

AN0047: Interfacing Graphical Displays



This application note introduces the software examples, libraries, documentation, and software tools available for interfacing to graphical displays with EFM32 microcontrollers.

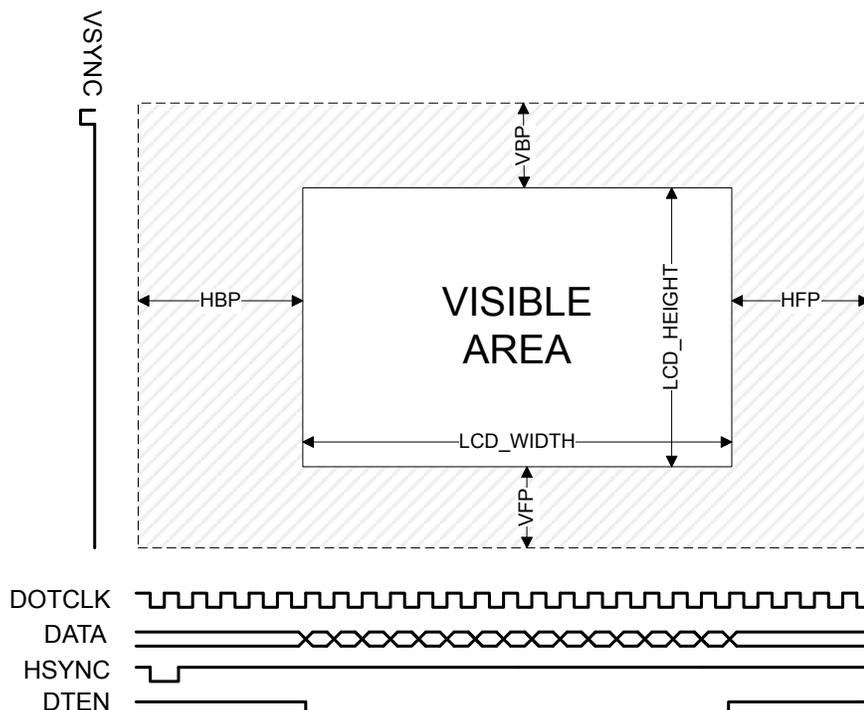
Following a brief description of the various MCU-to-display interfaces available, two key approaches are discussed in depth. System design considerations are described and their impacts to performance and complexity/cost are identified. Key concerns are explained and approaches that mitigate those risks are presented.

SEGGER's emWin Graphical Library is introduced and the reader is shown how this free benefit simplifies the rapid development of display-enabled solutions in Simplicity Studio.

Using the QVGA TFT-LCD on the Giant Gecko Development Kit (EFM32GG-DK3750), this document and the software examples it describes demonstrate how to drive the display in 8080 mode and in RGB mode using the integrated Direct Drive feature.

KEY POINTS

- Many options exist for interfacing graphical displays to MCUs.
- Understanding the performance impacts and resource requirements of each option empowers you to optimize your interface configuration for your system.
- The EFM32 MCUs provide additional features that enhance performance and reduce system overhead in many cases.
- Careful configuration can provide a flicker-free solution with no tearing.
- Theory is put into practice with multiple software examples using the EFM32GG Dev Kit.



1. Introduction

1.1 Graphical Displays

Graphical displays give more flexibility in creating a user interface, compared to segmented or character displays. However, they can be more complex to operate and they do require more CPU time to update than their simpler counterparts.

1.2 Display Interface

Microcontrollers can interface to a display controller in multiple ways. These options can be grouped into 4 main categories:

- Memory mapped — In a memory mapped configuration, the display controller is connected to both address and data lines of the MCU and both registers and video memory can be written to directly. This is the fastest type of configuration, but it also requires the most pins.
- Parallel interface — A parallel interface configuration usually has a full data width, but no address bus. To send commands or addresses to the controller the display often has a C/D pin (Command/Data) to select if the current value on the input is a command (or register address) or data. This C/D pin can also be called D/I or RS.
- Serial interface — A serial interface is the slowest form, but it requires very few pins. A SPI configuration can use as few as 3 or 4 pins, and a I2C bus can be implemented with only 2 pins. Since all pixel data has to be sent over a serial line, it takes many clock cycles to update the whole display. This configuration can make for cheap system solutions, but is normally not suitable for animations.
- RGB interface — An RGB interface is a special kind of parallel interface. This interface works for displays without a frame buffer. The MCU is responsible for updating the display manually, by providing both pixel data and timing signals.

This application note discusses two of the above options: a specific parallel interface known as the 'Intel 8080', and the RGB interface.

1.3 Graphical Library

To create a useful GUI, it is helpful to have a graphical library. A graphical library provides the software developer with functions to draw shapes, text, and images, and takes care of drawing operations like alpha blending and anti-aliasing.

Silicon Labs provides the emWin Graphical Library from SEGGER for free through Simplicity Studio. In addition to the standard drawing operations, this library also provides its own window manager and includes support for touch inputs, cursors, and skinnable widgets. It also has several PC tools available, including a bitmap converter and font converter. These tools produce C-files which can easily be compiled into your application.

For more information about emWin, please consult the emWin user manual. It can be found in the Simplicity Studio directory under `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\doc` (default path on Windows OS). The PC tools can be found in `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\exe`.

Note: the pre-compiled emWin distribution provided through Simplicity Studio includes a subset of the display drivers available for use within emWin. The available drivers are described in the emWin user manual, and also at <http://www.segger.com/emwin-display-drivers.html>. Header files for the display drivers included with the Simplicity Studio distribution are found in `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\inc`. Support for additional display drivers can be added to the Simplicity Studio emWin library by purchasing emWin GUIDRV source code from SEGGER - see <http://www.segger.com/emwin-source-code-upgrade.html?p=212> for more details.

2. Display Considerations

This section will introduce some concepts, challenges, and requirements which are important to be aware of when designing an application with a graphical display. Later it will be shown how to deal with these topics using the EFM32.

2.1 Memory and Frame Buffer

The frame buffer is the memory location holding the pixel data which are currently displayed. The display controller needs to read this memory every update of the display.

The controllers in some "smart" displays contain their own local frame buffer. On displays without an internal frame buffer, the frame buffer has to be stored in RAM. It is then the MCU's responsibility to update the display from this frame buffer.

If the display resolution and color depth are low enough, the frame buffer can be stored in internal RAM. However, this is likely not possible for larger displays or those with higher color depth. For example, a QVGA (320x240) display with 16-bit color depth requires $320 * 240 * 2 / 1024 = 150$ kB of RAM for one frame. Take the case where an EFM32GG990F1024 is used. This device has 128 kB of internal RAM - not enough to store the 150 kB frame buffer. In this case an external memory block is required to store the frame buffer.

2.2 Frame Rate

To calculate the frame rate, we need the pixel clock frequency, the size of the display (in pixels) and porch intervals. The size of the porch intervals in the equations below are expressed in pixel clock cycles.

$$N_{\text{LINE}} = \text{HBP} + \text{WIDTH} + \text{HFP}$$

$$N_{\text{FRAME}} = (\text{VBP} + \text{HEIGHT} + \text{VFP}) \times N_{\text{LINE}}$$

The equation above gives the number of pixel clock cycles needed to update one full frame. To calculate the total frame rate (FPS) we can then use the pixel clock frequency (F_{PXCLK}) as follows:

$$\text{FPS} = \frac{F_{\text{PXCLK}}}{N_{\text{FRAME}}}$$

2.3 Bus Access

When both an external frame buffer (RAM) and display controller is connected to the EBI bus, bus access can become the bottleneck of the system. The pixel data first has to be written to RAM over EBI and then transferred from RAM to the display over the same bus.

In RGB mode it is possible to take advantage of the porch intervals to optimize this (see [Timing and Porch Intervals](#)). During these intervals no pixel data is sent to the display, so this is a good time to write pixels to RAM.

2.4 Flickering and Tearing Effects

Flickering and tearing are visual artifacts that reduce the overall user impression of an application.

When drawing a frame, it is common to first fill the background with a color or image and then draw text, buttons, or other user interface elements on top. In this case, if the drawing is done directly to the display (i.e. in systems without a frame buffer) there will be a small time window where only the background is visible to the user. When the frame is redrawn several times per second it will look like the UI elements are blinking or flickering.

Tearing occurs if the display controller is in the middle of displaying a frame and then suddenly switches to the next frame. The top of the display will then show the old frame, while the bottom shows the new one. To the user it will appear to be visible lines across the screen between the two images. An approach to avoid tearing is discussed in a later section.

2.5 Multiple Buffering

Multiple buffering is a technique to avoid flickering. When multiple buffering is used, the MCU is drawing to one frame (the back buffer), while the display controller is showing the other (the front buffer).

There are two schemes: double or triple buffering. The advantage of triple buffering is that the CPU never has to wait for the display controller to finish. In the double buffering scheme, when the CPU has finished drawing a frame to the back buffer, it then has to wait for the display controller to switch buffers before it can start writing to the other buffer. In triple buffering, the CPU can just start drawing on the next buffer and flag the finished buffer as pending.

The drawback to using multiple buffering is the increased memory usage. For triple buffering, 3 complete frames have to be stored in memory. Using the QVGA example from before, the requirement now becomes 450 kB of RAM for storing the frame buffers.

Figure 2.1 Triple Buffering on page 3 shows an example of triple buffering. The current visible frame buffer is frame 0. This frame is currently being read by the display controller, and shown on the display. The CPU is finished drawing the entire frame 1, has flagged it as pending, and is currently drawing to frame 2. On the next VSYNC signal, the display controller will start reading from the memory address at frame 1.

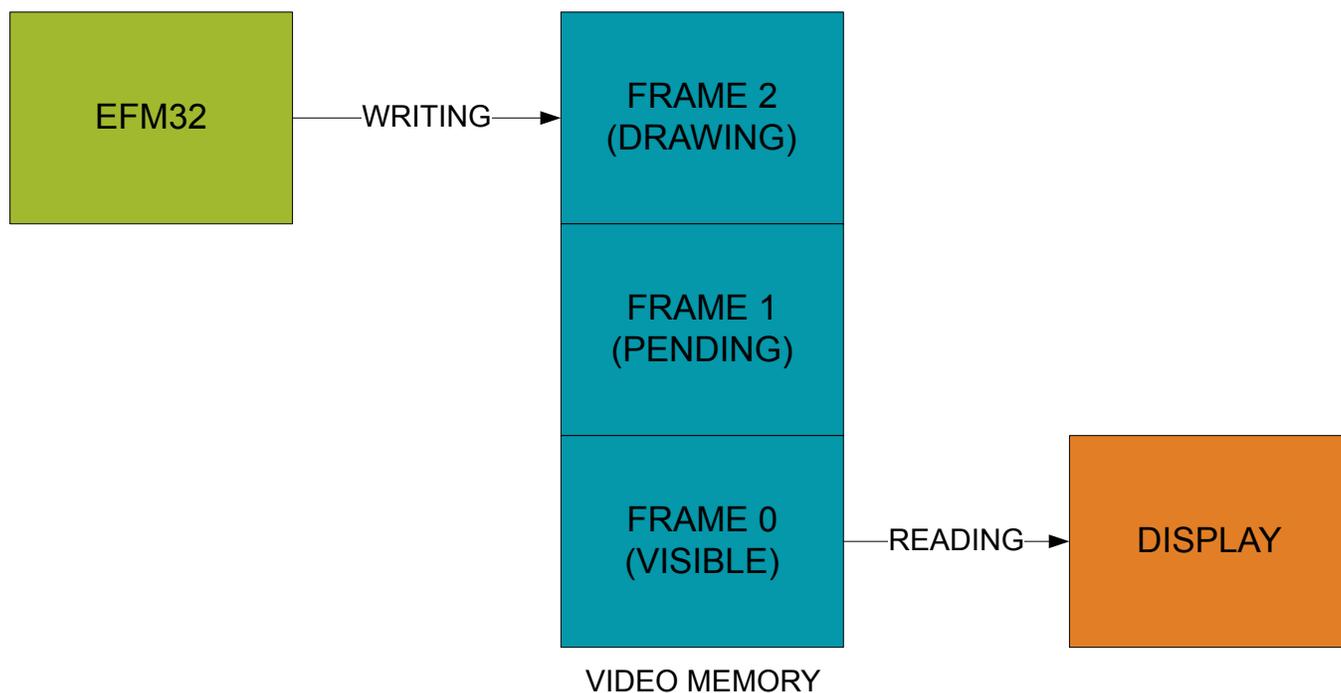


Figure 2.1. Triple Buffering

3. Driving a Display With the EFM32

This chapter will show how to drive a display from a EFM32 microcontroller using the External Bus Interface (EBI).

The EBI provides access to parallel interface devices including RAM, FLASH, and parallel displays. Devices connected to the EBI are memory mapped to the EFM32's memory space. All timing and control signals are controlled by hardware. Access to the devices from software are simple read and write instructions. The EBI also has an integrated Direct Drive feature. This feature is specifically designed to drive graphical displays in the RGB mode. For more information about EBI, consult the Reference Manual for your EFM32 device.

Note: Not all EFM32 devices with EBI have the Direct Drive feature. Please consult the EBI chapter in the Reference Manual to see if your device supports this feature.

3.1 Intel 8080 Mode

A common interface standard is the 'Intel 8080' mode. This mode should be used if the display has an integrated display controller and frame buffer. EBI write operations from the MCU update the frame buffer in the display. The display controller handles all the display updates from this frame buffer. An advantage of this mode, is that the MCU only needs to perform EBI write operations whenever the frame changes.

In addition to the data lines, the 8080 mode uses the following control signals: RE (read strobe), WE (write strobe), CS (chip select) and D/C (data/command). The polarity and timing of these signals can all be controlled by EBI. Refer to the datasheet of the display controller on how to set up these signals. The D/C signal can be thought of like a 1-bit address line, and allows software to program registers in the display controller.

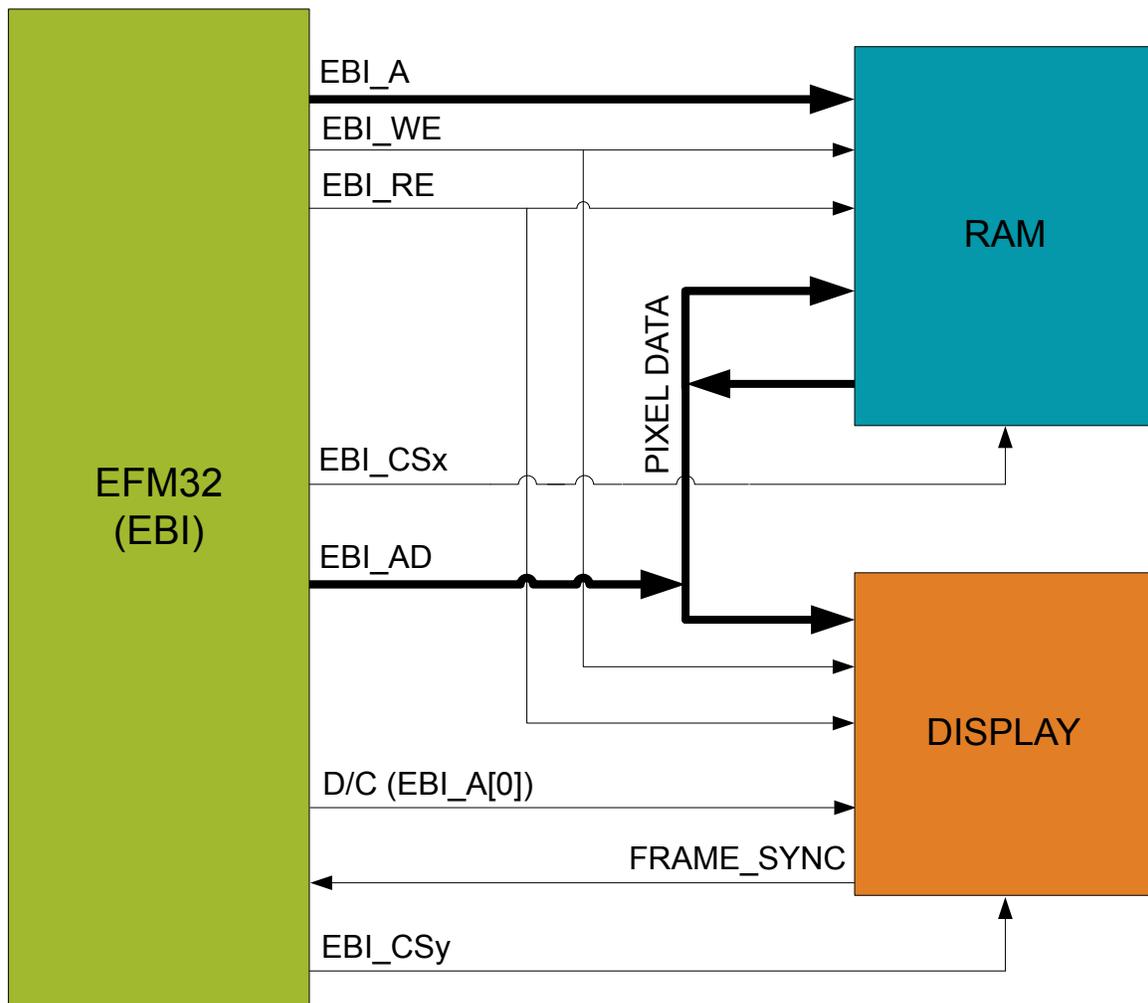


Figure 3.1. 8080 mode connection diagram

3.1.1 Avoiding Tearing

To avoid tearing in 8080 mode, the part of the frame buffer currently being written to the display by the controller should not be changed by the MCU. If the MCU is able to update the frame buffer faster than the controller is updating the display, a frame buffer update from the MCU should begin before the display controller starts the next display update, i.e. the VSYNC period. In this scenario, the display controller will always read a pixel from the 'new' frame buffer content, because MCU writes to the frame buffer are progressing through the frame at a faster rate than the display controller reads are progressing through it.

On the other hand, if the display controller can update the display faster than the EFM32 can write to the frame buffer, the MCU should begin updating the frame buffer with new content just after the display controller starts the next display update. This ensures that the display controller is always reading 'old' data, and as long as the MCU can finish the frame buffer update within 2 display updates there will be no tearing. See [Figure 3.2 Frame update in 8080 mode on page 6](#) for a visual depiction of these concepts.

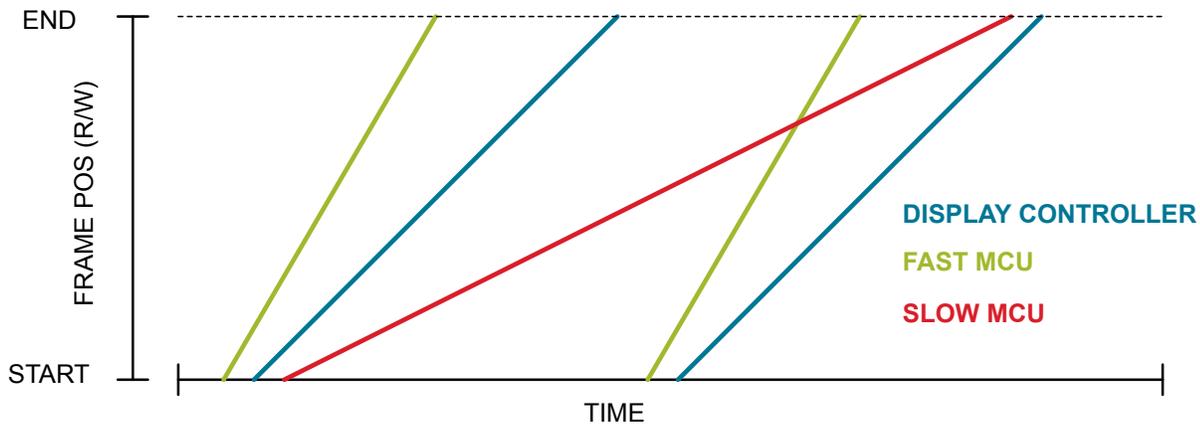


Figure 3.2. Frame update in 8080 mode

The display controller will normally have an output signal to help synchronize frame buffer updates. Refer to the data sheet for the display controller on how to respond to this signal.

3.2 RGB Mode With Direct Drive

Another common way to drive displays is the RGB mode. This mode should be used if the display does not have its own frame buffer. In this mode the MCU is responsible for sending pixels to the display directly. Note that the term 'RGB mode' used in this context has nothing to do with color displays. It refers instead to the type of display interface where the MCU itself is in charge of the timing signals (HSYNC, VSYNC, DTEN, DOTCLK).

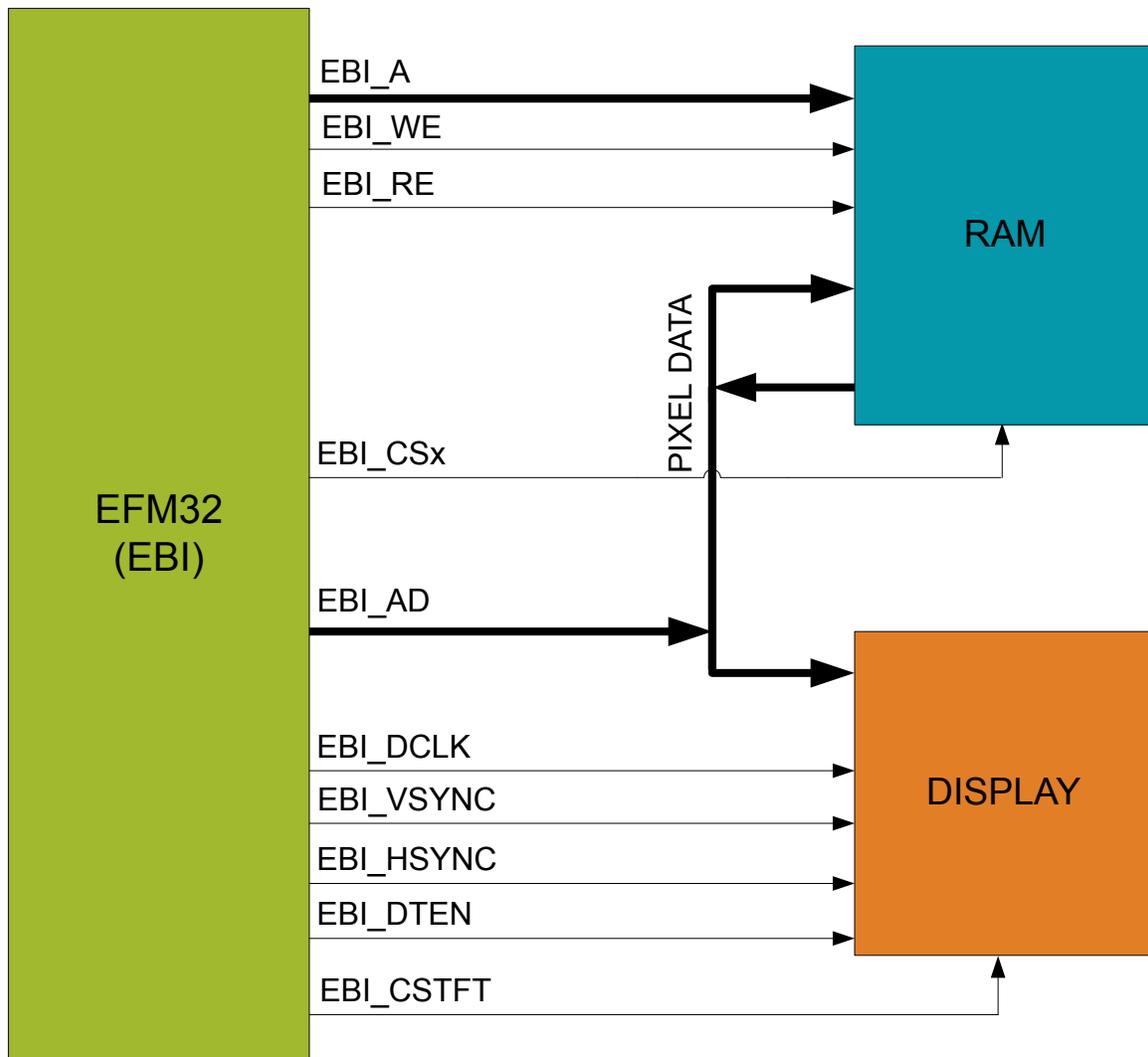


Figure 3.3. RGB mode with Direct Drive connection diagram

Both pixel data and timing signals for the display have to be supplied by the MCU. If this must be controlled by software, the CPU load can be significant. This is especially true for display panels with a large resolution. For this reason, the Direct Drive feature has been developed to drive displays autonomously. In this mode Direct Drive takes the role of the display controller, reading the frame buffer from memory and generating the necessary timing signals to transfer the pixels to the display.

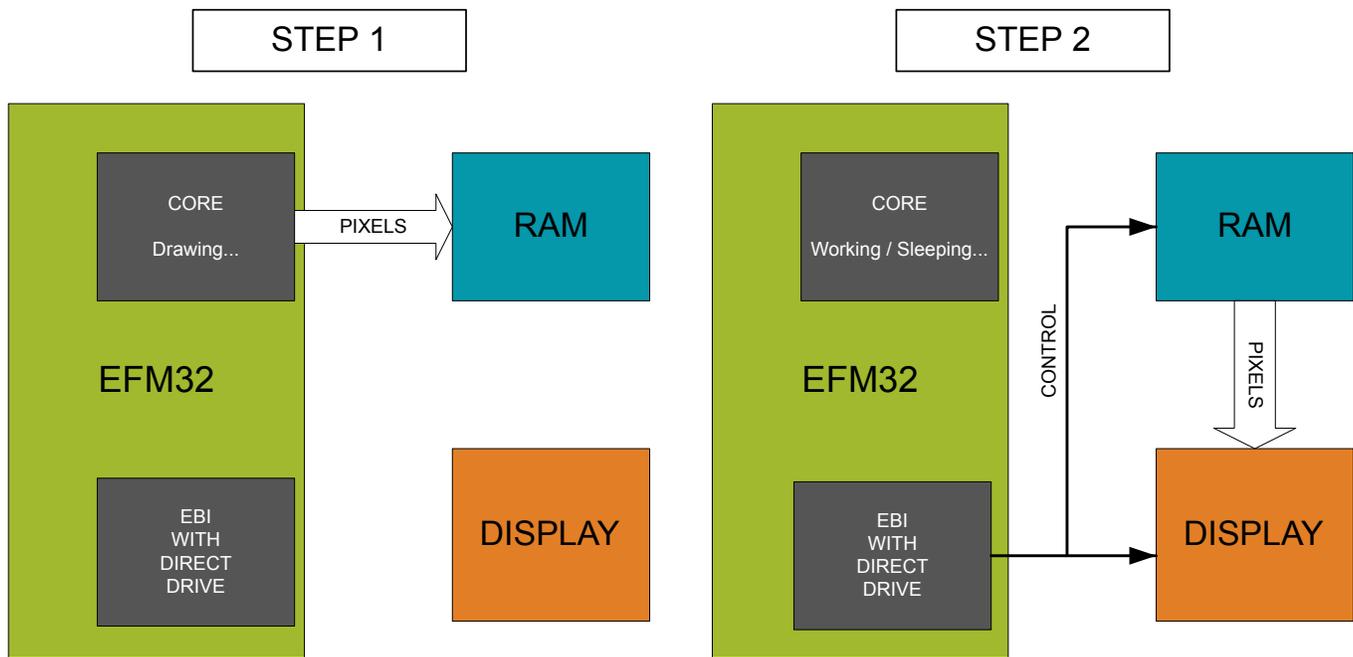


Figure 3.4. Direct Drive

When using Direct Drive, drawing to the display is divided into two steps. In step 1 the CPU calculates the image to be displayed and writes this to a frame buffer in memory. In step 2 Direct Drive uses the frame buffer to update the display. Note that these steps are performed in parallel: Direct Drive does not wait for the CPU to finish a frame - it continuously updates the display from the frame buffer in memory.

The CPU is completely decoupled from Step 2, so the dominant concern for CPU utilization is how fast it can draw to the frame buffer in RAM. After the CPU has completed a frame and written it to RAM, it is free to do other tasks, like calculating the next frame, or even going to sleep if no more frames are needed.

It is possible to use either internal or external RAM for the frame buffer with Direct Drive. External RAM is usually preferred for a number of reasons. On one hand, internal memory is limited and might even be too small to hold the frame buffer. Another consideration is that when using internal memory, this memory has to be accessed often by both Direct Drive and the CPU. This can cause a number of wait states on the CPU and slow down the application. When running Direct Drive with external memory, the CPU's working memory and frame buffer is completely decoupled.

Note: The alpha blending and masking features in Direct Drive can not be used together with emWin and are not discussed in this application note. To see how they can be used, refer to the `scroller` example for the EFM32GG-DK3750.

3.2.1 Multiple Buffering

Multiple buffering is easily implemented with Direct Drive. The hardware can read the frame buffer from any part of the available memory. So, to change the visible buffer, software only needs to change the value in the `TFT_FRAMEBASE` register. This register contains the start address of the visible frame buffer. To avoid tearing this should be done during the vertical porch interval (before the start of a new frame). Direct Drive can trigger an interrupt routine automatically during the VSYNC phase.

3.2.2 Timing and Porch Intervals

When updating the display, there is some time required between lines (and frames), when no pixels are touched. These intervals are called "porch" intervals. The Horizontal Back Porch (HBP) and Horizontal Front Porch (HFP) is the time before and after each line. Similarly, there are intervals specified before and after each frame called Vertical Back Porch (VBP) and Vertical Front Porch (VFP). An integrated display controller will control these intervals itself. However, if the display is run in RGB mode the MCU is responsible for their control. Direct Drive manages these intervals automatically.

The RGB interface has 4 control/timing signals in addition to the data lines. The VSYNC (Vertical Sync) signal is asserted before each frame. Similarly, the HSYNC (Horizontal Sync) signal is asserted before each line. The DOTCLK (Dot Clock) is a continuously toggling clock. The last signal, DTEN (Data Enable), is asserted whenever the controller should receive data. DTEN together with VSYNC and HSYNC controls the porch intervals. DTEN is deasserted for the entire duration of these intervals.

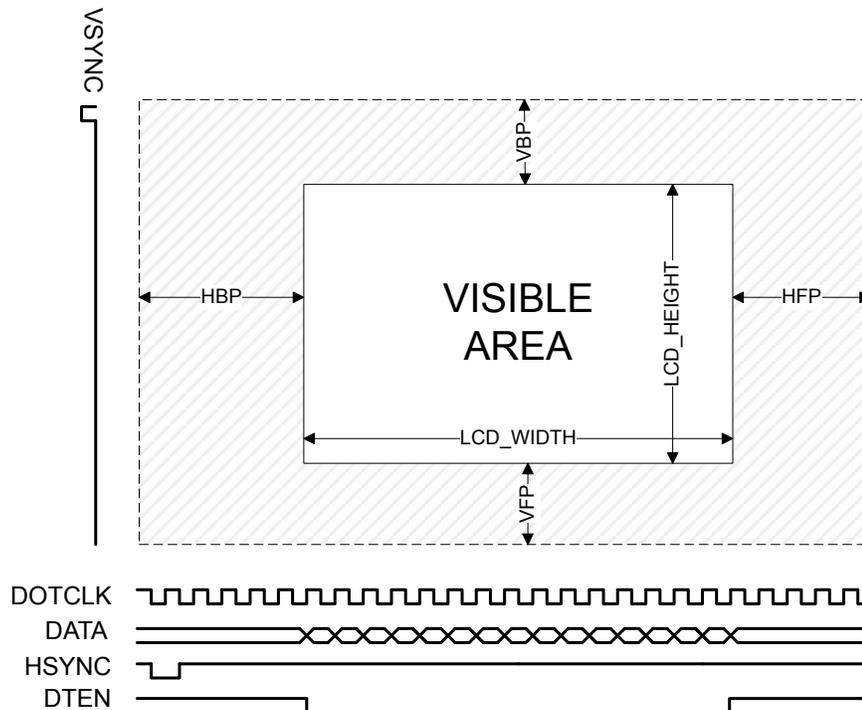


Figure 3.5. Timing Diagram RGB Mode

4. Software Configuration

This chapter will explain how to set up the software to drive a display in either 8080 mode or RGB mode with Direct Drive and perform drawing with emWin.

In 8080 mode emWin can draw pixels directly to the display using an integrated driver. In RGB mode the pixels are output to a frame buffer in external memory and Direct Drive takes care of updating the display from this frame buffer. It is also possible to use 8080 mode without an integrated emWin driver. In this configuration the pixels are output to external memory and DMA is used to transfer the frame buffer to the display. The software examples in this application note demonstrate all of these cases.

4.1 emWin

The emWin library contains 3 configuration files (discussed below) that need to be modified by the application.

The most important file is `LCDConf.c`. In this file, the following has to be configured:

- The emWin display driver. In this application note we use a linear driver, which will draw directly to a memory mapped video memory.
- A color conversion. This needs to correspond with the color format that the display is expecting. Internally emWin always uses a 32-bit ARGB format.
- The video memory start address. This is required by the linear driver to know where to start drawing.
- The display size. The linear driver needs to know the width and height (in pixels) of the display.

The file `GUI_X.c` configures system specific timer and delay. The supplied example should fit most applications.

The emWin library itself is initialized in `GUIConf.c`. emWin requires some memory for internal book keeping – that memory is allocated in this file. The size of this required memory depends on the features used and how many GUI objects are created. In general, a few kB should be enough.

Please refer to the emWin user manual (default location: `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\doc`) for a more detailed discussion of these files.

4.1.1 PC Tools

emWin also ships with a number of PC tools. These tools all generate C-files that can be compiled into your application. The tools can be found in the Simplicity Studio folder under `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\exe`.

- The bitmap converter (`BmpCvt.exe`) generates C-files from normal image files (BMP, GIF and PNG are supported).
- The font converter (`SetupFontCvtDemo_V528.exe`) can generate fonts for use with emWin. The input can be any normal windows font.
- The GUI Builder (`GUIBuilder.exe`) is a WYSIWYG tool for creating windows, menus etc.
- The binary to C converter (`Bin2C.exe`) converts arbitrary binary files into C arrays.
- The unicode to C converter (`U2C.exe`) converts text files in UTF-8 format into C string arrays.

4.2 8080 Mode With DMA

In the 8080 mode, DMA is used to transfer the frame from RAM to the display controller.

The scatter-gatherer mode is used to allow a full frame to be transferred in one DMA cycle. In this mode, multiple alternate descriptors are set up to transfer parts of the frame buffer sequentially. The maximum element size for one descriptor is 1024 elements. For a QVGA (320x240) display, this means a total of $320 * 240 / 1024 = 75$ descriptors are needed.

The destination address will depend on which EBI bank the display is mapped to, and which address pin is used for D/C. As an example, assume the display is mapped to EBI Bank 1 (address `0x84000000`), `EBI_A[0]` is connected to D/C and a 1 on this pin means 'data mode'. Furthermore if the address mode is 16-bit, all addresses are shifted one bit to the right (see the EBI Chapter, 16-bit address mode in the Reference Manual for an explanation of this). The DMA is in this case configured to write to address `0x84000002`.

The source increment is set to the same value as the pixel size (e.g. 2 byte for a 16 bits per pixel display). The destination increment is zero because we are always writing to the same address.

The timing parameters for the display controller need to be configured for the EBI bank that the controller is mapped to. The software example indicates how this is done for the display on the EFM32GG-DK3750 Development Kit. Additional guidance is provided in the application note *AN0034: External Bus Interface*.

4.3 Direct Drive

Direct Drive uses the RGB interface and automatically feeds the display with pixels from the frame buffer in memory.

Direct Drive automatically generates the control/timing signals (HSYNC, VSYNC, DTEN, DOTCLK) for the display and also controls the external memory (if needed) to feed the display with data. The timing and polarity of the control signals needs to be configured when initializing Direct Drive. Please refer to the Reference Manual for a complete description of these features.

emlib contains helper functions to ease Direct Drive configuration. The function `EBI_TFTInit()` will both configure and start Direct Drive. It takes a single parameter - an initialization structure - that must be filled beforehand. The software examples in this application note demonstrate this process.

Some of the most important parameters of Direct Drive are summarized here:

- Drive Mode—specifies whether Direct Drive should read from internal or external memory
- EBI Bank—specifies which EBI bank to use for memory, in case drive mode is set to external
- Frame Buffer Address Offset—the offset relative to EBI bank base address
- Data Width—width of pixel data, can be either byte or halfword
- Bus Interleave mode—controls interleaving of EBI bus
- Dotclock Period—prescaling of the dot clock
- Porch Intervals—control durations in which Direct Drive will leave the bus idle for each frame

The interleave flag specifies how interleaving of the EBI bus is performed when using Direct Drive. It has three possible values. "Unlimited" allows for interleaved bus access at any time. This is the fastest option (uses the most of the available bus time), but can cause visible jitter on the display if the CPU causes too much bus traffic during frame updates. "Once per Dot Clock" only allows one bus transfer to be interleaved per dot clock. The last option is "Porch only". With this option bus accesses are only allowed during porch intervals.

The porch parameters should be within the limits given by the display controller. However, it can be desirable to increase these intervals to give the CPU more bus access time.

4.4 External Memory Frame Buffer

If using external memory for the frame buffer, the timing and control signals for this memory also has to be programmed into the EBI registers.

The software examples show how this is configured on the EFM32GG-DK3750 Development Kit. For a more general description, please refer to the EBI Chapter in the Reference Manual for your device.

5. Software Examples

These examples show how to set up emWin and drive the QVGA TFT-LCD on-board the EFM32GG-DK3750 Development Kit. On this kit, the display controller is connected with both an 8080 and an RGB interface to the EFM32 Giant Gecko EBI. Also connected to the EBI bus is a 4 MB PSRAM.

Note that these examples contain several application specific source files which are common for all the examples. They can be found in the `\common` folder. This includes the emWin configuration files and `ebi_conf.c`, which configures the EBI timing parameters for PSRAM and the display controller in 8080 mode.

It is possible that the display on the EFM32GG-DK3750 will hang when switching between the two driving modes (8080 and RGB). If this happens, toggle the AEM button twice to get the display working again.

Several more examples demonstrating the TFT and emWin are available within Simplicity Studio: `guidemo`, `graphxy`, `reversi`, `realtime`, or `radialmenu`. In Simplicity Studio v4, these examples can be found in the "Software Examples" section under the "Getting Started" group when an EFM32 Giant Gecko Development Kit is the selected context. For users of Simplicity Studio v3 and prior, these examples are located under the [**Software Examples**] tile for the EFM32GG-DK3750.

5.1 8080 Mode With Integrated emWin Driver

This example uses a driver written for the SSD2119 display controller. This is the controller present on the TFT on the Development Kit. The driver interfaces directly with emWin so drawing commands translate into commands for the display controller to set the touched pixels.

The driver has to be fitted to the hardware layout. In this example the display is mapped to EBI bank 1 (address `0x84000000`) and `EBI_A[0]` is connected to D/C. Since the address mode is 16 all bits are shifted 1 bit to the right (see the EBI Chapter, 16-bit address mode in the Reference Manual for an explanation of this). The driver is therefore configured to write commands to `0x84000000` and pixels to `0x84000002`.

Segger provides drivers for many common displays, see the Segger website if your controller is supported: <http://www.segger.com/emwin-display-drivers.html>.

5.2 8080 Mode With DMA

This example uses emWin to draw a frame to PSRAM and then DMA to copy the entire frame to the display controller, over the 8080 interface.

The PSRAM block is mapped to EBI bank 2. Referring to the EFM32 Giant Gecko user manual, this corresponds to the memory range `0x88000000 – 0x8bffffff`.

emWin is configured to start drawing the frame at memory location `0x88000000`. This is done in the `LCD_X_Config()` function found in `LCDConf.c`. The linear 16-bit driver and an RGB565 color conversion are also selected in this function.

After the frame is drawn to memory it has to be transferred to the display controller. The display controller is mapped to EBI bank 1. As discussed in [Software Configuration](#), the DMA is configured to read the frame and write it to `0x84000002`.

The timing and control signals of both PSRAM and the SSD2119 display controller are configured in the `ebi_conf.c` file.

5.3 RGB Mode With Direct Drive

In this example, the display is driven in a RGB mode with Direct Drive.

emWin is configured the same way as for the 8080 example and the Direct Drive feature is also initialized and enabled.

In this mode, there is no need for software to specify when to send the frame buffer to the display. Direct Drive autonomously handles the display updates. After the initialization, the CPU never has to interact with Direct Drive again.

5.4 Multiple Buffering With Direct Drive

When only using one frame buffer with Direct Drive, both tearing and flickering can occur. If the application requires animations, multiple buffering should be added. This example contains a modified `LCDConf.c` file that adds the required code to use multi buffering with emWin and Direct Drive.

The first task is to tell emWin how many buffers to use. This is done with `GUI_MULTIBUF_Config(NUM_BUFFERS)`.

The second modification is to implement the `LCD_X_SHOWBUFFER` callback. This callback is received every time emWin is done drawing a frame, i.e. when the buffers should be flipped. It is possible to flip the buffers immediately in this routine. However, by doing so immediately the flipping is not synced with the display updates, so tearing will still occur. A better method is to use an interrupt routine that flips the buffers on the VSYNC signal.

To draw with multiple buffering, surround the drawing calls with `GUI_MULTIBUF_Begin()` and `GUI_MULTIBUF_End()`. This tells emWin when a frame is finished drawing.

One additional optimization is done with respect to multi buffering. The default behavior of emWin is to copy the entire front buffer to the back buffer, before each buffer flip. This creates much extra bus traffic, and since the drawing code overwrites the entire background each frame, it is completely unnecessary.

emWin specifies a method to implement a custom frame copy operation. Using the function `LCD_SetDevFunc()` it is possible to specify a function that will manage the frame copy, instead of the default emWin behavior. In the example, this function is implemented with an empty body to disable the frame copy.

5.5 Window Manager

This example illustrates the emWin window manager. In the window manager each window is responsible for drawing itself. The windows do this by specifying a callback routine and listening for the `WM_PAINT` message.

Multi buffering support is enabled for this example too, but the buffer copy optimization is not implemented as the window manager relies on the buffer copy for its drawing algorithm.

5.6 More emWin Examples

More examples for emWin can be found in the Simplicity Studio installation directory under `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\examples`. Most of these examples can easily be viewed by copy-pasting directly into one of these example projects.

6. Further Reading

Documentation for emWin can be found in the emWin User Manual, located in the `C:\SiliconLabs\SimplicityStudio\v3\developer\sdk\efm32\v2\reptile\emwin\doc` folder under Simplicity Studio, or on the SEGGER website: <http://www.segger.com/>.

7. Revision History

7.1 Revision 1.07

2016-12-22

Clarified connection diagrams.

Content edits for clarity.

Added emWin display controller support details.

Updated format.

7.2 Revision 1.06

2014-05-07

Updated example code to CMSIS 3.20.5.

Changed source code license to Silicon Labs license.

Added project files for Simplicity IDE.

Removed makefiles for Sourcery CodeBench Lite.

7.3 Revision 1.05

2013-10-14

New cover layout.

7.4 Revision 1.04

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

7.5 Revision 1.03

2012-11-12

Adapted software projects to new kit-driver and bsp structure.

7.6 Revision 1.02

2012-10-29

Included integrated emWin driver example.

7.7 Revision 1.01

2012-08-13

Updated file paths for display driver.

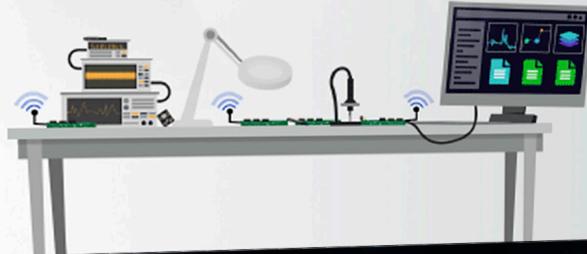
7.8 Revision 1.00

2012-08-01

Initial revision.

Silicon Labs

Simplicity Studio™4



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!



IoT Portfolio
www.silabs.com/IoT



SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality



Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

<http://www.silabs.com>