EFM32®

... the world's most energy friendly microcontrollers

EFM32 Getting Started

AN0009 - Application Note

Introduction

This application note is an introduction to development using the EFM32 devices from Energy Micro. It is intended as a hands-on tutorial and it is recommended to work through the tasks in the order they are presented. Even though prior EFM32 knowledge is not required, basic programming and electronics skills are needed to complete the tasks. The examples can be run on the following kits:

- EFM32 Gecko Development Kit (EFM32-Gxxx-DK)
- EFM32 Gecko Starter Kit (EFM32-Gxxx-STK)
- EFM32 Tiny Gecko Starter Kit (EFM32TG-STK3300)
- EFM32 Giant Gecko Starter Kit (EFM32GG-STK3700)
- EFM32 Zero Gecko Starter Kit (EFM32ZG-STK3200)

Things you will learn:

- Basic register operation
- Using emlib functions
- Blinking LEDs and reading buttons
- LCD controller
- Energy Modes
- Real-Time Counter operation

This application note includes:

- This PDF document
- Source files (zip)
  - Example C-code
  - IDE workspaces with example projects

ARM ZERO TINY GECKO LEOPARD GIANT WONDER

SILICON LABS

1 Introduction

1.1 How to use this application note

The source code for this application note is placed in individual folders named after the kit the different examples are intended for:

- EFM32-Gxxx-DK
- EFM32-Gxxx-STK
- EFM32TG-STK3300
- EFM32GG-STK3700
- EFM32ZG-STK3700

Projects for each of the supported IDEs are found in separate folders (iar, arm, etc.). The IAR projects are also collected in one common workspace called **efm32.eww**, whereas the ARM (Keil) projects must be handled separately. Since the projects are slightly different for the various kits, make sure that you open the project that is prefixed with the name of the kit you are using. The code examples in this application note are not complete, and the reader is required to fill in small pieces of code through out the exercises. If you get stuck, a completed code file (postfixed with ***_solution.c**) exists for each examples.

1.2 Prerequisites

The examples in this application note require that you have one of the supported EFM32 kits at hand. Before you start working on this tutorial you should make sure you have done the following steps:

- Installed the latest Segger J-Link drivers(**www.segger.com/cms/jlink-software.html**)
- Installed Simplicity Studio (**www.energymicro.com/software/simplicity-studio**)
- Installed an IDE that supports the EFM32 device you have on your kit

In Simplicity Studio you should also make sure that you have all the available packages installed and up to date. You can ensure this by clicking on **Add/Remove** and pressing the **Install All** button. You will also need to click on **Updates** and press the **Update All** button.

2 Register Operation

This chapter explains the basics of how to write C-code for the EFM32 devices, using the defines and library functions supplied in the **CMSIS** and **emlib** software libraries.

2.1 Address

The EFM32 consists of several different types of peripherals (CMU, RTC, ADCn...). Some peripherals in the EFM32 exist only as one instance, like the Clock Management Unit (CMU). Other peripherals like Timers (TIMERn) exist as several instances and the name is postfixed by a number (n) denoting the instance number. Usually, two instances of a peripheral are identical, but are placed in different regions of the memory map. However, some peripherals have a different feature set for each of the instances. E.g. USART0 can have an IrDA interface, while USART1 has not. Such differences will be explained in the device datasheet and the reference manual. The peripheral instances each have a dedicated address region which contains registers that can be accessed by read/write operations. The peripheral instances and memory regions are found in the device datasheets. The starting address of a peripheral instance is called the base address. The reference manual for the device series contains a complete description of the registers within each peripheral. The address for each register is given as an offset from the base address for the peripheral instance.

Register address calculated from base and offset

$$ADDRESS_{REGISTER} = BASE_{PERIPHERAL\_INSTANCE} + OFFSET_{PERIPHERAL} \quad (2.1)$$

2.2 Register description

The EFM32 devices use a 32-bit bus for write/read access to the peripherals, and each register in a peripheral contains 32 bits, numbered 0-31. Unused bits are marked as reserved and should not be modified. The bits used by the peripheral can either be single bits (e.g. DIFF bit in Figure 2.1 (p. 4)) or grouped together in bitfields (e.g. REFRSEL bitfield in Figure 2.1 (p. 4)). Each bitfield is described with the following attributes:

- Bit position
- Name
- Reset value
- Access type
- Description

Figure 2.1. Example register description

26.5.1 DACn_CTRL - Control Register

Offset	Bit Position																											
0x000	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset											0x0			0x0			0x0			0			0x0		0	0	0x1		0x0		0	0
Access											RW			RW			RW			RW			RW		RW	RW	RW		RW		RW	RW
Name											REFRSEL			PRESC			LPFFREQ			LPFEN			REFSEL		CH0PRESCRST	OUTENPRS	OUTMODE		CONVMODE		SINEMODE	DIFF

Bit	Name	Reset	Access	Description
31:22	*Reserved*	*To ensure compatibility with future devices, always write bits to 0.*		
21:20	REFRSEL	0x0	RW	**Refresh Interval Select**

Select refresh counter timeout value. A channel x will be refreshed with the interval set in this register if the REFREN bit in DACn_CHxCTRL is set.

Value	Mode	Description
0	8CYCLES	All channels with enabled refresh are refreshed every 8 prescaled cycles
1	16CYCLES	All channels with enabled refresh are refreshed every 16 prescaled cycles
2	32CYCLES	All channels with enabled refresh are refreshed every 32 prescaled cycles
3	64CYCLES	All channels with enabled refresh are refreshed every 64 prescaled cycles

2.3 Access types

Each register has a set access type for all of the bit fields within that register. The access types describes the reaction to read or write operation to the bit field. The different access types found for the registers in the EFM32 devices are found in Table 2.1 (p. 4)

Table 2.1. Register Access Types

Access Type	Description
R	Read only. Writes are ignored
RW	Readable and writable
RW1	Readable and writable. Only writes to 1 have effect
W1	Read value undefined. Only writes to 1 have effect
W	Write only. Read value undefined.
RWH	Readable, writable and updated by hardware

2.4 CMSIS and emlib

The Cortex Microcontroller Software Interface Standard (CMSIS) is a common coding standard for all ARM Cortex devices. The CMSIS library provided by Energy Micro contains header files, defines (for peripherals, registers and bitfields) and startup files for all EFM32 devices. In addition, CMSIS also includes functions which are common to all Cortex devices, like interrupt handling etc. Although it is possible to write to registers using hard coded address and data values, it is recommended to use the defines, to ensure portability and readability of the code.

To use these defines, we must include efm32.h in our c-file. This is a common header file for all EFM32 devices. Within this file the correct header file for the specific device is included according to the preprocessor symbols defined for your project.

To make programming of the EFM32 devices simpler, Energy Micro also provides a complete C-function library, called **emlib**, for all peripherals and core functions in the EFM32. These functions are found within the **efm32_xxx.c** (e.g. **efm32_rtc.c**) and **efm32_xxx.h** files in the **emlib** folder.

2.4.1 CMSIS Documentation

Complete Doxygen documentation for the EFM32 **CMSIS** library and **emlib** is available in **API Documentation** in **Simplicity Studio** and in the software section of the web page.

2.4.2 Peripheral structs

In the **emlib** header files, the register defines for each peripheral type are grouped in structs as defined in the example below.

Example 2.1. Peripheral struct

```
typedef struct
{
__IO uint32_t CTRL;
__I uint32_t  STATUS;
__IO uint32_t CH0CTRL;
__IO uint32_t CH1CTRL;
__IO uint32_t IEN;
__I uint32_t  IF;
__O uint32_t  IFS;
__O uint32_t  IFC;
__IO uint32_t CH0DATA;
__IO uint32_t CH1DATA;
__O uint32_t  COMBDATA;
__IO uint32_t CAL;
__IO uint32_t BIASPROG;
} DAC_TypeDef;
```

Together with the base address defines for the peripheral instance, writing to a register, CH0DATA, in the DAC0 peripheral instance, can then be done like this:

```
DAC0->CH0DATA = 100;
```

Together with the base address defines for the peripheral instance, writing to a register, CH0DATA, in the DAC0 peripheral instance, can then be done like this:

```
DAC0->CH0DATA = 100;
```

Reading a register can be done like this:

```
myVariable = DAC0->STATUS;
```

2.4.3 Bit field defines

Every EFM32 device has relevant bit fields defined for each peripheral. These are found within the **efm32xx_xxx.h** (e.g. **efm32tg_dac.h**) files and are automatically included with the appropriate **emlib** peripheral header file.

Example 2.2. Defines for REFRSEL bit field in DACn_CTRL.

```
#define _DAC_CTRL_REFRSEL_SHIFT          20
#define _DAC_CTRL_REFRSEL_MASK           0x300000UL
#define DAC_CTRL_REFRSEL_DEFAULT         (0x00000000UL << 20)
#define DAC_CTRL_REFRSEL_8CYCLES         (0x00000000UL << 20)
#define DAC_CTRL_REFRSEL_16CYCLES        (0x00000001UL << 20)
#define DAC_CTRL_REFRSEL_32CYCLES        (0x00000002UL << 20)
#define DAC_CTRL_REFRSEL_64CYCLES        (0x00000003UL << 20)
#define _DAC_CTRL_REFRSEL_DEFAULT        0x00000000UL
#define _DAC_CTRL_REFRSEL_8CYCLES        0x00000000UL
#define _DAC_CTRL_REFRSEL_16CYCLES       0x00000001UL
#define _DAC_CTRL_REFRSEL_32CYCLES       0x00000002UL
#define _DAC_CTRL_REFRSEL_64CYCLES       0x00000003UL
```

For every register field, shift, mask and default value bit fields are defined.

Example 2.3. Defines for LPFEN bit field in DACn_CTRL.

```
#define DAC_CTRL_LPFEN                   (0x1UL << 12)
#define _DAC_CTRL_LPFEN_SHIFT            12
#define _DAC_CTRL_LPFEN_MASK             0x1000UL
#define DAC_CTRL_LPFEN_DEFAULT           (0x00000000UL << 12)
#define _DAC_CTRL_LPFEN_DEFAULT          0x00000000UL
```

2.4.4 Register access examples

When setting a bit in a control register it is important to make sure you do not unintentionally clear other bits in the register. To ensure this, the mask with the bit you want to set can be OR'ed with the original contents as shown in the example below:

`DAC0->CTRL = DAC0->CTRL | DAC_CTRL_LPFEN;` or more compactly:

```
DAC0->CTRL |= DAC_CTRL_LPFEN;
```

Clearing a bit is done by ANDing the register with a value with all bits set except for the bit to be cleared:

`DAC0->CTRL = DAC0->CTRL & ~DAC_CTRL_LPFEN;` or

```
DAC0->CTRL &= ~DAC_CTRL_LPFEN;
```

When setting a new value to a bit field containing multiple bits, a simple OR function will not do, since you will risk that the original bit field contents OR'ed with the mask will give a wrong result. Instead you should make sure to clear the entire bit field (and only the bit field) before you OR in the new value like shown below:

```
DAC0->CTRL = (DAC0->CTRL & ~_DAC_CTRL_REFRSEL_MASK) |
DAC_CTRL_REFRSEL_16CYCLES;
```

2.4.5 Grouped registers

Some registers are grouped together within each peripheral. An example of such a group is the registers associated with each GPIO port, like the Data Out Register (DOUT) in Figure 2.1 (p. 4). Each GPIO port (A, B, C, ...) contains a DOUT register and the description below is common for all of these. The x in GPIO_Px_DOUT indicates the port wild card.

Figure 2.2. Grouped registers in GPIO

28.5.4 GPIO_Px_DOUT - Port Data Out Register

Offset	Bit Position																													
0x00C	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																	0x0000															
Access																	RW															
Name																	DOUT															

Bit	Name	Reset	Access	Description
31:16	*Reserved*	*To ensure compatibility with future devices, always write bits to 0.*		
15:0	DOUT	0x0000	RW	Data Out
	Data output on port.			

In the CMSIS defines the port registers are grouped in an array P[x]. When using this array, we must index it using numbers instead of the port letters (A=0, B=1, C=2, ...). Accessing the DOUT register for port C can be done like this:

```
GPIO->P[2].DOUT = 0x000F;
```

2.4.6 efm32_chip.h

In the source files included in this application note, you will see the efm32_chip.h included at the top and that CHIP_Init() is called at the beginning of the main functions. Since early versions of the EFM32 devices were programmed differently in production, the chip function is used to align the chip programming to the latest revision. Do not run any code prior to running the CHIP_Init() function in your main function.

3 Example 1: Register Operation

This example will show how to write and read registers using the CMSIS defines. You will also learn how to observe and manipulate register contents through the debugger in IAR Embedded Workbench. While the examples are shown only for IAR, the tasks can also be completed in other supported IDEs. Open up **efm32** workspace (an\an0009_efm32_getting_started\iar\efm32.eww) and select the **<kit_name>_1_register** project in IAR EW. In the main function in the **1_registers.c** (inside Source Files) there is a marker space where you can fill in your code.

3.1 Step 1: Enable timer clock

In this example we are going to use TIMER0. By default the 14 MHz RC oscillator is running but all peripheral clocks are disabled, hence we must turn on the clock for TIMER0 before we use it. If we look in the CMU chapter of the reference manual, we see that the clock to TIMER0 can be switched on by setting the TIMER0 bit in the HFPERCLKEN0 register in the CMU peripheral.

3.2 Step 2: Start timer

Starting the Timer is done by writing a 1 to the START bit in the CMD register in TIMER0.

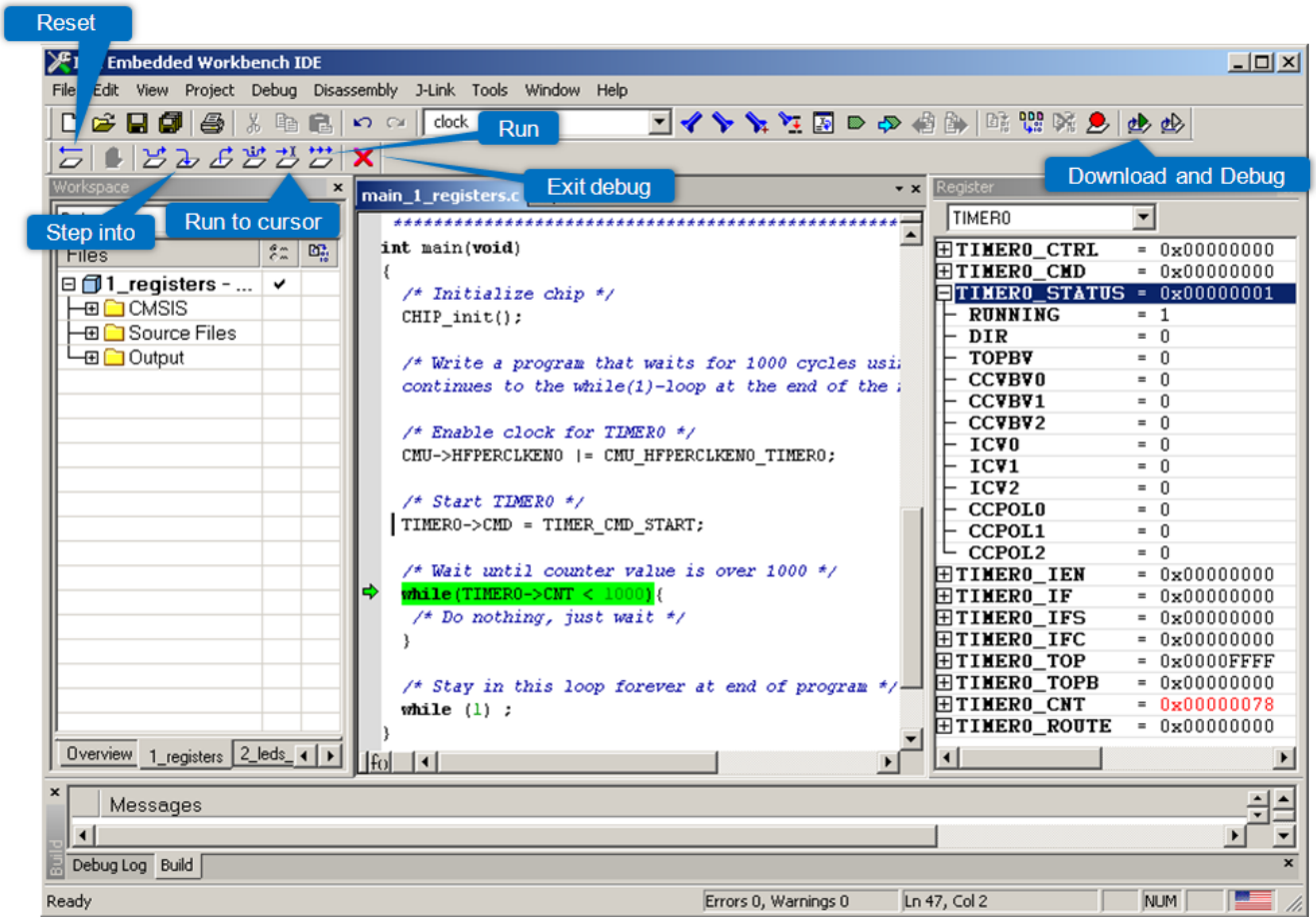
3.3 Step 3: Wait for threshold

Create a while-loop that waits until counter is 1000 before proceeding

3.4 Observation

Make sure the **<kit_name>_1_register** project is active by pressing the corresponding tab at the bottom of the Workspace window. Then press the Download & Debug button (Figure 3.1 (p. 9)). Then go to *View->Register* and find the STATUS register in TIMER0. When you expand this, you should see the RUNNING bit set to 0. Place your cursor in front of the line where you start the timer and press Run to Cursor. Then watch the RUNNING bit get set to 1 in the Register View when you Single Step over the expression. As you continue to Single Step you will see the content of the CNT registers increasing. Try writing a different value to the CNT register by entering it directly in the Register View.

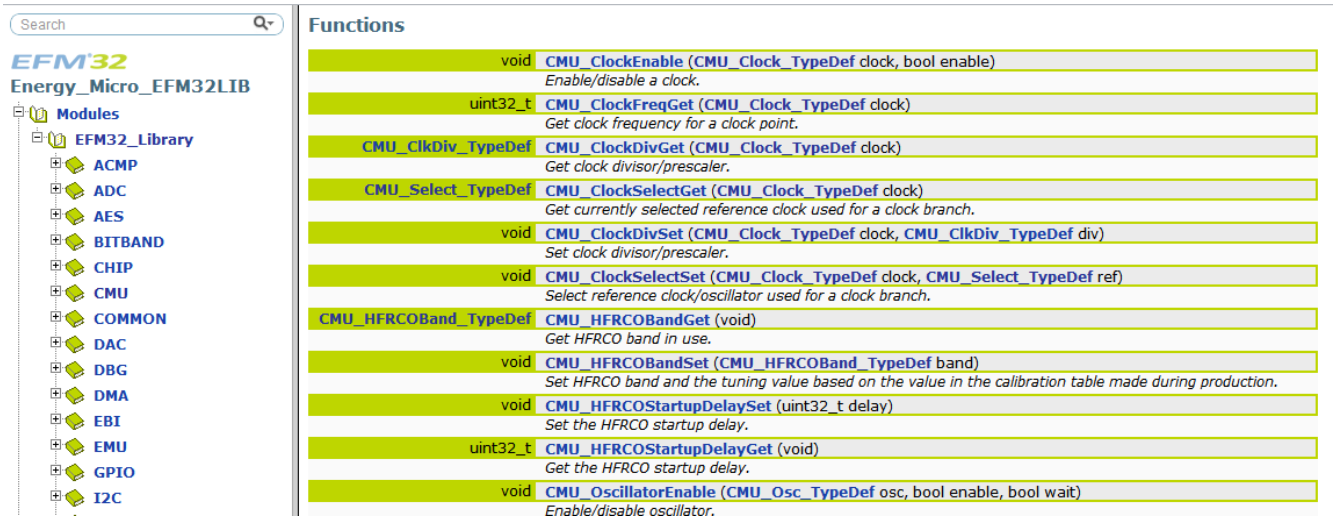
Figure 3.1. Debug View in IAR



4 Example 2a: Blinking LEDs with STK

Since accessing the LEDs is done differently for the Development Kits and the Starter Kits this example is split into two parts, 2a (for STK) and 2b (for DK). In this example for the STKs, the aim is to use the GPIO pins to light up the LEDs on the STK and change the LED configuration every time a button is pressed. Instead of accessing the registers directly, we will use the **emlib** functions to configure the peripherals. The **efm32** workspace contains a project called **<kit_name>_2_leds**, which will be used in this example. The **emlib** C-files are included in the project. The corresponding header files are included at the beginning of the C-files. For details on which **emlib** functions exist and how to use them we will open up **API Documentation** through **Simplicity Studio**. After clicking on the emlib link for the correct EFM32 series (Gecko, Tiny Gecko etc.) we open up Modules->EFM32_Library and select the CMU peripheral. Scrolling down, we find a list of functions (Figure 4.1 (p. 10)) that we can use to to easily operate the Clock Management Unit.

Figure 4.1. Documentation for the CMU-specific emlib functions



4.1 Step 1: Turn on GPIO clock

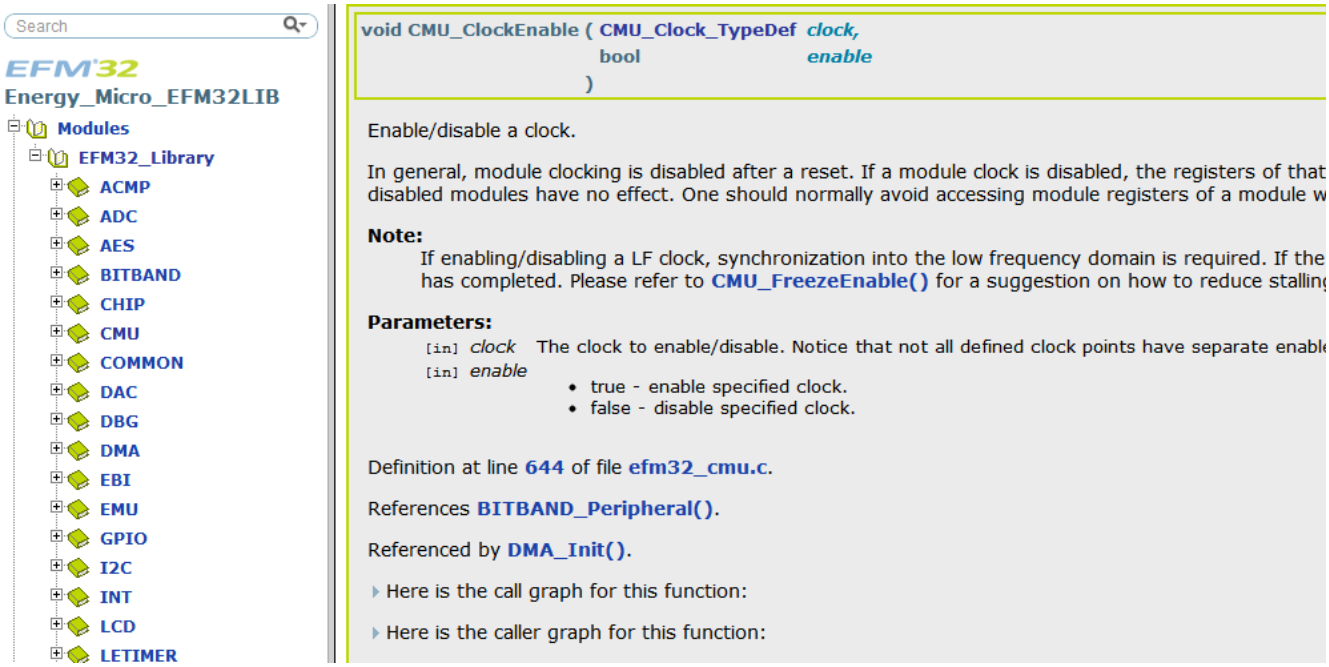
In the list of CMU functions we find the following function to turn on the clock to the GPIO:

```
void CMU_ClockEnable(CMU_Clock_TypeDef clock, bool enable)
```

If we click on the function, we are shown a description (Figure 4.2 (p. 11)) of how to use the function. If we also click on the CMU_Clock_TypeDef link we are also taken to a list of the allowed enumerators for the *clock* argument. To turn on the GPIO we then write the following:

```
CMU_ClockEnable(cmuClock_GPIO, true);
```

Figure 4.2. CMU_ClockEnable function description



4.2 Step 2: Configure GPIO pins for LEDs

In **Simplicity Studio** under **Kit Documentation** we find the User Manual for the STK that we are using, which tells us that the the user LED(s) are connected to the following pin(s):

- EFM32-Gxxx-STK: 4 LEDs on port C, pins 0-3
- EFM32TG-STK3300: 1 LED on port D, pin 7
- EFM32GG-STK3700: 2 LEDs on port E, pins 2-3
- EFM32ZG-STK3200: 2 LEDs on port C, pins 10-11

Looking into the available functions for the GPIO we find the following function to configure the mode of the GPIO pins:

```
void GPIO_PinModeSet(GPIO_Port_TypeDef port, unsigned int pin,
GPIO_Mode_TypeDef mode, unsigned int out)
```

Use this function to configure the LED pin(s) as Push-Pull outputs with the initial DOUT value set to 0.

4.3 Step 3: Configure GPIO pin for button

Looking into the User Manual for the STK we find that Push Button 0 (PB0) is connected to the following pin:

- EFM32-Gxxx-STK: Port B, pin 9
- EFM32TG-STK3300: Port D, pin 8
- EFM32GG-STK3700: Port B, pin 9
- EFM32ZG-STK3200: Port C, pin 8

Configure this pin as an input to be able to detect the button state.

4.4 Step 4: Change LED status when button is pressed

Write a loop that toggles the LED(s) every time PB0 is pressed. Make sure that you do not only check that the button is pressed, but also that it is released, so that you only toggle the LED(s) once for each time you press the button. PB0 is pulled high by an external resistor.

4.5 Extra Task: LED animation

Experiment with creating different blinking patterns on the LED(s), like fading and running LEDs (if there are multiple LEDs). Because the EFM32 runs at 14 MHz as default, you need a delay function to be able to see the LEDs changing real-time. Using what you wrote for TIMER0 in Example 1, you should be able to create the following function:

```
void Delay(uint16_t milliseconds)
```

Use the PRESC bitfield in TIMER0_CTRL to reduce the clock frequency to a desired value.

5 Example 2b: Blinking LEDs with DK

To run this example, you need a Development Kit. The aim is to use the GPIO pins to light up the LEDs on the DK and change the LED configuration every time the joystick is moved. The **efm32** workspace contains a project called **<kit_name>_2_leds**, which will be used in this example.

5.1 Board Support Library

The EFM32 Gecko Development Kit uses a board controller to configure the functions of the development kit, including lighting the user leds and reading buttons/joystick. These functions can be controlled by the EFM32 by communicating with the board controller via either SPI or External Bus Interface. The EFM32 can then configure the registers in the board controller to set up the functions wanted. The EFM32 Board Support Library includes drivers for handing access to the DK. An initialize function must be run first to configure either the EBI or the SPI connection:

```
void DVK_init(void);
```

Devices with LCD support will automatically be set up with SPI access, while devices without LCD support will use EBI by default. The yellow lights to the left of the MCU board on the DK will tell you which connection is active.

Note that the connection to the board controller will occupy the EBI or the USART2 (SPI) on your device as well as the pin connections for these. These resources can not be used for other purposes simultaneously.As the DVK functions also take care of all the settings needed for the DK connection like GPIO settings etc, you do not need to run any other **emlib** functions.

5.2 Step 1: Change LED status when joystick is moved

Write a while loop that increases a value every time the joystick is moved and display this value on the LEDs. Make sure that you do not only check that the button is pressed, but also that it is released, so that you only toggle the LED(s) once for each time you press the button. The 16 user LEDs on the DK are configured by the following function:

```
void DVK_setLEDs(uint16_t leds);
```

Look inside dvk.h for a function to read the joystick status as well. Please note that by default the buttons and joystick on the DK are used to control the menu on the TFT display. To use the buttons and joystick with the EFM32, you need to press the AEM button. The button/joystick status is shown in the upper right hand corner of the TFT display

5.3 Extra Task: LED animation

Experiment with creating different moving patterns on the LEDs, like fading and running LEDs. Because the EFM32 runs at 14 MHz as default, you need a delay function to be able to see the LEDs changing real-time. Using what you wrote for TIMER0 in Example 1, you should be able to create the following function:

```
void Delay(uint16_t milliseconds)
```

Use the PRESC bitfield in TIMER0_CTRL to reduce the timer clock frequency to a desired value.

6 Example 3a: Segment LCD Controller

This example requires either an STK or a DK with MCU board with a Segment LCD. This example will show you how to use the Segment LCD controller and display information on the LCD display. The LCD controller includes an autonomous animation feature which will also be demonstrated. The **efm32** workspace contains a project called **<kit_name>_3_lcd**, which will be used in this example.

6.1 Step 1: Initialize the LCD controller

The LCD controller driver is located in the development kit and starter kit library. First you need to run the initialize function found in **segmentlcd.h** to set up the LCD controller.

6.2 Step 2: Write to LCD display

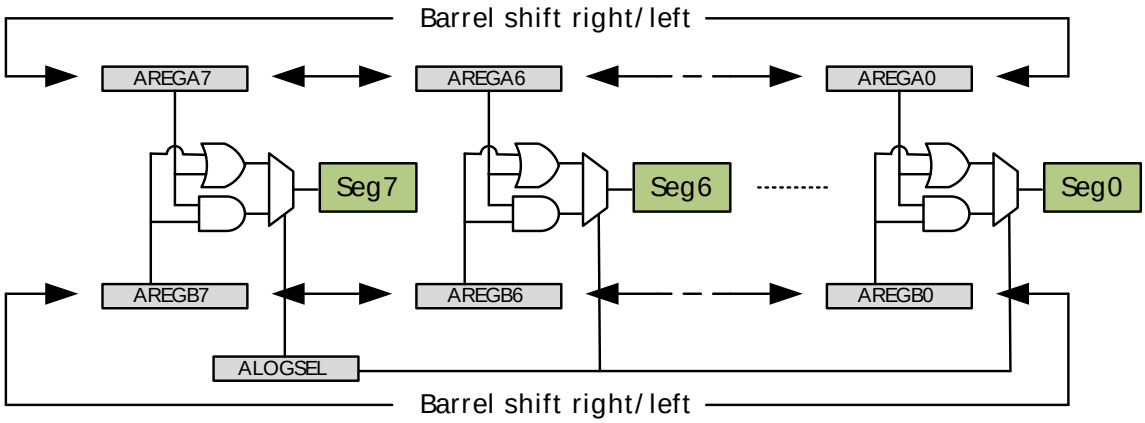
By default, all LCD segments are switched off after initialization. The lcd controller driver includes several functions to control the different segment groups on the display. A few examples are:

```
void SegmentLCD_Number(int value)

void SegmentLCD_Write(char *string)

void SegmentLCD_Symbol(lcdSymbol s, int on);
```

Experiment with putting your own text/numbers symbols on the display and try to make things move about a bit. You can use the Delay function from Example 2 (If you did not complete this part of the example, a Delay function can be found in the solution file.

6.3 Step 3: Animate segments

Figure 6.1. Animation Function



The LCD controller contains an animation feature which can animate up to 8 segments (8 segment ring on the LCD display) autonomously. The data displayed in the animated segments is a logic function (AND or OR) of two register bits for each segment (). The two register arrays (LCD_AREGA, LCD_AREGB) can then be set up to be barrelshifted either left or right every time the Frame Counter overflows. The Frame Counter can be set up to overflow after a configurable number of frames. This is not covered by the LCD driver, so here we have to manipulate the LCD registers by doing direct register writes. The following registers must be set up:

LCD_BACTRL:

- Set Frame Counter Enable bit
- Configure Frame Counter period by setting the FCTOP field
- Set Animation Enable bit
- Select either AND or OR as logic function
- Configure AREGA and AREGB shift direction
- For the STK3700, also set the ALOC bit to SEG8TO15

LCD_AREGA/LCD_AREGB:

- Write data used for animation to these registers.

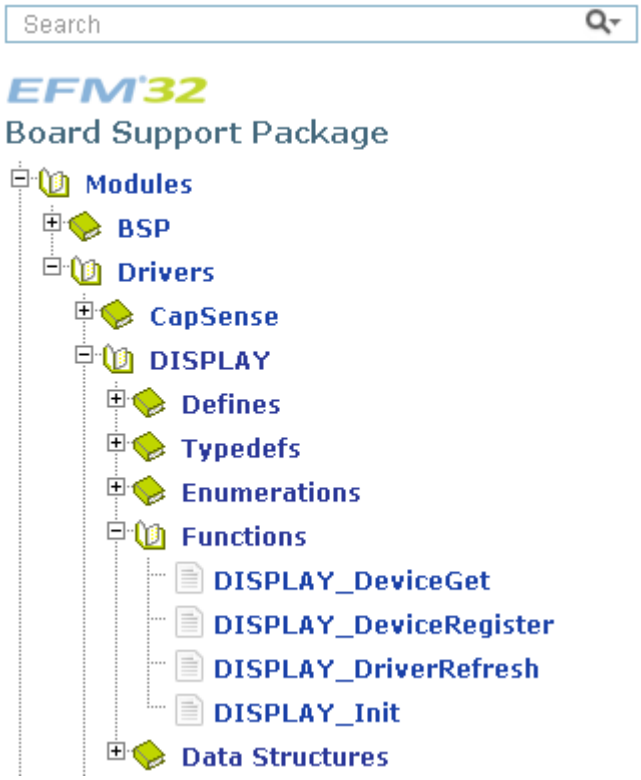
Play around a bit with the configuration of the animated segments and watch the results on the LCD display.

7 Example 3b: Memory LCD

This example requires either an STK or a DK equipped with a Memory LCD. This example shows how to configure the Memory LCD driver and write text on the Memory LCD. The software project called **<kit_name>_3_lcd**, which will be used in this example.

The Board Support Package for the EFM32ZG-STK3200 includes a driver for the memory LCD. Documentation for this driver is found in the "Kit BSP's and Drivers" section under API Documentation in Simplicity Studio. Figure Figure 7.1 (p. 16) shows the location of the driver documentation.

Figure 7.1. BSP Documentation for DISPLAY Driver



7.1 Step 1: Configure the display driver

First, initialize the DISPLAY driver with `DISPLY_Init()`.

The Board Support Package include TEXTDISPLAY, which is an interface for printing text to a DISPLAY device. Use `TEXTDISPLAY_New()` to ceate a new TEXTDISPLAY interface.

7.2 Step 2: Write text to Memory LCD

TEXTDISPLAY implements basic functions for writing text to the Memory LCD. Try `TEXTDISPLAY_WriteString()` and `TEXTDISPLAY_WriteChar()`.

8 Example 4: Energy Modes

This example shows how to enter different Energy Modes (EMx) including Energy Mode 2 and wake up using RTC interrupt. The **efm32** workspace contains a project called **<kit_name>_4_energymodes**, which will be used in this example.

8.1 Advanced Energy Monitor with DK

The Development Kits includes current measurement of the VMCU power domain, which is used to power the EFM32 and the LCD display on the MCU board. The VMCU domain is also available on the prototyping board, so other external components in your prototype can be added to the measurements. The current measurement can be monitored real-time on the TFT display on the main board and also on a PC using the **energyAware Profiler** available in **Simplicity Studio**.

8.2 Advanced Energy Monitor with STK

The EFM32 Starter Kits includes current measurement of the VMCU power domain, which is used to power the EFM32 and the LCD display in addition to other components in the application part of the starter kit. The real-time current measurement can be monitored on a PC using the **energyAware Profiler** available in **Simplicity Studio**.

8.3 Step 1: Enter EM1

To enter EM1, all you have to do is to execute a Wait-For-Interrupt instruction. An intrinsic function for this is shown below:

```
__WFI();
```

After executing this instruction you should see the current consumption drop.

8.4 Step 2: Enter EM3

Entering EM3 is also done by executing the WFI-instruction, only now with the SLEEPDEEP bit in the SCB_SCR register. Set this prior to the WFI-instruction:

```
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;

__WFI();
```

When in an active debug session the EFM32 will not be allowed to go below EM1. To measure the current consumption in EM2, you should therefore end the debugging session and reset the EFM32 with the reset button on the MCU board/STK.

8.5 Step 3: Enter EM2

To enter EM2, you must first enable a low frequency oscillator (either LFRCO or LFXO) before going to deep sleep (same as for EM3). In this example we will enable the LFRCO and wait for it to stabilize by using the following **emlib** function:

```
void CMU_OscillatorEnable(CMU_Osc_Typedef osc, bool enable, bool wait)
```

In addition the EMU **emlib** functions include functions for entering Energy Modes, which you can use instead of setting the SLEEPDEEP bit and executing the WFI-instruction manually:

```
void EMU_EnterEM2(bool restore)
```

8.6 Step 4: Configure Real-Time Counter

To wake up from EM2, we will configure the Real-Timer Counter (RTC) to give an interrupt after 5 seconds. First we must enable the clock to the RTC by using the CMU **emlib** functions. To communicate with Low Energy/Frequency peripherals like the RTC, you must also enable clock for the LE interface (cmuClock_CORE).

The **emlib** initialization function for the RTC, requires a configuration struct as an input:

```
void RTC_Init(const RTC_Init_TypeDef *init)
```

The struct is already declared in the code, but you must set the 3 parameters in the struct before using it with the RTC_Init function:

```
rtcInit.comp0Top = true;
```

Next we must set compare value 0 (COMP0) in the RTC, which will set interrupt flag COMP0 when the compare value matches the counter value. Chose a value that will equal 5 seconds given that the RTC runs at 32.768 kHz:

```
void RTC_CompareSet(unsigned int comp, uint32_t value)
```

Now the RTC COMP0 flag will be set on a compare match, but to generate an interrupt request from the RTC the corresponding interrupt enable bit must also be set:

```
void RTC_IntEnable(uint32_t flags)
```

Now the RTC interrupt request is enabled on a comparator match, but to trigger an interrupt, the RTC interrupt request line must be enabled in the Cortex-M. The IRQn_Type to use is RTC_IRQn.

```
NVIC_EnableIRQ(RTC_IRQn);
```

An interrupt handler for the RTC is already included in the code (RTC_IRQHandler), but it is empty. In this function, you should add a function call to clear the RTC COMP0 interrupt flag. If you do not do this, the Cortex-M will be stuck in the interrupt handler, since the interrupt is never deasserted. Look for the RTC **emlib** function to clear the interrupt flag.

8.7 Extra task: Segment LCD controller in EM2

As an extra task you can enable the LCD controller (given that you have a Segment LCD on your kit) and write something on the LCD display before going to EM2. You could also use the segment animation you made in the previous example in EM2.

This extra task does not apply to the EFM32ZG-STK3200 which features a Memory LCD and not a segment LCD.

9 Summary

Congratulations! You now know the basics of Energy Friendly Programming, including register and GPIO operation, use of basic functions on the DK/STK and LCD, in addition to handling different Energy Modes in the EFM32 and the **emlib**/CMSIS functions. The **Examples** section in **Simplicity Studio** contains several more examples for you to explore and more Application Notes are found in the **App Notes** section.

10 Revision History

10.1 Revision 1.17

2013-10-09

Added missing header file

10.2 Revision 1.16

2013-10-08

New cover layout

Added support for the Zero Gecko Starter Kit (EFM32ZG-STK3200)

New text in the Bit field defines chapter

Fixed issue with STK3700 and LCD animation

10.3 Revision 1.15

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

10.4 Revision 1.14

2012-11-12

Added support for the Giant Gecko Starter Kit(EFM32GG-STK3700)

Adapted software projects to new kit-driver and bsp structure

10.5 Revision 1.13

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3

10.6 Revision 1.12

2012-03-14

Added efm32lib file efm32_emu.c to LED projects

Fixed makefile-error for CodeSourcery projects

10.7 Revision 1.11

2011-10-21

Updated IDE project paths with new kits directory

10.8 Revision 1.10

2011-09-08

Added support for Tiny Gecko Starter Kit (EFM32TG-STK3300)

10.9 Revision 1.03

2011-05-18

Updated projects to align with new bsp version

10.10 Revision 1.02

2010-11-19

Corrected solution c-files for new EFM32LIB functions

Corrected pdf document for new EFM32LIB functions

10.11 Revision 1.01

2010-11-16

Changed software/documentation to use the segmentlcd.c functions, lcdcontroller.c is deprecated

Added section about CMSIS Doxygen documentation

Changed example folder structure, removed build and src folders

Updated chip init function to newest efm32lib version

10.12 Revision 1.00

2010-09-20

Initial revision

A Disclaimer and Trademarks

A.1 Disclaimer

Silicon Laboratories 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 Laboratories 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 Laboratories 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 Laboratories 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 must not be used within any Life Support System without the specific written consent of Silicon Laboratories. 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 Laboratories products are generally not intended for military applications. Silicon Laboratories 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.

A.2 Trademark Information

Silicon Laboratories Inc., Silicon Laboratories, the Silicon Labs logo, Energy Micro, EFM, EFM32, EFR, logo and combinations thereof, and others are the registered trademarks or trademarks of Silicon Laboratories Inc. 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.

B Contact Information

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

Please visit the Silicon Labs Technical Support web page:
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

Table of Contents

List of Figures

List of Tables

List of Examples

List of Equations

silabs.com


ZERO
ARM Cortex-M0+
TINY
ARM Cortex-M3
GECKO
ARM Cortex-M3
LEOPARD
ARM Cortex-M3
GIANT
ARM Cortex-M3
WONDER
ARM Cortex-M4