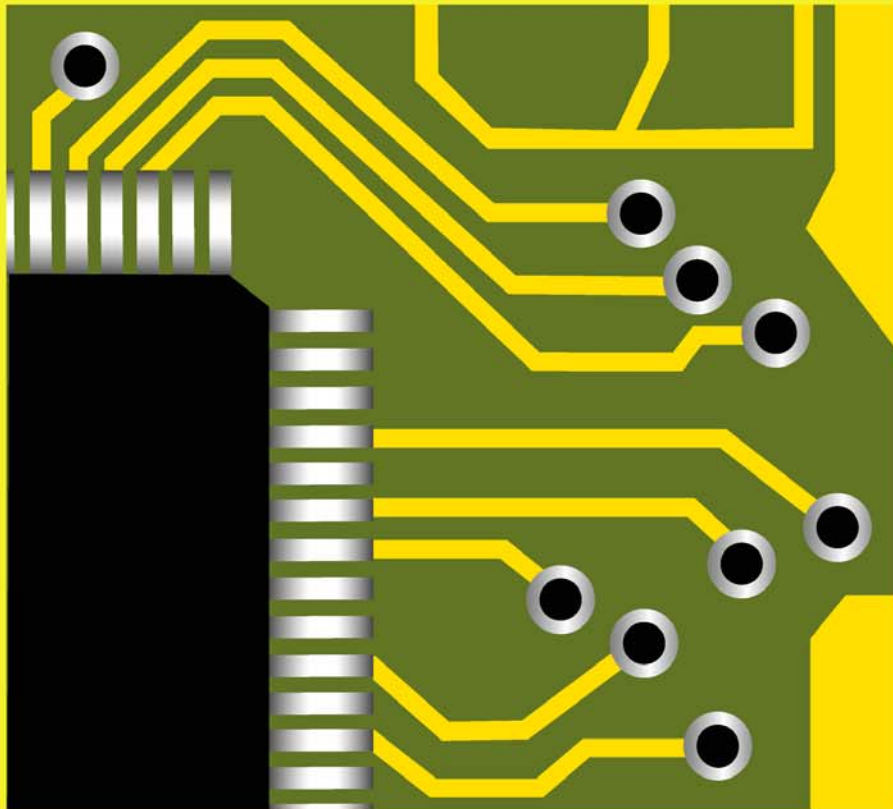


Graphical User Interface Tool for Embedded Systems

*easy*GUI

Version 5.3.0

Revision F



manual



A product from



Toldbodgade 95
DK-1253 Copenhagen K
Denmark
Phone: +45 7022 0495
Fax: +45 7023 0495
SE/VAT No. DK-27 06 03 07
www.easyqui.com
sales@ibissolutions.com

Copyrighted © 1999 - 2007 IBIS Solutions ApS.

*easy*GUI

CONTENT

1	PREFACE	12
2	INSTALLATION	13
	Installation	13
	easyGUI licensing	13
3	INTRODUCTION.....	15
	How does it work?	15
	The display	16
	Menus.....	17
	File functions	18
	Font functions.....	18
	Project functions	19
	C code generation	19
	Import / export.....	19
	Help functions.....	19
4	FONTS	20
	Font types	20
	Character modes	20
	Text fonts.....	21
	Character definition	22
	Proportional writing	24
	Font style.....	26
	Undefined characters	26
	Font compression	27
	Current fonts	27
5	FONT LIST WINDOW	29
6	FONT EDITING WINDOW	30
	Font setup	31
	Font selection	33
	Previously in easyGUI	34
	View filter	34
	Font editing	35
	Pixel editing	35
	PS mark editing	35
	Editing many characters at once.....	36
	Editing commands.....	36
	TTF import.....	38

	Character testing	39
7	PROJECT PARAMETERS WINDOW	41
	Basics	42
	Project panel	42
	Display panel.....	42
	Display controller.....	45
	Color	47
	Colors panel	47
	Color / Grayscale mode panel	48
	Color depth panel.....	48
	Palette handling	58
	Color handling	62
	RGB format	63
	Simulated display	65
	Compiler	66
	Type definitions panel	67
	Constant declarations panel	67
	Special compiler settings panel	68
	Buffer sizes panel.....	68
	Operation	69
	Text setup panel	69
	Auto redraw panel.....	70
	Cursor mode panel.....	70
	Scroll mode panel.....	70
	Module selection panel.....	70
8	LANGAUGE TRANSLATION WINDOW.....	72
9	POSITIONS WINDOW	75
10	VARIABLES WINDOW	76
	Importing definitions.....	77
	Import setup	77
	Import type.....	78
	Making the import	80
11	STRUCTURES WINDOW	81
	The basics	81
	Items.....	81
	Window layout	83
	Structure management panel	84
	Item list panel.....	86
	Item data panel.....	87
	Structure hierarchy sub-panel	87
	Primary position sub-panel	87

Secondary position sub-panel	88
Structure call sub-panel	88
Variable sub-panel	88
Active area sub-panel	89
Clipping sub-panel.....	89
Touch area sub-panel	89
Alignment sub-panel.....	89
Foreground color sub-panel	90
Background color sub-panel.....	90
Text sub-panel.....	91
Paragraph sub-panel.....	93
Bitmap sub-panel.....	94
Rectangle sub-panel	94
Variable formatting sub-panel	94
Miscellaneous sub-panel.....	95
Display panel	97
Use of Touch areas	104
1 - Touch interface hardware	104
2 - Coordinate training.....	105
3 - Event handling	106
12 C CODE GENERATION	108
13 IMPORT / EXPORT.....	110
Current project panel	111
External project panel	111
Middle panel - controls and settings	113
14 HOW TO SET UP YOUR SYSTEM	114
Minimum RAM and ROM requirements	114
Operating system	114
Setting up the system for easyGUI use	115
1 - Physical display connection.....	115
2 - Setting up easyGUI for your display type	116
3 - Display control functions	117
Display initialization.....	118
Selecting a display driver.....	118
Display writing.....	126
Light and contrast control.....	129
4 - Compiling the project.....	129
5 - easyGUI interfacing	129
GuiLib_Init	130
GuiLib_Refresh	130
GuiLib_ShowScreen	131
Testing the system	131
1 - Establishing some kind of connection	131
2 - Turning on a single pixel	132
3 - Showing the test pattern	132

4 - Showing an easyGUI structure	135
15 HOW TO UTILIZE easyGUI - A TUTORIAL	136
Efficient learning	136
Item types.....	136
Viewing the structure	138
Splash structure	139
Structure details	140
Clearing the screen.....	140
Finding this and that item.....	141
Drawing a logo	141
A centered, relative text.....	142
PS - nice texts	142
Big texts - small texts	144
Showing variables	144
Config structure	146
Structure details	146
Don't forget the coordinates	147
Using an indexed structure	148
Utilizing a disappearing indexed structure	151
An on/off text	152
Backgrounds are important.....	153
The fine art of cursor fields	155
Main Menu structure	158
Better looking menu items.....	159
Playing with cursor indices.....	159
Flash structure	161
Mixing structures and plain graphics	161
Let's scroll	162
16 easyGUI FUNCTION REFERENCE	167
GuiConst unit.....	168
Constants	168
GuiConst_AUTOREDRAW_FIELDS_MAX.....	168
GuiConst_AUTOREDRAW_MAX_VAR_SIZE.....	168
GuiConst_AUTOREDRAW_ON_CHANGE.....	168
GuiConst_AVR_COMPILER_FLASH_RAM	168
GuiConst_AVRGCC_COMPILER	169
GuiConst_BIT_BOTTOMRIGHT	169
GuiConst_BIT_TOPLEFT.....	169
GuiConst_BITMAP_SUPPORT_ON	169
GuiConst_BLINK_FIELDS_MAX	169
GuiConst_BLINK_SUPPORT_ON	169
GuiConst_BYTE_HORIZONTAL	169
GuiConst_BYTE_LINES	170
GuiConst_BYTE_VERTICAL.....	170
GuiConst_BYTES_PR_LINE.....	170
GuiConst_BYTES_PR_SECTION.....	170

GuiConst_CHAR	170
GuiConst_CHARMODE_ANSI	170
GuiConst_CHARMODE_UNICODE	170
GuiConst_CLIPPING_SUPPORT_ON	170
GuiConst_CODEVISION_COMPILER	171
GuiConst_COLOR_BYTE_SIZE	171
GuiConst_COLOR_DEPTH_1	171
GuiConst_COLOR_DEPTH_2	171
GuiConst_COLOR_DEPTH_4	171
GuiConst_COLOR_DEPTH_5	171
GuiConst_COLOR_DEPTH_8	171
GuiConst_COLOR_DEPTH_12	172
GuiConst_COLOR_DEPTH_15	172
GuiConst_COLOR_DEPTH_16	172
GuiConst_COLOR_DEPTH_18	172
GuiConst_COLOR_DEPTH_24	172
GuiConst_COLOR_MAX.....	172
GuiConst_COLOR_MODE_GRAY	172
GuiConst_COLOR_MODE_PALETTE	172
GuiConst_COLOR_MODE_RGB.....	173
GuiConst_COLOR_PLANES_1.....	173
GuiConst_COLOR_PLANES_2.....	173
GuiConst_COLOR_RGB_STANDARD.....	173
GuiConst_COLOR_SIZE	173
GuiConst_COLORCODING_B_MASK.....	173
GuiConst_COLORCODING_B_MAX.....	173
GuiConst_COLORCODING_B_SIZE	174
GuiConst_COLORCODING_B_START.....	174
GuiConst_COLORCODING_G_MASK	174
GuiConst_COLORCODING_G_MAX	174
GuiConst_COLORCODING_G_SIZE.....	174
GuiConst_COLORCODING_G_START	174
GuiConst_COLORCODING_R_MASK.....	174
GuiConst_COLORCODING_R_MAX.....	174
GuiConst_COLORCODING_R_SIZE	175
GuiConst_COLORCODING_R_START.....	175
GuiConst_CONTROLLER_COUNT_HORZ	175
GuiConst_CONTROLLER_COUNT_VERT	175
GuiConst_CURSOR_FIELDS_MAX.....	175
GuiConst_CURSOR_MODE_STOP_TOP	175
GuiConst_CURSOR_MODE_WRAP_AROUND.....	175
GuiConst_CURSOR_SUPPORT_ON.....	176
GuiConst_DECIMAL_COMMA	176
GuiConst_DECIMAL_PERIOD	176
GuiConst_DISPLAY_ACTIVE_AREA	176
GuiConst_DISPLAY_ACTIVE_AREA_CLIPPING	176
GuiConst_DISPLAY_ACTIVE_AREA_COO_REL	176
GuiConst_DISPLAY_ACTIVE_AREA_X1	176
GuiConst_DISPLAY_ACTIVE_AREA_Y1	177
GuiConst_DISPLAY_ACTIVE_AREA_X2	177
GuiConst_DISPLAY_ACTIVE_AREA_Y2	177
GuiConst_DISPLAY_BYTES.....	177

GuiConst_DISPLAY_HEIGHT	177
GuiConst_DISPLAY_HEIGHT_HW	177
GuiConst_DISPLAY_WIDTH	177
GuiConst_DISPLAY_WIDTH_HW	178
GuiConst_FLOAT_SUPPORT_ON	178
GuiConst_FONT_UNCOMPRESSED	178
GuiConst_ICC_COMPILER	178
GuiConst_INT8S	178
GuiConst_INT8U	178
GuiConst_INT16S	178
GuiConst_INT16U	178
GuiConst_INT24S	179
GuiConst_INT24U	179
GuiConst_INT32S	179
GuiConst_INT32U	179
GuiConst_INTCOLOR	179
GuiConst_ITEM_TEXTBLOCK_INUSE	179
GuiConst_ITEM_TOUCHAREA_INUSE	179
GuiConst_KEIL_COMPILER_REENTRANT	180
GuiConst_LANGUAGE_CNT	180
GuiConst_LANGUAGE_XXX	180
GuiConst_MAX_TEXT_LEN	180
GuiConst_MAX_VARNUM_TEXT_LEN	180
GuiConst_MIRRORED_HORIZONTALLY	180
GuiConst_MIRRORED_VERTICALLY	181
GuiConst_PALETTE_SIZE	181
GuiConst_PICC_COMPILER_ROM	181
GuiConst_PIXEL_OFF	181
GuiConst_PIXEL_ON	181
GuiConst_PTR	181
GuiConst_REL_COORD_ORIGO_INUSE	181
GuiConst_REVERSED_BYTE_PAIRS	182
GuiConst_ROTATED90DEGREE	182
GuiConst_ROTATED90DEGREE_LEFT	182
GuiConst_ROTATED90DEGREE_RIGHT	182
GuiConst_ROTATED_OFF	182
GuiConst_ROTATED_UPSIDEDOWN	182
GuiConst_SCROLL_MODE_STOP_TOP	182
GuiConst_SCROLL_MODE_WRAP_AROUND	183
GuiConst_SCROLL_SUPPORT_ON	183
GuiConst_TEXT	183
GuiConst_TOUCHAREA_MAX	183
GuiLib unit	183
Constants	184
GuiLib_CHR_SET	184
GuiLib_NO_CURSOR	184
GuiLib_NO_RESET_AUTO_REDRAW	184
GuiLib_RESET_AUTO_REDRAW	184
Variables	184
GuiLib_ActiveCursorFieldNo	184
GuiLib_CurStructureNdx	185
GuiLib_LanguageCharSet	185

GuiLib_LanguageIndex	185
GuiLib_ScrollActiveLine	185
GuiLib_ScrollTopLine.....	185
GuiLib_ScrollVisibleLines	185
Functions	186
GuiLib_BlinkBoxMarkedItem	186
GuiLib_BlinkBoxStart	186
GuiLib_BlinkBoxStop.....	187
GuiLib_BorderBox.....	187
GuiLib_Box.....	187
GuiLib_Clear	188
GuiLib_ClearDisplay	188
GuiLib_Cursor_Down	188
GuiLib_Cursor_End	189
GuiLib_Cursor_Home	189
GuiLib_Cursor_Select.....	189
GuiLib_Cursor_Up	190
GuiLib_Dot	190
GuiLib_DrawChar	190
GuiLib_DrawStr.....	191
GuiLib_FillBox	193
GuiLib_GetDot	193
GuiLib_GetTextLanguagePtr	193
GuiLib_GetTextPtr	194
GuiLib_GetTextWidth	194
GuiLib_GrayScaleToRgbColor	194
GuiLib_HLine	195
GuiLib_Init	195
GuiLib_InvertBox	195
GuiLib_InvertBoxStart.....	196
GuiLib_InvertBoxStop	196
GuiLib_Line	196
GuiLib_MarkDisplayBoxRepaint.....	197
GuiLib_PixelToRgbColor.....	197
GuiLib_RedrawScrollList	197
GuiLib_Refresh	198
GuiLib_ResetClipping	198
GuiLib_ResetDisplayRepaint	198
GuiLib_RgbColorToGrayScale	198
GuiLib_RgbToPixelColor.....	199
GuiLib_Scroll_Down.....	199
GuiLib_Scroll_End	199
GuiLib_Scroll_Home.....	200
GuiLib_Scroll_To_Line	200
GuiLib_Scroll_Up	201
GuiLib_ScrollLineOffsetY	201
GuiLib_SetClipping	202
GuiLib_SetLanguage	202
GuiLib_SetScrollPars	202
GuiLib_ShowBitmap.....	203
GuiLib_ShowBitmapAt.....	203
GuiLib_ShowScreen	203



	GuiLib_StrAnsiToUnicode	204
	GuiLib_TestPattern	204
	GuiLib_TouchAdjustReset	204
	GuiLib_TouchAdjustSet	205
	GuiLib_TouchCheck	205
	GuiLib_UnicodeStrCmp.....	205
	GuiLib_UnicodeStrCpy.....	206
	GuiLib_UnicodeStrLen	206
	GuiLib_VLine	206
	GuiDisplay unit.....	207
	Functions	207
	GuiDisplay_Init	207
	GuiDisplay_Lock.....	207
	GuiDisplay_Refresh	208
	GuiDisplay_Unlock.....	208
17	easyTRANSLATE	209
	Installation	209
	Principles.....	209
	How to use	210
18	easyGUI PC SIMULATION TOOLSET	212
	Purpose.....	212
	Necessary files	212
	Compilation	214
	Limitations.....	215
19	easyGUI VERSIONS	216


1 PREFACE


Welcome to the easyGUI world of fast and efficient graphical user interface editing.


easyGUI is an application for defining user interfaces on small displays using graphical primitives.

The use of an SQL database in easyGUI project files ensures maximum stability, speed and efficiency in the development environment.

This manual covers all versions of the easyGUI package. Sections specific to one or the other are marked with:

 Monochrome for the monochrome version.

 Color for the color version.

 Unicode for the Unicode version.

The color version contains all functionality of the monochrome version, plus support for more than one bit per display pixel, i.e. color depths of more than two colors. Grayscale displays are handled just as color displays by easyGUI.

The Unicode version contains all functionality of the color version, plus support for 16 bit Unicode character codes.

Furthermore, the PC Simulation Toolset, and the easyTRANSLATE accessories, is covered in this manual.

2 INSTALLATION

easyGUI can be installed on standard PC's running Windows 2000, Windows XP, Windows Vista, or higher. easyGUI will not function properly on Windows 95, Windows 98, Windows Me, or similar older operating systems.

INSTALLATION

Run the easyGUI installation program, following the instructions on screen.

Several fonts are required:

- Arial. Should be present in a standard Windows installation.
- Arial Narrow. Is part of e.g. Microsoft Office.
- Arial Unicode MS.

The installation program installs these fonts, if needed. The fonts can also be found in the `Fonts` sub folder of the easyGUI install folder.

EASYGUI LICENSING

easyGUI can be licensed in two ways:

- By software license. A temporary software license key is issued when purchasing easyGUI. When easyGUI is started for the first time it asks for license information. Enter the user name and temporary license key, as delivered with the package. easyGUI will now be fully functional for a limited amount of time (typically 14 days). During this period send an E-mail to sales@easygui.com, stating your user name and hardware ID, as shown on easyGUI's main window, or in the Help | License function. You will then receive a permanent license key corresponding to your user name and your PC's machine ID. If it is necessary to run easyGUI on other PC's, additional licenses must be purchased. In the event of PC equipment replacements and/or updates, where the hardware ID changes, just contact support@easygui.com.
- By dongle. This is an extra feature, which can be purchased on the easyGUI web site. You will still be supplied with a temporary software license key, in order to get started while waiting for the dongle to arrive by mail. When the dongle is received it is simply placed in a free USB connector on the PC, and easyGUI is restarted. easyGUI will then recognize the dongle. The advantage of using a

dongle is that easyGUI may be installed on several PC's, and the dongle simply moved around to the desired PC.

Usually the dongle just works with the drivers found in Windows. If not, the official HASP dongle driver must be installed. It is located in the dongle folder under the easyGUI installation folder. It can also be downloaded from the HASP site at (topmost item):

www.aladdin.com/support/hasp/enduser.asp

No customer problems have been reported after installing the HASP driver.

If you want the highest degree of freedom in using the easyGUI package, we recommend the dongle license.

The dongle license can also be purchased later to an existing easyGUI installation.

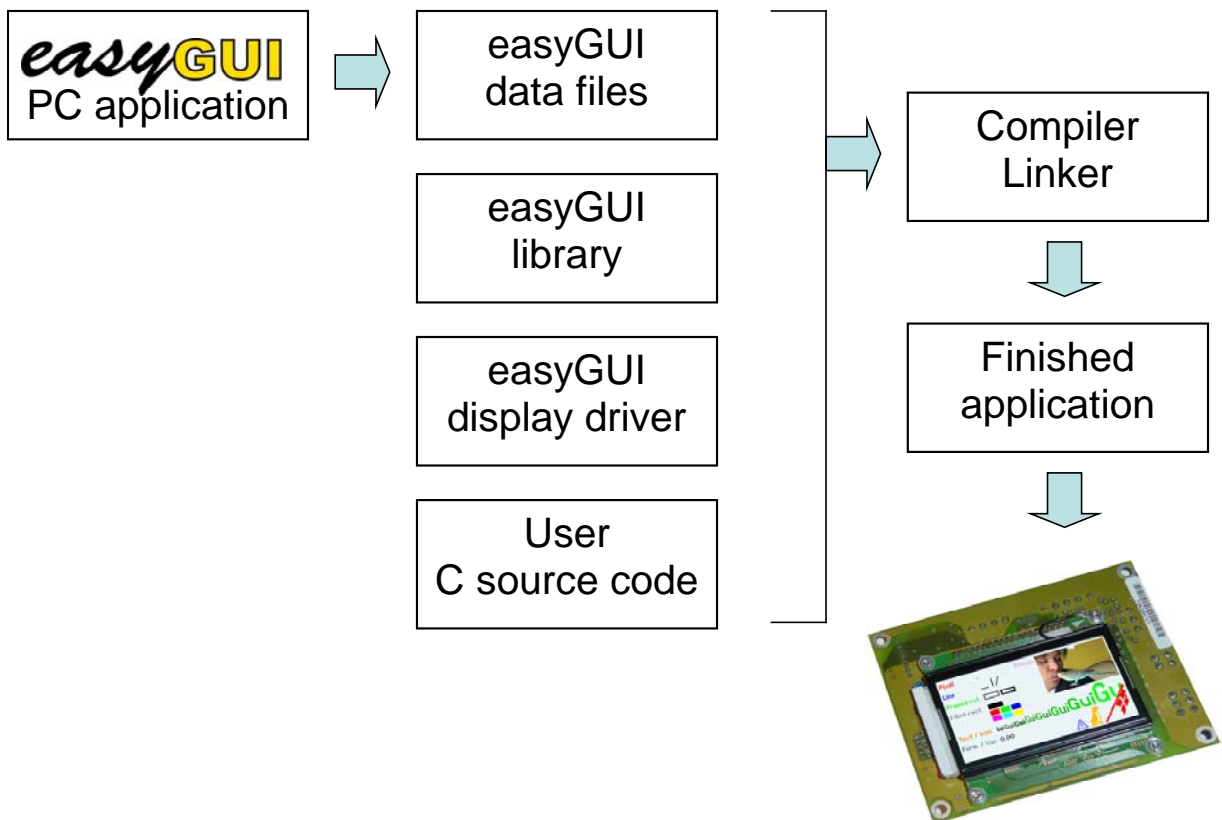
Please observe that purchasing the dongle license does *not* constitute a second license. For two licenses two full easyGUI packages must be purchased, with or without dongles.

We offer a discount if purchasing more than three licenses. Please contact sales@easygui.com if you need multiple licenses.

3 INTRODUCTION

HOW DOES IT WORK?

The process of development when using easyGUI can be summed up as:



- The **easyGUI PC application** is explained in detail in this manual. The major part of the work regarding the user interface takes place here, contrary to standard development work, where everything is done in the target system c code.
- The **easyGUI data files** are generated by easyGUI, at the command of a button. Each time something has been changed in the user interface, and it is desirable to test it on the target system (or the PC simulation toolkit) the files must be re-generated.
- An **easyGUI library** is delivered with the easyGUI package, in plain c code. This library must be linked into the target system application, just like the other modules making up the final system.

- **User c source code** is the last part of the target system source code, containing all the working code necessary for the finished system, with calls to the easyGUI library, and other code related to e.g. hardware in the target system.

This manual describes the various steps necessary to take, in order to use easyGUI as an efficiently tool for generating high quality user interfaces in embedded systems.

As easyGUI is complex there will of course be a learning curve, as with all other advanced tools, but the reward will be reduced development time, and a better final product, when the easyGUI tool has been mastered. Take your time, start with simple problems, and gradually work your way through the system, using more and more advanced functions, as the needs arise.

THE DISPLAY

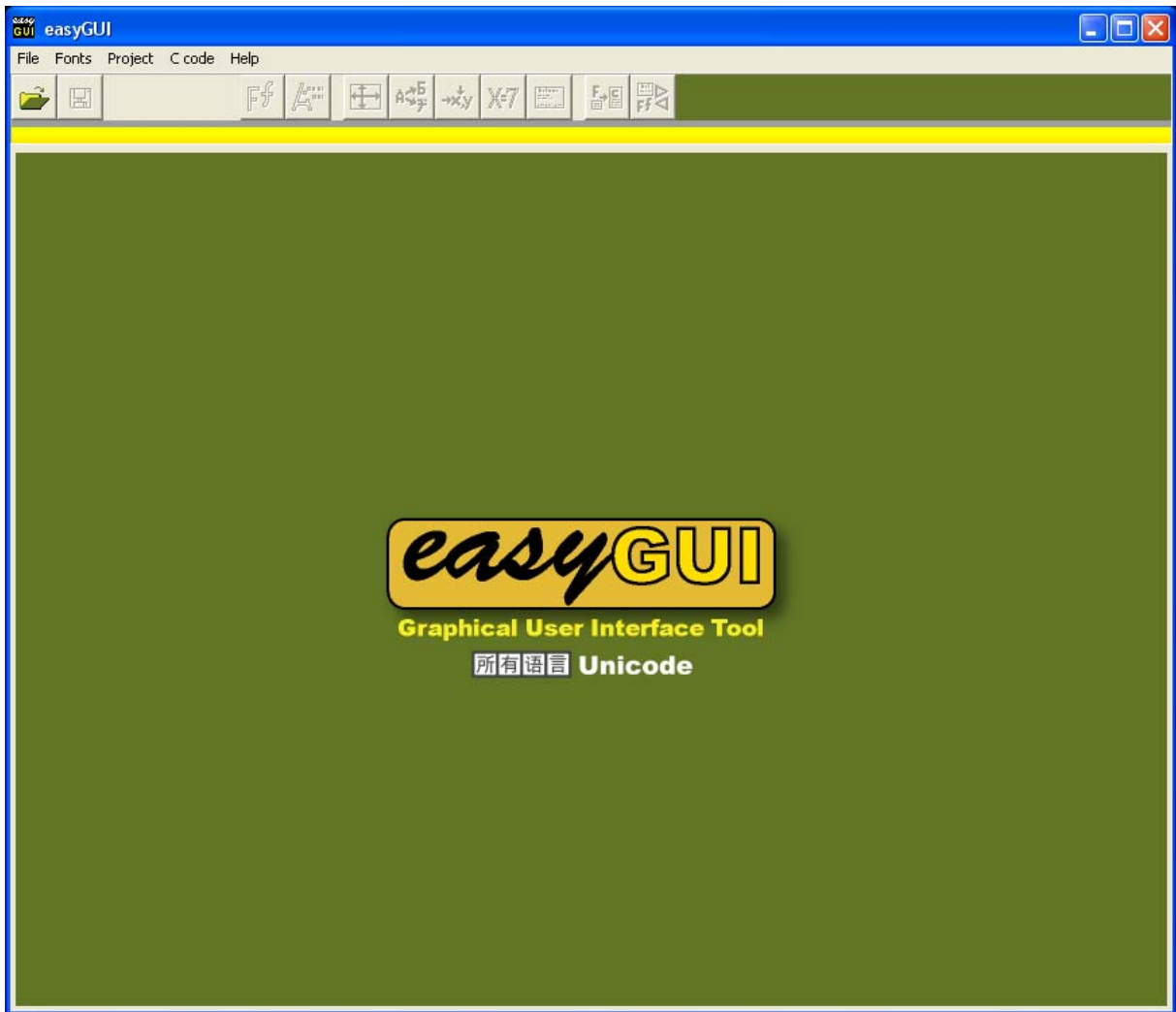
easyGUI treats the target system display as a graphical canvas, i.e. a drawing surface on which objects may be placed. All placement is free, so there is no predefined positions or grid which limits the artistic freedom.

The coordinate system has origo (0,0) at the upper left corner, with X coordinates going towards the right, and Y coordinates going downwards. The coordinates are counted in display pixels, so the display sets the limits for the coordinate system.

There is no formal limit on display size in easyGUI, but the vast majority of systems using easyGUI have display resolutions of 640×480 pixels (VGA) or lower, with most at or below 320×240 pixels (quarter VGA).

easyGUI supports any color depth, ranging from simple monochrome systems (one bit per pixel) all the way up to true color systems (24 bits per color).

MENUS



easyGUI is controlled through a number of functions, one for each main subject.

One project can be loaded at a time. The project contains all fonts, screen structures, etc. for one display. If the target system utilizes several different displays a project should be created for each display. Several displays handled by the same μ -processor is not supported by easyGUI.

easyGUI contains a number of items:

- Basic file functions.
- Font management.
- Font editing.
- Project parameters.

easyGUI

- Positions.
- Variables.
- Structures (screen designs).
- Language support.
- C-source code generation.
- Import / export of data between easyGUI projects.

The individual items are explained in the following chapters.

File functions



Open - opens an existing project. The ctrl + O command can also be used.



Save - saves changes to the project. The F2 key, or the ctrl + S command, can also be used.

SaveAs - Saves the project under another name. The ctrl + A command can also be used.

New - Creates a new project containing only basic data. The ctrl + N command can also be used.

Close - closes the currently open project. The ctrl + F4 command can also be used.

Exit - closes easyGUI. The ctrl + Q command can also be used.

The nine projects last opened are remembered by the system for easy access. They are presented in the File menu below the commands. Selecting one of them corresponds to opening it normally with the Open command.

Font functions



Font list - manages complete fonts. The F3 key activates this window.



Font editing - edits a single font, selected in the font list. Manages font selection for the target system. The F4 key activates this window.

Project functions



Project parameters. Basic settings for the project, defining e.g. display size and display controller memory layout. The F5 key activates this window.



Language translation - allows definition of languages in the project, and translation of texts. The F7 activates this window.



Positions - manages fixed positions for screen structures. The F8 key activates this window.



Variables - manages variables for dynamic screen structure control. The F9 key activates this window.



Structures - the core function in easyGUI. Manages screen structures. The F10 key activates this window.

C code generation



C code generation - converts easyGUI data to c and h files for inclusion in the target system code. The F11 key activates this window.

Import / export



Import / export - moves data between easyGUI projects. The F12 key activates this window.

Help functions

Help - displays this manual. The F1 key can also be used.

About - displays information about the easyGUI program. The ctrl + I command can also be used.

License - displays license data, allowing license key updating. The ctrl + L command can also be used. If a working dongle is attached the dongle ID will be shown.

4 FONTS

FONT TYPES

All characters and icons are organized into fonts. Fonts are divided into text fonts containing normal characters, and icon fonts containing graphical elements, but internally all fonts are treated identically. A text font contains one set of characters or icons in one size, covering one or more languages, e.g. both Western style and Asian style characters in the same font. All characters (or icons) in a font have the same basic parameters regarding height, style etc.

Observe that bitmaps can also be displayed by using a bitmap item, this is unrelated to the fonts, and will be explained later.

Fonts are by definition monochrome.

CHARACTER MODES

There are two fundamental character modes in easyGUI:

- **ANSI** mode. Each font can contain up to 224 primary characters (character codes 32~255), and up to 224 shadow characters. 8 bit character codes are used on the target system.

In order to make room for the shadow characters in an 8bit character code system the characters are organized into two character sets, each containing up to 224 characters. The primary character set is numbered 0, while the shadow set is numbered 1. Characters in character set 1 are numbered 256 - 511. The primary character set contains the normal ANSI Western style character set, as used in Windows. The shadow character set contains Japanese Katakana characters as default, but other uses are possible. When the target system code runs, a language is selected at all times. Each language has a character set number assigned (defined in language setup, see later). When easyGUI needs to display a character it first looks in the currently active character set, i.e. No. 0 or 1. If a character is defined in the character set it is displayed, if not easyGUI displays the corresponding character in character set 0. If no character is found here either a black block is displayed instead, indicating a font problem. With this system easyGUI can show more than 256 characters while still using only 8 bit character codes, but of course all characters in use in character set 1 will mask out the corresponding characters in character set 0, when character set 1 is specified for use. Therefore the supplied Katakana characters are placed in the last part of character set 1, so that the characters "lost" in character set 0 does not create a problem. These masked out characters are mostly national western characters (e.g. "ô"), which are of little use in a Japanese environment.

Please observe that the shadow character set feature is considered obsolete, and is only maintained because of backward compatibility issues. Instead the Unicode system described below should be used, whenever special characters are needed.

- **Unicode mode.** Each font can contain up to 65504 characters. 16 bit character codes are used on the target system.

All Unicode fonts should as a principle include the basic Windows ANSI Western style character set in character codes 32 - 255.

The International Unicode Consortium defines Unicode character codes. On their web page www.unicode.org can be found character code charts for a large portion of the worlds character sets. All easyGUI Unicode fonts conform to this standard, as this ensures easy compatibility with other IT systems, especially Windows, and thereby allows easy entry of characters in easyGUI. easyGUI supports 16 bit Unicode (basic multilingual plane), but not 32 bit (supplementary code planes).

Only 所有语言 Unicode version supports the Unicode character mode. This mode is far better suited for target systems requiring more than one character set. On the downside a Unicode mode target system requires a little more memory than an ANSI mode system. The 所有语言 Unicode version also supports ANSI character mode.

Each font in the system can support both character modes, but characters in the ANSI mode shadow character set are only visible if the project runs in ANSI mode, while Unicode characters are only visible in Unicode mode. Character codes in the 256 - 511 range are not the same in ANSI and Unicode modes, as they are mapped individually in the two modes. Only character codes in the 0 - 255 range are common to the two character modes:

ANSI mode:	0 - 31	32 - 255	256 - 287 <small>Do not use</small>	288 - 511 <small>Shadow characters</small>
Unicode mode:	<small>Do not use</small>	Windows ANSI	256 - 65535	

The character mode is selected in the Parameters window, on the Operation tab page, see later.



Character codes 0 - 31 should never be used for font characters.

TEXT FONTS

All text fonts are made after these guiding lines:

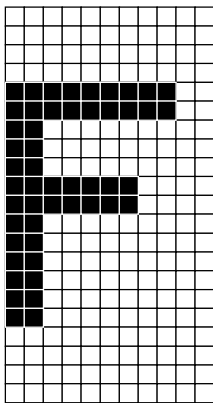
- Proportional writing.

- True hanging characters: The letter "g" is defined with the lower part going below the base line ("Eg", not "Eg"). This is however not a requirement, any font style can be created.
- ANSI/Unicode character codes as used in Windows. This makes it easier to exchange data between target system and PC.
- All kinds of characters, like e.g. Western world, Cyrillic, Japanese, can be placed in the same font, but must adhere to the limits of the character mode in use (ANSI or Unicode).

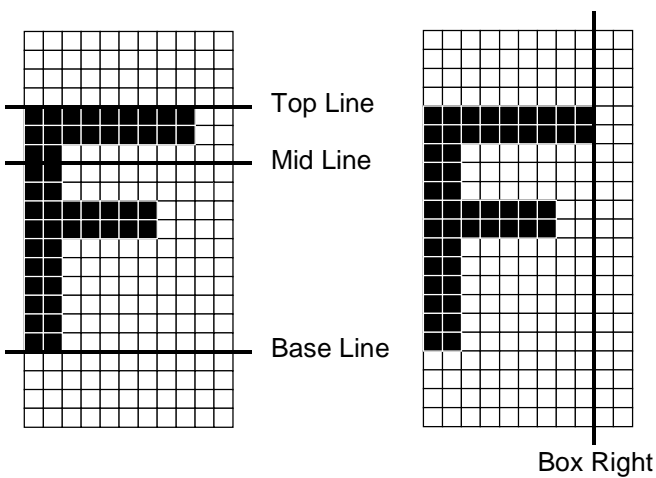
Character definition

A character is defined in a matrix of fixed coordinates, common for all characters in a font. An 11x21 text font is used as an example in the following chapters.

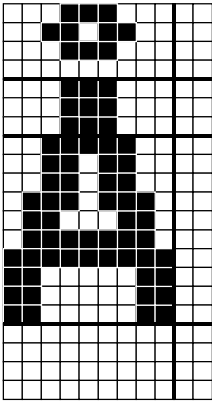
Each character is defined as a pixel pattern, e.g. a capital F:



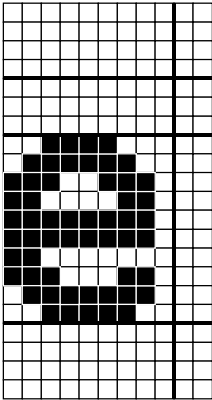
The character is placed according to certain fixed positions in the character cell. There are three horizontal and one vertical position:



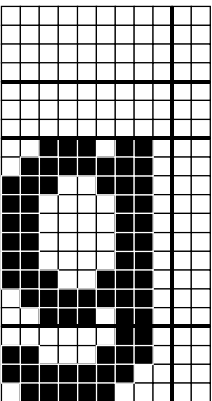
Capital letters should generally go from Base Line to Top Line. Letters with accents and the like at the top (e.g. "Å") utilizes the area above Top Line:



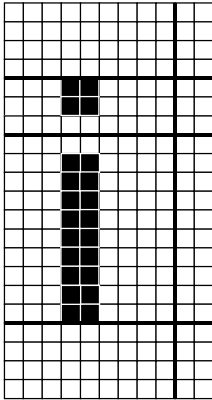
Lower case letters are placed between Base Line and Mid Line:



Lower case letters with hanging parts extends right down to the character cell bottom:

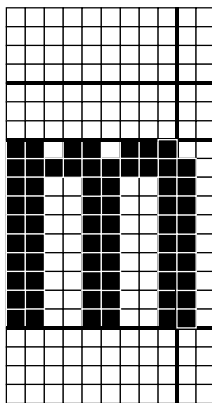


A character does not need to fill the area between the left border of the character cell and Box Right fully, and if not doing so it can be centered horizontally:



- or kept left adjusted. When easyGUI writes text in proportional mode the horizontal character placement inside the character cell doesn't matter. Only if writing with fixed spacing will the horizontal character placement be visible in the finished text.

A character may go beyond Box Right, but should not touch the right border of the character cell, unless it is intentional that the character shall be a continuous part of a following character, the character only will be used on its own (e.g. icons), or the font will only be used for proportional writing:



The various reference lines are only intended as a reference line when developing the font.

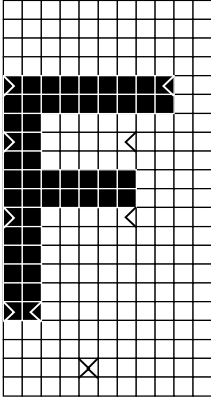
Proportional writing

All fonts can be used for proportional writing. There are three styles of writing:

- Fixed spacing. All characters are written with the same fixed width, equal to the font width (*Courier* style).
- Proportional. All characters are written with a width depending on the character size, and how it fits together with the previous character.

- Numerical proportional. As proportional writing, except that the characters "0" - "9", "+", "-", "*", "/", "=", " " (space) are written with fixed spacing. This writing style can be used when writing numerical values, especially in columns.

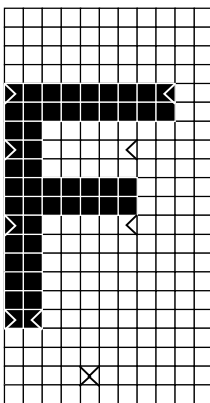
For each character a number of proportional position marks are defined. These marks are used when calculating horizontal character placements in proportional writing style. E.g. a capital "F":



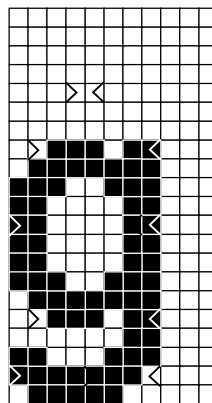
There are five horizontal pairs of marks, one on Top Line, one on Mid Line, one midway between Mid Line and Base Line, one on Base Line, and finally one pair 2/3 the way from Base Line to the bottom of the character cell. Each pair contains a left mark (>) and a right mark (<). These marks can be placed individually (horizontally) for each character in the font. When two characters shall be written next to each other proportionally the marks are used to calculate the distance between the characters. The characters are placed so that the left-marks of the second character keep a minimum distance in pixels to the right-marks of the first character. This minimum distance is common for all characters in the font, and is defined along with other basic font dimensions.

The proportional marks are not necessarily placed on the first or last used pixels in a pixel line. It is the overall shape of the character that determines where marks are placed.

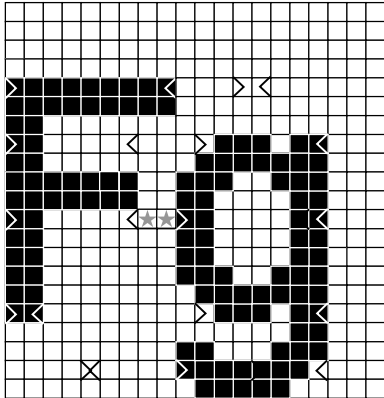
Proportional writing example: The text "Fg". "F" and "g" has the following proportional marks:



and



For this font the proportional distance is set to 2 pixels, so "g" is placed after "F" so that the two proportional marks coming closest to each other maintains a horizontal distance of two pixels. In this example it is the marks midway between Base Line and Mid Line, indicated by stars:



This system ensures reasonably nice proportional writing with a fast calculation routine, without demanding excessive resources from the target system by employing e.g. kerning tables.

Font style

As the displays used in embedded applications usually have limited resolution, it is difficult to develop nice Serif fonts, and the delivered fonts are therefore in Sans Serif style. Serif/Sans Serif: This is a Serif font: "Serif - Yes" (There are small attachments on the characters, e.g. the windows font "Times"), this is a Sans Serif font: "Serif - No" (e.g. the windows font "Arial"). If easyGUI is employed on a system with a bigger display resolution, or only limited amounts of text should be displayed, it is of course possible to develop a Serif font.

All text fonts in the system at the moment have the same Sans Serif overall style, as far as possible, but with some differences in the width-to-height ratio, to suit different purposes.

Undefined characters

If a character specified for writing isn't defined in the font, or isn't selected for the target system, it is shown as a filled-out block with the size of the font character cell.

FONT COMPRESSION

Fonts are saved in the target system C code in compressed form, in order to save space. For each character the individual scan lines (horizontal rows of pixels) are uncompressed, but only scan lines differing from the previous scan line are stored in the C code. Furthermore empty scan lines at the top and bottom of a character, and empty space to the left and right of the character, are not stored. This system occupies a little more space for small fonts because of the scan line accounting, but a lot less space for large fonts. Another advantage is that decompression execution time is minimal, actually shorter than uncompressed font writing in some cases.

CURRENT FONTS

The following list of fonts show what is currently in the easyGUI system. First text fonts are shown, then icon fonts. ANSI 2 is the normal text font:

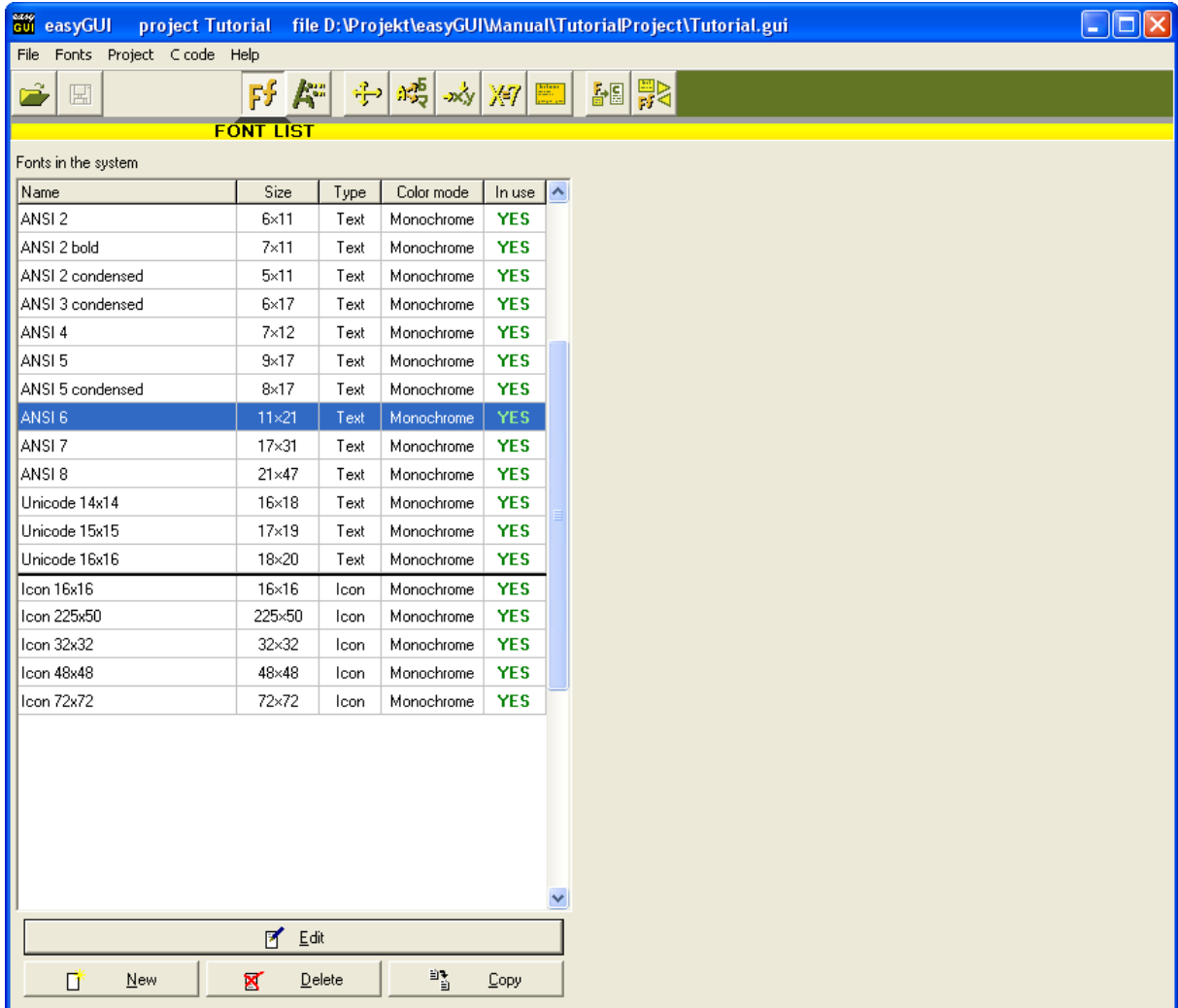
Font name	Size in pixels	Description
ANSI 2	6×11	Normal font. Used for most text. The character sizes are approximately the same as in old displays run in character mode with 8×8 matrices, allowing more text horizontally (if proportional writing is selected), and a little less vertically, because the characters occupy more than 8 pixels in height.
ANSI 2 bold	7×11	Bold font. Same height as ANSI 2, but with bolder characters. Can be used for emphasizing text parts, or for small headlines.
ANSI 2 condensed	5×11	Compressed font. Same height as ANSI 2, but with reduced character width. Should only be used if forced to do so, because it is a little hard to read, and does not have an appealing look.
ANSI 3 condensed	6×17	Compressed font. Same width as ANSI 2, but somewhat higher.
ANSI 4	7×12	Normal font. A little bigger than ANSI 2, can be used for headlines on small displays.
ANSI 5	9×17	Big font.
ANSI 5 condensed	8×17	Big font. A slightly compressed version of the ANSI 5 font.

ANSI 6	11×21	Big font. Can be used for headlines on medium sized displays.
ANSI 7	17×31	Very big font.
ANSI 8	21×47	Very big font. Can be used for e.g. splash screens.
Unicode 14x14	16×18	Compressed Unicode font. Asian characters are held within a 14x14 box, causing some characters to lose details. Also includes Cyrillic characters.
Unicode 15x15	17×19	Compressed Unicode font. Asian characters are held within a 15x15 box, causing a few characters to lose details. Also includes Cyrillic characters.
Unicode 16x16	18×20	Standard Unicode font. Asian characters are held within a 16x16 box. Also includes Cyrillic characters.
Icon 16x16	16×16	Very small icons.
Icon 32x32	32×32	Small icons.
Icon 48x48	48×48	Medium icons.
Icon 72x72	72×72	Big icons.
Icon 202x48	202×48	Very big icons. Can be used for e.g. Company logos.

Text fonts starting with "ANSI" contains only ANSI characters (character codes 32 - 255) and shadow characters. Text fonts starting with "Unicode" contain both ANSI, Unicode, and shadow characters.

Text and icon fonts are considered the like by easyGUI.

5 FONT LIST WINDOW

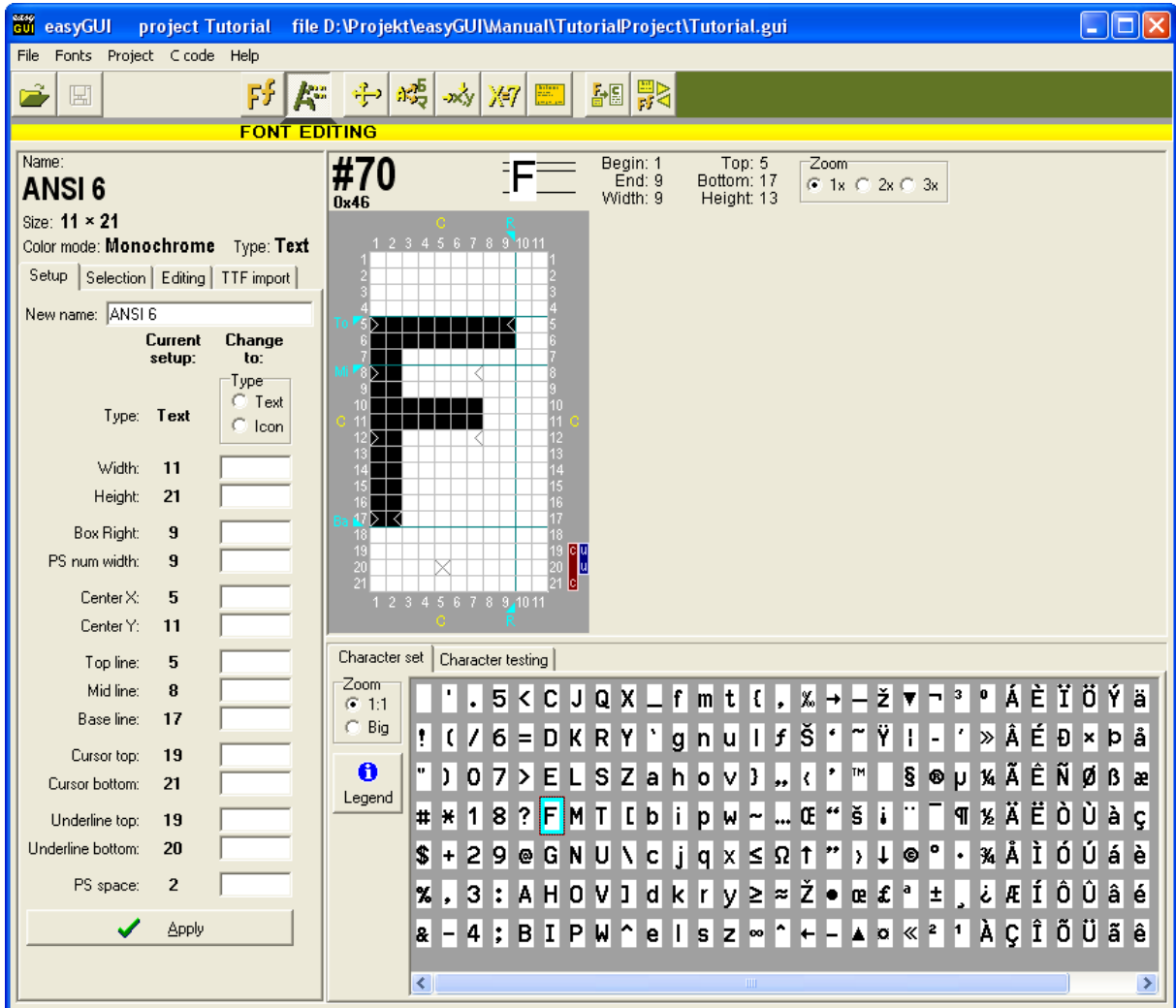


A list of available fonts in the system is shown. New fonts can be created, existing fonts erased, fonts copied, and a font selected for editing. Double-clicking on a font starts editing, just like pressing the **EDIT** button.

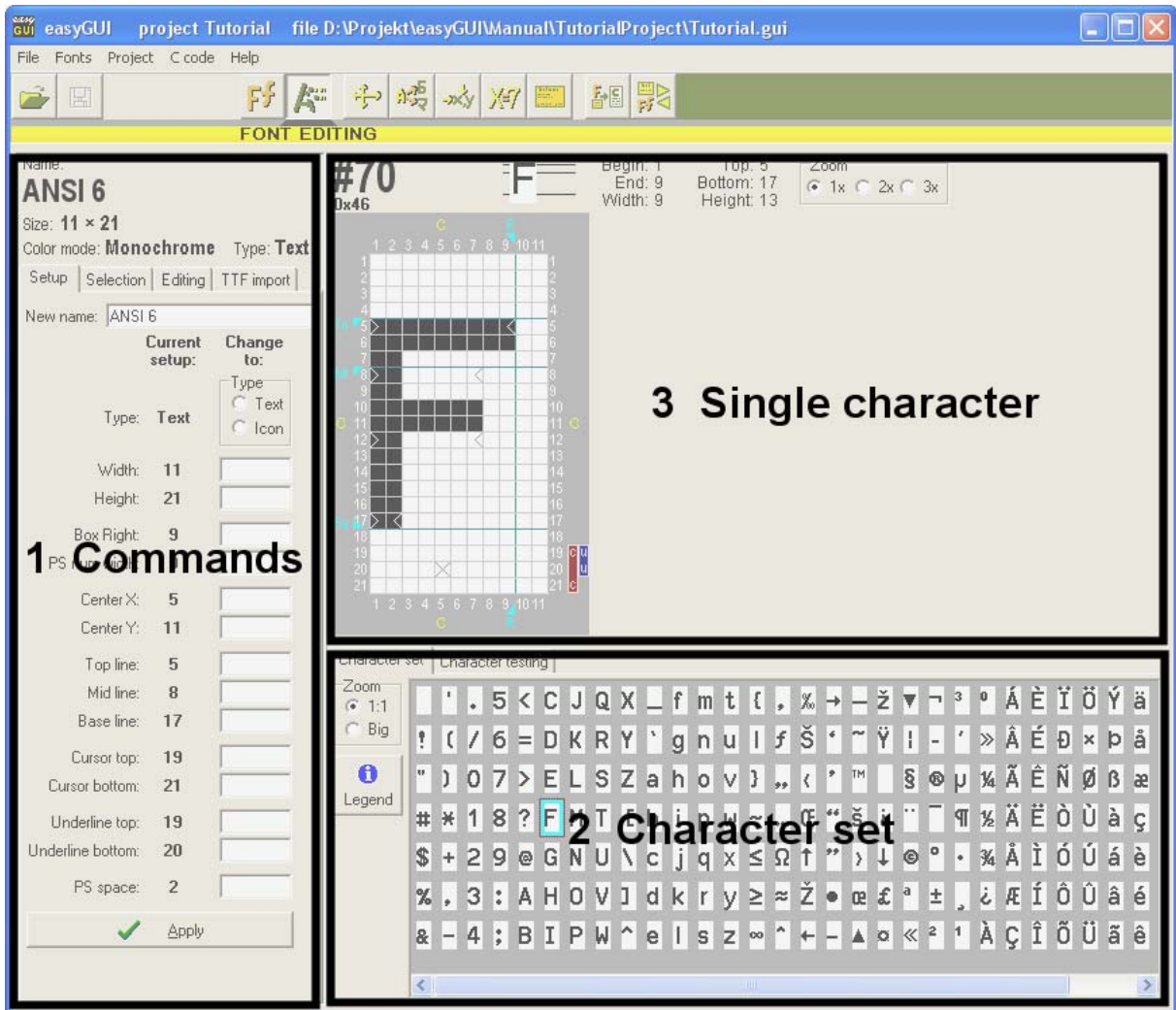
The last column indicates if the font is used in the project. Enabling just a single character for a font will show it as in use. Character enabling/disabling is done in the Font editing function.

6 FONT EDITING WINDOW

A single font is edited in this window:



There is a fair amount of controls, which will be explained in detail below. The window is divided into three parts:



Each panel handles a part of the editing process:

- 1 **Commands.** All commands for creating, handling, and selecting characters are placed here. Because of the number of commands the panel is subdivided into four tab pages: **Setup**, **Selection**, **Editing**, and **TTF import**.
- 2 **Character set.** Shows all characters in the current font. Also shows a test area, for checking proportional spacing in detail.
- 3 **Single character.** Shows the currently selected character.

FONT SETUP

The basic properties of the font are handled on the **Setup** tab page in the Commands panel. The following parameters can be edited:

- **New name.** Changes the font name. References to the font from structures in the project are not affected by a change of name, as the references are by internal pointers. Only deleting a font, and creating it again, will break the references.
- **Type.** A font can be one of two types:
 - **Text** for normal character fonts.
 - **Icon** for fonts containing icons. Internally the two font types are treated the same, the distinction is purely for convenience when presenting font lists.
- **Width** of character cell. Characters can be (indeed, normally is) smaller than this width, and the value merely sets the maximum character width possible. Value can be 1 - 255.
- **Height** of character cell. Characters can be (indeed, normally is) smaller than this height, and the value merely sets the maximum character height possible. Value can be 1 - 255.
- **Box right** limit. Value can be between 1 and width of character cell.
- **PS num. width.** Value can be between 1 and width of character cell. This width denotes how much horizontal space each numerical character takes up, when writing in PS numerical style. The specified width is only used for characters not written proportional in the PS numerical style, i.e. characters "0" - "9", "+", "-", "*", "/", "=", and " " (space).
- **Center X.** Value can be between 1 and width of character cell.
- **Center Y.** Value can be between 1 and height of character cell.
- **Top line.** Value can be between 1 and height of character cell.
- **Mid line.** Value can be between 1 and height of character cell.
- **Base line.** Value can be between 1 and height of character cell.
- **Cursor top.** Value can be between 1 and height of character cell. Not currently used.
- **Cursor bottom.** Value can be between 1 and height of character cell. Not currently used.
- **Underline top.** Value can be between 1 and height of character cell.
- **Underline bottom.** Value can be between 1 and height of character cell.
- **PS space.** Number of blank pixels between two adjacent characters when making proportional writing. Value can be between 1 and 99.

The current values are shown in the first column, while the second editable column is for new, revised values. Only the parameters where changes are wanted needs filling out.

Actual changes are made by pressing the **APPLY** button, which will check the entered values, and apply the changes. If the **APPLY** button is not pressed nothing changes.

In the case of changes to the width and height of the font a couple of special considerations arise:

- **Width.** Changes are made from the rightmost edge:
 - Widening the character cell adds blank pixels to the right. PS marks are not changed.
 - Narrowing the character cell removes pixels from the right, potentially truncating characters. PS marks are pushed to the left, if needed.
- **Height.** A small dialog window asks if the changes should be made from the top or the bottom. The action then commences, depending on the circumstances:
 - Enlarging the character cell adds blank pixels to the top or bottom, as selected. If changes are made to the top the Top line, Mid line, and Base line positions are shifted down accordingly. PS marks are not changed.
 - Making the character cell smaller removes pixels from the top or bottom, as selected, potentially truncating characters. If changes are made to the top the Top line, Mid line, and Base line positions are shifted up accordingly. PS marks are not changed.

If the desired operation is to enlarge or shrink the character cell evenly (or at some ratio) both at the top and bottom, it can be accomplished by making two height change operations.

FONT SELECTION

In order to save code space on the target system there is full control over which characters from which fonts will be included in the target system C code. This is handled on the **Selection** tab page in the Commands panel. Selection of characters can be done using four methods:

- **All characters in font.** An easy way of selecting all characters at once (or none).
- **All characters in use in structures.** Only characters used in structures are selected.
- **All numerical characters.** Selects characters "0" - "9", "+", "-", "*", "/", "=", and " " (space).
- **Manually selected characters.** The selection state of each character can be set either by double-clicking on a character in the character set, which toggles its selection state, or by using the five manual selection buttons:
 - **SELECT / DESELECT CURRENT CHARACTER.** Corresponds to double-clicking the current character in the character set.

- **SELECT RANGE OF CHARACTERS.** A small window allows setting the first and last character code to select.
- **SELECT ALL CHARACTERS.** A quick way of selecting everything, before possibly deselecting some characters.
- **DESELECT RANGE OF CHARACTERS.** A small window allows setting the first and last character code to deselect.
- **DESELECT ALL CHARACTERS.** A quick way of deselecting everything, before possibly selecting some characters.

The four methods are additive. If one of the methods selects a character, the character will be included in the target system data, no matter how the other methods treat the character. This means that selecting all characters through the **All characters in font** setting makes the other three methods redundant.

Previously in easyGUI

Previous versions of easyGUI (v5.1.5 and earlier) had their own window for selecting the font characters to include in the generated C files for the target system. This system has been changed, moving the font selection functionality into the font editing window.

View filter

In order to get an overview of the selection state, the characters shown in the character set can be filtered, using the settings in the **Character view filter** box:

- **All characters (no state indicators).** Essentially a clean representation of all characters in the font. Best used when making basic font editing, like adding and editing individual characters.
- **All characters (with state indicators).** Shows all characters in the font like the above setting, but with indicators showing the selection state of individual characters:



White background character. The character is selected, and will be included in the target system data.



Gray background character. The character is *not* selected, and will therefore be skipped when generating the target system data.



Character with red cross. The character has been manually deselected, and will only be included in the target system data if selected by other means. The example shows a white character background, which shows that the character *will* in fact be included in the target system data. The opposite (grayed character with a red cross) is also possible.



Character with green cross. The character is not used by any structures. The example shows a white character background, which shows that the character *will* in fact be included in the target system data. The opposite (grayed character with a green cross) is also possible.

This view filter is best used when changing character selection.

- **Currently selected characters.** Shows only the characters that will be included in the target system data.
- **All characters in use in structures.** Shows only the characters that are used in at least one text in the structures.
- **Manually selected characters.** Shows only the characters that have been manually selected.

At the bottom of the font selection panel is a statistics box showing character counts using various criteria.

FONT EDITING

Creation, deletion, and editing of characters take place both on the **Editing** tab page in the Commands panel, and in the single character panel to the right.

Pixel editing

Pixels can be edited directly in the single character panel. Clicking on a pixel toggles its color, white to black, or vice versa. Dragging with the mouse paints pixels as the mouse is moved, using the toggled color of the starting pixel.

PS mark editing

PS marks are moved by right-clicking and dragging horizontally. The vertical positions of PS marks are fixed, and defined as explained in the section on proportional writing above.

By using the **SET PS MARKS** and **RESET PS MARKS** buttons all PS marks for a character can be moved at once, see command explanations below.

Editing many characters at once

Many of the commands can work on a range of characters. This is controlled in the **Character range** panel to the right:

- **Current char.** Editing operations only work on the currently selected character, i.e. the character shown in the single character panel.
- **Range.** Editing operations work on the range of character codes indicated just below. All characters existing within the range (range limits included) are affected by editing commands.
- **All characters.** Editing operations work on all characters in the font.

For the latter two options a small warning (⚠) is shown, to remind of the potentially huge alterations to the font.

The character range setting works on the editing functions indicated by the gray lines going from command buttons to the Character range panel.

An alternative is to select a range of characters in the character set panel. Several characters can be selected, by dragging the mouse in the character set panel (a block), clicking on a character while holding down the Shift key (extend/shrink block), or clicking on a character while holding down the Ctrl key (include/exclude single character). This method of data selection is common in Windows. Please observe that the Character range panel setting takes precedence, so a range of characters selected in the character set panel is only useful if the Character range panel is set to Single character.

One character is always the active character (unless the font is empty). This character is shown in the single character panel.

Editing commands

A large number of commands are available in the Editing panel:

- **CREATE CHARACTERS.** A small window appears, allowing selection of the range of characters to create. Eventual characters already existing in the selected range are not affected. New characters are created blank, i.e. with all pixels set to off (white).
- **DELETE CHARACTERS.** A small window appears, allowing selection of the range of characters to delete. If a range of characters is currently selected they can be deleted instead.
- **CLEAR** blanks the current character, i.e. with all pixels set to off (white).




- **UNDO** reloads the character state from last time the project was saved.
- **INVERT** toggles all pixels of the current character, i.e. white pixels gets black, and vice versa.
- **HORIZONTAL MIRROR** mirrors the current character horizontally. PS marks are not moved.
- **VERTICAL MIRROR** mirrors the current character vertically. PS marks are not moved.
- **INSERT BITMAP** shows a file box, allowing selection of a graphical file. The file types must be a Windows bitmap file (`.bmp`). Pixels in the bitmap are treated as black or non-black, meaning that all pixels not purely black (RGB values = 0,0,0) are imported as white pixels. The imported bitmap can have any size, but easyGUI only uses the top left part matching the character size.
- Clipboard **COPY** copies the currently selected characters to both the Windows clipboard as a 24 bit color bitmap containing only black and white pixels (active character only), and to the internal easyGUI clipboard, allowing the characters to be pasted into another font.
- Clipboard **PASTE** imports the characters currently in the internal easyGUI clipboard. The imported characters can have any size, but easyGUI only uses the top left part matching the current font size.
- Move **UP**, **DOWN**, **LEFT**, and **RIGHT** rolls the current character pixels in the direction selected, with pixels spilling over an edge reemerging at the opposite edge. PS marks rolls left and right, but stops at edges.
- **SET PS MARKS** readjusts all PS marks so that they are touching the character. The function also takes pixel rows just above and below the PS row into consideration, in order to improve the selected PS mark positions. The PS marks can then be adjusted manually, if desired.
- **RESET PS MARKS** moves all PS marks to the left and right edges of the character box.
- **CHECK WHITE SPACE** controls the selected characters for sufficient white space at the edges of the character cell. This can be useful when creating large Asian fonts. In a quick action a range of characters can be checked for parts of characters extending out of the desired character box. Before starting the function the desired white space at the top, bottom, left and right edges is set. When pressing the **CHECK WHITE SPACE** button easyGUI will show the first character in the range which violates the criteria, or show an Ok message, if all selected characters passed the test. Nothing is edited with this function.
- **PREV** selects the character just before the current character. Characters can also be selected using the arrow keys on the keyboard, by clicking in the character set, or by using the **Search for character** field just below the **PREV** and **NEXT** buttons (character codes, both in decimal and hexadecimal, and direct characters, can be entered).
- **NEXT** selects the character just after the current character. Also see the **PREV** button above.

TTF IMPORT

Windows TTF fonts can be imported on the **TTF import** tab page in the Commands panel. Several parameters must be selected before import can start:

- **Character range.** Select the range of characters to import. Characters are created, if they don't exist already in the font. Only characters actually found in the selected TTF font will be created.
- **TTF font name.** Press the **SELECT TTF FONT** button to display a standard Windows font dialog. Select the desired font and size. The selected font / size is shown below in the white box.
- **Black/White ratio.** A Windows TTF font is vector based, and will be drawn using the full color depth of the Windows system. This results in many shades of gray being used to represent the individual character. As fonts in easyGUI are purely monochromatic (only two colors, or pixel states, on and off) there must be made some sort of conversion, reducing the gray-scale Windows characters to black-and-white easyGUI font characters. The black/white ratio slider determines how dark a pixel must be, in order for easyGUI to perceive it as black, and not white. Some fonts are best imported with the slider near the middle position, while others will be better represented if the slider is pushed more toward the right. It is easy to experiment, because the import can be repeated again and again, using slightly different slider settings. If the import covers a very large range of characters a smaller sub-range can be used when adjusting the slider, in order to speed up the response.

Examples - Windows TTF Mistral font, 16pt, normal font style:

Windows:	<i>ABC</i>
easyGUI import, 50% black/white ratio:	
easyGUI import, 70% black/white ratio:	
easyGUI import, 80% black/white ratio:	

In this example the best result is probably somewhere around 70% black/white ratio.

- **Vertical placement.** two options are possible for the vertical placement of imported characters:
 - **At font baseline.** easyGUI tries to place characters, so that capital letters are correctly placed at the font baseline. The letter "E" is used as a template.
 - **Other Y position.** The vertical placement can be selected manually.

It is advisable to start with the first option, and then shift to the manual placement, if the result is not as desired.

- **Only create characters existing in the font.** If this setting is checked the importer will only create font characters already existing in the TTF font. If unchecked all font characters in the indicated range will be created, no matter if they exist in the TTF font or not.

If importing e.g. Asian Unicode characters it is advisable to have this setting checked, as characters not existing in the TTF font are probably not usable in the Windows environment, and will therefore be difficult to enter in texts.

- The **IMPORT FONT** button executes the actual import / conversion process.

After import the font data must be controlled / adjusted, and PS marks must be set. This is the same process as if the font was created from scratch in the font editor. The proper sequence for importing TTF font data is:

- 1 Select font characteristics (name, size, style).
- 2 Make repeated imports, while adjusting the various parameters, especially TTF font size and black/white ratio, until a satisfactory result is achieved.
- 3 Move characters up/down so they are properly placed on the base line. This can be done *en masse* (i.e. use the character range panel on the Editing tab page).
- 4 Control each imported character, and turn pixels on and off as necessary to improve the result.
- 5 Set PS marks *en masse*.
- 6 Control PS marks for each imported character, and adjust as needed.

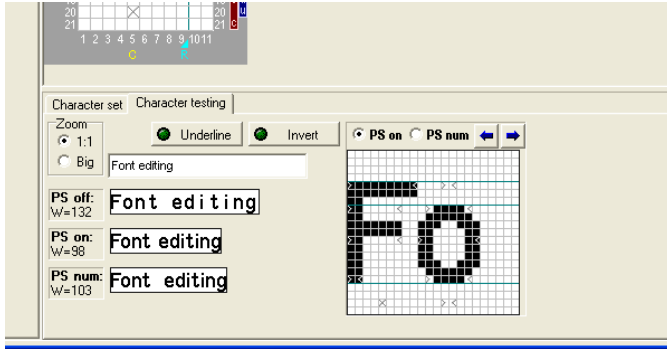
It is by far the easiest to import TTF characters into a font with a generous pixel dimensions, so that nothing is clipped. Later, when the correct sizes and settings have been determined, the font can be cut down in dimensions to a suitable size.

Also remember that the pixel dimensions of a font are not particularly important, if most text is written in transparent mode, with proportional spacing.

CHARACTER TESTING

Either the character set or a test function can be shown, by selecting the **Character set** or the **Character test** tabs.

The **Character test** tab page shows a number of fields and controls:



In the edit box a simple text can be entered ("Font editing" in the example above), and the result inspected in the three representations below (one for fixed spacing writing, and two for each type of proportional writing), and in the character pair window at right. The character pair window also shows the PS marks of both characters of a character pair. The two blue arrow buttons selects the starting character of the pair. The **PS ON** and **PS NUM** radio buttons selects one of the proportional writing styles for the character pair view.

Underlining and reversed writing can also be tested in detail.

7 PROJECT PARAMETERS WINDOW

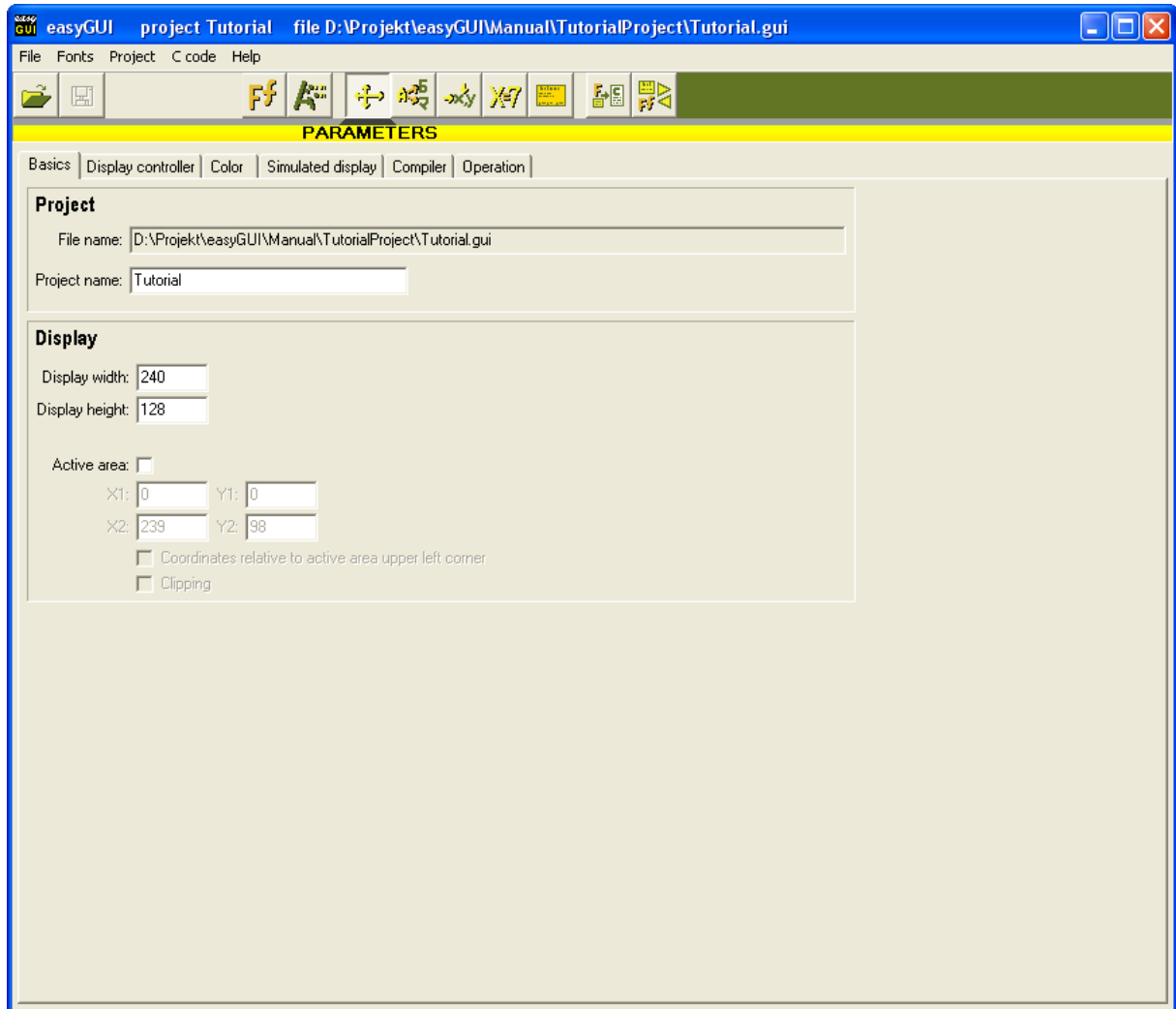
This window defines basic parameters in a project, relating to display and compiler. The parameters are subdivided into six tabbed pages, grouping the parameters logically. The pages are explained in the following sections.

A unit used extensively from this point is "bpp", or Bits Per Pixel. This parameter is a measure of the color depth of the display system. easyGUI can handle the following color depths:

1 bpp	Monochrome	Each pixel is either turned on or off.
2 bpp	Grayscale	4 shades of gray.
4 bpp	Grayscale / color	16 shades of gray, or 16 colors.
5 bpp	Grayscale	32 shades of gray. A special mode currently only used by the ST7529 display controller.
8 bpp	Grayscale / color	256 shades of gray, or 256 colors.
12 bpp	Color	4096 colors.
15 bpp	Color	32768 colors.
16 bpp	Color	65536 colors.
18 bpp	Color	262144 colors.
24 bpp	Color	16777216 colors. Same as TrueColor in Windows.

Monochrome version handles only 1 bpp.

BASICS



Project panel

- **File name.** Cannot be edited.
- **Project name.** For informative purposes.

Display panel

- **Display width** and **height** in pixels. Sets the basic dimensions of the target display. Only active pixels are counted, not border or overscan pixels.
- **Active area.** This function can be turned on and off using the checkbox. Turning it on allows definition of a part of the display as the active area. The rest is

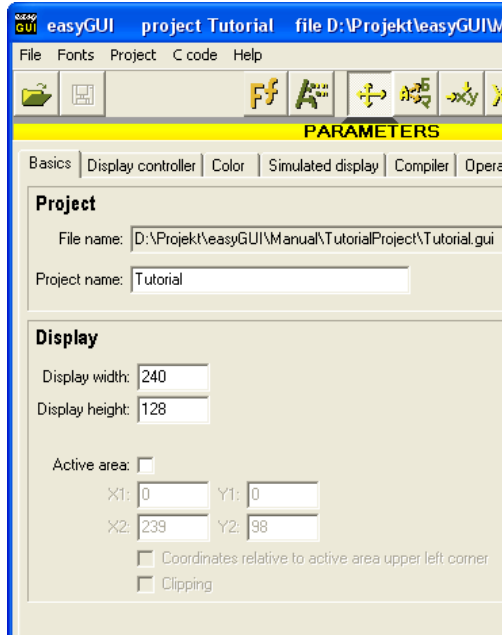
marked as inactive in the Structure editor, but drawing is still possible in the inactive area. The function is intended for target systems where part of the display is inaccessible, e.g. because it is covered by cabinet parts.

Examples:

Normal mode (active area unchecked), 240×128 pixels color display:

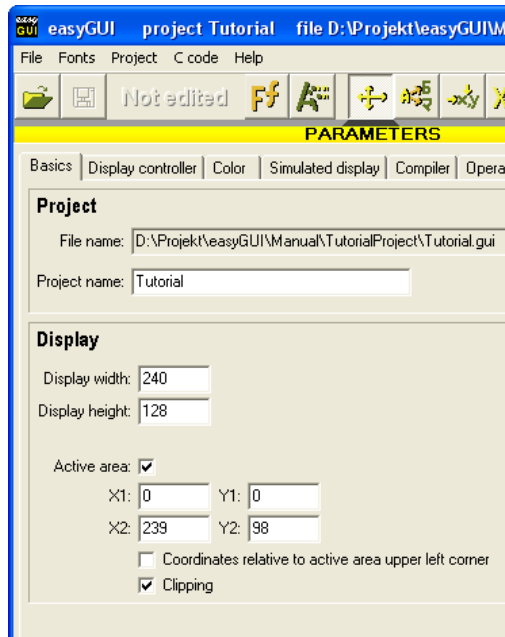
Settings:

Display will look like:



Activating the feature with the following parameters produces:

Settings:



Display will look like:



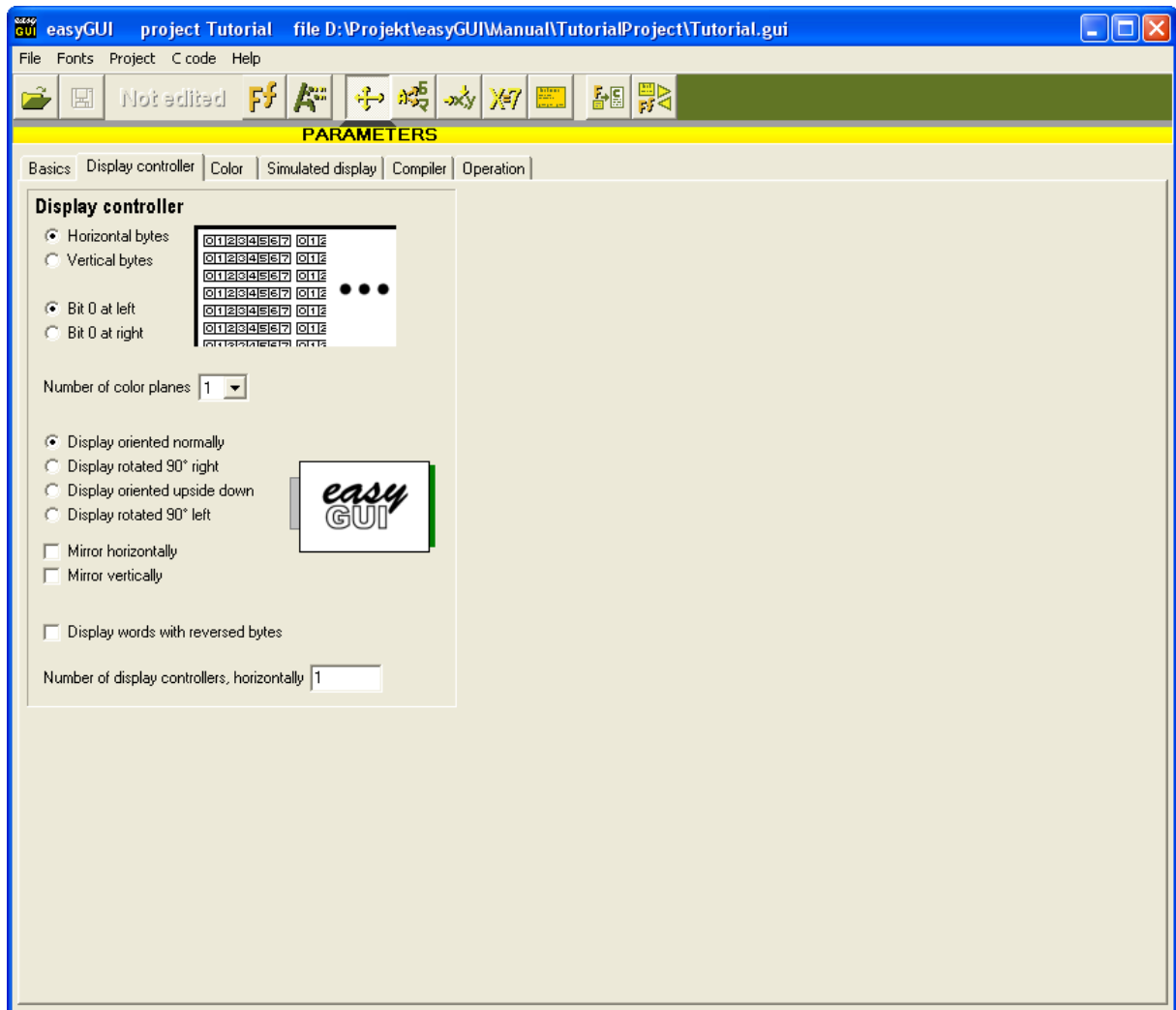
On the target system the display will look like:



- i.e. the inactive area of the display is clipped.

It is also possible to move the coordinate system, so that it starts at the active area upper left corner, instead of the normal display upper left corner.

DISPLAY CONTROLLER





Settings on this page define how display RAM is handled by the target system display controller. Furthermore the complete display image can be rotated in all four major directions, and mirrored in both directions, to facilitate mounting the display otherwise than initially intended by the display manufacturer.

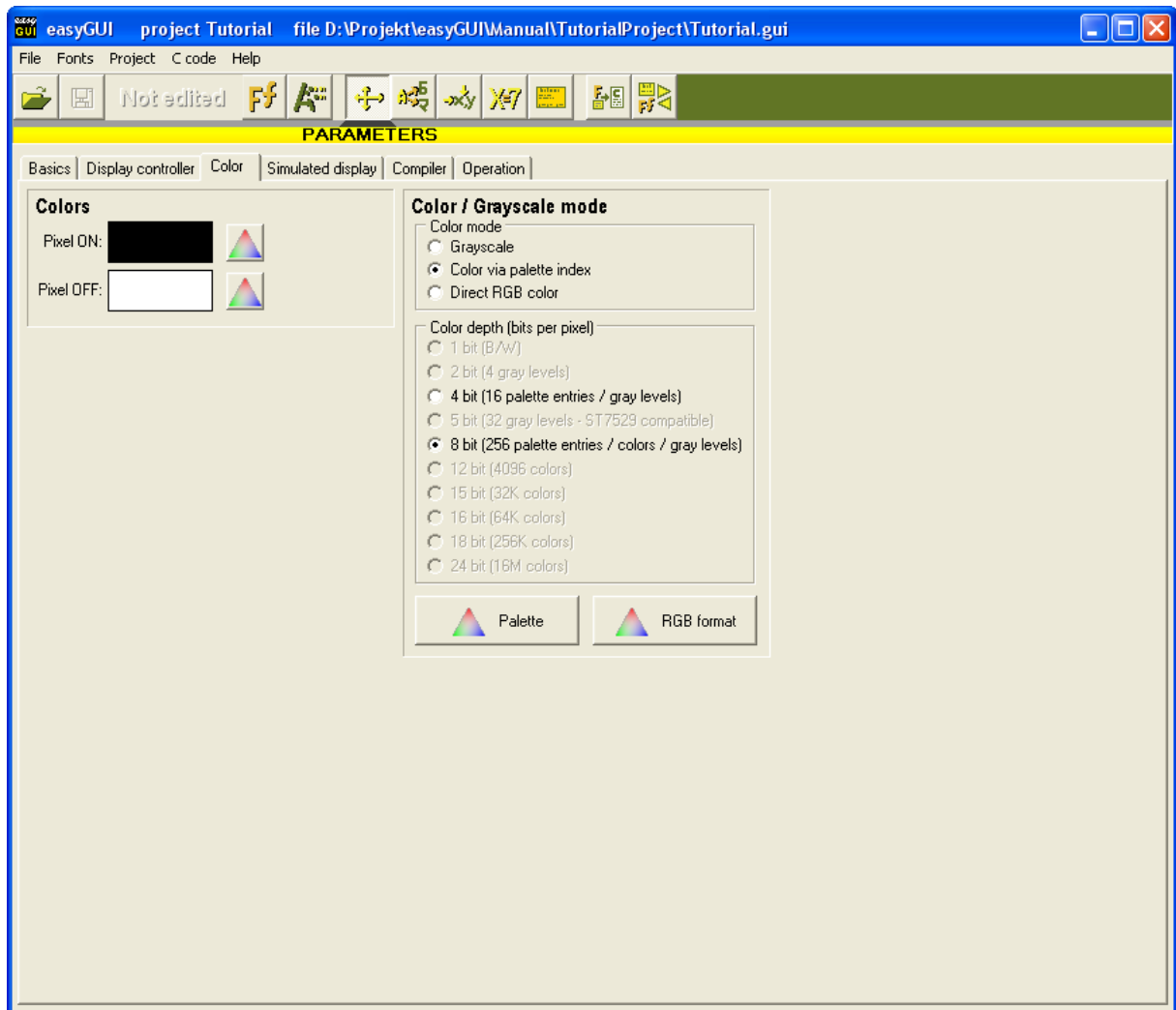
- **Byte orientation.** Set this setting according to the display controller in use. For displays with color depths of 5 bpp or higher the setting is irrelevant.
- **Bit orientation.** Set this setting according to the display controller in use. For displays with color depths of 5 bpp or higher the setting is irrelevant.
- **Color planes.** Some display controllers operates with several color planes in display RAM, effectively treating each plane as a monochrome image. Currently only one and two color planes are supported.
- **Display orientation.** The display contents can be oriented in the four primary directions: Normal, rotated 90° right, upside down, and rotated 90° left. This can

be utilized when mounting the display in other orientations than the one intended by the display manufacturer. Because easyGUI uses the display in a purely graphical way this can be done without penalties, except for a very small speed penalty when selecting orientations differing from normal.

Before deciding to use a display in abnormal orientations make sure that viewing of the display is satisfactory at the desired orientation. LCD displays can have very different contrasts when viewed from different directions. This can sometimes be used to advantage by rotating the display 180°, if the display is view from a direction not anticipated by the manufacturer. Conversely, this may prohibit e.g. 90° rotation of the display.

- **Mirroring.** The display contents may be mirrored both horizontally and vertically. Selecting both mirroring options results in an upside down image.
- **Display words with reversed bytes.** Only  Color and  Unicode versions. Some color displays with 16 bit words swaps the bytes in each word around. This setting can select both orientations. Only applicable to 4 bpp color depth with horizontal byte orientation, and 8 bpp color depth. Other display controller setups are unaffected.
- **No. of display controllers, horizontally.** Some displays uses more than one display controller horizontally, like when e.g. the Hitachi HD61202 controller (a 64×64 pixel controller) is used with 128×64 pixel displays.

COLOR





The color page controls how the display controller handles color modes.

Selecting the optimal color mode and color depth is a complex evaluation of needs in the user interface, capabilities of the selected display controller, available RAM and ROM, and processor power. Higher color depths puts increased demands on every aspect of the target system hardware, but also produces more pleasant results, increasing the quality feel of the product.



Colors panel


- **Pixel On** and **Pixel Off** colors.

Monochrome version: Used to make the display look like the real thing, when showing it in the easyGUI application. Has no influence on the target system.

 Color and  Unicode versions: Defines which colors are to be used when specifying Pixel ON and Pixel OFF colors for structure items. Default is black for Pixel ON, and white for Pixel Off. These colors are also used on the target system.


Color / Grayscale mode panel

Only  Color and  Unicode versions. Defines the type of color management used by the display controller. Can be:

- **Gray scale.** Can be used with from 1 bpp (2 color) to 8 bpp (256 color) color depths. No definition of how the colors are constructed is necessary for grayscale mode. easyGUI defines a range of gray colors ranging from purely white to purely black, with the number of colors defined by the next parameter, Color depth. Selecting Gray scale, and 1 bpp (2 color) color depth, corresponds to a monochrome display with only Pixel ON and Pixel OFF capability. This is the native mode of easyGUI  Monochrome version.
- **Color via palette index.** Can be used with 4 bpp (16 color) and 8 bpp (256 color) color depths. Each pixel on the display contains an index value, used by the display controller to look up the color in a palette table. easyGUI constructs the palette table based on the settings in this window, when generating c files for the target. The easyGUI palette table must be transferred to the display controller at target system startup.
- **Direct RGB color.** Can be used with from 8 bpp (256 color) to 24 bpp (16 million color) color depths. Each pixel on the display contains a direct color value with RGB values (Red, Green, and Blue) for the color.

Color depth panel

Only  Color and  Unicode versions. Defines the number of colors on the display. Can be:

- **1 bit (B/W).** Monochrome display mode, with only Pixel ON and Pixel OFF capability. This is the native mode of easyGUI  Monochrome version.
- **2 bit (4 gray levels).** Can show white, light gray, dark gray, and black pixels.
- **4 bit (16 palette entries / gray levels).** Can show 16 shades of gray, ranging from white to black, or 16 colors via a palette, with each color freely selectable.
- **5 bit (32 palette entries / gray levels).** Can show 32 shades of gray, ranging from white to black. This mode is special to the ST7529 display controller.
- **8 bit (256 palette entries / colors / gray levels).** Can show 256 shades of gray, ranging from white to black, or 256 colors via a palette, with each color freely selectable, or 256 colors directly as RGB values.

- **12 bit (4096 colors)**. Can show 4096 colors directly as RGB values.
- **15 bit (32K colors)**. Can show 32768 colors directly as RGB values.
- **16 bit (64K colors)**. Can show 65536 colors directly as RGB values.
- **18 bit (256K colors)**. Can show 262144 colors directly as RGB values.
- **24 bit (16M colors)**. Can show 16777216 colors directly as RGB values.

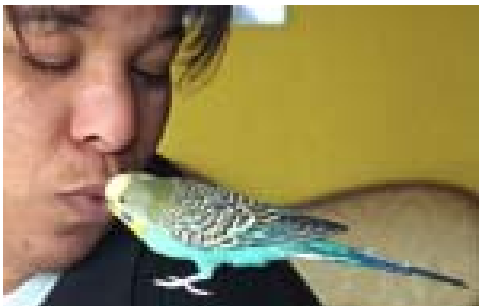
The type of display controller used on the target system determines which combinations of color modes and color depths can be used. Some controllers support only palette modes, some supports only 4 bpp and 8 bpp pixel color depths, and so on. The possible combinations in easyGUI are:

Support	Gray scale	Palette	RGB
1 bpp (2 colors)	Ok	Not possible	Not possible
2 bpp (4 colors)	Ok	Not possible	Not possible
4 bpp (16 colors)	Ok	Ok	Not possible
5 bpp (32 colors)	Ok	Not possible	Not possible
8 bpp (256 colors)	Ok	Ok	Ok
12 bpp (4096 colors)	Not possible	Not possible	Ok
15 bpp (32K colors)	Not possible	Not possible	Ok
16 bpp (64K colors)	Not possible	Not possible	Ok
18 bpp (256K colors)	Not possible	Not possible	Ok
24 bpp (16M colors)	Not possible	Not possible	Ok

The various combinations have different strengths and weaknesses, when used for ordinary text, simple bitmaps icons with few colors, and continuous tone bitmaps:

Quality	Text	Simple bitmaps / icons	Continuous tone bitmaps
1 bpp grayscale (2 shades)	High	Low	Not suitable
2 bpp grayscale (4 shades)	High	Medium	Low
4 bpp grayscale (16 shades)	High	High	Medium
5 bpp grayscale (32 shades)	High	High	Medium
8 bpp grayscale (256 shades)	High	High	High
4 bpp palette (16 colors)	High	High	Low
8 bpp palette (256 colors)	High	High	Medium
8 bpp RGB (256 colors)	High	High	Low
12 bpp RGB (4096 colors)	High	High	Medium
15 bpp RGB (32K colors)	High	High	Medium
16 bpp RGB (64K colors)	High	High	Medium
18 bpp RGB (64K colors)	High	High	High
24 bpp RGB (16M colors)	High	High	High

As an example on continuous tone bitmap quality the following bitmap (150×95 pixels) -



- is shown in the various color modes / depths:

- **1 bpp grayscale (2 shades):**



All colors are converted to either black (<50% gray) or white.

Another approach is to use an error diffusion raster:



However, this technique is outside the scope of easyGUI, and must be handles by a dedicated graphical editing application.

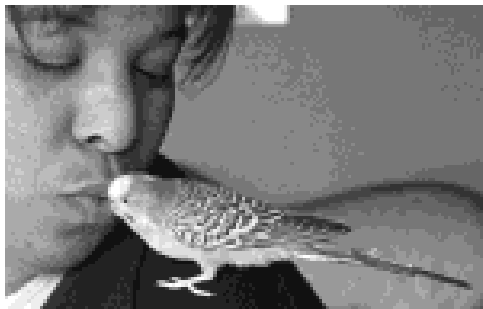
- **2 bpp grayscale (4 shades):**



With only four shades of gray (and the two of them black and white) the quality of the picture is not impressive. The palette is fixed as:



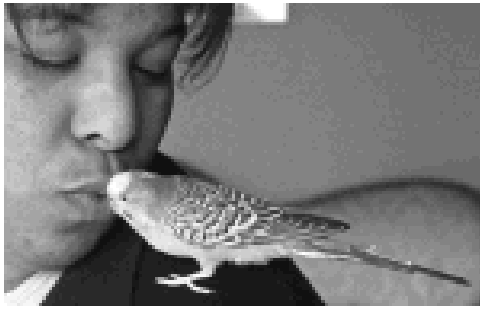
- **4 bpp grayscale (16 shades):**



16 shades of gray result in a much nicer bitmap. The palette is fixed as:



- 5 bpp grayscale (32 shades):



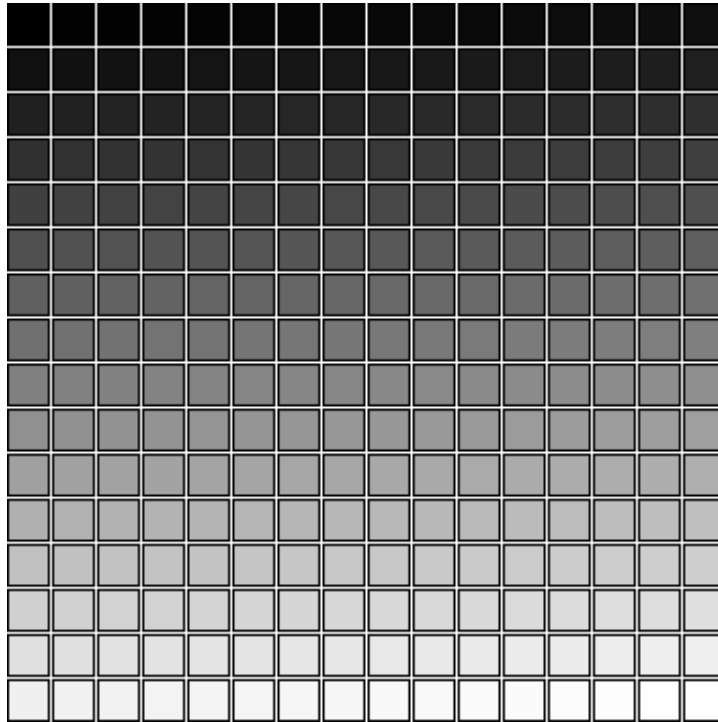
32 shades of gray result make an even better bitmap. The palette is fixed as:



- 8 bpp grayscale (256 shades):



256 shades of gray produce a high quality black & white bitmap. The palette is fixed as:



- **4 bpp palette (16 colors):**



As only 16 colors are available for the entire color space the quality is not suited for continuous tone bitmaps. The palette used here is the standard easyGUI 16 color palette:



The palette can be edited. The quality can be substantially raised by employing a palette specially suited for the bitmap:



This changes the picture to:



- but this approach is seldom practical, because a specialized palette is seldom useful for more than one image. An exception is when employing company logo colors for various graphical elements.

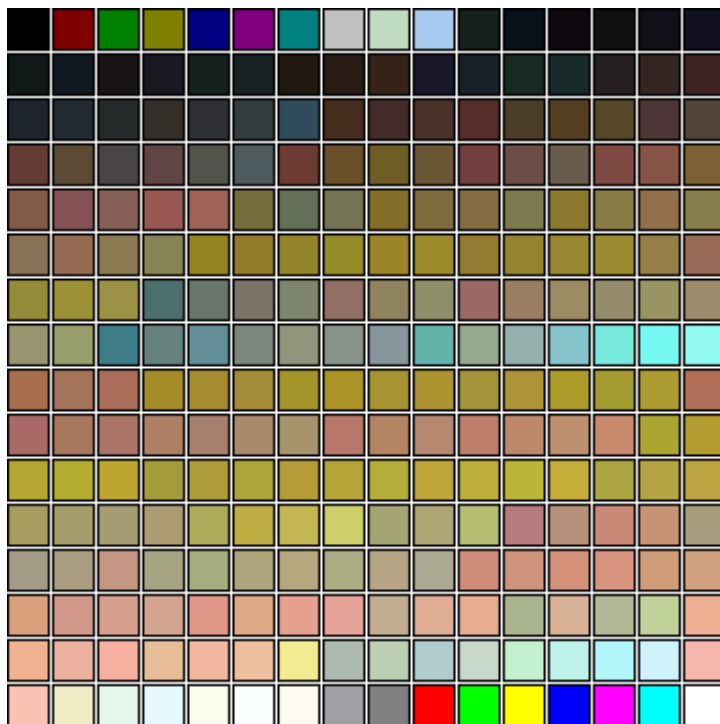
- **8 bpp palette (256 colors):**



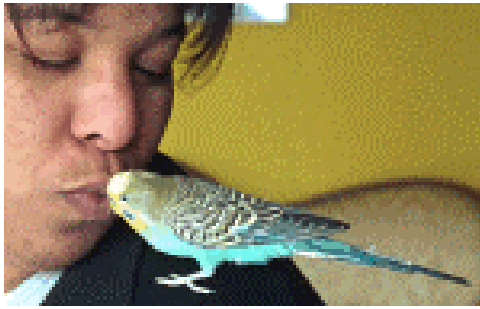
Still not a perfect bitmap, but much better than the 4 bpp mode with standard palette. The palette used here is the standard easyGUI 16 color palette:



The palette can be edited. Again the palette can be optimized for this particular bitmap:



- creating an almost perfect bitmap:

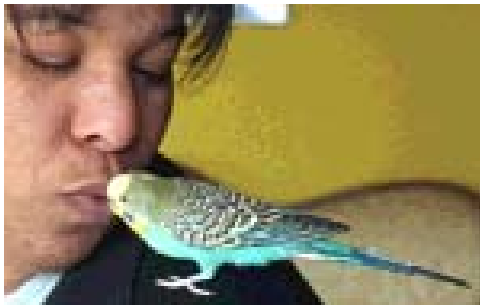


- **8 bpp RGB (256 colors):**



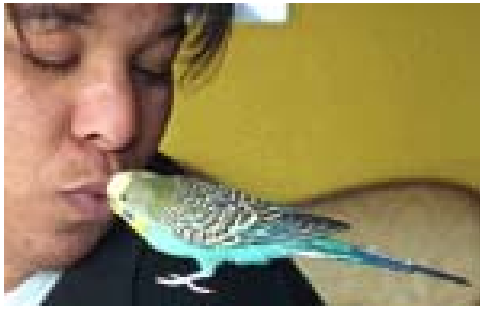
This bitmap was created using 3 bits for red, 3 bits for green, and 2 bits for blue intensity. The actual assignment of bits depends on the display controller.

- **12 bpp RGB (4096 colors):**



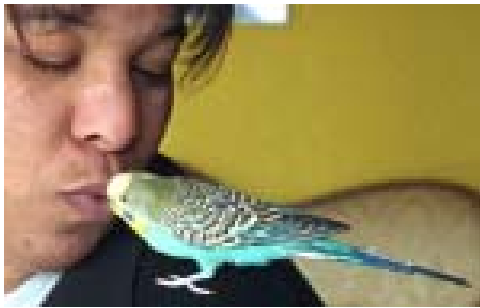
This bitmap was created using 4 bits for each of the RGB color intensities.

- **15 bpp RGB (32K colors):**



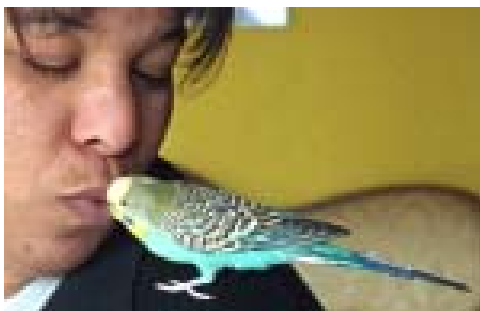
This bitmap was created using 5 bits for each of the RGB color intensities.

- **16 bpp RGB (64K colors):**



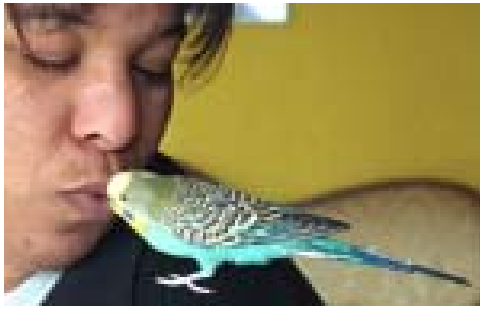
This bitmap was created using 5 bits for red, 6 bits for green, and 5 bits for blue intensity. Generally the green color should receive the most bits, if an even split between the three primary colors is not possible, because the human eye is most sensitive to yellow and green colors. One bit extra for green, compared with the 15 bpp example above might not seem like much, but it is visible in the example at the upper right corner, where the 16 bpp bitmap produces a more smooth transition than the 15 bpp bitmap.

- **18 bpp RGB (256K colors):**





This bitmap was created using 6 bits for each of the RGB color intensities. The quality is now really good, but can still be improved.

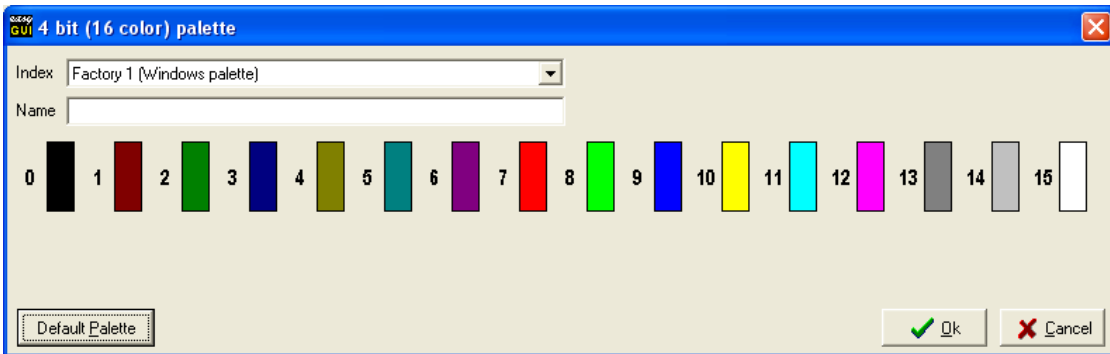
- 24 bpp (16M colors) RGB:



This bitmap was created using 8 bits for each of the RGB color intensities. This is the highest quality handled by easyGUI.

Palette handling

Only  Color and  Unicode versions. Permits editing of the 4 bpp and 8 bpp palettes (4 bpp palette shown as example):



The Index drop-down box allows selecting between several palettes. For 4 bpp palettes they are:

- **Factory 1 (Windows palette)**



- **Factory 2 (RGB=4/2/2 levels)** - i.e. all combinations of 4 red levels, 2 green, and 2 blue.



- **Factory 3 (RGB=2/4/2 levels)** - i.e. all combinations of 2 red levels, 4 green, and 2 blue



- **Factory 4 (RGB=2/2/4 levels)** - i.e. all combinations of 2 red levels, 2 green, and 4 blue



- **User defined 1.** Any color can be assigned to any color index. Initially all colors are black.
- **User defined 2.** Any color can be assigned to any color index. Initially all colors are black.
- **User defined 3.** Any color can be assigned to any color index. Initially all colors are black.

For 8 bpp palettes they are:

- **Factory 1 (RGB=6/6/6 levels + grayscale + spare)** - i.e. all combinations of 6 red levels, 6 green, and 6 blue, a 32 shade gray scale section, and finally 24 spare color indices, which are initially black.



- **Factory 2 (RGB=8/8/4 levels)** - i.e. all combinations of 8 red levels, 8 green, and 4 blue.



- **Factory 3 (RGB=8/4/8 levels)** - i.e. all combinations of 8 red levels, 4 green, and 8 blue.



- **Factory 4 (RGB=4/8/8 levels)** - i.e. all combinations of 4 red levels, 8 green, and 8 blue.



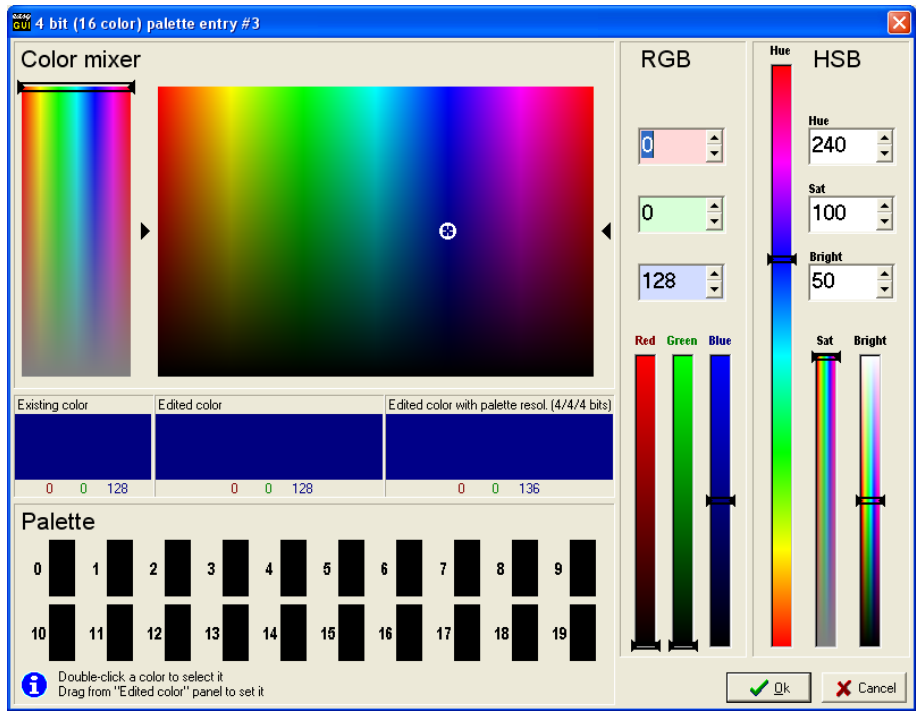
- **User defined 1.** Any color can be assigned to any color index. Initially all colors are black.
- **User defined 2.** Any color can be assigned to any color index. Initially all colors are black.
- **User defined 3.** Any color can be assigned to any color index. Initially all colors are black.

Each palette can be given a name, which is only for informational purposes.



The **DEFAULT PALETTE** button replaces all colors with the standard easyGUI palette colors for the palette in question - for the user defined palettes all colors will be reset to black.

Color handling



Each color in easyGUI can be edited by double-clicking it. A single color editing window appears:

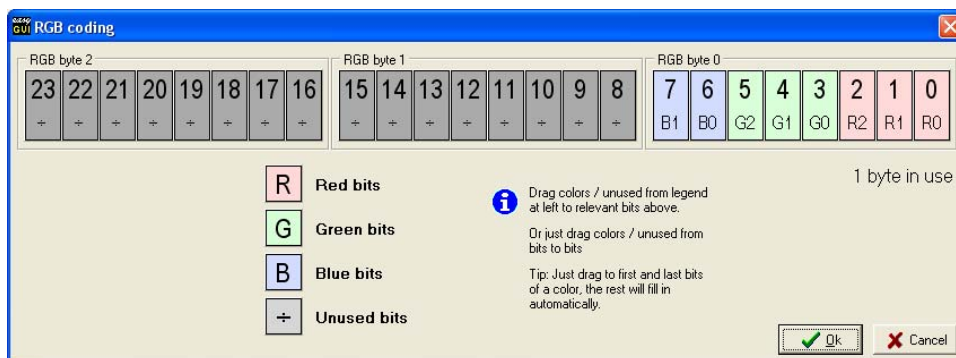


The color can be edited in a number of ways:

- **Color mixer.** Works by setting color based on hue, saturation, and brightness. Use the  slider to select color saturation, and the  field to select hue (horizontal direction) and brightness (vertical direction).
- **RGB values.** The red, green, and blue intensities (0-255) are entered directly as numerical values.
- **HSB values.** The hue (0-360), saturation (0-100), and brightness (0-100) are entered directly as numerical values.
- **Existing color.** Shows the color as it was before editing started.
- **Edited color.** Shows the current state of the color.
- **Edited color with palette resolution.** Shows the current state of the color, using the current color mode and depth (bits per red, green and blue color shown in parenthesis). This field is only shown if applicable.
- **Palette colors.** These colors are 20 colors that can be easily used and copied across the system. This internal palette must not be confused with the 4 bpp and 8 bpp color depth palettes described in the previous sections. To set the color being edited to one of the palette colors just double-click the palette color. To set the palette color drag the color from the Edited color field just above the palette using the mouse. These palette colors need not be set to anything, they are just meant as an easy way of selecting the same color for many items.

RGB format

Only  Color and  Unicode versions. Specifies how the target system display controller handles color information in display RAM. A window for color bit definitions is shown:



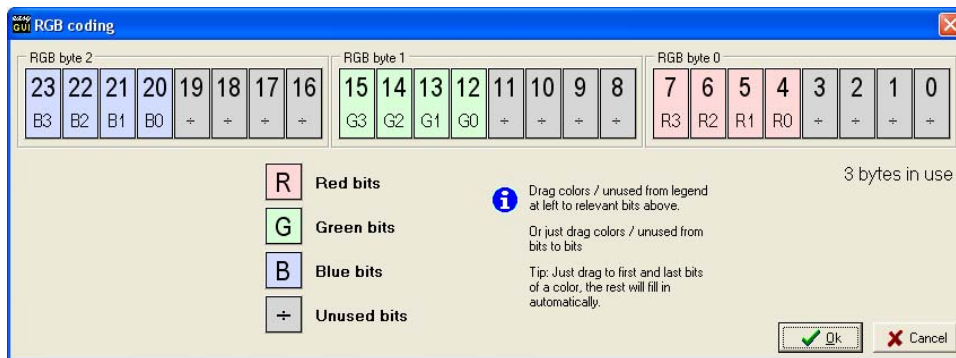
Bits for the three primary colors red, green, and blue, can be placed in the RGB bytes. Three bytes are shown, but how many bytes are actually in use depends on color depth and display controller type. In the example above RGB coding for an 8 bpp display controller mode is shown. Three red, three green, and two blue color bits has been placed in the first RGB byte.

The setup in this window is remembered for each combination of color mode and color depth where it is applicable (not gray scale modes). For palette modes the setup is used

to define the layout of the palette table colors, while for RGB modes it is used for defining the actual pixel bytes in display RAM.

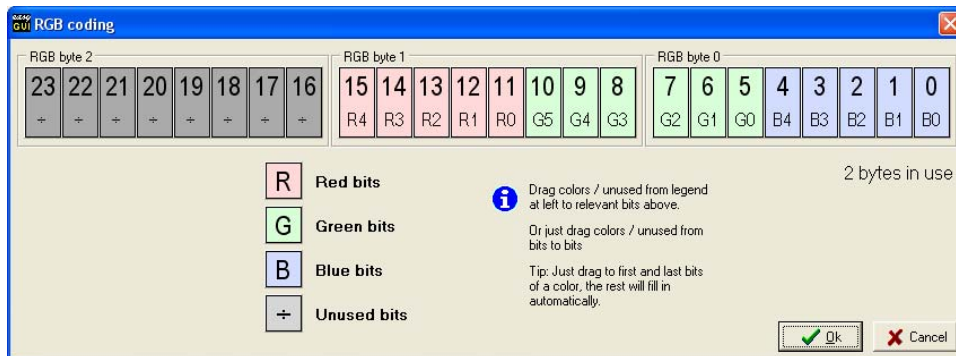
A bit is set to a color by dragging from the bit legends at lower left to the desired bit with the mouse. Alternatively colors can also be dragged from bit to bit in the three display bytes. Dragging the same color to a second bit automatically fills out any intermediate bits with the same color, i.e. bits for a particular color are always consecutive. Color bits can be erased again by dragging from the Unused bits legend, or from a gray bit to the desired bit location.

The number of display bytes in use does not necessarily correspond to the selected color depth. An example is:



This particular display controller uses three bytes for each color, despite the fact that the color depth is only 12 bits (three each of red, green, and blue). In reality the display controller only has 4 bit registers for each color, but they are accessed as bytes on distinct addresses, and therefore are considered individual bytes by the microprocessor.

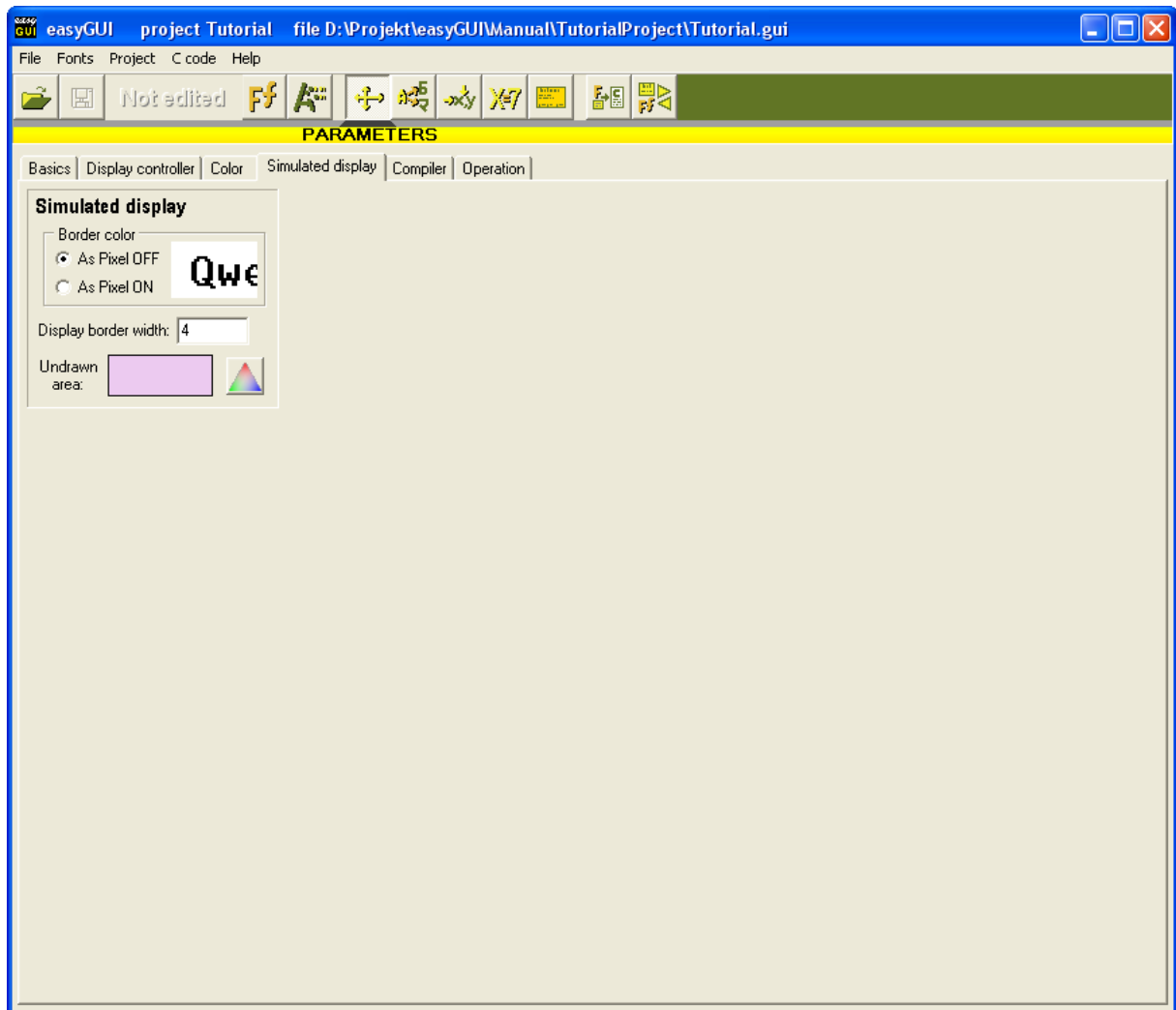
A third example is:



This display controller requires two bytes for the 15 bpp display depth shown, but the blue color is placed on the lowest bits, followed by the green and the red colors.

How the colors are arranged in a particular display controller, operating at a particular color mode and color depth, must be found in the documentation for the display controller. Do not despair if your particular display controller is impossible to configure in easyGUI. The display industry is in continuous development, and it is therefore difficult to cover all possible setups. Contact easyGUI support if you encounter trouble in the setup process.

SIMULATED DISPLAY

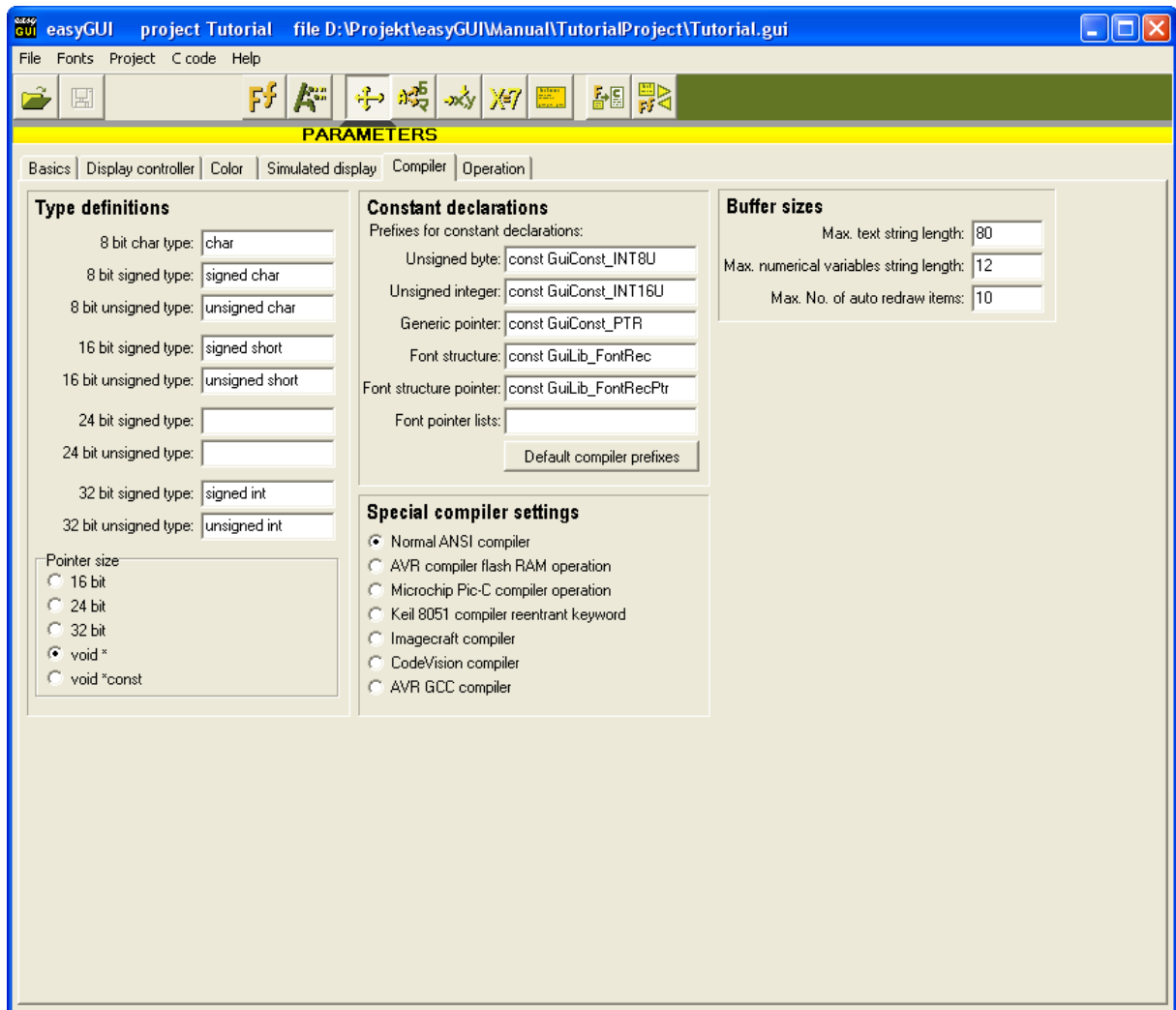


Settings on this page only affect the visual representation of the display inside the easyGUI environment, in order to more accurately reflect the "real thing". The selections are not transferred to the target system.

- **Border color.** Select between light and dark border. Most displays use light color for the border area (also called the overscan area), but some displays are dark in this area. Normally this is not user selectable, but depends on the technology used in the display. It is important to select the correct setting here, because a dark border makes it necessary for all dark text on light background to stay clear of the border with at least one pixel to avoid the text "gluing" to the border, where a light border does not have this problem. The border color thus somewhat affects the layout of the user interface. If light text on dark background is used the problem is of course reversed.
- **Display border width.** The border area is the visible area around the active pixels in the display. Measured in pixels. A sensible value for most displays is 3 or 4 pixels.

- **Undrawn area color.** Used by easyGUI to indicate areas of the display not touched by a particular screen structure. Set this color to a deviating color to enhance its visibility. In many instances it is nice to be able to see which areas of the display are drawn on by the structure. Without the Undrawn color area this is invisible, if drawing with the background color.

COMPILER



Settings on this page concerns the C compiler used on the target system. Unfortunately the various C compilers in use in the embedded world don't comply 100% to the same standard. Even if a particular compiler states that it adheres to the ANSI X3.159-1989 Standard C convention, it is not guaranteed to work without the need for some tweaks to these settings. It is therefore necessary for easyGUI to know the syntax for various subjects of the compiler. Furthermore, some buffer sizes are determined here.

Type definitions panel

- **8 bit char type.** Default is `char`.
- **8 bit signed type.** Default is `signed char`.
- **8 bit unsigned type.** Default is `unsigned char`.
- **16 bit signed type.** Default is `signed short`.
- **16 bit unsigned type.** Default is `unsigned short`.
- **24 bit signed type.** The 24bit variables are only used for compilers using a 24 bit addressing space (e.g. 8086 family processors). For other compilers the two 24 bit fields should just be left empty. Default is empty.
- **24 bit unsigned type.** Default is empty.
- **32 bit signed type.** Default is `signed long`.
- **32 bit unsigned type.** Default is `unsigned long`.
- **Pointer size.** Can be set to 16 bit, 24 bit, and 32 bit pointers. Two additional choices are `void *` and `void *const`. Most compilers are happy with the `void *` setting. It is very important that this setting is correct.

Constant declarations panel

The four prefix strings are inserted into the GuiStruct and GuiFont c & h files. The default values are:

- **Unsigned byte:** `const GuiConst_INT8U`
- **Unsigned integer:** `const GuiConst_INT16U`
- **Generic pointer:** `const GuiConst_PTR`
- **Font structure:** `const GuiLib_FontRec`
- **Font structure pointer:** `const GuiLib_FontRecPtr`
- **Font pointer lists:**

Reasons for changing them can be e.g. special code for Flash RAM systems.

The last setting is empty as default, but by setting it to "const" when using Keil compilers the placement of the font pointer list in RAM instead of flash memory can be prevented.

All the settings can be reset to the default by pressing the **DEFAULT COMPILER PREFIXES** button.

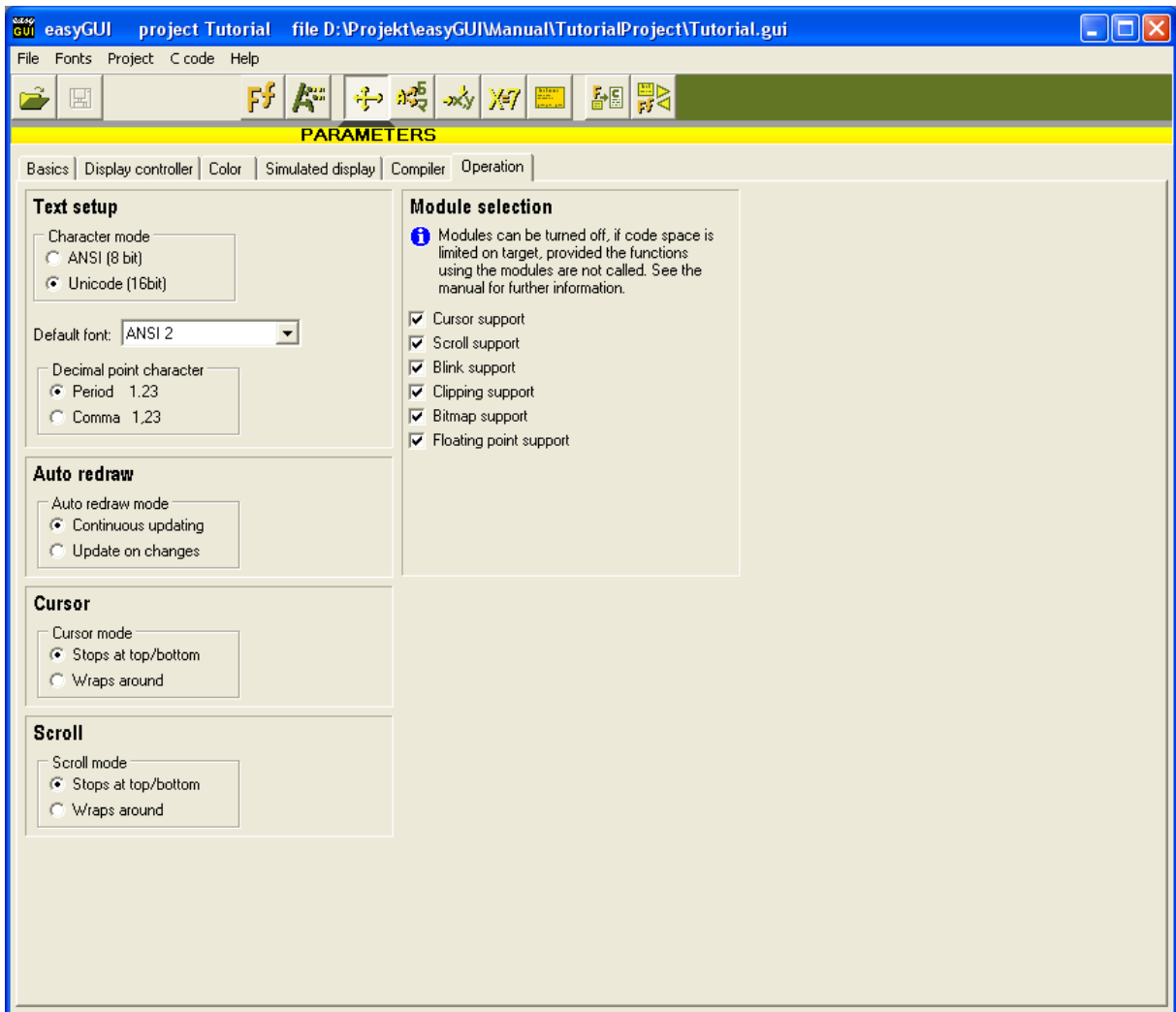
Special compiler settings panel

- **Normal ANSI compiler.** This is the default setting, which shall be used for X3.159-1989 Standard C compliant compilers.
- **AVR compiler flash RAM operation.** Use this setting if the AVR compiler is used, and RAM is of flash type. The flag sets some special settings in the easyGUI library, enabling use of flash RAM in the AVR development environment.
- **Microchip Pic-C compiler operation.** Use this setting if one of the Microchip Pic-C compilers is used. The flag inserts `rom` qualifiers where needed in the easyGUI library.
- **Keil 8051 compiler reentrant keyword.** Adds the keyword `reentrant` to all recursively called functions in the easyGUI library. If this setting is not used easyGUI will typically display graphics primitives and simple screen structures correctly, but fail to display complex screen structures.
- **Imagecraft compiler operation.** Use this setting if one of the Imagecraft compilers is used. The flag inserts `const` qualifiers where needed in the easyGUI library.
- **CodeVision compiler operation.** Use this setting if one of the CodeVision compilers is used. The flag inserts `flash` qualifiers where needed in the easyGUI library.
- **AVR GCC compiler operation.** Use this setting if the GCC AVR compiler is used. The flag inserts `PROGMEM` qualifiers where needed in the easyGUI library. Observe that only monochrome displays are supported with this compiler, due to the very small RAM sizes possible.

Buffer sizes panel

- **Max. text string length.** Determines buffer size in the target code for text writing. Enlarging the buffer permits longer text to be handled by easyGUI, but consumes more memory.
- **Max. numerical variables string length.** Determines buffer size in the target code for writing variables on screen. Enlarging the buffer permits longer variables (text representation) to be handled by easyGUI, but consumes more memory.
- **Max. No. of auto redraw items.** Determines how many auto redraw items easyGUI can handle simultaneously. A higher number consumes more memory.

OPERATION



Settings on this page determine miscellaneous parameters of the easyGUI library operational mode.

Text setup panel

- **Character mode.** Only 所有语言 Unicode version. Selects between ANSI mode (8 bit character codes) and Unicode mode (16 bit character codes).
- **Default font.** All new text items in structures use this font, until something else is selected (something else than the "No change" setting).
- **Decimal point character.** Used when displaying decimal numbers. Can be period (American style) or comma (Continental European style).

Auto redraw panel

- **Continuous updating.** All Auto redraw items are continuously updated, each time the `GuiLib_Refresh` function is called. This is the default setting, and the mode used by easyGUI before this Auto redraw mode parameter was implemented.
- **Update on changes.** Auto redraw items are updated only if the controlling variable / variable to be displayed has changed, or if the item does not involve a variable (not very useful).

Cursor mode panel

- **Stops at top/bottom.** When navigating cursor fields on the target system it is not possible to jump from the last cursor field to the first, when issuing the cursor down command, and vice versa. In the Structure editor the selected cursor fields always wrap around when testing the visual behavior.
- **Wraps around.** The opposite setting, when navigating cursor fields on the target system it is possible to jump from the last cursor field to the first, when issuing the cursor down command, and vice versa.

Scroll mode panel

- **Stops at top/bottom.** When navigating scroll boxes on the target system it is not possible to jump from the last scroll line to the first, when issuing the scroll down command, and vice versa. In the Structure editor the selected scroll line always wraps around when testing the visual behavior.
- **Wraps around.** The opposite setting, when navigating scroll boxes on the target system it is possible to jump from the last scroll line to the first, when issuing the scroll down command, and vice versa.

Module selection panel

Various parts of the easyGUI target library can be disabled out, by un-checking these checkboxes, in order to save code and memory space. Standard setup is all modules enabled.

The modules, and the consequences of deselecting them, are:

- **Cursor support.** Cursor fields cannot be used in the target code. Saves approximately 2kB of code.
- **Scroll support.** Scrolling boxes cannot be used in the target code. Saves approximately 5½kB of code.

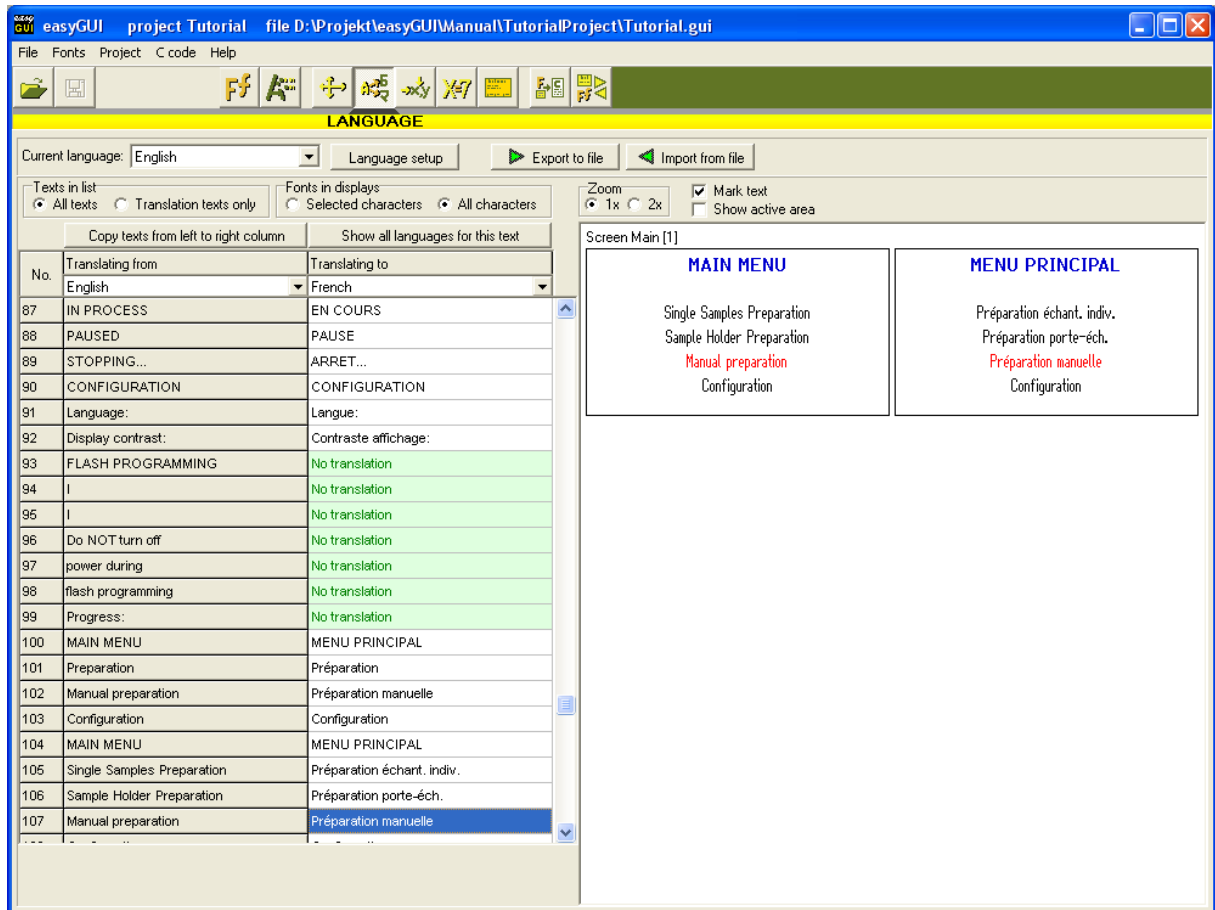
- **Blink support.** Blinking boxes (for e.g. blinking cursors) cannot be used in the target code. Saves approximately ½kB of code.
- **Clipping support.** Clipping rectangles cannot be used in the target code. More important, drawing of objects (text, lines, etc.) outside the display area is no longer caught by the easyGUI library, and can potentially cause memory area violations. Saves approximately 3kB of code.
- **Bitmap support.** Bitmaps cannot be used in the target code. Do not confuse with icons in fonts, which can still be used. Saves approximately 2½kB of code.
- **Floating point support.** Variables of types `float` and `double` cannot be used in the target code. The amount of code saved on the target system overall differs depending on the floating point library in the compiler in use, and more important, whether it is used by other parts of the target code. The amount of code saved in the easyGUI library is negligible.

Do only deselect modules if forced to do so by memory constraints. This saves troubleshooting, if a function using one of the de-selected modules is inadvertently used.

When creating c files easyGUI warns, if e.g. a scroll box definition is met, with the scroll box module turned off.

8 LANGUAGE TRANSLATION WINDOW

Displays a list of all texts in structures

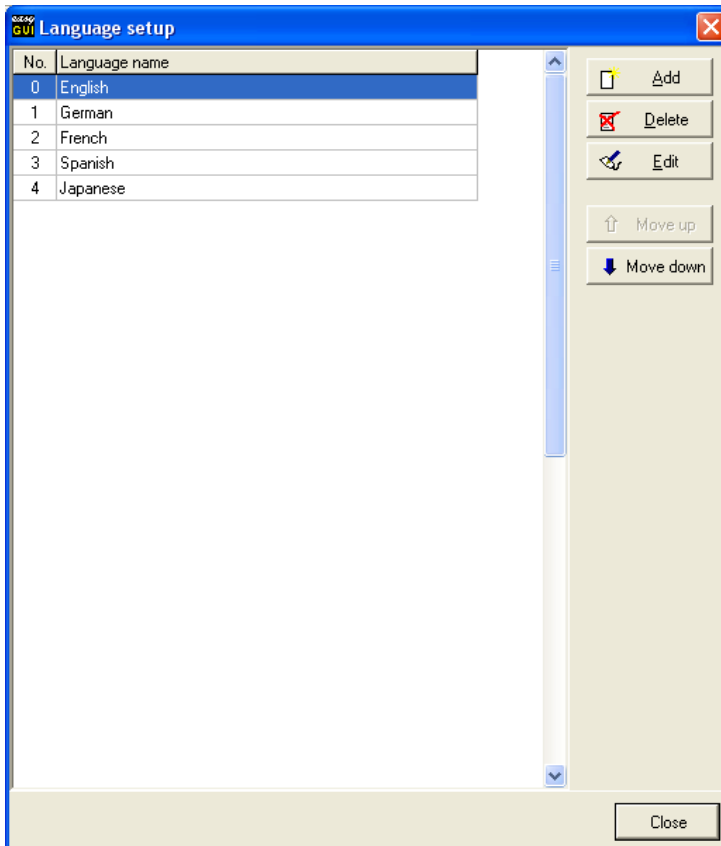


The list in the left part of the window contains two columns of texts, a reference column and an editable column. The reference column is then set to the reference language (which doesn't have to be the primary language), and the editable column is set to one of the other languages in the project (French in this example). For the text selected in the right text column (Préparation manuelle in this case) all structures containing this text is shown in the right half of the window (a single structure in this example). Each structure is shown in two versions, with the left one using the reference language (English here), and the right using the edited language (French here). The text under editing is shown in red, this can be suppressed if desired by un-checking the **MARK TEXT** check box above the structures. The other check box **SHOW ACTIVE AREA** determines if an additional rectangle shall be drawn around the active pixels in the display, marking the boundaries of active pixels, inside which text can be written.

At the top is a box for selecting the current language when editing structures. This setting has no immediate effect in the language window, but controls which language is used when showing structures in other windows in easyGUI.

Furthermore three buttons are placed at the top:

- **LANGUAGE SETUP.** Shows a windows with language definitions:



Individual languages can be added, removed, and moved up and down. The topmost language (index zero) is the primary language, which is automatically active at target code startup time.

For each language can be selected a character set, most languages will use character set 0 (Windows ANSI characters). In Unicode mode the **CHAR.SET** column is not relevant, and therefore not shown.

A project must always contain at least one language.

- **EXPORT TO FILE.** Exports all text with associated fonts and structure information to a special file with `egt` filename extension. This file is used by the translate utility `easyTRANSLATE`, used by exterior persons assigned to the task of translation. This utility can accomplish the same as the translation part of the Language window in `easyGUI`.
- **IMPORT FROM FILE.** Imports data back from `easyTRANSLATE`. Texts that were changed externally in `easyTRANSLATE` are marked with a little red **E** to the left, until the project is saved.

For more information on exporting and importing, see the `easyTRANSLATE` chapter later.

Above the text columns are two settings:

- **TEXTS IN LISTS.** Switches between showing all texts, or only show texts selected for translation. Translation is elected for each text individually in the structure editor. Texts not marked for translation are omitted in the right text column, and replaced by a green box stating "No translation".
- **FONTS IN DISPLAYS.** Switches between only allowing display of characters currently active in the project, or allowing display of all characters. Character and font selection is handled in the Font editor window.

At the top of the left text column is a button:

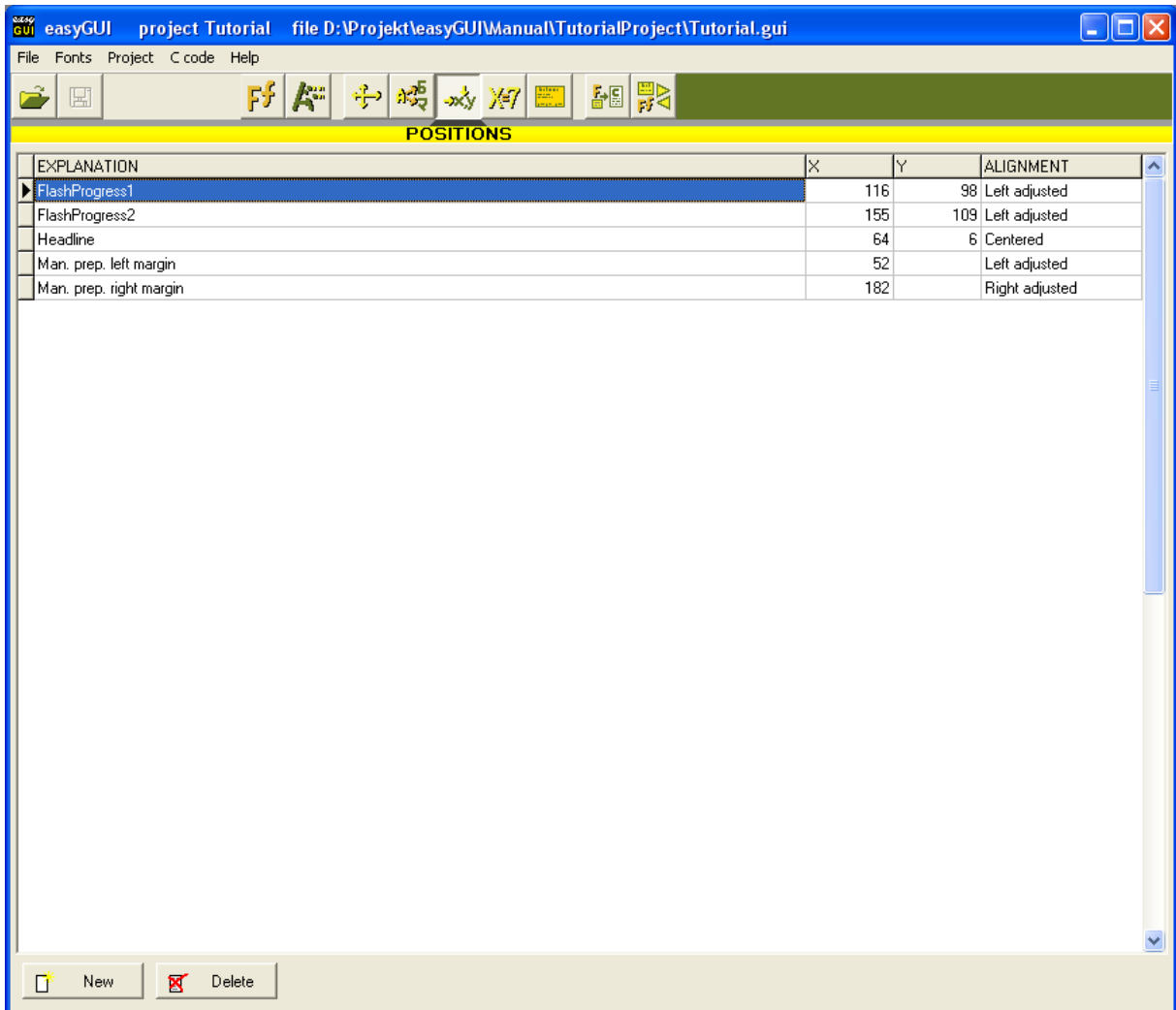
- **COPY TEXTS FROM LEFT TO RIGHT COLUMN.** This action can be used to reset the translation. When pressed easyGUI asks if all texts should be copied, or only texts for which the corresponding right text column line is empty. The latter option will be the usual action, as the former overwrites eventual translations in the right text column.

At the top of the right text column is a button:

- **SHOW ALL LANGUAGES FOR THIS TEXT.** Pressing the button shows a little window displaying all language texts for the active text line, not only the two visible in the left and right columns. This can be handy when checking other languages during translation. This dialog can also be invoked in the structure editor.

9 POSITIONS WINDOW

Displays a list of coordinates:



These coordinates can be used in structure editing, ensuring that related items are placed at the same position in different structures, e.g. the vertical position of headlines. easyGUI can be used without utilizing the position list, it is meant as an option.

For each position an **X**, a **Y**, and an **Alignment** can be set. Additionally, an explanation can be made, for information purposes only.

10 VARIABLES WINDOW

Displays a list of variables:

The screenshot shows the 'easyGUI' application window with the 'VARIABLES' window open. The window title is 'project Tutorial' and the file path is 'D:\Projekt\easyGUIManual\TutorialProject\Tutorial.gui'. The menu bar includes 'File', 'Fonts', 'Project', 'C code', and 'Help'. The toolbar contains various icons for file operations and variable management. The main area is a table with the following data:

Name	Type	Numerical value	String value	Explanation
DiConfigDisplayContrast	8 bit unsigned	25		
DiConfigLanguage	8 bit unsigned	0		
DiForceDoser1Level	8 bit unsigned	10		
DiForceForce	16 bit unsigned	80		
DiForceForceProcess	16 bit unsigned	30		
DiForceSpeed	16 bit unsigned	150		
DiForceTimeMM	16 bit unsigned	99		
DiForceTimeMMProcess	8 bit unsigned	4		
DiForceTimeSS	16 bit unsigned	59		
DiForceTimeSSProcess	8 bit unsigned	0		
DiForceWaterOnOff	8 bit unsigned	1		
DiMessageBoxErrorNo	16 bit unsigned	1		
DiMessageBoxKeys	8 bit unsigned	1		
DiMessageBoxNo	8 bit unsigned	1		
DiMessageBoxType	8 bit unsigned	0		
DiSoloUserDefSpeed	16 bit unsigned	300		
DiSoloUserDefWaterOnOff	8 bit unsigned	1		
Doser1SwitchPos	8 bit unsigned	2		
EditTimeMM	8 bit unsigned	99		
EditTimeSS	8 bit unsigned	59		
EditUnsignedInt	16 bit unsigned	1500		
ForceReductFlag	8 bit unsigned	1		
ForceReduction	8 bit unsigned	0		
GroupName	string	20	Group name test TEST	
GroupNo	16 bit unsigned	1		

Below the table, there is a legend: a hatched box represents 'Variable not in use'. A note states: 'OBS: For "string" variables the numerical value denotes string length [0-80] excluding terminating zero.' The bottom toolbar includes buttons for 'New', 'Copy', 'Delete', 'Used by', 'Load values', 'Save values', 'Import variable definitions', and 'Import setup'.

The variables can be used on the target system just as variables defined normally, but furthermore they can be used to control structures in structures, i.e. structures can be shown depending on the setting of a controlling variable. The variables are created, erased and copied in this window. Each variable has the following properties:

- **Name.** Must conform to standard C syntax. When used on the target system all variables will have the text "easyGUI_" added before the variable name, in order to make clear that this variable originates from the easyGUI system.
- **Type.** One of the following types:
 - **bool.** A special case of 8 bit unsigned which can only be assigned the values "false" (=0) and "true" (=1).

- **8 bit unsigned.**
 - **8 bit signed.**
 - **16 bit unsigned.**
 - **16 bit signed.**
 - **32 bit unsigned.**
 - **32 bit signed.**
 - **float.**
 - **double.**
 - **string.**
- **Numerical value.** The numerical value is only used inside easyGUI, it is up to the programmer to assign proper values on the target system. Has a special meaning for string type variables, where it defines the number of characters.
 - **String value.** Has only meaning for string type variables.
 - **Explanation.** Free text for information purposes only.

Variables not in use anywhere in easyGUI are marked with a special background pattern (e.g. `DebugLong1`).

All values can be saved in a file, and later reloaded, using the **SAVE VALUES** and **LOAD VALUES** buttons. This can come handy when setting up a lot of variables controlling dynamic structures to show a specific situation in easyGUI. The save/load feature has no effect on target system code.

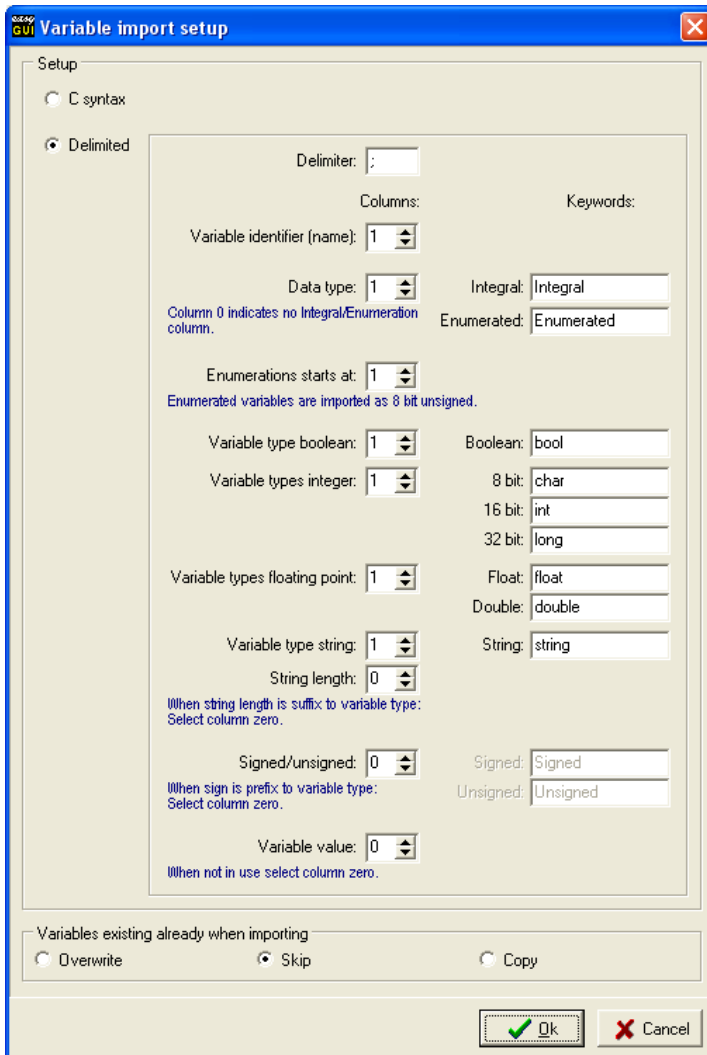
By pressing **USED BY** a list of structures referencing the currently selected variable is shown. Double-clicking on one of the structures jumps directly to the structure editing window.

IMPORTING DEFINITIONS

Variable definitions can be imported from various file formats, using the **IMPORT SETUP** button to specify the import format, and the **IMPORT VARIABLE DEFINITIONS** button to execute the actual import.

Import setup

The import setup window contains a lot of settings:



Import type

First of all there is a selection between C style import or Delimited import:

- **C syntax.** C style import follows standard C syntax, and tries to extract all variable definitions from the C code. Complex structures and arrays are skipped, as are constants. Variable types can both be standard C types, as defined in the Parameters window, Compiler tab page, Type definitions box (example):
 - **bool**
 - **char**
 - **signed char**
 - **unsigned char**
 - **signed short**
 - **unsigned short**
 - **signed long**

- **unsigned long**
- **float**
- **double**

- or they can be standard easyGUI variable types, like:

- **GuiConst_CHAR**
- **GuiConst_INT8S**
- **GuiConst_INT8U**
- **GuiConst_INT16S**
- **GuiConst_INT16U**
- **GuiConst_INT32S**
- **GuiConst_INT32U**
- **GuiConst_TEXT**
- **GuiConst_CHAR**

Any mix of definition types is allowed.

- **Delimited.** This format is for formalized, and is also known as e.g. comma-delimited text. The many parameters in the setup window allows for very flexible configuration of the import format.

The following parameters can be edited:

- **Delimiter.** The character separating parameters in the import file.

Most of the parameters are either column numbers, or keywords. Columns are divided by the delimiter character, and are numbered one, two, etc. in each imported text line.

- **Variable identified** column. The column containing the names of the variables.
- **Data type** column. Allows for variables to be divided between integral and enumerated variable definitions. If this division between types is not necessary the column index can be set to zero.

Please observe that enumerated types are currently not supported by easyGUI, but that these variable definitions are imported as variables of type unsigned char.

- **Integral** keyword. The keyword to be found in the Data type column, if a variable definition is of integral type.
- **Enumerated** keyword. The keyword to be found in the Data type column, if a variable definition is of enumerated type.
- **Enumerations starts at** column. For an enumerated variable type the first enumeration declaration is found in this column. The following columns are expected to contain additional enumerations, until the end of the line.

- **Variable type boolean** column.
- **Boolean** keyword. The underlying variable type for boolean is an unsigned char.
- **Variable types integer** column.
- **8 bit** keyword.
- **16 bit** keyword.
- **32 bit** keyword.
- **Variable types floating point** column.
- **Float** keyword.
- **Double** keyword.
- **Variable type string** column.
- **String** keyword.
- **String length** column. This column setting is to be used if the string length is specified in its own column, and not as part of the string keyword. If this feature is not necessary the column number can be set to zero.
- **Signed/unsigned** column. This column setting is to be used if the signed/unsigned choice for integer types is specified in its own column, and not as part of the integer keyword. If this feature is not necessary the column number can be set to zero.
- **Signed** keyword.
- **Unsigned** keyword.
- **Variable value** column. This column setting is to be used if values for variables are specified in their own column. If this feature is not necessary the column number can be set to zero - all variable values will then be set to zero/empty string.

The columns for boolean, integer, float, etc. can be the same column, in fact this is often the case.

Making the import

When pressing the **IMPORT VARIABLE DEFINITIONS** button a dialog is shown, which permits selecting the desired import file. After pressing Ok the import will start.

All variable definitions accepted by the importer function are then either created, skipped, or a similarly named variable overwritten, depending on the setup. Corrupt or illegal syntax in the import file is simply skipped.

11 STRUCTURES WINDOW

Screen structures are the basic ingredient in easyGUI. Structures are simple or complex collections of text and graphical elements, which together comprises the visual part of a user interface.

THE BASICS

A complete screen picture on the machine is made up of one or more screen structures, shown successively on top of each other. Typically there could be separate structures for headline, menu commands, and main functionality of the screen, but that is entirely up to the user to determine.

Each structure is identified by a name and an index number (0-99).

The index number is used when calling indexed structures. These are structure calls based on a variable value that determines which structure of several with identical names should be called (i.e. shown). An example could be: Two structures are made, each containing just one text, where the first structure has the name/No. TempUnitStr [0] (index number always shown in [x] brackets after the name) and contains a text "°C", while the second has the name/No. TempUnitStr [1] (same name, but differing index number) and contains the text "°F". A structure wanting to display a temperature could display the numerical value itself, followed by an indexed structure call to one of the "°C" and "°F" structures, based on the value of a variable called e.g. TempUnit. The call would specify structure name TempUnitStr, and if the value of variable TempUnit is 0 structure TempUnitStr [0] will be shown, if the value is 1 structure TempUnitStr [1] will be shown, and if the value is something else nothing will be shown. The last situation is not illegal, but can be used to great advantage to include or exclude parts of a screen layout, based on variables. If only a structure named XXX [1] exists, but not structure XXX [0], setting the controlling variable to 0 will show the screen without structure XXX [1], and setting the controlling variable to 1 will show the screen with structure XXX [1]. A structure shown dynamically this way can itself contain calls to other dynamic (or static) structures, enabling complex systems to be made in easyGUI, controlled by only a few variables in the target system C code. The only limits are stack space considerations on the target system, and the ability of the user to keep a mental picture of the constructs.

ITEMS

Each structure contains a number of items (0-255). Each item can be one of the types:

- **Text** A single text string with associated parameters.

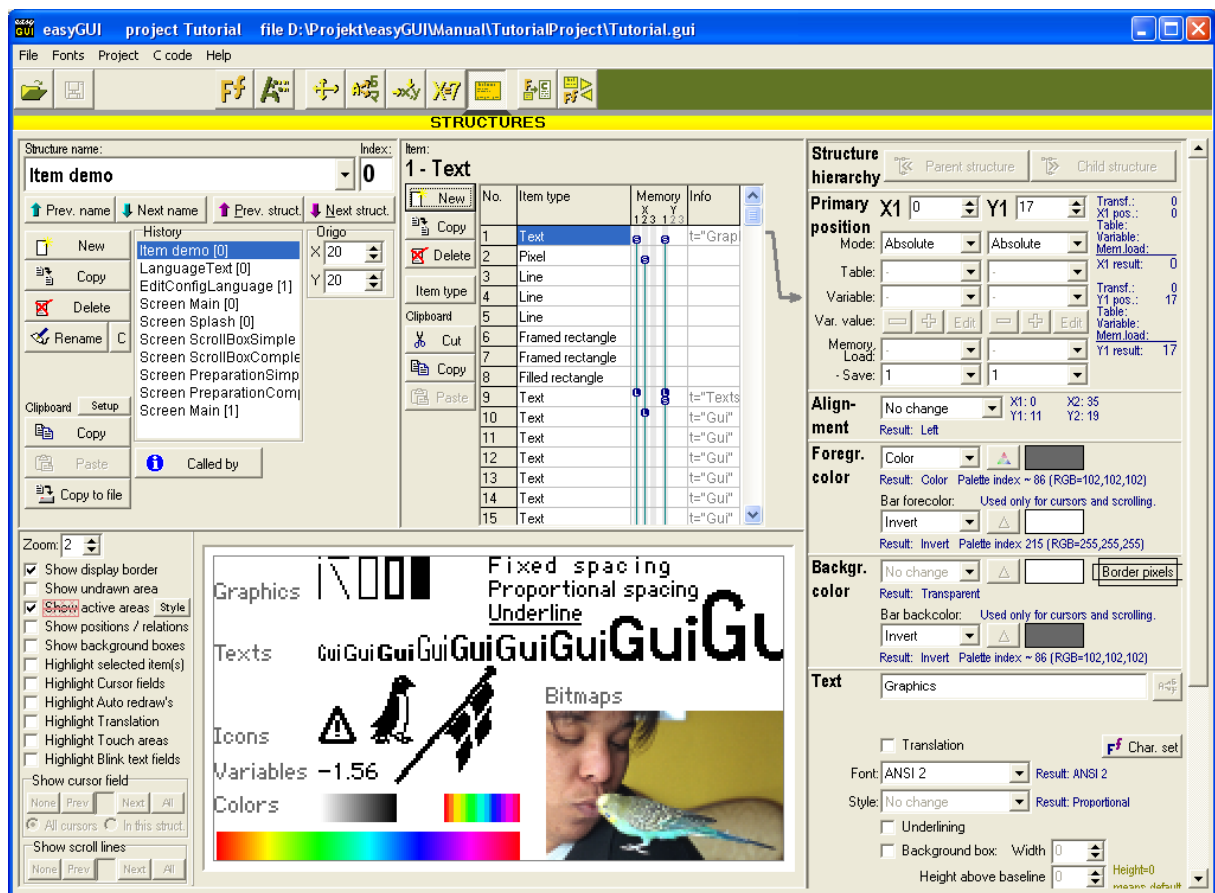
- **Paragraph** A text box with associated parameters, where text is automatically divided into lines at word spaces and hyphen characters.
- **Pixel** A single pixel.
- **Line** A line segment, can be in any angle.
- **Framed rectangle** A rectangle frame with specified thickness, optionally filled with another color.
- **Filled rectangle** A filled rectangle without border.
- **Bitmap** A bitmap file located outside the project file. The bitmap is shown in full color in easyGUI, but is reduced to the colors possible by the currently selected color mode and color depth on the target system.
- **Structure call** Unconditional call of another structure, specified by both name and index number.
- **Indexed structure call** Indirect call of another structure, specified only by name. Structure index number is specified dynamically at runtime through a variable.
- **Variable** Writes a variable according to current formatting settings.
- **Variable paragraph** A variable string box with associated parameters, where text is automatically divided into lines at word spaces and hyphen characters.
- **Formatter** Sets variable formatting, no visible output. Has no effect on string variables.
- **Active area** Defined an active area of the display. Display writing falling outside the active area is not prohibited, but can optionally be clipped. The coordinate system may optionally be transferred to the active area. The active area is in effect for the rest of the structure, or until another active area item is encountered.
- **Clipping rectangle** Defines a clipping rectangle. All following items are drawn clipped to the specified rectangle. Can be cancelled by another clipping rectangle.
- **Touch area** Defines an area of the display for the touch interface. The touch areas are individually numbered. There is no visual drawing associated with touch areas.

The two items "Structure call" and "Indexed structure call" are the ones that give easyGUI its dynamic properties, by enabling structures to be called from structures, either unconditionally, or controlled by a variable.

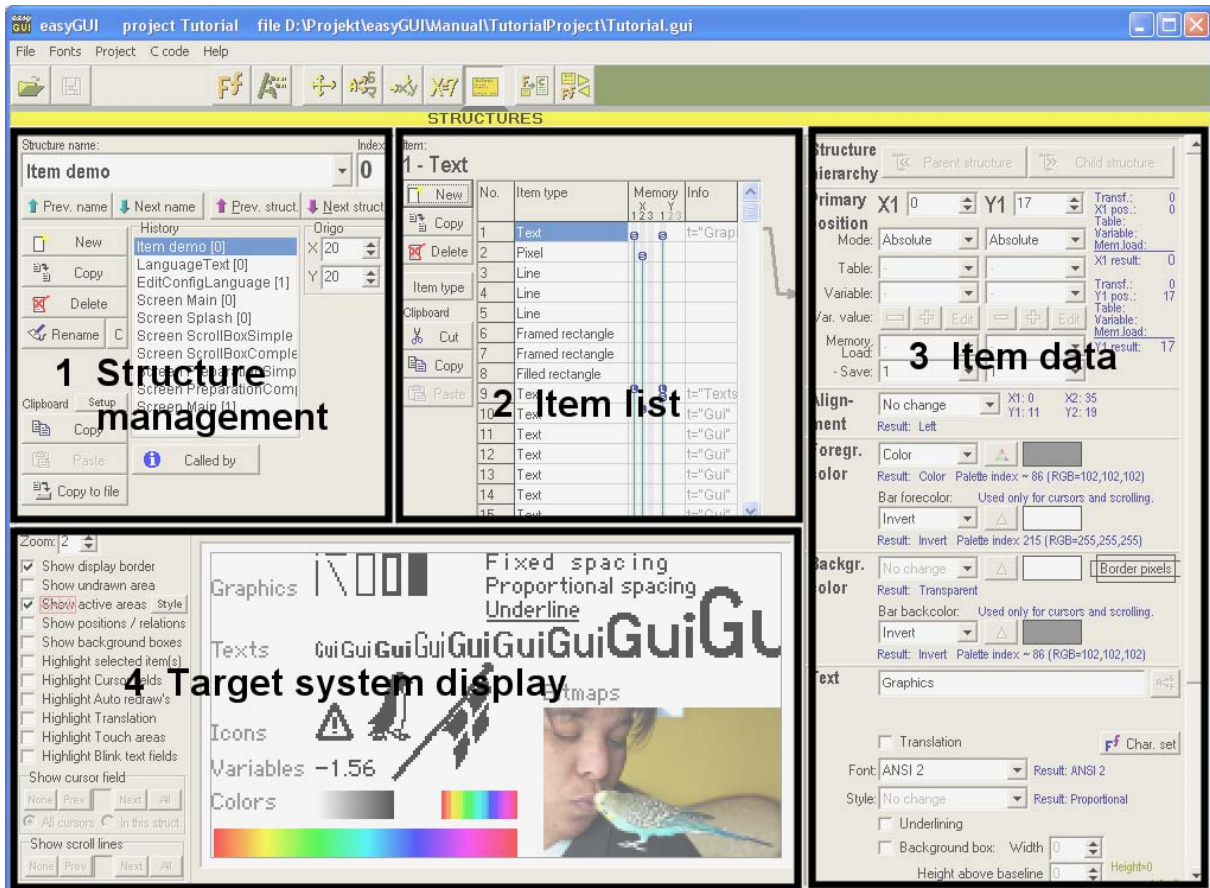
When a structure is drawn, items are drawn sequentially, starting with item one. When encountering a structure call (child structure) the items of the child structure are drawn completely, before continuing with the rest of the parent structure items.

WINDOW LAYOUT

easyGUI structures are handled in this window:



The window is rather complex, as it contains a lot of functionality, but breaking the window into its major panels shows the organization more clearly:



Each panel handles a part of the editing process:

- 1 Structure management.** Creates, deletes, copies, navigates etc. complete structures.
- 2 Item list.** Shows all items of the currently selected structure.
- 3 Item data.** Shows all parameters for the currently selected item.
- 4 Target system display.** Shows the end result for the currently selected structure.

The panels are explained in the following chapters.

STRUCTURE MANAGEMENT PANEL

Inside this panel are a number of various commands controlling complete structures. At the top is a drop-down box containing all structures in the project. Right next to it is an index number box, showing the index number of the currently selected structure. The drop-down box and index number box is not editable. Below these boxes are four buttons allowing quick selection of previous and following structures, both based on name and index number.

At the left are a number of buttons managing structures:


NEW	Creates a new structure with no items in it.
COPY	Copies the current structure. The new structure can be given a new name and index number, or just a new index number, making it a sister to the current structure.
DELETE	Deletes the current structure.
RENAME	Renames the current structure, either by just changing the index number, or by altering the name, or both.
C	Copies the structure name to the Windows clipboard, with the text <code>GuiStruct_</code> added before it. This ensures easy pasting into target system C code.
CLIPBOARD SETUP	Determines how structures exported to the clipboard shall look. Border thickness, white space above and below the structure (to make it easier to insert the bitmap in Word), colors used, and clean/easyGUI style appearance can be set.
CLIPBOARD COPY	Copies the current structure to the Windows clipboard as a graphic. Can then be inserted directly into another application, e.g. Word or Corel Draw. Furthermore, the structure are copied into an internal easyGUI clipboard, allowing it to be pasted into another project, or another easyGUI database, as long as easyGUI is not closed down.
CLIPBOARD PASTE	Pastes a structure from the internal easyGUI clipboard.
CLIPBOARD COPY TO FILE	Copies the current structure to a .bmp bitmap file.

In the middle of the panel is a history box showing the most recent structures selected for editing. A structure can be made current by clicking it. The topmost structure is the current structure, and nothing will happen if it is clicked.

The buttons below the history box controls service functions:

CALLED BY	Shows a list of all structures calling the current structure, either directly or indexed. The list may be empty, but if not, one of the calling structures can be selected by clicking it.
------------------	--

To the right of the panel is an origo box, containing an X and a Y coordinate. These coordinates are only used in easyGUI, not on the target system. They determine where the first item of a structure is placed, if this item uses relative coordinates. This is usual practice for structures containing parts of a display layout, so these structures will never be shown on their own on the target system. If they are shown directly on the target system the origo is set to 0,0. In easyGUI it is practical to set origo to e.g. 20,20 to bring relative texts into view in the display panel. If origo is kept at 0,0 a relative text will only be partially visible at the top left corner of the display, very inconvenient. If the first item in a structure uses absolute coordinates the origo setting has no effect. The origo setting is individual for each structure.

A warning button  may be shown below the origo box, if easyGUI detects a possible error condition. Pressing it shows the relevant warning. The following warnings are possible:

- "Largest text is X characters in length, max. text length selected in Project parameters is Y characters". The max length should be increased, or alternatively, the item text made shorter or divided into two separate texts.
- "Largest numerical variable is X characters in length, max. string length selected in Project parameters is Y characters". The max length should be increased, or alternatively, the variable formatting changed.
- "No. of Auto Redraw items in this structure is X, max. No. of auto redraw items selected in Project parameters is Y items". The max. No. of items should be increased, or alternatively, the number of auto redraw items should be reduced.

ITEM LIST PANEL

The item list shows all items in the current structure. The list may be empty if no structure is currently selected (only possible if no structure exists at all), or if the current structure contains no items (not very useful...). The items are numbered from one to at most 256, always sequentially. Items can be added, copied and deleted using the buttons to the left of the list:

NEW	Creates a new item above the current one (or optionally below, if the last item is current).
COPY	Copies the current item (or items) to a position below the current one.
DELETE	Deletes the current item (or items).
ITEM TYPE	Selects the item type.
CLIPBOARD CUT	Cuts (deletes) the current item (or items) from the structure and places it in an internal easyGUI clipboard, allowing it to be pasted into another structure, eventually in another project file, as long as easyGUI is not closed down.
CLIPBOARD COPY	Copies the current item (or items) to the internal easyGUI clipboard, allowing it to be pasted into another item, as long as easyGUI is not closed down.
CLIPBOARD PASTE	Pastes the item (or items) from the internal easyGUI clipboard into the structure before the current item. If the current item is the last, a selection box is shown, offering the item(s) to be pasted before or after the last item.

More than one item may be selected by dragging the mouse over the desired items (NOT in the grey area to the left of the item list).

An item (or more items) may be moved to a new position in the item list by dragging in the grey area to the left of the list.

The two last columns contain information about saved coordinate positions and special item attributes (e.g. cursor fields) making it easier to keep an overview of the situation.

ITEM DATA PANEL

The actual item editing is made in this panel. It contains a number of sub-panels organized vertically. The number and type of panels depends on the item type, but their ordering is fixed. The following sub-panels can be shown:

Structure hierarchy sub-panel

All item types.

Allows quick movement to connected structures, either **PARENT STRUCTURE** or **CHILD STRUCTURE**. The **PARENT STRUCTURE** button is active if the current structure was selected as a child of another structure. The **CHILD STRUCTURE** button is visible if the current item calls another structure. These buttons doesn't edit anything, they are just convenient ways of navigating between structures belonging to a common "family tree".

Primary position sub-panel

Item types: Text, Paragraph, Pixel, Line, Framed rectangle, Filled rectangle, Bitmap, Structure call, Indexed structure call, Variable, Variable paragraph, Active area, Clipping rectangle, and Touch area.

Edits the primary coordinate pair (X1,Y1). Coordinates has (0,0) at the top left corner of the display, with X coordinates running to the right, and Y coordinates running down. The following parameters can be edited for each coordinate:

- **Coordinate value.** 16 bit, can be negative too.
- **Mode.** Can be:
 - **Absolute.** The coordinate value is used directly.
 - **Relative.** The coordinate value is added to the calculated coordinate of the previous item. For item zero it is added to the origo coordinate value.
 - **Relative to start.** The coordinate value is added to the starting coordinate of the previous item. This is e.g. the left edge of a text (or top in case of Y coordinate). This is not the same as the calculated coordinate of the previous item, if that item e.g. contains a centered text, the calculated coordinate would then be at the centre of this text.

- **Relative to end.** The coordinate value is added to the starting coordinate of the previous item. This is e.g. the right edge of a text (or bottom in case of Y coordinate).
- **Table.** Fetches a coordinate from the Position window.
- **Variable.** Fetches a coordinate from a variable value. The variable must be of an integer type. The value can be edited using the small buttons below the variable box.
- **Memory load.** Fetches a coordinate from a memory buffer. There are three memory locations, individually for X and Y. The memory values are only stored during the writing of one structure, including calls to child structures. When structure writing begins all memory buffers are reset to zero.
- **Memory save.** Saves the current coordinate value in one of the three memory locations.

The calculated coordinate is a sum of all contributions (previous coordinate, relative coordinate, etc.), the calculation can be viewed at the right.


Secondary position sub-panel

Item types: Paragraph, Line, Framed rectangle, Filled rectangle, Variable paragraph, Active area, Clipping rectangle, and Touch area.

Edits the secondary coordinate pair (X2,Y2). It is almost identical to the primary coordinate pair, except that relative coordinates are not relative to the previous item, but relative to the primary coordinate pair (handy for the size of boxes).

Structure call sub-panel

Item types: Structure call and Indexed structure call.

Selects a structure for calling, either by name and index number (direct structure call), or by name only (indexed structure call). The **JUMP TO STRUCTURE** button does exactly the same as the **CHILD STRUCTURE** button. If a selected structure does not exist (deleted after selection, or index doesn't exist) a small warning ( **Structure missing!**) is shown. This is maybe not an error, at least not if the structure call is indexed. A direct structure call displaying this warning certainly deserves attention.

Variable sub-panel

Item types: Indexed structure call, Variable, and Variable paragraph.

Selects a variable for the indexed structure call or for displaying. The value can be edited using the small buttons below the variable box. The **C** button copies the variable name to

the Windows clipboard, with the text `GuiVar_` added before it. This ensures easy pasting into target system C code.

Active area sub-panel

Item types: Active area.

Contains only a check box that determines if the coordinate system origo shall be moved to the active area upper left corner, or left as is. The coordinate move is only in effect for items following the active area item.

Clipping sub-panel

Item types: Clipping rectangle and Active area.

Contains only a check box that determines if the clipping action is active.

Touch area sub-panel

Item types: Touch area.

The touch area number can be set. The allowed range is 0-255. This number is used in the easyGUI library when referencing to individual touch areas. Any numbering scheme can be used, even several touch areas with the same number, if desired.

Alignment sub-panel

Item types: Text, Paragraph, Line, Framed rectangle, Filled rectangle, Bitmap, Structure call, Indexed structure call, Variable, Variable paragraph, Active area, Clipping rectangle, and Touch area.

Edits the horizontal alignment of the item. There are five alternatives:

- **No change.** Keeps the alignment setting currently in use.
- **Left adjust.** The item is placed so that its left edge is at the calculated X coordinate.
- **Centre.** The item is placed so that it is centered over the calculated X coordinate.
- **Right adjust.** The item is placed so that its right edge is at the calculated X coordinate.
- **From X1 table.** Takes the alignment defined in the Position function for the table position defined under coordinate X1.


The vertical alignment cannot be freely selected. Texts are placed with their Base line over the calculated Y coordinate. Other objects have their top at the calculated Y coordinate.

Foreground color sub-panel

Item types: Text, Paragraph, Pixel, Line, Framed rectangle, Filled rectangle, Structure call, Indexed structure call, Variable, and Variable paragraph.

Selects the foreground and bar foreground colors. There are five alternatives:

- **No change.** Keeps the color currently on use.
- **Pixel ON.** This color is set in the display parameters function. Normally it means a dark pixel in monochrome systems, but could mean the opposite in inversed systems (light text on dark background).
- **Pixel OFF.** This color is set in the display parameters function.
- **Color.** The color can be freely selected from the possible colors on the target system, depending on the currently selected color mode and color depth. Not relevant in monochrome target display systems.
- **Invert.** Uses the current background color.

The  button shows a window for color selection. Only applicable when the "Color" type has been selected. Appearance of the color selection window depends on the currently selected color mode and color depth in Project parameters. How to operate the color selection window is also explained there.

The bar foreground color is used for cursor fields and scroll lines, when these are active, i.e. selected. In previous versions of easyGUI the foreground and background colors were merely swapped, that corresponds to both foreground bar color and background bar color being set to Invert. If the item in question is not used in cursor fields/scroll lines the bar foreground color has no effect.


Background color sub-panel

Item types: Text, Paragraph, Framed rectangle, Structure call, Indexed structure call, Variable, and Variable paragraph.

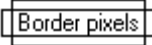
Selects the background and bar background colors. There are six alternatives:


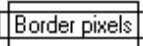
- **No change.** Keeps the color currently on use.
- **Pixel ON.** This color is set in the display parameters function. Normally it means a dark pixel in monochrome systems, but could mean the opposite in inversed systems (light text on dark background).

- **Pixel OFF.** This color is set in the display parameters function.
- **Color.** The color can be freely selected from the possible colors on the target system, depending on the currently selected color mode and color depth.
- **Invert.** Uses the current foreground color.
- **Transparent.** No background is drawn.


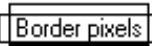
The  button shows a window for color selection, just like explained above for foreground color.

The bar background color is used for cursor fields and scroll lines, when these are active, i.e. selected. In previous versions of easyGUI the foreground and background colors were merely swapped, that corresponds to both foreground bar color and background bar color being set to Invert. If the item in question is not used in cursor fields/scroll lines the bar background color has no effect.

The  area controls the addition of an extra pixel row or column, to make the background more prominent, and to allow the background to fully contain e.g. a "g" letter, without the bottom of the "g" touching the background border. One or more of the four rectangles surrounding the "Border pixels" text can be clicked. If a rectangle is black it means that an extra pixel row/column is added on that side. Example:

 Normal setting ()

- versus:



 Extra bottom row of pixels ()

Border pixels also works for background boxes.


Text sub-panel


Item types: Text, Paragraph, Structure call, Indexed structure call, Variable, and Variable paragraph.

Controls the appearance of texts and variables. The following parameters can be set:

- **Text** box (Text items only). The actual text. The  button opens a small window allowing simultaneous editing of all languages in the project for the item, and furthermore allows selection of another language as the current. The  button is only enabled if translation is on for this item.
- **Translation** checkbox (Text items only). Determines if the text should be translated (all languages defined in the project) or only be kept in one version (the primary language, normally English) because the text is part of e.g. a service page not needing translation.

- **Character set** button (Text items only). Shows a window containing all available characters in the selected font. Select characters by double-clicking them. The character set window can also be used to see the character code for a specific character already in the text. Place the cursor just before the character in question, and invoke the **CHARACTER SET** button. The same character will be selected in the window, and its character code can then be inspected.
- **Character set override** (Text and Variable items only, ANSI character mode only). Selects a specific character set, disregarding the character set in effect due to language selection. Can be used in e.g. a language selection list, where Japanese should be written in Katakana, no matter which language (and thereby character set) is active. Should normally be selected off ("-" setting).
- **Font**. Can be set to any font which has at least one character included in the project (through the font selection function), or can be set to "No change" meaning that the currently selected font is used. It is easier to later change a font if only the first of a number of contiguous items selects a specific font, while the rest has the "No change" setting.
- **Style**. Select the writing style as:
 - **No change**. Keeps the writing style currently on use.
 - **Fixed spacing**. All characters occupy the same horizontal space (Courier style).
 - **Proportional**. Normal proportional writing is used.
 - **PS numerical**. Special numerical proportional writing is used, see font chapter.
- **Underlining**. Is selected in a checkbox. The size and placement of underlining is determined by font parameters.
- **Background box** (not Paragraph items). A background box is a special kind of background drawing. A box is drawn in the background color with the width and height as specified. Background boxes are useful for e.g. menus arranged vertically, so that each menu item has the same background width when selected. Example: Two menu items arranged vertically ("Preparation" and "Configuration")

"Preparation" selected: 

"Configuration" selected: 

Both texts are centered, and have background box widths of 75 pixels.

Another use is when dynamically changing texts are shown, e.g. by using an indexed structure call item, or for variables. To make sure the old text gets erased when displaying a new text in the same position the item can be supplied with a background box of sufficient size to cover the largest possible text. The individual

texts in the called structures then only needs a foreground color, the background color should be set to "Transparent". This also avoids any unnecessary background drawing, i.e. drawing first a background box, and then a normal background in the same position, which would waste processor time.

Parameters comprises of a checkbox (Background box on/off) and three edit boxes, specifying the background box width in pixels, background box height above text baseline in pixels, and finally background box height below text baseline in pixels. The two last parameters (background box heights) can be set to zero, in which case the height above and/or below the text baseline will be equal to normal background drawing.

Background boxes are normally (but not necessarily) used in conjunction with centered texts, as in the above example.

- **Blinking text field.** If a text is marked as Blinking it can also be assigned a number, just like cursor fields. In the target code the function `GuiLib_BlinkBoxMarkedItem` can then be used to blink single characters in the text, or the complete item text. The function `GuiLib_BlinkBoxStop` stops blinking again.

Paragraph sub-panel

Item types: Paragraph and Variable paragraph.

Selects special Paragraph settings regarding alignment and line height:

- **Horizontal alignment.** There are three alternatives:
 - **Left.** All text lines start at the left edge of the paragraph box.
 - **Center.** All text lines are centered between the left and right edges of the paragraph box.
 - **Right.** All text lines end at the right edge of the paragraph box.
- **Vertical alignment.** There are three alternatives:
 - **Top.** The first text line is placed at the top of the paragraph box.
 - **Center.** The text lines are centered between the top and bottom edges of the paragraph box.
 - **Bottom.** The last text line is placed at the bottom of the paragraph box.
- **Line height.** Determines the distance between lines in the Paragraph box, and hence the number of lines visible in the box.

The Alignment sub-panel described earlier is also visible for Paragraph items, but the alignment selected there only affects the positioning of the complete Paragraph box, not the placement of its contents inside the box.

Bitmap sub-panel

Item types: Bitmap.

Selects the bitmap file for display. The file can be specified with or without a path. Files without a path is read from the folder in which the project file (*.gui) resides. A partial path may also be entered, in which case it is taken as relative to the project file folder.

The bitmap is not stored in the project file, only its path and filename. Changing the bitmap therefore influences how it looks in easyGUI. The path and filename can be edited directly, or selected by pressing the **BROWSE** button.

The **REFRESH** button reads the bitmap again, and can be used to force a re-read, if the bitmap has been edited. Jumping to another structure, and back again, also forces a re-read.

The size of the bitmap in pixels, and its filename without the path, is shown above the file name edit box.

If the bitmap is not found a warning is shown, and the bitmap is drawn as a black rectangle with white fill, and a black cross covering it.

Rectangle sub-panel

Item types: Framed rectangle and Filled rectangle.

Displays rectangle size, based on the current coordinate settings. Framed rectangle items also show an edit box, allowing selection of border thickness in pixels.

Variable formatting sub-panel

Item types: Variable formatter.

Determines formatting for numeric variables. The following parameters can be edited:

- **Field width.** The number of digits allowed in the numeric representation of a variable. Zero indicates variable field width, i.e. the field width is made just sufficient to display the variable.
- **Decimals.** Determines the number of decimals after the decimal point. The setting works for both integers and floats. For integers the decimal point ("." or ",") is simply inserted to display the number of decimals, e.g. two decimals shows the value 123 as "1.23". Zeroes are inserted if needed: The value 23 is shown as "0.23". The type of decimal point ("." or ",") is set in the Parameters window.
- **Alignment.** Determines how the text is placed inside the field width. Can be left adjusted, centered, or right adjusted. Don't confuse this alignment with the normal alignment for texts, boxes, etc. The formatter alignment determines how

the digits/characters are placed in the field width. With the field width set to zero (dynamic width) this setting has no effect.

- **Format.** Can be decimal, hexadecimal, exponential, and time (HH:MM). The exponential notation works only on float variables. The time format works only on integer variables, and uses the variable value as a minute count.
- **Always show sign.** If set it will always show the sign, even for positive values ("+123"). Zero is shown as "+0". Has no effect on unsigned variables.
- **Zero padding.** Pads the value with leading zeroes. With the field width set to zero (dynamic width) this setting has no effect. Works only with alignment set to right adjusted.

The formatter parameters have no effect on string variables.

Miscellaneous sub-panel

Item types: Text, Paragraph, Pixel, Line, Framed rectangle, Filled rectangle, Bitmap, Structure call, Indexed structure call and Variable.

Contains various special item flags:

- **Cursor field.** If checked the item is considered a cursor field by the target system. The cursor No. can be selected in an edit box to the right of the check box (only visible when the check box is checked). Cursor fields consume a sizeable amount of memory in the target, because a copy of the complete item with all parameters must be made. When creating C code easyGUI finds the highest cursor number in the project, and assigns space on the target system based on this number. Cursor numbers should therefore always be kept as low as possible, i.e. always start on zero, but a missing number in a set of cursor fields is handled correctly by the target system, thereby allowing dynamic parts of the display containing cursor fields to be invisible without corrupting the cursor system. Cursor field visibility can be checked in the left part of the display panel.

Active cursors are drawn on the target system by reversing foreground and background colors. It is therefore essential not to make cursor fields transparent, i.e. without background.

- **Auto redraw.** If checked instructs the target system to automatically refresh the item periodically. The timing is controlled by the target system, easyGUI just makes a copy of each Auto redraw item and inserts it in a list. Each time the target system `GuiLib_Refresh` function is called the Auto redraw items are checked, and maybe redrawn, depending on the setup described below. An Auto redraw item can be a single variable, or e.g. a structure call, consisting of perhaps a variable and its associated unit text.

Keep in mind that Auto redraw items should as a rule always redraw their background, because e.g. a transparent variable will end up looking like a black box drawing more and more foreground pixels on a background that never gets erased.

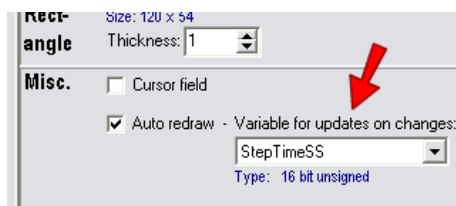
Another important thing to consider when designing structures with Auto redraw features is the fact that the Auto redraw item never gets recalculated, it is merely redrawn with the coordinates, colors, etc. calculated when its parent structure was initially displayed. So, if e.g. a coordinate is controlled by a variable it is *not* recalculated each time the `GuiLib_Refresh` function redraws the item. However, this fact can be circumvented, by making the Auto redraw item a structure call, which calls another structure containing e.g. variable controlled coordinates. This is because complete structures that are to be redrawn *do* get recalculated. It all stems from the fact that only the Auto redraw item is saved in the list of Auto redraw items, not its eventual underlying structures, which in principle could be a big construction of multi-level structures in structures.

There is two fundamentally different ways of using the Auto redraw feature:

- Continuous updating. All Auto redraw items are continuously updated, each time the `GuiLib_Refresh` function is called. This is the default setting.
- Update on changes. Auto redraw items are updated only if the controlling variable / variable to be displayed has changed, or if the item does not involve a variable.

The selection between these two methods is done in the Parameter window, under the Operations tab.



If Update on changes has been selected as the Auto redraw controlling method there is an additional way of controlling Auto redraw items:



This variable reference allows items not inherently using variables (all visible items except Indexed structure call and Variable items) to be controlled conditionally by a variable. A suitable variable is selected, and the Auto redraw item will then be updated each time the `GuiLib_Refresh` function detects that the variable value has changed.

- **Mark item as.** A special marking attribute can be assigned to an item:
 - Nothing ("-" setting). Normal setting.
 - **Scroll box.** Defines a scroll box for the target system scroll routines. Only makes sense when used on item types Framed rectangle, Filled rectangle, Structure call, Indexed structure call or Clipping rectangle.



- **Scroll bar.** Defines a scroll bar () for the target system scroll routines. Only makes sense when used on item types Framed rectangle, Filled rectangle, Structure call, Indexed structure call or Clipping rectangle. The scroll bar has a fixed appearance, apart from the scroll indicator (), with arrows at the top and bottom. The bar width is determined by the item defining the bar outline. In easyGUI the scroll indicator is always shown in the same vertical position. The indicator is only dynamic on the target system. Only vertical scroll bars are supported.
- **Scroll line.** Defines a scroll line for the target system scroll routines. This attribute is normally assigned to a structure call, where the structure called defines all fields and text of a single line in the scroll box.

When the Scroll line attribute is selected an extra edit box appears to the right, allowing selection of line height in the scroll box. This determines the distance between lines in the scroll box, and hence number of lines visible in the box. It is not necessarily the number of lines actually shown in the scroll box on the target system, it merely gives the maximum number of lines visible at any time. Only complete scroll lines are shown in a scroll box.

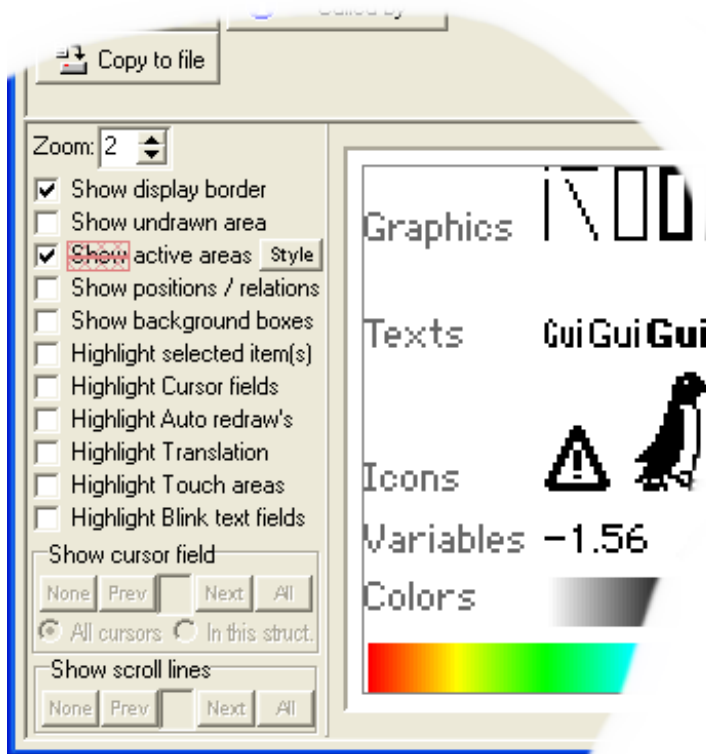
It will be convenient to specify a background box in the scroll line item (normally a structure call item) to make sure that a scroll line is completely redrawn each time the scroll box is updated.

Like cursor fields, the active scroll line is drawn on the target system by reversing foreground and background colors. In easyGUI any scroll line can be shown inversed, by using the "Show scroll lines" control in the Display panel.

DISPLAY PANEL

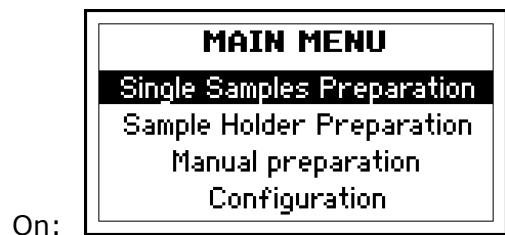
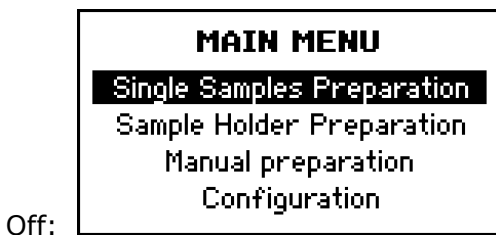
The display panel shows a representation of the target system display, lacking only certain dynamic features as e.g. scroll box operation. All pixels drawn corresponds 1:1 with the real display.

A cursor cross is shown, whenever the mouse is over the display area. Along with it the coordinates can be seen, both right next to the cross, and at the left of the panel:

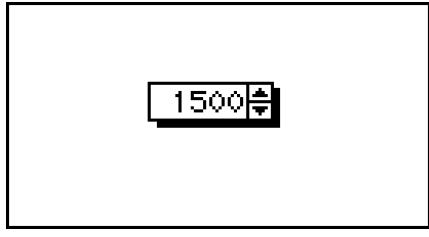


Along the left edge are a number of controls, allowing different views and help systems to be employed:

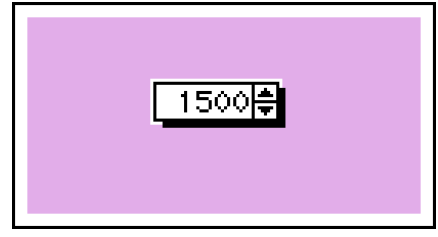
- **Zoom.** Can be set to 1x, 2x, 3x and 4x. The 1x setting maps PC screen pixels directly as target system pixels, creating a very small display.
- **Show display border.** If checked, a thin line is drawn around the active area of the display, making it easier to differentiate between active pixels and border area:



- **Show undrawn area.** Areas not touched by the current structure can be marked:



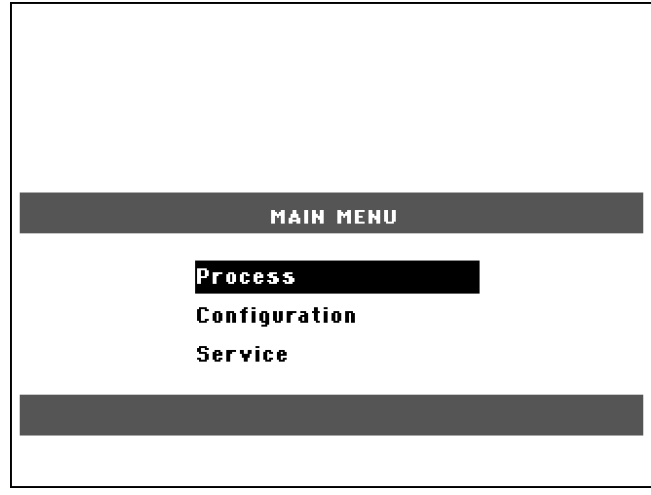
Off:



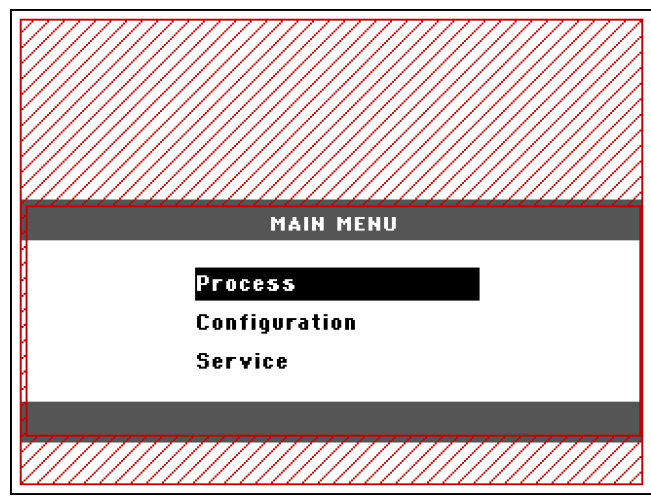
On:

The color used for the unmarked areas is defined in the Parameters window, Simulated colors tab page.

- Show active area.** Draws markings for areas of the display lying outside the active area. Both the general display active area (defined in the Parameters window, Basics tab page), and active areas defined through Active area items, are indicated. There are four different ways of indicators, with one pair using solid gray, and one pair using red hatching, and with one solid gray / red hatching pair showing behind the items, and one pair showing in front. The following example is a display, where a large part is physically masked out on the target system. The active area feature can therefore conveniently indicate the visible part of the display:

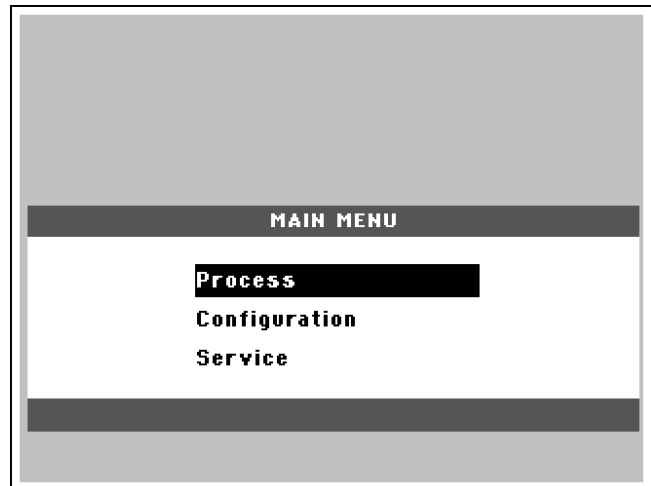


Off:

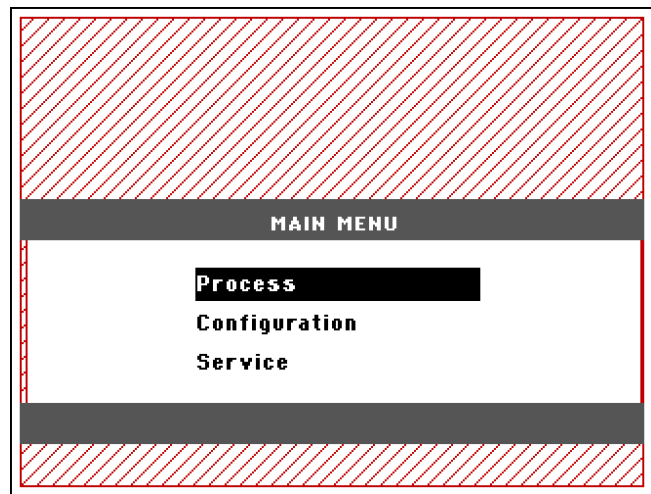


Red hatching, in front of items:

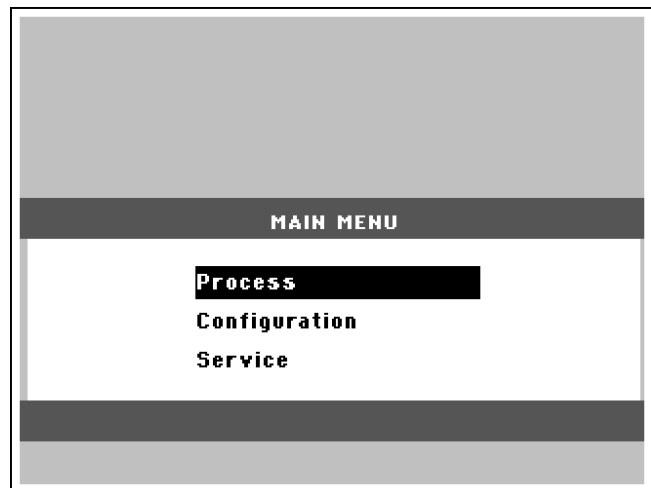
Solid gray, in front of items:



Red hatching, behind items:

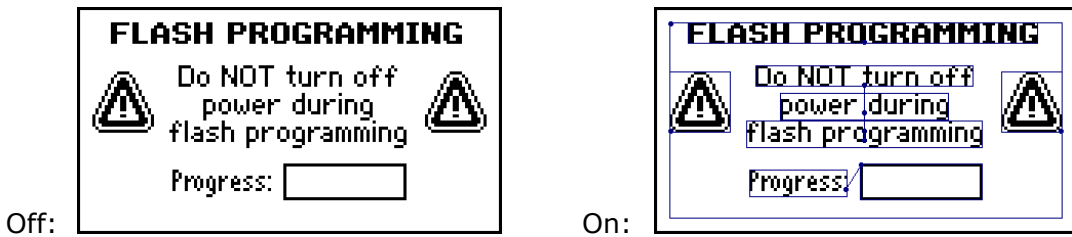


Solid gray, behind items:

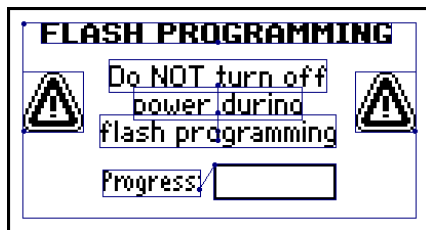


The small **STYLE** button next to the Show active areas checkbox cycle through the four possible indicator styles, in the order shown above.

- **Show positions / relations.** Draws blue boxes around all items, a little dot at the calculated position for each item, and draws interconnecting lines between items connected by relative coordinates:



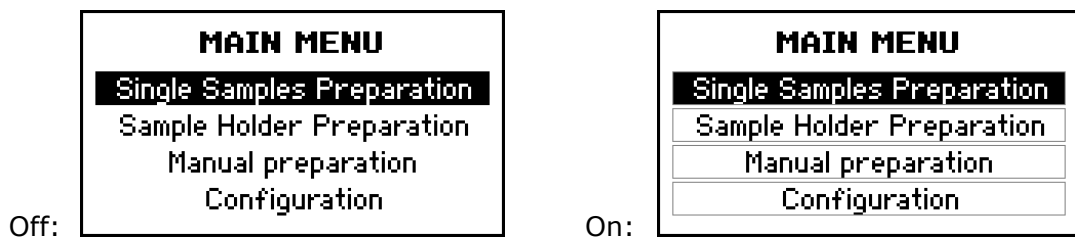
This structure starts with a white rectangle filling the entire display (in order to erase it), it can be seen as the outer thin rectangle with a little dot at the left top corner (The primary coordinate). The three middle texts ("Do NOT turn off", "power during" and "flash programming") are centered (note dots in the middle, at the Base lines), and the two lowest texts are placed relative to the top text (note interconnecting lines). Around each item is a box, surrounding all pixels belonging to the item. Note that the boxes around the three middle texts just grazes the character pixels, they doesn't indicate background extents. This is because the texts are drawn transparent, i.e. with no background. If the three texts had background drawing enabled (Back ground color = Pixel OFF) they would look like:



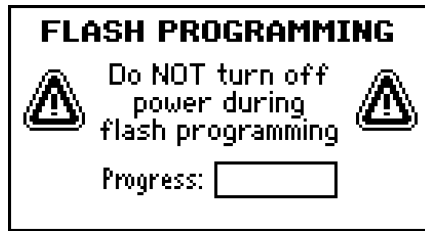
(look carefully, the differences are there!)

The boxes overlap because the texts have been placed rather close to each other. In fact, without transparent writing the "g" at the end of the middle text is partly cut off by the last text!

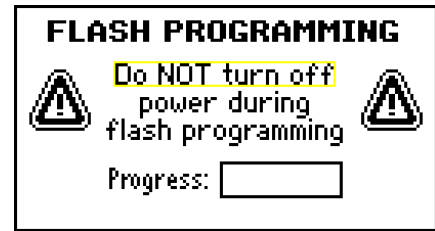
- **Show background boxes.** Draws grey boxes around all background boxes, showing their extends:



- **Highlight selected item(s).** The current item (or items) is highlighted by a yellow box (Maybe difficult to see in the example, it is the top of the three middle texts):

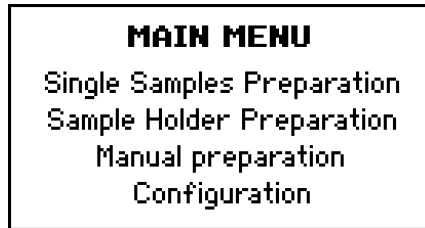


Off:

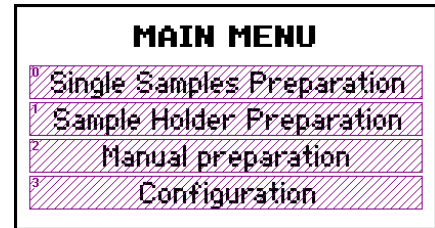


On:

- **Highlight cursor fields.** Every cursor field is highlighted by a purple box and shading, and a little number at the top left indicating the cursor number:

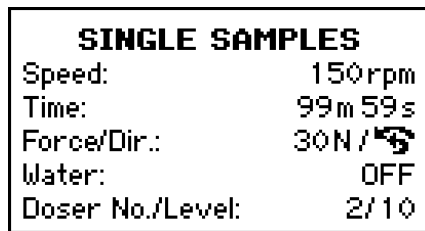


Off:

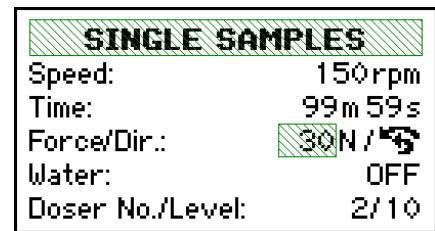


On:

- **Highlight auto redraw items.** Every auto redraw item is highlighted by a green box and shading:

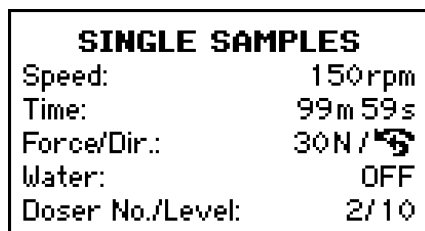


Off:

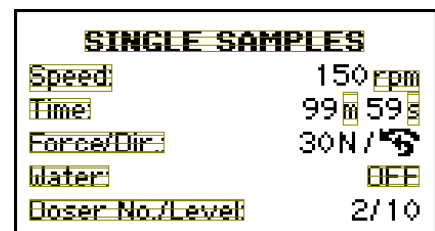


On:

- **Highlight translation.** Every text selected for translation is highlighted by a brown box and shading:

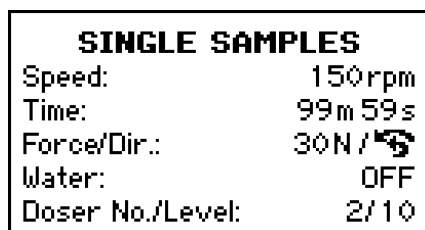


Off:

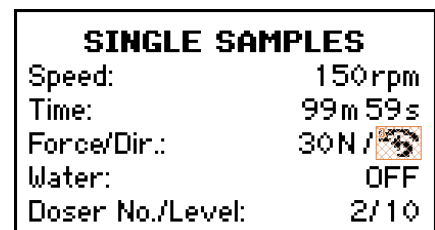


On:

- **Highlight touch areas.** Every touch area is highlighted by an amber box and shading:

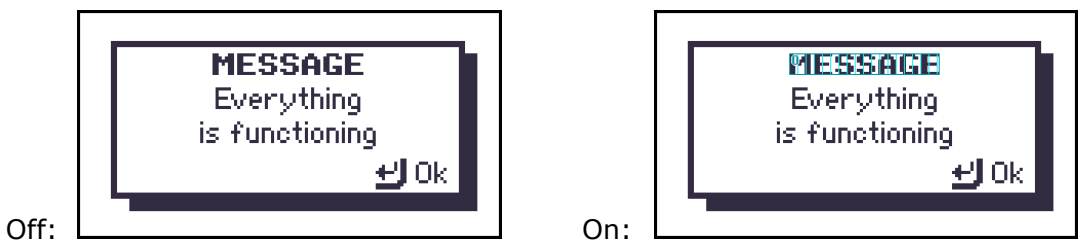


Off:

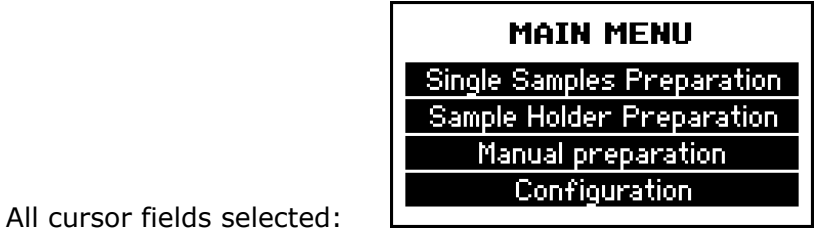
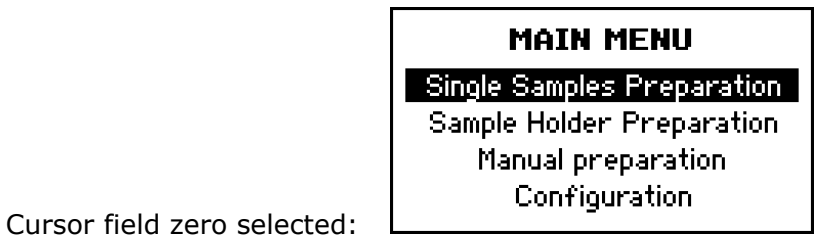
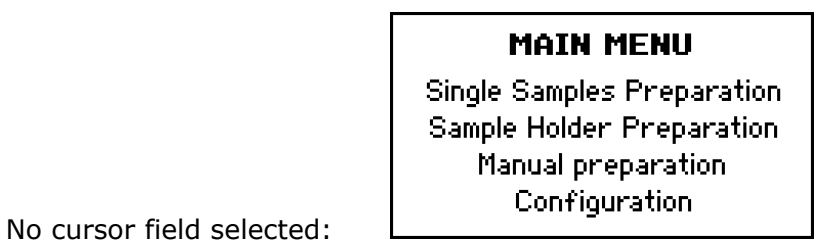


On:

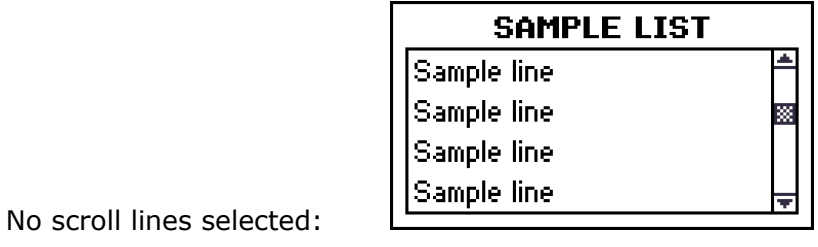
- **Highlight blink text fields.** Every item marked as a blinking item is highlighted by a cyan box and shading:



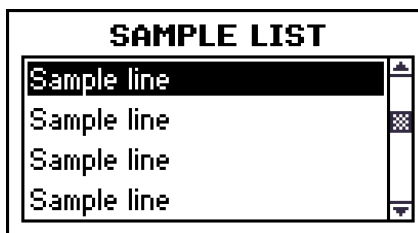
- **Show cursor field.** One or all cursor fields can be shown inversed, as on the target system:



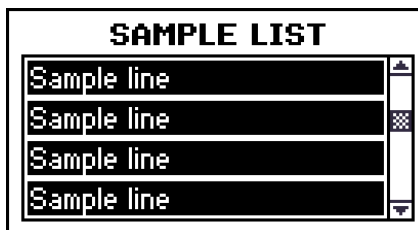
- **Show scroll lines.** One or all scroll lines can be shown inversed, as on the target system, in the same way as for cursor fields:



Scroll line zero selected:



All scroll lines selected:



The above settings may be combined as desired, but setting too many options on will make the display rather unreadable.

USE OF TOUCH AREAS

easyGUI separates the tasks of handling the touch interface hardware, and the actual handling of touch events.

easyGUI handles events from the touch interface, and handles eventual coordinate conversion from touch interface coordinates to display coordinates, should these differ.

In the following it is assumed that activating the touch interface produces an event with a coordinate of the touch position. This coordinate need not coincide with the display coordinates, but easyGUI needs to know how to convert from touch interface coordinates to display coordinates.

The proper sequence for implementing a touch interface is:

- 1 Touch interface hardware is tested and debugged.
- 2 easyGUI touch interface is trained in coordinate conversion, if needed.
- 3 Events from the touch interface hardware is fed to the proper easyGUI library routines, and easyGUI checks if any touch area falls under the position where a touch event happened. If so, the easyGUI library returns the touch area number, as set up in the Structure editor.

1 - Touch interface hardware

The touch interface hardware is not controlled by easyGUI, and can therefore be implemented in any way found suitable.

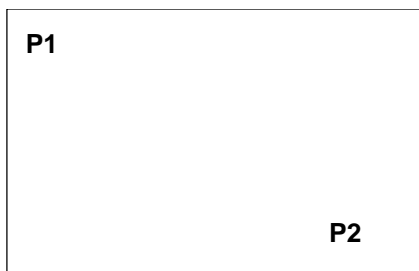
2 - Coordinate training

The touch interface coordinate training only needs to be accomplished at system power on. Preferably the values should be stored, so that coordinate training can be reduced to a minimum.

On systems where the touch hardware coordinates and display coordinates coincide by definition, training is not necessary.

There are two variants of coordinate training:

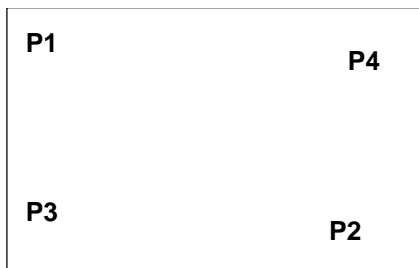
- **Two diagonal corner coordinate positions.** Correspondence between touch interface coordinates and display coordinates are defined for two diagonally opposing positions:



The positions can occupy any two opposing corners.

In this conversion mode coordinates are converted individually in the X and Y directions, i.e. X coordinate conversion is not affected by the Y coordinate, and vice versa.

- **Four corner coordinate positions.** Correspondence between touch interface coordinates and display coordinates are defined for four positions, one near each corner:



In this conversion mode when coordinates are converted the X conversion factor is affected by the Y coordinate, and vice versa. Four corner conversion mode therefore results in superior conversion accuracy, compared to two corner conversion mode. However, if it is guaranteed by the touch interface hardware that the X and Y coordinate directions are precisely as the display coordinate directions (i.e. no tilting), only two corner conversion mode is necessary.

The positions should be placed as near the corners as possible. Supplying positions lying nearer to the display center will reduce coordinate conversion precision.

Touch interface coordinates must lie in the range -32768 ~ 32767.

The coordinate training is accomplished by calling the `GuiLib_TouchAdjustReset` and `GuiLib_TouchAdjustSet` functions. `GuiLib_TouchAdjustReset` resets any previous conversion setup, and should always be called before supplying coordinates for the conversion function. `GuiLib_TouchAdjustSet` is then called two or four times, depending on the desired conversion strategy.

Two corner adjustment example:

```
:
GuiLib_TouchAdjustReset();
GuiLib_TouchAdjustSet( 12, 13, 160, 80);
GuiLib_TouchAdjustSet(230, 11, 2240, 40);
:
```

The touch interface coordinates (160,80) corresponds to display coordinates (12,13), while touch interface coordinates (2240,40) corresponds to display coordinates (230,11), i.e. upper left and lower right corners have been specified.

Four corner adjustment example:

```
:
GuiLib_TouchAdjustReset();
GuiLib_TouchAdjustSet( 12, 13, 16, 16);
GuiLib_TouchAdjustSet(230, 11, 224, 8);
GuiLib_TouchAdjustSet( 12, 119, 13, 115);
GuiLib_TouchAdjustSet(233, 121, 236, 117);
:
```

The touch interface coordinates (16,16) corresponds to display coordinates (12,13), touch interface coordinates (224,8) corresponds to display coordinates (230,11), touch interface coordinates (13,115) corresponds to display coordinates (12,119), and finally touch interface coordinates (236,117) corresponds to display coordinates (233,121).

The ordering of the `GuiLib_TouchAdjustSet` function calls is irrelevant.

For a simple system where touch and display coordinates are always in agreement, only `GuiLib_TouchAdjustReset` needs to be called. Strictly speaking even this call is not necessary, because it is also part of the `GuiLib_Init` function, which should always be called when initializing the system.

3 - Event handling

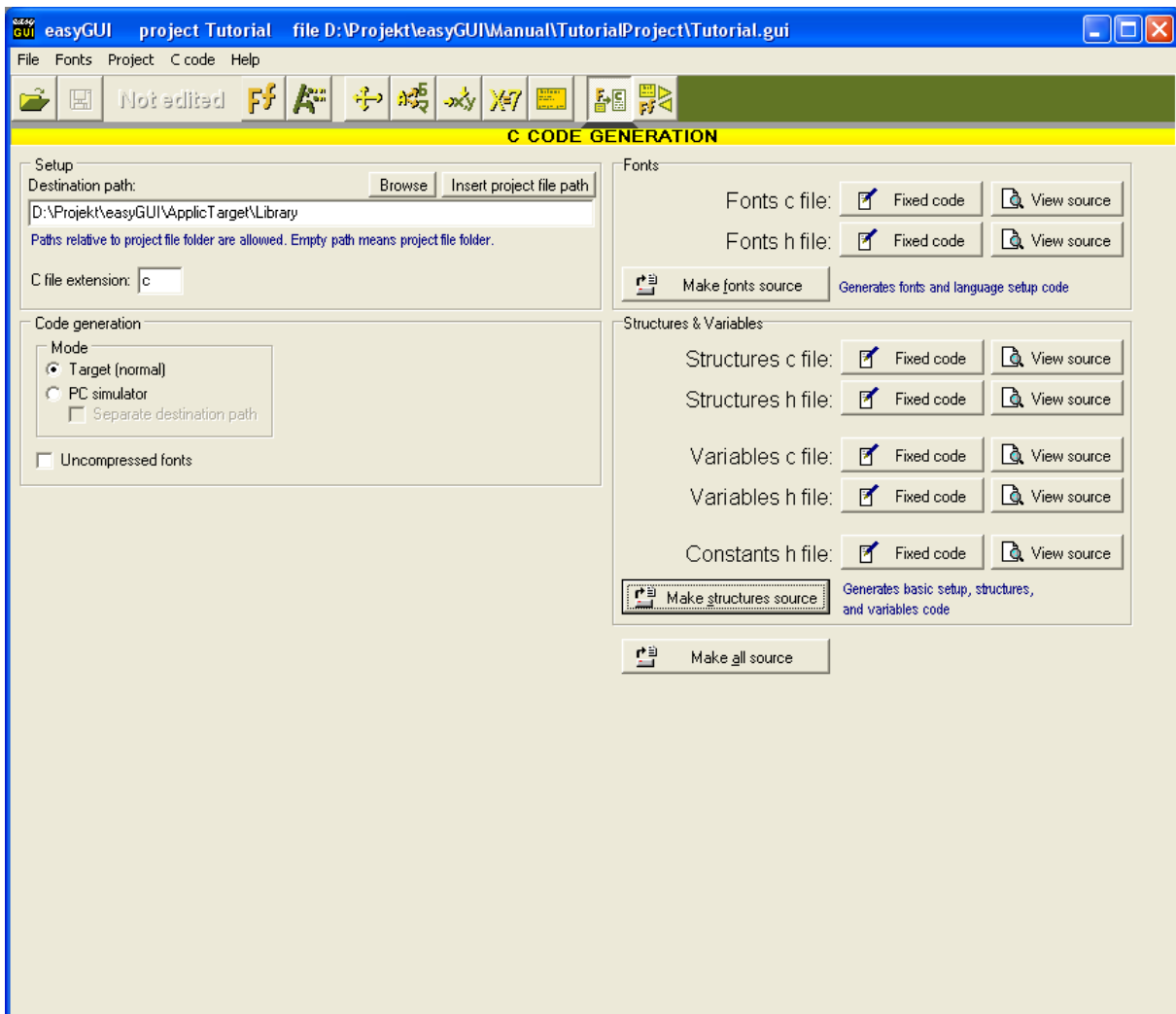
Each time the `GuiLib_ShowScreen` function is used to show an easyGUI structure the currently registered touch areas are lost. New touch areas found in the structure being displayed are remembered in a list.

When an event from the touch interface hardware is detected the `GuiLib_TouchCheck` function must be called. It shall be supplied with the touch event coordinates for the event (in touch interface hardware coordinates). `GuiLib_TouchCheck` first converts the touch event coordinates to display coordinates, using the conversion strategy specified

earlier. It then searches through the current list of touch areas, checking if the coordinate position lies inside one of the touch areas. The first touch area found to include the event coordinate is selected as a hit, and its touch area number returned by the `GuiLib_TouchCheck` function. If `GuiLib_TouchCheck` did not find any touch areas at the touch event coordinates it returns -1.

12 C CODE GENERATION

The final stage in easyGUI is generation of actual target system C code. This is accomplished in the code generation window:



On the left the **destination path** can be set. A partial path may be entered, in which case it is taken as relative to the folder in which the project file (*.gui) resides. The path box may also be left empty, in which case all target system files are placed in the project file folder. The **BROWSE** button allows selection of any folder, while the **INSERT PROJECT FILE PATH** button simply inserts the path to the project file, making it easy to edit it.

C file extension selects the file extension for C files. Most compilers use `c`, while some C++ compilers use `cpp`. Interface files are always treated as having extension `h`.

Below is a selection between normal mode and PC simulator mode:

- **Target (normal)**. This setting is used for all normal C code generation for the target system.
- **PC simulator**. This setting shall only be used when generating code for the easyGUI PC Simulation Toolset (see the easyGUI PC Simulation Toolset chapter). The setting overrides some of the compiler setup settings, in order to easily produce code suited for PC usage. To make sure C code generating is not left in this setting when intending to generate C code for the target system a small warning (⚠) is shown.

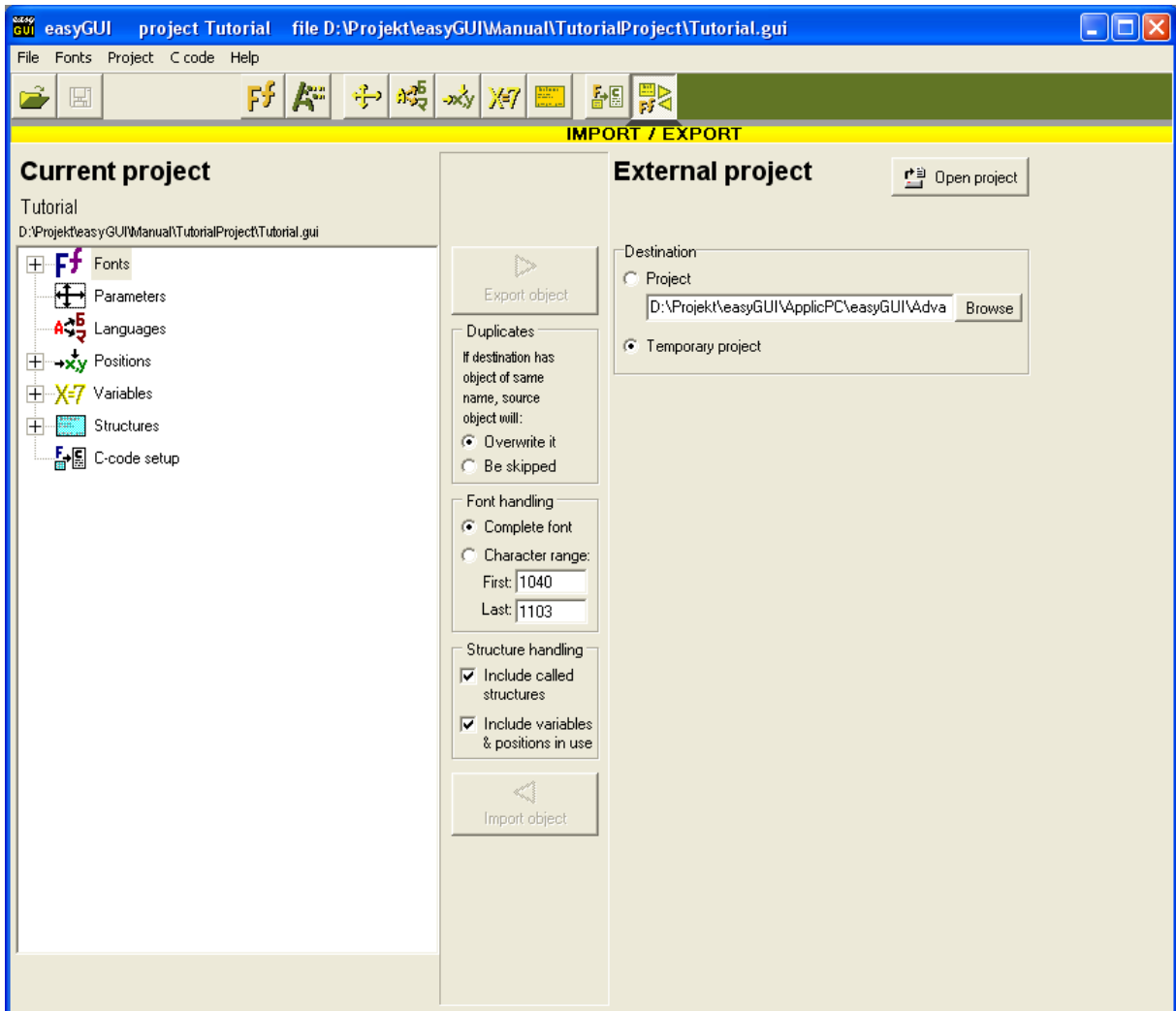
The target system path can optionally be different for normal and PC simulator modes. This is often handy.

The **Uncompressed font** option instructs easyGUI to generate font data uncompressed, i.e. all characters in a font takes up the same space. Normally this option should be left unchecked, ensuring that easyGUI will compress all font data as much as possible, saving considerable ROM space on the target system. The only reason for using uncompressed font data is if manual manipulation of font data is needed in the target system, e.g. when dynamically reading in font data during execution.

At the right are two panels named "**Fonts**" and "**Structures & variables**". Under most circumstances only structure data, and perhaps variable data, has been edited, and generation of font data is therefore not needed. Each of the three types of data are placed in its own set of c and h files, called `GuiFont.c` and `GuiFont.h`, `GuiStruct.c` and `GuiStruct.h`, and finally `GuiVar.c` and `GuiVar.h`. These files should be included in the C compiler setup. The content of each file is generated on request by pressing the appropriate button. Fixed code can be added at the beginning, and at the end of each file. This code is entered using the buttons to the right. The code could be compiler directives, include directives, copyright notices, etc. The code is saved in the easyGUI database.

13 IMPORT / EXPORT

The import / export function allows the copying of data between easyGUI projects. The window is divided into three parts:



On the left is the current project.

In the middle is a number of controls and settings for the import / export process.

On the right is the external project, closed in the above example.

CURRENT PROJECT PANEL

The current project panel is always open, and displays all components of the project as a tree:

- **Fonts.** This branch can be expanded, showing each individual font in the project.
- **Parameters.** Contains all basic project setup, like display size, compiler settings, etc.
- **Languages.** Contains all defined languages. Only language setup is included, not translated texts. Texts are part of the structures.
- **Positions.** This branch can be expanded, showing each individual fixed position in the project.
- **Variables.** This branch can be expanded, showing each individual variable in the project.
- **Structures.** This branch can be expanded, showing each individual structure in the project.
- **C-code setup.** All fixed headers / footers, and other C-code generation setup.

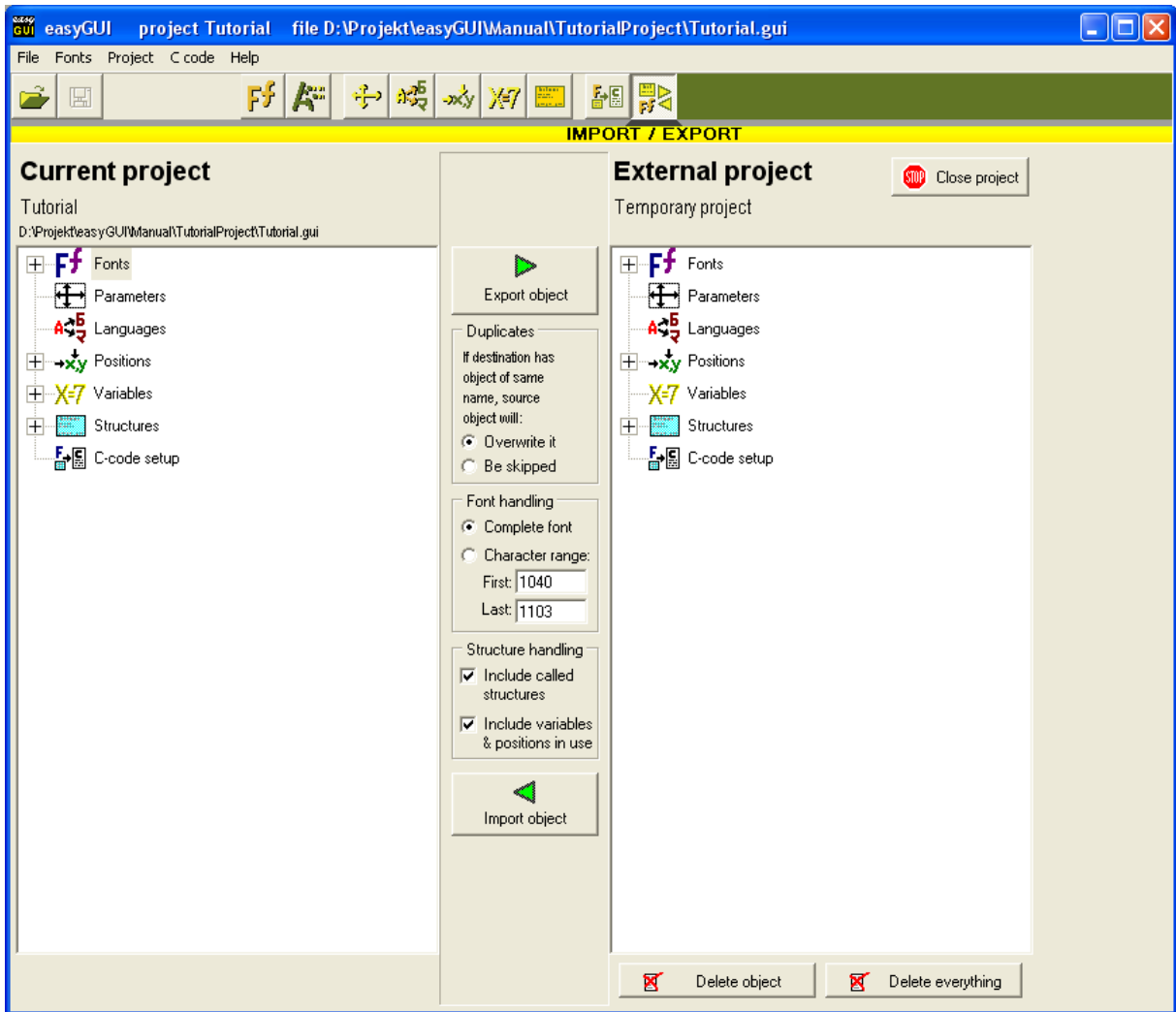
A single component (Parameters, a single font, etc.) is marked by clicking it. Several components are marked by holding the Ctrl button, and clicking the desired components. Fonts, positions, variables, and structures can be marked *en masse*, by clicking the root of the component type.

EXTERNAL PROJECT PANEL

The external project can be of two variants:

- **Project.** Another project, just like the current project.
- **Temporary project.** This is a special temporary holder of data, always present.

Before importing or exporting can happen the external project must be opened, by pressing the **OPEN PROJECT** button. This sets up the right panel in the same way as the left panel:



Only difference is that two buttons are present below the tree:

- **DELETE OBJECT** deletes the selected object.
- **DELETE EVERYTHING** deletes all objects.

These two buttons are only relevant for the temporary project. They allow the temporary project to be cleaned for objects. Observe that Parameters and C-code setup cannot be deleted, these types of objects are always present in a project.

The **CLOSE PROJECT** button at the top closes the project, and returns to the initial display, where selection between project / temporary project can be made.

MIDDLE PANEL - CONTROLS AND SETTINGS

The middle panel controls the importing / exporting. A number of controls and settings are available:

- **EXPORT OBJECT** button starts exporting of objects marked in the left panel. Objects marked in the right panel are irrelevant.
- **Duplicates.** Determines how duplicates are treated. They can be either overwritten, or skipped. This setting is irrelevant for Parameters and C-code setup, as these types of objects are always present in a project.
- **Font handling.** When importing / exporting fonts either the complete font, or only a subset of characters, can be copied.
- **Structure handling.** When importing / exporting structures it can be selected whether other structures used by the selected structures shall also be included in the copying. Furthermore, positions and variables used by the structures can also be included in the copying.
- **IMPORT OBJECT** button starts importing of objects marked in the right panel. Objects marked in the left panel are irrelevant.

14 HOW TO SET UP YOUR SYSTEM

MINIMUM RAM AND ROM REQUIREMENTS

Because easyGUI is a purely graphic system it must have access to a reasonable amount of RAM and ROM to function properly. How much RAM and ROM cannot be stated explicitly, because it depends on the complexity of the user interface build in easyGUI, but a couple of approximate levels can be stated:

- **RAM usage:** $2\text{KB} + (\text{Display width} \times \text{Display height} \times \text{Bits per color}) / 8$.

Examples:

- 128×64 pixels monochrome: $2\text{KB} + (128 \times 64 \times 1) / 8 \approx 3\text{KB}$
- 240×128 pixels monochrome: $2\text{KB} + (240 \times 128 \times 1) / 8 \approx 6\text{KB}$
- 320×240 pixels monochrome: $2\text{KB} + (320 \times 240 \times 1) / 8 \approx 12\text{KB}$
- 128×64 pixels 16 color: $2\text{KB} + (128 \times 64 \times 4) / 8 \approx 4\text{KB}$
- 240×128 pixels 256 color: $2\text{KB} + (240 \times 128 \times 8) / 8 \approx 33\text{KB}$

- **ROM usage:** 21KB + font data + structure data.

Examples:

- Low complexity GUI (50 structures), 2 full text fonts, 1 partial big font, 2 icon fonts: 30KB~50KB depending on display size.
- Medium complexity GUI (250 structures), 2 full text fonts, 2 partial big fonts, 4 icon fonts: 60KB~100KB depending on display size.
- High complexity GUI (400 structures), 4 full text fonts, 2 partial big fonts, 6 icon fonts: 100~150KB depending on display size.

These sizes are by no means definitive, they are only meant as a rough guideline.

OPERATING SYSTEM

easyGUI places only limited demands on the core of your target system. It can function with systems not having an operating kernel at all, up to systems employing a full-blown operating system. Its only demand is some kind of systematic calling based on a timer, to let easyGUI process the various kinds of dynamic operations:

- Low level drawing.
- High level structure drawing.

- Auto updating of fields.
- Cursor drawing.
- Blinking items.
- Scrolling.

Your target only needs to call a single easyGUI function regularly:

```
GuiLib_Refresh();
```

This routine handles all the activities mentioned above in easyGUI. Calling the refresh function more often leads to a more responsive system, but above a certain point there is no further advantage in increasing the frequency of calling. In most systems it will suffice to call the refresh function 5 times a second, i.e. every 200ms. Do not call it more often than every 10ms, and only so often on powerful systems with lots of available resources. 5 times a second may sound slow, but this is often fast enough, and ensures that the user experiences an adequately responsive system.

It is also feasible to call `GuiLib_Refresh()` on demand, e.g. after each new structure is displayed, and after any change of variables that affects structures. This is marginally quicker than the above described approach, but is not as elegant, and more error-prone.

SETTING UP THE SYSTEM FOR EASYGUI USE

Before easyGUI can be of any use your target system must be set up. The following items must be successfully implemented:

- 1 Physical display connection.
- 2 Setting up easyGUI for your display type.
- 3 Display control functions.
- 4 Compiling the project.
- 5 easyGUI interfacing.

When these items are properly implemented you are ready to start developing your GUI - your very own Graphical User interface.

1 - Physical display connection

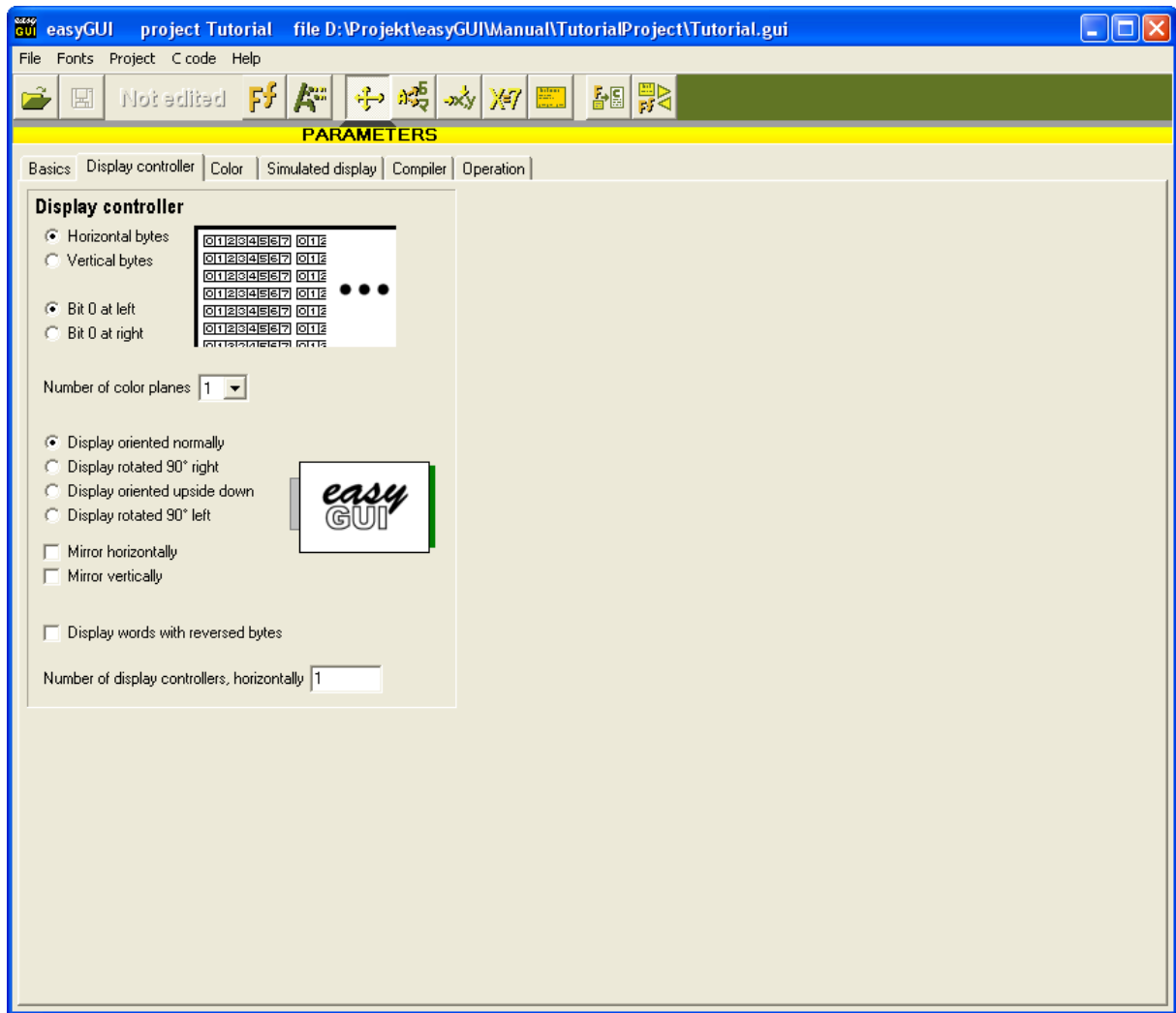
The display can be connected using port access, direct memory access, or a combination hereof. Most small displays are simply connected via a number of ports, but the most efficient connection depends on your actual hardware. It is irrelevant to easyGUI how this connection is made, as long as a single requirement is satisfied: easyGUI must be able to

address individual pixels on the display, by sending display RAM contents from its own display buffer in normal system RAM to the internal display controller RAM buffer.

It is beyond the scope of this manual to give details on how to make a proper connection of the display, so this issue will not be covered further.

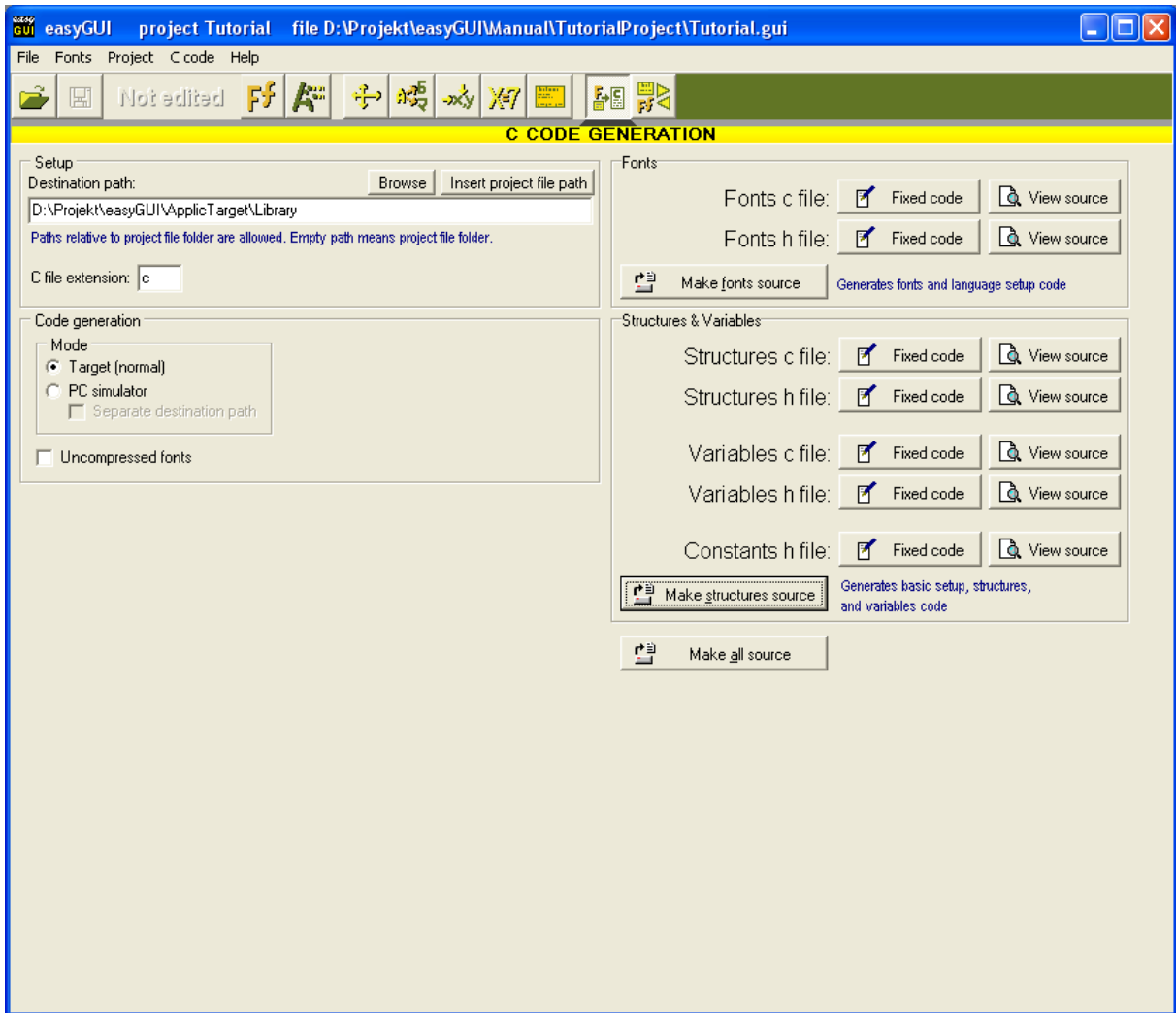
2 - Setting up easyGUI for your display type

It is essential that easyGUI is properly configured for your specific type of display and compiler. Enter the Parameters window:



Enter proper values in the various parameter fields. The parameters are explained in detail in the Project parameters chapter. Observe that there are multiple tabs with parameters.

After setting the values go into C code generation:



- and select Make All. The units `GuiConst.h`, `GuiFont.c/h`, `GuiVar.c/h` and `GuiStruct.c/h` are created. The important unit initially is the `GuiConst.h` unit, which contains the basic easyGUI settings of your system.

3 - Display control functions

The software must be able to control the display. This is done in the `GuiDisplay` unit.

The following actions must be implemented:

- Display initialization.
- Display writing.
- Light and contrast control.

easyGUI does not require display reading.

Display initialization

The display must be properly initialized. This involves:

- Setting up ports and/or addressing.
- Enabling the display.
- Selecting a purely graphics mode.
- Setting start address and address range.
- Initializing display memory.

These initial activities cannot be supplied by easyGUI in a ready-to-use form, as the actual routine depends on the type of display controller. The supplied `GuiDisplay` unit in the easyGUI library folder must therefore be edited to fit the selected display controller in your target system. Make a copy of the supplied `GuiDisplay` unit into your target system source code folder, and make adjustments to the `GuiDisplay_Init()` function as required.

Selecting a display driver

At the top of `GuiDisplay.c` is a number of compiler directives. One of them should be activated, corresponding to the desired display driver. When the target system is up and running the other display drivers may be deleted, if desired.

LH75401 display driver

Description:	This driver uses one LH75401 controller with built-in LCD interface. Frame buffer is for 16 bit memory width. Horizontal resolution must be divisible by 16. Graphic modes up to 640x480 pixels.
Compatible display controllers:	LH75400, LH75410, LH75411.
easyGUI setup:	Horizontal bytes. Bit 0 at right. Number of color planes: 1. Color mode: Direct color mode. Color depth: 12 bits. Display orientation: As you like. Display words with reversed bytes: Off. Number of display controllers, horizontally: 1.
Remarks:	-

PCF8548 display driver

Description:	This driver uses one PCF8548 controller with on-chip generation of LCD supply and bias voltages. LCD
--------------	--

display in I2C bus interface. Graphic modes up to 102x65 pixels.

Compatible display controllers:

-

easyGUI setup:

Vertical bytes.
 Bit 0 at bottom.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks:

Port addresses (P_x) must be altered to correspond to your μ -processor hardware and compiler syntax.

T6963 display driver

Description:

This driver uses one T6963 controller. LCD display in 8-bit parallel interface. Graphic modes up to 80x32 pixels. Combination of number of columns and number of lines must not cause the frequency to exceed 5.5MHz.

Compatible display controllers:

AX6963, WG24064

easyGUI setup:

Vertical bytes.
 Bit 0 at bottom.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks:

Port addresses (_Pxx) must be altered to correspond to your μ -processor hardware and compiler syntax.

SED1335 display driver

Description:

This driver uses one SED1335 display controller. LCD display in 8 bit parallel interface. Graphic modes up to 640x256 pixels.

Compatible display controllers:

S1D13700, S1D13305, RA8835

easyGUI setup:

Horizontal bytes.
 Bit 0 at right.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).

Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: Port addresses (Px) must be altered to correspond to your μ -processor hardware and compiler syntax.

HD61202 display driver

Description: This driver uses two HD61202 display controllers. LCD display in 8 bit parallel interface. Graphic modes up to 128x64 pixels.

Compatible display controllers: KS0108B/KS0107B, AX6108/AX6107, NT7108/NT7107, S6B0108A/S6B0107A, S6B0108/S6B0107, S6B0708/S6B0707, KS0708/KS0707, S6B0708/S6B0707, S6B2108/S6B2107

easyGUI setup: Vertical bytes.
 Bit 0 at top.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 2.

Remarks: Port addresses (Px) must be altered to correspond to your μ -processor hardware and compiler syntax.

SSD1815 display driver

Description: This driver uses one SSD1815 display controller with on-chip oscillator. LCD display in SPI interface. Graphic modes up to 135x65 pixels.

Compatible display controllers: KS07XX, S1D10605, S1D10606, S1D10607, S1D10608, S1D10609, S1D15600, S1D15601, S1D15602, S1D15605, S1D15705, S1D15707, S1D15708, S1D15710, SED1565, S1D15605, IST3015, NJU6570, NJU6575, NJU6673, NJU6675, NJU6676, NJU6677, NJU6678, NJU6679, NT7501, NT7502, NT7532, NT7534, ST7565S, KS0713, KS0715, KS0717, KS0718, KS0719, KS0723, KS0724, KS0728, KS0741, KS0755, KS0759, S6B0713, S6B0715, S6B0716, S6B0717, S6B0718, S6B0719, S6B0721, S6B0723, S6B0724, S6B0725, S6B0728, S6B0755, S6B0759, S6B1713, SSD1805, SSD1852, SPLC501, TL0313, UC1606.

easyGUI setup: Vertical bytes.

Bit 0 at top.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: function Spi_SendData must be altered to correspond to your μ -processor hardware and compiler syntax.

SSD0323 display driver

Description: This driver uses one SSD032 display controller with on-chip oscillator. OLED/PLED display in 8 bit parallel interface. Graphic modes up to 128x80 pixels.

Compatible display controllers: SSD01303

easyGUI setup: Horizontal bytes.
 Bit 0 at right.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 4 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: Port addresses (Px.x) must be altered to correspond to your μ -processor hardware and compiler syntax.

S1D13505 display driver

Description: This driver uses one S1D13505 display controller. LCD display in memory mapped interface. Graphic modes up to 800x600 pixels.

Compatible display controllers: -

easyGUI setup: Vertical bytes.
 Bit 0 at bottom.
 Number of color planes: 1.
 Color mode: Direct color mode or via palette.
 Color depth: up to 16 bits.
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: -

S1D13700 display driver

Description: This driver uses one S1D13700 display controller. LCD display in 8 bit parallel interface. Graphic modes up to 640x240 pixels.

Compatible display controllers: -

easyGUI setup: Horizontal bytes.
Bit 0 at right.
Number of color planes: 1.
Color mode: Grayscale.
Color depth: 1 bit (B/W).
Display orientation: As you like.
Display words with reversed bytes: Off.
Number of display controllers, horizontally: 1.

Remarks: -

S1D13705 display driver

Description: This driver uses one S1D13705 display controller. LCD display in memory mapped interface. Graphic modes up to 800x600 pixels.

Compatible display controllers: SED1375, S1D13706, SED1374, S1D13704, SSD1906, SSD1905, S1D13A04.

easyGUI setup: Vertical bytes.
Bit 0 at bottom.
Number of color planes: 1.
Color mode: Via palette index.
Color depth: 8 bits.
Display orientation: As you like.
Display words with reversed bytes: Off.
Number of display controllers, horizontally: 1.

Remarks: -

S1D13706 display driver

Description: This driver uses one S1D13706 display controller. LCD display in memory mapped interface. Graphic modes up to 320x240 pixels.

Compatible display controllers: SED1375, S1D13705, SED1374, S1D13704, SSD1906, SSD1905, S1D13A04.

easyGUI setup: Vertical bytes.
Bit 0 at bottom.
Number of color planes: 1.
Color mode: Via palette index.

Color depth: 8 bits.
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: Make sure 2ms elapses from system power on, before init routine is executed.

S1D13A04 display driver

Description: This driver uses one S1D13A04 display controller. LCD display in memory mapped interface. Graphic modes up to 320x240 pixels.

Compatible display controllers: -

easyGUI setup: Vertical bytes.
 Bit 0 at bottom.
 Number of color planes 1.
 Color mode: Direct or via palette index.
 Color depth: 8, 16 bits.
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontal: 1.

Remarks: -

LH155BA display driver

Description: This driver uses one LH155BA display controller. LCD display in 8 bit parallel interface. Graphic modes up to 128x64 pixels.

Compatible display controllers: -

easyGUI setup: Horizontal bytes.
 Bit 0 at right.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: Port addresses (PTT) must be altered to correspond to your μ -processor hardware and compiler syntax.

ST7529 display driver

Description: This driver uses one ST7529 display controller. LCD display in 8 bit parallel interface. Graphic modes up to 255x160 pixels. It uses 3 bytes for 3 pixels mode.

Compatible display controllers: -

easyGUI setup: Vertical bytes.
Bit 0 at bottom.
Number of color planes: 1.
Color mode: Grayscale.
Color depth: 5 bit (B/W).
Display orientation: As you like.
Display words with reversed bytes: Off.
Number of display controllers, horizontally: 1.

Remarks: Port addresses (Pxx.x) must be altered to correspond to your μ -processor hardware and compiler syntax.

S6B0741 display driver

Description: This driver uses one S6B0741 display controller with on-chip oscillator. LCD display in 8 bit parallel interface. Graphic modes up to 128x129 pixels.

Compatible display controllers: -

easyGUI setup: Vertical bytes.
Bit 0 at top.
Number of color planes: 1.
Color mode: Grayscale.
Color depth: 2 bits.
Display orientation: As you like.
Display words with reversed bytes: Off.
Number of display controllers, horizontally: 1.

Remarks: Port addresses (Pxx.x) must be altered to correspond to your μ -processor hardware and compiler syntax.

NJU6450A display driver

Description: This driver uses two NJU6450A display controllers, with each controlling half (left or right) of the display. LCD display in 8-bit parallel interface. Graphic modes up to 122x32 pixels.

Compatible display controllers: SED1520, AX6120, NJU6450, NJU6452, PT6520

easyGUI setup: Vertical bytes.
Bit 0 at top.
Number of color: planes 1.

Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 2.

Remarks: Port addresses (_P00, _P01, etc.) must be altered to correspond to your μ -processor hardware and compiler syntax.

UC1608 display driver

Description: This driver uses one UC1608 display controller. LCD display in 8 bit parallel interface. Graphic modes up to 240x128 pixels. Vertical resolution can be 96 or 128 pixels.

Compatible display controllers: -

easyGUI setup: Vertical bytes.
 Bit 0 at top.
 Number of color planes: 1.
 Color mode: Grayscale.
 Color depth: 1 bit (B/W).
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.

Remarks: Port addresses (Pxx.x) must be altered to correspond to your μ -processor hardware and compiler syntax.

μ PD161607 display driver

Description: This driver uses one μ PD161607 display controller. Serial SPI interface mode 2 is used for commands. Frame transfer is used for display data. Graphic modes up to 320x240 pixels.

Compatible display controllers: -

easyGUI setup: Horizontal bytes.
 Bit 0 at right.
 Number of color planes: 1.
 Color mode: Direct color mode.
 Color depth: 18 bits.
 Display orientation: As you like.
 Display words with reversed bytes: Off.
 Number of display controllers, horizontally: 1.
 RGB format depending on hardware wiring (e.g. BBBB $\div\div$ GGGGG $\div\div$ RRRRR $\div\div$).

Remarks: There should be a 1ms delay both before and after resetting the μ PD161607. Spi_SendData function must be altered to correspond to your μ -processor hardware and compiler syntax. It shall send one byte to the display controller. Enter suitable delay code for the 30 μ s and 20ms wait periods. Make sure 2ms elapses from system power on, before init routine is executed.

RA8822 display driver

Description: This driver uses one RA8822 display controller with built-in PLL module. LCD display in 8 bit parallel interface. Graphic modes up to 240x160 pixels.

Compatible display controllers: RA8803

easyGUI setup: Horizontal bytes.
Bit 0 at right.
Number of color planes: 1.
Color mode: Grayscale.
Color depth: 1 bit(B/W).
Display orientation: As you like.
Display words with reversed bytes: Off.
Number of display controllers, horizontally: 1.

Remarks: Port addresses (Px.x) must be altered to correspond to your μ -processor hardware and compiler syntax.

New drivers are constantly added to `GuiDisplay.c`, so if your display controller of choice is not found in the above list it may still be included in `GuiDisplay.c`. If not, contact easyGUI support.

Display writing

The `GuiDisplay_Refresh()` function transfers data from easyGUI's internal display buffer in system RAM to the display controller's own internal RAM. This data transfer is kept at a minimum, because most display controllers are rather slow to access. easyGUI therefore checks which parts of the display has been altered since the last display data transfer, and then only transfers the altered data. This data checking is done on a scan line level, ensuring a very efficient system.

The function must go through all scan lines (horizontal or vertical, depending on display controller type), and for each scan line transfer data from a starting position to an ending position, if anything has changed on that particular scan line. The basic skeleton of the function should therefore not be altered, only the actual display data writing. The skeleton looks like:

```
void GuiDisplay_Refresh(void)
{
    GuiConst_INT16S X,Y;
```

```

GuiConst_INT16S LastByte;
GuiConst_INT16U Address;

// Lock GUI resources
GuiDisplay_Lock ();

// Walk through all lines
for (Y = 0; Y < GuiConst_BYTE_LINES; Y++)
{
    if (GuiLib_DisplayRepaint[Y].ByteEnd >= 0)
        // Something to redraw in this line
        {
            // Set address Pointer
            Address = Y * GuiConst_BYTES_PR_LINE +
                GuiLib_DisplayRepaint[Y].ByteBegin;
            :
            Some display controller specific code
            :

            // Write display data
            :
            Some display controller specific code
            :

            // Reset repaint parameters
            GuiLib_DisplayRepaint[Y].ByteEnd = -1;
        }
}

// Free GUI resources
GuiDisplay_Unlock ();
}

```

Two parts of the routine are display controller specific, one that sets up the correct display controller RAM address, and one that writes display data to the display controller RAM.

Some displays uses more than one display controller, a very widespread example is 128×64 pixels displays using the HD61202 display controller (or similar, a large number of variants exists). This display controller type can handle 64×64 pixels, and two display controllers are therefore employed. The `GuiDisplay_Refresh()` function then looks like:

```

void GuiDisplay_Refresh(void)
{
    GuiConst_INT16S LineNo;
    GuiConst_INT16S LastByte;
    GuiConst_INT16S N;

    // Lock GUI resources
    GuiDisplay_Lock ();

    // Walk through all lines
    for (LineNo = 0; LineNo < GuiConst_BYTE_LINES; LineNo++)
    {
        if (GuiLib_DisplayRepaint[LineNo].ByteEnd >= 0)

```

```

    // Something to redraw in this line
    {
    if (GuiLib_DisplayRepaint[LineNo].ByteBegin <
        GuiConst_BYTES_PR_SECTION)
        // Something to redraw in first section
        {
        // Select controller
        ControllerSelect(1);

        // Set address Pointer
        :
        Some display controller specific code
        :

        // Write display data
        :
        Some display controller specific code
        :

        // Reset repaint parameters
        if (GuiLib_DisplayRepaint[LineNo].ByteEnd >=
            GuiConst_BYTES_PR_SECTION)
            // Something to redraw in second section
            GuiLib_DisplayRepaint[LineNo].ByteBegin =
                GuiConst_BYTES_PR_SECTION;
        else // Done with this line
            GuiLib_DisplayRepaint[LineNo].ByteEnd = -1;
        }
    }

    if (GuiLib_DisplayRepaint[LineNo].ByteEnd >= 0)
        // Something to redraw in second section
        {
        // Select controller
        ControllerSelect(2);

        // Set address Pointer
        :
        Some display controller specific code
        :

        // Write display data
        :
        Some display controller specific code
        :

        // Reset repaint parameters
        GuiLib_DisplayRepaint[LineNo].ByteEnd = -1;
        }
    }
}

// Finished drawing
ControllerSelect(0);

// Free GUI resources
GuiDisplay_Unlock();

```



```
}
```

The display data writing is divided into two parts, one for each controller, and the controller is selected by a `ControllerSelect(char index)` function, which sets some ports to enable and disable the controllers as required.

Light and contrast control

These topics are beyond easyGUI's control, but it is logical to put the routines for light and contrast regulation (if applicable) into the `GuiDisplay` unit.

4 - Compiling the project

Next item is to verify that your project can compile without errors, and preferable, without warnings.

Make sure to include the `GuiLib`, `GuiDisplay`, `GuiFont`, `GuiVar` and `GuiStruct` units into your project.

Compile and link, and make sure that everything works as intended.

Although we have tested easyGUI on a large number of compilers, and with many different types of displays, it is impossible to test every possible product and combination on the market. However, it is our experience that if the compiler conforms to ANSI X3.159-1989 Standard C, the compiling and linking should proceed without problems.

In case of problems try setting code optimization to a minimum, and then incrementing optimization one step at a time. It is especially important to make sure that code optimization is *not* applied to the `GuiFont`, `GuiVar` and `GuiStruct` units, because these units only contain constant declarations which cannot be optimized. Some compilers misunderstand this, and apply various kinds of optimization to the constant declarations (with the best intentions!), making them unusable.

C++ is not used in the easyGUI library.

5 - easyGUI interfacing

Interfacing easyGUI to your own target code is a fairly simple exercise. Two important function calls must be made for easyGUI to work at all:

- `GuiLib_Init` which initializes easyGUI.
- `GuiLib_Refresh` which executes easyGUI display writing.

Furthermore, if your operating system uses pre-emptive execution, i.e. it interrupts tasks at random instances, and transfers control to other tasks, some functions in easyGUI

must be protected from this, especially display writing. This is accomplished by writing code for the two protection functions:

- `GuiDisplay_Lock` which must prevent the OS from switching tasks.
- `GuiDisplay_Unlock` which opens up normal task execution again.

If your operating system does not use pre-emptive execution, i.e. if you must specifically release control in one tasks in order for other tasks to be serviced, or if you don't employ an operating system at all, you can just leave the two protection functions empty. Failing to write proper code for the protection functions will result in a system with unpredictable behavior.

GuiLib_Init

This function must be called once, during system start up. Normally this call is done after low level system initialization, but the sequence of events depends on the nature of your system. `GuiLib_Init` performs the following actions:

- Initializes the display by calling `GuiDisplay_Init`.
- Reset display clipping to full screen.
- Resets display drawing.
- Clears the display.
- Sets various easyGUI variables for normal display writing.
- Selects language zero (reference language defined in easyGUI).

GuiLib_Refresh

The display is updated by the `GuiDisplay` unit regularly (function `GuiDisplay_Refresh`), based on markers that indicate which parts of the display needs updating. However, many other tasks are performed by easyGUI, when refreshing the system:

- Auto redraw item updating.
- Cursor field updating.
- Scroll box updating.
- Blink box updating.
- Display updating.

You should therefore only call `GuiLib_Refresh`, *not* `GuiDisplay_Refresh`.

GuiLib_ShowScreen

The final basic easyGUI function is `GuiLib_ShowScreen`, which displays a specific structure, just like it is displayed in the easyGUI editor. The normal syntax is:

```
GuiLib_ShowScreen(GuiStruct_???_?,
                  GuiLib_NO_CURSOR,
                  GuiLib_RESET_AUTO_REDRAW);
```

The first function argument ID's the structure to show (the ID's are defined in the `GuiStruct.h` file). The two other arguments specify that no cursor shall be shown, and that eventual auto redraw items from previously shown structures should be discarded. This setup is the most usual. Structures containing cursor fields will use the second argument to specify which cursor field to initially show, instead of stating `GuiLib_NO_CURSOR`. The last argument can be used if several structures are shown in succession, where auto redraw items from the first structure should be maintained even after displaying the second structure. This is done by specifying `GuiLib_NO_RESET_AUTO_REDRAW` instead of `GuiLib_RESET_AUTO_REDRAW`.

For further information on the various easyGUI function calls, see the reference section.

TESTING THE SYSTEM

When the display controller specific code has been written, and the target system can be turned on without emitting smoke, it is time to verify that the display functions as intended. And it most certainly doesn't at the first try. So, how to test the display setup in the most efficient way? Do not start with a fine complex easyGUI structure, containing lots of text and icons, because probably nothing at all will be shown... Instead, use the following guidelines as an inspiration. They are not universally applicable, because of the diversity of target systems, but the guidelines are a result of considerable experience in the field, and could therefore spare your of some of your precious development time.

The guidelines are intended to be used in the order stated:

- 1 Establishing some kind of connection.
- 2 Turning on a single pixel.
- 3 Showing the test pattern.
- 4 Showing an easyGUI structure.

1 - Establishing some kind of connection

First item on the agenda is to verify that the contrast regulation is working properly. This of course only applies to LCD displays, but these are by far the most used in industry today. A common mistake is to save the contrast regulation for later, and concentrate on

getting *something* on the display. This could be a big mistake, because a contrast setting at the lower limit will mask all attempts to write on the display.

The contrast setting should be changed from minimum to maximum, and the display should correspondingly change from totally blank to rather dark. If not, something is wrong with the contrast regulating electronics (or controlling code).

When seemingly working ok, set the contrast to a middle value, and proceed.

2 - Turning on a single pixel

Write some code to turn on the top left pixel:

```
GuiLib_Dot(0, 0, GuiConst_PIXEL_ON);
```

This should turn on the upper left pixel. If nothing happens, try the opposite:

```
GuiLib_Dot(0, 0, GuiConst_PIXEL_OFF);
```

The last statement should normally not produce anything, but sometimes the meaning of black and white pixels are mixed up, because some display controllers use a zero bit as black, while others use a one bit (we are talking monochrome displays here).

Next, make sure that the relevant `GuiLib` functions (`GuiLib_Init()` and `GuiLib_Refresh()`) are actually called at all. A little embarrassing if they are not, and if this is overlooked before proceeding with the next attempts, because then guaranteed nothing will show up on the display.

Still no reaction: Time to recheck all settings, both easyGUI settings, and the display controller specific code written previously.

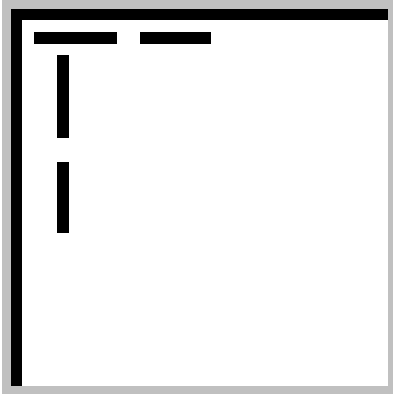
If this doesn't help either, it is prudent to measure all signals to the display with an oscilloscope or similar equipment, and make sure that they look satisfactory, regarding levels, flanks, and timings.

Last resort is to dig into the display controller data sheet, and double check if anything was overlooked, misunderstood, or misread.

Bitter experience has shown that *very carefully* rechecking the above issues one by one normally ends with the proper result, albeit some times not after a certain amount of frustration... This, however, has nothing to do with easyGUI, but is the normal process necessary to get things working.

3 - Showing the test pattern

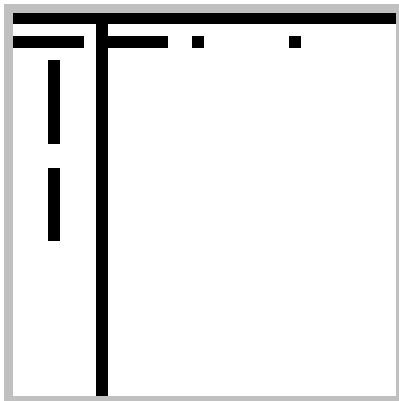
As an extra help and control that things are set up correctly a test pattern can be generated in easyGUI. Just call the `GuiLib_TestPattern` function, and the following pattern should be shown in the top left corner of the display:



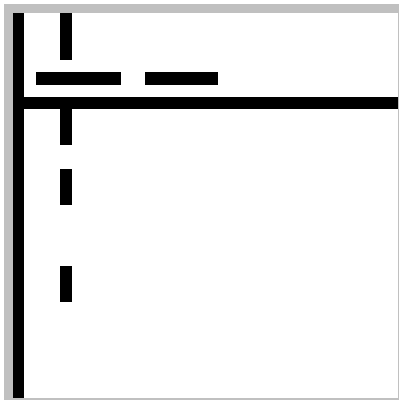
The top and left long lines are 32 pixels in length. The short lines nearest the corner are 7 pixels long, while the lines farthest from the corner are 6 pixels long. There is one pixel of white space between the long and short horizontal lines, while there are 3 pixels between the long and short vertical lines.

If nothing is shown at all go back to the previous step.

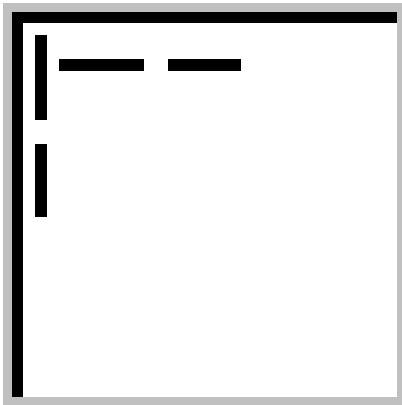
If the pattern does *not* look like shown above (look very carefully!) there are several possibilities:



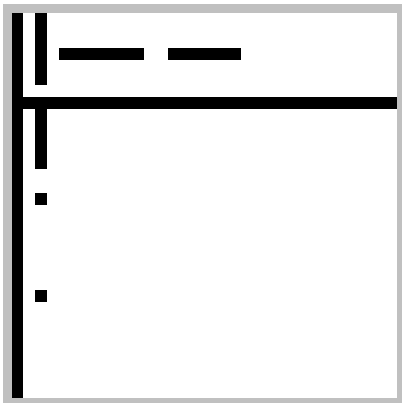
The display uses horizontal display bytes, but they are reversed, i.e. bit zero is at left and should be at right, or vice versa. Change the bit orientation layout in Project parameters, generate C code, compile, and try again.



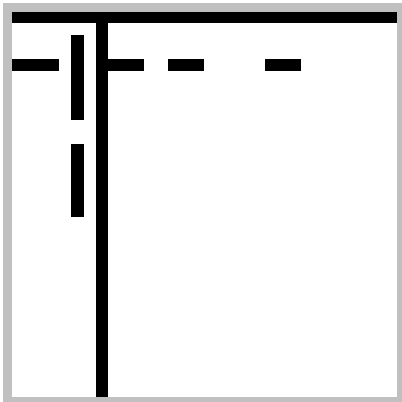
The display uses vertical display bytes, but they are reversed, i.e. bit zero is at top and should be at bottom, or vice versa. Change the bit orientation layout in Project parameters, generate C code, compile, and try again.



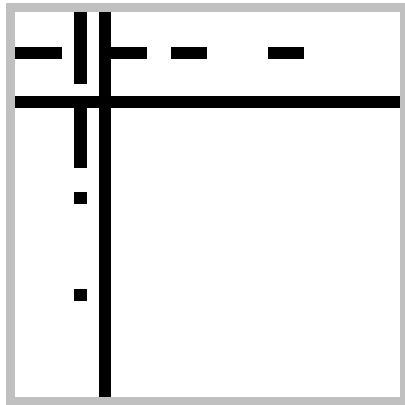
The display uses vertical display bytes, but horizontal bytes have been selected, or vice versa. Change the byte orientation layout in Project parameters, generate C code, compile, and try again.



The display uses vertical display bytes, but horizontal bytes have been selected, or vice versa. Furthermore, bit zero is at top and should be at bottom, or vice versa. Change the byte orientation layout in Project parameters, generate C code, compile, and try again.



Same as previous pattern.



Same as previous pattern.

4 - Showing an easyGUI structure

The first easyGUI structure to show should be simple - preferably just a single text. Normally this last test step doesn't pose problems. The difficult part is to get the display communication and addressing correct, and the previous items should have taken care of this by now.

If problems arise, they are almost always connected with variable type declarations and pointer sizes, as set up in Project parameters. Most important is to make sure that the memory model selected for the processor corresponds to the pointer size set in easyGUI. Errors on this subject almost certainly results in a non-operating system.

Other potential areas of trouble are stack sizes, and memory wrap-around, if the compiler only supports e.g. 64KB segments. Some linkers don't even warn on memory wrap-around.

A final source of errors are the two task switching protection functions `GuiDisplay_Lock` and `GuiDisplay_Unlock`. Errors arising from improper code in these functions show up as periodical problems with garbled display contents.

15 HOW TO UTILIZE easyGUI - A TUTORIAL

A system with the complexity of easyGUI takes some time getting used to. The tutorial in this chapter walks you through the various aspects of structure editing, from the very simple to the more complex tasks.

EFFICIENT LEARNING

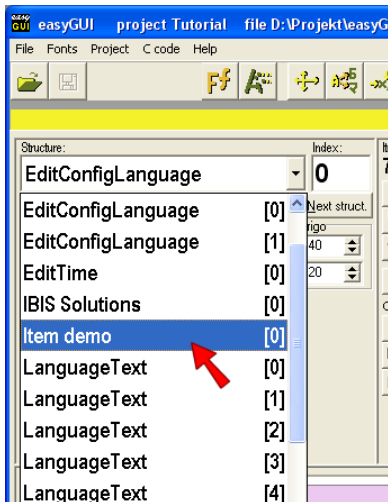
The tutorial is best read while running easyGUI. To achieve the quickest and most efficient learning easyGUI should be started, and the project file Tutorial.gui loaded. This project file is installed as part of the easyGUI system, and is found in the sub folder Manual under the folder containing easyGUI. If a standard install was performed it is found under C:\Program files\easyGUI\Manual.

ITEM TYPES

Let's start with reviewing the different types of items. Enter structure editing:



- and select the Item demo [0] structure using the drop down arrow in the top left combobox:



The structure can now be viewed in the display panel:



Observe that the bitmap with girl and bird is by external reference, and the corresponding Birdy nam nam.bmp file must be present along with the Tutorial.gui file, but in a standard installation this should be the case.

This structure is used in the following sections, and is just a collection of various items, made for demonstration purposes. It demonstrates the following item types:

- **Pixel** Draw a single pixel.
- **Line** Draws a line. Three different types of lines are shown - horizontal, vertical, and angled.
- **Framed rectangle** Draws a rectangle frame - two are shown, one with a single pixel in border thickness, and one with two pixels.
- **Filled rectangle** Draws a filled box.
- **Text** Draws a text - several different text fonts are shown.
- **Icon** Draws items just like texts - several different icons are shown.
- **Formatter** A non-visible item that instructs following variables on how to format them. A formatter is valid until another formatter is stated, so one formatter can be common to a series of variables.
- **Variable** Draws a string or numerical value, using the format set by the last formatter.

Besides these item types there are four more, which are not shown:

- **Paragraph** A text box with associated parameters, where text is automatically divided into lines at word spaces and hyphen characters.
- **Structure call** Call another structure, which is then drawn, before continuing with the current structure. Structures can be nested inside structures as deep as stack space permits on the target system.

- **Indexed structure call** Calls another structure with index number depending on a variable.
- **Clipping rectangle** Limit all later drawing commands to a rectangle. This item can also terminate clipping.

They will be demonstrated at a later state.

VIEWING THE STRUCTURE

To the left of the display are a number of settings, controlling how the display is viewed in easyGUI. Let's investigate a few of them:

- At the top is a **zoom setting** enabling enlargement of the display.
- Next is a **Show display border** setting that determines if the active drawing area of the display shall be indicated. The active area is all addressable pixels, while the inactive area is the border around the edges of the display. The size and color of this border area can be set in the Parameters window. The border area has no effect on anything drawn by easyGUI, it is shown purely to make the display representation look more real, and to show if items drawn on the display collides with the border in an unpleasant way. The Item demo [0] shows the difference clearly:

Show display border on:



Show display border off:



- Last setting this time is **Show undrawn area**, which indicates with a special color the areas of the display not touched by the current structure. This is handy when

checking where backgrounds are drawn. This setting should in most instances be left on. Again, the Item demo [0] shows the difference:

Show undrawn area on:



Show undrawn area off:



Observe that the white rectangle in the fourth line is only visible when Show undrawn area is on, i.e. it is then possible to view where this structure *actually* draws something.

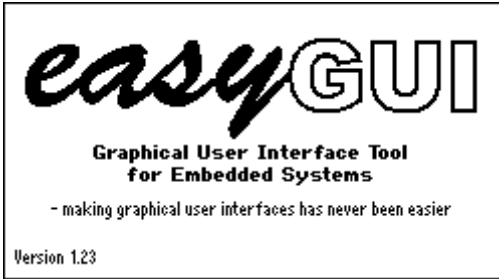
The other settings will be explained later, when a relevant situation arises.

Another handy feature is the crosshair, which is shown when the mouse enters the display area. Besides the crosshair itself the position in pixels is shown. The coordinate system has (0,0) at the upper left corner.

SPLASH STRUCTURE

We won't go through the details of how the Item demo [0] structure is build, but instead move on to a more practical example - a splash screen, or welcome screen.

Select the Screen Splash [0] structure:



This structure shows a logo, some texts, and a version number, which is dynamic, i.e. its value is controlled directly from the embedded code, thereby avoiding the need to edit the structure each time the version number changes.

Structure details

Because this is the first "real" structure we will dissect it in details. It consists of eight items:

Screen Splash [0]		
1	Filled rectangle	Clears the screen
2	Text	Actually an icon, showing the easyGUI logo
3	Text	"Graphical User Interface Tool"
4	Text	"for Embedded Systems"
5	Text	"- making graphical user interfaces has never been easier"
6	Text	"Version"
7	Formatter	Determines the format for the next item
8	Variable	Version number for the application

Clearing the screen

The first item (filled rectangle) is used to clear the screen, so it simply draws a white block with the same dimensions as the display (240x128 pixels in this case).

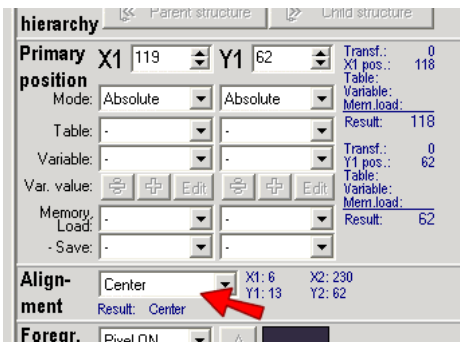
Finding this and that item

Next is the logo, item 2. Click on item 2 in the item list, so its properties are shown in the rightmost pane.

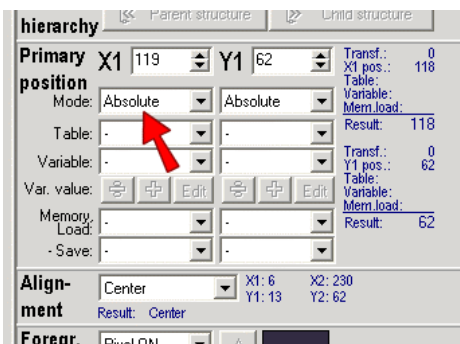
Oh - what if we can't remember what number the item has? Don't just click on all the items until you happen to hit the right one, click instead on the display area, hitting the logo. This item will then be selected. Another handy thing is the other way around - hold the left mouse button depressed while clicking on one of the items in the item list - the corresponding item is then indicated by a flashing red rectangle around it.

Drawing a logo

Logos are shown just like normal text, but the font is a little special. We want to place the logo at a fixed position, so the coordinates are absolute. The logo should always be placed centrally in the horizontal direction, so to make things a little simpler the alignment is set to Center,



- and the X coordinate is set to 119 (midpoint).



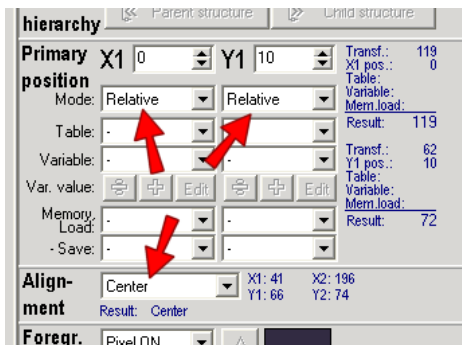
This way the logo can change size, but will still be placed centrally.

The logo is represented as the A character in the font Icon5, so Icon5 is selected as font, and A as text. Press the **CHARACTER SET** button just below the text field to view the icons in font Icon5. There is only one!

More than one character can of course be entered in the text field, but for logos it is almost never practical to enter more than one character.

A centered, relative text

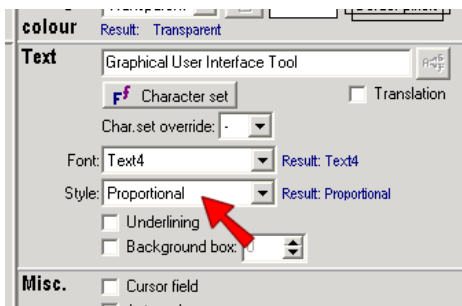
Items 3, 4 and 5 are simple texts. They are placed centered horizontally, just like the logo, so it would seem natural to use the same technique, i.e. absolute X coordinate 63, and centered alignment. But here another very useful feature of coordinate technique can be demonstrated: Relative coordinates. Both X and Y coordinates are selected as relative for the three text items, with X set to zero (keeping all items centered), and Y set to various values to ensure a nice separation between texts.



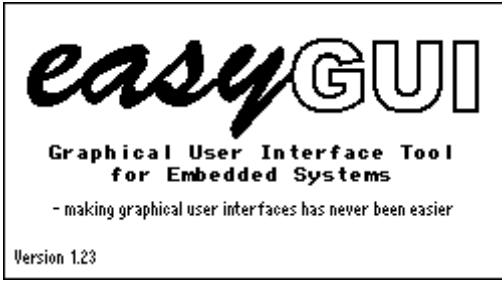
The effect of this is that the icon now determines X and Y coordinates for itself *and* the three texts. This can be demonstrated by selecting item 2 (the icon) and experiment with changing the coordinates. Observe that the four items now move as a united entity.

PS - nice texts

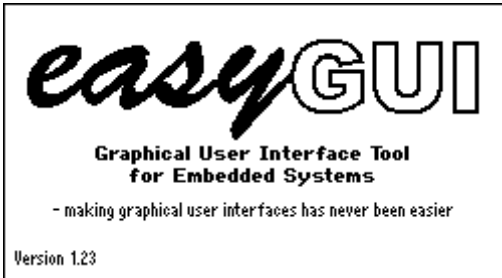
One feature of the texts which is maybe not immediately apparent is its style. The texts are written in proportional writing, just like the text you are reading right now. That looks much nicer than text written with fixed spacing. Select item 3 (the "Graphical User Interface Tool" text), and try changing the Style setting:



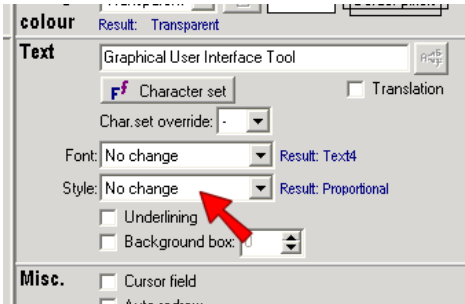
- to fixed spacing. The result is rather terrible:



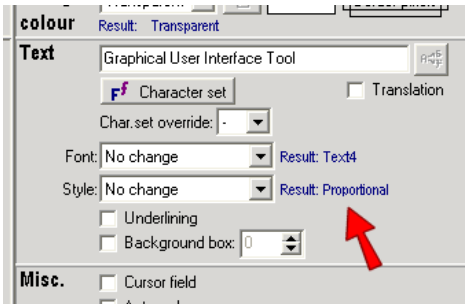
But look closer... Both texts ("Graphical User Interface Tool" and "for Embedded Systems") changed style, resetting to Proportional will bring both texts back to something a little more pretty:



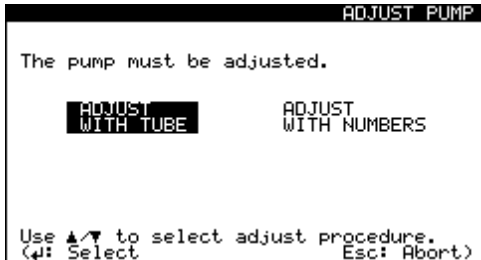
The reason for this can be found by inspecting item 4 (the "for Embedded Systems" text). Look at the Style setting for this item:



It is set to "No change". This is a common feature of many settings in the property panel. Selecting "No change" means that the settings from the previous item is maintained. And what *is* this setting then? That is revealed by the small blue info texts right next to most of the properties:

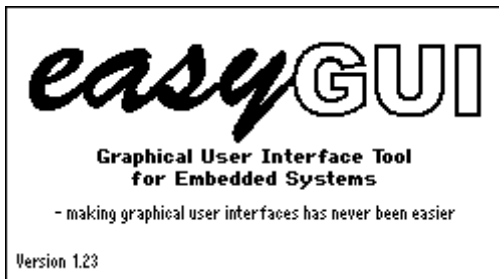


Proportional writing is one of the main advantages of using easyGUI - it assures a much more modern and professional looking user interface than can be achieved with traditional character based display modes, where all characters are placed at fixed coordinates (lines and positions):



Big texts - small texts

Our splash example uses two different text fonts: A fat one for the "Graphical User Interface ... " text, and a small compressed one for the "- making graphical ... ":



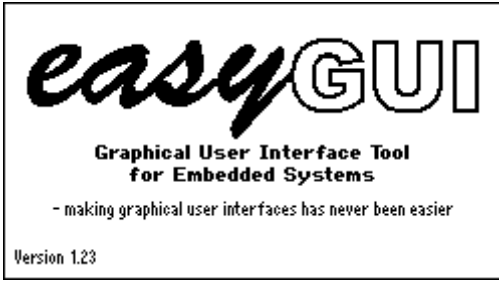
Fonts can be mixed freely in each structure, but the memory requirements on the target system will of course rise if many different fonts are employed.

Don't use too many different fonts, because that will only result in a messy user interface. It is better to select a specific font for headlines, one for normal text, and so on.

Another good reason not to use overly many fonts is ROM memory usage - unless your resources are unlimited...

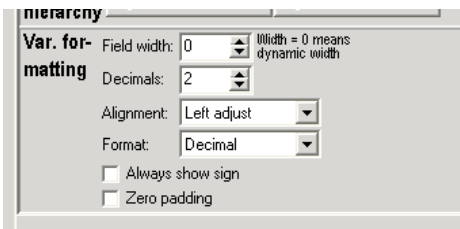
Showing variables

The version number in the lower left corner:



- could of course just be written in plain text, but then it would have to be edited each time the source code receives a new version number. Better to just show the version number from the source code. In the example the version number is a simple 16 bit constant, set to 123. This number should be shown as "1.23", i.e. major and minor version number. Many other schemes can of course be employed, but this example shows a couple of things concerning formatting and display of variables.

Item 7 is a formatter. The only properties for this type of item are:



The properties are set to:

- Field width** Zero, to indicate variable field width, i.e. the field width is just sufficient to contain the number in the selected style.
- Decimals** Two - we want to show the value 123 as "1.23".
- Alignment** Left adjusting. Don't confuse this alignment with the normal alignment for texts, boxes, etc, this alignment determines how the digits/characters are placed in the field width - but with the field width set to zero (dynamic width) this setting has no effect.
- Format** Decimal. Other possibilities are hexadecimal, exponential, and time (HH:MM).
- Always show sign** Off. Not relevant here.
- Zero padding** Off. With the field width set to zero (dynamic width) this setting has no effect.

All this results in the numerical value 123 being translated to "1.23".

If no formatter preceded the variable it would be formatted with default settings.



CONFIG STRUCTURE

Our next example is the Screen Config [0] structure, which shows a configuration screen containing three parameters:

CONFIGURATION	
Language:	English
Force Reduction:	OFF
Display contrast:	25

The first parameter is a language selection, where the language can be selected between five languages: English, German, French, Spanish, and Japanese.

The second parameter is optional, and shall only be shown in some instances. It is a Force Reduction on/off selection (whatever that is).

The third parameter is a display contrast setting, going from 0-50.

The intention is to let the user navigate the three parameters (or two) by using cursor fields, but more on that later.

Structure details

The Screen Config [0] structure uses nine items:

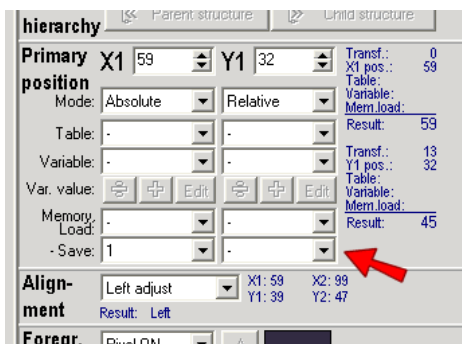
Screen Config [0]		
1	Filled rectangle	Clears the screen
2	Text	Headline
3	Text	Language explanation
4	Indexed structure	Language parameter text
5	Indexed structure	Force reduction line
6	Text	Display contrast explanation
7	Formatter	Formatting of display contrast number
8	Variable	Display contrast number

Like in the Splash structure the first item (filled rectangle) is used to clear the screen, by drawing a white block with the same dimensions as the display.

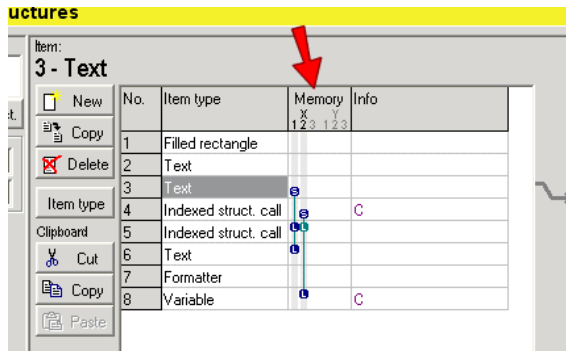
Don't forget the coordinates

The explanatory texts and the parameters in the Screen Config [0] structure are aligned in two columns, a left-aligned for the explanatory texts, and a right-aligned for the parameters. It would be simple to just enter all X coordinates as absolute values, and tweak the hole thing until everything is nicely lined up. Fair enough if you never plan to alter anything, but real life is seldom that simple, so it would be nice if the leftmost and rightmost items were linked, so that moving the top item in a column horizontally moved the other two with it. Then use relative coordinates, what's the problem? Well, the problem is that we can't maintain two sets of relative coordinates, one for the left column, and one for the right column, when the items for the two columns are interspersed.

The solution (of course there is a solution!) is to utilize the coordinate memory system. There are three coordinate registers for each of the X and Y coordinates, which are maintained across structures. At any time can a coordinate be saved in a coordinate register, or retrieved for use:



This is utilized here, by saving the X coordinate for the left column in X coordinate register 1, and the X coordinate for the right column in X coordinate register 2. The following items retrieves the X coordinates again. To aid in controlling coordinate register usage it can be surveyed at a glance by looking at the item list:



The small indicators show when a coordinate value is saved and retrieved.

Using an indexed structure

Items 3 and 4 constitute the language line. Item 3 is a simple explanatory text ("Language:"), while item 4 is more complex. It introduces the concept of indexed structures, which is without doubt the most important aspect of easyGUI display design. The task is to display a text based on a language setting. In this example we want to display the language designations with their native spelling/character set:

- English: "English"
- German: "Deutsch"
- French: "Français"
- Spanish: "Español"
- Japanese: "ニホンゴ"

First problem is to let easyGUI display a text based on a variable selection, second problem is the Japanese Katakana characters. We'll take them one by one.

It should be fairly clear that just showing a text variable is a solution, but a rather restricted solution, because it forces us to do string manipulations on the target system, and it is far easier to do things in easyGUI, in the comfort of our PC environment. And here the concept of indexed structures comes in handy. A variable has been declared, called DiConfigLanguage. It is a numerical variable (8 bit unsigned, but that is not essential here), and the values 0-4 determines the language selection, with zero indicating English. Look at the structure, with item 4 selected:



The two important properties in this discussion are Structure call and Variable. Structure call is set to LanguageText, and variable is set to DiConfigLanguage. This means that when easyGUI draws item 4, it first reads the variable value (Zero right now, look at the small blue text below the variable box), and then calls structure LanguageText with the index corresponding to the variable value. So the result here is that easyGUI displays structure LanguageText [0], *if it exists* (it does!). If the structure does not exist easyGUI merely goes on to the next item without drawing anything.

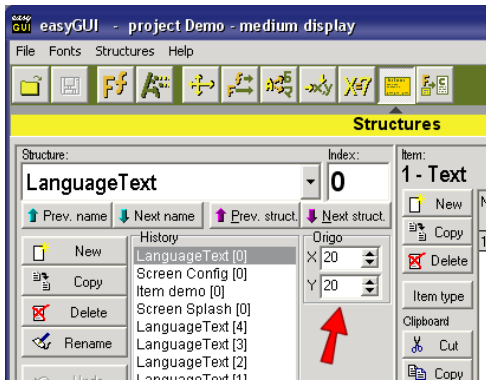
And what does LanguageText [0] contain? Easy to see, just press the **JUMP TO STRUCTURE** button. Pressing the **CHILD STRUCTURE** button (at the top of the properties panel) does the same. Now easyGUI displays the LanguageText [0] structure, which is very simple, just a single text item. Note that the X and Y coordinates are merely set to relative and (0,0). This means that the position of the text is solely determined by the calling structure, which sets it to (176,45). The attentive reader has perhaps now noted that the text is *not* shown at (0,0), but rather some distance down and to the right. If it was displayed at (0,0) it would look like -



- because Y=0 refers to the base line of the text, placing almost all of the text outside the display area. But instead it is shown as:



Much more readable, but how? Well, easyGUI checks if the first item in a structure has relative coordinates. If so (as is the case here both for X and Y) it sets the starting coordinate value to the value defined in the Origo box, located in the top left panel:



The values for this structure are $(X,Y)=(20,20)$, which is a nice value, because it displays the item properly for view, but doesn't use up more display area than necessary, a bonus if the text is long. This coordinate value $(20,20)$ is only used by the easyGUI PC program, it is not used by the target system. If the target system is forced to show a structure starting with relative coordinates (not a sensible thing to do) it uses $(0,0)$ as the starting point. The structure is *not* shown at the $(176,45)$ position specified by the calling structure (Screen Config [0]), because easyGUI cannot know exactly which structure is calling LanguageText [0]. It could theoretically be any structure in the system.

That was a little off topic, the main reason for looking at the LanguageText [0] structure was to inspect what it contains. To make the dynamic language text work another four structures (LanguageText [1] to LanguageText [4]) has been defined, each similarly containing a single text. Now, jump back to the calling structure (Screen Config [0]) by pressing the **PARENT STRUCTURE** button at the top of the properties panel. We are now back at the calling structure, still with item 4 selected (easyGUI remembers the selected item *individually* for each structure in the system, a big advantage).

Let's try to alter the DiConfigLanguage variable to something else. Press the small \div and $+$ buttons, and watch the variable value change, and more importantly, watch the language text change. Editing the variable value directly can be accomplished by pressing the **EDIT** button, which shows a little editing window. If the value is incremented past four, or below zero, easyGUI ascertains that no structure exists with e.g. the name LanguageText [5]:



At the same time no language text is shown:

CONFIGURATION	
Language:	
Force Reduction:	OFF
Display contrast:	25

In this situation it is clearly an error, but in the next section you will see that this feature can be used to your advantage. Main point is that easyGUI handles the situation gracefully, no error condition is entered.

Utilizing a disappearing indexed structure

The next line in our example structure is the "Force Reduction" line, which was supposed to disappear in some instances (perhaps for differently equipped systems?). The complete line is therefore moved to its own indexed structure, `ConfigForceReduction [1]`. Remark that the index is set to `[1]`, not `[0]`. No `ConfigForceReduction [0]` structure exists, this is perfectly legal.

The variable controlling the index structure is `ForceReductFlag`, and it is right now set to the value one. Try changing it to zero, and watch the indexed structure disappears:

CONFIGURATION	
Language:	English
Display contrast:	25

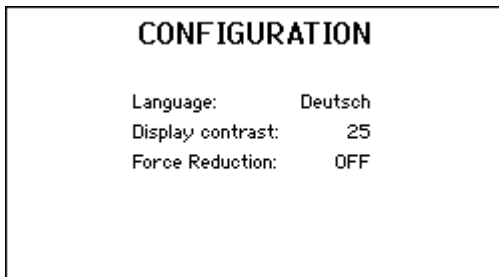
At this point it should be obvious what happens: `ConfigForceReduction [0]` doesn't exist, so nothing is shown. On the target system, all that needs to be done is making sure the `ForceReductFlag` variable has the correct value, *before* showing `Screen Config [0]`. Easy!

As a little bonus the third line, "Display Contrast", moved up on the position of the "Force Reduction" line. This only happens if the coordinates are carefully set, so that the "Force Reduction" line can be removed without upsetting the relative Y coordinates. It should be observed that the Y coordinate is placed *inside* the `ConfigForceReduction [1]` structure, not in the *calling* item. Otherwise, removing the "Force Reduction" line would leave an empty space between the two remaining lines, because the calling item is still there, but by now don't call anything.

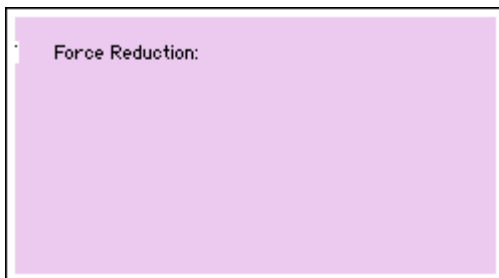
The indexed structure technique can be further refined, to show much more complex screen setups, with many parts of the display being dynamic, depending on the settings of variables. Not just for hiding parts of the display, but also for e.g. displaying *different* types of information in parts of the display. The latter concept will be demonstrated in our next example structure.

An on/off text

Turn the "Force Reduction" line on again, by setting the ForceReductFlag variable to one:



The parameter text is an on/off text, controlled by another variable called ForceReduction. Jump into the ConfigForceReduction [1] structure:



- and you will see that it consists of two items:

- Item 1 Text "Force Reduction".
- Item 2 Indexed structure "On"/"Off" text.

It must be admitted that item 2 is virtually non-visible, this is because it reads the X coordinate from a coordinate register, which is not set up properly when looking directly at the structure.

The second item is yet another indexed structure, controlled by the ForceReduction variable, which can adopt the values zero ("Off") and one ("On"). It calls the structure TxtOnOff [X]. By the way, the "On"/"Off" text is nearly invisible, that's because it uses X register 1 as a coordinate source, and its value is not set correctly inside this structure.

This construction is an example of an indexed structure within an indexed structure:

Screen Config [0]	
1	Filled rectangle
2	Text
3	Text
4	Indexed structure call:
	LanguageText [X]
1	Text
5	Indexed structure call:
	ConfigForceReduction [1]
1	Text
2	Indexed structure call:
	TxtOnOff [X]
1	Text
6	Text
7	Formatter
8	Variable

Backgrounds are important

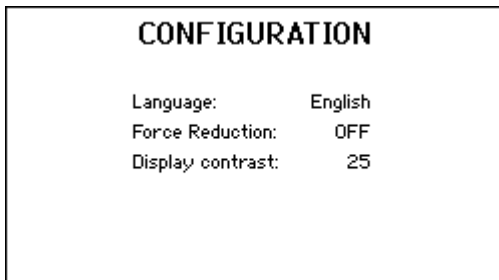
There is another interesting thing about the on/off text. It is supposed to change during editing (sounds reasonable), and that shouldn't involve rewriting the entire config structure, although that would be a solution, but a very inefficient and perhaps slow one (depending on CPU resources in the target).

There are two ways to get items redrawn, without redrawing everything:

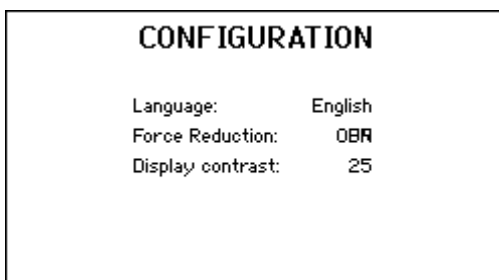
- Cursor fields
- Auto redraw items

The first approach is used here, but that is explained a little later. The second option will also be explained in due time. Suffice it to say that the item *is* redrawn. The interesting thing here is what happens when we redraw an already present item. Coordinates are no problem, easyGUI remembers the coordinates already calculated, and draws the item

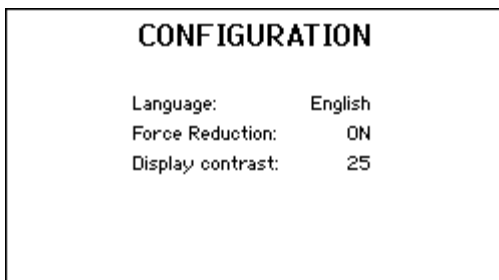
again in the same position. The problem can be the background. If no background is specified (transparent writing) the texts will pile up on each other, so changing Force Reduction from OFF to ON will change the display from:



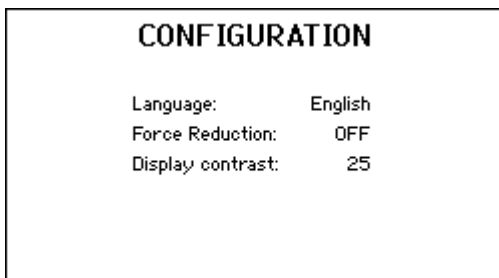
- to:



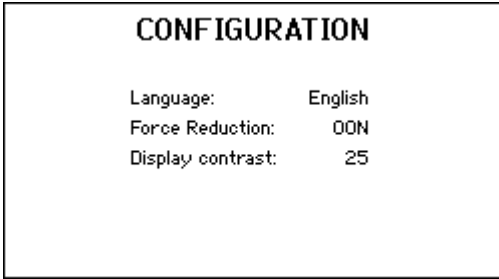
You can't see this in easyGUI, because the display is cleared each time a setting is changed, or another structure is selected, but on the target the above will happen. Not pretty. Something have to be done about the background. Changing the background from Transparent to Pixel Off should solve the problem. Indeed, going from ON back to OFF displays:



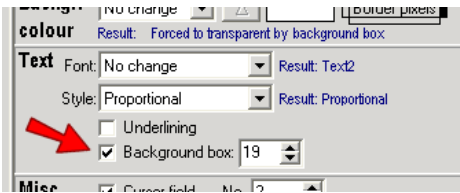
- to:



Very fine! Until you change the parameter to ON again:



My my... What happened now? Invented a new word? No, the "ON" text was displayed correctly, but the first letter of the "OFF" text is still very much visible. So - another solution must be found. And this is a concept called a background box. It is situated in the Text part of the properties panel:

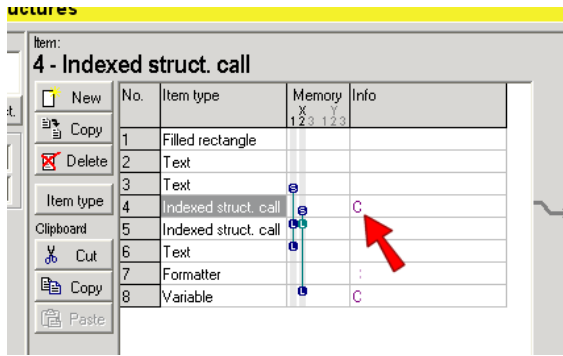


Turning it on for an item will instruct easyGUI to draw a filled box in the background color with a height determined by the item (Text: Font height, Rectangle: Rectangle height), and a width set directly as a property (to the right of the Background box checkbox). In our example structure the background box is enabled for the indexed structure that calls the ON/OFF texts (ConfigForceReduction [1]). The net result is that a background of (in this case) 19 pixels width is drawn before the "ON" or "OFF" text is drawn, sufficiently wide to cover any pre-existing text. As a side product the background color of an item with background box drawing enabled is automatically set to transparent, to avoid drawing the background twice.

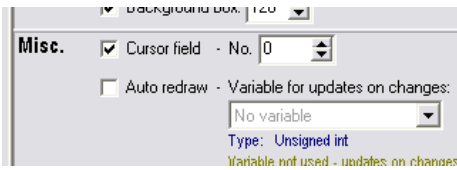
The fine art of cursor fields

We are not quite finished with the trusted and tried Screen Config [0] structure. It is supposed to be navigated by using a cursor to point out the desired parameter, but so far, not a word about cursor fields (except for several promises to do it later), so now is the time!

We want three cursor fields, one for Language, one for Force Reduction (optional), and one for Contrast. In the example they are of course already defined. Select Item 4 (the first indexed structure) in the Screen Config [0] structure. Note the **C** in the item list, rightmost column:

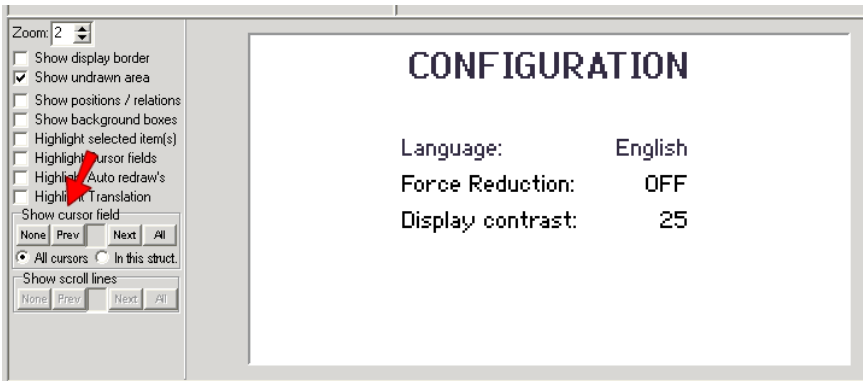


It indicates that a cursor field has been defined for this item. The definition is done in the Misc. part of the property panel (at the bottom):

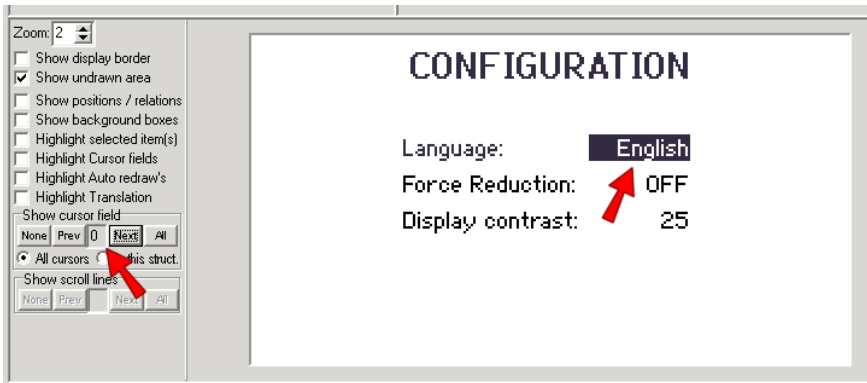


Along with the checkbox for cursor field activation is an edit box where the cursor field index number can be selected. Cursor fields are normally indexed from zero and upwards, but this is not a requirement. Negative field indices however, are not allowed.

In the easyGUI library is a number of routines for handling cursor fields, which can be called from the embedded code, when the user does something that shall change the active cursor field (normally a keyboard event of some kind). easyGUI then takes care of drawing the new active cursor field using inverted colors, and drawing the previous cursor field in normal appearance. This can be tested in easyGUI by using the Show cursor field box at the lower left corner:

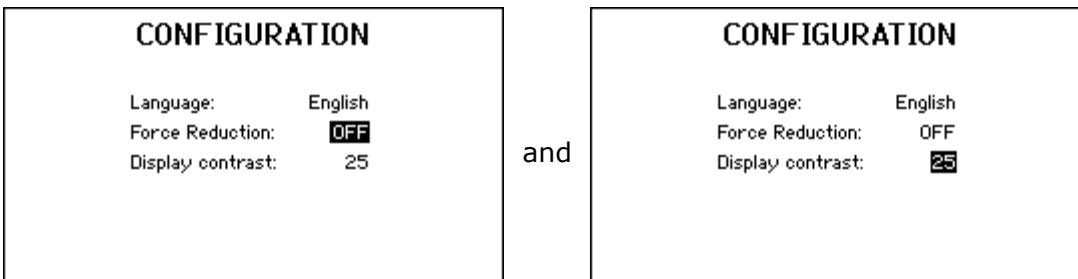


Try clicking the **NEXT** button, and see two things:



- A zero appears in the middle box.
- The "English" text is shown in inverted colors.

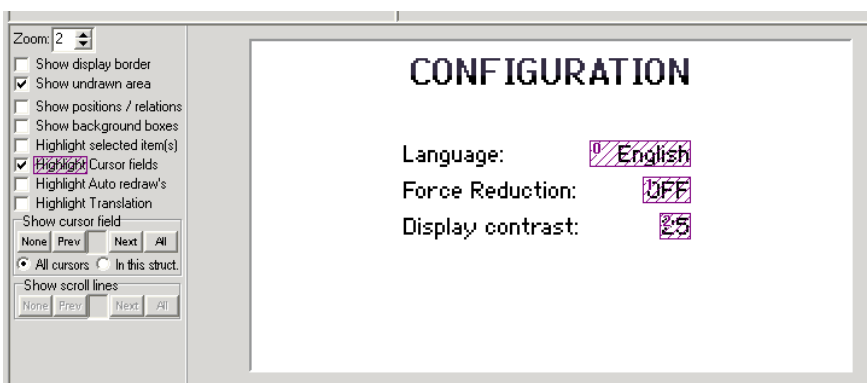
Pressing **NEXT** twice more shows the next cursor fields:



Repeated clicks on **NEXT** just wraps around to the first cursor field. Clicks on **PREV** will of course traverse the cursor fields the other way around. Pressing **NONE** clears cursor field displaying, while clicking on **ALL** shows all cursor fields at once. A specific cursor field index can also be entered directly in the centre box.

Another way to show cursor fields at a glance is to select the Highlight cursor fields checkbox just above the

Show cursor field box:



Purple indicators are drawn around all cursor fields, and their indices are shown. This is a quick way to spot if the correct items are marked as cursor fields, and that their numbering is as desired. easyGUI doesn't check for numbering inconsistencies, like e.g. gaps in the numbering sequence, or duplicated cursor indices.

It is legal to have gaps in the numbering sequence, and this situation is handled correctly by easyGUI. An example is the second cursor field in our test structure (Force Reduction ON/OFF). If the line containing it is *not* shown (ForceReductFlag = 0) the remaining cursor fields are 0 and 2:

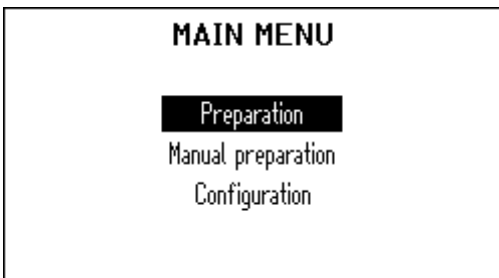


easyGUI still handles selecting the next and previous cursor field correctly, because it *searches* for the next/previous field, instead of just incrementing/decrementing the cursor index. On the target system the two remaining cursor fields still have the same indices, making code writing specific to a certain cursor field easy.

Duplicated cursor indices are not very useful, but maybe a situation can be constructed where it could be advantageous. Anyway, duplicate indices are handled by ignoring all duplicates except the last one.

MAIN MENU STRUCTURE

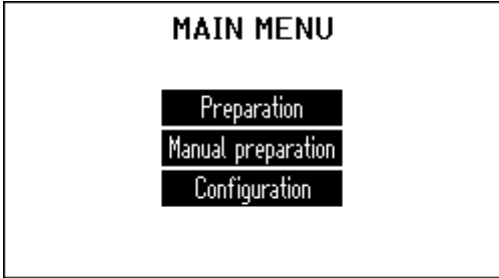
The Screen Main [0] structure is included here to show an ordinary menu page with, in this case, three menu items:



A couple of tricks have been used to enhance the look of the menu items.

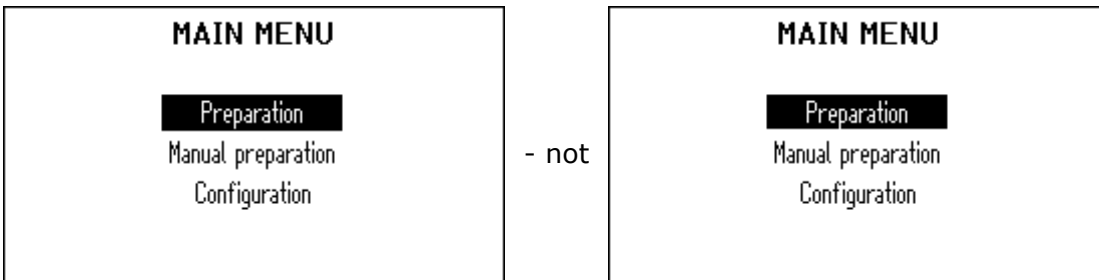
Better looking menu items

Look carefully at the "Preparation" menu item. It is assumed that cursor field zero is selected, i.e. shown inverted, like in the above illustration. The inverted colors stretch beyond the text itself (this was also the case for the "Language" item in the previous structure). The purpose here is to make all three menu items look the same when inverted - best shown by enabling all three of them in easyGUI:

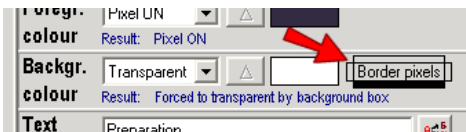


This effect is achieved by defining a background box for all three menu items when the same width (90 pixels in this example).

Another small feature which enhanced the look is much more difficult to spot. An extra line of dark pixels has been added to each menu item:



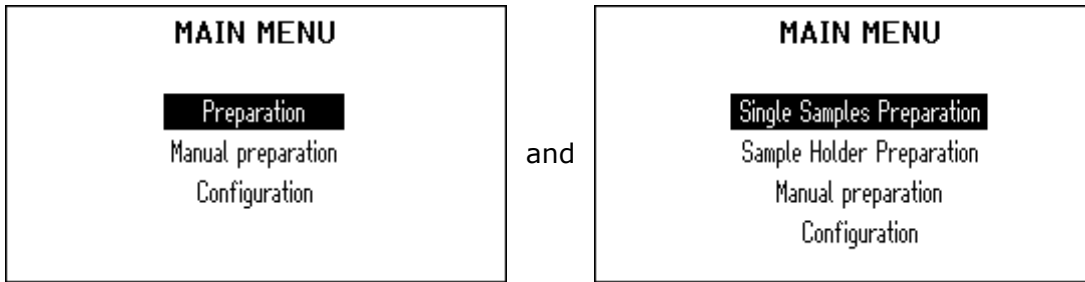
This has the effect that the "p" in "Preparation" doesn't touch the bottom of the dark box. It looks just a little better. One extra pixel line can be added to any of the four borders, by using the Border pixels control at the right side of the Background color panel:



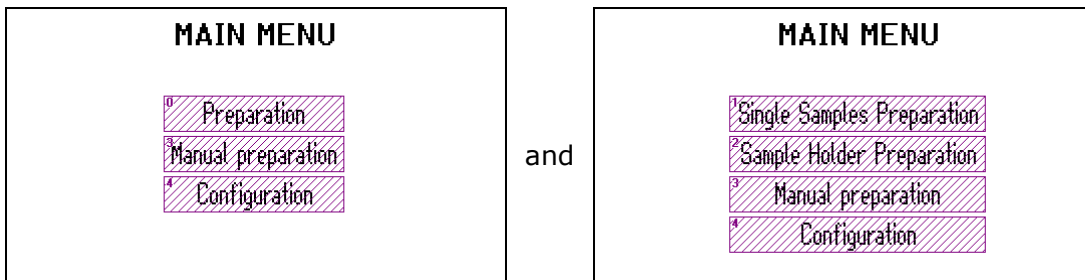
In this example the bottom border has an extra line of pixels added, as indicated by the black box below the "Border pixels" text. Clicking on any of the four boxes around this text will toggle its status.

Playing with cursor indices

In the demo database is two main menus:



They are made in the same style, but don't contain exactly the same menu items. The purpose is to show once more, that cursor fields can be numbered rather freely:



In the left structure is three menu items (cursor fields):

- 0 Preparation
- 3 Manual preparation
- 4 Configuration

- while the right one contains:

- 1 Single samples preparation
- 2 Sample holder preparation
- 3 Manual preparation
- 4 Configuration

One of the two main menus is shown on the target system, depending on some kind of selection (a standard and a deluxe version perhaps?):

```

if (machine_deluxe)
    GuiLib_ShowScreen(GuiStruct_Screen_Main_0,
                      GuiLib_NO_CURSOR,
                      GuiLib_RESET_AUTO_REDRAW);
else
    GuiLib_ShowScreen(GuiStruct_Screen_Main_1,
                      GuiLib_NO_CURSOR,
                      GuiLib_RESET_AUTO_REDRAW);

```


This code selects between the two structures. So, in this instance the index numbers of the two structures ([0] and [1]) are used manually, not for an indexed structure call, but that is of course just as legal.

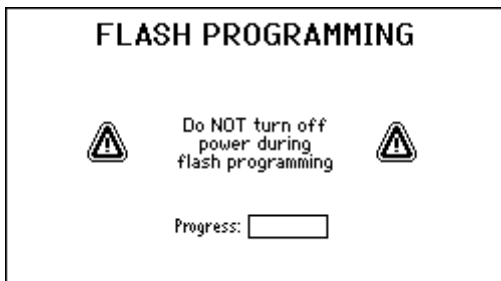
The two structures combined defined the menu items:

- 0 Preparation
- 1 Single samples preparation
- 2 Sample holder preparation
- 3 Manual preparation
- 4 Configuration

Now, because we have used the cursor indices a little intelligent, life has become easier on the target system, where we now only have to assign some kind of action to the five cursor indices, disregarding which of the two structures is active.

FLASH STRUCTURE

The Screen Flash [0] structure is an example of how to combine easyGUI structures and manual graphics. It looks like:

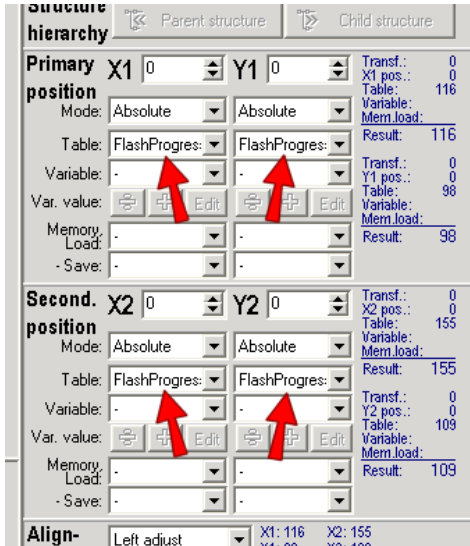


Mixing structures and plain graphics

The intention is to show a progress bar in the white rectangle at the bottom. The rectangle coordinates are defined using the position tables:

EXPLANATION	X	Y	ALIGNMENT
FlashProgress1	116	98	Left adjusted
FlashProgress2	155	109	Left adjusted
Headline	64	6	Centered
Man. prep. left margin	52		Left adjusted
Man. prep. right margin	182		Right adjusted

Here a number of fixed positions have been defined, among others the two positions FlashProgress1 and FlashProgress2. Each of these defines a fixed X and Y coordinate. The first is used in the structure (look at item 8) for the upper left corner of the rectangle (X1,Y1), and the second is used for the lower right corner (X2,Y2):



When the structure has been shown on the target system, the positions can be reused in the code, because they are exported as constants in the `GuiVar.h` file. The graphical primitives in the `GuiLib` unit can then be used for manual drawing, i.e. pixels, lines, boxes, etc.

The advantage of the above procedure is that the position and size of the box can be adjusted in easyGUI by tweaking the values in the position tables, without touching anything on the target system, and it will still work.

LET'S SCROLL

easyGUI supports scrolling information. A necessity if handling lists of data, because the relatively small displays used in most embedded applications doesn't allow much data to be visible at once.

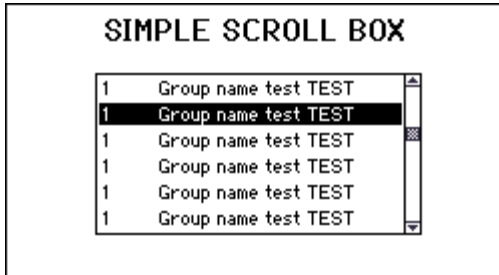
The scroll system uses three components to handle scroll boxes:

- Scroll box
- Scroll bar
- Scroll line

Scroll box is the primary object, which defines the size of the scroll box. To it can be added a scroll bar, which is the vertical box at the side of the scroll box. The scroll bar is not mandatory, it can be omitted if desired. A scroll line must be placed inside the scroll box, and it will then by easyGUI be repeated the number of times space in the scroll box

permits. Because the scroll line is only a single item, and a scroll line in most circumstances will need a more complex composition (data arranged in columns, units, etc) it is most usual to use a structure call as the scroll line item. This called structure can then be made as complex as the situation warrants. Eventually the scroll line structure can be a set of structures, controlled by one or more variables.

A simple scroll box could look like:



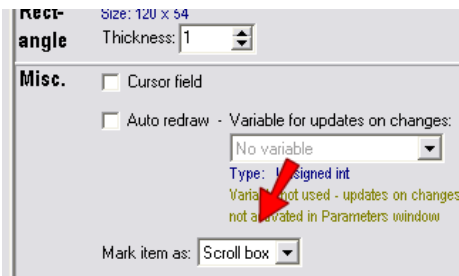
It is the Screen ScrollBoxSimple [0] structure. There are two columns in the scroll box, one with numbers, and one with text strings.

It can be observed that the (in this case) six scroll lines shown contain exactly the same data. This is not an error, but will always be the case when looking at the scroll box in easyGUI. On the target system the various scroll lines will be populated with real data, and hopefully look different (otherwise, why use a scroll box?...). The reason is that easyGUI merely repeats the defined scroll line the needed number of times, without querying after data.

The scroll line is in a separate structure, called ScrollBoxLineSimple [0]. The two structures consist of the following items:

Screen ScrollBoxSimple [0]	
1	Filled rectangle
2	Text
3	Framed rectangle (Scroll box)
4	Framed rectangle (Scroll bar)
5	Structure call: (Scroll line)
ScrollBoxLineSimple [0]	
1	Formatter
2	Variable (Number)
3	Variable (Text string)

The three special markings (scroll box, scroll bar, scroll line) are set in the Misc. box at the bottom of the properties panel. Item 3 in Screen ScrollBoxSimple [0] is the scroll box:



On the target system easyGUI must be told what to display in each individual scroll line. This is accomplished by defining a query function, which easyGUI can call each time it needs to draw a scroll line. The function can be named freely:

```
static void DemoScrollLine(GuiConst_INT16S LineIndex)
{
    GuiVar_GroupNo = LineIndex + 1;
    strcpy(GuiVar_GroupName, SomeTexts[LineIndex]);
}
```

- or, in the case of Unicode mode:

```
static void DemoScrollLine(GuiConst_INT16S LineIndex)
{
    GuiVar_GroupNo = LineIndex + 1;
    GuiLib_UnicodeStrCpy(GuiVar_GroupName, SomeTexts[LineIndex]);
}
```

- where it is assumed that the `SomeTexts[]` array is containing Unicode strings.

What's going on here? The function *must* have the parameters as shown above. It has a single 16bit signed parameter defining the scroll line index, with zero as the topmost line (the topmost *absolute* line, not the topmost *visible* line). The function in the example above first sets the line number in the scroll box as the index number + 1 (item 2 in the `ScrollBoxLineSimple[0]` structure), resulting in the scroll box showing 1, 2, 3, ... Next the text (item 3) is set to something meaningful based on the scroll line index.

Now the scroll box can be set up and shown:

```
GuiLib_SetScrollPars(&DemoScrollLine, 27, 0);

GuiLib_ShowScreen(GuiStruct_Screen_ScrollBoxSimple_0,
                  GuiLib_NO_CURSOR,
                  GuiLib_RESET_AUTO_REDRAW);
```

The first statement connects the easyGUI scroll box functions to our just defined query function `DemoScrollLine`. Besides the function address the total number of lines in the scroll box is stated (27 in this case), and the line being selected initially (zero, the topmost in this case).

The second statement shows the structure, just as usual. The scroll box is now displayed, with the topmost line selected (inverted).

Now the only thing missing is scroll box navigation. One line up:

```
GuiLib_Scroll_Up();
```

- and one line down:

```
GuiLib_Scroll_Down();
```

The two functions returns 1 if scrolling was possible, otherwise 0 if the list was already at the top or bottom. This return value can be used to e.g. signal when the scroll list ends have been reached, and to initiate actions based on scroll line changes.

The currently selected scroll line can be checked by reading the variable `GuiLib_ScrollActiveLine`. The variable `GuiLib_ScrollTopLine` tells which line is currently displayed at the top of the scroll box. Finally, the variable `GuiLib_ScrollVisibleLines` can be useful, it tells how many lines are visible in the scroll box (is calculated when displaying the scroll box structure).

More advanced calculations on scroll box contents can be done by using the `GuiLib_ScrollLineOffsetY()` function. It returns the number of pixels vertically between the currently active scroll line and the topmost visible scroll line.

A special case is a scroll box without a bar, i.e. only a window with scrolling lines in it. It is enabled by selecting -1 as the initial line number in the `GuiLib_SetScrollPars` function call (last parameter). This has two consequences:

- No inverted bar is shown.

- The scroll box contents scrolls immediately when issuing `GuiLib_Scroll_Up()` and `GuiLib_Scroll_Down()` commands (if scrolling is possible), because the bar no longer needs to transverse to the top or bottom of the scroll box before scrolling commences.

The `GuiLib_ScrollActiveLine` variable and the `GuiLib_ScrollLineOffsetY()` function has no meaning in this special case.

16 easyGUI FUNCTION REFERENCE

There are six modules in the easyGUI system:

- `GuiConst` definitions (h file only).
- `GuiLib` library.
- `GuiDisplay` display control unit.
- `GuiFont` easyGUI font definitions.
- `GuiVar` easyGUI user defined variable definitions.
- `GuiStruct` easyGUI structure definitions.

Most target system functions are found in the `GuiLib` unit, the rest are in the `GuiDisplay` unit.

The `GuiLib` unit uses a number of include files:

- `GuiGraph1H.c` library for 1 bpp monochrome displays with horizontal display bytes.
- `GuiGraph1V.c` library for 1 bpp monochrome displays with vertical display bytes.
- `GuiGraph2H.c` library for 2 bpp grayscale displays with horizontal display bytes.
- `GuiGraph2V.c` library for 2 bpp grayscale displays with vertical display bytes.
- `GuiGraph2H2P.c` library for 2 bpp grayscale displays with horizontal display bytes, and two bit planes.
- `GuiGraph2V2P.c` library for 2 bpp grayscale displays with vertical display bytes, and two bit planes.
- `GuiGraph4H.c` library for 4 bpp grayscale/color displays with horizontal display bytes.
- `GuiGraph4V.c` library for 4 bpp grayscale/color displays with vertical display bytes.
- `GuiGraph5.c` library for 5 bpp grayscale displays.
- `GuiGraph8.c` library for 8 bpp grayscale/color displays.
- `GuiGraph16.c` library for 12/15/16 bpp color displays.

- GuiGraph24.c library for 18/24 bpp color displays.

Monochrome version: Only the GuiGraph1H.c and GuiGraph1V.c include files are part of the installation.

GUICONST UNIT

This unit is only an h file, containing definitions set up in easyGUI in the Parameters window. The constants unit should not be edited directly, instead the values are set from inside easyGUI.

The following constants are defined in easyGUI (in alphabetical order):

Constants

GuiConst_AUTOREDRAW_FIELDS_MAX

Purpose: Max. number of concurrent auto redraw items. A large number will consume more RAM space on the target system. Each auto redraw item consumes approximately 60 bytes.

Full declaration: `#define GuiConst_AUTOREDRAW_FIELDS_MAX XXX`

GuiConst_AUTOREDRAW_MAX_VAR_SIZE

Purpose: Size of biggest Auto redraw variable of all structures.

Full declaration: `#define GuiConst_AUTOREDRAW_MAX_VAR_SIZE XXX`

GuiConst_AUTOREDRAW_ON_CHANGE

Purpose: Auto redraw items are updated only if the controlling variable has changed, *or* if the item does not have a controlling variable. If this directive is not present Auto redraw items will be continuously updated, each time the GuiLib_Refresh function is called.

Full declaration: `#define GuiConst_AUTOREDRAW_ON_CHANGE`

GuiConst_AVR_COMPILER_FLASH_RAM

Purpose: Special flag for AVR compilers when using flash RAM.

Full declaration: `#define GuiConst_AVR_COMPILER_FLASH_RAM`

GuiConst_AVRGCC_COMPILER

Purpose: Special flag for AVR GCC compilers. Replaces the keyword `const` with the keyword `PROGMEM` in all easyGUI generated c/h files.

Full declaration: `#define GuiConst_AVRGCC_COMPILER`

GuiConst_BIT_BOTTOMRIGHT

Purpose: Defines that display bytes are oriented with bit zero at right (horizontal display bytes) or bottom (vertical display bytes).

Full declaration: `#define GuiConst_BIT_BOTTOMRIGHT`

GuiConst_BIT_TOPLEFT

Purpose: Defines that display bytes are oriented with bit zero at left (horizontal display bytes) or top (vertical display bytes).

Full declaration: `#define GuiConst_BIT_TOPLEFT`

GuiConst_BITMAP_SUPPORT_ON

Purpose: Bitmap module enabled.

Full declaration: `#define GuiConst_BITMAP_SUPPORT_ON`

GuiConst_BLINK_FIELDS_MAX

Purpose: Defines highest blink field number currently in use.

Full declaration: `#define GuiConst_BLINK_FIELDS_MAX`

GuiConst_BLINK_SUPPORT_ON

Purpose: Blink module enabled.

Full declaration: `#define GuiConst_BLINK_SUPPORT_ON`

GuiConst_BYTE_HORIZONTAL

Purpose: Defines that display bytes in the display controller are arranged in a horizontal manner.

Full declaration: `#define GuiConst_BYTE_HORIZONTAL`

GuiConst_BYTE_LINES

Purpose: No. of byte lines in the display (vertical or horizontal).

Full declaration: `#define GuiConst_BYTE_LINES XXX`

GuiConst_BYTE_VERTICAL

Purpose: Defines that display bytes in the display controller are arranged in a vertical manner.

Full declaration: `#define GuiConst_BYTE_VERTICAL`

GuiConst_BYTES_PR_LINE

Purpose: Bytes per display line (horizontal or vertical).

Full declaration: `#define GuiConst_BYTES_PR_LINE XXX`

GuiConst_BYTES_PR_SECTION

Purpose: Bytes per display line for each display controller. Only used if more than one display controller is used.

Full declaration: `#define GuiConst_BYTES_PR_SECTION XXX`

GuiConst_CHAR

Purpose: Character definition (8 bit unsigned). Default `char`.

Full declaration: `#define GuiConst_CHAR XXX`

GuiConst_CHARMODE_ANSI

Purpose: Character coding is ANSI, using 8 bit character size.

Full declaration: `#define GuiConst_CHARMODE_ANSI`

GuiConst_CHARMODE_UNICODE

Purpose: Character coding is Unicode, using 16 bit character size.

Full declaration: `#define GuiConst_CHARMODE_UNICODE`

GuiConst_CLIPPING_SUPPORT_ON

Purpose: Clipping module enabled.

Full declaration: `#define GuiConst_CLIPPING_SUPPORT_ON`

GuiConst_CODEVISION_COMPILER

Purpose: Special flag for CodeVision compilers. Inserts `flash` keywords where appropriate in easyGUI code.

Full declaration: `#define GuiConst_CODEVISION_COMPILER`

GuiConst_COLOR_BYTE_SIZE

Purpose: Size of color variables. Can be from 1 to 3 bytes in size.

Full declaration: `#define GuiConst_COLOR_BYTE_SIZE XXX`

GuiConst_COLOR_DEPTH_1

Purpose: Indicates a 1 bpp (monochrome) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_1`

GuiConst_COLOR_DEPTH_2

Purpose: Indicates a 2 bpp (grayscale) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_2`

GuiConst_COLOR_DEPTH_4

Purpose: Indicates a 4 bpp (grayscale or color) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_4`

GuiConst_COLOR_DEPTH_5

Purpose: Indicates a 5 bpp (grayscale) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_5`

GuiConst_COLOR_DEPTH_8

Purpose: Indicates a 8 bpp (grayscale or color) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_8`

GuiConst_COLOR_DEPTH_12

Purpose: Indicates a 12 bpp color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_12`

GuiConst_COLOR_DEPTH_15

Purpose: Indicates a 15 bpp color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_15`

GuiConst_COLOR_DEPTH_16

Purpose: Indicates a 16 bpp color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_16`

GuiConst_COLOR_DEPTH_18

Purpose: Indicates a 18 bpp color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_18`

GuiConst_COLOR_DEPTH_24

Purpose: Indicates a 24 bpp (truecolor) color depth.

Full declaration: `#define GuiConst_COLOR_DEPTH_24`

GuiConst_COLOR_MAX

Purpose: Defines the highest allowed color index (0~16777216).

Full declaration: `#define GuiConst_COLOR_MAX XXX`

GuiConst_COLOR_MODE_GRAY

Purpose: Specifies grayscale color mode. Only applicable to 1 bpp, 2 bpp, 4 bpp, and 8 bpp color depths.

Full declaration: `#define GuiConst_COLOR_MODE_GRAY`

GuiConst_COLOR_MODE_PALETTE

Purpose: Specifies palette based color mode. Only applicable to 4 bpp and 8 bpp color depths.

Full declaration: `#define GuiConst_COLOR_MODE_PALETTE`

GuiConst_COLOR_MODE_RGB

Purpose: Specifies RGB color mode. Only applicable to 8 bpp or higher color depths.

Full declaration: `#define GuiConst_COLOR_MODE_RGB`

GuiConst_COLOR_PLANES_1

Purpose: Indicates a normal single bit plane color system.

Full declaration: `#define GuiConst_COLOR_PLANES_1`

GuiConst_COLOR_PLANES_2

Purpose: Indicates a two bit plane color system, with a monochrome image in each plane, which combined gives a 2 bpp (grayscale) system.

Full declaration: `#define GuiConst_COLOR_PLANES_2`

GuiConst_COLOR_RGB_STANDARD

Purpose: Indicates that the system uses 24 bit color codes, (directly or via palette), with color bits organized as bits 0~7 red, bits 8~15 green, and bits 16~23 blue. Such a system is directly compatible with the internally used color system in the easyGUI library, and thus do not need conversion of color codes.

Full declaration: `#define GuiConst_COLOR_RGB_STANDARD`

GuiConst_COLOR_SIZE

Purpose: Color depth in bits per pixel (1~24).

Full declaration: `#define GuiConst_COLOR_SIZE XXX`

GuiConst_COLORCODING_B_MASK

Purpose: Bit mask for blue bits in color value. Indicated in hexadecimal value.

Full declaration: `#define GuiConst_COLORCODING_B_MASK XXX`

GuiConst_COLORCODING_B_MAX

Purpose: Defines the highest allowed blue color index (1~255).

Full declaration: `#define GuiConst_COLORCODING_B_MAX XXX`

GuiConst_COLORCODING_B_SIZE

Purpose: No. of bits with blue color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_B_SIZE XXX`

GuiConst_COLORCODING_B_START

Purpose: First bit with blue color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_B_START XXX`

GuiConst_COLORCODING_G_MASK

Purpose: Bit mask for green bits in color value. Indicated in hexadecimal value.

Full declaration: `#define GuiConst_COLORCODING_G_MASK XXX`

GuiConst_COLORCODING_G_MAX

Purpose: Defines the highest allowed green color index (1~255).

Full declaration: `#define GuiConst_COLORCODING_G_MAX XXX`

GuiConst_COLORCODING_G_SIZE

Purpose: No. of bits with green color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_G_SIZE XXX`

GuiConst_COLORCODING_G_START

Purpose: First bit with green color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_G_START XXX`

GuiConst_COLORCODING_R_MASK

Purpose: Bit mask for red bits in color value. Indicated in hexadecimal value.

Full declaration: `#define GuiConst_COLORCODING_R_MASK XXX`

GuiConst_COLORCODING_R_MAX

Purpose: Defines the highest allowed red color index (1~255).

Full declaration: `#define GuiConst_COLORCODING_R_MAX XXX`

GuiConst_COLORCODING_R_SIZE

Purpose: No. of bits with red color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_R_SIZE XXX`

GuiConst_COLORCODING_R_START

Purpose: First bit with red color information in palette/RGB entries.

Full declaration: `#define GuiConst_COLORCODING_R_START XXX`

GuiConst_CONTROLLER_COUNT_HORZ

Purpose: Indicates number of display controllers used horizontally, usually 1.

Full declaration: `#define GuiConst_CONTROLLER_COUNT_HORZ XXX`

GuiConst_CONTROLLER_COUNT_VERT

Purpose: Indicates number of display controllers used vertically, usually 1.

Full declaration: `#define GuiConst_CONTROLLER_COUNT_VERT XXX`

GuiConst_CURSOR_FIELDS_MAX

Purpose: Max. number of cursor fields, automatically calculated by easyGUI.

Full declaration: `#define GuiConst_CURSOR_FIELDS_MAX XXX`

GuiConst_CURSOR_MODE_STOP_TOP

Purpose: Indicates that cursor movement stops at the top/bottom cursor fields, i.e. no wrap around to the other end.

Full declaration: `#define GuiConst_CURSOR_MODE_TOP_STOP`

GuiConst_CURSOR_MODE_WRAP_AROUND

Purpose: Indicates that cursor movement wraps around at the top/bottom cursor fields, going from top cursor field to bottom cursor field, and vice versa.

Full declaration: `#define GuiConst_CURSOR_MODE_WRAP_AROUND`

GuiConst_CURSOR_SUPPORT_ON

Purpose: Cursor module enabled.

Full declaration: `#define GuiConst_CURSOR_SUPPORT_ON`

GuiConst_DECIMAL_COMMA

Purpose: The decimal character for floating point variables is a comma (12,34).

Full declaration: `#define GuiConst_DECIMAL_COMMA`

GuiConst_DECIMAL_PERIOD

Purpose: The decimal character for floating point variables is a period (12.34).

Full declaration: `#define GuiConst_DECIMAL_PERIOD`

GuiConst_DISPLAY_ACTIVE_AREA

Purpose: Indicates that a general active area has been defined globally for the display.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA`

GuiConst_DISPLAY_ACTIVE_AREA_CLIPPING

Purpose: Indicates that clipping is enabled for the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_CLIPPING`

GuiConst_DISPLAY_ACTIVE_AREA_COO_REL

Purpose: Indicates that the coordinate system origo is moved from the usual upper left corner of the display to the upper left corner of the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_COO_REL`

GuiConst_DISPLAY_ACTIVE_AREA_X1

Purpose: X1 coordinate of the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_X1 XXX`

GuiConst_DISPLAY_ACTIVE_AREA_Y1

Purpose: Y1 coordinate of the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_Y1 XXX`

GuiConst_DISPLAY_ACTIVE_AREA_X2

Purpose: X2 coordinate of the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_X2 XXX`

GuiConst_DISPLAY_ACTIVE_AREA_Y2

Purpose: Y2 coordinate of the general active area.

Full declaration: `#define GuiConst_DISPLAY_ACTIVE_AREA_Y2 XXX`

GuiConst_DISPLAY_BYTES

Purpose: No. of bytes for a full display image.

Full declaration: `#define GuiConst_DISPLAY_BYTES XXX`

GuiConst_DISPLAY_HEIGHT

Purpose: Virtual height of display in pixels. This is the height seen by easyGUI when handling the display. Differs from the physical display height only if the display is rotated 90°.

Full declaration: `#define GuiConst_DISPLAY_HEIGHT XXX`

GuiConst_DISPLAY_HEIGHT_HW

Purpose: Physical height of display in pixels. This is the height used by the display driver in `GuiDisplay.c`. Differs from the virtual display height only if the display is rotated 90°.

Full declaration: `#define GuiConst_DISPLAY_HEIGHT_HW XXX`

GuiConst_DISPLAY_WIDTH

Purpose: Virtual width of display in pixels. This is the width seen by easyGUI when handling the display. Differs from the physical display width only if the display is rotated 90°.

Full declaration: `#define GuiConst_DISPLAY_WIDTH XXX`

GuiConst_DISPLAY_WIDTH_HW

Purpose: Physical width of display in pixels. This is the width used by the display driver in `GuiDisplay.c`. Differs from the virtual display width only if the display is rotated 90°.

Full declaration: `#define GuiConst_DISPLAY_WIDTH_HW XXX`

GuiConst_FLOAT_SUPPORT_ON

Purpose: Float support module enabled.

Full declaration: `#define GuiConst_FLOAT_SUPPORT_ON`

GuiConst_FONT_UNCOMPRESSED

Purpose: Indicates that all font data are uncompressed.

Full declaration: `#define GuiConst_FONT_UNCOMPRESSED`

GuiConst_ICC_COMPILER

Purpose: Special flag for Imagecraft compilers.

Full declaration: `#define GuiConst_ICC_COMPILER`

GuiConst_INT8S

Purpose: 8 bit signed definition. Default `signed char`.

Full declaration: `#define GuiConst_INT8S XXX`

GuiConst_INT8U

Purpose: 8 bit unsigned definition. Default `unsigned char`.

Full declaration: `#define GuiConst_INT8U XXX`

GuiConst_INT16S

Purpose: 16 bit signed definition. Default `signed short`.

Full declaration: `#define GuiConst_INT16S XXX`

GuiConst_INT16U

Purpose: 16 bit unsigned definition. Default `unsigned short`.

Full declaration: `#define GuiConst_INT16U XXX`

GuiConst_INT24S

Purpose: 24 bit signed definition. Normally only used on systems with 24 bit addressing. Default undefined.

Full declaration: `#define GuiConst_INT24S XXX`

GuiConst_INT24U

Purpose: 24 bit unsigned definition. Normally only used on systems with 24 bit addressing. Default undefined.

Full declaration: `#define GuiConst_INT24U XXX`

GuiConst_INT32S

Purpose: 32 bit signed definition. Default signed long.

Full declaration: `#define GuiConst_INT32S XXX`

GuiConst_INT32U

Purpose: 32 bit unsigned definition. Default unsigned long.

Full declaration: `#define GuiConst_INT32U XXX`

GuiConst_INTCOLOR

Purpose: Specifies the type of color variables. Can be from 8 to 32 bits in size.

Full declaration: `#define GuiConst_INTCOLOR GuiConst_INTXXXU`

GuiConst_ITEM_TEXTBLOCK_INUSE

Purpose: One or more Paragraph items in use. If no Paragraph items are present this directive will not be present either, and some library code related to paragraph text drawing is spared from compiling.

Full declaration: `#define GuiConst_ITEM_TEXTBLOCK_INUSE`

GuiConst_ITEM_TOUCHAREA_INUSE

Purpose: One or more touch areas in use. If no touch areas are present this directive will not be present either, and some library code related to touch areas handling is spared from compiling.

Full declaration: `#define GuiConst_ITEM_TOUCHAREA_INUSE`

GuiConst_KEIL_COMPILER_REENTRANT

Purpose: Special flag for Keil 8051 compilers when using recursive functions. Adds the keyword `reentrant` to all recursively called functions in the easyGUI library. If this setting is not used easyGUI will typically display graphics primitives and simple screen structures correctly, but fail to display complex screen structures.

Full declaration: `#define GuiConst_KEIL_COMPILER_REENTRANT`

GuiConst_LANGUAGE_CNT

Purpose: Count of defined languages.

Full declaration: `#define GuiConst_LANGUAGE_CNT XXX`

GuiConst_LANGUAGE_XXX

Purpose: Defines a specific language. The XXX is the language name, taken from Language setup in easyGUI. The keyword is followed by the language index.

Full declaration: `#define GuiConst_LANGUAGE_XXX XXX`

GuiConst_MAX_TEXT_LEN

Purpose: Max. length of single text string, when displaying structures. Can be up to 255 characters. Smaller values will conserve RAM space on the target system.

Full declaration: `#define GuiConst_MAX_TEXT_LEN XXX`

GuiConst_MAX_VARNUM_TEXT_LEN

Purpose: Max. length of numerical variable text representations, when displaying structures. Can be up to 255 characters. Smaller values will conserve RAM space on the target system.

Full declaration: `#define GuiConst_MAX_VARNUM_TEXT_LEN XXX`

GuiConst_MIRRORED_HORIZONTALLY

Purpose: Indicates that the display output is mirrored horizontally.

Full declaration: `#define GuiConst_MIRRORED_HORIZONTALLY`

GuiConst_MIRRORED_VERTICALLY

Purpose: Indicates that the display output is mirrored vertically.

Full declaration: `#define GuiConst_MIRRORED_VERTICALLY`

GuiConst_PALETTE_SIZE

Purpose: No. of entries in the palette. Can be 16 or 256.

Full declaration: `#define GuiConst_PALETTE_SIZE XXX`

GuiConst_PICC_COMPILER_ROM

Purpose: Special flag for Microchip PicC compilers when handling ROM.

Full declaration: `#define GuiConst_PICC_COMPILER_ROM`

GuiConst_PIXEL_OFF

Purpose: Color of Pixel OFF pixels.

Full declaration: `#define GuiConst_PIXEL_OFF XXX`

GuiConst_PIXEL_ON

Purpose: Color of Pixel ON pixels.

Full declaration: `#define GuiConst_PIXEL_ON XXX`

GuiConst_PTR

Purpose: Pointer size definition. Will be equal to `GuiConst_INT16U`, `GuiConst_INT24U`, `GuiConst_INT32U`, `void *`, or `void *const`.

Full declaration: `#define GuiConst_PTR XXX`

GuiConst_REL_COORD_ORIGO_INUSE

Purpose: Indicates that either a globally defined active area, or at least one Active area item, is using relative coordinate system.

Full declaration: `#define GuiConst_REL_COORD_ORIGO_INUSE`

GuiConst_REVERSED_BYTE_PAIRS

Purpose: Swaps bytes for 16 bit display RAM values. Only applicable to 4 bpp color depth with horizontal byte orientation, and 8 bpp color depth.

Full declaration: `#define GuiConst_REVERSED_BYTE_PAIRS`

GuiConst_ROTATED90DEGREE

Purpose: Defines that the display is used in a 90° rotated mode (either left or right).

Full declaration: `#define GuiConst_ROTATED90DEGREE`

GuiConst_ROTATED90DEGREE_LEFT

Purpose: Defines that the display is used in a 90° rotated left mode.

Full declaration: `#define GuiConst_ROTATED90DEGREE_LEFT`

GuiConst_ROTATED90DEGREE_RIGHT

Purpose: Defines that the display is used in a 90° rotated right mode.

Full declaration: `#define GuiConst_ROTATED90DEGREE_RIGHT`

GuiConst_ROTATED_OFF

Purpose: Defines that the display is used in a normal 0° rotation mode.

Full declaration: `#define GuiConst_ROTATED_OFF`

GuiConst_ROTATED_UPSIDEDOWN

Purpose: Defines that the display is used in a 180° rotated upside-down mode.

Full declaration: `#define GuiConst_ROTATED_UPSIDEDOWN`

GuiConst_SCROLL_MODE_STOP_TOP

Purpose: Indicates that scroll box navigation stops at the top/bottom scroll line, i.e. no wrap around to the other end of the scroll list.

Full declaration: `#define GuiConst_SCROLL_MODE_TOP_STOP`

GuiConst_SCROLL_MODE_WRAP_AROUND

Purpose: Indicates that scroll box navigation wraps around at the top/bottom scroll line, going from top line to the bottom line, and vice versa.

Full declaration: `#define GuiConst_SCROLL_MODE_WRAP_AROUND`

GuiConst_SCROLL_SUPPORT_ON

Purpose: Scroll module enabled.

Full declaration: `#define GuiConst_SCROLL_SUPPORT_ON`

GuiConst_TEXT

Purpose: Character definition. Will be equal to `GuiConst_CHAR` or `GuiConst_INT16U`, depending on the use of ANSI or Unicode character coding.

Full declaration: `#define GuiConst_TEXT XXX`

GuiConst_TOUCHAREA_MAX

Purpose: Defines highest touch area ID number currently in use.

Full declaration: `#define GuiConst_TOUCHAREA_MAX`

GUILIB UNIT

This unit contains the core library for handling the easyGUI system. The unit should *never* be edited, as this will make updating to newer versions of the easyGUI library difficult. The unit must have the same version number as the easyGUI Windows application in use.

The `GuiLib` unit includes one of the graphics primitives files `GuiGraph1H.c`, `GuiGraph1V.c`, `GuiGraph2H.c`, `GuiGraph2V.c`, `GuiGraph2H2P.c`, `GuiGraph2V2P.c`, `GuiGraph4H.c`, `GuiGraph4V.c`, `GuiGraph5.c`, `GuiGraph8.c`, `GuiGraph16.c`, or `GuiGraph24.c`, depending on the selected display controller setup.



These graphical primitives files should *not* be included in the compiler/linker setup of the target system.

The following constants, variables and functions are available (in alphabetical order):

Constants

Constants only relevant internally between the easyGUI units are not mentioned.

GuiLib_CHR_SET

Purpose: No. of character sets in use in the fonts. Only relevant in ANSI character mode.

Full declaration: `#define GuiLib_CHR_SET 2`

GuiLib_NO_CURSOR

Purpose: Used in `GuiLib_ShowScreen` function call. No cursor should be displayed.

Full declaration: `#define GuiLib_NO_CURSOR -1`

GuiLib_NO_RESET_AUTO_REDRAW

Purpose: Used in `GuiLib_ShowScreen` function call. Auto redraw items from previously shown structures shall be maintained.

Full declaration: `#define GuiLib_NO_RESET_AUTO_REDRAW 0`

GuiLib_RESET_AUTO_REDRAW

Purpose: Used in `GuiLib_ShowScreen` function call. Auto redraw items from previously shown structures shall be erased.

Full declaration: `#define GuiLib_RESET_AUTO_REDRAW 1`

Variables

Variables only relevant internally between the easyGUI units are not mentioned.

GuiLib_ActiveCursorFieldNo

Purpose: Contains the currently active cursor field number, with zero being the first cursor field. Do not change it directly, but call functions `GuiLib_Cursor_Select`, `GuiLib_Cursor_Up`, or `GuiLib_Cursor_Down` instead.

Full declaration: `GuiConst_INT16S GuiLib_ActiveCursorFieldNo;`

GuiLib_CurStructureNdx

Purpose: Contains the index number to the currently displayed structure. Is initially set to -1. After a call to the `GuiLib_Clear` function it is reset to -1.

Full declaration: `GuiConst_INT16S GuiLib_CurStructureNdx;`

GuiLib_LanguageCharSet

Purpose: Contains the currently selected language character set. Character set zero is the default character set containing the standard ANSI characters, as used in Windows. Only relevant in ANSI character mode.

Full declaration: `GuiConst_INT16S GuiLib_LanguageCharSet;`

GuiLib_LanguageIndex

Purpose: Contains the currently selected language index, with index zero being the reference language. To change the language use the `GuiLib_SetLanguage` function.

Full declaration: `GuiConst_INT16S GuiLib_LanguageIndex;`

GuiLib_ScrollActiveLine

Purpose: Contains the currently active scroll line, with zero being the topmost scroll line. Do not change it directly, but call functions `GuiLib_Scroll_Up`, or `GuiLib_Scroll_Down` instead.

Full declaration: `GuiConst_INT16S GuiLib_ScrollActiveLine;`

GuiLib_ScrollTopLine

Purpose: Contains the currently topmost visible line in a scroll box.

Full declaration: `GuiConst_INT16S GuiLib_ScrollTopLine;`

GuiLib_ScrollVisibleLines

Purpose: Contains the number of visible lines in a scroll box. The number is the theoretical number of visible lines in the box, not the current count of lines with some kind of contents.

Full declaration: `GuiConst_INT16S GuiLib_ScrollVisibleLines;`

Functions

GuiLib_BlinkBoxMarkedItem

Purpose: Sets parameters for blinking item.

Remarks: Removed if blink support is disabled.

Full declaration:

```
void GuiLib_BlinkBoxMarkedItem(
    GuiConst_INT16U BlinkFieldNo,
    GuiConst_INT16U CharNo,
    GuiConst_INT16S Rate);
```

Input: Blink item index number.

Character number. If character zero is selected the entire text will blink. If character one or higher is selected only that single character will blink.

Blinking rate, in multiples of `GuiLib_Refresh` refresh rate, valid range 0-255.

Output: None.

Related functions: `GuiLib_BlinkBoxStop`

GuiLib_BlinkBoxStart

Purpose: Sets parameters for blinking box function.

Remarks: Removed if blink support is disabled.

Full declaration:

```
void GuiLib_BlinkBoxStart(
    GuiConst_INT16S X1,
    GuiConst_INT16S Y1,
    GuiConst_INT16S X2,
    GuiConst_INT16S Y2,
    GuiConst_INT16S Rate);
```

Input: Rectangle coordinates.

Blinking rate, in multiples of `GuiLib_Refresh` refresh rate, valid range 0-255.

Output: None.

Related functions: `GuiLib_BlinkBoxStop`

GuiLib_BlinkBoxStop

Purpose: Stops blinking, both if started by a `GuiLib_BlinkBoxStart` call or a `GuiLib_BlinkBoxMarkedItem` call.

Remarks: Removed if blink support is disabled.

Full declaration: `void GuiLib_BlinkBoxStop(void);`

Input: None.

Output: None.

Related functions: `GuiLib_BlinkBoxStart`

GuiLib_BorderBox

Purpose: Draws a filled rectangle with single pixel border.

Full declaration:

```
void GuiLib_BorderBox(
    GuiConst_INT16S X1,
    GuiConst_INT16S Y1,
    GuiConst_INT16S X2,
    GuiConst_INT16S Y2,
    GuiConst_INTCOLOR BorderColor,
    GuiConst_INTCOLOR FillColor);
```

Input: Coordinates.
Fill and border colors.

Output: None.

Related functions: `GuiLib_Box`
`GuiLib_FillBox`

GuiLib_Box

Purpose: Draws a single pixel wide rectangle.

Full declaration:

```
void GuiLib_Box(
    GuiConst_INT16S X1,
    GuiConst_INT16S Y1,
    GuiConst_INT16S X2,
    GuiConst_INT16S Y2,
    GuiConst_INTCOLOR Color);
```

Input: Coordinates.
Color.

Output: None.

Related functions: `GuiLib_BorderBox`
`GuiLib_FillBox`

GuiLib_Clear

Purpose: Clears the screen. Clears flags for cursors, auto redraw items, and scrolling.

Full declaration: `void GuiLib_Clear(void);`

Input: None.

Output: None.

Related functions: `GuiLib_ClearDisplay`

GuiLib_ClearDisplay

Purpose: Clears the screen.

Full declaration: `void GuiLib_ClearDisplay(void);`

Input: None.

Output: None.

Related functions: `GuiLib_Clear`

GuiLib_Cursor_Down

Purpose: Makes next cursor field active, redrawing both current and new cursor field.

Remarks: Removed if cursor support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Cursor_Down(void);`

Input: None.

Output: 0: Cursor at end of range.
1: Cursor moved.

Related functions: `GuiLib_Cursor_End`
`GuiLib_Cursor_Home`
`GuiLib_Cursor_Select`
`GuiLib_Cursor_Up`

GuiLib_Cursor_End

Purpose: Makes last cursor field active, redrawing both current and new cursor field.

Remarks: Removed if cursor support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Cursor_End(void);`

Input: None.

Output: 0: Cursor at end of range.
1: Cursor moved.

Related functions: `GuiLib_Cursor_Down`
`GuiLib_Cursor_Home`
`GuiLib_Cursor_Select`
`GuiLib_Cursor_Up`

GuiLib_Cursor_Home

Purpose: Makes first cursor field active, redrawing both current and new cursor field.

Remarks: Removed if cursor support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Cursor_Home(void);`

Input: None.

Output: 0: Cursor at end of range.
1: Cursor moved.

Related functions: `GuiLib_Cursor_Down`
`GuiLib_Cursor_End`
`GuiLib_Cursor_Select`
`GuiLib_Cursor_Up`

GuiLib_Cursor_Select

Purpose: Makes requested cursor field active, redrawing both current and new cursor field.

Remarks: Removed if cursor support is disabled.

Full declaration: `void GuiLib_Cursor_Select(
GuiConst_INT16S NewCursorFieldNo);`

Input: New cursor field No.

Output: None.

Related functions: `GuiLib_Cursor_Down`
`GuiLib_Cursor_End`
`GuiLib_Cursor_Home`
`GuiLib_Cursor_Up`

GuiLib_Cursor_Up

Purpose: Makes previous cursor field active, redrawing both current and new cursor field.

Remarks: Removed if cursor support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Cursor_Up(void);`

Input: None.

Output: 0: Cursor at end of range.
 1: Cursor moved.

Related functions: `GuiLib_Cursor_Down`
`GuiLib_Cursor_End`
`GuiLib_Cursor_Home`
`GuiLib_Cursor_Select`

GuiLib_Dot

Purpose: Draws a single pixel.

Full declaration: `void GuiLib_Dot(
 GuiConst_INT16S X,
 GuiConst_INT16S Y,
 GuiConst_INTCOLOR Color);`

Input: Coordinates.
 Color.

Output: None.

GuiLib_DrawChar

Purpose: Draws a single character on the display.

Full declaration: `void GuiLib_DrawChar(
 GuiConst_INT16S X,
 GuiConst_INT16S Y,
 GuiConst_INT16S FontNo,
 GuiConst_CHAR Character,
 GuiConst_INTCOLOR Color);`

Input: Coordinates.
 Font index.
 Character.
 Color.

Output: None.

Related functions: GuiLib_DrawStr

GuiLib_DrawStr

Purpose: Draws a formatted string on the display.

Full declaration: ANSI character mode:

```
void GuiLib_DrawStr(
    GuiConst_INT16S X,
    GuiConst_INT16S Y,
    GuiConst_INT16U FontNo,
    GuiConst_INT8S CharSetSelector,
    GuiConst_TEXT *String,
    GuiConst_INT8U Alignment,
    GuiConst_INT8U PsWriting,
    GuiConst_INT8U Transparent,
    GuiConst_INT8U Underlining,
    GuiConst_INT16S BackBoxSizeX,
    GuiConst_INT16S BackBoxSizeY1,
    GuiConst_INT16S BackBoxSizeY2,
    GuiConst_INT8U BackBorderPixels,
    GuiConst_INTCOLOR ForeColor,
    GuiConst_INTCOLOR BackColor);
```

Unicode character mode:

```
void GuiLib_DrawStr(
    GuiConst_INT16S X,
    GuiConst_INT16S Y,
    GuiConst_INT16U FontNo,
    GuiConst_TEXT *String,
    GuiConst_INT8U Alignment,
    GuiConst_INT8U PsWriting,
    GuiConst_INT8U Transparent,
    GuiConst_INT8U Underlining,
    GuiConst_INT16S BackBoxSizeX,
    GuiConst_INT16S BackBoxSizeY1,
    GuiConst_INT16S BackBoxSizeY2,
    GuiConst_INT8U BackBorderPixels,
    GuiConst_INTCOLOR ForeColor,
    GuiConst_INTCOLOR BackColor);
```

Input: Coordinates.
 Font index.

Character set selection. -1 selects the default character set, 0 selects the standard ANSI character set, >0 selects special national character sets. Only used in ANSI character mode.

String. A zero terminated text string.

Alignment. Can be:

- `GuiLib_ALIGN_LEFT` starts text writing from the X coordinate.
- `GuiLib_ALIGN_CENTER` centers text writing around the X coordinate.
- `GuiLib_ALIGN_RIGHT` positions the text so that it ends on the X coordinate.

Proportional writing. Can be:

- `GuiLib_PS_OFF` turns off proportional writing.
- `GuiLib_PS_ON` turns on proportional writing.
- `GuiLib_PS_NUM` uses numerical proportional writing.

Transparent. Can be:

- `GuiLib_TRANSPARENT_OFF` turns transparent writing off, i.e. the background is painted.
- `GuiLib_TRANSPARENT_ON` turns transparent writing on, i.e. only the text is painted.

Underlining. Can be:

- `GuiLib_UNDERLINE_OFF`.
- `GuiLib_UNDERLINE_ON`.

Background box size X. Determines the width of a background box. Zero means no background box.

Background box size Y1. Determines the height of a background box, measured from the font baseline and up. Zero means same height as font height above the baseline.

Background box size Y2. Determines the height of a background box, measured from the font baseline and down. Zero means same height as font height below the baseline.

Border pixels for background box. One extra pixel can be added to the background box on each of its edges:

- `GuiLib_BBP_NONE`. No extra pixels.
- `GuiLib_BBP_LEFT`. One extra pixel on the left edge.
- `GuiLib_BBP_RIGHT`. One extra pixel on the right edge.
- `GuiLib_BBP_TOP`. One extra pixel on the top edge.
- `GuiLib_BBP_BOTTOM`. One extra pixel on the bottom edge.

The last four settings can be combined, like e.g.
`GuiLib_BBP_TOP + GuiLib_BBP_BOTTOM.`

Foreground color. Determines the text color.

Background color. Determines the background color (if used, either for normal background (=non transparent) or background box).

Output: None.

Related functions: `GuiLib_DrawChar`

GuiLib_FillBox

Purpose: Draws a filled rectangle.

Full declaration:

```
void GuiLib_FillBox(
    GuiConst_INT16S X1,
    GuiConst_INT16S Y1,
    GuiConst_INT16S X2,
    GuiConst_INT16S Y2,
    GuiConst_INTCOLOR Color);
```

Input: Coordinates.
 Color.

Output: None.

Related functions: `GuiLib_BorderBox`
`GuiLib_Box`

GuiLib_GetDot

Purpose: Returns the color of a single pixel.

Full declaration:

```
char *GuiLib_GetDot(
    GuiConst_INT16S X,
    GuiConst_INT16S Y);
```

Input: Coordinates.

Output: Color.

GuiLib_GetTextLanguagePtr

Purpose: Returns pointer to text in structure. Language of selected text can be freely selected, no matter what language is active.

Full declaration:

```
char *GuiLib_GetTextLanguagePtr(
    GuiConst_INT16U Structure,
    GuiConst_INT16U TextNo,
    GuiConst_INT16S LanguageIndex);
```

Input: Structure ID.
 Text No. - 0 is first text in structure, Items other than texts are ignored.
 Language index.

Output: Pointer to text based on structure, text No. and current language.
 Returns Nil if no text was found.

Related functions: `GuiLib_GetTextPtr`
`GuiLib_GetTextWidth`

GuiLib_GetTextPtr

Purpose: Returns pointer to text in structure.

Full declaration: `char *GuiLib_GetTextPtr(
 GuiConst_INT16U Structure,
 GuiConst_INT16U TextNo);`

Input: Structure ID.
 Text No. - 0 is first text in structure, Items other than texts are ignored.

Output: Pointer to text based on structure, text No. and current language.
 Returns Nil if no text was found.

Related functions: `GuiLib_GetTextLanguagePtr`
`GuiLib_GetTextWidth`

GuiLib_GetTextWidth

Purpose: Returns width of text in pixels.

Full declaration: `GuiConst_INT16U GuiLib_GetTextWidth(
 char *String,
 GuiLib_FontRecConstPtr Font,
 GuiConst_INT8U PsWriting);`

Input: Pointer to text string.
 Pointer to easyGUI font.

Output: Width of text in pixels, returns zero if an error is encountered.

Related functions: `GuiLib_GetTextLanguagePtr`
`GuiLib_GetTextPtr`

GuiLib_GrayScaleToRgbColor

Purpose: Translates from 0~255 gray scale value to RGB color.

Full declaration: `GuiConst_INT32U GuiLib_GrayScaleToRgbColor(
GuiConst_INT8U GrayValue);`

Input: Gray scale value, 0~255.

Output: RGB color value (32 bit, 24 bits used, low byte = Red, middle byte = Green, high byte = Blue).

Related functions: `GuiLib_RgbColorToGrayScale`

GuiLib_HLine

Purpose: Draws a horizontal line.

Full declaration: `void GuiLib_HLine(
GuiConst_INT16S X1,
GuiConst_INT16S X2,
GuiConst_INT16S Y,
GuiConst_INTCOLOR Color);`

Input: Coordinates.
Color.

Output: None.

Related functions: `GuiLib_Line`
`GuiLib_VLine`

GuiLib_Init

Purpose: Initializes the easyGUI modules. Shall only be called once at application startup.

Full declaration: `void GuiLib_Init(void);`

Input: None.

Output: None.

GuiLib_InvertBox

Purpose: Inverts a block.

Full declaration: `void GuiLib_InvertBox(
GuiConst_INT16S X1,
GuiConst_INT16S Y1,
GuiConst_INT16S X2,
GuiConst_INT16S Y2);`

Input: Coordinates.

Output: None.

GuiLib_InvertBoxStart

Purpose: Sets parameters for inverted box function.

Full declaration:

```
void GuiLib_InvertBoxStart(  
    GuiConst_INT16S X1,  
    GuiConst_INT16S Y1,  
    GuiConst_INT16S X2,  
    GuiConst_INT16S Y2);
```

Input: Rectangle coordinates.

Output: None.

Related functions: `GuiLib_InvertBoxStop`

GuiLib_InvertBoxStop

Purpose: Stops inverted box function.

Full declaration:

```
void GuiLib_InvertBoxStop(void);
```

Input: None.

Output: None.

Related functions: `GuiLib_InvertBoxStart`

GuiLib_Line

Purpose: Draws a line. Lines with any slant are handled.

Full declaration:

```
void GuiLib_Line(  
    GuiConst_INT16S X1,  
    GuiConst_INT16S Y1,  
    GuiConst_INT16S X2,  
    GuiConst_INT16S Y2,  
    GuiConst_INTCOLOR Color);
```

Input: Coordinates.
Color.

Output: None.

Related functions: `GuiLib_HLine`
`GuiLib_VLine`

GuiLib_MarkDisplayBoxRepaint

Purpose: Sets the repainting scan line markers, indicating that all pixels inside the specified rectangle must be repainted. The display bytes covering this rectangle will be sent to the display controller next time the display is refreshed.

Full declaration:

```
void GuiLib_MarkDisplayBoxRepaint(  
    GuiConst_INT16S X1,  
    GuiConst_INT16S Y1,  
    GuiConst_INT16S X2,  
    GuiConst_INT16S Y2);
```

Input: Rectangle coordinates.

Output: None.

Related functions: `GuiLib_ResetDisplayRepaint`

GuiLib_PixelToRgbColor

Purpose: Translates from pixel value for display controller color setup to RGB color.

Full declaration:

```
GuiConst_INT32U GuiLib_PixelToRgbColor(  
    GuiConst_INTCOLOR PixelColor);
```

Input: Encoded pixel color value.

Output: RGB color value (32 bit, 24 bits used, low byte = Red, middle byte = Green, high byte = Blue).

Related functions: `GuiLib_RgbToPixelColor`

GuiLib_RedrawScrollList

Purpose: Redraws scroll list items

Remarks: Removed if scroll support is disabled.

Full declaration:

```
void GuiLib_RedrawScrollList(void);
```

Input: None.

Output: None.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_Home`

GuiLib_Scroll_End
GuiLib_Scroll_To_Line

GuiLib_Refresh

Purpose: Refreshes variables and updates display.

Full declaration: `void GuiLib_Refresh(void);`

Input: None.

Output: None.

GuiLib_ResetClipping

Purpose: Resets clipping. Drawing can be limited to a rectangular portion of the screen, this routine resets the clipping limits to the entire screen.

Remarks: Removed if clipping support is disabled.

Full declaration: `void GuiLib_ResetClipping(void);`

Input: None.

Output: None.

Related functions: `GuiLib_SetClipping`

GuiLib_ResetDisplayRepaint

Purpose: Resets the repainting scan line markers, so that no part of the image is marked. Therefore, next time the display is refreshed nothing is sent to the display controller.

Full declaration: `void GuiLib_ResetDisplayRepaint(void);`

Input: None.

Output: None.

Related functions: `GuiLib_MarkDisplayBoxRepaint`

GuiLib_RgbColorToGrayScale

Purpose: Translates from RGB color to 0~255 gray scale value.

Full declaration: `GuiConst_INT8U GuiLib_RgbColorToGrayScale(
GuiConst_INT32U RgbColor);`

Input: RGB color value (32 bit, 24 bits used, low byte = Red, middle byte = Green, high byte = Blue).

Output: Gray scale value, 0~255.

Related functions: `GuiLib_GrayScaleToRgbColor`

GuiLib_RgbToPixelColor

Purpose: Translates from RGB color to proper pixel value for display controller color setup.

Full declaration: `GuiConst_INTCOLOR GuiLib_RgbToPixelColor(GuiConst_INT32U RgbColor);`

Input: RGB color value (32 bit, 24 bits used, low byte = Red, middle byte = Green, high byte = Blue).

Output: Encoded pixel color value.

Related functions: `GuiLib_PixelToRgbColor`

GuiLib_Scroll_Down

Purpose: Makes next scroll line active, and scrolls list if needed.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Scroll_Down(void);`

Input: None.

Output: 0: No change, list already at bottom.
1: Active scroll line changed.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_Home`
`GuiLib_Scroll_End`
`GuiLib_Scroll_To_Line`

GuiLib_Scroll_End

Purpose: Makes last scroll line active, and scrolls list if needed.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Scroll_End(void);`

Input: None.

Output: 0: No change, list already at bottom.
1: Active scroll line changed.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_Home`
`GuiLib_Scroll_To_Line`

GuiLib_Scroll_Home

Purpose: Makes first scroll line active, and scrolls list if needed.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Scroll_Home(void);`

Input: None.

Output: 0: No change, list already at top.
1: Active scroll line changed.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_End`
`GuiLib_Scroll_To_Line`

GuiLib_Scroll_To_Line

Purpose: Makes specified scroll line active, and scrolls list if needed.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Scroll_To_Line(
GuiConst_INT16S NewLine);`

Input: Scroll line, zero is first line.

Output: 0: No change, list already at specified line.
1: Active scroll line changed.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`


```
GuiLib_ScrollLineOffsetY  
GuiLib_Scroll_Down  
GuiLib_Scroll_Up  
GuiLib_Scroll_Home  
GuiLib_Scroll_End
```

GuiLib_Scroll_Up

Purpose: Makes previous scroll line active, and scrolls list if needed.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT8U GuiLib_Scroll_Up(void);`

Input: None.

Output: 0: No change, list already at top.
1: Active scroll line changed.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Home`
`GuiLib_Scroll_End`
`GuiLib_Scroll_To_Line`

GuiLib_ScrollLineOffsetY

Purpose: Returns Y coordinate offset for active scroll line, zero if active line is at top of visible scroll area.

Remarks: Removed if scroll support is disabled.

Full declaration: `GuiConst_INT16S GuiLib_ScrollLineOffsetY(void);`

Input: None.

Output: Y coordinate offset in pixels.

Related functions: `GuiLib_SetScrollPars`
`GuiLib_RedrawScrollList`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_Home`
`GuiLib_Scroll_End`
`GuiLib_Scroll_To_Line`

GuiLib_SetClipping

Purpose: Sets clipping. Drawing can be limited to a rectangular portion of the screen, this routine sets the clipping limits expressed as two corner coordinates. Eventual drawing falling outside the clipping rectangle is ignored. Default for the clipping rectangle is the entire screen.

Remarks: Removed if clipping support is disabled.

Full declaration:

```
void GuiLib_SetClipping(  
    GuiConst_INT16S X1,  
    GuiConst_INT16S Y1,  
    GuiConst_INT16S X2,  
    GuiConst_INT16S Y2);
```

Input: Rectangle coordinates.

Output: None.

Related functions: `GuiLib_ResetClipping`

GuiLib_SetLanguage

Purpose: Selects current language. Index zero is the reference language.

Full declaration: `void GuiLib_SetLanguage(GuiConst_INT16S NewLanguage);`

Input: Language index.

Output: None.

GuiLib_SetScrollPars

Purpose: Sets parameters for scroll box functions. Should be called immediately *before* `GuiLib_ShowScreen` function call for the structure containing the scroll box.

Remarks: Removed if scroll support is disabled.

Full declaration:

```
void GuiLib_SetScrollPars(  
    void (*DataFuncPtr) (GuiConst_INT16S LineIndex),  
    GuiConst_INT16S NoOfLines,  
    GuiConst_INT16S ActiveLine);
```

Input: **DataFuncPtr:** Address of function of type
`void F(GEdit_INT16S LineIndex)`

NoOfLines: Total No. of lines in scroll box.

ActiveLine: Active scroll line (will be shown inverted), -1 means no bar, just a scrolling window.

Output: None.

Related functions: `GuiLib_RedrawScrollList`
`GuiLib_ScrollLineOffsetY`
`GuiLib_Scroll_Down`
`GuiLib_Scroll_Up`
`GuiLib_Scroll_Home`
`GuiLib_Scroll_End`
`GuiLib_Scroll_To_Line`

GuiLib_ShowBitmap

Purpose: Displays a stored bitmap.

Remarks: Removed if bitmap support is disabled.

Full declaration:

```
void GuiLib_ShowBitmap(
    GuiConst_INT8U BitmapIndex,
    GuiConst_INT16S X,
    GuiConst_INT16S Y);
```

Input: Bitmap index in `GuiStruct_BitmapPtrList`.

Coordinates for upper left corner

Output: None.

Related functions: `GuiLib_ShowBitmapAt`

GuiLib_ShowBitmapAt

Purpose: Displays a bitmap at a specific address.

Remarks: Removed if bitmap support is disabled.

Full declaration:

```
void GuiLib_ShowBitmapAt(
    GuiConst_INT8U * BitmapPtr,
    GuiConst_INT16S X,
    GuiConst_INT16S Y);
```

Input: Pointer to memory area.

Coordinates for upper left corner

Output: None.

Related functions: `GuiLib_ShowBitmap`

GuiLib_ShowScreen

Purpose: Instructs structure drawing task to draw a complete structure.

Full declaration: `void GuiLib_ShowScreen(
 GuiConst_INT16U Structure,
 GuiConst_INT16S CursorFieldToShow,
 GuiConst_INT8U ResetAutoRedraw);`

Input: Structure ID.

Active cursor field No. - enter `GuiLib_NO_CURSOR` if there is no cursor to show.

Maintain or erase old auto redraw items - use `GuiLib_NO_RESET_AUTO_REDRAW` Or `GuiLib_RESET_AUTO_REDRAW`.

Output: None.

GuiLib_StrAnsiToUnicode

Purpose: Converts ANSI string to Unicode string.

Remarks: Only accessible in Unicode character mode. Unicode string must have sufficient space for converted string.

Full declaration: `void GuiLib_StrAnsiToUnicode(
 GuiConst_TEXT *S2,
 GuiConst_CHAR *S1);`

Input: ANSI and Unicode string references.

Output: None.

Related functions: `GuiLib_UnicodeStrCmp`
`GuiLib_UnicodeStrCpy`
`GuiLib_UnicodeStrLen`

GuiLib_TestPattern

Purpose: Shows the test pattern used for initial development of the display controller driver. See the chapter on how to set up the system.

Full declaration: `void GuiLib_TestPattern(void);`

Input: None.

Output: None.

GuiLib_TouchAdjustReset

Purpose: Resets touch coordinate conversion.

Full declaration: `void GuiLib_TouchAdjustReset(void);`

Input: None.

Output: None.

GuiLib_TouchAdjustSet

Purpose: Sets one coordinate pair for touch coordinate conversion. Must be called two times, once for each of two diagonally opposed corners, or four times, once for each of the corners. The corner positions should be as close as possible to the physical display corners, as precision is lowered when going towards the display center.

Full declaration:

```
void GuiLib_TouchAdjustSet(
    GuiConst_INT16S XTrue,
    GuiConst_INT16S YTrue,
    GuiConst_INT16S XMeasured,
    GuiConst_INT16S YMeasured);
```

Input: XTrue,YTrue: Position represented in display coordinates.
XMeasured, YMeasured: Position represented in touch interface coordinates.

Output: None.

GuiLib_TouchCheck

Purpose: Returns touch area No. corresponding to the supplied coordinates. If no touch area is found at coordinates -1 is returned. Touch coordinates are converted to display coordinates, if conversion parameters have been set with the GuiLib_TouchAdjustSet function.

Full declaration:

```
void GuiLib_TouchCheck(
    GuiConst_INT16S X,
    GuiConst_INT16S Y);
```

Input: Touch position in touch interface coordinates.

Output: -1: No touch area found.
>=0: Touch area No.

GuiLib_UnicodeStrCmp

Purpose: Compares two Unicode strings.

Remarks: Only accessible in Unicode character mode.

Full declaration:

```
GuiConst_INT16S GuiLib_UnicodeStrCmp(
    GuiConst_TEXT *S1,
    GuiConst_TEXT *S2);
```

Input: Unicode string references

Output: <0: S1 is less than S2
 =0: S1 and S2 are equal
 >0: S1 is greater than S2

Related functions: `GuiLib_StrAnsiToUnicode`
`GuiLib_UnicodeStrCpy`
`GuiLib_UnicodeStrLen`

GuiLib_UnicodeStrCpy

Purpose: Copy from one Unicode string to another.

Remarks: Only accessible in Unicode character mode.

Full declaration: `void GuiLib_UnicodeStrCpy(
 GuiConst_TEXT *S2,
 GuiConst_TEXT *S1);`

Input: S1: Unicode source string reference
 S2: Unicode destination string reference
 Unicode destination string must have sufficient space

Output: None.

Related functions: `GuiLib_StrAnsiToUnicode`
`GuiLib_UnicodeStrCmp`
`GuiLib_UnicodeStrLen`

GuiLib_UnicodeStrLen

Purpose: Calculates length of Unicode string.

Remarks: Only accessible in Unicode character mode.

Full declaration: `GuiConst_INT16U GuiLib_UnicodeStrLen(
 GuiConst_TEXT *S);`

Input: Unicode string reference.

Output: Length in characters, excluding zero termination.

Related functions: `GuiLib_StrAnsiToUnicode`
`GuiLib_UnicodeStrCmp`
`GuiLib_UnicodeStrCpy`

GuiLib_VLine

Purpose: Draws a vertical line.

Full declaration: `void GuiLib_VLine(
 GuiConst_INT16S X,
 GuiConst_INT16S Y1,
 GuiConst_INT16S Y2,
 GuiConst_INTCOLOR Color);`

Input: Coordinates.
 Color.

Output: None.

Related functions: `GuiLib_Line`
 `GuiLib_HLine`

GUIDISPLAY UNIT

This unit must be edited to suit the target system display controller, as described previously.

The following functions are available (in alphabetical order):

Functions

GuiDisplay_Init

Purpose: Initializes the display. Should never be called directly, because `GuiLib_Init` calls it as part of the easyGUI initialization process. However, `GuiLib_Init` must be called at some time during system startup.

Full declaration: `void GuiDisplay_Init(void);`

Input: None.

Output: None.

Related functions: `GuiDisplay_Refresh`

GuiDisplay_Lock

Purpose: Prevents the operating system from making task shifts. The contents of this function must be created by the user, because it is OS dependent.

Full declaration: `void GuiDisplay_Lock(void);`

Input: None.

Output: None.

Related functions: `GuiDisplay_Unlock`

GuiDisplay_Refresh

Purpose: Refreshes display controller RAM, based on the internal easyGUI display buffer and refresh flags. Should never be called directly, because `GuiLib_Refresh` calls it as part of the easyGUI refresh process.

Full declaration: `void GuiDisplay_Refresh(void);`

Input: None.

Output: None.

Related functions: `GuiDisplay_Init`

GuiDisplay_Unlock

Purpose: Allows normal operating system task shifting again. The contents of this function must be created by the user, because it is OS dependent.

Full declaration: `void GuiDisplay_Unlock(void);`

Input: None.

Output: None.

Related functions: `GuiDisplay_Lock`

17 easyTRANSLATE

easyTRANSLATE is a utility to aid in translating easyGUI structure texts. The utility is not strictly needed, because the complete translation job can be done inside easyGUI, in the Language window. However, if external translation is desired, easyTRANSLATE comes handy, because it:

- Hides the complexity of the easyGUI development environment from the Translator, which possibly is a non-technical person.
- Prevents the Translator from changing anything else than the texts of a single language.
- Avoids license key problems when using another PC, possibly at a remote location, for the translation work.

It can be purchased separately from the [easyGUI](#) web page.

INSTALLATION

Run the easyTRANSLATE installation program, following the instructions on screen.

Several special fonts are required:

- Arial. Should be present in a standard Windows installation.
- Arial Narrow. Is part of e.g. Microsoft Office.
- Arial Unicode MS.

The installation program installs these fonts, if needed. The fonts can also be found in the `Fonts` sub folder of the easyTRANSLATE install folder.

The easyTRANSLATE utility shows a warning message if one or more fonts are missing.

PRINCIPLES

easyTRANSLATE functions by reading a special data file (*.egt) produced by easyGUI, containing all fonts, structures, variables, etc., in short, a complete copy of your project, with the master language texts, and the working language texts to be translated.

But why the complete project, why not just the texts? This is to enable easyTRANSLATE to show not only the texts, but also the complete structures containing these texts, just

like in the Language window in easyGUI. This is a huge advantage when translating, as it enables the Translator to see the *context* of the texts, not just the bare texts. The Translator can thus make correct translations of pieces of text, and judge if the translated texts take up too much space. Remember that texts in easyGUI are proportionally spaced (unless otherwise instructed), and it is therefore not possible to set specific maximum number of characters for the texts to ensure that they keep inside the allotted limits. This method of visual feedback to the Translator has been proven in practice to produce translations with a very low number of errors. The only condition that must be met is that the Translator must initially learn to use this visual way of translating.

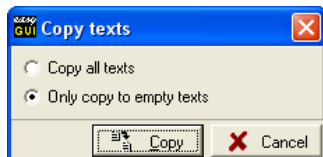
When the Translator has finished the work the data file is returned to the developer, and read back into easyGUI.

The Translator can only edit the working language texts, not the master language texts or anything else.

HOW TO USE

The procedure for using easyTRANSLATE is as follows (it is assumed that easyGUI and easyTRANSLATE has been correctly installed):

- 1 Make sure that all texts in the Structure window are marked/unmarked correctly for translation. The "Highlight translation" option in the lower left corner of the Structure window comes handy for this task.
- 2 In easyGUI, go to the Language window.
- 3 Select the master language in the left text column (usually the first language).
- 4 Select the language to be translated (called the working language) in the right text column.
- 5 Copy all non-translated texts from the master language to the working language, by pressing the **COPY TEXTS FROM LEFT TO RIGHT COLUMN** button, and selecting the "Only copy to empty texts" option:



This ensures that all texts to be translated are initially filled with a copy of the master language text, in many instances making it easier for the Translator to edit the text, because some texts will usually be quite alike in different languages.

- 6 Create a *.egt data file for easyTRANSLATE by pressing the **EXPORT TO FILE** button, and selecting a suitable destination filename and folder. The filename is as default the language name.

- 7 Send the *.egt file to the Translator. The data file is compressed, so that it is easier to e.g. mail.
- 8 Do NOT make major edits to the structures while the texts are exported to the Translator. It is not an error to edit while text are "out of town", but it can obviously lead to some problems when importing texts back in.
- 9 The Translator uses easyTRANSLATE to edit texts. Texts are automatically saved when closing easyTRANSLATE.
- 10 The *.egt file is returned from the Translator.
- 11 Read back the *.egt file into easyGUI by pressing the **IMPORT FROM FILE** button. easyGUI automatically selects the correct language before importing starts. Texts that were changed externally in easyTRANSLATE are marked with a little red E to the left, until the imported data is saved.
- 12 Select the translated language as the current language.
- 13 Check all structures in the Structure window for correct appearance. Things to look for are texts that take up too much space, texts that overflow eventual background boxes, misunderstandings by the Translator, texts that was not marked for translation, or shouldn't have been marked for translation, etc.
- 14 Create C files for the target system, and check for correct function.

18 easyGUI PC SIMULATION TOOLSET

The easyGUI PC simulation toolset is a special display driver that allows your target system code to run on a PC under the Windows environment. It can be purchased separately from the [easyGUI](#) web page.

PURPOSE

There are several interesting uses for the PC simulator:

- Demonstration software. Can show the intentions of the user interface for e.g. sales staff, or potential customers, with the added bonus that the screen presentation can be made to look like the target system, or part of it.
- Experimental parts of the user interface. It is generally much easier and faster to develop purely on the PC, than to download code into the target system, in order to test user interface specific items.
- Development of the user interface before the actual target system hardware becomes available.



Demonstrating the user interface as a Windows application developed with the PC simulator is far superior compared to using the easyGUI structure editor. This is especially true if the persons receiving the demonstration are not technically skilled.

NECESSARY FILES



In order for the PC simulator to work the following items is necessary:

- A PC based compiler. The simulator is delivered in three different versions, suitable for:
 - Borland C++ Builder 5 (or higher) C compiler for Windows.
 - Microsoft Visual Studio 2003 (or higher) C compiler for Windows.
 - DEV C++ 4.9.9.2 (or higher) GNU compiler for Windows. This product is free, if used under the license rules of the GNU General Public License. Please look at www.bloodshed.com for further information.



The final executable will look almost 100% the same, no matter which compiler is used. If other compiler must be used some work must be anticipated on the visual components.

- The easyGUI PC simulation toolset. For the Borland C++ Builder version the important files are:
 - `Main.cpp` and `Main.h`. The main Windows application source. As delivered `Main.cpp` does nothing more than contain a simple visualization of the target system display, and show a simple call to the `GuiLib` library.
 - `GuiLib.cpp` and `GuiLib.h` library. Observe that the `GuiLib.c` library file has been renamed to `GuiLib.cpp`, in order to be recognized by Borland C++ Builder. The content of the file is the same.
 - `GuiDisplay.cpp` and `GuiDisplay.h` display control unit. These files are radically different from the normal `GuiDisplay.c` and `GuiDisplay.h` display control files of the easyGUI library. This is the Windows display driver, which is the core of the PC simulation toolset. It uses the data from the `GuiLib` library in exactly the same way as the normal target system, i.e. `GuiLib` doesn't "know" that it is working in a Windows environment.
 - `GuiGraph1H.c` and `GuiGraph1V.c` include libraries. These files are identical to the target system library files.
 -  Color and  Unicode versions: `GuiGraph2H.c`, `GuiGraph2V.c`, `GuiGraph2H2P.c`, `GuiGraph2V2P.c`, `GuiGraph4H.c`, `GuiGraph4V.c`, `GuiGraph5.c`, `GuiGraph8.c`, `GuiGraph16.c` and `GuiGraph24.c` include libraries. These files are identical to the target system library files.

For the Microsoft Visual Studio version the important files are:

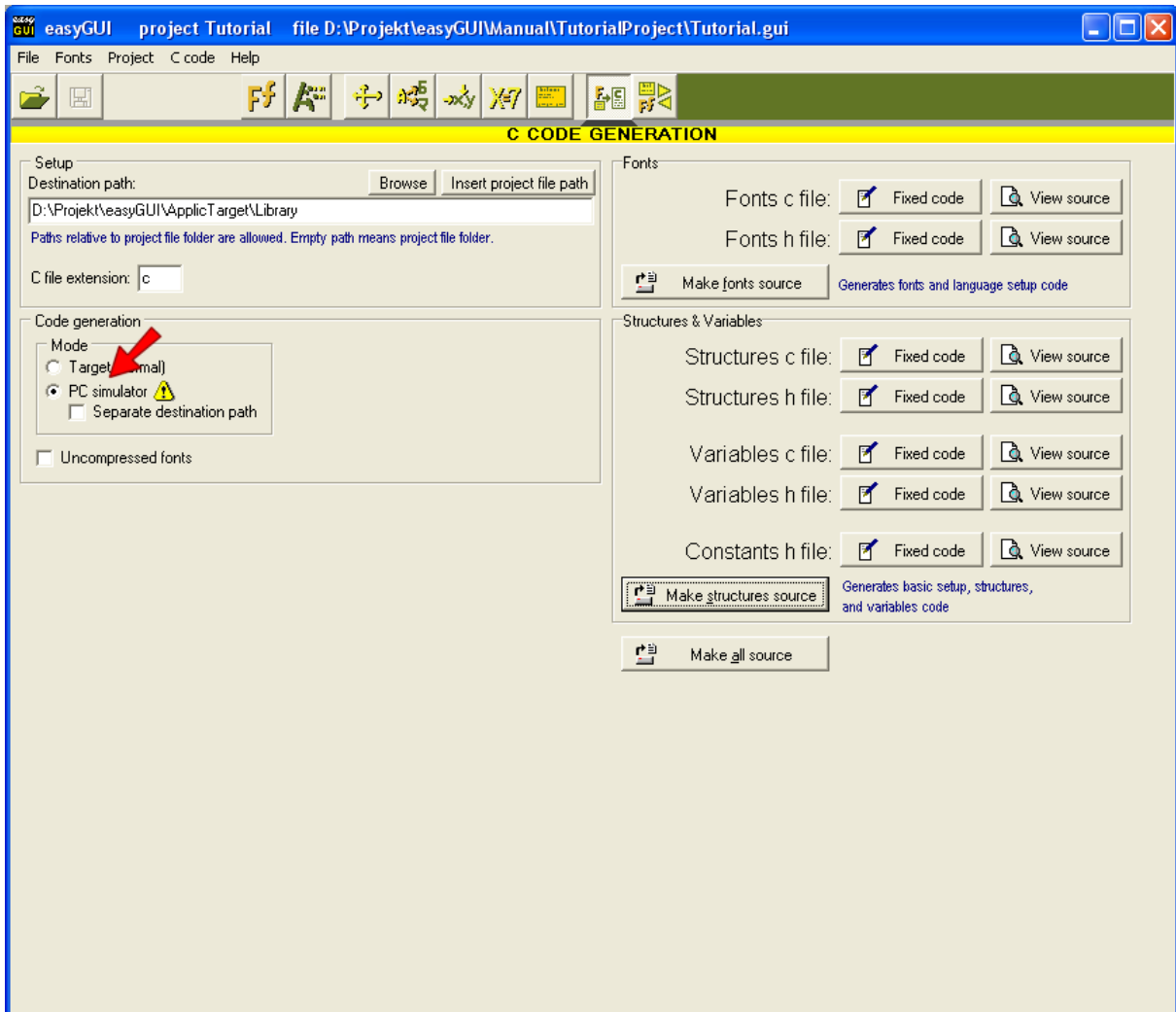
- `WinSimulator.cpp`. The main Windows application source. As delivered `WinSimulator.cpp` does nothing more than contain a simple visualization of the target system display, and show a simple call to the `GuiLib` library.
- `GuiLib.c` and `GuiLib.h` library. These files are identical to the target system library files.
- `GuiDisplay.c` and `GuiDisplay.h` display control unit. These files are radically different from the normal `GuiDisplay.c` and `GuiDisplay.h` display control files of the easyGUI library. This is the Windows display driver, which is the core of the PC simulation toolset. It uses the data from the `GuiLib` library in exactly the same way as the normal target system, i.e. `GuiLib` doesn't "know" that it is working in a Windows environment.
- `GuiGraph1H.c` and `GuiGraph1V.c` include libraries. These files are identical to the target system library files.
-  Color and  Unicode versions: `GuiGraph2H.c`, `GuiGraph2V.c`, `GuiGraph2H2P.c`, `GuiGraph2V2P.c`, `GuiGraph4H.c`, `GuiGraph4V.c`, `GuiGraph5.c`, `GuiGraph8.c`, `GuiGraph16.c` and `GuiGraph24.c` include libraries. These files are identical to the target system library files.

For the DEV C++ version the important files are:

- `GNUSimulator.cpp`. The main Windows application source. As delivered `GNUSimulator.cpp` does nothing more than contain a simple visualization of the target system display, and show a simple call to the `GuiLib` library.
- `GuiLib.c` and `GuiLib.h` library. These files are identical to the target system library files.
- `GuiDisplay.c` and `GuiDisplay.h` display control unit. These files are radically different from the normal `GuiDisplay.c` and `GuiDisplay.h` display control files of the easyGUI library. This is the Windows display driver, which is the core of the PC simulation toolset. It uses the data from the `GuiLib` library in exactly the same way as the normal target system, i.e. `GuiLib` doesn't "know" that it is working in a Windows environment.
- `GuiGraph1H.c` and `GuiGraph1V.c` include libraries. These files are identical to the target system library files.
-  Color and  Unicode versions: `GuiGraph2H.c`, `GuiGraph2V.c`, `GuiGraph2H2P.c`, `GuiGraph2V2P.c`, `GuiGraph4H.c`, `GuiGraph4V.c`, `GuiGraph5.c`, `GuiGraph8.c`, `GuiGraph16.c` and `GuiGraph24.c` include libraries. These files are identical to the target system library files.
- And finally, of course the normal `GuiConst`, `GuiFont`, `GuiStruct`, and `GuiVar` files generated by easyGUI.

COMPILATION

Because the PC environment is a 32 bit system easyGUI must be set accordingly, when generating C code. To make things easier easyGUI can remember the target system settings, and still generate C code for the PC simulator, by using the special PC simulator setting in the C code generation window:



This setting overrides some of the compiler setup settings, in order to easily produce code suited for PC usage. To make sure C code generating is not left in this setting when intending to generate C code for the target system a small warning (⚠) is shown.

LIMITATIONS

Simple target system applications can usually be run on the PC environment without major changes. However, hardware specific operations are of course not possible on the PC. If the goal is to run the complete target system on the PC for demonstration and/or simulation purposes it will therefore be necessary to mask out, or simulate, the action of hardware in the normal target system. This is most conveniently accomplished by the use of compiler directives. A well designed target system application, with simulation of hardware specific routines, can be a great asset when debugging and testing the target system.

19 easyGUI VERSIONS

Below is a listing of changes and version numbers for the easyGUI PC system.

v4.03

Released March 9th 2004

- First commercial easyGUI version. Previous versions were customer specified.

v4.09

Released June 1st 2004

- Parameters: Changing pointer size didn't turn "Edited" flag on.

v4.10

Released June 3rd 2004

- Parameters: Upside-down display supported.
- Structures: Reloading project and selecting Structures windows sometimes caused Access violation error.
- Fonts: Reloading project and switching from and to easyGUI application sometimes caused Access violation error.

v4.11

Released June 10th 2004

- C-code: GuiConst.h file header and footer text were not saved.

v4.12

Released June 15th 2004

- C-code: Projects without need for variable/structure pointers didn't create pointer array, and compilation of target therefore failed.

v4.13

Released June 20th 2004

- Parameters: Compiler prefix editing added.

v4.13a

Released June 23th 2004

- Library changes.

v4.13b

Released August 8th 2004

- C-code: Very complex structures with >100 items caused application error due to internal static size buffers becoming too small.

v4.13c

Released October 27th 2004

- Some smaller errors corrected.

v5.0.0

Released December 9th 2004

- Color version released.
- Structures: Bitmap item type implemented.

v5.0.1

Released December 13th 2004

- Corrections of minor errors.

v5.0.2

Released December 16th 2004

- InterBase is no longer needed. Instead the Firebird dll file is used.

v5.0.3

Released January 17th 2005

- C code generation: Setting for producing special PC simulator code implemented.

v5.0.4

Released January 19th 2005

- Parameters: Support for AVR flash RAM operation added.

- C code generation: Special PC simulator flag added.
- Library: GuiLib_DrawStr formatted string drawing function added.
- Library: Fixed floating point decimal character error.

v5.0.5

Released February 19th 2005

- Auto redraw mode function added. Possible settings are:
 - Continuous updating - functions like before.
 - Update on changes - Auto redraw is only performed if the variable in question has changed.
- Structure editor, bitmap selection: Fixed error that prevented Browse button from being used, when file name edit box was empty.

v5.0.6

Released March 18th 2005

- Structures: Paragraph item type implemented.

v5.0.7

Released March 22nd 2005

- Paragraph item: Various errors corrected.

v5.1.0

Released March 31st 2005

- Parameters: Support for KEIL reentrant keyword added to the compiler setup.
- easyTRANSLATE released.

v5.1.1

Released March 31st 2005

- Library: Error in structure data decoding when handling Auto redraw flags corrected.
- Structures: Variable types was not displayed correctly.

v5.1.3

Released April 21st 2005

- Library: Error in auto redrawing items could lead to memory access violations.

v5.1.4

Released May 4th 2005

- Library: GuiLib_Clear now resets cursor, scroll, and auto redraw flags.
- Library: Scrolling is now possible without showing a scroll bar (highlighted line) by specifying line -1 in the GuiLib_SetScrollPars function call.
- Library: GuiLib_CurStructureIndex variable remembers last displayed structure.
- Variables: Strings with more than 255 characters could not be specified or entered.

v5.1.5

Released May 28th 2005

- Help function: PDF Active-X component used didn't function with Acrobat Reader 7. PDF file is now called directly as an application.
- File | Close: Hotkey changed from ctrl+C to ctrl+F4, because ctrl+C is used by the clipboard function.
- GuiFont: Font identifiers was indexed wrongly. Indices was:


```

      #define  GuiFont_Text1           0
      #define  GuiFont_Text2         1
      #define  GuiFont_Text2Bold     2
      #define  GuiFont_Text4         3
      #define  GuiFont_Icon1         4
      #define  GuiFont_DEFAULT_TEXT_FONT 0
      
```

- should be:

```

      #define  GuiFont_Text1           1
      #define  GuiFont_Text2         2
      #define  GuiFont_Text2Bold     3
      #define  GuiFont_Text4         4
      #define  GuiFont_Icon1         5
      #define  GuiFont_DEFAULT_TEXT_FONT 0
      
```
- Manual: GuiLib_CurStructureNdx variable misspelled as GuiLib_CurStructureIndex.

v5.2.1

Released February 2nd 2006

- Unicode support added.
- Pointer size: A special `void *const` pointer setup setting added, to be used with H8S μ -processor and the IAR compiler.
- Library: `GuiLib_DrawChar` function in library didn't properly support the `GuiConst_AVR_COMPILER_FLASH_RAM` compiler directive, which is activated by setting the "AVR compiler flash RAM operation" option in the Parameter window, Compiler tab page.
- Library: Clipping didn't always work correctly with upside-down display and negative Y coordinates.
- Structures: Background box Y size parameters implemented, so that background box height can be set to any size.
- Font selection window (F6) removed, functionality moved to font editor.
- Font names streamlined:

<u>Old name</u>	<u>New name</u>
Text1	ANSI 2 condensed
Text2	ANSI 2
Text2 bold	ANSI 2 bold
Text3	ANSI 3
Text4	ANSI 4
Text5	ANSI 5
Text6	ANSI 6
Text7	ANSI 7
Text8	ANSI 8
Icon1	Icon 16x16
Icon2	Icon 32x32
Icon3	Icon 48x48
Icon4	Icon 72x72
Icon5	Icon 202x50
- Unicode fonts added to Unicode version.

v5.2.1a

Released February 14th 2006

- Language window, language setup: Editing, deleting and moving languages were not handled properly.

- Library: GuiLib_GetTextWidth function declaration in GuiLib.h was in error.
- Structures: Text displayed in complex structures sometimes used the wrong font for individual characters.

v5.2.2

Released February 21st 2006

- Library: Cursor fields was not handled properly in Unicode mode. Initial display was ok, but any cursor movement gave garbled cursor fields.

v5.2.2d

Released April 3rd 2006

- Touch interface test version.

v5.2.3

Released April 18th 2006

- Dongle support implemented.
- Windows TTF font import implemented.
- Import/export function implemented.
- Touch screen interface implemented.
- Library: SetClipping function improved.
- Library: GuiLib_Box didn't handle clipping correctly.
- Library: New Unicode support functions for comparing Unicode strings, and for copying from one Unicode string to another.

v5.3.0

Released March 5th 2007

- Fonts: Cyrillic characters added to Unicode fonts.
- Font editor: PS numerical width value for fonts added.
- Font editor: Selection of a range of characters now possible in the character set panel.
- Font editor: Check white space control of font characters added.
- Font editor: Enhanced ttf font import.

- Font editing: New selection criteria for font character selection implemented: "All numerical characters". This setting includes the following characters into GuiFont.c, no matter which other selections are active: Space, numbers 0~9, letters A~F, letter h, and the characters period, comma, minus, plus, and colon.
- Parameters, Display: Active area of display feature added.
- Parameters, Display controller: 1 or 2 color planes mode selection added.
- Parameters, Display controller: Mirroring feature added, both horizontally and vertically.
- Parameters, Color: 5bpp gray scale mode added. ST7529 display controller driver added, using 5bpp gray scale.
- Parameters, Color: 18bpp color mode added.
- Parameters, Color: Now more than one palette for each color depth.
- Parameters, Compiler: Font pointer lists prefix field implemented. This field is normally empty, but by setting it to "const" when using Keil compilers the placement of the font pointer list in RAM instead of flash memory can be prevented.
- Parameters, Operation: Cursor and scroll wrap around feature added.
- Variables: Advanced variable definition import function.
- Structures, background boxes, didn't work properly when using inverted colors, i.e. white text on black background.
- Structures: Active area item type added.
- Structures: Better support for Color, B&W and monochrome copy of structure image.
- Structures: C-button for structure name.
- C code generation: Special compiler settings were also enforced on generated c/h files, when generating for PC simulator, causing code to be impossible to compile.
- C code generation: Better font compression.
- GuiFont.c/h in AVR compiler mode: "__flash" prefix was duplicated in GuiLib_FontRecPtr declaration, resulting in compiler warning.
- Library: More Unicode string functions added.
- Library: Color convert functions added.
- GuiDisplay: Hitachi HD61202 driver, GuiDisplay_Init function - C variable used in loop construct clashed with some compilers using "C" symbol for internal register.

- S6B0741 display controller driver added, using 2 bits per pixel gray scale mode, and two color planes.

v5.3.0b

Released April 17th 2007

- Parameters, Operation: In easyGUI Monochrome and Color the Cursor box partly hid the Scroll box.

v5.3.0c

Released June 25th 2007

- Library: Errors with clipping contra relative coordinate origo resolved.

v5.3.0f

Released August 24th 2007

- Library: In some instances dynamic updating of structure elements (cursor fields, auto redraw fields, etc.) didn't happen.



- end of document -