

**Programmers manual for our  
2.1" color TFT with 65.536 colors  
from  
[www.display3000.com](http://www.display3000.com)**

**V 1.50  
23. January 2009**



## Table of contents:

Table of contents: .....	2
Overview.....	4
Schematics of the connection of the display to the microcontroller.....	5
Note to our delivered software library / utilities / other files.....	7
The code (Bascom Basic):.....	8
The code (WinAVR C): .....	11
Driving the display – general comment (C and Basic): .....	12
Your new commands for the display (Bascom Basic) .....	13
Your new commands for the display (WinAVR C) .....	15
The data output to the display.....	18
The output window .....	19
The output of characters .....	21
Characters and numbers with fixed width.....	21
<i>Font 1 (5x8)</i> .....	21
<i>Font 2 (8x14)</i> .....	22
Characters and numbers with variable width (proportional font).....	23
Special case: extended characters (like umlauts and other foreign characters).....	24
The output of colored graphics.....	26
Minimal goal: setting one single pixel: .....	27
Introduction to the color system of the display.....	28
<i>65,536 (65K) colors</i> .....	30
<i>Alternative I: 256 color mode (RGB format 3-3-2)</i> .....	31
<i>Alternative II : xxx out of 65,536 colours (colour table)</i> .....	34
<i>Needed software adaptation for the native 256-color-mode (8 bits per pixel)</i> .....	36
<u>Bitmap graphics</u> .....	37
<i>Compression / decompression of graphics files</i> .....	38
Creation and output of graphical elements / photos.....	42
The GLCD_Convert program.....	43
<i>Overview – how to get bitmaps into the microcontroller?</i> .....	48
<i>Some hints from personal experience</i> .....	49
Changing the output direction (rotating the display) .....	50
Conversion of the software to other systems or other programming languages.....	52
Reference of driving the 2.1“ Display.....	53

The serial interface of the graphics controller.....	53
8 Bits-Interface .....	53
16 Bits-Interface .....	53
Command / Parameter.....	53
Logic analyzer screens .....	54
(Zoom):.....	55
Timing of the signals / clock line vs. data line .....	57
The display commands: .....	58
Initializing (65.536-color-mode – 16 bits per pixel .....	58
Alternate initializing (256-color-mode – 8 bits per pixel .....	59
Switch off sequence.....	60
Define output window (where to write pixels).....	61
Switch the display to white or black screen (without losing the screen content) .....	61
Set vertical offset / scrolling.....	62
Coordinates and output direction .....	63
Possible problems and their solutions:.....	65
Contact:.....	70

Congratulations on the acquisition of our 2.1" display module kits.

You will recognize, that you will shortly never think of the idea working without a color display. The projects realized by you will get automatically a professional touch now.

## Overview

The connection between the microcontroller and the graphics TFT is done serial – this means the needed data is moved to the TFT bit by bit.

Unfortunately this display is pretty „dumb“ and does not offer too much support for our needs. For this reason, we need to do most activities by our software. The complete pixel data of the display content needs to be sent by our software. No matter if a string is been written, an icon needs to be displayed or a line needs to be drawn: Every pixel needs to be calculated by the software and needs to be sent to the display.

Luckily the display knows a command to reduce the pixel output to a specified area (think of opening a window) and it allows selecting an output direction. This is very helpful as it reduces the number of pixels we need to transfer to change a character with a size of 5x8 to 40. We just open a virtual window of a size of 5x8 and then we send 40 x pixeldata (which is basically the individual pixel color). Due to the extra display memory, we do not have to take care of refreshing the whole display 60 times a second. If we would have to do this – the microcontroller would definitely not be able to do anything else. Now the microcontroller only needs to do anything if we change something on the screen.

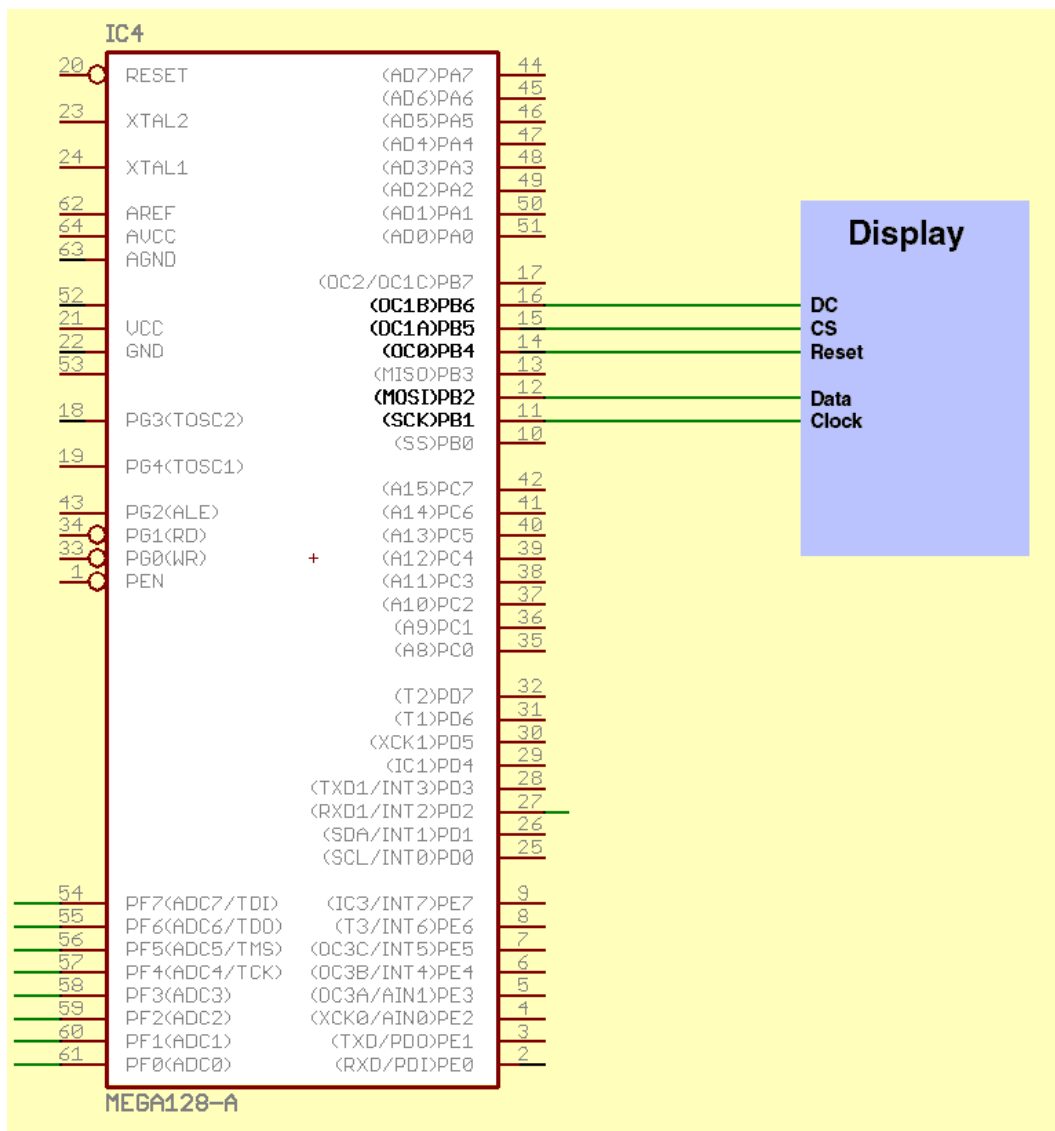
As mentioned above, the data is been sent serial to the display. To save our resources and to speed up this serial transmission we are using the hardware SPI module, buildt into the ATMega. For this, we connected the display with the SPI port of the microcontroller - important is only Data and Clock, however the display needs some more lines which our software needs to toggle manually if needed. These other lines are Select (CS), Reset, Data/Command (DC)

## Schematics of the connection of the display to the microcontroller

The connection of the display happens through 5 wires:

- CS (Select)
- Clock (Clock signal)
- Data (the data bits)
- DC (Data / Command identification)
- Reset (Full Reset)

At the following picture this connection is shown. Please note: on our module, we do not connect the display directly to the microcontroller – we are using a buffer chip between display and controller.



A data sheet of the unknown display controller is not available. Why not? This display is being used only in OEM consumer electronics devices. You only are able to buy this display if you order at least 500,000 of these. Only then, you will get any detailed information – and

only on a 1:1 basis. The manufacturer of this display has no interest in selling this display in smaller numbers to anybody. If you need only ten or ten thousand of these displays, they want you to buy their “regular” display which they sell through the distribution / reseller for a higher price (the distribution display is of course using a totally different graphics controller). Therefore nobody is interested in helping with this display as everybody is losing money (beside you of course) if you buy this display instead of the expensive official display.

**How comes Display3000 know how to drive the display?**

We did reverse engineering and we peeked out the data communication between the original hardware and the display with our logic analyzer. Then, we were able to develop our software. More on the logic analyzer signals at the end of this manual starting from page 54.

**If you ordered a module with integrated microcontroller, you may just add voltage to the module (see separately hardware manual to your board). Each module has been pre-programmed for testing – you will see a display output right away.**

## Note to our delivered software library / utilities / other files

On your CD, you will find several data sheets of the Atmel microcontroller, sample software for driving the display (in C and Basic) and several manuals for our hard- and software.

There are also two compiler placed on our CD:

- WinAVR: a pretty good Open Source-C-Compiler
- A limited version of Bascom, a commercial Basic-Compiler

Regarding Bascom: The delivered Basic-Software has been created for the Atmega in Bascom. Bascom is a very good Basic compiler which is available usually for around 83 Euro (approx. 108 US\$). On our CD you will find the latest demo version. Actually it is a full version including all sample files, help files, IDE, simulator etc. but it is limited to 4 Kbyte compiled code. This is fine for a first test, but of course is not enough for most “real” projects. We are offering a special hardware bundle deal which only is valid up to 1 week after you got your display module: full licensed version Bascom (includes all future updates for free) for only 65 Euro (approx. 84 US\$) you may order our hardware bundle version – this gives you 25 US\$ discount. This is only valid up to 2 week after receiving your goods and only if you originally purchased a display module costing 63 Euro (or 82 US\$) or more.

**No shipping costs then. If you are interested in Bascom: this deal saves real money!**

Both the Basic and C-files can be opened with any usual program editor (Bascom and WinAVR are also bringing their own editor).

### Basic or C?

Both does have the right to live. Even professionals are using Basic if they need a running solution quickly. Basically the following applies:

**Basic:** quick learning curve, quick development, quick success – but usually limited to the available libraries (which are in case of Bascom really a lot!)

**C:** more difficult to learn, much closer to the hardware and much more flexible than Basic, usually needed to development for a customer. Generally approx. 30% faster then Basic.

**Note: As we heavily commented the software code, we will not go into many code-details in this manual. In this manual, we explain the general idea behind our code – if you then look at the code, it will be easy to understand (hopefully).**

## The code (Bascom Basic):

**Note: If you want to program in C, just move forward 2 pages.**

The file *GLCD21\_Display3000.bas* does contain all needed code for driving the display. If you open this file, you will recognize the structure of the code:

- 1) Definition of the variables
- 2) Definition of the used subroutines which then may be called through your program by `Call Routine (parameter)`. FYI: The usage of the option *byval* is important to allow you entering direct parameters like `LCD_Print („Hello“, 10, 10...)` (you enter text and position it directly). Without this option *byval* you would need to fill variables first with the needed content. If your software later has the need of the content/parameter available in variables, you might get rid of the *byval* - option. This would save you a lot of valuable memory with each Call-Statement (check Bascom help on further details).
- 3) Definition of the font-arrays; Here, all details about the used font is saved. Our subroutine `LCD_Print` uses these parameters to learn about the size and needed space between the characters. If you want to add more fonts, you need to add the used data about the fonts.
- 4) Definition of the used ports of the ATmega. Here you define which port is needed as an output port and which port as an input port. In our example, we define most ports as an Input - port with switched on Pull-Up-Resistor. This need to be adapted to the needs of your project. Check the ATmega datasheet for this.
- 5) Definition of the display connection. We did place all display ports here in constants. If you later want to change the used ports for the display connection, you just need to change these entries. Advantage: instead of already remembering that Port B.4. is being used as Reset, you just use `LCD_Port.LCD_Reset`.
- 6) Predefined colors as constants are making your life easier. You do not have to remember all the needed data for the colors. Just define any color you like and add here.
- 7) Predefined constants for color mode and orientation
- 8) Hardware-SPI of the microcontroller needs to be configured at the beginning
- 9) Initializing of the display needs to be done once – usually at the beginning of the code

**As the code comes with a lot of explaining comments, we will not go into detail with this manual. If there are questions: just mail us.**

Our code is well documented and self-explaining. Play around to learn all functions. You will recognize: running this display is easy.

If you ordered a module including a Atmel microcontroller from us, the controller has been programmed from us for testing and demonstration. Due to lot of requests in the past, we added this default program to our CD. It is called `portcheck.bas` and is located in the directory of the Bascom Basic program files.



### **Integrating the new display commands in your Bascom Basic programs.**

Your main program will quickly become crowded if all the needed subroutines would be embedded in your program file all the time. To avoid this, we recommend the usage of the command `$Include` which will include the needed subroutines only during the compilation process. Instead hundreds of program lines you only see those `$Include-Commands`. As of some specialities of Basic (e.g. Data lines are only allowed at the end of the program), you need to embed 3 different files by using `$Include`. If these files are not located in the same directory with your main program file, you need to add the path to them.

At the beginning of the program, behind the definition of the variables but before the first call of a display routine:

```
$include Init21_display3000.bas
```

This initializes the display, defines all variables needed by our subroutines, defines colors, etc.

At the end of the program, but before any DATA-lines and also before any graphic files which will be embedded using the `$Include` command:

```
$include Glcd21_display3000.bas
```

This file contains all routines for driving the display. If you do not need specific commands you may delete them to save memory.

At the very end of the program (e.g. last line):

```
$include Glcd21_fonts.bas
```

Here all the font data and the display initialisation codes are embedded.

**Check the sample program Portcheck.Bas.** This uses some commands and shows how to include the display files.

Using these `$Include` technique offers you another advantage. You will now be able to present your main code on the internet or to send it to somebody else. You will not violate our copyright if you skip the include files. Anybody who purchased a display from us, owns his own version of our files

**Init21\_display3000.bas**

**Glcd21\_display3000.bas**

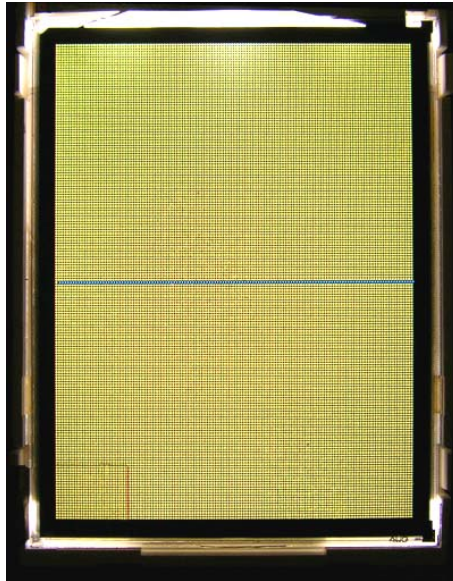
**Glcd21\_fonts.bas**

on his/her CD and will be able to compile your code and run it on their display. To avoid any misinterpretation, we say it again: You are not allowed to send our display library code to anybody or to upload it to the internet etc. These are your personal licensed files and these are not freeware.

### **Your first programming in Bascom Basic**

If you want to be sure that you can do the programming of the module correctly, just start with the program `blue_line.bas` on our CD. This Code contains only a few possible problems and runs even in the demo version of Bascom. With this code you will get a quick success.

After compiling and programming the module the display should show a yellow background and a blue line. That's all:



Hint: If `blue_line.bas` works, but other programs don't, you probably encountered an usual standard problem:

You did not enter (btw: this is only valid for Bascom) the needed numbers of `HWSTACK`; `SWSTACK` and `FRAMESIZE`. See comment at the beginning of the source code. Bascom does not know about the needed stack size etc. so you need to do this manually, if your code grows, you need to raise these numbers (see also page 65).

A nice example how to use the different commands can be found in the folder `\sample with graphics\`. There, three full-size-pictures with different graphics mode (64.536 colors, 256 colors with color table, and the later with compressed data ) are showed.

**We now suggest that you go and read this manual from beginning to end quickly to get a brief overview. Then you might read again either the full manual in detail or just specific chapters as you need.**

### **The code (WinAVR C):**

The C-Code is similar set up as the Bascom-Basic code: specific subroutines are defined as a function and are being called when needed.

The following files are being used during the compile-process and need to be available (*include*):

#### glcd-Display3000-211.h

- definition of all functions
- definition of the used constants (e.g. colors)
- definition the ports for the display communication

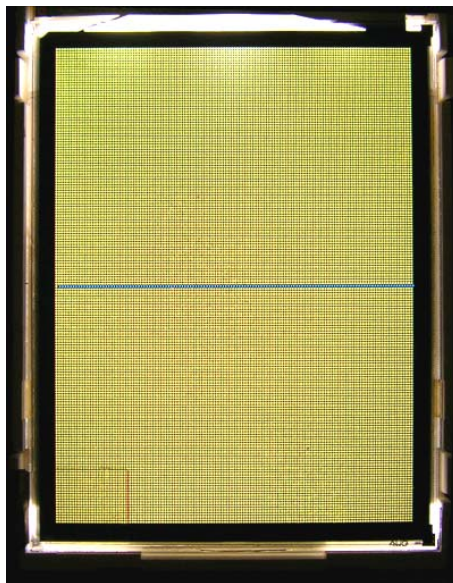
#### glcd-Display3000-211.c

- includes all subroutines for setting up the pixels for the display
- includes the SPI-output software code
- includes an alternative for the SPI-output (manual setting of data bits and clock, e.g. for controllers without any integrated hardware SPI)

### **The first programming in WinAVR**

If you want to be sure that you can do the programming of the module correctly, just start with the program `blueLine.c` on our CD. This Code contains only a few possible problems. With this code, you will get a quick success.

After compiling and programming the module the display should show a yellow background and a blue line. That's all:



**We now suggest that you go and read this manual from beginning to end quickly to get a brief overview. Then you might read again either the full manual in detail or just specific chapters as you need.**

### **Driving the display – general comment (C and Basic):**

Our sample software shows how to use the display. You will recognize that we did not use the most efficient way to create a function. We wanted that anybody, even unexperienced programmers, being able to fully understand our code. We could have made the code a bit faster and more efficient but than it would be much more difficult to understand.

But no problem: you do not need to understand the display and its driving. You can just use our subroutines without knowing exactly what is going on there. That's pretty normal: usually most programmer do not disassemble the Print-Command of a programming language because they want to know exactly what is happening there – they just want to use it.

If you only want to use our display routines, you may just use the sample program as a template and start writing your own program. If memory is becoming a problem, just throw out some code sequences you do not need (e.g. in the Basic code: just delete in LCD\_Window the code for any needed orientation directions or in the LCD\_Bitmap code these parts of bitmap conversions you do not need.

Comment to Bascom: With our existing subroutines the commands showed on the next page are available. These are being called with a preceding `Call` command. This has the advantage that you always can pass manually a needed parameter (e.g. you may enter the command `CALL LCD_Box(0,0,10,10,red)` directly. But there is a disadvantage: Bascom needs to place all parameters on the stack thus needing a lot of time and memory.

You may change the code to use two different strategies:

a) Eliminate the `byval` command at the definition – you then can only use variables for passing parameters (example above: you first need to define some variables like `Color_tmp`, `X1_tmp`, `X2_tmp` etc. – then you pass the needed parameters to these variables (`x1_Tmp=0 ... Color_tmp = Red` and then use the command

```
CALL LCD_Box(X1_tmp,Y1_tmp,X2_tmp,Y2_tmp,Color_tmp)
```

b) You define all subroutines as a “regular” subroutine, called with the command `Gosub`.

You will then have to define also all variables first, then you call a subroutine with a `Gosub LCD_Box` – then no parameter passing is possible, that's why you need to set the variables needed by the subroutine manually first. More on this at page 66.

## Your new commands for the display (Bascom Basic)

The following commands are now available for you:

**Orientation** = Portrait | Portrait180 | Landscape | Landscape180

Here you define, which reference (0,0) the following outputs will use (portrait mode or 180° turned portrait mode – the same with landscape)

**Graphics\_mode** = 65k\_uncompressed | 65k\_compressed

| 256low\_uncompressed | 256low\_compressed |  
256high\_uncompressed | 256high\_compressed |

Defines if the following bitmaps are coded in 65.536 color mode, in 256 out of 65K colors or at 256 color modes, it also defines if these are compressed or uncompressed.

**Call LCD\_CLS**

Clear screen with white background

**Call LCD\_Print (String, x, y, Font, ScaleX, ScaleY, FColor, BColor)**

Prints a string at position X and Y with the given font number. The next parameters are X-scaling, Y-scaling (e.g. 3, 2 means: triple width and double height) followed by foreground- and background-color.

Example: **Lcd\_print("Hello World", 1, 10, 2, 1, 2, Dark\_red, Yellow)** displays "Hello World" at x-y-position 1,10 with font 2, normal width and double height in dark red letters on yellow background.

Numerical variables will need some preparation for printing them on the screen with LCD\_Print. More about this and about printing variables containing numbers and not strings at page 25

**Call LCD\_Plot (x, y, Type, Color)**

Displays one single pixel at position X and Y. *Type* = 0 (or Thin) means 1 Pixel large; *Type* = 1 means 2x2 pixel wide (the second pixel is added always at left and below – this you should take in mind if you place pixel at the outer border of the display as they otherwise would be set in an unseen area .

**Call LCD\_Draw (x1, y1, x2, y2, Type, Color)**

Draws a line from X1,Y1 to X2,Y2. The direction does not matter. This algorithm works only with integer and is very quick. Parameter *Type*: see LCD\_Plot

**Call LCD\_Box (x1, y1, x2, y2, Color)**

Draws a filled box. Important: X1 and Y1 need to be the upper left coordinate; X2 and Y2 the lower right corner. Parameter *Type*: see LCD\_Plot

**Call LCD\_Rect (x1, y1, x2, y2, Dicke, Color)**

Draws an unfilled rectangular from X1,Y1 to X2,Y2. Parameter *Type*: see LCD\_Plot

**Call LCD\_Bitmap (x1, y1, x2, y2)**

Opens a “window” and fills it with any many bitmap data as the window pixel has. The bit-  
map format has to be set by the command Graphics\_Mode. These bitmap data needs to be  
converted with our graphics tool GLCD\_Convert on our CD and can be provided either as  
a bunch of Data lines or as a binary file. With “Bitmap” you turn over the array name of the  
bitmap.

X1 and Y1 need to be the upper left coordinate; X2 and Y2 are the lower right corner. If  
Compressed=1 the subroutine expects compressed pixel data.

## Your new commands for the display (WinAVR C)

The following commands are now available for you:

**Orientation** = `Portrait` | `Portrait180` | `Landscape` | `Landscape180`

Here you define, which reference (0,0) the following outputs will use (portrait mode or 180° turned portrait mode – same with landscape)

**Graphics\_mode** = `65k_uncompressed` | `65k_compressed`

| `256low_uncompressed` | `256low_compressed` |

| `256high_uncompressed` | `256high_compressed` |

Defines if the following bitmaps are coded in 65.536 color mode, in 256 out of 65K colors or at 256 color modes, it also defines if these are compressed or uncompressed.

### **LCD\_CLS (Color)**

Clear screen with white background (C: with a given color)

### **LCD\_Print (String, x, y, Font, ScaleX, ScaleY, FColor, BColor)**

Prints a string at position X and Y with the given font number. The next parameters are X-scaling, Y-scaling (e.g. 3, 2 means: triple width and double height) followed by foreground- and background-color.

Example: `Lcd_print("Hello World", 1, 10, 2, 1, 2, Dark_red, Yellow)` displays "Hello World" at x-y-position 1,10 with font 2, normal width and double height in red letters on yellow background.

Numerical variables will need some preparation for printing them on the screen with `LCD_Print`. More about this and about printing variables containing numbers and not strings at page 25.

### **LCD\_Plot (x, y, Type, Color)**

Displays one single pixel at position X and Y. `Type = 0` (or `Thin`) means 1 Pixel large; `Type = 1` means 2x2 pixel wide (the second pixel is added always left and below – this you should take in mind if you place pixel at the outer border of the display as they otherwise would be set in an unseen area).

### **LCD\_Draw (x1, y1, x2, y2, Type, Color)**

Draws a line from X1,Y1 to X2,Y2. The direction does not matter. This algorithm works only with integer and is very quick. Parameter `Type`: see `LCD_Plot`

### **LCD\_Box (x1, y1, x2, y2, Color)**

Draws a filled box. Important: X1 and Y1 need to be the upper left coordinate; X2 and Y2 the lower right corner. Parameter `Type`: see `LCD_Plot`

### **LCD\_Rect (x1, y1, x2, y2, Dicke, Color)**

Draws an unfilled rectangular from X1,Y1 to X2,Y2. Parameter `Type`: see `LCD_Plot`

### **LCD\_Circle(x1, y1, Radius, Fill, Type, Color)**

Draws a circle. Center at X1, Y1 with a given radius. If `Fill=1` the circle will become solid, with `Fill=0` only the frame is being drawn. `Type` defines the thickness of the frame (1 or 2 pixels)



### **LCD\_Bitmap\_65k(x1, y1, x2, y2, Bitmap, Compressed)**

Opens a “window” and fills it with any many bitmap data as the window pixel has. Here two byte per pixel are needed (65k colors). These bitmap data needs to be converted with our graphics tool GLCD\_Convert on our CD and can be provided either as a bunch of Data lines or as a binary file. With “Bitmap” you turn over the array name of the bitmap.

### **LCD\_Bitmap\_256low(x1, y1, x2, y2, Bitmap, Compressed)**

Open a “window” and fills it with any many bitmap data as the window pixel has. Here 1 byte per pixel is expected (need to be available in RGB332 format). These bitmap data needs to be converted with our graphics tool GLCD\_Convert on our CD and can be provided either as a bunch of Data lines or as a binary file. With “Bitmap” you turn over the array name of the bitmap.

### **LCD\_Bitmap\_256high(x1, y1, x2, y2, Bitmap, Colortable, Compressed)**

Opens a “window” and fills it with any many bitmap data as the window pixel has. Here 1 byte per pixel is expected. This mode needs a separate color table which array need to be provided at parameter “Colortable”. These bitmap data needs to be converted with our graphics tool GLCD\_Convert on our CD and can be provided either as a bunch of Data lines or as a binary file. With “Bitmap” you turn over the array name of the bitmap.

**All Bitmaps:** X1 and Y1 need to be the upper left coordinate; X2 and Y2 are the lower right corner. If Compressed=1 the subroutine expects compressed pixel data.

## **Limitations of the WinAVR C-Compiler / GCC**

**Limitation 1:** The address range of a ATmega128 or ATmega2561 is separated in 64KByte-blocks. GCC (which is being used by the WinAVR) is not able to use flash constants which are positioned above the first 64K block. Because of this WinAVR will position all constants like bitmap data, pixel data of fonts in the first 64K so the program can access these. The problem: If you have more than 64K of this kind of data, you will not have access to them. If you try to access you always get the data from the first 64K block. You will recognize this if fonts are not being written or unexpected bitmap pieces are showed.

For this reason you shall reduce the size of your graphics elements with the GLCD\_Convert tool on our CD to a total size below 64K.

Hint 1: Use the eeprom for the font data if you do not need the full eeprom. The eeprom is a perfect memory for this kind of data – just write them once and they are always available without needing the main memory.

Hint 2: Instead of using 2-Byte graphics, create an indexed graphics file (such as GIF) in a graphics program first. Try to use the same color table with all graphics so you only need to save one color table.

**Limitation 2:** The pointer for accessing these flash constants (such as bitmaps) are fixed as signed int (thus -32768 up to 32768). The negative numbers are not usable so we have 32768 possible bytes left in an array. This is the maximum of one complete data block you



can access. The problem: One full size graphics (full color) needs 132 x 176 pixel x 2 byte = 46464 byte which cannot be displayed at once! The only workaround: split the graphics in a graphics program and display both parts one after another (part 1 from position 0,0 and part 2 from 0,88).

For these above reasons it would always be advisable to use 1Byte graphics instead (e.g. with color table).

**If you know any workaround for these two limitations, we would appreciate your help.**

These two limitations are not happening (as far as we know) with commercial C-compilers like CodeVision or Keil and will also not happen with the Basic-Compiler Bascom.

**Comment for programmer who are using C as their programming language:** For an easier reading, we decided not to split our explanation about the display programming between Basic and C. As both software drivers are using a similar code base, we do use Basic as our example codes at the following pages. As the C code is very identical even a non experienced C programmer will be able to get the idea.

## The data output to the display

In principle the output of data to the display always functions in the same way:

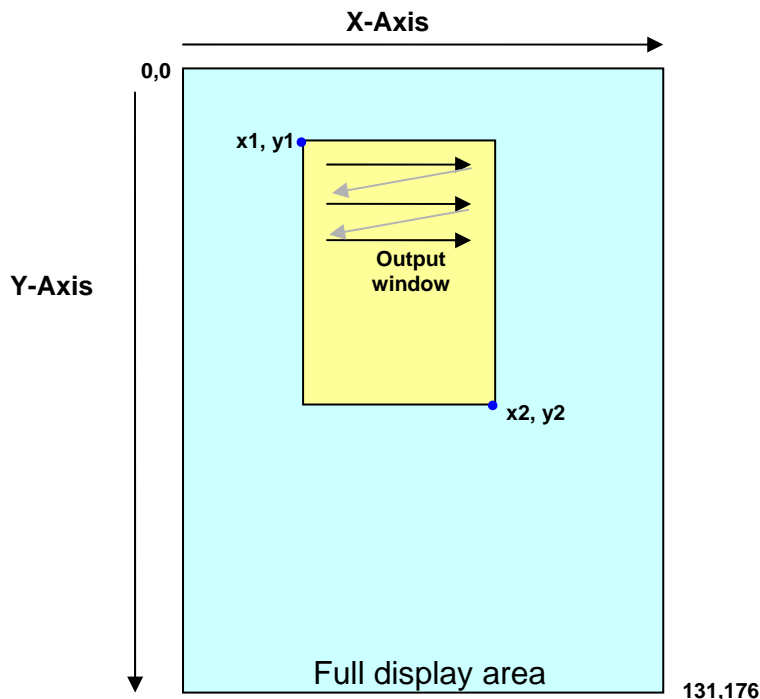
- 1) By setting the „window coordinates“ you communicate to the display, within which range of the display RAM the following data (pixels) has to be placed (fortunately you do not have to update or refresh the complete display content if just a defined area is changing – e.g. a single character).
- 2) Each sent data now is interpreted from the display as a pixel and will be set. It starts with the upper left corner of the opened “window” and moves forward one position to the right automatically. If the right border of the window is reached the pointer automatically moves to the beginning of the next line.
- 3) The wire **DC** (Data / Communication ID) of the display tells it if the received data is containing pixel data or a command. A “1” at the DC line means command, a “0” means pixel.
- 4) A pixel usually needs 2 Byte of data as each pixel can consist of one of 65,536 colors. For a full page picture the software has to transmit  $132 \times 176 \text{ pixel} \times 2 \text{ byte} = 46 \text{ KByte}$  of data or 371,712 bits.

### The output window

After determination of the output data, an output window is "opened" and the needed pixel data are sent. At the display of a character in font the output window is always exactly 6 pixels wide (the font uses 5 pixels plus one empty column as the characters usually shall not stick to each) as well as 8 pixels high. Larger fonts are appropriately higher and/or broader.

The definition of the output window happens in the sub-routine **LCD\_Window** by four values. X and Y position of the left upper corner as well as X & Y of the right lower corner. Using font I with I x scaling the lower right corner is always 6 pixels further on the right and 8 pixels down than the top left corner. The distance grows by using larger fonts or by using a scaling factor. Using scaling, the pixels are just being multiplied by the scaling factor. This saves storage because we must not save font data for the larger font but at the downside these scaled characters looks a little "rougher" through this, though. A twice large signs could of course be defined in data (array) lines in a higher resolution but this font would then need 4 times the memory of the smaller font (2 times the pixel vertically x 2 times the pixel horizontally); a 3 times larger font would use 9x the memory. For most applications the enclosed fonts should be sufficient by using our scaling routine – at these you may also always add any special character if you need.

Our routine **LCD\_Window** uses the window positions as follows:



X1 and Y1 are defining the upper left corner, X2 and Y2 the lower right corner. The arrows show the output direction of the sent bits (pixel), therefore from left to right.

To define the output area, the following command sequence of the display is used:

**Start: EF08h:** Starts a command sequence with parameters for the definition of the output window.

**Command 18h** – Output direction; (00h Portrait, 03h Portrait turned by 180°, 05h Landscape, 06h Landscape turned by 180°)

**Command 12h** – Output area start X, followed by **X-Position as parameter** (1 Byte)

**Command 15h** – Output area end X, followed by **X-Position as parameter** (1 Byte)

**Command 13h** – Output area start Y, followed by **Y-Position as parameter** (1 Byte)

**Command 16h** – Output area end Y, followed by **Y-Position as parameter** (1 Byte)

Example: To draw a yellow box, sized 10x10 pixel to position X=20 and Y=50 in Portrait mode, the following sequence is necessary:

EFh 08h

18h 00h

12h 14h    thus from X-Pos. 20 (=h14)

15h 32h    thus from Y-Pos. 50 (=h32)

13h 1Dh    thus to Y-Pos. 29 (=h1D)

16h 3Bh    thus to Y-Pos. 59 (=h3B)

With this, the output area (output window) is defined;

now 100 times „FFh E0h“ follows (= 100 yellow pixel)

**The modes for landscape or the 180°turned modes are a little more complicated and are explained more precisely at the end of the manual (chapter output directions).**

**If you use our predefined software routines, you are not really concern with this, however, further since we already carry out the necessary conversions there. No matter how you then turn the display: The position 0,0 is always at the top left of the display and X2 and Y2 is always at the lower right.**

## The output of characters

### Characters and numbers with fixed width

#### **Determine the graphical data for each character**

We would like to start with a fundamental summary of the operation of our software at the output of letters or numbers:

First the graphical data for each character are gathered from predefined tables/arrays and then transmitted with the necessary colour parameters to the display pixel by pixel.

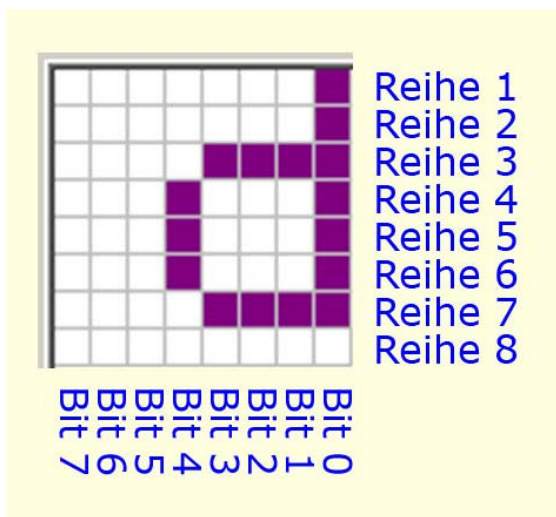
The software (sub-routine *LCD\_Print*) reads the submitted string character by character, determines the corresponding ASCII value and then selects the data from the corresponding storage position.

With the software to our modules we provide 2 fixed fonts a fine, small font (font 1) with a size of 5 x 8 pixels as well as a larger and more detailed font (font 2) with a size of 8 x 14 pixels.

For each character the data for the pixels are pre-defined in the DATA lines (Bascom: at the end of the program) and in separate files for C (Fontxxx.h – these will be automatically included during compilation).

#### **Font 1 (5x8)**

A „d“ of font 1 looks like shown at the following graphics:



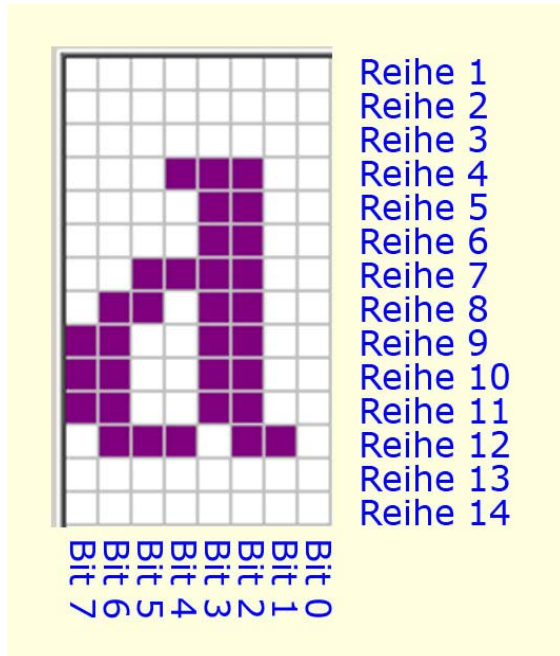
If you now calculate the bits of each columns you will get:

Row	Binäre	Decimal	Hex
1	00000001	1	01
2	00000001	1	01
3	00001111	15	0F
4	00010001	17	11
5	00010001	17	11
6	00010001	17	11
7	00001111	15	0F
8	00000000	0	00

Exactly this values you will find in the data array at the position where the “d”-data is stored.

## Font 2 (8x14)

The character “d” of font 2 is being stored using the same principle. As this font is larger, more data has to be stored. A “d” of font 2 looks like:



Row	Binäry	Decimal	Hex
1	00000000	0	00
2	00000000	0	00
3	00000000	0	00
4	00011100	28	1C
5	00001100	12	0C
6	00001100	12	0C
7	00111100	60	3C
8	01101100	108	6C
9	11001100	204	CC
10	11001100	204	CC
11	11001100	204	CC
12	01110110	118	76
13	00000000	0	00
14	00000000	0	00

Again, you will find these values stored at the array at font 2.

Using this principle, you can now create any character you want (could be foreign extended characters or icons) and add it to our existing font table.

### Fonts with a larger width:

An adaptation to a larger height is relatively easily to manage (just add more rows of data) but creating a font with a width of more than 8 pixels you need to store two bytes per pixel row – then you may create fonts up to 16 pixel width. However you need to change our display subroutine LCD\_Print as this is actually prepared to show fonts with 1 Byte rows – but changing this is not too difficult. But again: We already deliver a scaling routine so you can display any of our fonts today with any scaling you like (e.g. with triple width and 5x height).

### The storage requirements of fonts with a fixed width

Right now we need 102 (this already includes 7 predefined German characters) characters with 8 bytes each = 816 bytes for the character table of font 1 and 102 signs at 14 bytes each (1428 bytes) for the character table of font 2. We would like to mention this again: you may save the font data in the internal eeprom of you microcontroller (if any) thus saving 2KByte of valuable memory.

### Characters and numbers with variable width (proportional font)

Proportional fonts does use different width for each characters and these uses a different amount of space on the screen. Examples are shown below:

<b>Arial (Proportional)</b>	<b>Courier (fixed)</b>
iiii	iiii
wwwww	wwwww
<b>wwwww</b>	<b>wwwww</b>
oooo	oooo
eeee	eeee
mmmm	mmmm
nnnn	nnnn

Proportional spaced fonts are good for longer *regular* texts but difficult to handle if you want to show text line which should align. Usually for regular output a fixed font will be used, only if you print larger amounts of text, proportional fonts make sense.

We are planning to develop some output routines for fonts with proportional width. This is planned for the future development of this library.

### **Special case: extended characters (like umlauts and other foreign characters)**

The subroutine `Lcd_print` shows a routine with some special treatment if (in this case) German extended characters are being used. This because, these characters (äöüÄÖÜß) are placed far away in the 2<sup>nd</sup> 128 Byte of the usual ASCII-table. We would not need any special treatment if we could provide the complete font data for the 256 Byte ASCII table. The problem with microcontrollers is always the shortage of memory. Therefore, we deliver our font table only with the ASCII characters 32 (“ “ space) up to ASCII 125 (“}”). Then from ASCII 127-133 we are placing “äöüÄÖÜß” which would otherwise need to be at position 228, 246 etc. Of course, we then have to include a special treatment if these characters should be displayed. This is what the Select Case code is doing in the subroutine `LCD_Print`. If you do not need special characters, just delete this part – this speeds up this routine a bit and it saves you some memory.

If you want to create your own characters, just do the same way we did with the German characters.

If you add some characters needed by your language (e.g. Spanish, French, etc.) we would appreciate to get these array data from you. We would then include them on our CD for all the other users needing the same.

### **The selection of the font data**

With Bascom-Basic you may read such font data, stored in Data lines with the command `Lookup`. Data which could (should) be stored in the Eeprom can be read with a similar command: `ReadEeprom`.

The principle of both commands is the same: `B = Lookup(X, Font2)` reads entry # X from the Data areas starting with the label “Font2” and stores this read entry at the variable B.

Example. If the Data-line looks like this.....

Font1:

```
Data &H38 , &H44 , &H44 , &H44 , &H7F, &H44 , &H78 , &H44
```

The variable B will contain a &H78 after `B = Lookup(6, Font2)`



### The output of a string

To display a string at the display, the subroutine **LCD\_String** works through each single character of the given string. This subroutine needs some more parameters like:

- X-position of the string (start)
- Y-position of the string
- Font number
- Horizontal scaling
- Vertically scaling
- Color of font
- Background color

This subroutine then creates the needed data for pixel and sends it to the display.

### The output of numbers (or numeric variables) instead of a string

If you want to output a number instead of a string, you first must convert this number into a string.

#### Example:

```
A=123
```

```
Call LCD_Print(A, 10,10,.....)
```

...will not work as A is a numeric variable and our subroutine only works with strings. Thus you need to change the type first:

```
A=123
```

```
Tempstring = Str(A)
```

```
Call LCD_Print(Tempstring, 10,10,.....)
```

**Hint for users of Bascom-Basic:** There is a great command in Bascom called **Format**.

With this you may format the content of a numerical variable during the placing in a string variable. You then can for example always align these to the right or include leading zeros, add decimal points at needed positions etc. (very similar to what Microsoft Excel® can do in a cell). This command is very useful for writing numbers to the display.

In C this transfer from numeric to string needs to be done with *itoa()* or *sprintf()*.

#### Example:

```
A=123
```

```
itoa(a, tempstring, 5);
```

```
LCD_Print(tempstring, 10, 10, .....);
```

### The output of colored graphics

At the output of multi-coloured graphic data the storage requirements are quite large as each single pixel not only needs one bit for on/off but also the information about the corresponding color. As the display itself only knows by default a mode with 65,536 colours, i.e. every pixel covers 2 bytes of colour information. The theory of the colour modes is being discussed later (starting from page 49).

On the CD there is a file `Colorbars.bas`. This produces a row of 12 random colors. The little program is probably self explaining: One after another, boxes with a height of 12 pixels are being drawn.

As this is a 65.536 color display, each color needs two bytes (16 bit) on color information. The photo shows how it should look like.

Look at the sample program `Colorbars.bas` and play around a bit. This program uses the drawing commands:

- `LCD_Draw`
- `LCD_Box`
- `LCD_Plot`

As we deleted the font and bitmap subroutines, this demo program (now approx 3 KByte of code) also runs with the Bascom demonstration compiler on our CD.



### Minimal goal: setting one single pixel:

If you want to set one single pixel to the display you just have to set a 1x1 “window” and you then send the needed color of the pixel. A single blue pixel at X-Y-position 100,50 (Hex: 64h, 32h) will need the following sequence of data:

EF, 08, 18, 00, 12, 64, 13, 32, 15, 64, 16, 32, 00, 1F

#### Explanation:

EFh, 08h: Start of sequence

18h, 00h: Portrait-mode

12h, 64h: X1 = 100

13h, 32h: Y1 = 50

15h, 64h: X2 = 100

16h, 32h: Y2 = 50

00h, 1Fh: blue (later you will learn why this is blue)

This exact sequence will be sent if you set one blue pixel with our subroutine LCD\_Plot.

Before we now go into details about displaying bitmap graphics, you must first learn something about the color systems and data compressing. But you must not be too afraid. You will quickly learn the theory behind it.

## Introduction to the color system of the display

An output of multi color graphic data always needs memory and time of your microcontroller as each pixel not only needs 1 bit for on or off, but also still need the information about the appropriate color. This is the difference to a monochrome display.

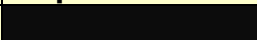









The usual graphical formats such as BMP, JPG etc. offers 16 million and more colors, this means 24 bits for each individual pixel – 8-bits for each color (red, green and blue – by mixture of these three colors every other color is represented). For each of these 3 colors there are thus 256 intensity values available.

Usual graphic programs work with CMYK (only needed for printing on paper and therefore not relevant for us) and RGB colors (RGB: **Red, Green, Blue**). There you can represent each color in 256 shades (0 to 255). From the combination of red, green and blue we receive all other visible colors. The combination of 256 x 256 x 256 colors results in an abundance of 16 million colors at the PC.

### HINT

If you want to check this: open PowerPoint or a graphics program. After drawing a box you may change the color and you may select from 3 x 256 shades each of Red Green and Blue.

### Examples of the color mixing at the PC:

Red	Green	Blue	Result	Remark	Representation
0	0	0	Black	No color portion of a color	
255	255	255	White	Each color to 100%	
255	0	0	Red	100% red, no other color	
136	0	0	Dark red	Only 50% red	
0	255	0	Green	100% green	
0	0	255	Blue	100% blue	
255	255	0	Yellow	Ever 100% red and green mixed	
136	0	136	Lilac	Ever 50% red and blue mixed	
119	119	119	Grey	For each color 47%	
255	136	0	Orange	100% red and 50% yellow	

However a screen-filling graphics for our display with 132x176 pixel (=23,232 pixel) would need 3x8=24 bit color information per pixel, that results in approx. 70 KByte program space which we rarely have available in a microcontroller. Due to this fact one already sees that we must reduce the amount of color information to reduce the memory usage.

## **Native color modes of the display: 256 or 65.536 colors**

The present display works with 256 or 65.536 colors, depending on the startup initialization routine. At the 256 color mode each pixel is represented by one byte of information (=256 possible values). At the 65.536 color mode, each pixel needs two bytes of information. Our software library is concentrating on the 65K color mode but can be quickly adapted to the 256 color mode (we will give you some instructions later in this manual).

One advantage of the 256 color mode is its doubled speed as we only need to transfer 8 bit per pixel instead of 16 bit. Unfortunately the display cannot be switched between 65K color mode and 256 color mode during its operation mode. It needs a complete new initialization routine (call of LCD\_Init).

Mixing both modes are therefore difficult but not impossible. Imaginable would be for example a startup with a nice 65K color logo, then a new initializing and a further working of the main program in 256 color mode.

One disadvantage of the 256 color mode is the restriction to 256 given colors which you cannot define (these 256 color are predefined by the 3 bit for red and green and 2 bit for blue as explained previously). However, for many applications, which are using only plain text and some simple graphics, this mode is fine. Photos or logos are usually difficult to be used in the 256 color mode.

## 65,536 (65K) colors

At the **65K-mode** each pixel needs 16 bit color information. Usually for this, a variable type with 2 Bytes of storage is being used (Word (Bacom), unsigned int (C)). These 16 bits are being splitted as follows:

- 5 Bits for red
- 6 Bits for green
- 5 Bits for blue

These 16 bits are placed in two sequential bytes

Byte # 1								Byte # 2							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
R	R	R	R	R	G	G	G	G	G	G	B	B	B	B	B

The first byte contains 5 bits for red and the 3 most significant bits (MSB) for green, the second byte the remaind 3 bits for green (LSB) and the 5 bits for blue.

A 100% red pixel would now need the color bits: 11111000 (=248 decimal or F8 hex) and 00000000 (=0); (total of the double byte value: binäry 1111100000000000 or 63,488 decimal).

green = 00000111 (7) and 11100000 (224) = 000001111100000 = 2016 decimal etc.

**Thus: 1 pixel = 2 byte color information; 1 full size picture = > 45 KByte**

The graphically demonstration: each 32 shadings for red and blue and 65 possible shadings for green. With 16 Bits you now have **32 x 64 x 32 = 65.536** color combinations at your hand.



Well, very very rarely you will ever need 65,536 color at the same time at such a small display. This would be a tremendous waste of resources. For this, we prepared two alternatives and are going to show you the theory at the following pages:

## Alternative 1: 256 color mode (RGB format 3-3-2)

For the usage with a microcontroller a 256-color-mode is recommendable. This mode is effective (low memory usage) and is often sufficient for many needs.

Well known and often used is the RGB 332 mode. With this mode usually, we will have the following

**Red and Green** will get **8 shades each** (= 3 bits of color information each)

And **blue** will get **4 shades**

3 bits for red

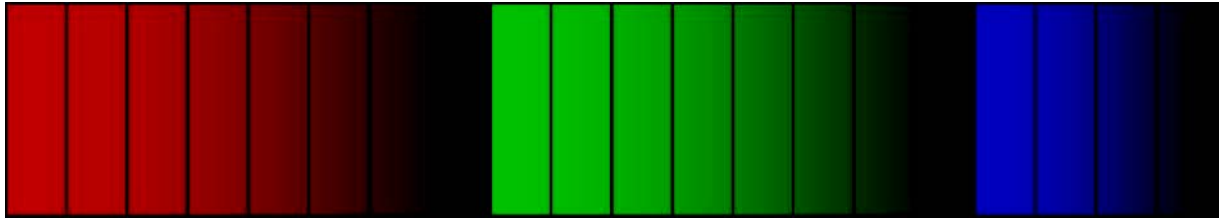
2 bits for green

2 bits for blue

Pixel-Byte							
7	6	5	4	3	2	1	0
R	R	R	G	G	G	B	B

**Thus: 1 pixel = 1 byte color information; 1 full size picture = > 22 KBytes**

The graphically demonstration: each 8 shadings for red and green and 4 possible shadings for blue. With 8 Bits you now have  $8 \times 8 \times 4 = 256$  color combinations at your hand.



This 256-color-mode (RGB332) has one big disadvantage: While it is very useful for showing simple and clear graphics (like chart, drawings etc.) it shows its limitations quickly when a display of a photo, etc... is needed (e.g. only 4 shades of blue – actually there are only 3 blue shades possible, as 0 means no blue=black).

The color range is therefore very limited which is fine for many usages, but there are also many usages where this might become a problem.

## Usage / converting of a 256-color (RGB 332) graphics

There are two possibilities to work in this mode:

Alternative 1: During initialisation the display will be switched into the 256 color mode. Then, all pixels will only need / use 8 bits (= 256 colors).

Alternative 2: The display is working in the 65K color mode. If we want to use 256 color mode (usually because it consumes only half the memory), we need to convert these 256 color values (1 byte) back to a 65k (2 bytes) color value – otherwise the display would not display the color correctly.

Our subroutine `Convert256` does this job – it converts these 256 possible RGB332-colors back to the expected double-byte color.

What we do there is calculating the correct color. These 8 possible red/green and 4 blue shadings correspond to a fixed value. We calculate the 1 byte value to a 2 byte value but we still have not more than 256 different colors.

For this, you might either use a so called lookup-table, or (this is what we are doing in our subroutine), calculate the correct number.

Bit values of Red: 3bits Green: 3bits Blue: 2bits	Red-value of this color in 5 bits-format	Red 65,536 colors value	Green- value of this color in 6 bits-format	Green 65,536 colors value	Blue-value of this color in 5 bits-format	Blue 65,536 colors value
0	0	0	0	0	0	0
1	4	8,192	9	288	10	10
2	8	16,384	18	576	20	20
3	12	24,576	27	864	31**	31
4*	16	32,768	36	1,152	-	-
5*	20	40,960	45	1,440	-	-
6*	24	49,152	54	1,728	-	-
7*	31**	63,488	63	2,016	-	-

\*= only red and green

**The red value**, which finally will be placed into 5 bits (thus max. 32 shadings), originally is only present with 3 bits (8 shadings). We multiply this 3-bits-value with four and get value between 0 and 28. As the “7” originally meant 100% red, we have to correct this now: We correct the dec. 28 (=11100) to a dec. 31 (=11111), to get a 100% red (all 5 Bits to 1).

**The green value** originally is also present with 3 bits (0-8 dec.), so we need to multiply with 9 to use the band width of the 6-bits for green. As  $7 \times 9 = 63$  (111111 = all bits already set), we do not have to correct anything here.

**The blue value** is present originally only with 4 shadings. Therefore, we multiply with 10 and get the 4 possible values 0, 10, 20, 30. Here we need to correct again: to switch on all blue bits, we need to correct the 30 into a 31.



Example: The original color yellow from the 256-color-mode (RGB332) is decimal 252, and binary 11111100 – this will be separated first into its red, green and blue values:

111 = red: decimal 7  
111 = green: decimal 7  
00 = blue: decimal 0

Converted, this gives a double byte value of  $63,488 + 2,016 + 0 = 65,504$  or binary 11111111100000

Another example: bright green with binary 00111110 (decimal 62) gives  $8,192 + 2,016 + 20 = 10,228$

Of course, you may set up a lookup table in your software (this probably will speed up the conversion process as no calculations are needed) and then you might change the shading value of each color as you like (e.g. to show the blue value generally brighter you then save 0,15,25,31 instead of the 0,10,20,31 of our calculation).

**STOP:** Before you now dig too deep into this mode, read first the upcoming alternative 2 – very likely, this will be much more interesting for you than the RGB332 mode.

### **Alternative II : xxx out of 65,536 colours (colour table)**

This mode works with indexed colours (also called palette mode) and is far more interesting than the previous mode (RGB 332). It permits the use of indexed graphics files (e.g. : GIF or BMP). All 65,536 possible colours of the display can be used here.

Instead of the limited 256 colors of the previous mentioned RGB332-mode (e.g. maximum of 4 blue shades) you only have one limitation: you may use no more than 256 different colors at the same time at one graphics file. If you like, your graphic may contain 200 blue shades and 56 red shades. You also may use one small graphics file (e.g. an icon) with 256 colors and another small one with 256 completely different colors and display both at the same time.

This is done by a predefined colour table. instead of assigning the colour to the individual pixel, the individual pixel data of the graphics file contains a pointer to a table from which the correct colour is read (the color table). A 256 color graphics will come with a color table with 256 entries, a 16 color graphics comes with a color table with 16 entries (each entry containing a 16 bit color), etc...

Graphics like this, you meet on the internet every day. The usual GIF graphics always work with indexed colours and then is also is valid: more than 256 colours can not be contained in one file.

The advantage of this mode: Every single pixel can take use of the full 16 bit colour spectrum, however needing only 8 bits of information per pixel. a picture with 176 x 132 pixels now only needs 23,232 bytes despite the full colour spectrum of 65,536 colours instead of 46,464 bytes. One must add the colour table of a size of up to 512 bytes though (per colour 2 bytes of colour information for the use of 65,536 colours).

**Hint I:** If you are going to use many similar graphics, best would be to use the same color table for all graphics instead of having each file using its own one. This saves you additional memory. In most graphic programs, you are able to save and load a color table in GIF-mode. Then, you just save it once and you load it in for all the other files you want to save in GIF mode (e.g. in Adobe Photoshop® at menu "Image – Mode – Color Table" and then "Save"). Of course, this makes only sense when all your graphics are looking very similar regarding the used color. Another hint: If you know what graphics you are going to use: Create one big graphics file where you place all of your graphics side by side. Then create a 256 color color table and save it (e.g. Photoshop now will take care that all needed colors are present). Next you load each single graphic file and save it to GIF by using the color table you just created. Then all your graphics files will look nice and all will use the only one color table.

**Hint II:** If you want to use a graphics with only 16 or less colours, another useful variant takes effect: The pointer on the 16 possible color table cells needs only 4 bits. This means 2 pixels may fit in every byte of the pixel data now. We need 4 pixels in a byte fit and 8 pixels in a byte fit at only 2 colours at only 4 colours.

	<b>65,536 colours</b>	<b>256 colours</b>	<b>16 colours</b>	<b>4 colours</b>	<b>2 colours</b>
<b>Storage requirements Frame with 176 x132 pixels</b>	46,464 bytes	23,232 bytes	11,616 bytes	5,808 bytes	2,904 bytes

To use such graphics, please enter the preferences in our tool GLCD\_Convert. Then enter Palette mode and then check at data output “1-8 pixel per byte”. However, this format is not being used by our subroutines today. If you need this right away, you need to change our subroutines by yourself.

### ***Needed software adaption for the native 256-color-mode (8 bits per pixel)***

You need to change our software a bit if you like to use the 8 bits, 256 color mode of the display as our software only works in the 65K-color-mode (= we always transfer 16 bits per pixel). For using the 256-color-mode, you need to do some changes.

#### **Remark:**

We expect that the display usually is being used in the 65K-color-mode, therefore we only ship our software for this mode. Supporting both in one library would expand this too much. If you want to use the 256 color mode (not only for testing), send us a mail and if we received enough requests of our users that a library for 256 colors is needed, we will create one. But as it is not very difficult to change our software to this, you probably will be able to do this by your own.

#### Needed steps for this:

(Important: At the initializing sequence you need to change 7F3F to 7F1F):

The following steps shall be a hint and are probably not complete. As you understood the color modes and the principles behind it, it should be no problem for you to adapt our software from 16 bits to 8 bits mode.

#### *a) new definition of colors*

Actually, our colors are predefined as 16 bit colors. You need to change these constants in the software to the following 8 bits value (Basic needs &B before and C needs 0b )

```
Blue = 00000011
Yellow = 11111100
Red = 11100000
Green = 00011100
Black = 00000000
White = 11111111
Bright_green = 00111110
Dark_green = 00010100
Dark_red = 10100000
Dark_blue = 00000010
Bright_blue = 00011111
Orange = 11111000
```

*b) Changing of the format of the variables at all display routines*

All display routines such as LCD\_Print, LCD\_Box etc... are working with color variables with the numeric format Word (Basic: Word; C: unsigned int) – these need to be change to a byte format (Basic: Byte; C: Char).

Basic: The value 5808 need to be changed to the half: 2904 at the routine LCD\_CLS.

*c) Changing of the output routine*

Where actually 2 bytes per pixel are beeing transferred, you need to change the routine to that effect, that it is only transferring one byte (transfer the byte color value directly by SPI).

**In Basic:**

Spiout Data\_array(1) , 2 will be replaced by Spiout Color , 1  
The routine Set\_color does not need to be called anymore

**In C:**

You need to setup a new routine LCD\_SPI\_Byte which works like LCD\_SPI\_Int, but transfers only one Byte.

Thus:

```
void LCD_SPI_Byte(unsigned char Value)
{
    SPCR |= _BV(SPE);
    SPDR = Value;
    LCD_Wait();
}
```

Bitmap graphics

Graphics with 65.536 colors can't be used in the 256-color-mode of course, as these have 2 bytes of color information and the display is expecting 1 byte per pixel.

**Basic:**

Only available graphics mode is: Graphics\_mode=256low\_uncompressed and Graphics\_mode=256low\_uncompressed

At the subroutine Sub Lcd\_bitmap change the line

Spiout Data\_array(1) , 2 into Spiout Data\_array(1) , 1

At the subroutine Sub Interpret\_pixeldata delete the line

Gosub Convert256 and replace by Data\_array(1) = Data\_out

**C:**

The only available bitmap routines are: Bitmap\_256low\_compressed and Bitmap\_256low\_uncompressed.

Instead LCD\_SPI\_Int( CalcColor(PixelColor256) ) you will call LCD\_SPI\_Byte(PixelColor256) in these routines

### Compression / decompression of graphics files


Our tool GLCD\_Convert also offers an option for compressing the graphics files. Since the program memory in the microcontroller is limited, a compression offers itself virtually. A decompression routine like JPG would not be possible with a microcontroller as it would take too much memory and time. The relatively simple RLL (Run length limited) code was therefore used here. We explain the compression method in detail now:

Lets first summarize this briefly: Pixels of the same colour which are repeated will be summarized as [colour and # of apperances].

#### In detail (here using the 65,536-colour-mode as an example):

If one coloured pixel is present alone (no repeat, the next pixel shows a different color), this data looks identical as without any compression.

#### Example:




Data representation:

h001F	hF800	h07E0	h0000	hFFFF	h001F	hFFFF	hF800	hFFFF	h0000	h07E0	hFFFF
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

**I.e If a pixel is never followed by an identical pixel, a compression is not possible.**  
12 pixels need 12 x 2 bytes = 24 bytes

**If a pixel recurs, i.e. 2 pixels will have the identical colour, this pixel is present in the data 2 times followed by the # of repetitions.**

#### Example (6 blue and 4 white pixels):



Data representation:

h001F	h001F	h0004	hFFFF	hFFFF	h0002	h07E0	hFFFF
2 x blue		Blue 4 x repeated	2 x white		White 2 x repeated	Green	White

**If at least 2 pixels succeeding one another are identical, the number of repetitions is always inserted behind them.**

**Note:** In the 65k color mode, each pixel is presented by 2 bytes of color information. Then, also the number of repeats will be stored in 2 bytes number.

At the example above, 12 pixels does need 8 x 2 bytes (=16 bytes) which results in a 33% compression rate.

**Special case 2 pixels:** If a pixel colour succeeds one another only 2 times, the third byte (which contains the number of repetitions) has the value 0.

**Example (only 2 white pixels in the middle):**

Data representation:											
h001F	h001F	h0004	hFFFF	hFFFF	h0000	h0000	hFFFF	h07E0	hFFFF		
2 x blue		Blue 4 x repeated	2 x white		Nothing to repeat	Black	White	Green	White		

At this special case it is demonstrated, that compression not always will have a positive result. If you very often have a maximum of identical 2 pixels, the total usage of memory might be larger than without compression.

**The decompression:**

Data compressed by the compression routine explained above, can be relatively easy decompressed by the microcontroller. It just has to check if a read pixel has the same color as the pixel before; if yes, it reads the # of further occurrences; if no, it reads the next pixel.

**Compression and colour depth**

It is obvious that compression of a picture which show only few colours is more promising than at a picture which contains a lot of different colours in not coherent areas (the less colors are used, the bigger the chance, that pixel colors are repeated again and again).

For this reason, 256 color pictures usually show a better compression result than 65K color pictures.

**Compression and colour depth**

It is obvious that compression of a picture which show only few colours is more promising than a picture which contains a lot of different colours in not coherent areas ( the less colors are used, the bigger the chance, that pixel colors are repeated again and again).

For this reason, 256 color pictures usually show a better compression result than 65K color pictures.

The Display3000 logo for example (see photo, also on our CD), contains only 256 colors and offers several areas with the identical color. Because of this is can be compressed by 76% (in 256 color mode) or 71% (in 65k color mode). It is a big difference if a graphics consumes 46 KBytes of data or only 13 Kbytes (65K-color-mode) or 5Kbytes (256-color-mode).



Comment: In the real 256-color-mode (RGB332), this graphics will not look very nice as of the limited number of only 4 possible blue shadings at this mode. Here, the 256 color palette mode would be the best choice.

**Note:** If the data is stored in a 256-color-mode, of course only 1 byte for each pixel is being used then. Then, also the number of repeats is being stored in a 1 byte cell which means, only a maximum of 255 repeated pixels are possible. If the number of repeats should be larger, the sequence starts again with 2 pixels and then the number of remaining repeats.



### Compressing and using of indexed colors (palette mode)

The compress-algorithm of GLCD\_Convert is identical when using indexed color mode. Again, as with the RGB332-mode, the number of repeats cannot exceed 255 as repeats are only stored in a byte value.

#### Example: graphics with 256 indexed colors

Color table (here: 256 values)

Color 01 = blue                      Color 02 = green                      Color 03 = red  
 Color 04 = white                      ..... Color 05 to FE.....                      Color FF = black

#### Example: (280 blue pixel)



Data representation:

h01	h01	hFF	h01	h01	h15	h04	h04	h02	h02	hFF
2 x blue		blue repeats 255 x	2 x blue		Remaining repeats: 21 (280-2-255-2)	White	White	White needs two more repeats	Green	black

$$2 + 255 + 2 + 21 = 280 \text{ blue Pixel}$$

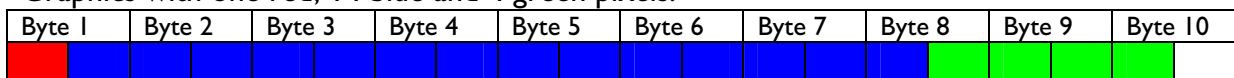
### Special case with reduced colors and reduced memory usage (>1 pixel per byte)

As mentioned before, GLCD\_Convert is able to store graphics, which just a few colors in a more efficient way. By using only 16 colors, there is room for 2 pixels per byte as these 16 colors only need 4 bits of information. We do not support this mode yet, but you will be able to reprogram the bitmap output routine if you need this.

The special case comes up if several pixel are stored in one byte. The compress algorithm is always checking the output byte in this case – single pixel are ignored.

#### Example: 16-color-mode (= 2 Pixels per Byte)

Graphics with one red, 14 blue and 4 green pixels:



Data representation:

h31		h11		h04	h12		h22		h24	
h03	h01	h01	h01	h04	h01	h02	h02	h02	h02	h04
red	blue	blue	blue	4 repeats	blue	green	green	green	green	white

Explanation: Byte 1 is not identical to byte 2: no repeat

Byte 3 is identical to byte 2 and will be repeated 4 times (Bytes 4-7)

Byte 8 is not identical to byte 7 (and this is the reason, why one single blue pixel follows the other repeated blue pixels) and also the 4 green pixels does not fall in any byte frame where a repeat would be possible.

### **Creation and output of graphical elements / photos**

After lots of theory, now to the real work with the  $\mu$ controller. Basically the output is similar to an output done with our subroutine LCD\_Box: We open an output window and send a lot of pixel data until the box has been filled. The only difference is that now, each pixel might have its individual color.

The biggest problem: As long as you do not create graphical elements by your program (as a colored box), you need to save the pixel data somewhere where your program can access these data. This can be the program memory (flash) or the eeprom of the microcontroller.

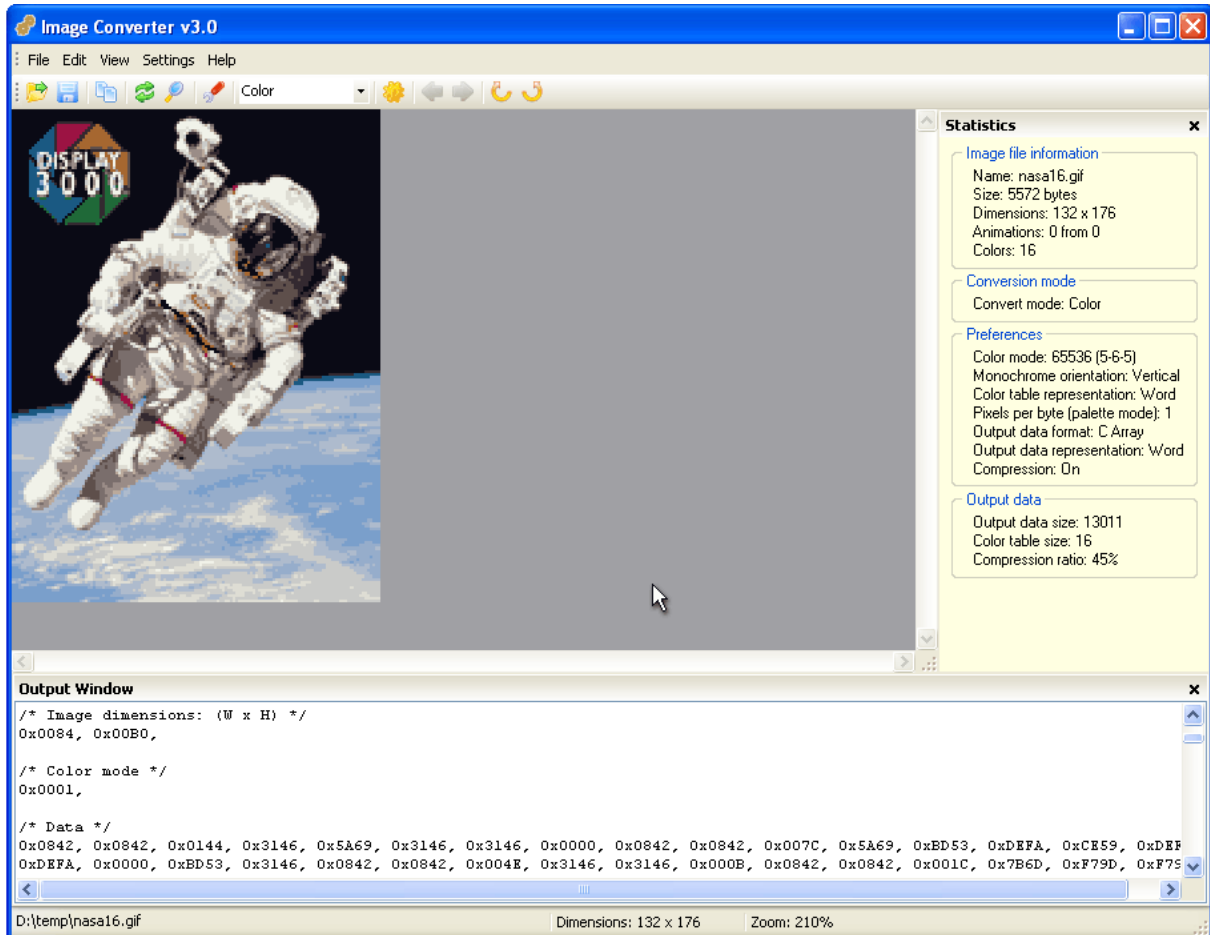
Beside the fact, that this amount of data consumes a lot of Flash memory (depending on size and number of colors), you need to include these data into your software (either as a data area in Basic or as an array in C).

There was no affordable software available which could deal with all different programming languages and graphics formats and which exports a usable output. For this, the program GLCD\_Convert was developed for us. This comfortable Windows program allows you a lot of options, including color reduction and indexing (more on this later).

The windows software *GLCD\_Convert.exe* is being found on our CD. For your first quick-start, we already placed several graphics files on our CD at the directory \graphics. Just start this program and play around.

## The GLCD\_Convert program

This program has been written exclusively for this display, but we also kept in mind our further developments so you will be able to use it also in the future.



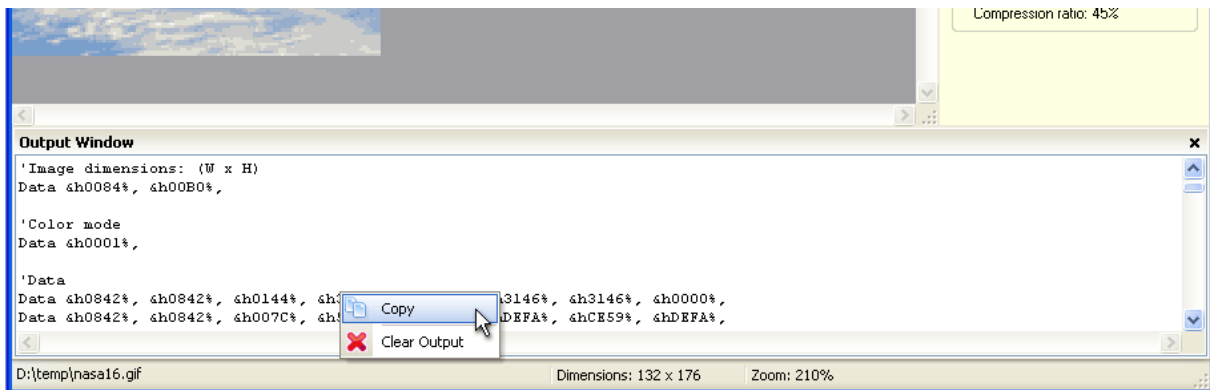
Beside the menu, the display screen consists of 3 main areas

- Input: Here, the loaded graphics file is shown
- The output window: Here, after starting the conversion process (with F5 or *Edit/Convert*), the data for the microcontroller is showed.
- The statistics/parameter window: All selected parameters and the data about the current output data (if any) are showed.

### The user interface:

Using this program is very simple and almost self explaining.

- 1) With drag and drop (if switched on in the preferences) or using *File/Open* you load your selected graphic file.
- 2) Then, at *Settings/Preferences*, you select the needed conversion parameters. These are being saved in a configuration file, so you do not need to do this parameter selection again if you want to stick with the last choice. Hint: You will be able to see most of these selections at the statistics windows and you may also click on these directly.
- 3) With F5, the conversion process is started.
- 4) The output window then shows you the data.

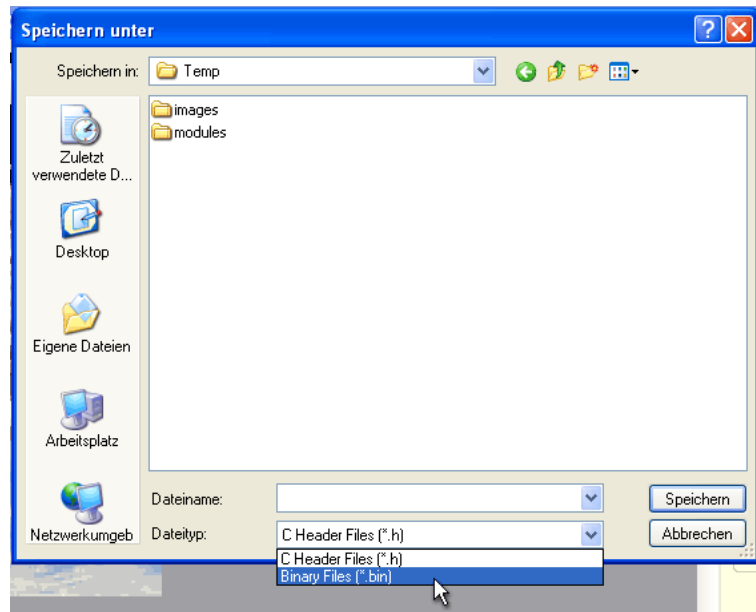


With right mouseclick and *Copy*, you copy these to the clipboard and you may then paste the converted data into you program editor. After changing a parameter, you need to start the conversion process again – the output window will not be updated automatically.

Alternatively, you may save the output data to a binary or a .h file. For this, you select *File/Save As* after the conversion finishes and then choose either a binary file or a header-file for C (xxx.h) (see picture beneath).

By this, you do not need to include hundreds of code lines into your program code (this will then be included during compilation).

At Bascom Basic for example, such a binary file will be included with the command `$INC:`



Example:

```

Set Lcd_port.lcd_cs
Return
$inc T1a , Nosize , "D:\temp\Nasa-L.bin"
Display init:

```

This one line would be the same as hundreds of Data-lines preceded with the label "T1a".

## Necessary manual work on the output data

The output data will contain some additional information that we do not use in our software routines yet. You might go and change these routines for using those or you go and delete these information from output data of GLCD\_Convert.

See the next code fragments. **Everything which is yellow, needs to be commented or deleted by you.** Green means: is already commented by GLCD\_Convert.

Important for the C-Array: The array size needs to be adapted after this (example: 8715). As you delete 3 values, you need to reduce the array size by 3 (then 8712).

### Example for a C-Code:

**Important !      Please read!**

```
/*
Graphics data created with GLCD_Convert developed by Konstantinos Halakatevakis.
Looking for cool displays? Visit "http://www.Display3000.com"
*/

const unsigned int Table_colortest1-32gif[32] PROGMEM = {
0xCE5F, . . . . .
};

/* Color mode byte:

Bit      0:    0    --> Compression on
           1    --> Compression off

Bit 1 - 4:    0000 --> 65536 colors
              0001 --> 4096 colors
. . . . .    etC. . . . .
*/

const unsigned int colortest1-32gif[8715] PROGMEM = {
8715 needs to be corrected to 8712 after removing the three blue marked values
/* Image dimensions: (W x H) */
0x0084, 0x0084,

/* Color mode */
0x0306,

/* Data */
0x0101, 0x0101, . . . . .

```

The output of the Basic data also needs a manual deletion of the size-data (an array size as in C is not available).

```
'Image color table size: 32
Table_colortest1-32gif:
Data &hCE5F%, . . . . .

'Color mode byte:
'
'Bit      0:    0    --> Compression on
. . . . .

colortest1-32gif:
'Image size (including dimensions, color mode and palette colors): 8715
'Image dimensions: (W x H)
Data &h0084%, &h0084%,

'Color mode
Data &h0306%,

'Data
Data &h0101%, &h0101%, &h0101%, . . . . .

```

### Three different conversion modes

Basically, there are three different main color modes, which you can select at the top menu line. These three modes are:

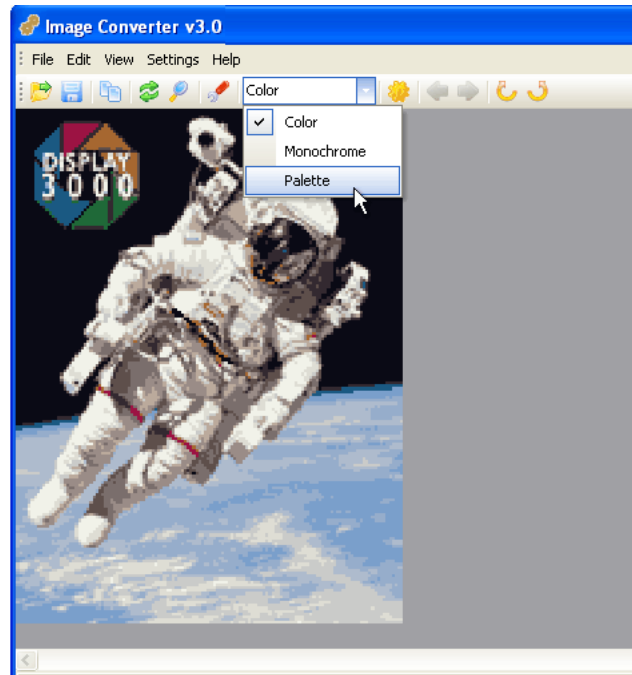
#### **Color / Monochrome / Palette**

Color creates „normal“ data of a color-graphic as a JPG-File

Monochrome creates data for a monochrome file (black and white or if you need and do the change in the software: blue and white or red and yellow etc...).

We will not go into detail about monochrome graphics in this manual.

Palette: This mode works with a maximum of 256 colors out of the complete 65k color palette. More about this at page 34. The mode *Palette* only works correctly, if you load a graphic file with an indexed color table (usually GIF-Format).



**Basically the following facts should be known: The mode Color can be displayed more quickly if no calculation or extra code is needed (just read and send). Mode Palette does need much less memory but, it is a bit slower due to the extra code for reading the color table.**

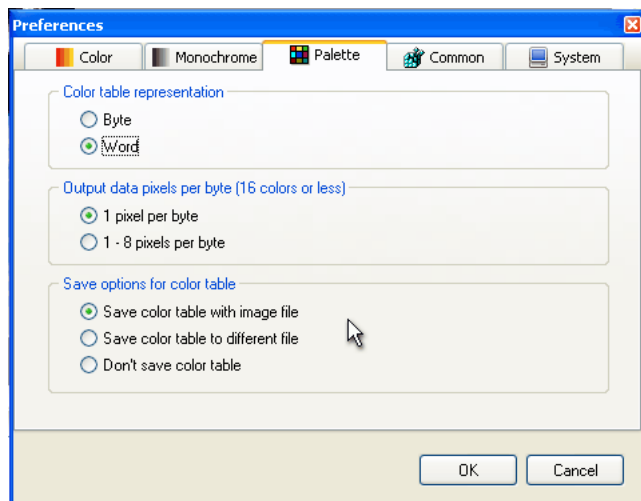
### The parameters !

The most important settings being mentioned here:

Color table representation: Here you select, if the entries in the color table should be 2 x 8 bits value (2 Bytes MSB & LSB) or 1 bit value (word).

Output data pixels per byte: This defines if (in palette mode only) more pixels may be placed in one byte or if each pixel shall get a complete individual byte (right now, only the latter is supported by our software routines).

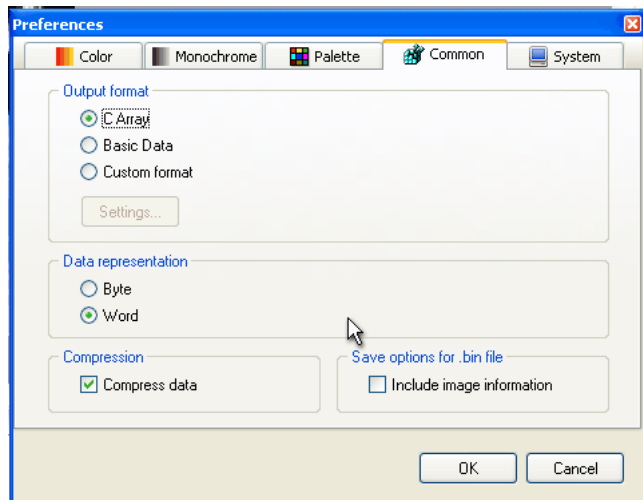
Save options for color table: This is only for interest, if you work in palette mode and if you have loaded a file with indexed color table! e.g. GIF). If you save a file with output data (File Save as), this option defines if the color table shall be saved to the beginning of this file to or if the color table shall be saved as an own individual file (this makes sense if several pictures are using the same color table and then might share the color table thus saving memory).



## The parameters II

**Output Format:** Defines if you would like to format the output for C, for Bascom Basic or any other user defined programming language.

**Data representation:** Defines, if the pixel data shall be formatted as 8 bits (Byte) or 16 bits (Word) in the output window.



The best used combinations are: **(Important – memorize this !)**

Color mode	Number of colors (Menu Color)	Compression (Menu Common)	Data Representation (Menu Common)	Color table Representation (Menu Palette)
Color	65536	An	Word	Don't care – not used
Color	65536	Aus	Egal	Don't care – not used
Color	256	An	Byte	Don't care – not used
Color	256	Aus	Byte	Don't care – not used
Palette	Don't care – not used	An	Byte	Word
Palette	Don't care – not used	Aus	Byte	Word

**The (usually) most efficient mode is:**

**GIF-file used!!** (up to 256 colors);

**Menu Palette:** Color table representation: Word

**Menu Palette:** Output data, pixels per byte: 1 pixel per byte

**Menu Common:** Data representation: Byte

**Menu Palette:** Compression: Yes

**Save options for .bin file:** Here, you define if, by saving a binary file, the information regarding size and color mode will be present in this binary file or not. It makes sense to write these informations to the file if you create a subroutine which shall work with any binary file, without knowing upfront of size and color format.

**Options, the GLCD\_Convert tool does not offer:**

This tool is not a graphics editor. It does not make sense to offer functionality that any graphic program is able to do much better. Some of these functions are:

**Resizing / Cutting of graphic files:** You need to offer our tool exactly the file size you want to use.

**Drawing:** Sorry, lines, pixel, etc... you better draw with a graphic editor.

**Creation and/or modifying of color tables:** Again, for this, it would be better do with a program like Adobe Photoshop & Co. They use algorithms that are highly developed and they have years of experience. To do this job better than a tiny tool.

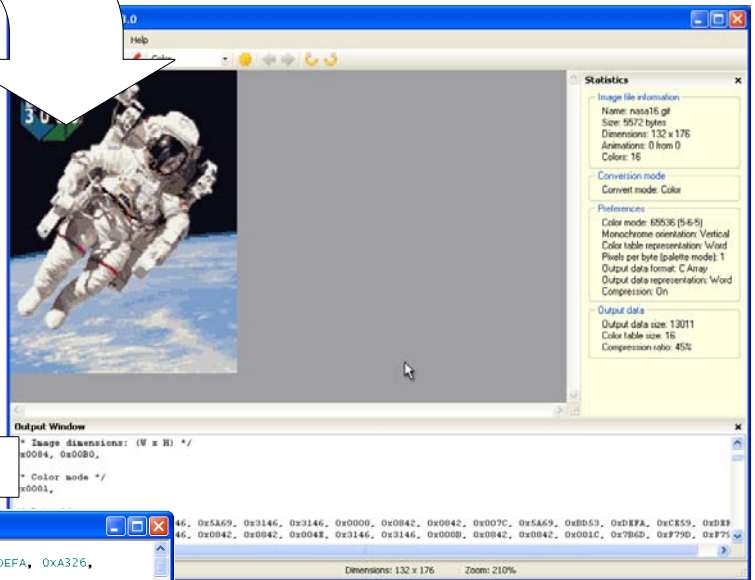


## Overview – how to get bitmaps into the microcontroller?

1) resize / cut graphics in a graphics program to a size of 176x132 pixel (or less) and save as JPG or GIF (eventually with a reduced number of colors)



2) Use GLCD\_Convert.exe from our CD to convert these graphics files into program data



3) Copy&Paste these data in your program

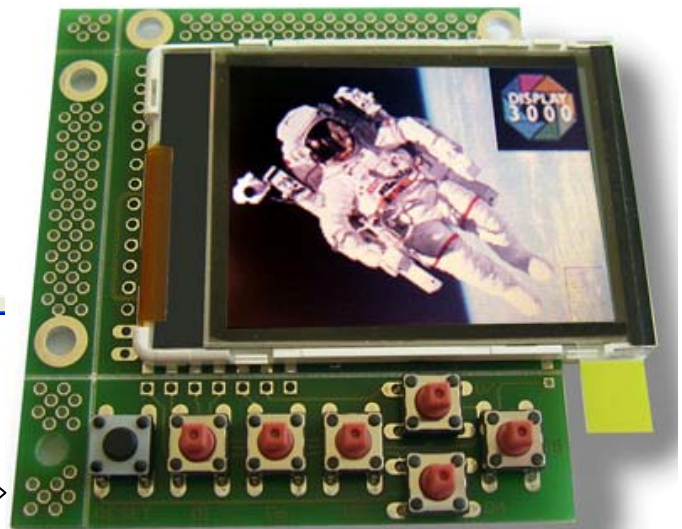
```

1  const unsigned char Table_nasa16[16] PROGMEM = {
2  0x9888, 0x3146, 0x0842, 0x9579, 0xA0D9, 0x7CF8, 0x
3  };
4
5
6  /* Compression ratio = 44% */
7  const unsigned char nasa16[13017] PROGMEM = {
8  /* Image dimensions: (w x h) */
9  0x00, 0x84, 0x00, 0xB0,
10
11  /* color mode */
12  0x04, 0x07,
13
14  /* Data */
15  0x02, 0x02, 0xFF, 0x02, 0x02, 0x43, 0x01, 0x0E, 0x01, 0x01, 0x00, 0x02, 0x02, 0x7C, 0x0E, 0x0C,
16  0x0A, 0x08, 0x0A, 0x0A, 0x00, 0x0C, 0x01, 0x02, 0x02, 0x4E, 0x01, 0x01, 0x0B, 0x02, 0x02, 0x1C,
17  0x0F, 0x09, 0x09, 0x05, 0x08, 0x01, 0x02, 0x01, 0x02, 0x02, 0x4A, 0x01, 0x00, 0x00, 0x0A, 0x01,
18  0x01, 0x00, 0x08, 0x01, 0x02, 0x02, 0x18, 0x01, 0x09, 0x09, 0x07, 0x0A, 0x0C, 0x0A, 0x0F, 0x02,
19  0x02, 0x48, 0x06, 0x01, 0x00, 0x00, 0x09, 0x01, 0x01, 0x00, 0x08, 0x0B, 0x01, 0x01, 0x02, 0x02,
20  0x17, 0x0E, 0x09, 0x09, 0x01, 0x0A, 0x09, 0x09, 0x05, 0x08, 0x0B, 0x02, 0x02, 0x47, 0x06, 0x06,
21  0x00, 0x01, 0x00, 0x00, 0x0B, 0x01, 0x01, 0x00, 0x0B, 0x08, 0x03, 0x01, 0x02, 0x02, 0x16, 0x01,
22  0x0A, 0x04, 0x01, 0x09, 0x08, 0x00, 0x0C, 0x0A, 0x09, 0x09, 0x02, 0x0C, 0x01, 0x02, 0x02, 0x45,
23  0x06, 0x06, 0x01, 0x01, 0x00, 0x00, 0x07, 0x01, 0x01, 0x00, 0x0B, 0x05, 0x01, 0x02, 0x02,
24  0x16, 0x00, 0x0A, 0x09, 0x0C, 0x01, 0x01, 0x01, 0x0F, 0x0A, 0x09, 0x09, 0x02, 0x0C, 0x01, 0x02,
25  0x02, 0x43, 0x06, 0x06, 0x02, 0x01, 0x00, 0x00, 0x06, 0x01, 0x01, 0x00, 0x0B, 0x0B, 0x07, 0x01,
26  0x02, 0x02, 0x15, 0x01, 0x0A, 0x08, 0x02, 0x02, 0x02, 0x01, 0x0C, 0x09, 0x09, 0x03, 0x0A, 0x0E,
27  0x02, 0x02, 0x41, 0x06, 0x06, 0x03, 0x01, 0x00, 0x00, 0x05, 0x01, 0x01, 0x00, 0x0B, 0x0A, 0x09,
28  0x01, 0x02, 0x02, 0x14, 0x00, 0x09, 0x0C, 0x02, 0x02, 0x03, 0x0F, 0x09, 0x0C, 0x08, 0x0A, 0x0A,
29  0x00, 0x09, 0x00, 0x02, 0x02, 0x40, 0x06, 0x06, 0x04, 0x01, 0x00, 0x00, 0x04, 0x01, 0x01, 0x00,
30  0x0B, 0x0B, 0x0B, 0x01, 0x02, 0x02, 0x12, 0x01, 0x09, 0x09, 0x00, 0x08, 0x01, 0x02, 0x02, 0x01,
31  0x01, 0x00, 0x08, 0x08, 0x00, 0x0C, 0x08, 0x0A, 0x0C, 0x01, 0x02, 0x02, 0x3F, 0x06, 0x06, 0x05,
32  0x01, 0x00, 0x00, 0x03, 0x01, 0x01, 0x00, 0x0B, 0x0B, 0x00, 0x01, 0x02, 0x02, 0x11, 0x0E, 0x09,
33  0x09, 0x01, 0x0C, 0x01, 0x0E, 0x01, 0x0F, 0x0C, 0x08, 0x08, 0x00, 0x0C, 0x0F, 0x0F, 0x00, 0x0E,
34  0x02, 0x02, 0x3F, 0x06, 0x06, 0x06, 0x01, 0x00, 0x00, 0x02, 0x01, 0x01, 0x00, 0x0B, 0x0E, 0x0B,

```

4) Adapt output data to our library (delete size information and change array size)

5) Now compile and write the program to your microcontroller. Ready.





## Some hints from personal experience

**Bascom-programmers take care:** Some older editions of Bascom basic do have problems when large areas of data are used. You then shall read our FAQ at page 67.

**C-Programmers take care:** With WinAVR you are not able to use arrays of a size larger than 32,768 Bytes. A full size picture would have 46,464 bytes (132 x 176 pixels x 2 bytes). If you want to work in full color mode (65K color), you must divide your graphic in two halves. You need to convert them separately and to set up two different arrays for them. There is no way around this as the internal pointer of the compiler is being used as signed integer, thus not being able to point to any number larger 32,768.

Our hint: Use GIF graphics with 256 colors (or less). Usually this is absolutely OK to display anything you like without any visible quality reduction and the graphics then needs – even without compressing – a maximum of 23,232 bytes.

Also the total amount of all bitmap data and other flash constants needs to be taken care of with WinAVR. You will run into problems if, you have more then 64 KByte of bitmaps – more details on this at page 16.

### **Bascom: Alternative to zillions of Data-lines**

To keep the code short you may save a binary file from GLCD\_Convert instead using big bunch of data-lines.

Using the command `$INC Label, nosize, "filename.bin"` at the end of your program (do not use this in the middle of the program – data should always be placed at the end) you inform Bascom, that it should include this file during compiling and that this should be treated as many lines of data statements.

Example `$INC Nasa-L, nosize, "D:\Temp\Nasa-L.bin"` adds the file Nasa-L.bin. To read these data out, you use the same commands as before

`Restore Nasa-L` and then read `READ` or alternatively: `Lookup`

### **Bascom: Usage of many color tables**

When using the command `Lookup`, you can unfortunately not pass the label over to this command. For this reason, our code is expecting the label `colortable` for this (subroutine „`Get_indexed_color`“). You need to change this label if you want to use a different name for your color table. If you need more than one color table you need to change the code e.g. with `“Select case”` to get the data from the correct color table.

## Changing the output direction (rotating the display)

Depending on your needs it might be necessary that the display needs to be mounted by 90° or 180°. Luckily the display offers some basic rotation functions which will allow us an easy programming.

With the command `Orientation =` you switch to a different orientation with the next output. This is always only valid for new outputs, the content, currently showed at the display will never be changed.

`Portrait` is the standard mode for module D071

With `Portrait180` the content will be turned by 180° – this is the default mode for module D072 (if the switches shall be below the display).

`Landscape` is the default mode for module D073, as the switches there are located at the longer side of the display.

`Landscape180` will rotate the landscape by 180°.

Of course any of our modules can work with any output direction.

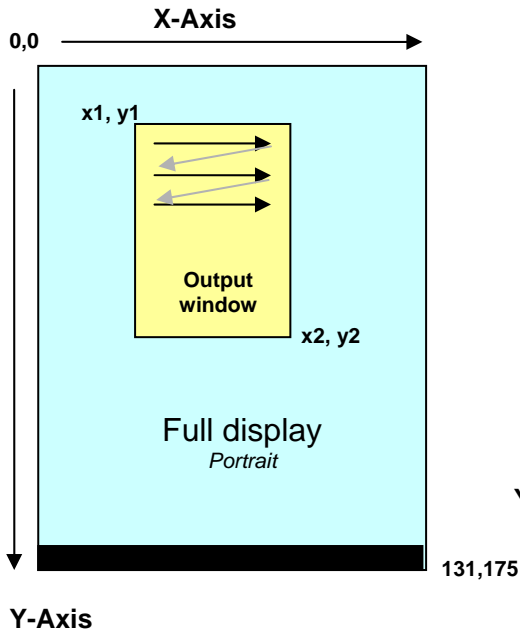
### **Comment:**

Also take the output quality into account when you choice of the output direction of the module. The quality of the picture is, as usual with TFT displays, depending of the viewing angle. Always consider the final mounting situation of your module. E.g. if you work with module D072 you look to the display from the bottom (6 O'clock orientation) during testing because the module lays in front of you on the table. If you mount this later to a wall in 40 inch height (1 meter) you then look to the display from the top (12 O'clock orientation) which will give you different display quality. It might be that you then notice that you would better buildt in the module turned by 180° (then you just need to change the orientation parameter in our software).

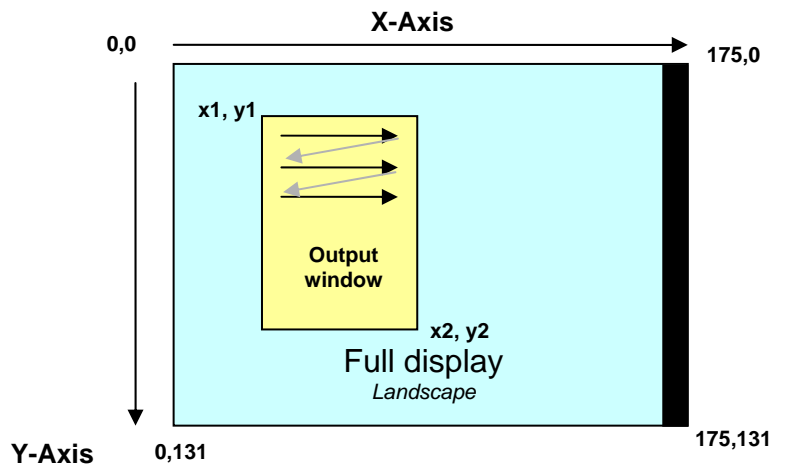
In landscape mode, the picture contrast etc... will differ depending if you look on the display from the left, center or right.

The following display is valid if you use our subroutines which will do all needed calculation. The black stripe represents our display connector (not the silver stripe at the display).

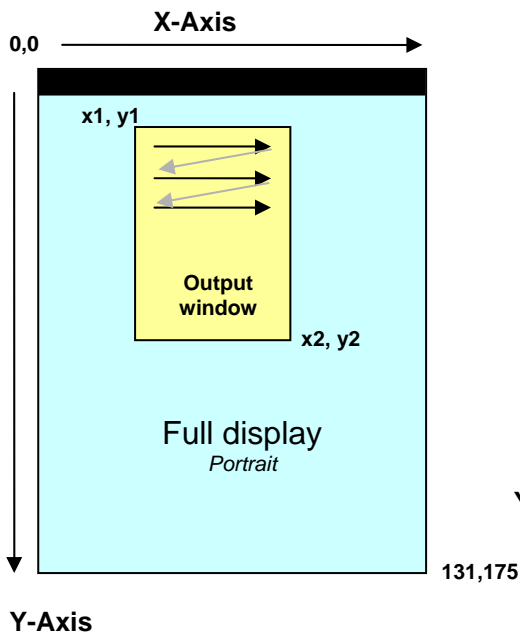
**Usage in Portrait mode  
(Orientation = Portrait)**



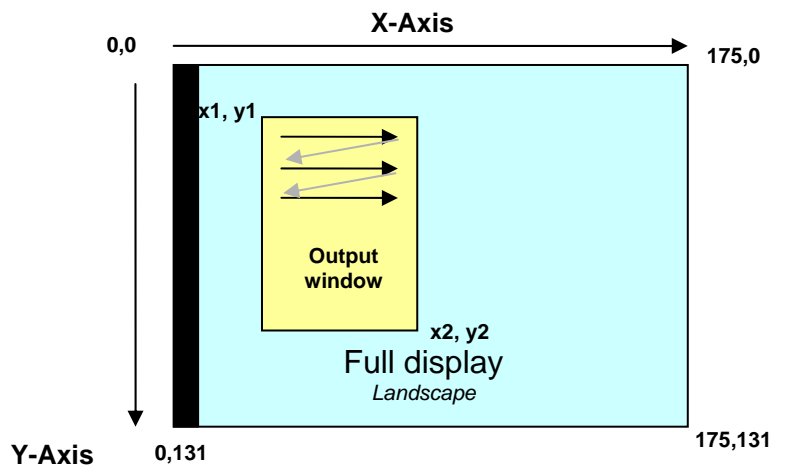
**Usage in Landscape mode  
(Orientation = Landscape)**



**Usage in rotated Portrait mode:  
(Orientation = Portrait | 80)**



**Usage in rotated Landscape mode:  
(Orientation = Landscape | 80)**



## **Conversion of the software to other systems or other programming languages**

Our program code was well documented and should be able to be rewritten therefore by any programmer in short time into other programming languages or for other  $\mu$ controller.

We tried to use no special instructions, nevertheless, there are three instructions used in most sample programs, which are not offered by all systems and need a closer description. Besides we already reprogrammed them for using these commands at other environments.

### **Bit inquiry**

In Bascom, the bit No.  $x$  of the variable  $A$  can be queried by means of  $A.x$  (this can also be used to set a bit). With a loop from 0 to 7, thus each individual bit of a byte variable can be queried. Programming languages, which do not offer this instruction, could use the AND instruction for substitution.

You may use the **AND** command with the value of the demanded bit.

$Y = A \text{ AND } 2$  results in 0 only if bit No. 2 is not set.  $Y = A \text{ AND } 32$  is 0 only if bit No. 6 is not set.

### **Set, RESET**

Set  $X$  is identical to  $X=1$

RESET  $X$  is identical to  $X=0$

### **SPIOut**

With this command, the given byte is automatically transferred to the display using the hardware SPI buildt into the ATmega (the controller is taking care of setting the data and clock line correctly). As the SPI buffer does only allow 1 Byte, we always need to split our color values into 2 bytes and pass then one after the other. SPI ports are predefined and cannot be changed.

### **Shiftout (not used here)**

Shiftout is similar to SPIOut, but is not using any buildt in hardware – then, this is simulated by the software. The advantage: very flexible as you can decide which port to use. The only disadvantage: this is much slower than using the buildt in hardware SPI.

## Reference of driving the 2.1“ Display

The driving of the display was determined by us by reverse engineering. Software reverse engineering is defined as “reversing a program's code (the string of 0s and 1s that are sent to the logic processor) back into the source code that it was written in”.

We have operated the display in its original application and logged the complete data traffic between display and its hardware. The analysis of this data (we are talking of several hundreds of Mbytes where we had to split important command data from non important pixel data) took its time but finally, the display revealed its secrets.

### The serial interface of the graphics controller

This display works in two modes: 8 Bits per pixel or 16-Bits per pixel. Commands will be always 16 bits large, the single pixel however can be defined as a 1 of 256 colors (8 bits) or a 1 of 65.536 colors (16 bits). The color mode needs to be defined during the start up initialization and cannot be changed (except by running a new LCD\_Init sequence) during the regular operating of the display.

#### 8 Bits-Interface

Each command needs 2 bytes, each pixel needs one byte. The pixel colors are predefined (RGB 332 as explained in our color modes previously).

#### 16 Bits-Interface

The display works in a 16 bits mode, i.e. every command and every pixel needs 2 transferred bytes.

#### Command / Parameter

The display distinguishes between the two different modes when receiving data

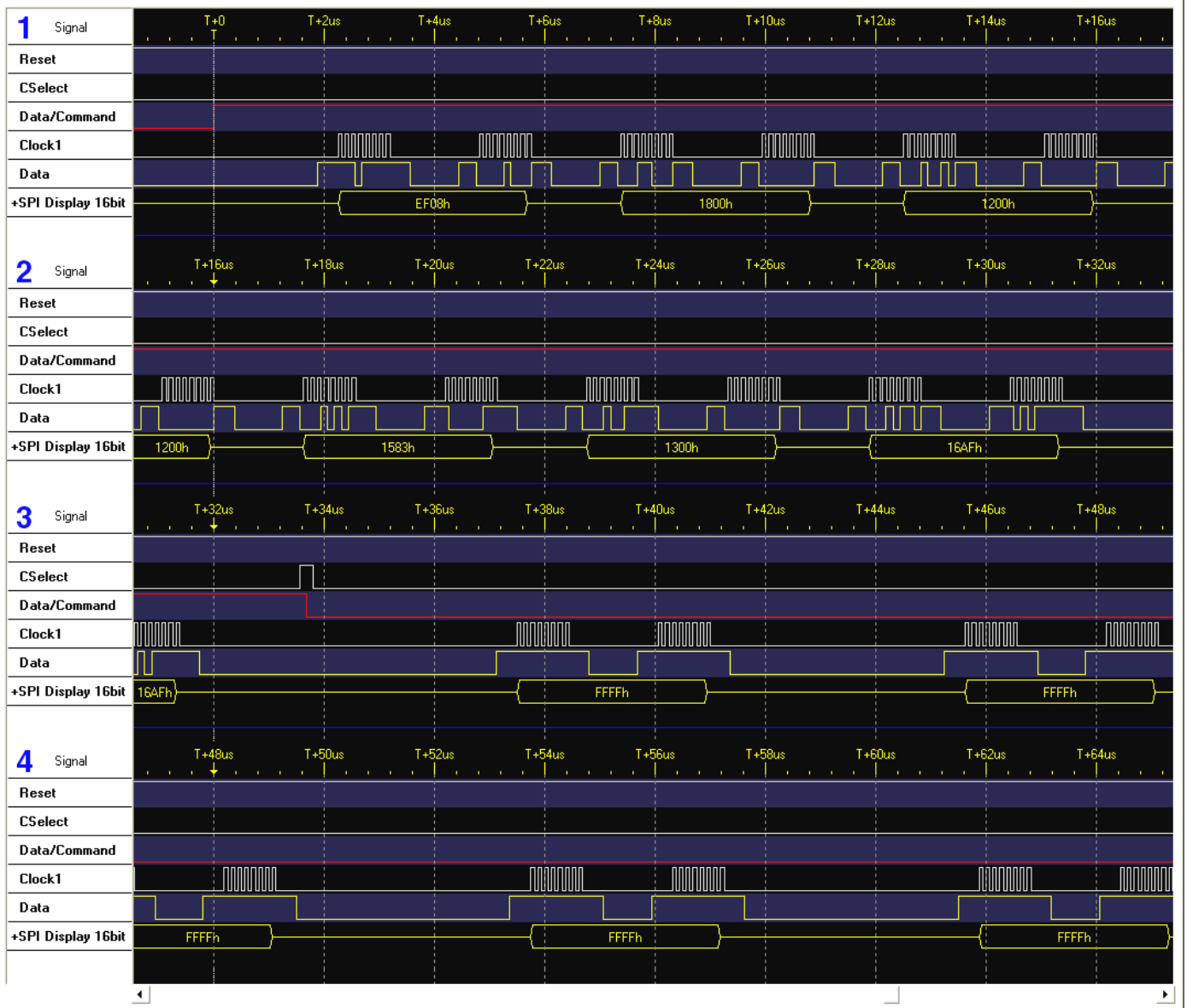
- Command mode
- Pixel mode – burst mode

**The command mode is indicated by a 0 at the line DC (Data/Command) during the data transmission.** The display therefore interprets the incoming data as commands to be executed. If a High (1) is set on line DC, the display interprets the incoming data as pixel data.

After receiving a complete set of commands, the display expects a one (1) at the line CS. Only then the data will be executed.

### Logic analyzer screens

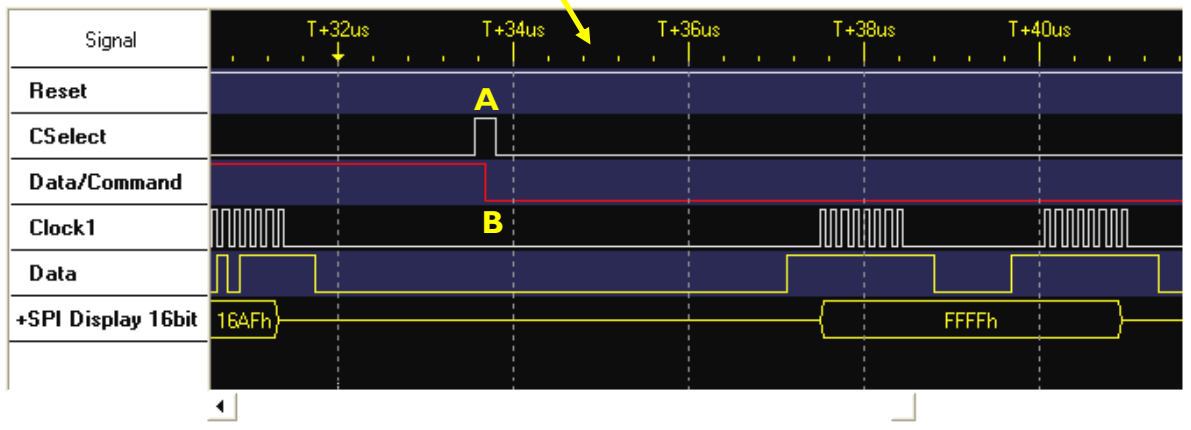
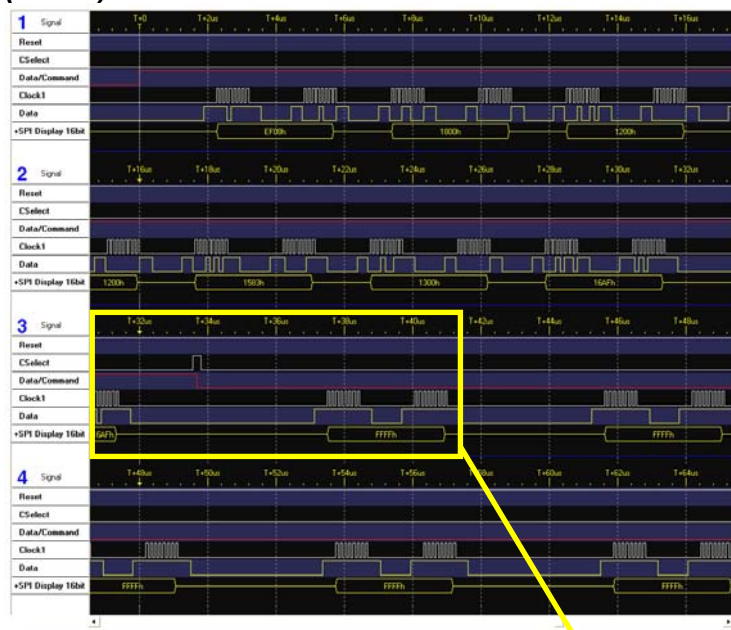
At the following picture, you see a small amount of data we have read in with our logic analyzer. Here you see 20 bytes (10 words) of data gathered in a time frame of approx. 64  $\mu$ s; four areas (marked with the blue numbers 1 - 4) are showing each 16  $\mu$ s – the time scale helps as guidance. The triggering (start of the record) starts with T=0  $\mu$ s. The display is working in 65K-color-mode (2 bytes per pixel).



This sequence shows the beginning of the command LCD\_CLS, the clear screen subroutine. With the first 6 commands, the output window is set to 131,175 (&h83, &hAF) (see also page 20 in this manual to get detailed information about setting an output window) and is then followed by 23,232 times a white pixel (&hFFFF). We show only 4 of these pixels and leave the remaining 23,228 to your imagination. ☺

To avoid any confusion by looking at this picture: The time scale of 0 to 64  $\mu$ s is divided in 4 areas, each showing 16  $\mu$ s. The end of one section and the beginning of the next section will show some overlapping. Just look at the time scale of each section – this helps.

**(Zoom):**



The picture above is a zoomed area of the picture of the last page (at position 34  $\mu$ s). After finishing sending the commands for opening the window, the short peak at line CD (marked with “A”) tells the display that this command now shall be executed. The Data/Command line (DC) was high all the time – the information for the display that a command is being transferred. As soon as DC goes to low (marked with “B”) the display exists the command mode and interprets the following data as pixel data.

The state of line DC therefore is important for the display to make difference between command data and pixel data.

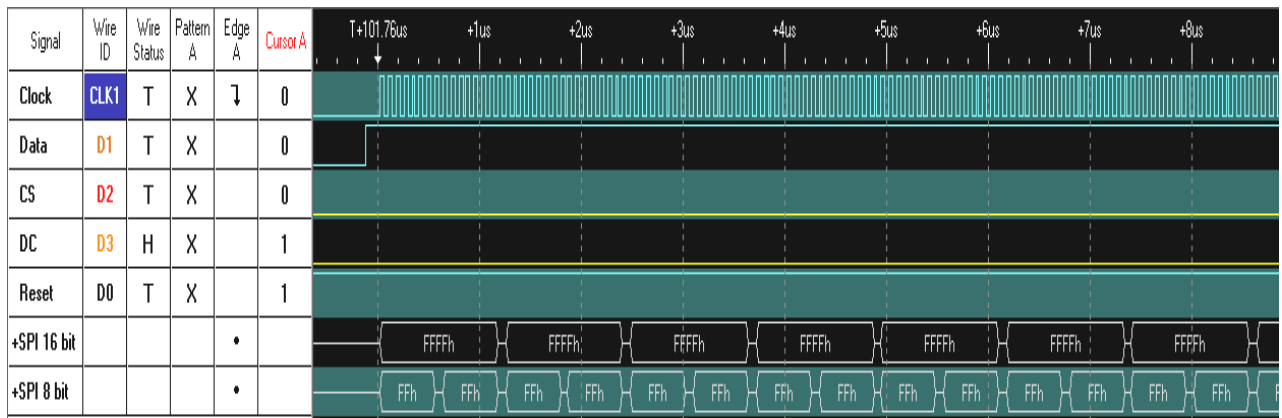
The pixel mode is realized as a burst mode, which means, the display get the data without the need of information on any further line. This speeds up transmitting the data.

At this example you might see which potential is in an optimization of the code. Theoretically the display can work with the data without any break. As of the data preparation of our software, these pauses are not avoidable but might be shortend through optimization.

Like many  $\mu$ controllers have a hardware SPI included, it makes sense to use this interface, as its send the data automatically by an included piece of hardware without any software overhead. For this reason, we are using the hardware SPI on our modules. How does the hardware SPI works: We place a byte into the SPI buffer and then the bits on the data line and the signals on the clock line are produced automatically by the hardware SPI. Usually, this is much faster than software.

Comment: As the serial interface (SPI) of a 8-Bit-Microcontrollers only offers a 1 Byte buffer, we can only transfer 8 bits at a time. For this reason we must split all 2-Bytes commands and 2-Bytes Pixel colors into two single bytes and transfer them through SPI one after the other. First the most significant byte (MSB), then the least significant byte (LSB).

Then here is a picture of the original application. This would be perfect: regarding speed: Pixeloutput at high speed and without any single pause (here white pixel, therefore always FFh): 10 pixels are transferred in approx 12.2 $\mu$ s; this makes one full picture in 28ms or theoretically 😊 35 full frames per second - (the display can work with a pixel clock of up to 13 Mhz).

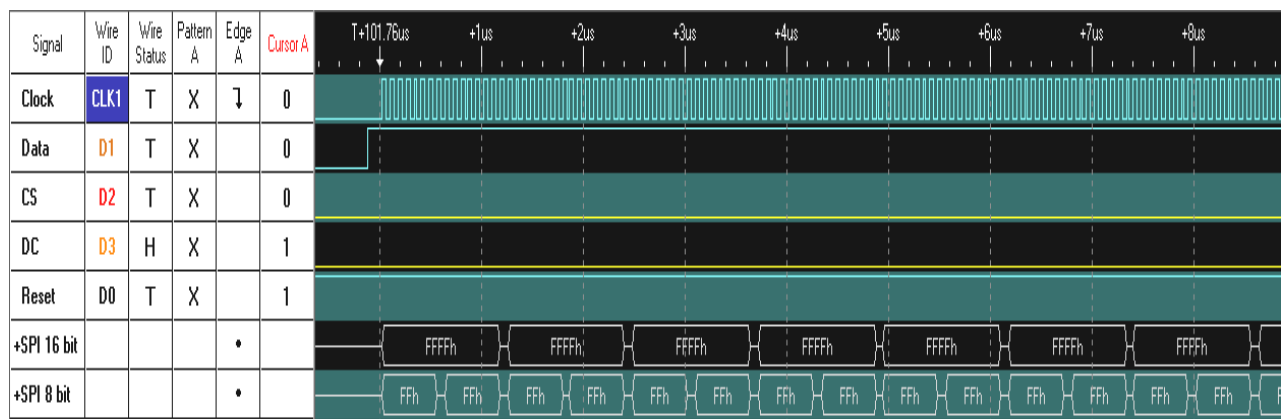




### Timing of the signals / clock line versus data line

If you are having problems check the timing, The complete clock cycle needs to be at least 70ns and the single clock peak at least 25ns high and at least 25ns low. There is no maximal length of a clock cycle known therefore you do not have to take about this if you are running a slow microcontroller. Only if you are running a very fast microcontroller (>30 Mhz) you might need to slow down some output procedures.

Also take care of the cable length when you want to mount the display in distance of the microcontroller. you will probably encounter problems, if you run the display on 40 inch cables (one meter) with a clock cycle of 70ns. Try to avoid any cable length of more than 20 cm (4 inch) and use a shielded cable.



## The display commands:

### **Initializing (65.536-color-mode – 16 bits per pixel)**

Even with a lot of effort, we were not able to isolate the different initializing commands of the display and to assign them to single actions. Even as we tried hard by sending all possible combinations: we also did not find out any other commands which would have an effect to the display.

Until this might be clarified some day, the sequence for initializing the display needs to be seen as a whole block.

1) **After powering the display up approx. 300ms break** is needed before a low at the Reset-line does a hardware reset.

Then several lines needs to be set and reset, **each action with a break of approx 75 ms** (you may shorten this break, but we found out during our tests, that 75ms is needed to ensure always a proper initializing):

**High at CS** (then a 75ms break)

**Low at Clock** (then a 75ms break)

**High at DC** (then a 75ms break)

**High at Reset** (then a 75ms break)

Then the initilaizing sequence follows, starting with **FDFDh**. This is probably the power up sequence of the display. After this sequence, **a break of 75ms is a must** (not much longer, not much shorter).

**Then the following initializing commands follows (line DC is always high):**

EF , 00 , EE , 04 , 1B , 04 , FE , FE , FE , FE , EF , 90 , 4A , 04 ,  
7F , 3F , EE , 04 , 43 , 06

**25-75 ms break**

EF , 90 , 09 , 83 , 08 , 00 , 0B , AF , 0A , 00 , 05 , 00 , 06 , 00 ,  
07 , 00 , EF , 00 , EE , 0C , EF , 90 , 00 , 80 , EF , B0 , 49 , 02 ,  
EF , 00 , 7F , 01 , E1 , 81 , E2 , 02 , E2 , 76 , E1 , 83 , 80 , 01 ,  
EF , 90 , 00 , 00

### **Alternate initializing (256-color-mode – 8bits per pixel)**

EF , 00 , EE , 04 , 1B , 04 , FE , FE , FE , FE , EF , 90 , 4A , 04 ,  
**7F** , **3F** , EE , 04 , 43 , 06

#### **25-75 ms Pause**

EF , 90 , 09 , 83 , 08 , 00 , 0B , AF , 0A , 00 , 05 , 00 , 06 , 00 ,  
07 , 00 , EF , 00 , EE , 0C , EF , 90 , 00 , 80 , EF , B0 , **49** , **02** ,  
EF , 00 , 7F , 01 , E1 , 81 , E2 , 02 , E2 , 76 , E1 , 83 , 80 , 01 ,  
EF , 90 , 00 , 00

**The sequence 7F, 3F** switches the display to the 16 bits mode as explained in this manual. 7F, 1F however will switch the display to the 8 bits per pixel mode – thus 256 colors per pixel.

**The sequence 49, 02** can be replaced by:

49, 03: inverse color or

49, 42; instead the bit sequence RGB, the display now expects the colors in RBG - for you, this mostly will be of very little interest.

#### **Switching the color mode after the initialization:**

EF, 90, E8, 0F switches to the 16 bits-mode

EF, 90, E8, 01 switches to the 8 bits-mode

Switching the color mode in runtime might make sense if you need to realize a very quick output where it is no problem to live with the limitation of the 256 color mode (RGB332).

### **Switch off sequence**

A user Controlled Switch Off of the display will enhance its life span. If you switch off and on the display (or the complete microcontroller board) very often and if you are able to know about this before, you shall send the Switch Off sequence first. Starting mid of March 2008 our modules D071 and D072 will offer an option to Switch Off the display electronically and power booster by the microcontroller (useful to save power at battery powered devices).

If you want to wake up the display after sending the following sequence, you need to call our initialization routine Init\_LCD.

### **The Switch Off sequence (line DC is at high level):**

EF , 00 , 7E , 04

Wait 50 ms

EF, B0, 5A, 48, EF, 00, 7F, 01

Wait 60 ms

E2, 92

Wait 90 ms

E2, 02

Wait 60 ms

EF, B0, BC, 02, EF, 00, 7F, 01

Wait 25 ms

80, 00

E2, 04

Wait 25 ms

E2, 00

Wait 25 ms

E1, 00

Wait 25 ms

EF, B0, BC, 00, EF, 00, 7F, 01

Wait 90 ms

Switch Power Off.

### **Define output window (where to write pixels)**

**For all activities of the display, there are command sets to define the output window and the output direction – we use them all in our software:**

**Introduction EF08h:** this starts a command sequence with followed parameters

**command 18h – Orientation;** it follows the parameter for the output direction (1 Byte)

00h **Portrait**

03h **Portrait rotated by 180°**

05h **Landscape**

06h **Landscape rotated by 180°**

**command 12h –window:** start X, followed by the **X-position as a parameter** (1 Byte)

**command 13h –window:** start Y, followed by the **Y-position as a parameter** (1 Byte)

**command 15h –window:** end Y, followed by the **X-position as a parameter** (1 Byte)

**command 16h –window:** end Y, followed by the **Y-position as a parameter** (1 Byte)

### **Switch the display to white or black screen (without losing the screen content)**

With the command EF90h 0040h, the screen will become completely dark, with EF90h 0080h the screen will become completely white.

By using this command, the display RAM will not be deleted and is even changeable during this black or white-mode: you may use all of our display commands. With EF90h 0000h, the display RAM will become displayed again – including all changes made by you in the meantime. This command is usable if you do not want to show the user how the screen will become updated. Then you switch it white or black, make all your changes you want and then you switch it back – all data will become visible at once.

#### **Usage in Bascom Basic**

```
D_data_out = &HEF90%  
Gosub Lcd_send_dbcommand  
D_data_out = &H0040%  
Gosub Lcd_send_dbcommand
```

**... Screen is black ...**

#### **To restore the content:**

```
D_data_out = &HEF90%  
Gosub Lcd_send_dbcommand  
D_data_out = &H0000%  
Gosub Lcd_send_dbcommand
```

## Set vertical offset / scrolling

With the command EF90h IIxxh the display will be informed about a needed offset (xxh). For example: if you send EF90h II20h, the screen will be moved by 32 pixels (20h) upwards (at mode Portrait180 it moves downwards, in mode landscape it moves to the right or to the left). With a loop and –important – a short waiting period of 2ms, you can realize some scrolling. Important: With this command all coordinates will move too: 0,0 will not be the upper left corner anymore – it is moved by the offset. Moving the screen is possible by a maximum of 176 pixel (thus, the parameter may contain values between 00h and B0h).

The screen content will not move out of the display (invisible) but will reappear at the bottom of the screen.

This command sets an absolute offset. This means you do not move the screen content by a number of pixel – you move the screen to a specific position.

Example: After moving the screen to position 10, a new call with position 11 (0Bh) will move the screen only by one pixel. To bring it back to the default position, you always need to use the parameter 00h.

### Usage in Bascom Basic:

```
D_data_out = &HEF90%
Gosub Lcd_send_dbcommand
D_data_out = &HII20%
Gosub Lcd_send_dbcommand
```

... screen will be moved by 32 pixel (20h) ...

### To move it back:

```
D_data_out = &HEF90%
Gosub Lcd_send_dbcommand
D_data_out = &HII00%
Gosub Lcd_send_dbcommand
```

## Scrolling / Offset of a predefined area instead the complete screen

Instead moving the complete screen, you may also move only an area of the screen. Remark: this area always will use the complete horizontal line.

The command sequence for this is:

EF90h, 0F start, 10 size, 11 offset

For complete scrolling of an area of 50 pixels (32h) from position 40 (20h) exactly 1 time (for a 50 pixels large area you need to move by 1 pixel 50 times), the code beside will be needed:

### Remark:

1) For scrolling you shall always use a short wait cycle, otherwise the screen moves too fast for your eye to realize a scrolling effect.

2) Scrolling in portrait mode is only possible in the vertical direction and in landscape mode only in horizontal direction.

### How to use in Bascom:

```
D_data_out = &HEF90%
Gosub Lcd_send_dbcommand
D_data_out = &H0F20%
Gosub Lcd_send_dbcommand
D_data_out = &H1032%
Gosub Lcd_send_dbcommand
D_data_out = &H1100%
For X = 1 To 50
  Incr D_data_out
  Waitms 50
  Gosub Lcd_send_dbcommand
Next X
```

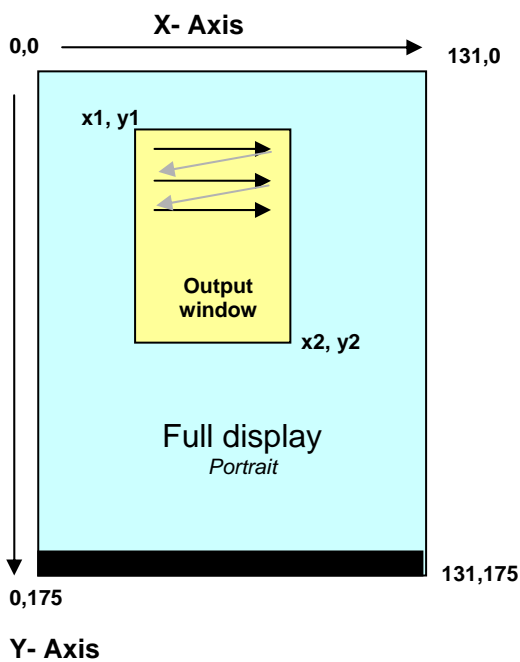
... Display scroll within this range...

### Coordinates and output direction

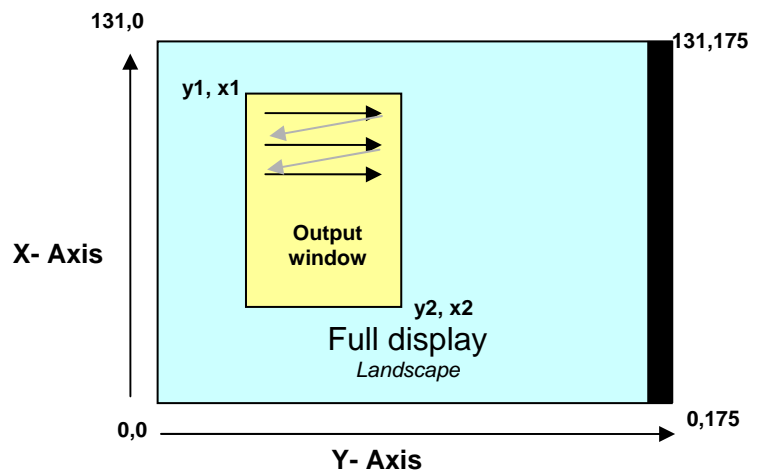
Comment: If you use our software subroutine LCD\_Window you better use the information on page 51 as we do there all recalculation for you. Then, if you rotate the display and if you then use the correct Orientation setting in the code, the upper left corner is always the reference for X=0 und Y=0.

Following, we will provide you with all information about the native display orientation. If you want to use our subroutine LCD\_Window, you might ignore this page totally, as this subroutine does all calculation for you (then better got to page 51). For your orientation: the black stripe shows the position of our display connector.

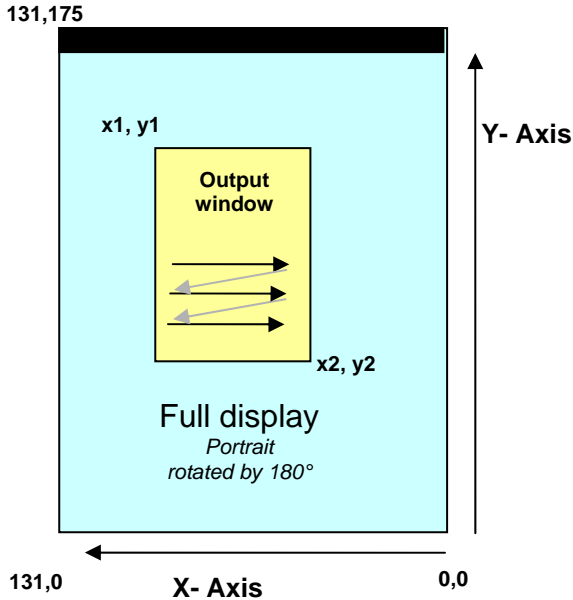
#### Usage in portrait mode: (Orientation = Portrait)



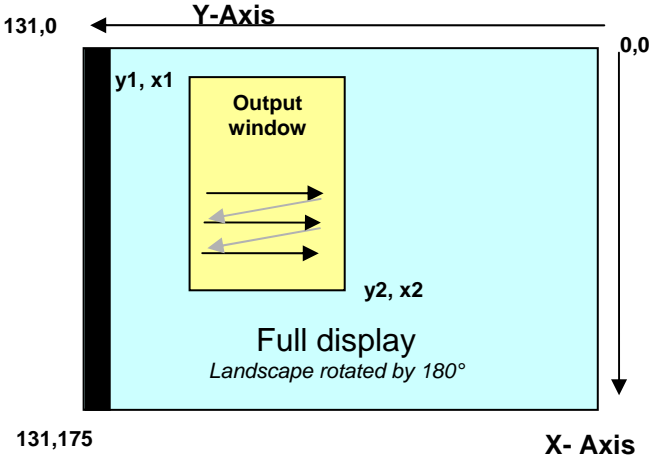
#### Usage in landscape mode: (Orientation = Landscape)



**Usage in rotated portrait mode:  
(Orientation = Portrait I 80)**



**Usage in rotated landscape mode:  
(Orientation = Landscape I 80)**





## Possible problems and their solutions:

### Nothing is happening; the display does not show anything.

#### 1) Did you rewrite the program for a different microcontroller or another programming language?

Examine all outputs and also the speed. Eventually you are sending the data too fast. Try to throttle the output rate (insert short waiting loops) if you are using a microcontroller with a speed of >30 Mhz (below, it is very unlikely that the speed is too fast).

#### 2) Did you define the correct port in your program?

Check

#### 3) Did you assign the port lines of the display correctly and did you solder the lines correctly (if not using our ready to run controller boards)?

Check. Hint: If you plan to use more often with  $\mu$ controllers you might think about purchasing a logic analyzer. Such a device will save you many many hours of work as you never need to guess again what is happening. Without a logic analyzer, we would'nt ever be able to discover how to drive the display.

#### 4) Bascom: Displaying graphics works fine, but displaying text does not work

You did not use the needed selections in the compiler settings of HWSTACK; SWSTACK and FRAMESIZE when using Bascom (**Menu: Options, Compiler, Chip**). See the comments at the beginning of each of our sample programs. You need the following settings with our sample programs:

```
HWSTACK = 64
SWSTACK =128
FRAMESIZE =16
```

If you do changes (especially when you use “Call” or “Gosub” you might raise these numbers (see Bascom-Help)

Hint: you might include the lines (with “\$”)

```
$HWSTACK = 64
$SWSTACK =128
$FRAMESIZE =16
```

at the beginning of your programs, so these values are set automatically during compilation.

#### 5) Electrical/static disturbances

The signals of the microcontroller are actually non-critical. Nevertheless, it is possible that longer cords or other unfavorable factors impair the usage. Then, the display works for a while and does not react then any longer (that might also be a timing problem). In this case, you should either use shorter cables (max 20 cm), or with necessary larger lengths use shielded cables.

Also a good strategy for devices which may run a long time: refresh the complete display content (eventually including initialization) in frequent periodics (e.g. once per hour or once per day).

### 6) With bright sunlight the display goes crazy

The silver shining part at the top of the display is a chip. Bright light might disturb it from working correctly. You may darken this with a frame or with a stripe of tape.

## My program needs so much memory.

### Hints for decreasing memory requirement.

1) We did write our code in a way, it is easy to understand. This is of course not the most efficient code as we proposed, that understanding the code is more important for you than getting effective but cryptic code.

2) **Bascom Basic:** Using the Call-Statements needs a lot of memory due to turning over the many parameters. Each "Call" command takes about 128 Bytes of memory. If you use these call statements very often memory may run short. We suggest 2 alternatives:

a) Reduce using the Call-statements by trying to share code segments with other parts of your program or

b) use the command `Gosub` instead `Call`. There is one disadvantage connected then: With `Call` you pass the content of variables or static content to the subroutine. You cannot do this with `Gosub`. This means, you have to set all variables the subroutine needs manually before you call the subroutine. An example:

Instead:

```
Call Lcd_print( "Hello world" , 1 , 1 , 1 , 1 , 1 , Dark_red , Yellow)
```

you need to call the subroutine `LCD_Print` (after doing some changes to the program) with-  
`Gosub LCD_Print`.

The needed parameters like `LCD_Text`, `XScale`, `YScale`, `Fontnr` etc. now need to be set by yourself before the `Gosub`. This might then look like:

```

Lcd_text = "Hello world"
Lcd_posx = 1
Lcd_posy = 2
Fontnr = 1
XScale = 1
YScale = 1
Lcd_fcolor = dark_red
Lcd_bcolor = yellow
gosub lcd_print

```

Often, many variables need to be set/used anyway in your program thus the output of the display only needs the Gosub command which needs only 2-4 bytes per usage. This means, you will win approx. 100 Bytes of memory for each eliminated Call statement.

Hint: For the above mentioned modification, there are two changes needed:

- a) Change the header subroutine "Sub LCD\_Print..." to an ordinary label "LCD\_Print:"
- b) Change the end of the subroutine from „End Sub“ to „Return“
- c) Delete the line "Declare Sub LCD\_Print .." at the beginning of the program.
- d) Change all occurrences of "Call LCD\_Print" to "Gosub LCD\_Print" and take care that all needed variables are set before the Gosub.

3) Graphical output of icons or other bitmaps always need more memory as you need to place the bitmap data together with the program memory. A full size picture with 65,536 colors does need more than 46.000 Bytes program memory as each pixel needs two bytes color information. If you want to use 3 pictures like that, the memory of an ATmega128 is filled up and there is not space left for any software code.

### Strategies for saving memory space:

- a) Include small graphical elements and use them often. E.g. a button can be set up as a default button without any text in it. You then use this button all the time and you then write „Exit“ or „OK“ or whatever into the button using the routine LCD\_Print.
- b) Use only graphical data with indexed colors and reduce the number of colors if possible.
- c) Use monochrome graphics if possible and set up a routine which shows them in the wanted two colors (does not have to be black and white, the same graphical element can also be red and white at one time and then green and white later).
- d) Use the compressing feature of our tool GLCD\_Convert and the decompression algorithm of our software. You may then compress data easy by 50% and more. Hint: The less colors you are using in your graphics the better usually the compression works as the chance is larger that a color repeats.
- e) The ATmega beside its flash memory for your program and data, also has a Eeprom with 4 Kbytes size which can also be used for often used data such as font data or graphics data.

## **Bascom does not compile correctly anymore after I inserted a lot of data statements**

Bascom had a bug in older versionn if you placed too many data statements. It was not possible to insert a lot of data statements without creating errors during compilation. Our first advise: updates are always free with Bascom (for registered users). Go to [www.mcselec.com](http://www.mcselec.com), log in with your user data and download the latest release.

But there is also a workaround: Just place a label after approx. each hundred lines of data statements. Just insert a new line and set a label (`Test1: etc...`). You will never use this label – probably it helps Bascom structuring the data.

## **The enclosed C-Compiler gives compile errors when I try to compile your sample code**

Please use for a first test only the make file we ship on our CD. There is nothing special with this makefile, but we know it works.

## **I cannot program the code to my ATmega module with the C-compiler on the CD.**

Please use for a first test only the make file we ship on our CD. This is preconfigured for a parallel programming interface (as our item E-H003a, which is STK-200 compatible). If you want to use our serial or USB programmer, you need to use the other makefile on our CD instead. Then:

- 1) rename the file *Makefile* to *Makefile-parallel*
- 2) rename the file *Makefile-seriell* to *Makefile*

Both files are identical but the information of the used programmer. If these are not connected to LPT1 (parallel) or COM1 (USB / serial) you need to change this port to your configuration at the file makefile.

## **Writing to the display takes a relatively long time**

Well, a lot of data needs to be transferred to the display. You will not be able to watch a movie at this display but 5-10 frames per second will be possible, depending on your microcontroller and the used software. One technique to speed up the general output is using hardware SPI only, another is updating only the part of the display which changed. It might also make sense to rewrite parts of the code to assembler if you really need to speed things up.

Then another request: Just this manual took us more than 100 hours to write (and another 20 hours to translate into English). After this long time working on such a project one often does not see even obvious faults or inaccuracies anymore.

Please inform us:

- If we have confused you with statements in this programming manual (even if this was cleared to you perhaps later)
- If we used a totally wrong translation somewhere
- If there are questions left unanswered
- If there are wrong or ambiguous statements
- If you think that we did leave out important aspects

We then will integrate these into the further development of the manual and also provide you with a copy.

**By the way: The software also will be developed further. If you have ideas for further developments, tell us and it might make it into the code.**

**Thank you very much in advance!**

## Contact:

Speed IT up  
Inhaber Peter Küsters  
Wekeln 39  
47877 Willich  
Germany  
Telefon: +49 (21 54) 88 27 5-10  
Telefax: +49 (21 54) 88 27 5-22

More informations: [www.display3000.com](http://www.display3000.com)

Author and copyright of manual informations: Peter Kuesters

**This document is copyright protected. It is not permitted to change any part of it. It is not permitted to publish it in any way, to make it available as a download or to pass it to other people. Offenses are pursued.**