Embedded and Ambient Systems

Topic of exercise 4

During the previous exercise, we looked at how to use a UART peripheral in a simple way: initialization, writing characters, reading characters (blocking). During this exercise, we will see how a UART peripheral can be operated a little more efficiently (non-blocking character reception, interrupt handling) and more conveniently (UART - stdio connection).

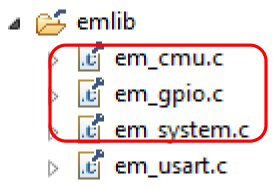
1. UART initialization

The initialization of the UART is done in the manner learned in the previous exercise. Extracted below is all the data required for initialization. Those who have the old code can still use it.

The following header files must be included:

```
#include "em_cmu.h"
#include "em_usart.h"
#include "em_gpio.h"
```

The following C files must be added to the project:



emlib
- em_cmu.c
- em_gpio.c
- em_system.c
- em_usart.c

Path to the files:

[install location]\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\platform\emlib\src\

You need to initialize the UART:

```
        // Enable clock for GPIO
        CMU->HFPERCLKEN0|= CMU_HFPERCLKEN0_GPIO;

        // Set PF7 to high
        GPIO_PinModeSet(gpioPortF, 7,gpioModePushPull, 1);

        // Configure UART0
        // (Now use the "emlib" functions whenever possible.)

        // Enable clock for UART0
        CMU_ClockEnable(cmuClock_UART0, true);


        // Initialize UART0 (115200 Baud, 8N1 frame format)
```

```
// To initialize the UART0, we need a structure to hold
// configuration data. It is a good practice to initialize it with
// default values, then set individual parameters only where needed.
USART_InitAsync_TypeDefUART0_init = USART_INITASYNC_DEFAULT;

USART_InitAsync(UART0, &UART0_init);
// USART0: see in efm32ggf1024.h

// Set TX (PE0) and RX (PE1) pins as push-pull output and input respectively.
// DOUT for TX is 1, as it is the idle state for UART communication
GPIO_PinModeSet(gpioPortE, 0,gpioModePushPull, 1);
// DOUT for RX is 0, as DOUT can enable a glitch filter for inputs,
// and we are fine without such a filter
GPIO_PinModeSet(gpioPortE, 1,gpioModeInput, 0);

// Use PE0 as TX and PE1 as RX (Location 1, see datasheet (not refman))
        // Enable both RX and TX for routing
UART0->ROUTE|= UART_ROUTE_LOCATION_LOC1;
        // Select "Location 1" as the routing configuration
UART0->ROUTE|= UART_ROUTE_TXPEN | UART_ROUTE_RXPEN;
```

The difference compared to the previous exercise is that the initialization parameters are set to USART_INITASYNC_DEFAULT , i.e., we initialized it with a predefined array, which contain exactly the settings we want, so there is no need to set each parameter separately.

2. Non-blocking character bet

The USART_Rx() function blocks the CPU (does not return until a character is received):

```
uint8_t USART_Rx(USART_TypeDef*usart)
{
 while(!(usart->STATUS & USART_STATUS_RXDATAV)) {
}

 return(uint8_t)usart->RXDATA;
}
```

It can be seen that we remain in a while loop until the RXDATAV bit in the STATUS register of the UART/USART peripheral passed as a parameter[1] is set. If this is done, the function returns with the contents of RXDATA (RX Buffer Data Register) containing the received character.

This did not cause any problems with the simple program we wrote in the previous exercise, because the main loop did nothing but wait for a character, and when data had been received, it had been sent

[1]"RX Data Valid: Set when data is available in the receive buffer. Cleared when the receive buffer is empty." EFM32GG Reference Manual

to the screen ("echo"). However, it can be seen that if the main loop had any other useful task, then it would not be expedient to wait for incoming characters in a blocking way.

Blocking can be avoided if we check the status register ourselves and call the USART_Rx() function only if there is an incoming character. We can access the status register directly, but we can also use the following function of the emlib USART API:

```
__STATIC_INLINE uint32_t USART_StatusGet(USART_TypeDef *usart)
{
 return usart->STATUS;
}
```

Then the part of the code responsible for the "echo" function in our main loop, it can look like this:

```
 if(USART_StatusGet(UART0) & USART_STATUS_RXDATAV) {
 USART_Tx(UART0,USART_Rx(UART0));
}
```

Note: we can say that it is not very efficient to call a function instead of directly reading a register, which in the same way just reads the given register and then returns with the read value. To eliminate this, the __STATIC_INLINE macro in the function declaration is used, whose definition is as follows (cmsis_gcc.h):

```
 #define __STATIC_INLINE static inline
```

The inline keyword asks the compiler to insert the body of the function at the point where it have been called, instead of calling the function. In this way we can write transparent code without losing performance. (About the inline keyword, it is worth noting that it only "requests" - and does not "oblige" - the compiler to handle the function call inline. Furthermore, even without this keyword, the compiler can decide to handle a function inline .)

We may have additional questions:

1. Why do we put the body of the function in the header file (when it is usually in the C file)?
2. Why do you even need the static keyword?
3. Why hide keywords behind a macro?

Since the answers to these questions are out of the scope of the exercise, they can be found in the Appendix for those who are more interested.

If we want to avoid that character reception block our program, another alternative is to create a non-blocking function to receive characters, e.g.:

```
intUSART_RxNonblocking(USART_TypeDef *usart)
{
 if(usart->STATUS& USART_STATUS_RXDATAV) {
 return(int)(usart->RXDATA);
}else{
 return-1;
}
}
```

If there is a received character (in the range 0...255 for the most common 8-bit UART frame format), we return with it, otherwise with a value outside the valid value set (-1). (For this reason, the return value is no longer uint8_t, but int.)

Note: multiple encoding standards[2] nor does recommend that we return from a function in more than one place. The explanation is that this way the code is less transparent, and with an early return, we might miss parts of the function that we didn't want. The suggested solution is to return only at one point, at the end of the function. Our function is small enough to be clear without it, but it is not useless to practice methods aimed at safer programming:

```
intUSART_RxNonblocking(USART_TypeDef *usart)
{
 Int retVal = -1;

 if(usart->STATUS& USART_STATUS_RXDATAV) {
	retVal = (int)(usart->RXDATA);
}

 return retVal;
}
```

And the relevant code fragment of the main loop can be something like:

```
intch;
ch = USART_RxNonblocking(UART0);
 if(ch != -1) {
 USART_Tx(UART0, ch);
}
```

3. Interrupt handling

The methods detailed in the previous point offer a solution for how to receive characters without blocking the main loop and so the program. In the simple example of the previous exercise, no problems can arise with the blocking solution since no other tasks to do existed. On the other hand we can suppose that the main loop does other useful things, and if applicable, it does it for a longer time than the checking rate of the existance of new data that may had been received. Therefore, if there is a new received character, then data loss can occur.

Interrupt can provide a solution to the latter problem. Interrupts can interrupt the execution of arbitrary code, and the part of code dedicated to handling them can thus perform time-critical tasks (reading a character in our example).

If we want to service a peripheral using interrupts, the following steps are generally required:

1. The peripheral in question must be configured to request an interrupt when a specific event occurs
2. Interrupt request servicing must be enabled on the processor side

[2]For example, it is made for safety-critical systems MISRA C among his latest (2012) recommendations, rule 15.5 states that it is recommended to return from a function only at one point (at the end).

3. The interrupt handling routine (ISR[3]) pointing vector
4. The interrupt handling routine must be written
    a. The processor context must be saved
    b. The exact reason for the interrupt request must be found out
    c. The interrupt requested by the given peripheral must be serviced
    d. The interrupt requested by the given peripheral must be acknowledged
    e. The processor context must be reloaded
    f. You must return from the interrupt handling routine

The above steps are general, not all of them are always necessary. Now let's look at these one by one for the (UART0) peripheral of the microcontroller we use (ARM Cortex-M3 core)!

3.1. Configuring the interrupt request

The UART/USART peripherals of the EFM32 Giant Gecko microcontrollers can request an interrupt due to several possible events. These can be divided into two large groups: interrupts related to transmission and reception.

Transmission related interrupts[4]:

• TXT:	TX Complete
• TXBL:	TX Buffer Level
• TXOF:	TX Overflow
• CCF:	Collision Check Fail

Reception related interrupts[5]:

• RXDATAV:	RX Data Valid
• RXFULL:	RX Buffer Full
• RXOF:	RX Overflow
• RXUF:	RX Underflow
• PERR:	Parity Error
• FERR:	Framing Error
• MPAF:	Multi-Processor Address Frame
• SSM:	Slave-Select In Master Mode

Considering that we want to solve the reception of characters with interrupts, the interrupts related to the reception are the interesting ones for us. MPAF and SSM belong to modes that we do not use. RXOF, RXUF, PERR and FERR occur in case of various errors, but we are mainly interested in successfully received characters. There are two possible interrupts left: RXDATAV and RXFULL. There is a difference between the two because the receive buffer has two elements (FIFO type) for the UART/USART

[3]Interrupt Service Routine
[4]EFM32_GG-RM.pdf: 17.5.16 - USARTn_IF - Interrupt Flag Register (page 488 - 2016-04-28 - Giant Gecko Family - d0053_Rev1.20)
[5]EFM32_GG-RM.pdf: 17.5.16 - USARTn_IF - Interrupt Flag Register (page 488 - 2016-04-28 - Giant Gecko Family - d0053_Rev1.20)

peripherals in the EMF32GG microcontrollers. RXDATAV is true if there is at least one successfully received character in the buffer. And RXFULL is true when this buffer is full (i.e. there are already two characters in it).

> *Comment*: if a third character is received, it still does not cause data loss. The received characters arrive bit by bit, so they first enter a shift register. From there, they are forwarded to the receive FIFO (if there is space). If there is none, then the last one remains in the shift register. Thus, the peripheral can store three received characters. If, on the other hand, a fourth character arrives, it already causes data loss: the third character currently in the shift register will be overwritten.

Basically, we want an interrupt immediately when a valid character is received, so we will set the RXDATAV interrupt. We can use emlib's USART_IntEnable() function for this! Its first parameter is the peripheral in question: UART0 (not USART0). And the second is the interrupt you want to enable, which is defined with USART_IF_<interrupt_short_name> (efm32gg_usart.h - access: em_device.h→efm32gg990f1024.h→efm32gg_usart.h) can be selected (even more at once if we OR them).

3.2. Enable interrupt request

In the previous point, we only solved so far that the UART0 peripheral requests an interrupt if there is new (not yet read) data in its receive buffers. However, this does not mean that the processor will actually service this interrupt! For this, in the processor interrupt controller (NVIC[6]) the service of the given interrupt must also be enabled!

We have seen that a specific peripheral can request an interrupt when several events occur. We also know that there are usually several types of peripherals in a microcontroller, and that many types of peripherals are possible. All in all, there would be too many interrupt requests. Cortex-M type processors save a little on this and often combine certain interrupts. For example, the processor sees only two interrupt requests from the UART/USART peripherals: one for reception and one for transmission. The interrupt handling routine must find out, if there could be more than one source of the interrupt request, then what exactly it was. Since we only configured a receive-related interrupt, we won't have to deal with that.

According to the data sheet of the microcontroller, the following two interrupt lines reach the processor: UART0_RX and UART0_TX. In the interrupt controller, individual interrupt sources can be enabled with the NVIC_EnableIRQ() function (core_cm3.h, which is already referenced indirectly from device.h). Let's see what type of parameter the function expects and select the appropriate value (attention, don't confuse the UART0 peripheral with the USART0 here either).

The names of the individual NVIC IT lines can be found in the file efm32gg990f1024.h.

It usually does not cause problems, but it is a good practice to clear the associated interrupt flag before enabling interrupts, so that any stuck interrupts do not take effect.

[6]Nested Vectored Interrupt Controller

3.3. Filling the interrupt vector table

The interrupt vector table is located at the beginning of the program memory. Each interrupt has an entry in this table. The vector here tells you where the routine serving the given interrupt is.

On many architectures, a vector should be understood as jumping instructions, because the processor expects instructions in this table. In the case of the Cortex architecture, memory addresses must be placed here, because the processor does not treat them as instructions to be executed, but jumps fixedly, the only question is where.

The interrupt vector table is pre-defined in the startup code (CMSIS/EFM32GG/startup_gcc_efm32gg.s) (see under __Vectors: tag). Assigns an address to each interrupt (in the form <interrupt name>_IRQHandler). Later (at the end of the file), each of these addresses is assigned a value with the help of a macro (def_irq_handler), namely as the address of a routine to be used by default: Default_Handler.

The default server routine contains nothing but a jump statement (b as branch). The address specified for the jump is ".", which can be used as the address of the current instruction in the ARM assembly. That is, we get an endless loop. This is good so that if, for any reason, an interrupt that our code is not prepared for is triggered, then the execution is not completely uncontrolled.

OK, but what do we need to do if we want an ISR we've written to be called? If we take a closer look at the def_irq_handler macro, we can see that it appends a ".weak" keyword to each assignment. This means that these assignments can be overridden. So if we write a function called <interrupt name>_IRQHandler, the linker will insert its address in the appropriate place of the vector table, and not the default routine.

So we see that we don't have to deal with filling out the interrupt vector table, we get it ready with the startup code. We just have to make sure that we name our own ISR correctly.

3.4. Writing the interrupt handling routine

In the case of Cortex microcontrollers, the interrupt handling routine can be implemented with a traditional C language function.

> *Comment*:
>
> This doesn't seem like a big deal at first, but it's not the case on simpler architectures. The instructions in the ISR use the processor's registers in the same way as the code running in the background that was just interrupted. This would spoil the operation if the ISR did not ensure that the registers it uses are saved somewhere at the beginning of its run, and then restored at the end (for this purpose, the data memory, to be exact, is usually used).
>
> Furthermore, there are processors that have a special return instruction for ISRs (because, for example, when entering an ISR, it prohibits any further interrupts, which can be enabled again with this special return instruction).
>
> However, a normal C function does not save registers or use a special return statement. C language extensions must be used on architectures where this is required. Moreover, these extensions are compiler dependent.

Cortex-based microcontrollers use a different set of registers when entering interrupt mode. In this way, there is no need to save or restore the registers. Also, there is nothing that requires a special return statement.

3.4.1. We define the function of the interrupt handling routine

So let's see what routine belongs to the UART0_RX interrupt in the startup code (the startup code can be found in the file CMSIS/EFM32GG/startup_gcc_efm32gg.s)! Let's create a function with this name in our own code (both its return value and its parameter are void, and don't confuse the USART0 peripheral with the UART0 here)!

3.4.2. Let's read and send back the received character

Since we have configured only one receive-type interrupt request (RX Data Valid) for the UART0 peripheral, we can be sure that if we have reached this point, there is a new (and error-free) character in the receive buffer of UART0. Let's read it!

The familiar USART_Rx() can be used for reading, but USART_RxDataGet() is more convenient. The former waits until the peripheral indicates by flipping a corresponding bit that a new character has arrived before reading it. This is unnecessary if we are already in our ISR, because it is called when a character has arrived.

The read character is sent back, thereby implementing the "echo" function from the ISR.

3.4.3. Acknowledge the interrupt

For many interrupts, the mere fact that we entered the ISR does not acknowledge that particular interrupt. That is, after exiting the ISR, the same interrupt will occur again. To delete the interrupt request, e.g. it can be done with the USART_IntClear() function.

On the other hand, the interrupt we use is such that if we read the received character, it also corresponds to an acknowledgement, so we can now explicitly dispense with acknowledging the interrupt.

3.5. Modification of the interrupt handling routine

Since an interrupt handling routine can interrupt any code running in the background, it is advisable to write it as short as possible. It is not "appropriate" to wait in it, and it is advisable to outsource everything that is not time-critical, otherwise the code running in the background may be blocked for a while.

In our example, the ISR is quite short, and we can also say that returning the character is no longer a time-critical task. Also, if we look at the USART_Tx() function, we can see that it is also blocking. If there is currently a transmission, it does not write the new data into the transmitter buffer of the UART/UASRT peripheral until it has finished:

```
void USART_Tx(USART_TypeDef *usart, uint8_t data)
{
 /* Check that transmit buffer is empty */
 while(!(usart->STATUS& USART_STATUS_TXBL)) {
}
usart->TXDATA= (uint32_t)data;
}
```

This is likely to cause a less frequent and shorter delay than the similar wait for USART_Rx(), but it can still delay the quick service of the interrupt.

3.5.1. Exception to return character from ISR

In the ISR, instead of returning immediately, read the received character into a global variable. The code running in the background must somehow be notified of the arrival of the new character. We can also do this through a global variable

3.5.2. Effect of compiler optimizations

Let's try what happens if we compile in an optimized way (this requires a "Release" type build instead of the previous "Debug").

There is a good chance that our program will not work. The reason for this is that, as a result of the optimizations, the compiler works as follows. "The variable through which the ISR signals to the main loop has a false value by default and is not rewritten to true anywhere after that. Although there is such a line in the function of the ISR, no one calls the ISR as a function, so it will not run. So the variable used for the signal has the value false throughout. In this way, in the main loop, the complete code bound to the condition can be extracted, because it will never run anyway, and what is the need for it to take up space in the program's memory."

The problem can be solved by entering the keyword "volatile" in the definition of the variable used for the signal. With it, we tell the compiler that the contents of the memory behind the marked variable can change at any time, so don't try to optimize it, but actually read the given variable every time you access it.

Comment: most likely the problem affects only the variable used as a signal. There is no such problem with the variable used to pass the character. Although, according to the logic above, it would be possible to save a little by not reading the variable in every loop, but always assuming its initial value. Maybe because it would no longer be a big advantage, maybe for other reasons, but the compiler does not optimize here. Anyway, if that's the case, let's mark this variable with the keyword "volatile"!

3.5.3. Increasing the character buffer - additional task

After we removed the return of characters from the ISR, there was no unnecessary code left. However, we can say that this comes at a price. If the main loop does anything other than return new characters, several interrupts may be executed before the main loop checks again to see if there are any new characters received. So, even though the ISR read the current new character immediately, it was lost from the point of view of the function.

A solution to this could be if our global variable, as a character buffer, could store several characters. The size of the buffer must be large enough so that there is no loss of data despite the assumed maximum number of characters arriving in a short time. (If the characters normally come more often than we can process them, then of course there is no such buffer that can protect this.)

You don't have to solve this task, you just have to think about it.

4. Energy efficient operation

It's not very efficient to keep waiting for a variable to pop into a main loop. Processors can usually be put to sleep to a certain depth, from which they can usually be woken up by interrupts.

There are several such modes for Giant Gecko microcontrollers as well. In the simplest case, only the CPU itself is put to sleep (it does not receive a clock signal), but the other peripherals are operational. This is called EM1.

First, let's see how much the panel is currently consuming energy!

After that, modify the code to enter the processor in EM1 mode in the main loop! The EMU (Energy Management Unit) peripheral of the microcontroller is responsible for switching between energy-saving modes. Thus, we will need "em_emu.c" from among the emlib source files. This is probably part of the project by default. If not, let's put it in!

The function to use is EMU_EnterEM1(). Of course, in order to be able to call this correctly, we must refer to the "em_emu.h" header file! Let's see if consumption has been reduced.

Comment: given that we only use one interrupt now, we can be sure that a character has arrived after waking up. In this way, even the variable used for the signal and its handling can be omitted.

5. Using Stdio (UART0)

So far, the UART0 peripheral has been used directly using the USART_Rx() and USART_Tx() functions. However, we would get a more portable code if we could use the standard getchar() and putchar() functions instead. Not to mention e.g. about the convenience provided by printf().

Fortunately, the standard I/O of the C language can be redirected. The high-level routines we use use lower-level functions that are capable of performing basic functions such as sending and receiving a character, etc. If we implement these functions in such a way that, e.g. use a UART peripheral, then in the end e.g. a printf() will also write to the serial port.

5.1. Add the required files

For standard I/O redirection to work, we need two files: retargetio.c and retarget<peripheral>.c (in our case retargetserial.c). retargetio.c defines a very thin software layer, in which the parts that are common are included, no matter which specific peripheral we want to redirect the standard in- or output. The other contains code fragments implemented for the selected peripheral.

So add the following files to the project (e.g. under a "drivers" subfolder).[7]:

- retargetio.c
- retargetserial.c

Path to the files:

[install directory]\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\hardware\kit\common\drivers\

[7]In the virtual machine you are using now, here are:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\drivers

Notes:

- If we look into the files, it can be seen that characters are received using an interrupt. Therefore, when using retargetserial.c, we can no longer use the Rx Data Valid interrupt of the selected U(S)ART peripheral.
- It is also worth noting that getchar() will return even if there is no received character, but the return value is -1 (therefore this function call does not block).

The files above require other files. Depending on the project we added them to, we may need to add more or less other files.

First, the trivial dependency of "retargetserial.c" is the following, since we redirect to the serial port. If this is not included in the project, add it (e.g. under the "emlib" folder):

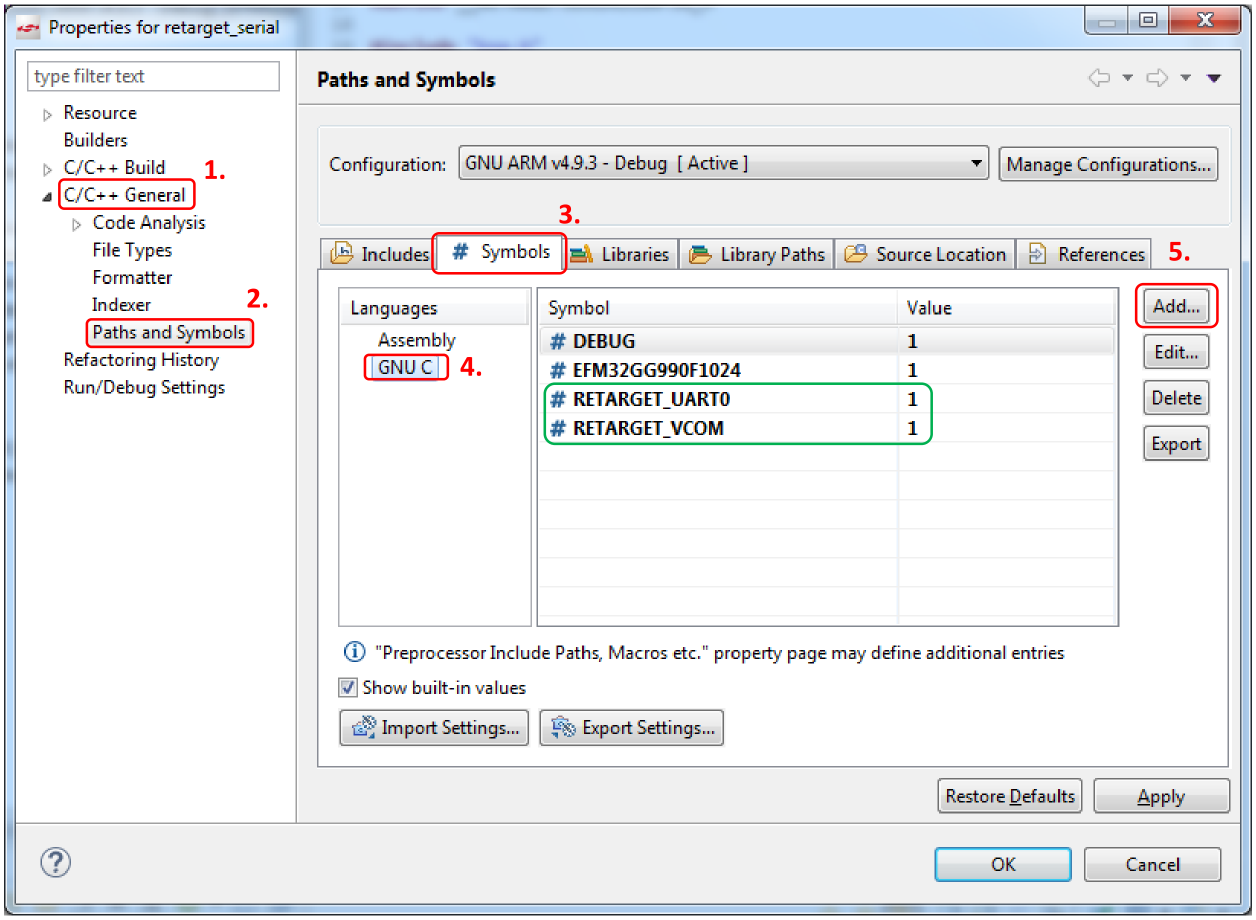
- em_usart.c

Since a clock signal is required to drive the serial port peripherals, their pins must be properly routed, and emlib uses other basic functions during their setup, the following three files and the corresponding header files are also needed (if the project does not already contain them):

- **em_core.c**(this was not part of the project until now)
- em_cmu.c
- em_gpio.c

5.2. Select the serial port to use

We can do this with preprocessor directives, which we have to define in advance for the project (then they are passed as command line parameters during compilation). Two values are needed: RETARGET_UART0 to use UART0, and RETARGET_VCOM to enable output of UART0 via the Board Controller to the USB port as a virtual serial port. Defines should have a value of 1-1. We can do this in the following location (Project / Properties):


Properties for retarget_serial

type filter text

- Resource
- Builders
- C/C++ Build 1.
- C/C++ General
  - Code Analysis
  - File Types
  - Formatter
  - Indexer 2.
  - Paths and Symbols
- Refactoring History
- Run/Debug Settings

Paths and Symbols

Configuration: GNU ARM v4.9.3 - Debug [Active] Manage Configurations...

3.

Includes | # Symbols | Libraries | Library Paths | Source Location | References 5.

Languages	Symbol	Value
Assembly	# DEBUG	1
GNU C 4.	# EFM32GG990F1024	1
	# RETARGET_UART0	1
	# RETARGET_VCOM	1

Add... Edit... Delete Export

"Preprocessor Include Paths, Macros etc." property page may define additional entries

☑ Show built-in values

Import Settings... Export Settings...

Restore Defaults Apply

OK Cancel

5.3. Usage

Let's first refer to the following two headers:

- stdio.h
- retargetserial.h

After that, we invite the following:

- RETARGET_SerialInit()

Optionally, we can call the following:

- RETARGET_SerialCrLf(true)

If we set this to a true value as a parameter, any end-of-line character sent (\r (carriage return) or \n (new line, line feed)) will automatically insert the other one. This makes it a bit more convenient to use printf().

After that, the following should work:

- printf("Hello!\n");

6. Using the LCD - Optional

6.1. Using factory functions

Using the LCD on the panel is very easy thanks to the functions that are part of the SDK. The required file is "segmentlcd.c".[8]This is added to the project (e.g. in a "drivers" subfolder). The declarations of the functions in it are in the "segmentlcd.h" header, so we refer to it.

Since "segmentlcd.c" uses the LCD controller peripheral in the microcontroller to control the LCD, "em_lcd.c" is also needed[9]also add a file to the project.

The LCD must be initialized before use, for this you can use "SegmentLCD_Init()". A single parameter enables stronger propulsion. Basically, we're good without it, so it should be "fasle" (if someone's LCD image is very weak, you can try setting it to "true" and see if it gets better).

We can try writing to the LCD, e.g. by completing our main loop by writing the ASCII code of the received character (SegmentLCD_Number()). The use of the other functions is self-explanatory.

6.2. Segment-by-segment drive

For homework, it is necessary to be able to control the display on the lower half of the LCD, capable of writing out seven alphanumeric characters, segment by segment. The factory SDK does not support this. We had to supplement and modify it. There is a demo project for this (displaySegmentField). You can import this and see how to use it. Since it is relatively clear, it is also basically "self-explanatory"

Appendix: __STAIC_INLINE

This appendix answers the following questions:

1. **Why do we put the body of the function in the header file (when it is usually in the C file)?**

   Normally, during translation, the assembly instruction (e.g. call) responsible for the subroutine call of the given architecture is inserted instead of a function call. This usually has an operand, a memory address at the beginning of the code of the intended function. Since the function may be defined in another C file, the compiler does not know a specific address at this point, it only places a label. Then, if we also compile the C file hosting the intended function, we will already have its code. After all source files have been translated, the linking phase follows. The linker is responsible for putting together the code fragments (object files) translated by the compiler and resolving the tags in them to actual memory addresses.

   If, on the other hand, we want to use the function inline, then its code is needed already during the compilation (since a call instruction must not be placed, but the body of the function must be inserted). This means that the compiler must already know the body of the function in question. Basically, it is not possible to compile something like an inline call function, because the body of the function will only be in another source file, so you would have to wait for the linker to solve it, but the linker is not a compiler. Supposedly there are compilers where the linker would call the compiler again to help out in such a case, but let's

[8]On the virtual machine you are currently using, it is located here:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\drivers
[9]And the files for emlib are here:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\platform\emlib\src

ignore that for now. So we need the body of the function. The easiest way to do this is to put it in the H file, not the C file belonging to a specific module. That's why the bodies of the inline functions are also in the header files.

2. **Why do you even need the static keyword?**

   If, on the other hand, the function is included in a header file together with its body, then - if several source files refer to the given header - then the function definition will be included in several source files (and this is not allowed in the default case, because how would the linker later know that a a function with a given name, then actually which one it is). This is where the static keyword comes into play. It limits the visibility of the function from the default global to local (i.e. only to the given source file). That's why it's static.

3. **Why hide keywords behind a macro?**

   The CMSIS standard wants to be translator-independent. Therefore, if somewhere there is a suspicion that a specific function might be dependent on a compiler, it hides it behind a macro. Thus, if we were to use a compiler where, for some reason, these keywords are not needed to make a function static and inline, it will still work (with an alternative macro).