Universidade Federal do Espírito Santo

Developing for the EFM32GG-STK3700 Development Board

All source code including text of this document is available at https://github.com/hans-jorg/efm32gg-stk3700-gcc-cmsis

Hans-Jörg Schneebeli

2020

MIT License

Version	**1.3**
Date	**08/09/2020**

Foreword

This material is intended `to be` a companion for the lectures in a course in Embedded Systems. The main objective is to present how to develop code for embedded systems and along the way introduces the inner works of the compilation process and advanced patterns of software for embedded systems.

It begins with small applications and ends with the use of real time kernels.

It assumes a basic knowledge of C. It consists of many small projects, each introducing one, and sometimes two, aspects of the development process. Appendices show how to install the needed tools, how to use them and the structure of a project.

Table of Contents

The EFM32GG-STK3700 Development Board

The EFM32 Giant Gecko is a family of Cortex M3 microcontrollers manufactured by Silicon Labs (who bought Energy Micro, the initial manufacturer). The EFM32 microcontroller family has many subfamilies with different Cortex-M architectures and features as shown below.

Family	Core	Features	Flash (kB)	RAM (kB)
Zero Gecko	ARM Cortex-M0+		4- 32	2- 4
Happy Gecko	ARM Cortex-M0+	USB	32- 64	4- 8
Tiny Gecko	ARM Cortex-M3	LCD	4- 32	2- 4
Gecko	ARM Cortex-M3	LCD	16- 128	8-16
Jade Gecko	ARM Cortex-M3		128-1024	32-256
Leopard Gecko	ARM Cortex-M3	USB, LCD	64- 256	32
Giant Gecko	ARM Cortex-M3	USB. LCD	512-1024	128
Giant Gecko S1	ARM Cortex-M4	USB, LCD	2048	512
Pearl Gecko	ARM Cortex-M4		128-1024	32-256
Wonder Gecko	ARM Cortex-M4	USB, LCD	64- 256	32

The EMF32GG-STK3700 Development Board

The EMF32GG-STK3700 is a development board featuring a EFM32GG990F1024 MCU (a Giant Gecko microcontroller) with 1 MB Flash memory and 128 kB RAM. It has also the following peripherals:

- 160 segment LCD
- 2 user buttons, 2 user LEDs and a touch slider
- Ambient Light Sensor and inductive-capacitive metal sensor
- EFM32 OPAMP footprint
- 32 MB NAND flash
- USB interface for Host/Device/OTG

It has a 20 pin expansion header, breakout pads for easy access to I/O pins, different alternatives for power sources including USB and a 0.03 F Super Capacitor for backup power domain.

For developing, there is an Integrated Segger J-Link USB debugger/emulator with debug out functionality and an Advanced Energy Monitoring system for precise current tracking.
The development board EMF32GG-STK3700 features a EFM32GG990F1024 microcontroller from the Giant Gecko family. It is a Cortex M3 processor with the following features:

Flash (KB)	RAM (KB)	GPIO	USB	LCD	USART/ UART	LEUART	Timer/ PWMRTC	ADC	DAC	OpAmp
1024	128	87	Y	8x34	3/2	2 2	4/12	1(8)	2(8)	3

Basic References

The most important references are:

- EFM32GG Reference Manual[1]: Manual describing all peripherals, memory map.
- EFM32GG-STK3700 Giant Gecko Starter Kit User's Guide[2]: Information about the STK3700 Board.
- EFM32GG990 Datasheet[3]: Technical information about the EMF32GG990F1024 including electrical specifications and pinout.
- EFM32 Microcontroller Family Cortex M3 Reference Manual[4]

Peripherals

There are a lot of peripherals in the board.

- LEDs using pins PE2 and PE3.
- Buttons using pins PB9 and PB10
- LCD Multiplexed 20×8 with a 7 character alphanumeric field, a 4 digit numeric field and other symbols using pins PA0-PA11, PA15,PB0-PB6,PD9-PD12,PE4-PE7.
- Touch Sensor using pins PC8-11.
- Light Sensor using PD6,PC6
- LC Sensor using PB12,PC7
- NAND Flash using PB15, PE8-15, PC1-2, PF8-9, PD13-15
- USB OTG using PF11-12, PF5-6

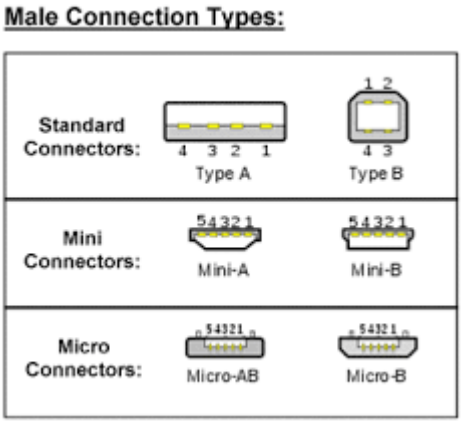
It is also possible to use the header connectors to add more peripherals.

Connections

The EMF32GG-STK3700 board has two USB connectors: One, a Mini USB B Connector, for

1 https://www.silabs.com/documents/public/reference-manuals/EFM32GG-RM.pdf
2 https://www.silabs.com/documents/public/user-guides/efm32gg-stk3700-ug.pdf
3 https://www.silabs.com/documents/public/data-sheets/EFM32GG990.pdf
4 https://www.silabs.com/documents/public/reference-manuals/EFM32-Cortex-M3-RM.pdf

development and other, a Micro B-Type USB Connector, for the application.



For development, a cable (delivered) must be connected between the Mini USB connector on the board and the A-Type connector on the PC. Among the devices listed by the `lsusb` command, the following must appear.

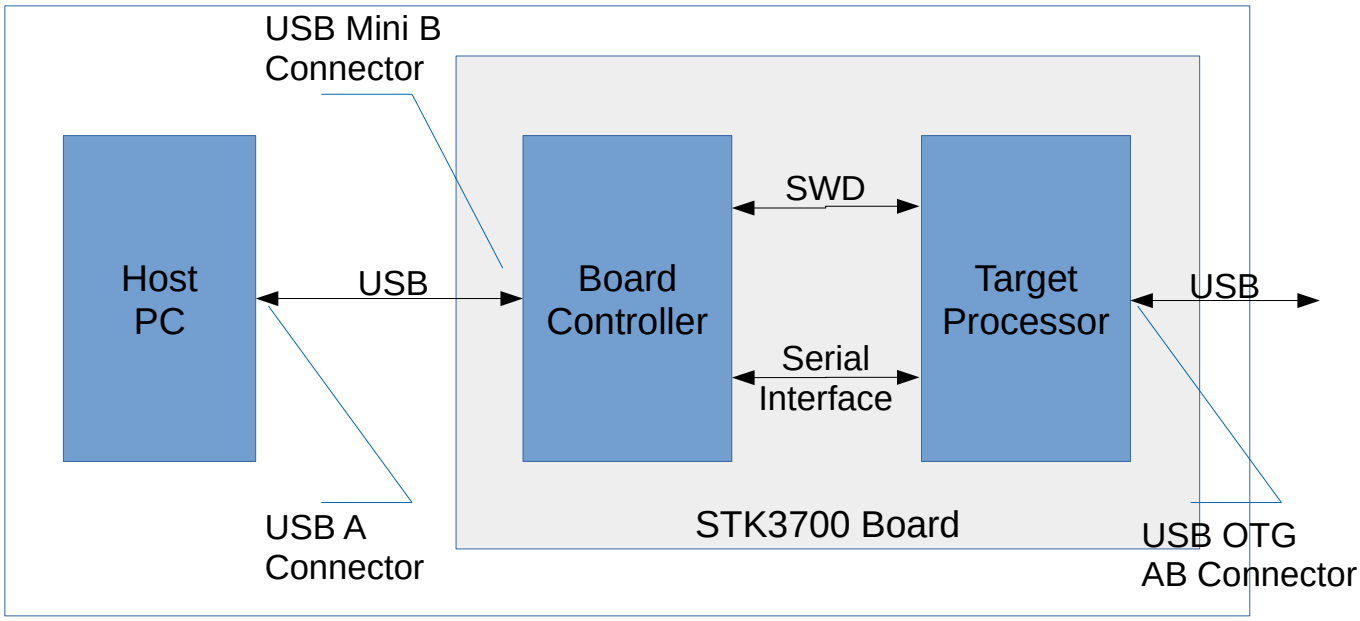
```
Bus 001 Device 019: ID 1366:0101 SEGGER J-Link PLUS
```

The STK3700 board has two microcontrollers: one, called target, is a EFM32GG990F1024 microcontroller, and the other, called Board Controller, implements an interface for programming and debugging.

For a Cortex M microcontroller the following programming interfaces are used:

- SWD : 2 pinos: SWCLK, SWDIO - Serial Wire Debug.
- JTAG : 4 pinos: TCK, TMS, TDI, TDO - Not used in this board

In the STK3700 only the SWD interface is used, and there is a connector on the SWD lines which permits the debugging of off-board microcontrollers.



Generally, in this kind of boards there is a serial interface between the Target and the Board Controller. It can be implemented using a physical channel with 2 lines or a virtual, using the SWD/JTAG channel. Both appears to the Host PC as a serial virtual port (`COMx` or `/dev/ttyACMx`). In the STK3700 board the serial channel uses the UART0 unit (pins PE0 and PE1).

Examples for the EFM32GG-STK3700 Development Board

In all examples, a direct access to registers approach was used. It means that no library besides CMSIS was used.

- 01-Blink: Blink LEDs with direct access to registers
- 02-Blink: Blink LEDs with a HAL for GPIO
- 03-Blink: Blink LEDs with a HAL for LEDs above GPIO HAL
- 04-Blink: Blink LEDs with a HAL for LEDs without a HAL for GPIO
- 05-SysTick: Blink LEDs using SysTick based delays
- 06-`SysTickInterrupt`: Blink LEDs and changing the core frequency
- 07-`StateMachine`: Blink LEDs using `State Machines`
- 08-Button: Control LEDs using buttons (without debounce)
- *09-Button: Control LEDs using buttons (without debounce)
- 10-Debounce*: Control LEDs using buttons (without debounce)
- 11-UART: Uses UART1 to communicate with host computer via USB cable
- *12-UART: USART1 interface using interrupts
- 13-Ministdio: Implements a mini stdio package (printf, gets, etc)
- 14-Newlib: Implements full access to newlib

- 15-TimeTriggered: Uses a Time-Triggered approach[5]
- 16-Protothreads: Uses a proto thread approach
- 17-FreeRTOS: Uses a (free) preemptive real time kernel
- *18-ucos2: Uses ucos version 2
- *19-ucos3: Uses ucos version 3
- 20-LCD: Controls the LCD display

Those marked with an asterisk are unfinished!!!

5 Pontt. Patterns for time-triggered embedded systems 2001

0

Using C

Access to registers

The peripherals and many functionalities of the CPU are controlled accessing and modifying registers.

Silicon Labs as others manufactures provides a set of header with symbols that can be used to access the registers. Following CMSIS standard, the registers are grouped in C `struct`'s.
As an example, the `struct` to access the register for a UART is defined as below. The strange looking comment are part of the documentation using Doxygen.

```
typedef struct
{
  __IOM uint32_t CTRL;       /**< Control Register  */
  __IOM uint32_t FRAME;      /**< USART Frame Format Register  */
  __IOM uint32_t TRIGCTRL;   /**< USART Trigger Control register  */
  __IOM uint32_t CMD;        /**< Command Register  */
  __IM uint32_t  STATUS;     /**< USART Status Register  */
  __IOM uint32_t CLKDIV;     /**< Clock Control Register  */
  __IM uint32_t  RXDATAX;    /**< RX Buffer Data Extended Register  */
  __IM uint32_t  RXDATA;     /**< RX Buffer Data Register  */
  __IM uint32_t  RXDOUBLEX;  /**< RX Buffer Double Data Extended Register  */
  __IM uint32_t  RXDOUBLE;   /**< RX FIFO Double Data Register  */
  __IM uint32_t  RXDATAXP;   /**< RX Buffer Data Extended Peek Register  */
  __IM uint32_t  RXDOUBLEXP; /**< RX Buffer Double Data Extended Peek Register  */
  __IOM uint32_t TXDATAX;    /**< TX Buffer Data Extended Register  */
  __IOM uint32_t TXDATA;     /**< TX Buffer Data Register  */
  __IOM uint32_t TXDOUBLEX;  /**< TX Buffer Double Data Extended Register  */
  __IOM uint32_t TXDOUBLE;   /**< TX Buffer Double Data Register  */
  __IM uint32_t  IF;         /**< Interrupt Flag Register  */
  __IOM uint32_t IFS;        /**< Interrupt Flag Set Register  */
  __IOM uint32_t IFC;        /**< Interrupt Flag Clear Register  */
  __IOM uint32_t IEN;        /**< Interrupt Enable Register  */
  __IOM uint32_t IRCTRL;     /**< IrDA Control Register  */
```

```
  __IOM uint32_t ROUTE;      /**< I/O Routing Register  */
  __IOM uint32_t INPUT;      /**< USART Input Register  */
  __IOM uint32_t I2SCTRL;    /**< I2S Control Register  */
} USART_TypeDef;             /** @} */
```

The base addresses of the USART units are defined as

```
#define USART0_BASE       (0x4000C000UL) /**< USART0 base address  */
#define USART1_BASE       (0x4000C400UL) /**< USART1 base address  */
#define USART2_BASE       (0x4000C800UL) /**< USART2 base address  */
```

And the symbols to access the units as

```
#define USART0 ((USART_TypeDef *) USART0_BASE)        /**< USART0 base pointer */
#define USART1 ((USART_TypeDef *) USART1_BASE)        /**< USART1 base pointer */
#define USART2 ((USART_TypeDef *) USART2_BASE)        /**< USART2 base pointer */
```

To read a register, just write

```
uint32_t w = USART0->CTRL;          // Read
...
USART0->CTRL = w;                   // Write
```

Each register can have many fields. Some fields are just one bit long. One bit fields can be modified with one operation. For example, to set the `SYNC` bit at position 0, use one of these forms (the first is more used). An or operator must be used (Never use addition represented by a + because if the bit is already set, it gives wrong results).

```
USART0->CTRL |= 1;
USART0->CTRL = USART0->CTRL | 1;
```

To clear it, one of these forms (Do not use minus represented by a -)

```
USART0->CTRL &= ~1;
USART0->CTRL &= 0xFFFFFFFEU;
USART0->CTRL = USART0->CTRL & ~1;
USART0->CTRL = USART0->CTRL & 0xFFFFFFFEU;
```

The legibility can be enhanced using a macro `BIT` defined as

```
USART0->CTRL |= BIT(0);                 // Set bit
USART0->CTRL &= ~BIT(0);                // Clear bit
```

But much better is the use of symbols defined in the header for the field.

```
USART0->CTRL |= USART_CTRL_SYNC;        // Set bit
USART0->CTRL &= ~USART_CTRL_SYNC;       // Clear bit
```

There are some fields, that are many bits long. The procedure to modify them is different. One must

be assured that the field is all zero, before setting a new value using an or operation. To set the `OVS` field, that has 2 bits, it must be cleared first.

```
uint32_t w = USART0->CTRL;
w &= ~0x60;
w |= 0x40;
USART0->CTRL = w;
```

Or in just one line

```
USART0->CTRL = (USART0->CTRL&~0x60)|0x40;
```

Or using shift operators

```
USART0->CTRL = (USART0->CTRL&~(3<<5)|(2<<5);
```

Again, a better way is to use symbols defined in the header.

```
USART0->CTRL = USART0->CTRL&~_USART_CTRL_OVS_MASK)|USART_CTRL_OVS_X8;
```

Attention with the underscore before the symbol for the mask. Generally symbols started by underscore are reserved for implementation and should be not used in production code. But the symbols ending in MASK, and in a minor scale, in SHIFT symbols are useful and very used. It is not necessary to use the other symbols, whose names start with underscore. Actually, it is very dangerous because their values are not in the correct position and they must be shifted to the correct position to give the correct results. This can be seen in the header, as shown below, but with the comments omitted for better readability.

```
#define _USART_CTRL_OVS_SHIFT           5
#define _USART_CTRL_OVS_MASK            0x60UL
#define _USART_CTRL_OVS_DEFAULT         0x00000000UL
#define _USART_CTRL_OVS_X16             0x00000000UL
#define _USART_CTRL_OVS_X8              0x00000001UL
#define _USART_CTRL_OVS_X6              0x00000002UL
#define _USART_CTRL_OVS_X4              0x00000003UL
#define USART_CTRL_OVS_DEFAULT          (_USART_CTRL_OVS_DEFAULT << 5)
#define USART_CTRL_OVS_X16              (_USART_CTRL_OVS_X16 << 5)
#define USART_CTRL_OVS_X8               (_USART_CTRL_OVS_X8 << 5)
#define USART_CTRL_OVS_X6               (_USART_CTRL_OVS_X6 << 5)
#define USART_CTRL_OVS_X4               (_USART_CTRL_OVS_X4 << 5)
```

When setting a field with a user calculated value, it is a good idea to ensure that no other bit in the register undetected is modified. For example, there is a 3-bit field called HFCLKDIV, that specifies the divisor of the crystal frequency. Yo set it to `div`, the following instruction can be used.

```
CMU->CTRL = (CMU->CTRL&~_CMU_CTRL_HFCLKDIV_MASK)|
(div<<_CMU_CTRL_HFCLKDIV_SHIFT)&_CMU_CTRL_HFCLKDIV_MASK));
```

Or using a multi step approach.

```
uint32_t w = CMU->CTRL;                          // Get present value
w &= ~_CMU_CTRL_HFCLKDIV_MASK;                   // Set field HFCLKDIV to 0
uint32_t n = div<<_CMU_CTRL_HFCLKDIV_SHIFT;      // Set field with new value
n &= _CMU_CTRL_HFCLKDIV_MASK;                    // Ensure it does not extrapolate
w |= n;                                          // Calculate new value
CMU->CTRL = w;                                   // Set new value
```

About CMSIS

CMSIS stands for *Cortex Microcontroller Software Interface Standard* and is a initiative from ARM to make easier to learn and port applications between ARM Cortex M processors. It defines the headers, initialization code, libraries and many other aspects related to Cortex M programming. The CMSIS is built on many components. One of them creates a Hardware Abstraction Layer (HAL) for the ARM peripherals but not for the manufactured added ones. It standardizes the access to registers and resources of the microcontroller. This enables the developing without reliying too much on libraries provided by the manufacturer, that hampers the portability. See CMSIS Site[6].

CMSIS Header files

The CMSIS header files have two sources. The manufactures provides the device specific files, like efm32gg990f1024.h, found in folder GECKOSDK/platform/Device/SiliconLabs/EFM32GG/Include. ARM provides a another set of them in folder CMSIS/Include/, which describe the common features of the Cortex-M processors. They include, among others, the architecture specific files: core_cm0.h, core_cm0plus.h, core_cm3.h, core_cm4.h, core_cm7.h, core_sc000.h, core_sc300.h. You should never include them, because the device specific file already does it.

NVIC functions

Some operations can not be done directly in C. There is a set of standard functions to execute some operations as listed below.

CMSIS function	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.

6 https://developer.arm.com/embedded/cmsis

void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.
void NVIC_SystemReset (void)	Reset the system.

Special functions

CMSIS function	**Instruction**	**Description**
void __enable_irq(void)	CPSIE I	Change processor state - enable interrupts
void __disable_irq(void)	CPSID I	Change processor state - disable interrupts
void __enable_fault_irq(void)	CPSIE F	Change processor state - enable fault interrupts
void __disable_fault_irq(void)	CPSID F	Change processor state - disable fault interrupts
void __ISB(void)	ISB	Instruction Synchronization Barrier
void __DSB(void)	DSB	Data Synchronization Barrier
void __DMB(void)	DMB	Data Memory Barrier
uint32_t __REV(uint32_t int value)	REV	Reverse Byte Order in a Word
uint32_t __REV16(uint32_t int value)	REV16	Reverse Byte Order in each Half-Word
uint32_t __REVSH(uint32_t int value)	REVSH	Reverse byte order in bottom halfword and sign extend
uint32_t __RBIT(uint32_t int value)	RBIT	Reverse bits
void __SEV(void)	SEV	Send Event
void __WFE(void)	WFE	Wait for Event
void __WFI(void)	W	Wait for Interrupt

Functions to access processor register

CMSIS function	**Description**
uint32_t __get_PRIMASK (void)	Read PRIMASK
void __set_PRIMASK (uint32_t value)	Write PRIMASK
uint32_t __get_FAULTMASK (void)	Read Write
void __set_FAULTMASK (uint32_t value)	Write Write
uint32_t __get_BASEPRI (void)	Read BASEPRI
void __set_BASEPRI (uint32_t value)	Write BASEPRI
uint32_t __get_CONTROL (void)	Read CONTROL
void __set_CONTROL (uint32_t value)	Write CONTROL

uint32_t __get_MSP (void)	Read MSP
void __set_MSP (uint32_t TopOfMainStack)	Write MSP
uint32_t __get_PSP (void)	Read PSP
void __set_PSP (uint32_t TopOfProcStack)	Write PSP

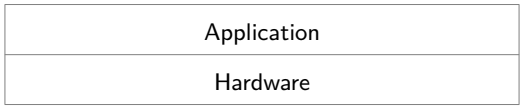
1

First Blink

This is the 1th version of Blink. It uses direct access to register of the EFM32GG990F1024 microcontroller.

It is possible to use the Gecko SDK Library, which includes a HAL Library for GPIO. For didactic reasons and to avoid the restrictions imposed by the license, the direct access to registers is used in this document.

The architecture of the software is shown below.



To access the registers, it is necessary to know their addresses and fields. These information can be found the data sheet and other documents from the manufacturer. The manufacturer (Silicon Labs) provides a CMSIS compatible header files in the platform folder of the Gecko SDK Library.

The `platform` folder has the following sub-folders of interest: `Device` and `CMSIS`. In the Device/SiliconLabs/EFM32GG/Include/ folder there is a header file named emf32gg990f1024.h, which includes the definition of all registers of the microcontroller. One has to be careful because it includes a lot of other header files (`emf32gg_*.h`). It is possible to include the emf32gg990f1024.h file directly in the code, like below.

```
#include <emf32gg990f1024.h>
```

But a better alternative is to use a generic include and define which microcontroller is used as a parameter in the command line (actually a definition of a preprocessor symbol).

```
#include "em_device.h"
```

The command line must then include the `-DEMF32GG990F1024` parameter. To use this alternative

one, has to copy the em_device.h file to the project folder and used quote marks (“”) instead of angle brackets (<>) in the include line.

Instead of using symbols like 0x2 to access the bit to control the LED1, it is better to use a symbol LED1 as below.

```
#define LED1 0x2
```

To define it, a common idiom is to use a BIT macro defined as below (the parenthesis are recommended to avoid surprises).

```
#define BIT(N) (1<<(N))
```

The symbols to access the LEDs in the GPIO Port E registers can then be defined as

```
#define LED1 BIT(2)
#define LED2 BIT(3)
```

To use the GPIO port, where the LEDs are attached, it is necessary to:

- Enable clock for peripherals
- Enable clock for GPIO
- Configure pins as outputs
- Set them to the desired values

To enable clock for peripherals, the HFPERCLKEN bit in the HFPERCLKDIV register must be set. To enable clock for the GPIO, the GPIO bit of the HFPERCLKNE0 register must be set. Both of them are done by or'ing the mask already defined in the header files to the registers.

```
/* Enable Clock for GPIO */
CMU->HFPERCLKDIV |= CMU_HFPERCLKDIV_HFPERCLKEN;     // Enable HFPERCLK
CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_GPIO;           // Enable HFPERCKL for GPIO
```

To access the register for GPIO Port E, a constant is defined, that points to the corresponding memory address.

```
GPIO_P_TypeDef * const GPIOE = &(GPIO->P[4]);  // GPIOE
```

To configure the pins as outputs one has to set the mode fields in the MODE registers. There are two MODE registers: MODEL to configure pins 0 to 7 and MODEH, for pins 8 to 15. To drive the LEDs, the fields must be set to Push-Pull configuration, but just or a binary value is not enough. The field must be cleared (set to 0) before.

```
/* Configure Pins in GPIOE */
GPIOE->MODEL &= ~(_GPIO_P_MODEL_MODE2_MASK|_GPIO_P_MODEL_MODE3_MASK);       // Clear bits
GPIOE->MODEL |= (GPIO_P_MODEL_MODE2_PUSHPULL|GPIO_P_MODEL_MODE3_PUSHPULL);  // Set bits
```

Finally, to set the desired value, one can or a value with a bit 1 in the desired position and all other

bits set to 0.

```
GPIOE->DOUT |= LED1;
```

To clear it, one must AND a value with a bit 0 in the desired position and all other bit set to 1

```
GPIOE->DOUT &= ~LED1;
```

To toggle a bin, one can XOR a value with a bit 1 in the desired position (and other bits set to 0).

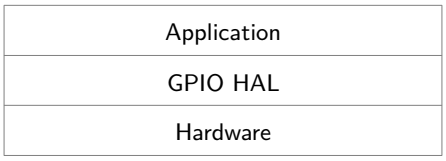
```
GPIOE->DOUT ^= LED1;
```

2

Blink again

This is the 2nd version of Blink. It does not use direct access to register of the EFM32GG990F1024 microcontroller but instead uses a simple Hardware Abstraction Layer (HAL) for the GPIO module.

The architecture is shown below.



The HAL is implemented in the gpio.c and gpio.h files.

In the gpio.h files, a type `GPIO_t` is defined to be used to specify the GPIO port used.
To use a GPIO Port, it is necessary to:

- Enable clock for peripherals
- Enable clock for GPIO
- Configure pins as outputs
- Set them to the desired values

The `GPIO_Init` routine cares of the initialization. It uses the `GPIO_ConfigPins` routine to set the mode (input or output) for the specified pins. To specify the pin status, there are two alternatives: `GPIO_WritePins`, `GPIO_TogglePins`. To read (not implemented yet), there is `GPIO_ReadPins`.

So, it is possible to use the GPIO without detailed knowledge of the internal works. Another advantage is that this routines can be ported to other architectures, with different GPIO implementations.

`GPIO_Init`

`GPIO_Init(gpio,inputs,outputs)` initialized the specified GPIO port `gpio`. The pins specified by `inputs` are configured to be input, and the pins specified by `output`, configured to Push-Pull Output mode. If a pin is specified in both, it is first configured as input. `inputs` and `outputs` are bit masks, with 1 in the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

GPIO_WritePins

`GPIO_WritePins(gpio,zeroes,ones)` set the pins of GPIO port `gpio`, specified by `zeroes` to zero, and the set the pins specified by `ones` to one. If a pin is specified in both, it is first cleared and then set. `zeroes` and `ones` are bit masks, with 1 in the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

GPIO_TogglePins

`GPIO_TogglePins(gpio,pins)` inverts the output pins of GPIO port `gpio`. The `pins` parameter is a bit mask, with 1 in the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

GPIO_ConfigPins

`GPIO_ConfigPins(gpio,pins,mode)` is used to configure the pins of the GPIO port specified by the `gpio` parameter to the desired `mode`. The pins to be configured are specified in the bit mask `pins` parameter, with 1 in the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

The parameter mode can be one of the symbols listed in the table below.

Symbol	Description
GPIO_MODE_DISABLE	Disabled. Pin floats.
GPIO_MODE_INPUT	Input.
GPIO_MODE_INPUTPULL	Input with pull up resistor.
GPIO_MODE_INPUTPULLFILTER	Input with pull up resistor and glitch suppression.
GPIO_MODE_PUSHPULL	Output in a push pull configuration
GPIO_MODE_PUSHPULLDRIVE	Output in a push pull configuration with extra strength
GPIO_MODE_WIREDOR	Output in a wired OR (Open Source) configuration
GPIO_MODE_WIREDORPULLDOWN	Output in a wired OR configuration and pull down resistor
GPIO_MODE_WIREDAND	Output in a wired AND configuration (Open Drain)
GPIO_MODE_WIREDANDFILTER	Output in a wired AND configuration and glitch filter
GPIO_MODE_WIREDANDPULLUP	Output in a wired AND configuration and pull up resistor
GPIO_MODE_WIREDANDPULLUPFILTER	Output in a wired AND configuration, pull up resistor and glitch filter
GPIO_MODE_WIREDANDDRIVE	Output in a wired AND configuration and extra drive
GPIO_MODE_WIREDANDDRIVEFILTER	Output in a wired AND configuration, extra drive and glitch filter
GPIO_MODE_WIREDANDDRIVEPULLUP	Output in a wired AND configuration, extra drive and pull up resistor
GPIO_MODE_WIREDANDDRIVEPULLUPFILTER	Output in a wired AND configuration, extra drive, pull up resistor and

	glitch filter

Acessing LEDs

The symbols to access the LEDs in the GPIO Port E registers can be defined as

```
#define LED1 BIT(2)
#define LED2 BIT(3)
```

To configure the LED pins, the instruction below can do the job.

```
GPIO_Init(GPIOE,0,LED1|LED2);
```

To turn on the LED1, just code

```
GPIO_WritePins(GPIOE,0,LED1);
```

To turn off the LED1, just

```
GPIO_WritePins(GPIOE,LED1,0);
```

To toggle the LED1, just

```
GPIO_TogglePins(GPIOE,LED1);
```

In all cases above, it is possible to modify more than on LED. For example, to clear both LEDs,

```
GPIO_WritePins(GPIOE,LED1|LED2,0);
```

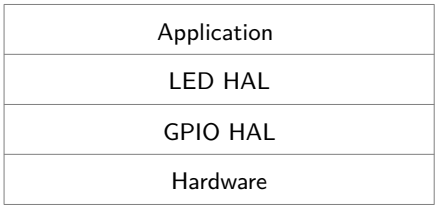
3

Blink revisited

This is the 3rd version of Blink. It **does not use** direct access to registers of the EFM32GG990F1024 microcontroller but instead, uses a layered approach, with a simple Hardware Abstraction Layer (HAL) for the LED module and another HAL for the GPIO.

The main objective of **LEDs** HAL is to permit the control the LEDs without **worrying** about GPIO.

The architecture is shown below.



The LED HAL is implemented in the `led.c` and `led.h` files. The GPIO HAL is implemented in `gpio.c` and `gpio.h` as before.

The main functions for a LED are:

- Initialize (Configure processor, gpio, etc)
- Turn it on
- Turn if off
- Toggle it

The symbols to access the LEDs are defined as bit masks to easy the access the corresponding pins of the GPIO Port E.

```
#define LED1 BIT(2)
#define LED2 BIT(3)
```

LED_Init

`LED_Init(leds)` initializes the GPIO for the leds specified by `leds`. `leds` is a bit mask, with 1 in

the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

LED_Write

`LED_Write(off,on)` turns off the LEDs specified by `off`, and then turns on the LEDs specified by `on`. If a LED is specified in both, it is first cleared and then set.

LED_Toggle

`LED_Toggle(leds)` inverts the status of the LEDs specified by `leds`.

LED_On

`LED_On(leds)` turns on the LEDs specified by `leds`.

LED_Off

`LED_Off(leds)` turns off the LEDs specified by `leds`.

Acessing LEDs

To configure the LED pins, the instruction below can do the job.

```
LED_Init(LED1|LED2);
```

To turn on the LED1, just code

```
LED_Write(0,LED1);
```

To turn off the LED1, just

```
LED_Write(LED1,0);
```

To toggle the LED1, just

```
LED_Toggle(LED1);
```

In all cases above, it is possible to modify more than on LED. For example, to clear both LEDs,

```
LED_Write(LED1|LED2,0);
```

4

Another Blink

This is the 4th version of Blink. It does not use direct access to register of the EFM32GG990F1024 microcontroller but instead a simple Hardware Abstraction Layer (HAL) for the LEDs. This HAL is built directly upon the hardware, using direct access to registers, and so, it does not use a GPIO HAL (See version 2 of Blink).

The main objective is to control the LEDs without knowing about GPIO and less overhead than the previous approach. The architecture is shown below.

The LED HAL is implemented in the led.c and led.h files.

The main functions for a LED are:

- Initialize (Configure processor, gpio, etc)
- Turn it on
- Turn if off
- Toggle it

The symbols to access the LEDs are defined as bit masks to easy the access the corresponding pins of the GPIO Port E.

```
#define LED1 BIT(2)
#define LED2 BIT(3)
```

LED_Init

`LED_Init(leds)` initializes the GPIO for the LEDs specified by `leds`. `leds` is a bit mask, with 1 in the desired position. Bit 0 is the Least Significant Bit (LSB) of the 32 bit word.

LED_Write

`LED_Write(off,on)` turns off the LEDs specified by `off`, and then turns on the LEDs specified by `on`. If a LED is specified in both, it is first cleared and then set.

LED_Toggle

`LED_Toggle(leds)` inverts the status of the LEDs specified by `leds`.

LED_On

`LED_On(leds)` turns on the LEDs specified by `leds`.

LED_Off

`LED_Off(leds)` turns off the LEDs specified by `leds`.

Acessing LEDs

To configure the LED pins, the instruction below can do the job.

```
LED_Init(LED1|LED2);
```

To turn on the LED1, just code

```
LED_Write(0,LED1);
```

To turn off the LED1, just

```
LED_Write(LED1,0);
```

To toggle the LED1, just

```
LED_Toggle(LED1);
```

In all cases above, it is possible to modify more than on LED. For example, to clear both LEDs,

```
LED_Write(LED1|LED2,0);
```

5

Using SysTick to implement delays

The SysTick Timer

This is the 5th version of Blink. It uses the same LED HAL used in the last example. The main aspect is the use of a standard timer present in all Cortex M processors. Doing so, the timing is not influenced by the overhead of interrupt processing and in a lesser extend, changing clock frequencies.

The SysTick counter is a 24-bit regressive counter, which can be clocked by the main clock or by an alternate clock source. In the EMF32 Giant Geck family, the alternate clock source is the LFBCLK clock signal. This clock signal can be derived from the following clock signals according fields LFG and LFBE of CMU_LFCLKSEL register.

Clock source		**Range**
LFXOCLK	Crystal oscillator	32768 Hz
HFCORECLK/2	HFCLK/prescaler/2	(2-24 MHz)/prescaler
HFCORECLK/4	HFCLK/prescaler/4	(1-12 MHz)/prescaler
LFRCOCLK	RC oscillator	32768 Hz
BURTCCLK	LFRCOCLK/prescaler/ divider32	(4096-32768)/ divider32
	LFXOCLK/prescaler/ divider32	(4096-32768)/ divider32
	ULFRCO/divider32	2000/divider32
	ULFRCO/2/divider32	1000/divider32

The maximal value of the SysTick counter is 2^{24}-1 = 16777215. Every time the SysTick counter reaches zero, the counter is loaded with a predefined value and an interrupt is generated.

The are CMSIS standard functions to control the SysTick timer. The main routine is the `SysTick_Config`. The parameter is the reload value. If the reload value, is the frequency of the clock source, an interrupt is generated every second. But this only works for clock frequencies smaller than 16 MHz (It is a 24 bit counter!!). If the value is the clock frequency divided by 1000, an interrupt is generated every 1 ms. Take care of rounding!

To get the core clock frequency, when using CMSIS one can use the `SystemCoreClock` global variable. According CMSIS standard, it must be updated every time the clock frequency is updated.

In this implementation, the global variable `TickCounter` is incremented when a SysTick interrupt is generated, i.e., every 1 ms. Since it is incremented every 1 ms, the variable `TickCounter` overflows after 49 days. Is is possible to use a larger variable (`uint64_t`), but the operation would not be atomic, because it will need more cycles to increment.

The global variable `TickCounter` must be defined as `volatile`. This instructs the compiler to get the value of the variable from memory every time it is needed. This avoid the compiler, trying to optimize, to keep a copy of the variable in a register. In this case, the code would not detected a change of the variable.

The routine `SysTick_Handler` is standardized by CMSIS. It is defined in the correct position in the interrupt vector inside the start_efm32gg.c as a weak symbol. This means that it can be redefined without generating an error and the new symbol overrides the old (weak) one.

To wait for a certain number of milliseconds, one can implement a Delay routine like this.

```
void Delay(uint32_t delay) {

    volatile uint32_t limit = TickCounter+delay;

        while( TickCounter < limit ) {}
}
```

But this implementation has problems when the `TickCounter` reaches the maximal value (2^{24}-1) and wraps around. A better alternative is like this.

```
void Delay(uint32_t delay) {
    volatile uint32_t initialvalue = TickCounter;

    while( (TickCounter-initialvalue) < delay ) {}
}
```

Note 1 – Delay with smaller values

To implement delay with smaller values, one can use the SysTick with the maximum value and to take the difference between the values read in different times. This works for delay as high as 16777215 clock pulses. For a 48 MHz, this means approximately 3,5 seconds.

```
void DelayInClock(uint32_t delay) {
volatile uint32_t initialvalue = SysTick->VAL;
```

```
    while( ((initialvalue-SysTick->VAL)&0xFFFFFF) < delay ) {}
}

main() {
    ...
    SysTickConfig(0xFFFFFF);
    ...
}
```

Note 2 – Another approach

It is possible to configure the SysTick to use a reload value and just wait until it reaches zero. It is not necessary to use interrupts. Again, the maximum time is approximately 3,5 seconds.

```
void DelayPulses(uint32_t delay) {

    SysTick->LOAD = delay;
    SysTick->VAL  = 0;
    SysTick->CTRL = SysTick_CTRL_ENABLE_Msk;

    while( (SysTick->CTRL&SysTick_CTRL_COUNTFLAG_Msk) == 0 ) {}
    SysTick->CTRL = 0;
    }
```

Note 3 – Yet another approach

For very small delays, it is possible to use a cycle counter, optionally available in some ARM microcontrollers in the Data Watchpoint and Trace (DWT) module. The DWT Cycle Counter Register is a 32 bit register, that is incremented at every core clock cycle. This means, it wraps around at approximately 89 seconds.

```
#define DWT_CONTROL *((volatile uint32_t *)0xE0001000)
#define DWT_CYCCNT  *((volatile uint32_t *)0xE0001004)
#define SCB_DEMCR   *((volatile uint32_t *)0xE000EDFC)

static inline uint32_t getDwtCyccnt(void)
{
    return DWT_CYCCNT;
}

static inline void resetDwtCyccnt(void)
{
    DWT_CYCCNT = 0; // reset the counter
}

static inline void enableDwtCyccnt(void)
```

```
{
    SCB_DEMCR   = SCB_DEMCR |BIT(24);       // TRCENA = 1
    DWT_CONTROL = DWT_CONTROL | BIT(0) ;    // enable the counter (CYCCNTENA = 1)
    DWT_CYCCNT  = 0;                        // reset the counter
}
```

Note 4 - The same as before but using CMSIS

This the same as before but using CMSIS. According it, a DWT struct with many fields enables the access to the Cycle Counter.

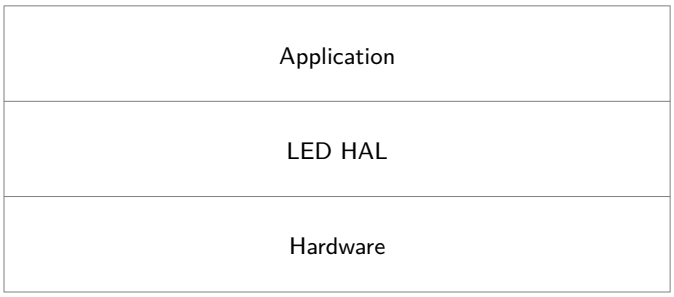
```
/**
 * Delay using Data Watchpoint and Trace (DWT)
 */

static inline uint32_t getDwtCyccnt2(void)
{
    return DWT->CYCCNT;
}

static inline void resetDwtCyccnt2(void)
{
    DWT->CYCCNT = 0; // reset the counter
}

static inline void enableDwtCyccnt2(void)
{
    CoreDebug->DEMCR = CoreDebug->DEMCR|BIT(24);   // TRCENA = 1
    DWT->CTRL  = DWT->CTRL  | BIT(0);              // enable the counter (CYCCNTENA = 1)
    DWT->CYCCNT= 0;                                // reset the counter
}
```

6

Changing the core clock frequency

Controlling the clock frequency

This is the 6th version of Blink. It uses the same HAL for LEDs (STK3700). The main modification is the use of a non default clock frequency.
The routines to change the clock frequency are not part of CMSIS because the clock circuitry is up to the manufacturer. The routines to control clock frequency for the EFM32GG are in the `clock-efm32gg.c` file, with the interface defined in clock-`efm32gg.h`.

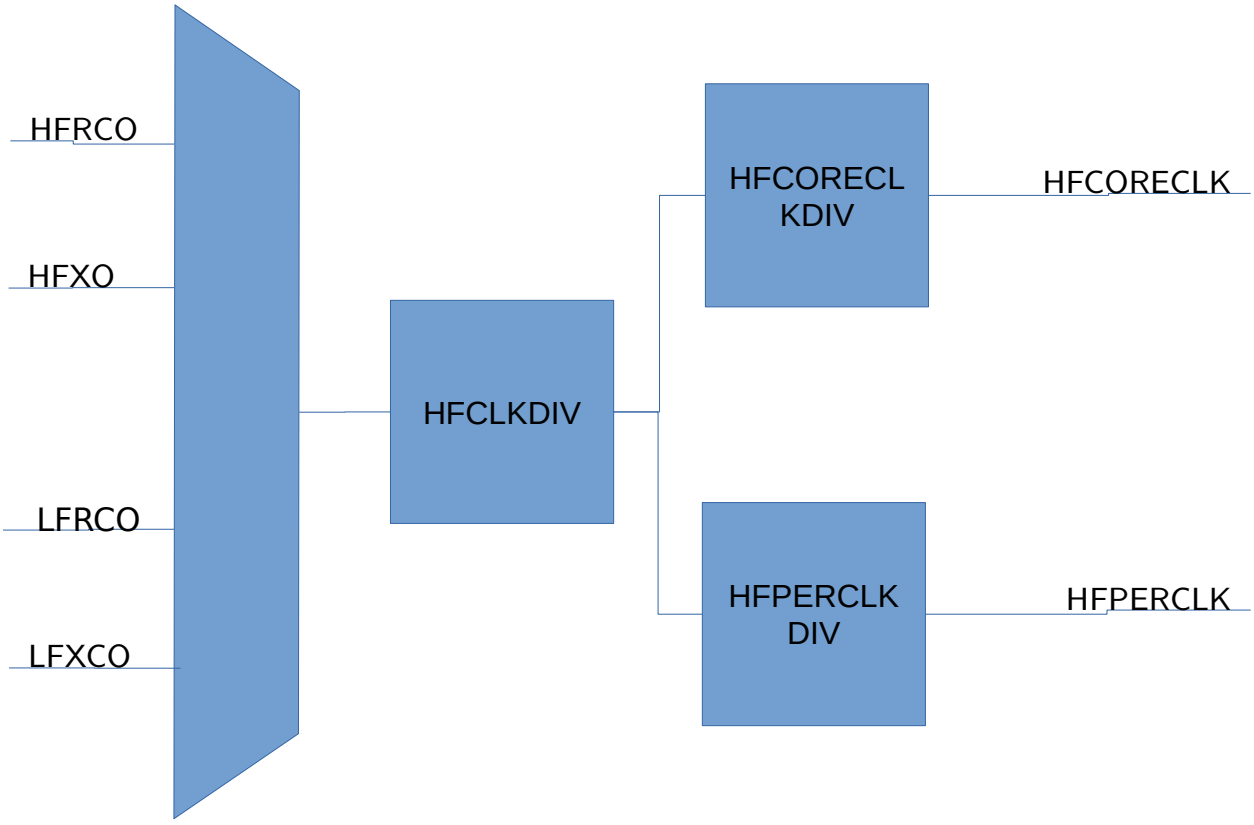


Clock Circuitry

In the EFM32GG the clock source can be:

- LFRCO: 32678 Hz internal RC oscillator
- LFXCO: 32768 Hz crystal oscillator with an external 32768 Hz crystal
- HFRCO Internal RC oscillator (factory calibrated) that runs at 1, 7, 11, 14 (default), 21 and 28 MHz
- HFXCO: Crystal oscillator with an external crystal with an external crystal (48 MHz in the STK32700 Kit)

The clock source signal is then divided by the HFCLK divisor to generate the HFCLK signal. The HFCLK divisor is in the range 1 to 8.
The core clock is generated from the HFCLK by a prescaler, which is a power of 2 up to 512 (1,2,4,8,...,512). There is another prescaler to generate the peripheral clock.



Clock Control

The following routines enables the control of the clock for the EFM32GG and adjust accordingly the number of Flash Wait States and other configuration tidbits.

`SystemCoreClockSet`

The main routine to control core clock frequency is

```
uint32_t SystemCoreClockSet(ClockSource_t source, uint32_t hfclkdiv, uint32_t corediv);
```

The source can be:

- CLOCK_LFXO to use the Low Frequency Crystal Oscillator: 32768 Hz
- CLOCK_LFRCO to use the Low Frequency Internal RC Oscillator: 32768 Hz
- CLOCK_HFRCO_1MHZ to use the Factory Calibrated High Frequency Internal RC Oscillator: 1 MHz
- CLOCK_HFRCO_7MHZ to use the Factory Calibrated High Frequency Internal RC Oscillator: 7 or 6.6 MHz
- CLOCK_HFRCO_11MHZ to use the Factory Calibrated High Frequency Internal RC Oscillator: 11 MHz
- CLOCK_HFRCO_14MHZ to use the Factory Calibrated High Frequency Internal RC

Oscillator: 14 MHz (default)

- CLOCK_HFRCO_21MHZ to use the Factory Calibrated High Frequency Internal RC Oscillator: 21 MHz
- CLOCK_HFRCO_28MHZ to use the Factory Calibrated High Frequency Internal RC Oscillator: 28 MHz
- CLOCK_HFXO to use the High Frequency Crystall Oscillator: 48 MHz (STK3700)

This routine sets the Core and Peripheral Clock to use the same frequency.

ClockSetHFClockDivisor

It is possible to change the HFCLK divisor using the following routine:

```
void     ClockSetClockDivisor(uint32_t hfclkdiv);
```

ClockSetPrescalers

It is possible to change the core and peripheral prescalers using the following routine:

```
void     ClockSetPrescalers(uint32_t hfcoreclkdiv, uint32_t hfperclkdiv);
```

7

Processing inside the SysTick Interrupt

This is the 7th version of Blink. It uses the same HAL for LEDs (STK3700). The main modification is the use a better way to control time.
In all examples shown, the delay between blink was controlled by a routine called Delay.

```
void Delay(uint32_t delay) {
volatile uint32_t counter;
int i;

    for(i=0;i<delay;i++) {
        counter = 10000;
        while( counter ) counter--;
    }
}
```

This form has two HUGE drawbacks:

1. During this time, no work is done. This routine is just a waste of processor and energy.
2. It is dependent on processor clock, and compiler. Code generated by different compilers lead to different delays. Changing the clock frequency changes all timing.

A better way is to use a periodical interrupt. All Cortex M devices have a System Times (SysTick) peripheral. It is a 24 bit counter. (beware, it is not 32 bits). It is basically controlled by two registers: SysTick Reload Value Register (RVR) and the SysTick Current Value Register (CVR). The Current Value Register counts downward until zero. Then it loads automatically the value stored in the Reload Value Register. Most implementations permit to choose different sources for the clock signal, but the default is the Core Clock.

Using CMSIS it is easy to set the `SysTick` using the `SystemCoreClock` variable (contains the frequency of core). In the `startup_DEVICE.c` there is a weak definition of a `SysTick` Interrupt Handler. When a routine called `SysTick_Handler` is defined in other module, the weak definition is not used, and the pointer in the Interrupt Vector points to the newly defined `SysTick_Handler`.

```
void SysTick_Handler(void) {
```

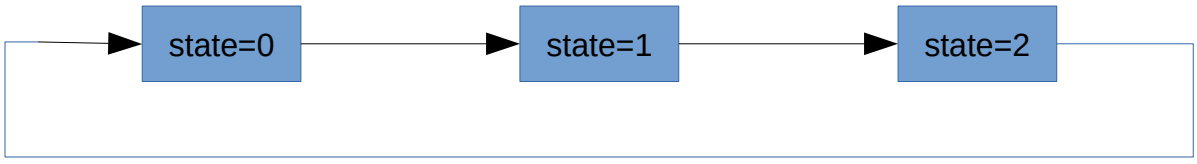
```
    ... processing of

}
...
(void) SysTickConfig(SystemCoreClock/1000);    // Tick every millisecond
...
```

In the Cortex-M family an interrupt processing routine is a C function with no parameters and no return value. This is quite different from other architectures.

The structure of a program using interrupts is very different of the structure of sequential programs, like the former examples. Interrupts must be fast and so, it never should have a wait loop. Instead a sequence of actions, separated by delay calls, an interrupt routine must take a different action every time it is called. In order to take the correct action, the interrupt must have a state.

```
void SysTick_Handler(void) {
static int8_t state = 0;            // must be static

    switch(state) {
    case 0:
        LED_Toggle(LED1);
        state = 1;
        break;
    case 1:
        LED_Toggle(LED2);
        state = 2;
        break;
    case 2:
        LED_Write(0,LED1|LED2);
        state = 0;
        break;
    }
}
```

This can be represented by a very simple state machine.



But there is the fact that the SysTick counter is a 24 bit counter. It can counts from 2^{24}-1=16.777.215

to 0. If the frequency of core clock is a little greater than 16 MHz, it is not possible to make the SysTick handler be called every second. The solution is to use a software divider.

```
void SysTick_Handler(void) {
static int counter = 0;            // must be static

    if( counter != 0 ) {
        counter--;
    } else {
        // Processing
        ...
        counter = DIVIDER-1;
    }
```

The processing part is then called once in every DIVIDER calls.

8

Timers

This is the 8th version of Blink. It uses the same HAL for LEDs (STK3700). The main modification is the use a better way to control time.

When using a periodical interrupt, there is a tendency to code a convoluted interrupt routine, because it concentrates the processing of different tasks, each running at different rates and priorities.

All Cortex M devices have a System Timer (SysTick) peripheral. It is a 24 bit counter (Beware, it has not 32 bits). It is basically controlled by two registers: SysTick Reload Value Register (RVR) and the SysTick Current Value Register (CVR). The Current Value Register counts downward until zero. Then it loads automatically the value stored in the Reload Value Register. Most implementations permit to choose different sources for the clock signal, but the default is the Core Clock.

One approach is to have different software counters. Each one associated with a function. When the a counter reaches zero, the function is called and the counter is reloaded. Otherwise the counter is decremented.

A structure called Timers_t is used to keep the information needed.

```
typedef struct {
    int counter;
    int period;
    void (*function)(void);
} Timers_t;
```

The timer information is stored in an array called Timers. The size is defined by the preprocessor symbol `TIMERS_N`, and the default value is 10. This can be redefined in the command line by the use of the `-DTIMERS_N=20` parameters.

```
Timer_t Timers[TIMERS_N];
```

A variable called `Timers_cnt` controls the number of counters in use.

The main routine is the `Timers_dispatch`, that must be called in the SysTick interrupt routine. It scans the `Timers` array, decrements the counter values and calls the corresponding function, when it reaches zero.

```
void Timers_dispatch(void) {
int i;

    for(i=0;i<Timers_cnt;i++) {
        if( Timers[i].counter == 0 ) {
            Timers[i].function();
            Timers[i].counter = timers[i].period;
        }
        Timers[i].counter--;
    }
}
```

The interrupt routine has the following structure.

```
void SysTick_Handler(void) {
static int counter = 0; // must be static

    if( counter != 0 ) {
        counter--;
    } else {
        Timers_dispatch();
        counter = DIVIDER-1;
    }
}
```

The time unit is SysTick Frequency divided by DIVIDER.

To use it, one should create the functions to be called and add it to the timers list.

```
void t1(void) {
    // Called every time
}
```

```
    void t3(void) {
        // Called every 3rd time
    }

    void t10(void) {
        // Called every 20th time
    }


    main() {
        ...
        Timers_add(t1,1);
        Timers_add(t3,3);
        Timers_add(t10,10);
        ...
        while(1) {}
    }
```

An example application is show below.

```
void BlinkLED1(void) {
    LED_Toggle(LED1);
}

void BlinkLED2(void) {
    LED_Toggle(LED2);
}

int main(void) {

   // Set clock source to external crystal: 48 MHz
    (void) SystemCoreClockSet(CLOCK_HFXO,1,1);

    /* Configure Pins in GPIOE */
    LED_Init(LED1|LED2);

    /* Configure SysTick */
    SysTick_Config(SystemCoreClock/SYSTICKDIVIDER);    // Every 1 ms

    Timers_add(2,BlinkLED1);
    Timers_add(3,BlinkLED2);

    /* Blink loop */
    while (1) {}
```

```
}
```

9

Buttons

This is the 1st version of Button. It implements a polling method to read the push buttons. It uses the HAL for LEDs (STK3700) used in the last Blink example.

Button API (Application Programming Interface)

The buttons are represented as bits in a 32 bit unsigned integer.

The functions implemented are:

```
void        Button_Init(uint32_t buttons);
uint32_t    Button_Read(void);
uint32_t    Button_ReadChanges(void);
uint32_t    Button_ReadPressed(void);
uint32_t    Button_ReadReleased(void);
```

As an example, the function `Button_ReadReleased` is shown. There in a static variable called `lastread`, updated every time a read is done. The expression `changes = newread&~lastread` returns 1 in the corresponding bit position when the present read value is 1 and the last read value is 0.

```
uint32_t Button_ReadReleased(void) {
uint32_t newread;
uint32_t changes;

    newread = GPIOB->DIN;
    changes = newread&~lastread;
    lastread = newread;

    return changes&inputpins;
}
```

Main function

The main function is implemented in a very direct way.

```
int main(void) {
uint32_t b;

    /* Configure LEDs */
    LED_Init(LED1|LED2);

    /* Configure buttons */
    Button_Init(BUTTON0|BUTTON1);

    /* Blink loop */
    while (1) {
        b = Button_ReadReleased();
        if( b&BUTTON0 ) {
            LED_Toggle(LED1);
        }
        if( b&BUTTON1 ) {
            LED_Toggle(LED2);
        }
    }

}
```

10

More about buttons

This is the 2nd version of Button. It implements an interrupt based method to read the push-buttons. It uses the HAL for LEDs (STK3700) used in the last Blink example.

Interrupts

The buttons are connected to pins 9 and 10 por GPIO Port B. GPIO pins of all ports can generate two interrupts: `IRQ_GPIO_EVEN` and `IRQ_GPIO_ODD` according the pin number.

Given a pin number, only one GPIO port can generate an interrupt. Since each GPIO port has 16 pins, there are 16 sources of interrupt. The registers `EXTIPSELL` and `EXTISELH` specify which port can generate an interrupt for a given pin number.
The registers `EXTIRISE` and `EXTIFALL` specify which changes on input generate an interrupt. The registers `IEN`, `IF`, `IFS` and `IFC` control the interrupt of each pin.

Since the pins used for buttons are 9 and 10, both interrupts can be generated and so, both interrupt routines `GPIO_ODD_IRQHandler` and `GPIO_EVEN_IRQHandler` must be defined. They are defined inside the file button.c, and this can be a problem when other interrupts generated by GPIO pins but not generated by buttons, must be handled.

The API can replicate the one used in the last example, based on polling. But the advantage of interrupt based implementation, is that it is possible to take notice of an event as soon as it occurs. This can be done using a callback mechanism. The user application specifies a function, which is called by the interrupt routine.

Interrupt in Cortex M Microcontrollers

Interrupts in Cortex M microcontrollers are controller by the Nested V Interrupt Controller (NVIC), which is part of the microcontroller core.

The NVIC is controlled by a set of specific functions of CMSIS.

CMSIS function	Description
`void NVIC_EnableIRQ(IRQn_Type IRQn)`	Enables an interrupt or exception.

`void NVIC_DisableIRQ(IRQn_Type IRQn)`	Disables an interrupt or exception.
`void NVIC_SetPendingIRQ(IRQn_Type IRQn)`	Sets the pending status of interrupt or exception to 1.
`void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`	Clears the pending status of interrupt or exception to 0.
`uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)`	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
`void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`	Sets the priority of an interrupt or exception with configurable priority level to 1.
`uint32_t NVIC_GetPriority(IRQn_Type IRQn)`	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.
`void NVIC_SystemReset (void)`	Reset the system.

Two important observations:

1. All interrupts after reset are defined to occur at level 0, i.e. the highest level. They must ALWAYS be defined to another level.
2. All interrupts are disabled after reset. They must explicitly be enabled.

Button API (Application Programming Interface)

The buttons are represented as bits in a 32 bit unsigned integer.

The function implemented are:

```
void        Button_Init(uint32_t buttons);
uint32_t    Button_Read(void);
uint32_t    Button_ReadChanges(void);
uint32_t    Button_ReadPressed(void);
uint32_t    Button_ReadReleased(void);
void        Button_SetCallback( void (*callback)(uint32_t parms) );
```

Main function

The functionalities are split between the main function and the callback function. A call to WFI in the main loop puts the processor in a energy saving mode until an interrupt occurs.

```
void buttoncallback(uint32_t v) {

    if( Button_ReadReleased() )
        LED_Toggle(LED2);

}
...
int main(void) {

    /* Configure LEDs */
    LED_Init(LED1|LED2);

    /* Configure buttons */
    Button_Init(BUTTON0|BUTTON1);
    Button_SetCallback(buttoncallback);

    /* Configure Sys Tick */
    SysTickConfig(SystemCoreClock/1000);

    LED_Write(0,LED1|LED2);
    /*
     * Read button loop. ATTENTION: No debounce
     */
    while (1) {
        __WFI();
    }

}
```

11

Debouncing

This is the 3rd version of Button. It implements a debounce process using an interrupt based method. It uses the HAL for LEDs (STK3700) used in the last Blink example.

Bounce

When a contact occurs between two metal parts, there is a cycle of contact and release pulses until in settles in the expected level. The main cause is the elastic collision between the two parts. It generally takes about 10-50 ms to get a stable read.

The basics of debouncing is to wait until the transitory vanished.

The basic approaches are:

1. Implement a state machine where a stable output is generated when the read is confirmed (repeated) after a predetermined time,
2. Read multiple times, and generates an output when the read is repeated (confirmed) N times, where N corresponds to the debounce time.

In boot approaches, it is important to use a strict timing control to enhance portability and reliability.

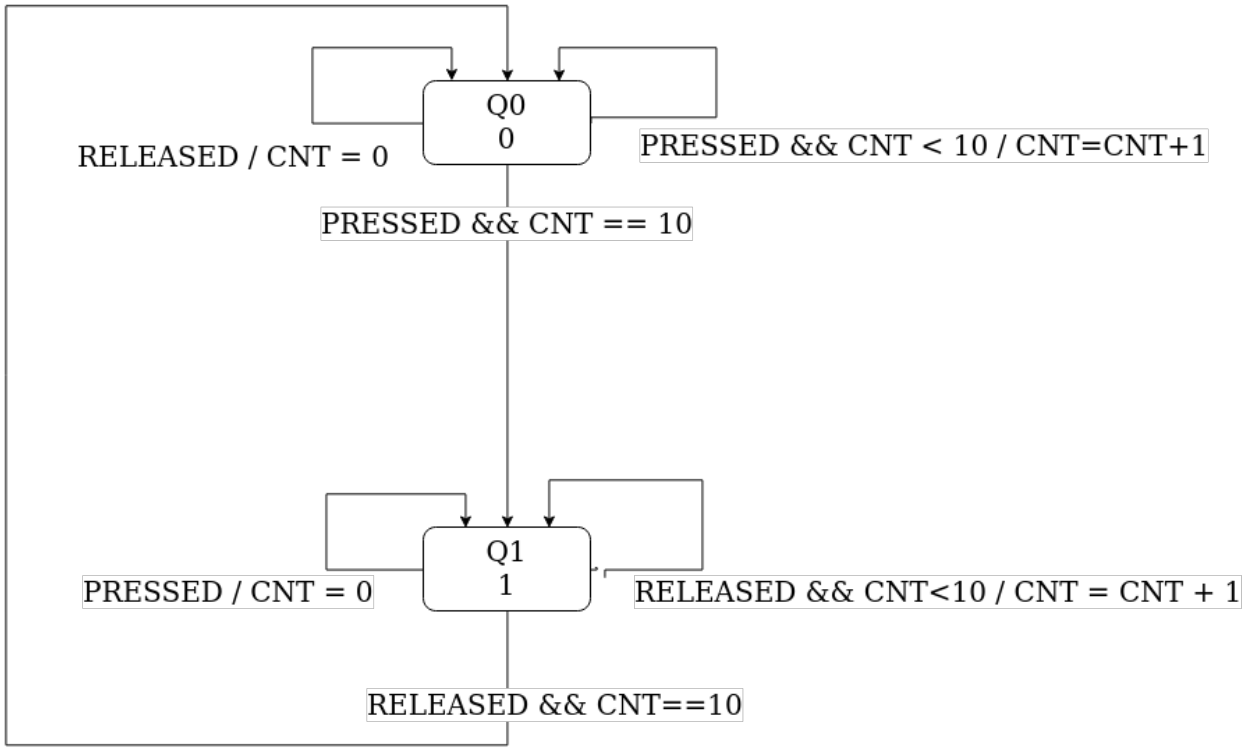
A sequential approach would be.

```
    // Wait until button is pressed
   while (Button_Read()==BUTTON_PRESSED) {}

cnt = 0;
do {
      Delay(T1MS);
      if ( Button_Read()!=BUTTON_PRESSED) // Not pressed
          cnt = 0;                        // Start again
      else                                // Pressed
```

```
            cnt++;                          // Count
    } while (cnt <= DEBOUNCE_TIME);
```

A better approach, is to use interrupts. In this case it is `better` to use a timer interrupt. Generally, the use of interrupt generated by the pin attached to the button is a bad idea.

A state machine diagram describes the working of the interrupt routine.




```
int state = 0;      // 0 = Q0 (BUTTON RELEASED),  1 = Q1 (BUTTON PRESSED)
int cnt;

void Button_processing(void) {
uint32_t b;
    b = Button_Read();
    switch(state) {
    case 0:         // Released
        if( b == PRESSED ) {
            if( cnt == 10 ) {
                state = 1;
            } else {
                cnt++;
```

```
            }
        } else { // Not Pressed
            cnt = 0;
        }
    case 1:         // Pressed
        if( b == PRESSED ) {
            cnt = 0;
        } else { // Not Pressed
            if( cnt == 10 ) {
                state = 0;
            } else {
                cnt++;
            }
        }
    }
}

int Button_Status(void) {

    if( state == 1 )
        return 1;
    else
        return 0;
}
```

A clever approach is to use bit masks instead of counters.

In the source code (button.c and button.h) one can find the generalization of the above for more than one switch.

More information

- Ganssle. My favorite software debouncers[7]
- Ganssle. Debounce Part 1[8]
- Ganssle. Debounce Part 2[9]
- Embarcados. Leitura de chaves[10]
- Hackday. Debounce code. one post to rule them all[11]
- Cleghorn. Button Debouncer[12]

7 https://www.embedded.com/electronics-blogs/break-points/4024981/My-favorite-software-debouncers
8 http://www.ganssle.com/debouncing.htm
9 http://www.ganssle.com/debouncing-pt2.htm
10 https://www.embarcados.com.br/leitura-de-chaves-debounce/
11 https://hackaday.com/2010/11/09/debounce-code-one-post-to-rule-them-all/
12 https://github.com/tcleg/Button_Debouncer

- http://ohm.bu.edu/~pbohn/__Engineering_Reference/debouncing.pdf

12

Serial Communication using polling

Serial interface using USB

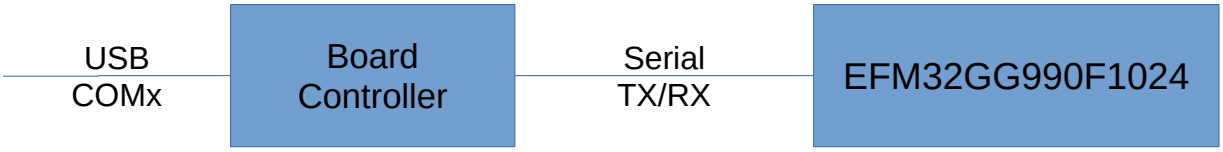
Before developing application that uses the serial-USB bridge, it is necessary to update the firmware in
the board controller.

Using Simplicity

1. With the board disconnected, start Simplicity
2. If there is a message, suggesting to upgrade, accept it.
3. Connect the board
4. In the *Device* panel, right click over the *J-Link Silicon Labs* line and select *Device Configuration*.
5. Accept the offer to upgrade the firmware.

Using J-Link

Serial interface



In Linux, a device named `ttyACM0` appears in the `/dev` folder. In Windows, a new COM device appears. There is a permission problem in some Linux machines. There are two ways to solve it. One is add the user to the dialout group with the following command

```
sudo usermod -a -G dialout $(USER)
```

The dialout group can access the serial devices (`/dev/tty*`). Other way is add a file (`50-segger.rules`) with following rules to /etc/udev/rules.d folder.

```
KERNEL=="ttyACM[0-9]*",MODE="0666"
```

The serial interface between EFM32GG and the board controller has a fixed configuration: 115200 bps, 8 bits, no parity, 1 stop bit, no hardware flow control.

The board is wired to use the `PE0 (TX)` and `PE1 (RX)` pins. There is an enable signal on PF7. It must be set to 1 to communication between the EFM32GG and Board Controller happens. It signals a TS3A4751 SPST (Single Pole - Single Throw) Analog Switch to connect the EFM32GG and the Board Controller.
The I/O pins used for the UART0 can be choosen from three sets (Location) and one of them include PE0 and PE1. Attention: they are controlled by UART0, not USART0.

Signal	0	1	2
U0_RX	PF7	PE1	PA4
U0_TX	PF6	PE0	PA3

The same device is used to implement a serial bootloader. See AN0042[13] and AN0042-KB[14].

The UART uses the HFPERCLK clock signal, which is derived from the HFClock with a divisor specified by the field `HFPERCLKDIV` in register `CMU_HFPERCLKDIV`. The signal must be enable by setting the `HFPERCLKEN` in register `CMU_HFPERCLKDIV`. The divisor is a power of 2 in the range 1 to 512, specified by a value 0 to 8 in the field `HFPERCLKDIV`.
The HFClock source can be the HFXO (external high frequency crystal oscillator), HFRCO (internal high frequency RC oscillator, the default) or a low frequency source: LFXO (external low frequency crystal oscillator) or LFRCO (internal low frequency RC oscillator).

Clock source	Frequency
HFRCO	1-28 MHz (default: 14 MHz nominal)
HFXO	4-48 MHz (crystal)
LFRCO	32768 Hz nominal
LFXO	32768 Hz (crystal)

The crystal on the STK3700 has a 48 MHz frequency, the maximal frequency of the device.

There is an important note in the Item 13.1 of Application Note *USART/UART - Asynchronous mode* (AN0045).

> *Often, the HFRCO is too unprecise to be used for communications. So using the HFXO with an external crystal is recommended when using the EFM32 UART/USART.*

13 https://www.silabs.com/application-notes/an0042-efm32-usb-uart-bootloader.pdf
14 https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2017/09/01/use_an0042_bootloade-N9Zn

In some cases, the internal HFRCO can be used. But then careful considerations should be taken to ensure that the clock performance is acceptable for the communication link.

The reason for this note is hidden in the page 24 of the datasheet of the EFM32GG990F1024. RC based oscillators are inherently not very precise. They have a thermal drift, which must be compensated by calibration.

Clock source	Nominal frequency	Minimum frequency	Maximum frequency	Obs
LFXO	32768 Hz	31290 Hz	34280 Hz	
HFRCO	1 MHz	1.15 MHz	1.25 MHz	
HFRCO	7 MHz	6.48 MHz	6.72 MHz	
HFRCO	11 MHz	10.8 MHz	11.2 MHz	
HFRCO	14 MHz	13.7 MHz	14.3 MHz	default
HFRCO	21 MHz	20.6 MHz	21.4 MHz	
HFRCO	28 MHz	27.5 MHz	28.5 MHz	

To configure UART0:

1. Configure pins PE0 and PE1 to be controlled by UART0. Configure UART0 to use LOC1 for RX and TX pins.
2. Configure oversampling factor 16 in `UART0_CTRL` (higher is better), setting OVS to 00.
3. Speed is configured by setting the `CLKDIV` field (in `UART0_CLKDIV`) according the formula

$$speed = \frac{f_{HFPERCLK}}{Oversampling\,(1 + CLKDIV/4)}$$

or

4. The formulas are different from EMF32GG Reference Manual, which use the whole register value. The formulas above use field values.
5. Configure stop bits setting `STOPBITS` field in `UART0_CTRL` to 01.
6. Configure 8 bits data setting `DATABITS` field in `UART0_CTRL` to 0101.
7. Configure no parity by setting `PARITY` field in `UART0_CTRL` to 00.
8. Enable transmit operations by writing `TXEN` to `UART0_CMD`.
9. Enable receiving operations by writing `RXEN` to `UART0_CMD`.

To use it in polling mode:

1. To transmit: test `TXC` in `UART0_STATUS`. If set, write data to `UART0_TXDATA`.
2. Yo receive: test `RXDATAV` in `UART0_STATUS`. If set, get data from `UART0_RXDATA`.

More information

EFM32 STK Virtual COM port[15]
Using stdio on Silicon Labs platforms[16]

15 https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2015/07/06/efm32_stk_virtualco-aT2m
16 https://os.mbed.com/teams/SiliconLabs/wiki/Using-stdio-on-Silicon-Labs-platforms

13

Serial Communication using interrupts

Interrupt based serial communication

This describes an interrupt based implementation of serial communication. The device used for serial communication using USB is the UART0 (not USART0). It uses the PE0 and PE1 pins for RX and TX, respectively, and the PF7 to enable a transceiver.
The EMF32GG990F1024 has two interrupts for the UART0: `UART0_TX_IRQn` and `UART0_RX_IRQn`. The generated interrupts calls the routines `UART0_RX_IRQHandler` and `UART0_TX_IRQHandler`. Both use a FIFO buffer to store the characters handled. This avoid to need to wait. When the buffer is full, the extra characters are discarded. The FIFO buffer is defined in buffer.h and buffer.c.

Enabling interrupts

The following code enables the interrupt. Part of code is used to configure the UART, and other the NVIC. To avoid problems with already triggered interrupts, the interrupts are disabled and then reenabled.

```
// Enable interrupts on UART
UART0->IFC = (uint32_t) -1;
UART0->IEN |= UART_IEN_TXC|UART_IEN_RXDATAV;

// Enable interrupts on NVIC
NVIC_SetPriority(UART0_RX_IRQn,RXINTLEVEL);
NVIC_SetPriority(UART0_TX_IRQn,TXINTLEVEL);
NVIC_ClearPendingIRQ(UART0_RX_IRQn);
NVIC_ClearPendingIRQ(UART0_TX_IRQn);
NVIC_EnableIRQ(UART0_RX_IRQn);
NVIC_EnableIRQ(UART0_TX_IRQn);

// Disable and then enable RX and TX
UART0->CMD  = UART_CMD_TXDIS|UART_CMD_RXDIS;
UART0->CMD  = UART_CMD_TXEN|UART_CMD_RXEN;
```

Hazard conditions

When using interrupt, it is possible to access data while it is being modified. This can lead to data corruption and unexpected behaviors. In this case, the access to buffer must be done without interrupts by using the `ENTER_CRITICAL` and `EXIT_CRITICAL` macros pairs. They are just `__disable_irq()` and `__enable_irq()`, respectively.

Transmitting a char

The interrupt routine is called when there is a modification in `TXC` flag in `STATUS` register and enable by setting the `TXC` bit in `IEN`.

```
void UART0_TX_IRQHandler(void) {
uint8_t ch;

    // if data in output buffer and transmitter idle, send it
    if( UART0->IF&UART_IF_TXC ) {
        if( UART0->STATUS&UART_STATUS_TXBL ) {
            if( !buffer_empty(outputbuffer) ) {
                // Get from output buffer
                ch = buffer_remove(outputbuffer);
                UART0->TXDATA = ch;
            }
       }
       UART0->IFC = UART_IFC_TXC;
    }
}
```

To transmit, just put the data in the buffer and if it was empty, raise an interrupt.

```
void UART_SendChar(char ch) {
    if ( buffer_empty(outputbuffer) ) {
        while ( (UART0->STATUS&UART_STATUS_TXBL) == 0 ) {}
        UART0->TXDATA = ch;
    } else {
        ENTER_ATOMIC();
        buffer_insert(outputbuffer,ch);
        EXIT_ATOMIC();
    }
}
```

Receiving a char

The interrupt routine is straightforward.

```
void UART0_RX_IRQHandler(void) {
uint8_t ch;
```

```
    if ( UART0->IF&(UART_IF_RXDATAV|UART_IF_RXFULL) ) {
        // Put in input buffer
        ch = UART0->RXDATA;
        (void) buffer_insert(inputbuffer,ch);
    }
}
```

The API for receiving a char includes a non block `UART_GetCharNoWait` to get the char of buffer if there is one there. Otherwise returns 0.

```
unsigned UART_GetCharNoWait(void) {
int ch;

    if( buffer_empty(inputbuffer) )
        return 0;

    ENTER_ATOMIC();
    ch = buffer_remove(inputbuffer);
    EXIT_ATOMIC();
    return ch;
}
```

Notes

1. Before developing application that uses the serial-USB bridge, it is necessary to update the firmware in the board controller.
2. The interface appears as a `/dev/ttyACMx` in Linux machines and `COMx` in Window machines.

More information

1. EFM32 STK Virtual COM port[17]
2. Using stdio on Silicon Labs platforms[18]
3. AN0045[19]

17 https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2015/07/06/efm32_stk_virtualco-aT2m
18 https://os.mbed.com/teams/SiliconLabs/wiki/Using-stdio-on-Silicon-Labs-platforms
19 http://www.silabs.com/Support Documents/TechnicalDocs/AN0045.pdf

14

Mini Standard I/O Package

Input/output standard routines

This implements a minimal set of routines provided in the standard input/output library:

- `int printf(const char *fmt, ...)`: equivalent to standard printf but with no support for float, field sizes, zero filling, etc.
- `int puts(const char *s)`: same as standard puts
- `int fputs(const char *s, void *ignored)`: same as standard fputs, but redirects all output to stdout
- `char *fgets(char *s, int n, void *ignored)`: same as standard fgets but gets all data from stdin.

These routines make use of `putchar` and `getchar` routines, which must be provided by the application.

Conversion routines

In the files `conv.[ch]` the following (non standard) set of conversion and character manipulation was implemented:

- `int atoi(char s)*` : standard routine to convert a string of digits in *s* considering it as a decimal integer and returns its value.
- `void itoa(int v, char s)*` : non standard routine to convert the integer *v* into a decimal representations in *s*.
- `void utoa(unsigned x, char s)*` : non standard routine to convert the unsigned *x* into a string in *s*.
- `int hextoi(char s)`: *non standard routine to convert a string in* s* considering it as a hexadecimal integer and returns its value.
- `int itohex(unsigned x, char s)*` : non standard routine to convert the integer *x* into a hexadecimal representations in *s*.

Character classification routines

Besides these routines, the following routines for classification of char was implemented:

- `int isspace(int c)`: returns 1 if *c* is a space, tab, carriage return or line feed, otherwise returns 0.
- `int isdigit(int c)`: returns 1 if *c* is a decimal digital, otherwise returns 0.
- `int isxdigit(int c)`: returns 1 if *c* is a decimal digit or a character in the range *a* to *f* or *A* to *F* , otherwise 0.
- `int isalpha(int c)`; returns 1 if *c* is a character in the range *a* to *z* or *A* to *Z*.
- `int isupper(int c)`: returns 1 if c is in the range *A* to *Z*, otherwise returns 0.
- `int islower(int c)`: returns 1 if c is in the range *a* to *z*, otherwise returns 0.
- `int iscntrl(int c)`: returns 1 if *c* is a control character (range 0 to 31, includes tab, carriage return, line feed, etc), otherwise returns 0.
- `int isalnum(int c)`: returns 1 if *c* is in the range *a* to *z*, or *A* to *Z* or a digit, otherwise returns 0.

15

Newlib

A port of stdio library to embedded systems

To enhance portability, a multiple layer systems composed of libraries of functions was added to the C language. In UNIX/Linux machines, the different layers are shown below.

Application
C Lib
POSIX
Operating System
Hardware

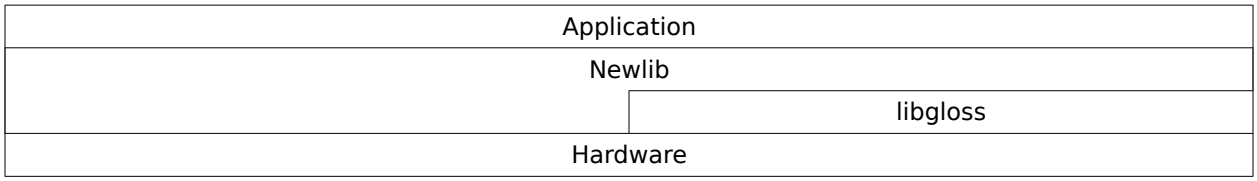
For clarification, both the C standard library and the POSIX provide functions to open, read, write and close files. The C standard library provides much more, including a stream based abstraction to I/O and a mathematical library.

Library	Open a file	Read from file	Write to file	Close file
C lib	fopen	fread	fwrite	fclose
POSIX	open	read	write	close
C99/C++ conform	_open	_read	_write	_close

In UNIX/Linux machines there is, for historical and compatibility, the symbols (function names, struct names, etc.) share the same space, which can lead to unexpected name collision (Try to give the name write to a function in your application!!). The correct way is to preprend an underscore (_) to these names, because all symbols started by underscore are reserved for the implementation.

Newlib

A full implementation of the standard C library is part of the embedded arm gcc. It is based on newlib[20]. The capacity of the library is only limited by the board hardware. For example, if there is no file system, all functions related to files are limited to `stdin` and `stdout`/`stderr`.



The libgloss is the glue between software and hardware. It includes routines that depends only on the microcontroller architecture (e.g. Cortex-M3) and routines that depends on the microcontroller and board.

The newlib includes the following libraries:

- `libc.a`: standard c library
- `libm.a`: standard mathematical library
- `libnosys.a`: library with stub
- `libc-nano.a`: small footprint libc optimized for bare bone systems (without operating system)
- `libg.a`: other name for `libc.a`
- `libgcc.a`: routines needed in the code generated by the compiler.

There are different versions of the newlib, in the lib folder. The compiler will use one of them according the command line parameters.

```
./thumb/v8-m.main/fpv5-sp/hard/libc.a
./thumb/v8-m.main/fpv5-sp/softfp/libc.a
./thumb/v8-m.main/fpv5/hard/libc.a
./thumb/v8-m.main/fpv5/softfp/libc.a
./thumb/v8-m.main/libc.a
./thumb/v7-ar/fpv3/hard/libc.a
./thumb/v7-ar/fpv3/softfp/libc.a
./thumb/v7-ar/libc.a
./thumb/v8-m.base/libc.a
./thumb/v7e-m/fpv5/hard/libc.a
./thumb/v7e-m/fpv5/softfp/libc.a
./thumb/v7e-m/fpv4-sp/hard/libc.a
./thumb/v7e-m/fpv4-sp/softfp/libc.a
./thumb/v7e-m/libc.a
./thumb/v6-m/libc.a
./thumb/v7-m/libc.a
```

20 https://sourceware.org/newlib

```
./thumb/libc.a
./hard/libc.a
./libc.a
```

One of the problems of using a library is that, inadvertently, a lot of dependencies is generated and the code size exploeds. For example, `printf` includes support for floating point, and using it, all support for floating point is added, and this is big for processors without floating point units.

To avoid it, newlib provides a `iprintf` without support for floating point.

Compiling using newlib

In the previous examples, the library was not used. Symbols from the library were defined outside it and there was no need to use objects from the library.

NewliB depends only on a set of functions, that mimic the corresponding POSIX functions. The functions that must be implemented are shown it the table below.

_exit	close	environ	execve	fork	fstat	getpid	isatty	kill
link	lseek	open	read	sbrk	stat	times	unlink	wait
write								

Most of these function can be only a stub, returning a (non) fatal error or other meaningful result. Exceptions are the read and write function, that redirect the input and output to UART;

The makefile is configured to use the nano version of the libraries. Commenting the line `SPECFLAGS= --specs=nano.specs` the normal libraries are used. The spec file specifies a rewriting of the linker parameters.
The size of the code generated can be obtained by the command

```
arm-none-eabi-size gcc/uart-cdc.axf
```

The results of using nano or normal version of the libraries and the ministdio (last example) can be compared in the table below.

Section	Size using nano	Size using normal	Size using ministdio
Code	9079	27437	5282
Data	108	2484	8
BSS	276	356	256
Total	9463	30277	5546

Implementation

The main part of this implementation is located in the syscalls.c file. It implements a minimal set of routines needed by the newlib. They corresponds to the POSIX layer in UNIX/Linux machines.

```
void _main(void);
void _exit(void);
int _close(int file);
int _execve(char *name, char **argv, char **env);
int _fork(void);
int _fstat(int file, struct stat *st);
int _getpid(void);
int _isatty(int file);
int _kill(int pid, int sig);
int _link(char *old, char *new);
int _lseek(int file, int ptr, int dir);
int _open(const char *name, int flags, int mode);
int _read(int file, char *ptr, int len);
caddr_t _sbrk(int incr);
int _stat(char *file, struct stat *st);
int _times(struct tms *buf);
int _unlink(char *name);
int _wait(int *status);
int _write(int file, char *ptr, int len);
```

According to the C Standard the following files are already opened when an application starts. They corresponds to files with

Files opened at start	Description	file parameter
stdin	standard input	0
stdout	standard output (buffered)	1
stderr	standard output (unbuffered)	2

For this case, most of this routines are implemented as dummy ones. The others ignores the file parameter, because there is only one interface, the UART.

The _main routine is called by the initialization routine, before calling the main in application code. It is used to initialize the UART.

The code uses the UART routines of project 13-UART, but preparing for a future expansion, it uses a

set of functions to map the functions used to the UART routines

```
void SerialInit(int chn)           { UART_Init();               }
void SerialWrite(int chn, char c)  { UART_SendChar(c);          }
int  SerialRead(int chn)           { return UART_GetChar();     }
int  SerialStatus(int chn)         { return UART_GetStatus();   }
int  SerialFlush(int chn)          { return UART_Flush();       }
```

Only the following routines are actually implemented.

```
void _main(void) {
    SerialInit(0);
}

void _exit(void) {
    while (1) {}        // eternal loop
}

int _read(int file, char *ptr, int len) {
    if( ttyconfig & TTY_UNBUFF )
        return tty_read_un(0,ptr,len);
    return tty_read_lb(0,ptr,len);  // Default
}

int _write(int file, char *ptr, int len) {
    return tty_write(0,ptr,len);
}
```

A TTY layer is used to implement the editing capabilities of a terminal, specially for input. Although this could be implement in the UART interface, a separation permits a more robust code, eases the testing e a future reuse.

The TTY interface for output is direct. The only interesting aspect is the mapping CR to CR/LF.

```
int tty_write(int chn, char *ptr, int len) {
int cnt;
char ch;
int i;
    cnt = 0;
    for (i = 0; i < len; i++) {
        ch = *ptr++;
        if( (ch == '\n') && ttyconfig&TTY_OCRLF ) {
            SerialWrite(chn,'\r');
            cnt++;
        }
        SerialWrite(chn,ch);
        cnt++;
```

```
    }
    return cnt;
}
```

For input, there are two implementations, one used a line buffered approach and the other a unbuffered approach. The unbuffered version is direct. The line buffered permits the use of backspace to erase a character and a CR means the End of Line.

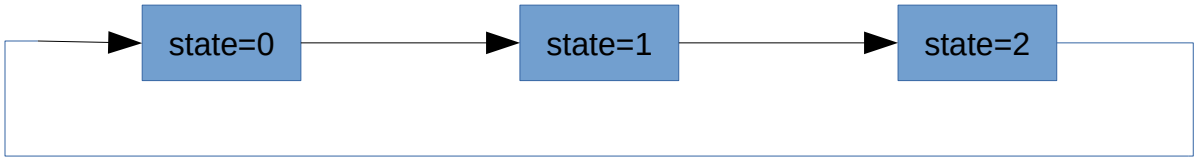
```
int tty_read_lb(int chn, char *ptr, int len) {
int cnt;
int ch;

    cnt = 0;
    SerialFlush(chn);
    while ( ((ch=SerialRead(chn)) != '\n') && (ch!='\r') ) {
        if( ch == TTY_BS ) {
            if( cnt > 0 ) {
                cnt--;
                SerialWrite(chn,'\b');
                SerialWrite(chn,' ');
                SerialWrite(chn,'\b');
            }
        } else {
            if( ttyconfig&TTY_IECHO )
                SerialWrite(chn,ch);
            if( cnt < len )         // overflow characters not stored
                ptr[cnt++] = ch;
        }
    }

    if( cnt < len ) {
        ptr[cnt++] = '\n';
    }
    if(ttyconfig&TTY_ICRLF ) {
        SerialWrite(chn,'\r');
        SerialWrite(chn,'\n');
    }
    return cnt;
}
```

16

Time Triggered Systems

A time triggered kernel

This is the 7th version of Blink. The main modification is the use of a time triggered scheduler. In the last example, a periodical interrupt (SysTick) was used to implement a simple state machine.



Standard approach

In the last example a counter was used to divide further the interrupts, because the divider must be less then 2^24 (=16777216) and because other tasks can have different periods.

```
#define DIVIDER 1000
...
void SysTick_Handler(void) {
static int counter = 0;            // must be static
static int8_t state = 0;            // must be static

    if( counter != 0 ) {
        counter--;
    } else {
        // Processing
        switch(state) {
        case 0:
            LED_Toggle(LED1);
            state = 1;
            break;
```

```
        case 1:
            LED_Toggle(LED2);
            state = 2;
            break;
        case 2:
            LED_Write(0,LED1|LED2);
            state = 0;
            break;
        }
        counter = DIVIDER-1;
    }
```

Dividing the interrupts is generally repeated for other tasks with different periods. An important point, is that doing the processing in the interrupt is a bad idea, because during the processing the interrupts are disabled. This approach is called back-foreground, with the interrupt routines as foreground tasks and the main routine as the background test and should only be used in very simple cases.

Better alternative

A better alternative is the interrupt processing routine to signalize that a task must be run and the task runs outside the interrupt routine.

```
#define DIVIDER 1000
...
int run = 0;
...
void SysTick_Handler(void) {
static int counter = 0;            // must be static
static int8_t state = 0;            // must be static

    if( counter != 0 ) {
        counter--;
    } else {
        run++;
        counter = DIVIDER-1;
    }
...
    #define DIVIDER 1000
...
void Blinker(void) {
static int8_t state = 0;            // must be static

    switch(state) {
    case 0:
        LED_Toggle(LED1);
```

```
        state = 1;
        break;
    case 1:
        LED_Toggle(LED2);
        state = 2;
        break;
    case 2:
        LED_Write(0,LED1|LED2);
        state = 0;
        break;
    }
}
...
int main(void) {
    ...
    while(1) {
        if( run ) {
            Blinker();
            run--;
        }
    }
}
```

Time Triggered Scheduler

The approach shown before is the basis of a time triggered task scheduler according Pontt[21].
A list of tasks, including information about period and the counter value is maintained through an API. A task is then started (by calling the corresponding function) when the period is over as indicated by the run variable.

```
typedef struct {
    void    (*function)(void);          ///< pointer to function with task code
    INT     counter;                    ///< counter (in ticks). when 0, task should be
run
    INT     period;                     ///< period (in ticks). when 0, task in run once
    INT     run;                        ///< run counter. when above 1, task is delayed
} Task_t;
```

The scheduler is initialized through Task_Init. Tasks are inserted using the Task_Add routine. During the interrupt, it is only necessary to update the list by calling Task_Update. And in the main loop, a Task_Dispatch must be called.

```
void SysTick_Handler(void) {
    Task_Update();
}
```

21 https://www.safetty.net/products/publications/pttes

```
void Blinker(void) {
static int8_t state = 0;            // must be static

    switch(state) {
    case 0:
        LED_Toggle(LED1);
        state = 1;
        break;
    case 1:
        LED_Toggle(LED2);
        state = 2;
        break;
    case 2:
        LED_Write(0,LED1|LED2);
        state = 0;
        break;
    }
}

...

int main(void) {
    ...
    Task_Init();
    Task_Add(Blinker,1000,0);

    SysTick_Config(SystemCoreClock/DIVIDER);

    /* Blink loop */
    while (1) {
        Task_Dispatch();
    }

}
```

17

Using Protothreads

Multithreading with one stack

This is the 8th version of Blink. The main modification is the use of protothreads. This emulates the quasi parallel execution of tasks by implementing a cooperative multitasking kernel.

It uses a 'hack' call *Duff's Device* (see below) to emulate multitasking by repeated calls. At each call, the executes resumes at the point of the last one. Due the nature of this hack, all local variables must be static, retaining value between calls.

Blinker Task

The code for the Blinker Task is very similar of the main code in example 03. It has a drawback. There is a glitch when the counter reaches the limit for a 32 bit unsigned integer. At 1 kHz, about 49 days.

```
#include "pt.h"

struct pt pt;
uint32_t threshold;

#define PT_DELAY(T) threshold = timer_counter+(T);
PT_WAIT_UNTIL(pt,timer_counter>=threshold);
...

PT_THREAD(Blinker(struct pt *pt)) {

    PT_BEGIN(pt);

    while(1) {
        // Processing
        LED_Toggle(LED1);
        PT_DELAY(1000);

        LED_Toggle(LED2);
```

```
        PT_DELAY(1000);

        LED_Write(0,LED1|LED2);
        PT_DELAY(1000);

        }
    PT_END(pt);
}
```

The main routine is

void main(void) {

```
PT_INIT(pt);
```

Duff's Device

The following is a valid C code and is the base of the protothread library.

```
void send(char *to, char *from, int count)  {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
                } while (--n > 0);
    }
}
```

18

Using FreeRTOS

A free real time kernel

FreeRTOS is an open source real time preemptive kernel with a small footprint. It can be downloaded from www.freertos.org.

It is composed of a small set of C source and header files common to all platforms and another set, specific to a target. There are also different implementations of memory managers.

Folder	Files
FreeRTOSv10.0.0/FreeRTOS/Source/include/	croutine.h mpu_prototypes.h stack_macros.h deprecated_definitions.h mpu_wrappers.h StackMacros.h event_groups.h portable.h FreeRTOS.h projdefs.h stream_buffer.h list.h queue.h task.h message_buffer.h semphr.h timers.h
FreeRTOSv10.0.0/FreeRTOS/Source	croutine.c tasks.c event_groups.c list.c queue.c stream_buffer.c timers.c
FreeRTOSv10.0.0/FreeRTOS/Source/portable/GCC/ARM_CM3	port.c and portmacro.h
FreeRTOSv10.0.0/FreeRTOS/Source/portable/MemMang	heap_1.c heap_2.c heap_3.c heap_4.c heap_5.c

FreeRTOS is highly configurable. The main configuration is done by editing the FreeRTOSConfig.h file. Several examples of it can be found in the Demo folder. A tip is to copy one for a similar architecture to the project folder. In this case, the LM3S102 from Texas is a Cortex M3 too.

The contents of the FreeRTOSConfig.h includes the following:

```
#define configUSE_PREEMPTION                1
#define configUSE_IDLE_HOOK                 0
#define configUSE_TICK_HOOK                 0
#define configCPU_CLOCK_HZ                  ( ( unsigned long ) 48000000 )
#define configTICK_RATE_HZ                  ( ( TickType_t ) 1000 )
```

```
#define configMINIMAL_STACK_SIZE           ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE              ( ( size_t ) ( 2048 ) )
#define configMAX_TASK_NAME_LEN            ( 3 )
#define configUSE_TRACE_FACILITY           0
#define configUSE_16_BIT_TICKS              0
#define configIDLE_SHOULD_YIELD            0
#define configUSE_CO_ROUTINES              0
#define configMAX_PRIORITIES               (2)
#define configMAX_CO_ROUTINE_PRIORITIES    (2)

/* Set the following definitions to 1 to include the API function, or zero to exclude the
API function. */

#define INCLUDE_vTaskPrioritySet           0
#define INCLUDE_uxTaskPriorityGet          0
#define INCLUDE_vTaskDelete                0
#define INCLUDE_vTaskCleanUpResources      0
#define INCLUDE_vTaskSuspend               0
#define INCLUDE_vTaskDelayUntil            0
#define INCLUDE_vTaskDelay                 1

#define configKERNEL_INTERRUPT_PRIORITY         255 //
configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero!!
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    191 // equivalent to 0xa0, or priority 5.
```

The most important parameters are:

- `configCPU_CLOCK_HZ`
- `configTOTAL_HEAP_SIZE`
- `configTICK_RATE_HZ`

Memory managers

There are five implementation of memory managers. A short description of each is shown below.

Implementation	Description
heap_1.c	Does not free memory
heap_2.c	Uses a best fit allocation algorithm, but does not combine adjacent areas
heap_3.c	Uses the malloc/free routines provided
heap_4.c	Full implementation but not deterministic
heap_5.c	Can use multiple memory pools

Setting up a project

- Setup a project folder
- Configure FreeRTOS using the file FreeRTOSConfig.h in the project folder

- Add the FreeRTOS source files in FreeRTOS/Source to project. It is not advisable to copy them to project folder.
- Add the port.c in FreeRTOS/source/portable/GCC/ARM_CM3 to project.
- Add the heap_X.c file to project (X can be 1,2,3,4 or 5, see above).
- Add the following folders to the include search path: FreeRTOS/source/include, FreeRTOS/source/portable/GCC/ARM_CM3

When using CMSIS, it is necessary to add these lines to the FreeRTOSConfig.h file.

```
#define vPortSVCHandler                 SVC_Handler
#define xPortPendSVHandler              PendSV_Handler
#define xPortSysTickHandler             SysTick_Handler
```

Modifications to Makefile

1. Insert the following lines at the beginning

```
############################################################################################
# FreeRTOS Configuration                                                                   #
############################################################################################

#
# FreeRTOS Dir
#
FREERTOSDIR=../../FreeRTOSv10.0.0

# Virtual path: Where to find the source files
VPATH=      $(FREERTOSDIR)/FreeRTOS/Source:\
            $(FREERTOSDIR)/FreeRTOS/Source/portable/GCC/ARM_CM3:\
            $(FREERTOSDIR)/FreeRTOS/Source/portable/MemMang

# FreeRTOS Include Path
FREERTOSINCPATH=    $(FREERTOSDIR)/FreeRTOS/Source/include \
                    $(FREERTOSDIR)/FreeRTOS/Source/portable/GCC/ARM_CM3/

#
# There are 5 alternatives for heap management
#
#     heap_1.c: Does not free memory
#     heap_2.c: Uses a best fit allocation algorithm, but does not combine adjacent areas
#     heap_3.c: Uses the malloc/free routines provided
#     heap_4.c: Full implementation but not deterministic
#     heap_5.c: Can use multiple memory pools
```

```
#
# FreeRTOS Source Files
# Some files can be ommited if their funcionalities are not used
#
# The most important files are:
#   tasks.c                   Task management routines. Generally a must.
#   list.c                    List management routines. Include if you use lists.
#   queue.c                   Queue management routines. Include if you use queues.
#   timers.c                  Timer management routines. Include if you use timers.
# Additional files are:
#   stream_buffer.c
#   event_groups.c
#   croutine.c                Coroutines management routines. include if you use them
#
FREERTOSFILES=      tasks.c list.c queue.c timers.c \
                    event_groups.c stream_buffer.c  \
                    croutine.c \
                    portable/GCC/ARM_CM3/port.c \
                    portable/MemMang/heap_2.c

#
# The same files with a relative path from the present directory
#
FREERTOSSRCFILES= $(addprefix $(FREERTOSDIR)/FreeRTOS/Source/,$(FREERTOSFILES))
```

1. Modify the line

   ```
   OBJFILES=     $(addprefix ${OBJDIR}/,$(SRCFILES:.c=.o))
   ```

   to

   ```
   OBJFILES=     $(addprefix ${OBJDIR}/,$(notdir $(SRCFILES:.c=.o) \
                 $(FREERTOSSRCFILES:.c=.o)))
   ```

2. Modify the line

   ```
   INCLUDEPATH= ${CMSISDEVINCDIR} ${CMSISINCDIR}
   ```

   to

   ```
   INCLUDEPATH= . $(FREERTOSINCPATH) ${CMSISDEVINCDIR} ${CMSISINCDIR}
   ```

Code

Before starting a small observation about symbol names in FreeRTOS. An old naming scheme, used in the 90s, called Hungarian Notation, is still used in FreeRTOS. The idea is to incorporate some type information in the name of a function or a variable. For example, function, whose name started with v returns no value, i.e., return void.

The main program must have the following structure:

1. Include FreeRTOS header files

```
#include "FreeRTOS.h" /* Must come first. */
#include "task.h"     /* RTOS task related API prototypes. */
#include "queue.h"    /* RTOS queue related API prototypes. */
#include "timers.h"   /* Software timer related API prototypes. */
#include "semphr.h"   /* Semaphore related API prototypes. */
```

1. Include project header files, including manufacturer supplied files
2. Define constants and parameters
3. Implement a setup hardware function (initialization)
4. Optionally, implement tasks, but this can be done in other source files.
5. Configure FreeRTOS by setting parameters in the FreeRTOS.h file. For example, it is possible to enable or disable the functions listed below.

```
#define INCLUDE_vTaskPrioritySet        0
#define INCLUDE_uxTaskPriorityGet       0
#define INCLUDE_vTaskDelete             0
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            0
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1
```

6. Implement the main function as bellow

```
int main(void) {

    // Setup hardware

    // Create queues and semaphores


    // Create tasks

    // Start FreeRTOS

    vTaskStartScheduler();
```

```
    // Just in case

    while (1) {}


}
```

1. Tasks are functions with the `void task(void *param)` prototype and have the following structure

```
void Task(void *param) {
// local variables

// initialization


// infinite loop
while(1) {   // Could be for(;;) {

    // actions

    // wait for something. vTaskDelay, vQueueReceive, xSemaphoreGet, etc.

    }
    // never reached
 }
```

`param` can be used to share code. For example, the same routine can be used for different UARTs or ADCs.

The *wait for something* is essential. Without it, tasks with lower priority would not run.

More information

- FreeRTOS[22]
- FreeRTOS on Cortex M3/4[23]

22 https://www.freertos.org/
23 https://www.freertos.org/RTOS-Cortex-M3-M4.html

19

Using uC/OS-II

uC/OS II is a proprietary[24] real time preemptive kernel with a small footprint. It can be downloaded from www.micrium.com for non commercial purposes. If it is used in a commercial product, there are license fees.

The main features of uC/OS II are:

- Preemptive
- Coded in (mainly) C
- Portable
- Fixed priority scheduling
- Up to 255 tasks
- Semaphores, Mailboxes, Queues, Timers and other mechanisms
- Small fooprint, but each task must have a stack

It was programmed mainly in C, with a small part, dependent on the compiler and processor, in C and Assembly. This enables the port of this kernel to other processors and compilers.

To port the uC/OS to another system, the following requirements must be attended (See uC/OS II book, page 350):

1. The processor has a C compiler that generates reentrant code.
2. Interrupts can be disabled and enabled from C.
3. The processor supports interrupts and can provide an interrupt that occurs at regular intervals (typically between 10 and 100Hz).
4. The processor supports a hardware stack that can accommodate a fair amount of data (possibly many kilobytes).
5. The processor has instructions to load and store the stack pointer and other CPU registers, either on the stack or in memory

Download

It can be downloaded from Micrium website (register required).

24 Micrium was acquired by Silicon Labs and it is free to use uc/os on Silicon Labs microcontrollers.

It is composed of a small set of C source and header files common to all platforms and another set, specific to a target. There are also different implementations of memory managers.

Folder	Files
Micrium/Software/uCOS-II/Source	os_cfg_r.h os_dbg_r.c os_mbox.c os_mutex.c os_sem.c os_time.c ucos_ii.c os_core.c os_flag.c os_mem.c os_q.c os_task.c os_tmr.c ucos_ii.h
Micrium/Software/uCOS-II/Ports/ARM-Cortex-M3/Generic/GCC	os_cpu.h os_cpu_a.asm os_cpu_c.c os_dbg.c
Project Folder	os_cfg.h includes.h

The uC/OS II version from Micrium website lacks the port folder entirely. Ports available in the Micrium site do not use GCC compiler. You can find many Cortex M3 port for GCC searching the internet, for example, this one at github[25].

But a better approach is to use the uC/OS-II port in the Gecko SDK. You can find it in the GECKOSDK/util/third_party/micrium/uCOS-II subfolder.

Configuration

uC/OS is highly configurable. The main configuration is done by editing the file os_cfg.h.

Modifications to Makefile

1. Insert the following lines at the beginning
2. Modify the line
3. Modify the line

Tasks

The uC/OS has two low priorities already installed:

- Idle task with the lowest possible priority (`OS_LOWEST_PRIO`)
- Stats task with the next lowest possible priorit (`OS_LOWEST_PRIO-1`), which can be disabled by setting `OS_TASK_STAT_EN` to 0.

The tasks are identified by their priorities. So, no two tasks can have the same priority.

But beware. From uC/OS II book:

> *If your application uses the statistic task, you must call*

25 https://github.com/huyugui/STDFS/tree/master/lcd-demo/ucos/uCOS-II/Ports/ARM-Cortex-M3/Generic/GCC

> *OSStatInit() (see OS_CORE.C) from the first and only task created in your application during initialization. In other words, your startup code must create only one task before calling OSStart(). From this one task, you must call OSStatInit() before you create your other application tasks. The single task that you create is, of course, allowed to create other tasks, but only after calling OSStatInit().*

The initialization task must have the highest priority, i.e. 0, because if it creates a higher priority task, a switch occurs immediately, and it can takes a long time to create the other tasks.
A task generally implements an infinite loop as bellow.

```
void Task(void *pdata) {
    // local variables

    // initialization

    // infinite loop
    while(1) {  // Could be for(;;)

        // actions

        // wait for something: OSTaskDelay, OSSemPend, etc.

     }
 }
```

Another important point from uC/OS II book (page 134) is

> *You must enable ticker interrupts after multitasking has started, that is, after calling OSStart() . In other words, you should initialize ticker interrupts in the first task that executes following a call to OSStart(). A common mistake is to enable ticker interrupts after OSInit() and before OSStart() , as shown in Listing 3.22. Potentially, the tick interrupt could be serviced before µC/OS-II starts the first task. At this point, µC/OS-II is in an unknown state, so your application crashes.*

The start task, generally called `StartTask`, has the following pattern.

```
void StartTask(void *param) {
#if OS_CRITICAL_METHOD == 3
```

```
    OS_CPU_SR cpu_sr;
#endif
    // Initialize
    ...

    // Start timer
    ...

    // Initialize statistics
    OSStatInit();

    // Create tasks
    OSCreateTask(Task1,.......);
    OSCreateTask(Task2,.......);
    OSCreateTask(Task3,.......);

    while(1) { // Could be for(;;)

        OSTimeDly(OS_TICKS_PER_SEC); // Could be OSTimeDlyHMSM(0,0,1,0);
    }
}
```

The infinite loop can be replaced by `OSTaskDel(0)`, which autodelete the `TaskStart`.

Code

1. Implement an os_app.h.

```
#ifndef OS_APP_H
#define OS_APP_H
/* Nothing yet */
#endif
```

1. Implement an os_conf.h header file. Better copy one from an example in Examples folder or the os_conf_r.h from sources folder and modify it.
2. Implement a main function with the following pattern:

```
```
void TaskStart(void *param) {
       #if OS_CRITICAL_METHOD == 3
       OS_CPU_SR cpu_sr;
       #endif
       // Initialize
       ...

       // Start timer
```
```

```
        ...

        // Initialize statistics
        OSStatInit();

        // Create tasks
        OSCreateTask(Task1,.......);
        OSCreateTask(Task2,.......);
        OSCreateTask(Task3,.......);

        OSTaskDel(0);       // Auto destroy
}

}

        void main(void) {
        // Local variables
        ...
        // Setup hardware but do not enable timer
        ...

        // Initialize uC/OS
        OSInit();

        // Create Semaphores, Events, MessageQueue etc
        ...

        // Create a single task (TaskStart) with priority 0 (highest), which will create
        the other tasks
        OSTaskCreate(TaskStart, void *) 0, TaskStartStack, 0);

        // Start uC/OS
        OSStart();

        }
        ~~~
```

More information

uC/OS II on Cortex M[26]
uC/OS II Book[27]
uC/OS II on Cortex M4 Book[28]
uC/OS II port on Cortex M3 and M4[29]

26 https://www.state-machine.com/qpc/ucos-ii.html
27 https://www.micrium.com/download/µcos-ii-the-real-time-kernel-2nd-edition/
28 https://www.micrium.com/download/µcos-ii-the-real-time-kernel-for-the-freescale-kinetis/
29 https://github.com/tony/gpc/tree/master/3rd_party/uCOS-II

us/os II port on Cortex M3 Application Note[30]

30 https://www.element14.com/community/docs/DOC-35592/l/micrium-an1018-application-note-for-µcos-ii-and-the-arm-cortex-m3-processors

20

Using uC/OS-III

A newer version of uC/OS

uC/OS-iii is a proprietary real time preemptive kernel with a small footprint. It can be downloaded from www.micrium.com for non commercial purposes. If it is used in a commercial product, there are license fees.

The main features of uC/OS III are:

- Preemptive
- Coded in (mainly) C
- Portable
- Fixed priority scheduling
- Unlimited number of tasks
- Semaphores, Mailboxes, Queues, Timers and other mechanisms only limited by RAM
- Small fooprint, but each task must have a stack

It was programmed mainly in C, with a small part, dependent on the compiler and processor, in C and Assembly. This enables the port of this kernel to other processors and compilers.

To port the uc/os to another system, the following requirements must be attended (See uc/os-iii book, page 350):

1. The processor has a C compiler that generates reentrant code.
2. Interrupts can be disabled and enabled from C.
3. The processor supports interrupts and can provide an interrupt that occurs at regular intervals (typically between 10 and 100Hz).
4. The processor supports a hardware stack that can accommodate a fair amount of data (possibly many kilobytes).
5. The processor has instructions to load and store the stack pointer and other CPU registers, either on the stack or in memory.

It depends on two other packages: os/lib and os/cpu.

Download

It can be downloaded from Micrium website (register required). But the most visible version

available does not include the required packages and lacks the port folder entirely. A better idea is to download it from the download center[31] or more specifically, the Cortex M3 port page[32], one with the same or similar architecture. In this case, Texas Instruments LM3S9B92 is a good approximation, because it is a Cortex M3 too.

Another possibility is to download the Simplicity Studio[33]. It includes a full uc/os iii for EFM32 devices.

The uc/os-iii package

The uC/OS iii package, after unpacking, has the following folders:

- uC/OS-III: the uC/OS-III
- uC/LIB: non standard functions and macros to test characters, generate random numbers, manage memory, manipulate string, etc.
- uC/CPU: manager (some) timers and clock frequency. Specifies some parameters like stack growth.
- uC/CSP: (optionally) chip support funcions like GPIO pins, etc.

It is composed of a small set of C source and header files common to all platforms and another set, specific to a target. There are also different implementations of memory managers.

Folder	Files
Micrium/Software/uCOS-III/Source	os_cfg_app.c os_core.c os_dbg.c os_flag.c os_int.c os_mem.c os_msg.c os_mutex.c os_pend_multi.c os_prio.c os_q.c os_sem.c os_stat.c os_tick.c os_time.c os_tmr.c os_var.c os_h os_type.h
Micrium/Software/uCOS-III/Ports	os_cpu_c.c os_cpu_a.asm os_cpu.h os_cpu_a.inc
Micrium/Software/uC-CPU	cpu_core.c cpu_core.h cpu_def.h
Micrium/Software/uC-CPU/ARM-Cortex-M3/GCC	cpu.h cpu_a.asm cpu_c.c
Micrium/Software/uC-LIB	lib_ascii.c lib_ascii.h lib_def.h lib_math.c lib_math.h lib_mem.c lib_mem.h lib_str.c lib_str.h
bsp	bsp.c bsp.h bsp_int.c bsp_int.h
project	app.c app_hooks.c includes.h app_cfg.h cpu_cfg.h lib_cfg.h os_app_hooks.h os_cfg.h os_cfg_app.h

Configuration

uC/OS-III is highly configurable. The configuration is done by editing header files.

31 https://www.micrium.com/downloadcenter/
32 https://www.micrium.com/downloadcenter/download-results/?searchterm=pa-cortex-m3&supported=true
33 https://www.silabs.com/products/development-tools/software/simplicity-studio

File	Description
app_cfg.h	
cpu_cfg.h	
csp_cfg.h	
lib_cfg.h	
os_app_hooks.h	
os_cfg.h	
os_cfg_app.h	
os_type.h	
os_cfg.h	
os_app	

Modifications to Makefile

1. Insert the following lines at the beginning
2. Modify the line
3. Modify the line

Tasks

The uc/os has two low priorities already installed:

- Idle task with the lowest possible priority (OS_LOWEST_PRIO)
- Stats task with the next lowest possible priority (OS_LOWEST_PRIO-1), which can be disabled by setting OS_TASK_STAT_EN tgo 0.
- Tick task

The tasks implements an infinite loop as bellow

```
void Task(void *pdata) {
    // local variables

    // initialization

    // infinite loop
    while(1) {  // Could be for(;;)

        // actions

        // wait for something: OSTaskDelay, OSSemPend, etc.
```

```
    }
}
```

From uC/OS III book, page 75

> *A few important points are worth noting. For one thing, you can create as many tasks as you want before calling OSStart(). However, it is recommended to only create one task as shown in the example because, having a single application task allows µC/OS-III to determine the relative speed of the CPU. This allows µC/OS-III to determine the percentage of CPU usage at run-time. Also, if the application needs other kernel objects such as semaphores and message queues then it is recommended that these be created prior to calling OSStart(). Finally, notice that interrupts are not enabled.*

This is emphasized again in page 134.

> *If the application uses the statistic task, it should call OSStatTaskCPUUsageInit() from the first, and only application task created in the main() function as shown in Listing 5-5. The startup code should create only one task before calling OSStart(). The single task created is, of course, allowed to create other tasks, but only after calling OSStatTaskCPUUsageInit()." This means that the OSStatTaskCPUUsageInit must be run without any other task to calibrate the measurement of CPU usage.*

The initialization task must have the highest priority, i.e.. 0, because if it creates a task with a higher priority, a switch occurs immediately, and it can takes a long time to create the other tasks. This task can enter an infinite loop or kill itself (using `OSTaskKill`).
The start task, generally called `StartTask`, has the following pattern.

```
void TaskStart(void *param) {
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    // Initialize
    ...

    // Start timer
    ...
```

```
    // Initialize statistics. Must be  run as only task.
    OSStatTaskCPUUsageInit();

    // Create tasks
    OSCreateTask(Task1,.......);
    OSCreateTask(Task2,.......);
    OSCreateTask(Task3,.......);

    while(1) { // Could be for(;;)

        OSTimeDly(OS_TICKS_PER_SEC); // Could be OSTimeDlyHMSM(0,0,1,0);
    }
}
```

Code

1. Implement `os_app.h` file.

```
#ifndef OS_APP_H
#define OS_APP_H
/* Nothing yet */
#endif
```

1. Implement the os_conf.h header file. Better copy one from an example in Examples folder or the `os_conf_r.h` from source folder and modify it.
2. Implement a main function with the following pattern:

```
void main(void) {
    // Local variables
    ...

    // Setup hardware but do not enable timer
    ...

    // Initialize uc/os
    OSInit();

    // Create Semaphores, Events, MessageQueue etc
    ...

    // Create a single task (TaskStart) with priority 0 (highest),
    // which will create the other tasks
    OSTaskCreate(TaskStart, void *) 0, TaskStartStack, 0);
```

```
    // Start uc/os
    OSStart();
}
```

More information

uC/OS[34]
uC/OS III books[35]
Myths about uC/OS[36]
uC/OS ii port on Cortex M3 Application Note[37]

34 https://www.micrium.com/
35 https://www.micrium.com/books/ucosiii/
36 http://www.electronicdesign.com/embedded-revolution/11-myths-about-cos
37 https://www.element14.com/community/docs/DOC-35592/l/micrium-an1018-application-note-for-µcos-ii-and-the-arm-cortex-m3-processors

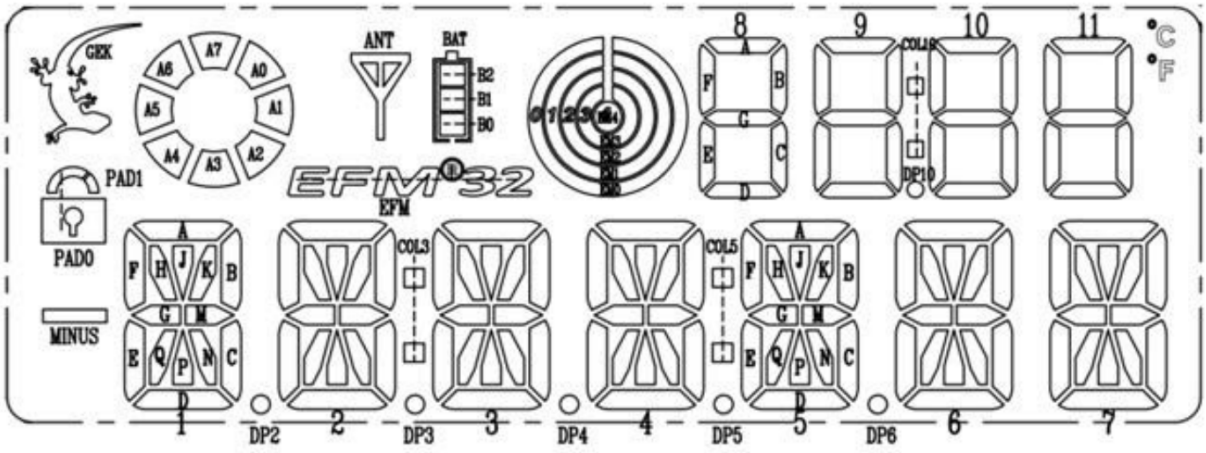
21

Using the LCD display

A HAL for the LCD

This implements an HAL (hardware abstraction Layer) for the LCD on the STK3700 board. The STK3700 board has a 160 segment LCD display. It is multiplexed and user 20 segments pins and 8 common pins of the microcontroller. The onchip LCD controller can driver up to 288 segments using 8 common signals. It uses non standard voltages to get null average voltages to avoid electrolysis. See AN0057.0: EFM32 Series 0 LCD Driver[38].

The layout of the LCD display is shown below.



The pins used are shown below.

Pin	MCU Signal	LCD Signal
PA11	LCD_SEG39	S19
PA10	LCD_SEG38	S18
PA9	LCD_SEG37	S17
PA8	LCD_SEG36	S16

38 https://www.silabs.com/documents/public/application-notes/AN0057.pdf

PA7	LCD_SEG35	S15
PB2	LCD_SEG34	S14
PB1	LCD_SEG33	S13
PB0	LCD_SEG32	S12
PD12	LCD_SEG31	S11
PD11	LCD_SEG30	S10
PD10	LCD_SEG29	S9
PD9	LCD_SEG28	S8
PA6	LCD_SEG19	S7
PA5	LCD_SEG18	S6
PA4	LCD_SEG17	S5
PA3	LCD_SEG16	S4
PA2	LCD_SEG15	S3
PA1	LCD_SEG14	S2
PA0	LCD_SEG13	S1
PA15	LCD_SEG12	S0
-	-	-
PB6	LCD_COM7	COM0
PB5	LCD_COM6	COM1
PB4	LCD_COM5	COM2
PB3	LCD_COM4	COM3
PE7	LCD_COM3	COM4
PE6	LCD_COM2	COM5
PE5	LCD_COM1	COM6
PE4	LCD_COM0	COM7

Only two groups of segments pins are used: one between LCD12 and LCD19 and other, between LCD28 and LCD39. The LCD segments and the corresponding activation are show below. Note the reversal of numbering of the common lines.

LCD Map

Seg	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
COM0	DP2	1E	1D	2E	2D	3E	3D	4E	4D	DP5
COM1	DP4	1Q	1N	2Q	2N	3Q	3N	4Q	4N	5E

COM2	DP3	1P	1C	2P	2C	3P	3C	4P	4C	5Q
COM3	COL3	1G	1M	2G	2M	3G	3M	4G	4M	5P
COM4	MIN	1F	1J	2F	2J	3F	3J	4F	4J	5G
COM5	PAD1	1H	1K	2H	2K	3H	3K	4H	4K	5F
COM6	GEK	1A	1B	2A	2B	3A	3B	4A	4B	5H
COM7	A7	A6	A5	A4	A3	A2	A1	A0	EFM	5A
Seg	**S10**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**
COM0	5D	DP6	6D	7E	7D	11A	10A	9A	8A	EM2
COM1	5N	6E	6N	7Q	7N	11F	10F	9F	8F	EM4
COM2	5C	6Q	6C	7P	7C	11B	10B	9B	8B	COL10
COM3	5M	6P	6M	7G	7M	11G	10G	9G	8G	DP10
COM4	5J	6G	6J	7F	7J	11E	10E	9E	8E	PAD0
COM5	5K	6F	6K	7H	7K	11C	10C	9C	8C	EM3
COM6	5B	6H	6B	7A	7B	11D	10D	9D	8D	EM1
COM7	COL5	6A	ANT	BAT	.C	.F	B1	B0	B2	EM0

API (Application Programming Interface)

The LCD display is divided in segments:

- an alphanumerical field with 14 segments characters (position 1 to 7)
- a numerical field with 7-segments digits (position 8 to 11)
- a special field containing miscellaneous signs. Internally codified as positions 12 to 14.

The functions implemented are:

Special signs encoding	
LCD_GECKO	Gecko in the upper left
LCD_MINUS	Minus in the left
LCD_PAD0	Lower part of pad
LCD_PAD1	Upper part of pad Z
LCD_ANTENNA	Antenna
LCD_EMF32	EFM32 Logo in the center
LCD_BAT0	Battery level 0
LCD_BAT1	Battery level 1
LCD_BAT2	Battery level 2
LCD_ARC0	Arc segment 0
LCD_ARC1	Arc segment 1

LCD_ARC2	Arc segment 2
LCD_ARC3	Arc segment 3
LCD_ARC4	Arc segment 4
LCD_ARC5	Arc segment 5
LCD_ARC6	Arc segment 6
LCD_ARC7	Arc segment 7
LCD_TARGET0	Target circle 0 (external)
LCD_TARGET1	Target circle 1
LCD_TARGET2	Target circle 2
LCD_TARGET3	Target circle 3
LCD_TARGET4	Target center
LCD_C	Celsius sign
LCD_F	Fahrenheit sign
LCD_COLLON3	Collon at the left of alphanumerical field
LCD_COLLON5	Collon at the right of alphanumerical field
LCD_COLLON10	Collon in the middle of the numerical field
LCD_DP2	Decimal point left of character at position 2
LCD_DP3	Decimal point left of character at position 3
LCD_DP4	Decimal point left of character at position 4
LCD_DP5	Decimal point left of character at position 5
LCD_DP6	Decimal point left of character at position 6
LCD_DP10	Decimal point in the middle of the numerical field
For the encoding below the parameter v is the value to set	range
LCD_ARC	0 to 7
LCD_BATTERY	0 to 2
LCD_LOCK	0 to 1
LCD_TARGET	0 to 5

Implementation

A multistep approach to display a character in a certain position is used. It uses many tables:

- Segments for characters for 14 and 7 segment displays.
- Controller segment and common numbers to display a display segment in a certain position

- Controllers segments for all common signal that must be cleared before displaying a segment in a certain position.

The tables are consulted in order show above. To write a new char, a sequence of 8 write operations is needed, because the position must be cleared.

More information

Example code for Segment LCD on EFM32 Giant Gecko development kit EFM32GG-DK3750 (Without using Emlib)[39]

Segmented LED Display - ASCII Library[40]
AN0009: EFM32 and EZR32 Series Getting Started[41]
AN0057.0: EFM32 Series fo LCD Driver[42]

39 http://embeddedelectrons.blogspot.com.br/2016/12/example-code-for-segment-lcd-on-efm32.html
40 https://github.com/dmadison/LED-Segment-ASCII
41 http:////www.silabs.com/documents/public/application-notes/an0009.0-efm32-ezr32-series-0-getting-started.pdf
42 https://www.silabs.com/documents/public/application-notes/AN0057.pdf

22

Better newlib support

Problems in the implementation

In the previous implementation, there is a violation of encapsulation rules when the clock frequency is changed. After the changing of clock frequency, many devices need to be reconfigured. This is the case of the UARTs. After the clock is changed by calling SystemCoreClockSet, the UART_Init routine must be called to reconfigure the UART.

The main purpose of the newlib module is to hidden the implementation details, including the UART interface. The initialization of UART is done before the main routine starts (See _main in syscalls.c) and the need to call UART_Init in the main procedure leads to run errors difficult to analyze. This problem happens in the drivers for other devices.

In order to avoid this problem, one has to implement:

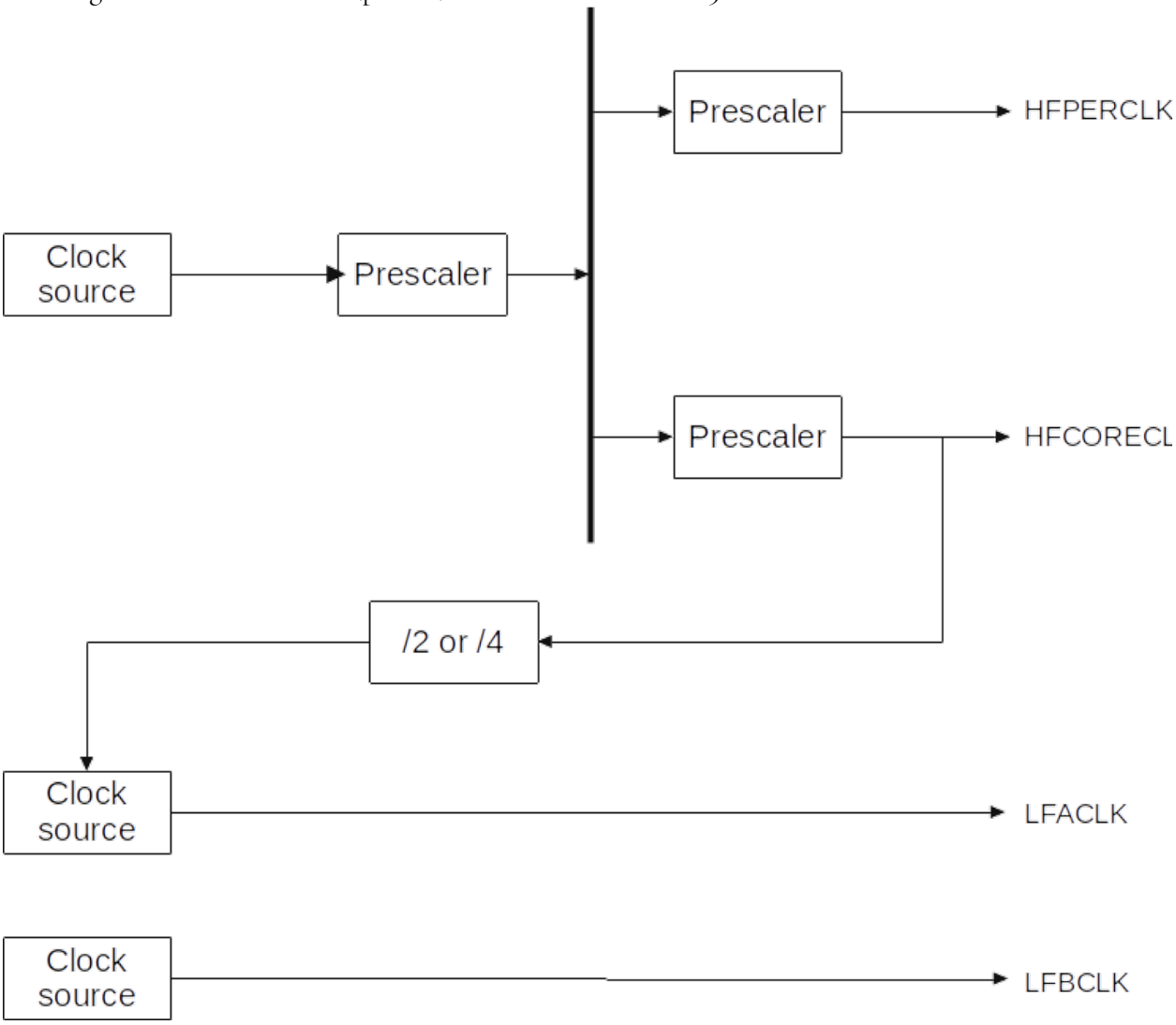
1) A callback mechanism. Whenever a clock changed, a registered routine is called and does the reconfiguration. There must be two callbacks functions, one is called before the change (it permits to turn off any transmission or processing) and the other after (it does the actual reconfiguration).
2) Functions to handle this change and register them, to be called, when this change happers. For the UART, there must be a callback function to reconfigure the baud rate according the new clock frequency and another one, to be called before the change, to stop all UART operations.

To accommodate the modifications without disrupting the previous projects, the following files were renamed in order to avoid confusion with files with the same names in other projects.

clock_efm32gg_ext.c -> clock_efm32gg_ext2.c
clock_efm32gg_ext.h -> clock_efm32gg_ext2.h
uart.c -> uart2.c
uart.h -> uart2.h

Modification in the clock management routines

The diagram below shows the (partial) clock tree in the EFM32GG.



There is interdependences between the various clock signals. So, whenever HCLK changes, possibly all other clock signals change too. This is controlled by a bit mask, that signalizes which clock has changed and if a reconfiguration is needed.

The function `ClockRegisterCallback` was added. It registers two functions `pre` and `post`. The `pre` function is called before the clock change, for example, to stop all transmission and/or processing. The `post` function is called after the clock change. It is used to reconfigure the device according the new clock configuration. Both functions are only called when a clock change occurs as specified by the `clock` parameter.

```
ClockRegisterCallback( uint32_t clock, void (*pre)(uint32_t), void (*post)(uint32_t))
```

Both functions have a parameter specifying which clock changes occurred. The is a OR of the following values: `CLOCK_CHANGED_HFCLK`, `CLOCK_CHANGED_HFCORECLK`, `CLOCK_CHANGED_HFPERCLK`, `CLOCK_CHANGED_HFCORECLKLE`, `CLOCK_CHANGED_LFCLKA` and `CLOCK_CHANGED_LFCLKB`.

There are other modifications:

1) SystemCoreClockSet function was renamed to ClockSetCoreClock. So all function names in the clock module start with Clock. An alias was added as a preprocessor symbol to clock_efm32gg2.h to enable compatibility.
2) The following aliases were added to clock_efm32gg_ext2.h as preprocessor symbols. The clock acronyms used in the reference manual and data sheet.

   ClockSetHFCORECLK is an alias to ClockSetCoreClock
   ClockGetHFCORECLK is an alias to ClockGetCoreClockFrequency
   ClockGetHFPERCLK is an alias to ClockGetPeripheralClockFrequency

Modification in the UART routines

The main modification is the addition of callback routines. The `pre` routine just turn off the receive and the transmission operations of the UART. The `post` routine reconfigures the baud rate according the new clock configuration.

Another modification is the possibility to use the same set of routines to control the two UARTS on the microcontroller: UART0 and UART1. This is enabled by the preprocessor USE_UART1 parameter.

Modification in syscalls.c

Only the mapping Serial* to UART* functions is modified, because UART* functions have a `uart` parameter. For now, all operations are still mapped to UART0.

References

EMF32GG Reference Manual
EFM32GG990 Data Sheet

23

Getting the CPU temperature

ADC Converters

The EFM32GG99 is part of the EMFG32 Gecko Series 0 Family. The members of this family have a 12-bit SAR (Successive Approximation Register) ADC (Analog-Digital Converter).

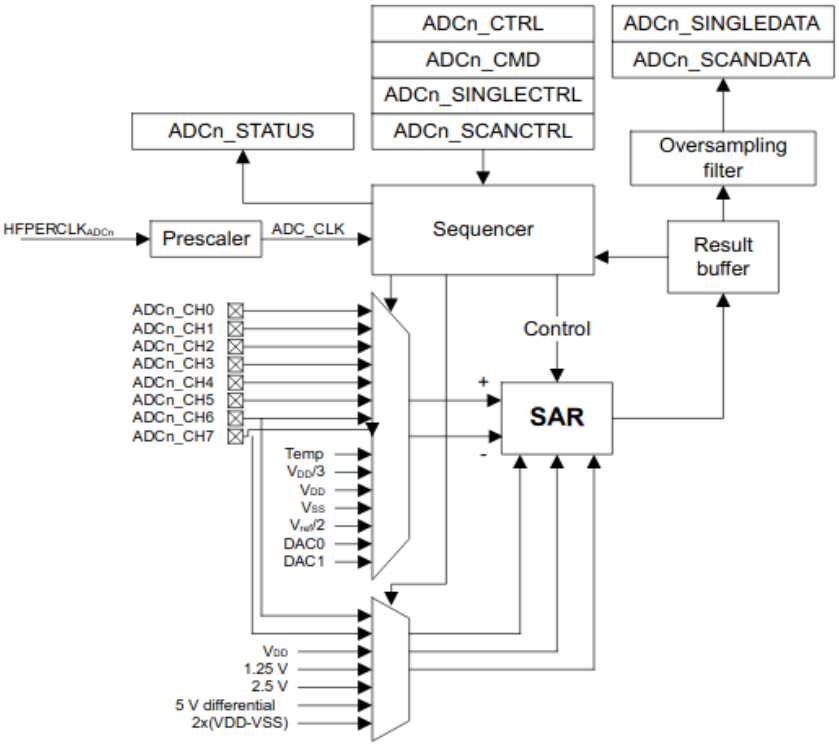


Figure 1.1. ADC Overview of EFM32 Giant Gecko

The ADC has 15 channels, 8 external and 6 internals channels.

Channel	Signal	Input
0	CH0	External
1	CH1	External
2	CH2	External
3	CH3	External
4	CH4	External
5	CH5	External
6	CH6	External
7	CH7	External
8	Temperature	Internal
9	VDD/3	Internal
10	VDD	Internal
11	VSS	Internal
12	Vref/2	Internal
13	DAC0	Internal
14	DAC1	Internal

The 8 external signals can be used as differential pairs.

Channel	Positive Signal	Negative Signal
CH0_1	CH0	CH1
CH2_3	CH2	CH3
CH4_5	CH4	CH5
CH6_7	CH6	CH7

For the conversion, a stable reference voltage is necessary. There are the options shown bellow.

Reference Voltage	Notes	Single/Differential
V_DD	internal buffered	Single/Differential
1.25 V	internal	Single
2.5 V	internal	Single
5 V differential	internal	Differential
EXTSINGLE	external (CH6)	Single
2xEXTDIFF	external (CH6-CH7)	Differential
2xVDD	unbuffered	Differential

There are basically two modes of operation:

- Single Sample. Each conversion must be started by a command
- Scan mode. A set of conversions is started by a command

On the EFM32GG990F1024 the ADC ports use the PORT D pins, and those, on the STK3700, appear on the Expansion Header (EXP Header). The diagram below show the main signals on those pins. See the Datasheet for detailed information about each pin.

Top	2	4	6	8	10	12	14	16	18	20	Top
		ADC0 DAC0 US1TX	ADC1 DAC1 US1RX	ADC2 US1CLK	ADC3 US1CS	ADC4 LEU0TX	ADC5 LEU0RX	ADC6 I2CSDA			
	VMCU	PD0	PD1	PD2	PD3	PD4	PD5	PD6	5V	3V3	
Bottom	GND	PC0	PC3	PC4	PC5	PB11	PB12	PC6	PD7	GND	Bottom
		USTX I2CSDA	US2RX	US2CLK I2CSDA	US2CS I2CSCL				ADC7 US1TX I2CSCL		
	1	3	5	7	9	11	13	15	17	19	

Timing

A conversion takes TCONV = (TA+N)xOSR cycles, where TA is the acquisition time, N, the number of bits and OSR, the oversampling factor.

The duration of a cycle depends on the clock source used and the prescaler factor. The clock source is the Peripheral Clock (HFPERCLK), shared with many other peripherals. There is a prescaler specific for ADC, that can have a value in the range between 1 and 128. But the clock frequency must be greater than 32 KHz and less than 13 MHz.

The is also an automatic warm up time after enabling the ADC or changing the reference voltage. It is a 1 us, plus an additional 5 us if an internal reference (bandgap) voltage is used (1.25 or 2.5 V).

There are 4 different warm-up modes:

- NORMAL: The ADC and reference voltage are shut off when there is no samples waiting.
- FASTBG: No warm up for the voltage reference but the accuracy is reduced.
- KEEPSCANREFWARM: The bandgap reference is kept warm during a scan. But there there must be a warm up before starting another scan. Only for bandgap reference voltage.
- KEEPADCWARM: The ADC and the bandgap reference are kept warm for a scan. Only for bandgap reference voltage.

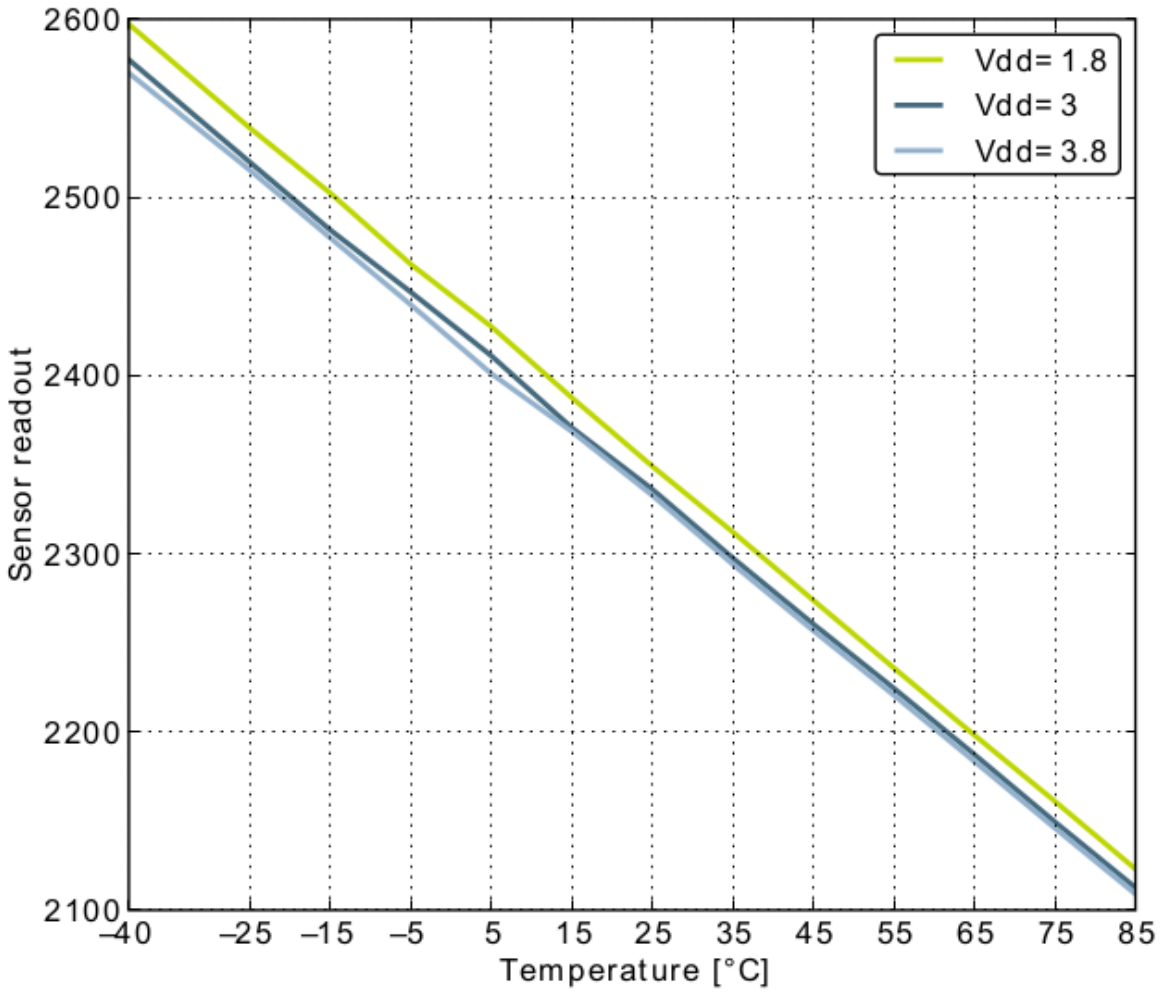
The warm-up is done automatically by the ADC, when the correct number of clock cycles is written in the TIMEBASE field of ADC0_CTRL. The number of cycles must correspond to, at least, a 1 us delay.

There are bits in the ADC_STATUS register, that gives information about the warm-up status.

Bit	Symbol	Description
WARM	ADC_STATUS_WARM	ADC is warmed up when 1
SCANREFWARM	ADC_STATUS_SCANREFWARM	Reference is warmed up for scan mode when 1
SINGLEREFWARM	ADC_STATUS_SINGLEREFWARM	Reference is warmed up for single mode when 1

Temperature measurement

The Channel 8 is connected to an internal temperature sensor. The figure below shows the relation between ADC reading and temperature.



An approximate temperature value can be obtained by interpolating between the points (-40,2600) and (85,2125). The slope of the curve is approximately give by the formula

$$slope = \frac{2600 - 2125}{-40 - 85} = -\frac{475}{125}$$

and an uncalibrated value, corresponding to a reading, can be given by

$$T_{Celsius} = -40 + \frac{125}{475}(reading - 2600)$$

This sensor is calibrated during manufacturing and the calibration parameters are written to a ROM area, called Device information (DI) page.

To get the temperature it is necessary to use the following formula.

$$T_{Celsius} = \text{CAL\_TEMP\_0} - \frac{V_{REF}}{4096 \times \text{TGRAD\_ADCTH}_V}(\text{ADC0\_TEMP\_0\_READ\_1V25} - \text{ADC\_Result})$$

The `CAL_TEMP_0` and `ADC0_TEMP_0_READ_1v25` values can be found in the DI page (See bellow). The processor used has the following values.

CAL_TEMP_0	25
ADC0_TEMP_0_1V25	2335

The `TGRAD_ADCTH` can be expressed in two different units and both can be found on the device datasheet.

Name	Value	Units
TGRAD_ADCTH_V	-1.92	mV/C
TGRAD_ADCTH_U	-6.3	units/C

This corresponds to the line slope, which is expressed in Celsius degrees by ADC unit, and is given by

$$m = \frac{V_{REF}}{4096 \times \text{TGRAD\_ADCTH}_V} = \frac{1.25\,V}{4096\,u \times (-0{,}00192\,V/C)} = \frac{1}{-6.3u/C} = \frac{1}{\text{TGRAD\_ADCTh}_U}$$

To avoid the use of float point data (variables and constants, the parameters are expressed as ratios. So, instead of 6.3, one uses

$$\frac{63}{10}$$

Calibration

During manufacturing a set of additional information are written in the DI page, that can be used to calibrate the measurements.

Name	Address	Information	Size
ADC0_CAL	0x0FE08040	??	32
ADC0_BIASPROG	0x0FE08058	??	32
CAL_TEMP_0	0x0FE081B2	Calibration temperature (7:0)	16
ADC0_CAL_1V25	0x0FE081B4	Gain (14:8) / Offset (6:0)	16
ADC0_CAL_2V5	0x0FE081B6	Gain (14:8) / Offset (6:0)	16
ADC0_CAL_VDD	0x0FE081B8	Gain (14:8) / Offset (6:0)	16
ADC0_CAL_5VDIFF	0x0FE081BA	Gain (14:8) / Offset (6:0)	16
ADC0_CAL_2xVDD	0x0FE081BC	Gain (14:8) / Offset (6:0)	16
ADC0_TEMP_0_READ_1V25	0x0FE081BE	Temperature reading (15:4)	16

The values should be accessed using the DEVINFO structure as seen in `efm32gg_devinfo.h`. But there is inconsistency between the info in the reference manual and the structure defined in the source code.

So, to get a value, one should use the following code

```
uint16_t val16 =  *( (uint16_t *) address )
uint32_t val32 =  *( (uint32_t *) address )
```

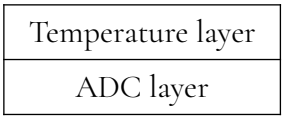
Or define them as macros

```
#define  GET16(ADDR)      *( (uint16_t *) (ADDR) )
#define  GET32(ADDR)      *( (uint32_t *) (ADDR) )
...
uint16_t val16 = GET16(address);
uint16_t val16 = GET16(address);
```

During reset, the values for 1V25 are automatically written in ADCo_CAL register. The values for calibration (ADCo_CAL_*) must be written to the ADCo_CAL register in order the ADC module automatically corrects the value read.

Implementation

It is a two layer implementation.



The Temperature layer has the following functions.

Temperature_Init	Initializes the temperature module and the corresponding ADC channel. The parameter is the frequency to be used for the ADC. It must be in the range 32 Khz-13 MHz.
Temperature_GetRawValue	Returns the value without calibration, i.e., the reading of the ADC channel
Temperature_GetCalibratedValue	Returns the calibrated value of temperature

The ADC layer is straightforward.

ADC_Init(freq, config)	Initializes the ADC unit
ADC_Read(ch)	Reads channel ch waiting for the conversion completion
ADC_StartReading(ch)	Starts a conversion on channel ch.
ADC_GetReading(ch)	Reads channel ch waiting for the conversion completion that was started by ADC_StartReading.

References

EMF32GG Reference Manual
EFM32GG990 Data Sheet
AN0021: Analog to Digital Converter(ADC)

A

Installing the toolchain

Tools needed for development

The tools needed for developing embedded system for ARM microcontrollers are:

1. Toolchain (Compiler/Linker/Debugger/Tools) supporting the microcontroller family.
2. CMSIS library
3. Files specific to the microcontroller
4. Flash tools
5. Debugging tools
6. Debugging interface

Toolchain

A toolchain for ARM is the ARM GNU CC, which includes compiler, assembler, linker, debugger, and many utilities. It can be downloaded from ARM GNU GCC[43] or installing using a package manager like apt. Until recently this site was hosted in Launchpad[44].
The toolchain is a version of the GNU CC ported to ARM processors in a project backed by ARM itself. The are other alternatives: one free alternative comes from Linaro[45].
After installing the toolchain, the following applications are avaliable:

```
arm-none-eabi-addr2line  arm-none-eabi-gcc-ar      arm-none-eabi-nm
arm-none-eabi-ar         arm-none-eabi-gcc-nm      arm-none-eabi-objcopy
arm-none-eabi-as         arm-none-eabi-gcc-ranlib  arm-none-eabi-objdump
arm-none-eabi-c++        arm-none-eabi-gcov        arm-none-eabi-ranlib
arm-none-eabi-c++filt    arm-none-eabi-gcov-dump   arm-none-eabi-readelf
arm-none-eabi-cpp        arm-none-eabi-gcov-tool   arm-none-eabi-size
arm-none-eabi-elfedit    arm-none-eabi-gdb         arm-none-eabi-strings
arm-none-eabi-g++        arm-none-eabi-gprof       arm-none-eabi-strip
arm-none-eabi-gcc        arm-none-eabi-ld
```

43 https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
44 https://launchpad.net/gcc-arm-embedded
45 https://releases.linaro.org/components/toolchain/binaries/latest/arm-eabi/

CMSIS

The CMSIS library is built on many components. It creates a Hardware Abstraction Layer (HAL) for the ARM peripherals but not for the manufactured added ones. It standardizes the access to registers and resources of the microcontroller.
This enables the developing without reliying too much on libraries provided by the manufacturer, that hampers the portability. See CMSIS Site[46].
There are two versions of CMSIS:

- Version 4[47]: older version but still heavily used with a BSD or zlib license.
- Version 5[48]: new version with a broader license (apache) and support for new cores.

The EFM32 support library (see next) comes with a version 4 CMSIS library!

To download the version 4, one have to use the following command on the target folder

```
git clone https://github.com/ARM-software/CMSIS
```

For the version 5, use the command below.

```
git clone https://github.com/ARM-software/CMSIS_5
```

Files specific to the EFM32 microcontroller

The files needed for development for microcontroller are:

- *headers*: that define the registers to access the hardware.
- *link script*: that tells the linker (ld) how to built the objects files to build an executable
- *initialization files*: according CMSIS, two files, in this case, `start_efm32gg.c` e system_efm32gg.c, have the code to initialize the processing, including the initialization of data section.

There is a Software Development Kit SDK[49], provided the manufacturer, but no more supported. It can be still dowloaded from github using the following command.

```
git clone https://github.com/SiliconLabs/Gecko_SDK
```

There is an older version, which can be downloaded from Gecko_SDK.zip[50]. IT is a huge file, about 300 MBytes zipped and 2,4 GBytes after unzipping.

46 https://developer.arm.com/embedded/cmsis
47 https://github.com/ARM-software/CMSIS
48 https://github.com/ARM-software/CMSIS_5
49 https://github.com/SiliconLabs/Gecko_SDK
50 http://www.silabs.com/Support Documents/Software/Gecko_SDK.zip

Silicon Labs provides a Development Environment called Simplicity Studio[51], based on Eclipse[52], which downloads the files as they are needed. It is possible to use Simplicity just to download the files and copy them to another folder. this has the advantage of downloading only what is needed.

Header files

The ARM CMSIS files are in `Gecko_SDK/platform/CMSIS/` folder, but there is never a need to include them, because they are used in the component include files. In the `Gecko_SDK/platform/Device/SiliconLabs/EFM32GG/Include/` folder there are the device specific header files like the `efm32gg990f1024.h`, which defines symbols por registers according the CMSIS Standard for the EFM32GG990f1024 Microcontroller. There are also header files for peripheral like `efm32gg_gpio.h, efm32gg_adc.h` and others, which are already included by the device specific header file.

Although it is possible and quite common to include the device specific file directly, a better approach is to use a *generic* header, and define which device using a prepreprocessor symbol during compilation. To do this, copy the file `em_device.h` from the header files folder to the project folder and define the symbol `EFM32GG990F1024` with the `-DEFM32GG990F1024` compiler parameter.

There is other header file, called `em_chip.h` located in the `Gecko_SDK/platform/emlib/inc` folder that defines the `CHIP_Init` function. This function must be called at the very beginning and it corrects some bugs in core implementation. Since it calls other header files from the same folder, a better idea is to add this folder to the include path with the `-IGecko_SDK/platform/emlib/inc/` parameter.

Libraries

In folder `Gecko_SDK/platform/CMSIS/Lib/GCC/libarm_cortexM` there are the following libraries:

```
libarm_cortexM0l_math.a     libarm_cortexM7lfdp_math.a
libarm_cortexM3l_math.a     libarm_cortexM7lfsp_math.a
libarm_cortexM4lf_math.a    libarm_cortexM7l_math.a
libarm_cortexM4l_math.a
```

Tools to write to the flash memory of the microcontroller

The communication protocol used by the EFM32 boards is compatible with the J-Link. Two software application implement it:

- JLink

51 https://www.silabs.com/products/development-tools/software/simplicity-studio
52 www.eclipse.org

- OpenOCD

JLink

To use the JLink software download from Segger the JLink package (`JLink_Linux_V622b_x86_64.deb` or a more recent one), and if optionally the Ozone package (`ozone_2.54_x86_64.deb`), an GUI interface for JLink. To install them on Linux, run the following commands after downloading them:

```
sudo dpkg -i JLink_Linux_V622b_x86_64.deb
sudo dpkg -i ozone_2.54_x86_64.deb
```

After installing JLink, the following application are available in the /opt/SEGGER/JLink folder, with the corresponding links in /usr/bin folder.

```
JFlashSPI_CL          JLinkExe              JLinkGDBServer      JLinkLicenseManager
JLinkRegistration     JLinkRemoteServer     JLinkRTTClient      JLinkRTTLogger
JLinkSTM32            JLinkSWOViewer        JTAGLoadExe         Ozone
```

OpenOCD

NOT TESTED YET!!!
To use the `openocd` software, install a `openocd` using a package manager (apt) or compiling a new version from source code dowloaded from openocd[53].
Define a symbol `OOCDSCRIPTSDIR` as below.

```
OOCDSCRIPTSDIR=/usr/share/openocd/scripts
```

To run it, it is necessary to specify interface and board.

```
openocd -f $OOCDSCRIPTSDIR/interface/jlink.cfg -f $OOCDSCRIPTSDIR/board/efm32.cfg
```

Debugging Tools

In the GCC toolchain, the tool for debugging is the GNU Project Debugger (GDB), in the form specific for ARM (`arm-none-eabi-gdb`). It runa on the PC and since it can not communicate directly with the microcontroller, a relay mechanism is needed in the form of a GDB proxy. The GDB proxy can be the Segger `JLinkGDBServer` or the `openocd`.
This is a two step process:

1. In a separated window, start the GDB Proxy

53 http://www.openocd.org

```
$JLinkGDBServer
```

2. In other window, start GDB and establishes a connection

```
arm-none-eabi-gdb
(gdb) target remote localhost:2331
```

The ARM GNU GDB has a Command Line Interface (CLI) as default and a Text User Interface (TUI), which can be activated with a -ui parameter. There are many Graphic User Interface (GUI) applications for GDB including `ddd` and `nemiver`.

Integrated Development Environment (IDE)

The are many IDEs for Embedded Development, mostly based on Eclipse. The manufactures provides the Simplicity Studio[54].

Alternatives are:

- VisualStudio (Windows)
- emIDE (Windows)
- CodeBlocks
- Embedded Studio

54 https://www.silabs.com/products/development-tools/software/simplicity-studio

B

Using the tools

Connecting

Connect the board to the PC using the Mini USB cable. When connected a blue LED should lit. Starting `JLinkExe` prompt as shown below.

```
$JLinkExe
SEGGER J-Link Commander V6.22b (Compiled Dec  6 2017 17:02:58)
DLL version V6.22b, compiled Dec  6 2017 17:02:52

Connecting to J-Link via USB...O.K.
Firmware: Energy Micro EFM32 compiled Mar  1 2013 14:08:50
Hardware version: V7.00
S/N: 440112411
License(s): GDB
VTref = 3.329V
```

Type `connect` to establish a target connection, `?` for help

```
J-Link>si swd
J-Link>speed 4000
J-Link>device EFM32GG990F1024
J-Link>connect
```

See the transcript below.

```
$JLinkExe
J-Link>si swd
Selecting SWD as current target interface.
J-Link>speed 4000
Selecting 4000 kHz as target interface speed
J-Link>device EFM32GG990F1024
J-Link>con
```

```
Device "EFM32GG990F1024" selected.


Connecting to target via SWD
Found SW-DP with ID 0x2BA01477
Found SW-DP with ID 0x2BA01477
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x412FC231. Implementer code: 0x41 (ARM)
Found Cortex-M3 r2p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000
ROMTbl[0][0]: E000E000, CID: B105E00D, PID: 000BB000 SCS
ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 003BB002 DWT
ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 002BB003 FPB
ROMTbl[0][3]: E0000000, CID: B105E00D, PID: 003BB001 ITM
ROMTbl[0][4]: E0040000, CID: B105900D, PID: 003BB923 TPIU-Lite
ROMTbl[0][5]: E0041000, CID: B105900D, PID: 003BB924 ETM-M3
Cortex-M3 identified.
J-Link>
```

A command line makes all easier.

```
JLinkExe -if swd -device EFM32GG990F1024 -speed 2000
```

Considering that the debugging session will be done with GDB, the only important commands for `JLink` are related to flashing a program:

- `exit`: quits JLink
- `g`: start the CPU core
- `h`: halts the CPU core
- `r`: resets and halts the target
- `erase`: erase internal flash
- `loadfile`: load data file into target memory

Flashing

To use the `JLinkExe` to write the flash contents one has to use the following command:

```
$JLinkExe -Device EFM32GG990F1024 -If SWD -Speed 4000 -CommanderScript Flash.jlink
```

The `Flash.jlink` file has the following contents:

```
r
h
loadbin <path>,<base address>
r
```

Debugging

To use `gdb` as a debugging tool, the GDB Proxy must be started first using the command

```
JLinkGDBServer -if SWD -device EFM32GG995F1024 -speed 4000
```

In other window to start the gdb the following command is used:

```
$arm-none-eabi-gdb <execfile>
(gdb) monitor reset
(gdb) target remote localhost:2331
(gdb) break main
(gdb) monitor go
```

To debug using GDB see Debugging with GDB[55].
The most used commands are:

- `cont`: continues the execution
- `break`: sets a breakpoint in a function or in a line
- `list`: lists source file
- `next`: steps skipping over functions
- `step`: steps entering functions
- `display`: Displays the contents of a variable at each prompt.
- `print`: Prints the contents of a variable

The monitor commands enable direct access to JLink functionalities, as shown below.

- `monitor reset`: resets the CPU
- `monitor halt`: halts he CPU

The commands can be abbreviated by typing just enough characters. For example, `b main` is the same of `break main`.

55 https://sourceware.org/gdb/current/onlinedocs/gdb/

C

Sample project

Structure

A sample project for a EFM32GG microcontroller consists of:

- application source files (e,g, `main.c`)
- startup file (`startup_efm32gg.c`)
- CMSIS system initialization file (`system_efm32gg.c`)
- linker script (`efm32gg.ld`)
- header file (`em_device.h`)
- `Makefile`, a build automation tool
- Optionally, `README.md`, a description of project
- Optionally, `Doxyfile`, configuration file for doxygen, a documentation generator

Makefile

The make application can automate a lot of task in the software development. The recipes are specified in a `Makefile`. The `Makefile` provides a lot of options, which can be specified in the command line. In the `Makefile`, it is possible (and sometimes needed) to adjust compilation parameters and specify where the required folders and applications are. For example, `OBJDIR` is specified as gcc and is the folder where all object files and the executable are generated. `PROGNAME` is the project name

- `make all` generate image file in OBJDIR
- `make flash` write to flash memory in microcontroller
- `make clean` delete all generated files
- `make dis` disassemble output file into `OBJDIR`)/`PROGNAME.S`
- `make dump` generate a hexadecimal dump file into `OBJDIR/PROGNAME.dump`
- `make nm` list symbol table in standard output
- `make size` list size of output file
- `make term` open a terminal for serial communication to board
- `make debug` start an GDB proxy and the GDB debugger
- `make gdbproxy` start the GDB proxy

- `make docs` generates docs using Doxygen

Notes

The files for EFM32GG are in:

- **headers**:
  `Gecko_SDK/platform/Device/SiliconLabs/EFM32GG/Include`
- **linker script**:
  `Gecko_SDK/platform/Device/SiliconLabs/EFM32GG/Source/GCC/emf32gg.ld`
- **initialization**:
  `Gecko_SDK/platform/Device/SiliconLabs/EFM32GG/Source/GCC/startup_efm32gg.c`
  `Gecko_SDK/platform/Device/SiliconLabs/EFM32GG/Source/GCC/startup_efm32gg.S`
  `}]`

The software Simplicity Commander[56] enables the access to the JLink functionalities using a CLI. The Ozone[57] software provides a GUI for the JLink system.

56 https://www.silabs.com/documents/public/software/SimplicityCommander-Linux.zip
57 https://www.segger.com/downloads/jlink/#Ozone