI_Clear 62
 - GUI_ClearKeyBuffer 187
 - GUI_ClearRect 79
 - GUI_Color2Index 163
 - GUI_Color2VisColor 164
 - GUI_ColorIsAvailable 164
 - GUI_COMPILER_SUPPORTS_FP 269–270
 - GUI_CONST_STORAGE 269
 - GUI_DEBUG_LEVEL 260
 - GUI_DEFAULT_BKCOLOR 260
 - GUI_DEFAULT_COLOR 260
 - GUI_DEFAULT_FONT 260
 - GUI_Delay 229–230
 - GUI_DispBin 72
 - GUI_DispBinAt 73
 - GUI_DispCEOL 62
 - GUI_DispChar 53
 - GUI_DispCharAt 54
 - GUI_DispChars 54
 - GUI_DispDec 66
 - GUI_DispDecAt 67
 - GUI_DispDecMin 67
 - GUI_DispDecShift 68
 - GUI_DispDecSpace 68
 - GUI_DispFloat 70
 - GUI_DispFloatFix 71
 - GUI_DispFloatMin 71
 - GUI_DispHex 73
 - GUI_DispHexAt 74
 - GUI_DispNextLine 54
 - GUI_DispSDec 68
 - GUI_DispSDecShift 69
 - GUI_DispSFloatFix 72
 - GUI_DispSFloatMin 72
 - GUI_DispString 55
 - GUI_DispStringAt 55
 - GUI_DispStringAtCEOL 55
 - GUI_DispStringHCenterAt 56
 - GUI_DispStringInRect 56
 - GUI_DispStringInRectEx 56
 - GUI_DispStringInRectWrap 56
 - GUI_DispStringLength 57
 - GUI_DrawArc 95
 - GUI_DrawBitmap 83

- I2C bus interface, configuration 245–246
- Initializing emWin 22
- Input devices 185–188
 - keyboard ??- 188
- Interrupt service routines 168–170
- ISO 8859-1 101, 107, 109

- K**
- Kernel interface routines 168–170, 172
- Keyboard input support ??- 188

- L**
- LCD 215
 - caching in memory 13
 - configuration of 147, 231–256
 - connecting to microcontroller 15
 - initialization of 22
 - simulated 30
 - without LCD controller 15
- LCD controller
 - configuration of 231–257
 - connected to port/buffer 15
 - initialization of 235
 - memory-mapped 15
 - support for 15, 200
 - with LUT hardware 164, 255
- LCD driver 26
 - availability/selection 200
 - customization of 15
- LCD_BITSPERPIXEL 234
- LCD_BUSWIDTH 215, 248
- LCD_CACHE 256
- LCD_CNF4 215
- LCD_CONTROLLER 202, 234
- LCD_ENABLE_MEM_ACCESS 215, 249
- LCD_ENABLE_REG_ACCESS 215, 249
- LCD_ENDIAN_BIG 207, 210
- LCD_FILL_RECT 207, 210, 215
- LCD_FIXEDPALETTE 234
- LCD_GetBitsPerPixel 224
- LCD_GetBitsPerPixelEx 224
- LCD_GetFixedPalette 225
- LCD_GetFixedPaletteEx 225
- LCD_GetNumColors 225
- LCD_GetNumColorsEx 225
- LCD_GetVXSize 226
- LCD_GetVXSizeEx 226
- LCD_GetVYSize 226
- LCD_GetVYSizeEx 226
- LCD_GetXMagEx 226
- LCD_GetXSize 227
- LCD_GetXSizeEx 227
- LCD_GetYMagEx 226
- LCD_GetYSize 227
- LCD_GetYSizeEx 227
- LCD_INIT_CONTROLLER 235
- LCD_L0_ControlCache 223
- LCD_L0_DrawBitMap 220
- LCD_L0_DrawHLine 221
- LCD_L0_DrawPixel 221
- LCD_L0_DrawVLine 221
- LCD_L0_FillRect 221
- LCD_L0_GetPixelIndex 222
- LCD_L0_Init 220
- LCD_L0_Off 220
- LCD_L0_On 220
- LCD_L0_SetLUTEntry 223
- LCD_L0_SetPixelIndex 222
- LCD_L0_XorPixel 222
- LCD_LIN_SWAP 210–211
- LCD_LUT_COM 255
- LCD_LUT_SEG 255
- LCD_MAX_LOG_COLORS 238
- LCD_MIRROR_X 237
- LCD_MIRROR_Y 237
- LCD_NUM_CONTROLLERS 256
- LCD_NUM_DUMMY_READS 217
- LCD_OFF 207, 210, 215, 257
- LCD_ON 207, 210, 215, 257
- LCD_PHYSCOLORS 238
- LCD_PHYSCOLORS_IN_RAM 165
- LCD_READ_A0 239, 245
- LCD_READ_A1 239, 245
- LCD_READ_MEM 210, 214, 247
- LCD_READ_REG 214–215, 247
- LCD_READM_A1 217
- LCD_REG01 217
- LCD_REVERSE 238
- LCD_SERIAL_ID 217
- LCD_SET_LUT_ENTRY .. 207, 210, 215, 239
- LCD_SUPPORT_CACHECONTROL 256
- LCD_SWAP_BYTE_ORDER 215, 249
- LCD_SWAP_RB 215, 239
- LCD_SWAP_XY 238
- LCD_TIMERINIT0 257
- LCD_TIMERINIT1 257
- LCD_USE_BITBLT 215, 256
- LCD_USE_PARALLEL_16 217
- LCD_USE_SERIAL_3PIN 217
- LCD_VRAM_ADR 207, 210
- LCD_VXSIZE 251
- LCD_VYSIZE 251
- LCD_WRITE 242
- LCD_WRITE_A0 217, 240, 243, 245
- LCD_WRITE_A1 217, 240, 243, 245
- LCD_WRITE_BUFFER_SIZE 217
- LCD_WRITE_MEM 210, 214, 248
- LCD_WRITE_REG 214–215, 248
- LCD_WRITEM 242
- LCD_WRITEM_A0 217
- LCD_WRITEM_A1 217, 240, 243, 246
- LCD_X_InitController 215, 224
- LCD_XSIZE 234
- LCD_YSIZE 234
- LCD667XX driver 216–217
- LCDConf.c 21
- LCDConf.h 15, 21, 160, 199, 231
- LCDDummy driver 217
- LCDLin driver 205–215
- LCDLin32 driver 207
- LCDLin32168 driver 206
- LCDNull driver 218
- LCDTemplate driver ??- 218
- Library, creating 19
- Linearization 165
- Lines, drawing 85–87
- Linking source files 18
- Lookup table (LUT) 158, 160, 164, 218, 255

- M**
- Memory, reducing consumption of .134, 139
- Multitask environments 168–171

- multiple tasks call emWin ... 168, 170–171
- one task calls emWin 168–169
- N**
- NORMAL drawing mode 77
- Normal text 58
- Numerical value macro 21
- O**
- Optional software 21
- P**
- Palettes (see Color palettes)
- Performance 271
- Pixels 14
- Polygons, drawing 89–93
- Proportional fonts (see Fonts)
- R**
- Real bus-interface 240
- resource 30
- Resource file 30, 33
- Resource semaphore 172
- Resource usage 271
- Reverse text 58
- RLE compression, of bitmaps .135, 140, 145
- S**
- S1D13806 controller 215
- Sample programs 14, 22
- SED1386 controller 215
- Selection switch macro 22
- SIM_GUI_CreateLCDInfoWindow() 41
- SIM_GUI_CreateLCDWindow 41
- SIM_GUI_Exit 42
- SIM_GUI_Init 42
- SIM_GUI_SetLCDColorBlack 31
- SIM_GUI_SetLCDColorWhite 31
- SIM_GUI_SetLCDPos 32
- SIM_GUI_SetLCDWindowHook 42
- SIM_GUI_SetMag 32
- SIM_GUI_SetTransColor 32
- SIM_HARDKEY_GetNum 34
- SIM_HARDKEY_GetState 34
- SIM_HARDKEY_SetCallback 35
- SIM_HARDKEY_SetMode 33, 35
- SIM_HARDKEY_SetState 36
- SIM_SetTransColor 30
- Simple bus interface, configuration 239–241
- Simulator 13, 25
 - directory structure for 28
 - usage of with emWin source 28–29
 - usage of with emWin trial version .. 26–27
- Single task system 168–169
- Source files, linking 18
- Sprintf 65
- Standard fonts 101
- Subdirectories, of GUI 18
- Superloop 168–169
- Support 273
- Syntax, conventions used 14
- T**
- Text
 - alignment 60–61
 - displaying 51–58
 - modes 58–59
 - positioning 52, 61–62
- Tick 229–230
- Time-related functions 229
- Toggle behavior, of hardkeys 33, 36
- Touch-screens 13
- Transparency
 - in device simulation 30
- Transparent reversed text 58
- Transparent text 58
- Trial version, of emWin 26–27
- Tutorial 22–23
- U**
- uC/OS 167
 - kernel interface routines for 174
- Unicode 101, 109
 - API reference 192
 - displaying characters in 190
- UTF-8 strings 191
- V**
- Values, displaying 65
- Vectorized symbols 89
- Viewer 13–49
- Virtual display 13
- Virtual screen support 177–181
- Visual C++ 26, 28
 - directory structure for 28
- W**
- Western Latin character set (see ISO 8859-1)
- Widgets 229
- Win32, kernel interface routines for 175
- Window manager 230
- Windows
 - clearing 62
- WM_Exec 230
- X**
- X-axis 14
- XOR drawing mode 77
- XOR text 58
- Y**
- Y-axis 14