

# Embedded and ambient systems

2020.11.03.

## Practice 4

### Development of UART communications: a more sophisticated approach



Méréstechnika és  
Információs Rendszerek  
Tanszék

# Problems with our UART implementation

- Remember the final solution:

```
/* Infinite loop */  
while (1) {  
    USART_Tx(UART0, USART_Rx(UART0));  
}
```

- This solution is a blocking implementation since USART\_Rx will not return until data is received
  - Better solution to call USART\_Rx function only if a character can be found in the buffer
  - An other good way to use interrupt
- Better to start a new project in the same way done before
  - See the following slides to remember stating a new project

# Strating with a new project

- File->New->Project->Silicon Labs MCU Project:

New Silicon Labs Project

Project setup

Select the board, part, and SDK for the project.

Boards:

Search

EFM32 Giant Gecko Starter Kit board (BRD2200A Rev A03) x

Part:

Search

EFM32GG990F1024

SDK:

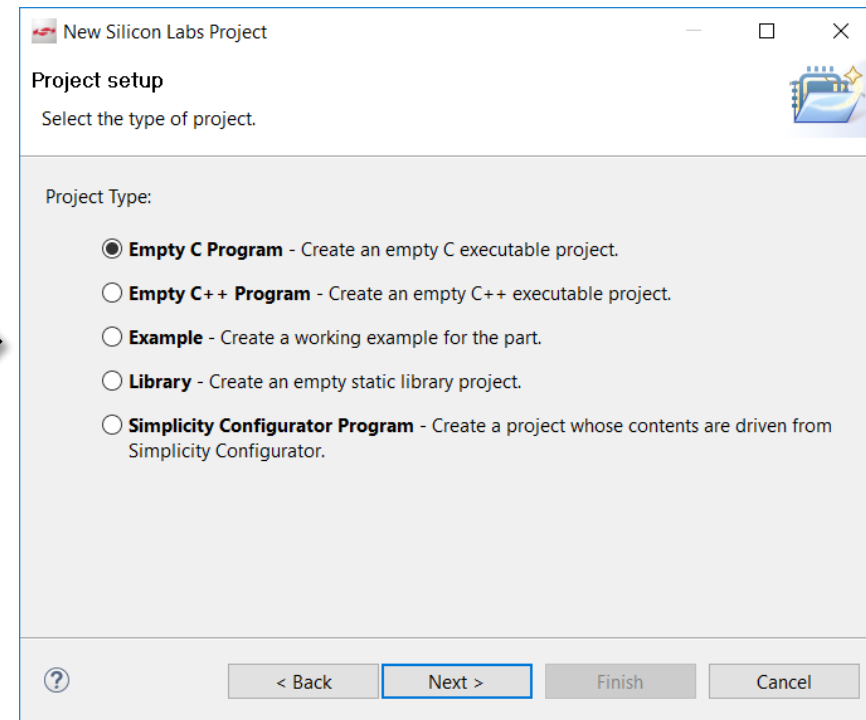
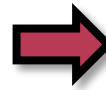
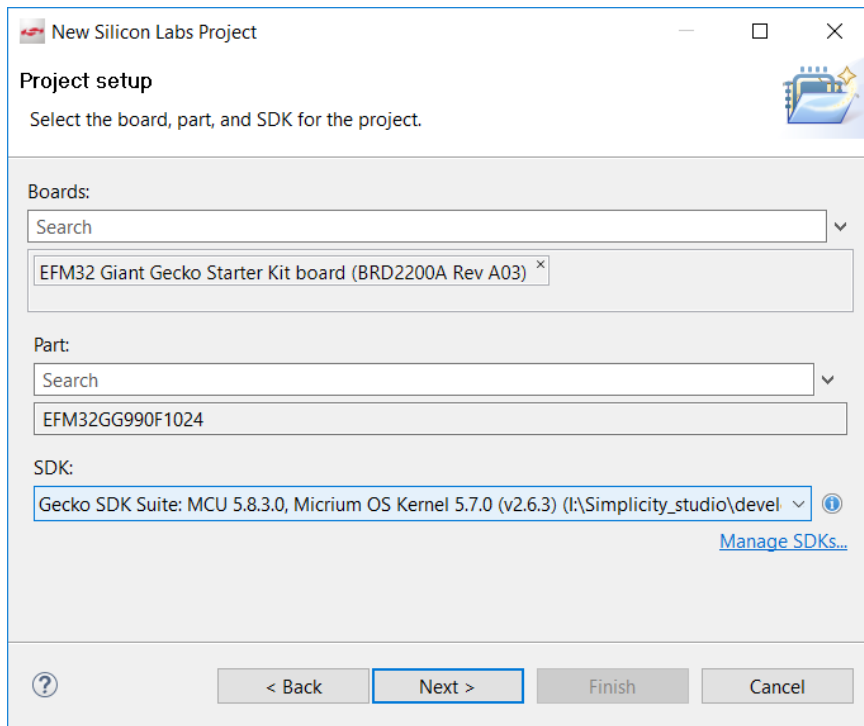
Gecko SDK Suite: MCU 5.8.3.0, Micrium OS Kernel 5.7.0 (v2.6.3) (I:\Simplicity\_studio\devel ⓘ

[Manage SDKs...](#)

? < Back Next > Finish Cancel

# Strating with a new project

- File->New->Project->Silicon Labs MCU Project:



# Strating with a new project

- Give project name and location, and set Copy content:

New Silicon Labs Project

Project Configuration

Select the project name and location.

Project name:

Use default location

Location:

With project files:

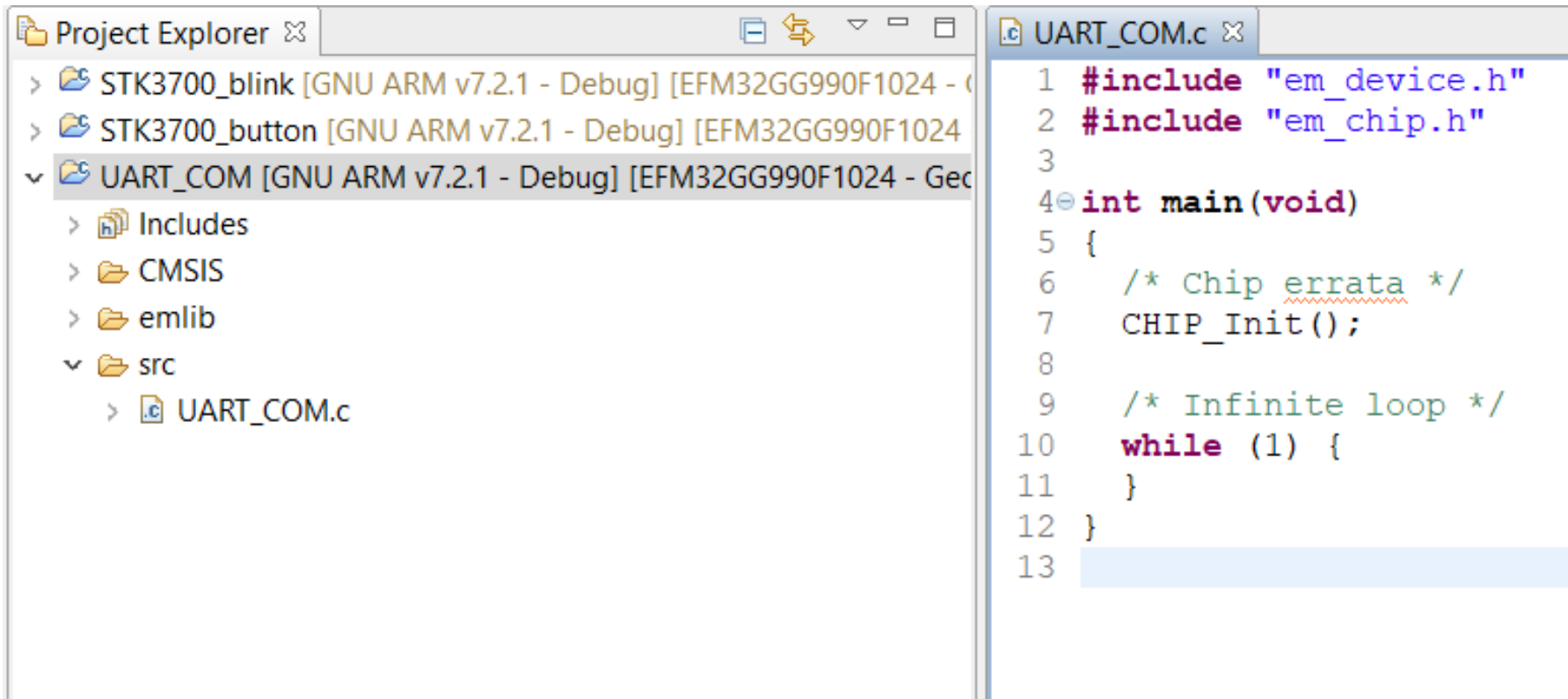
Link to sources

Link sdk and copy project sources

Copy contents

# Project created – start programming

- Main.c can be also renamed to UART\_COM.c
- Although an empty C project has been created a program skeleton is offered automatically



```
Project Explorer
> STK3700_blink [GNU ARM v7.2.1 - Debug] [EFM32GG990F1024 - Gec
> STK3700_button [GNU ARM v7.2.1 - Debug] [EFM32GG990F1024
v UART_COM [GNU ARM v7.2.1 - Debug] [EFM32GG990F1024 - Gec
  > Includes
  > CMSIS
  > emlib
  v src
    > UART_COM.c

UART_COM.c
1 #include "em_device.h"
2 #include "em_chip.h"
3
4 int main(void)
5 {
6     /* Chip errata */
7     CHIP_Init();
8
9     /* Infinite loop */
10    while (1) {
11    }
12 }
13
```

# Files to be added to the project

- Search the library where Simplicity Studio is installed
  - Contains include (inc: \*.c) and source (src: \*.h) files:  
i:\Simplicity\_studio\developer\sdk\gecko\_sdk\_suite\v2.6\platform\emlib\
- Following files have to be drag-and-dropped into emlib library of the project (see next slide):
  - em\_cmu.c (clock management unit)
  - em\_gpio.c
  - em\_usart.c
  - em\_core.c
  - em\_emu.c (energy management unit)

# Files to be added to the project

- Furthermore they have to be included into the program:

The screenshot shows the Project Explorer on the left and the source code editor on the right. The Project Explorer displays the 'emlib' folder under 'Includes', which contains the following files:

- em\_cmu.c
- em\_core.c
- em\_emu.c
- em\_gpio.c
- em\_system.c
- em\_usart.c

The source code editor shows the following code in `*UART_COM.c`:

```
1 #include "em_device.h"
2 #include "em_chip.h"
3 #include "em_cmu.h"
4 #include "em_gpio.h"
5 #include "em_usart.h"
6 #include "em_core.h"
7 #include "em_emu.h"
8
9 int main(void)
10 {
11     /* Chip errata */
12     CHIP_Init();
13
14
15     /* Infinite loop */
16     while (1) {
17     }
18 }
```



# Code to start with

Use the following code as a reference for your work (continue from previous result):

```
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_usart.h"
#include "em_core.h"
#include "em_emu.h"

int main(void)
{
    /* Chip errata */
    CHIP_Init();

    // 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_TypeDef UART0_init = USART_INITASYNC_DEFAULT;

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

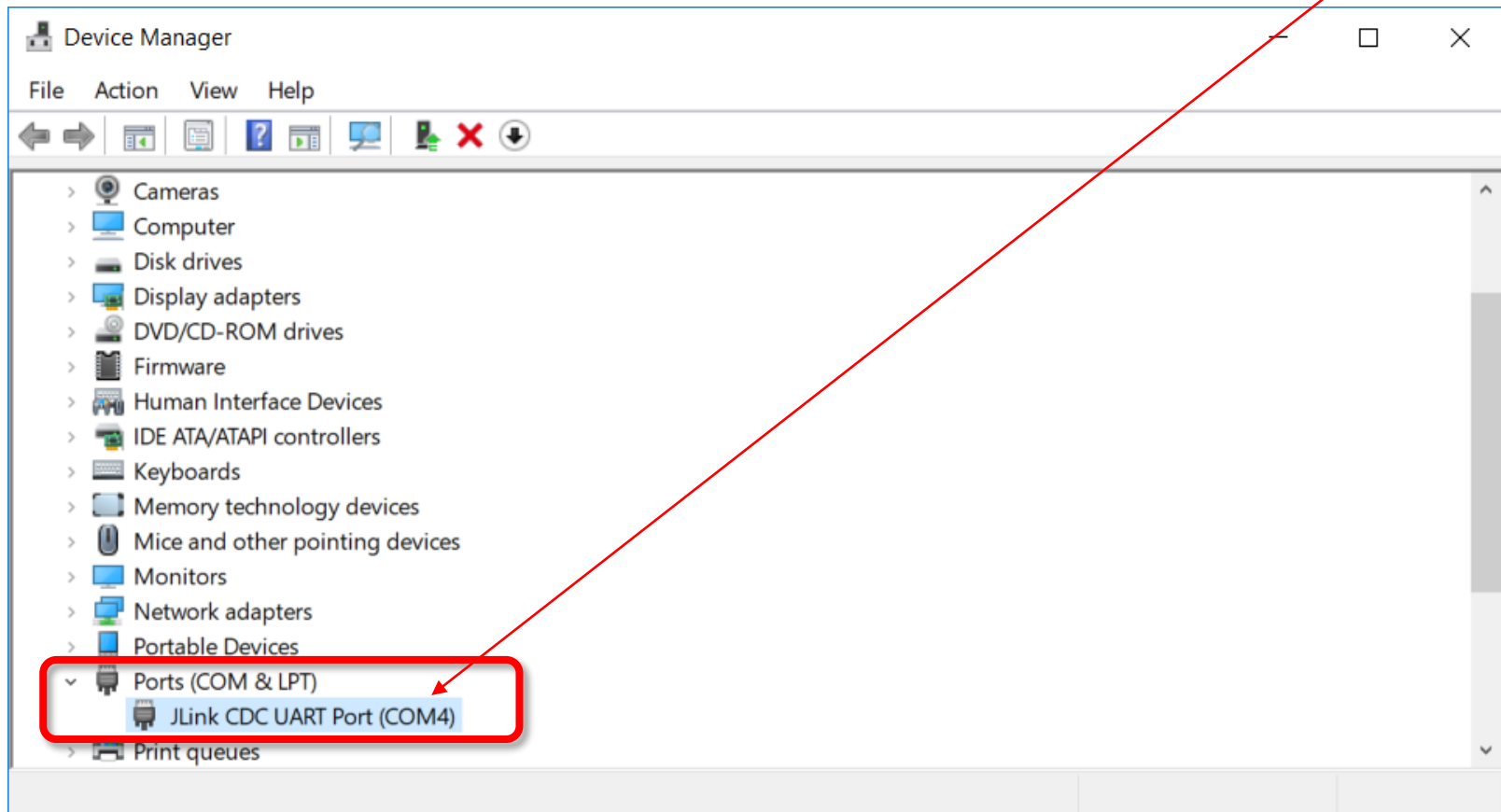
    // Set TX (PE0) and RX (PE1) pins as push-pull output and input resp.
    // DOUT for TX is 1, as it is the idle state for UART communication
    GPIO_PinModeSet(gpioPortE, 0, gpioModePushPull, 1);
    // DOUT for RX is 0, as DOUT can enable a glitch filter for inputs,
    // and we are fine without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

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

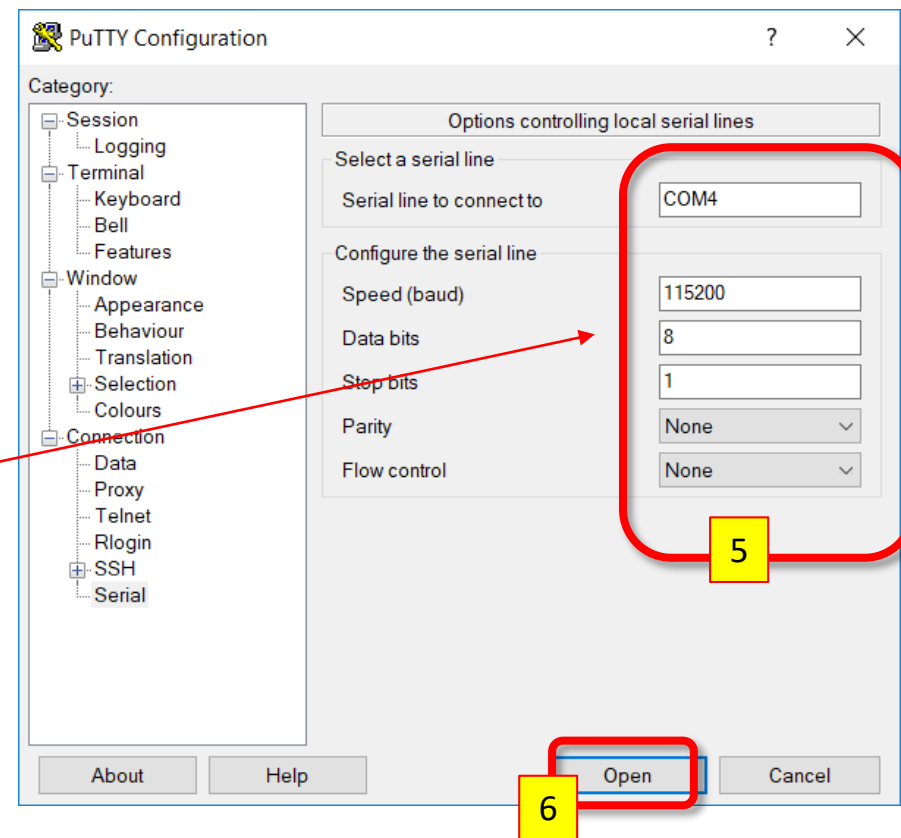
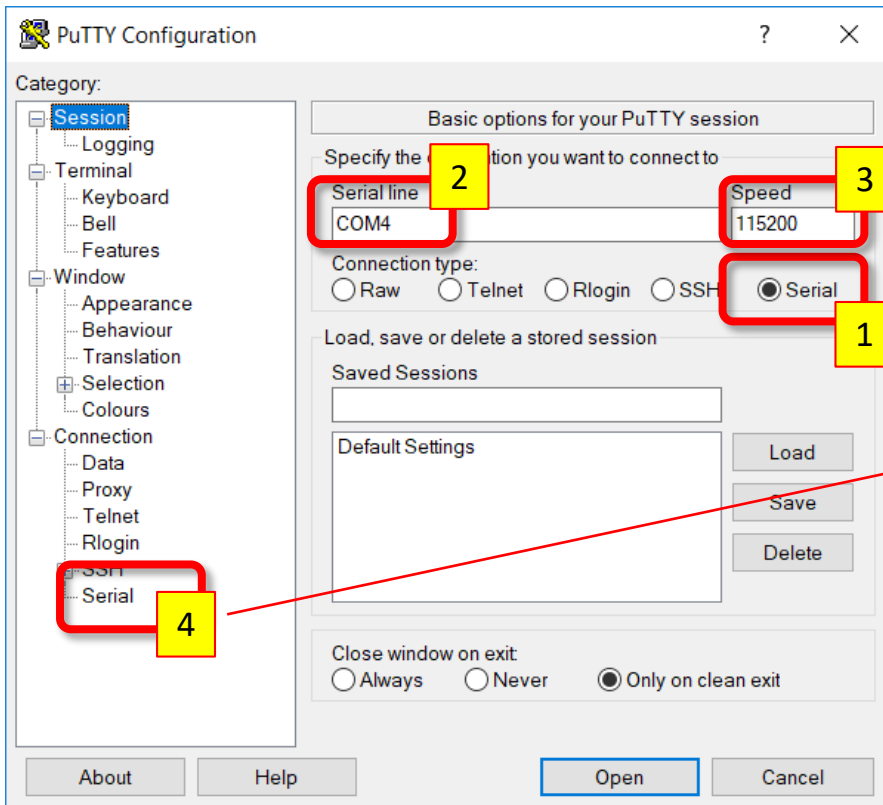
# Setting the terminal program

- Check UART (COM port number and its settings) in Device Manager in Windows (now it is COM4)



# Setting the terminal program

- A PC-based terminal program is needed to get access to COM4 port: an option is putty.exe



# Non-blocking character reception

- Check our previous solution again
  - What does USART\_Rx do(stay on it by mouse pointer)?

```
/* Infinite loop */
while (1) {
    USART_Tx(UART0, USART_Rx(UART0));
}
}

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

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

Press 'F2' for focus

- Operation: remains in while loop until in USART\_STATUS\_RXDATAV bit flips to 1, then returns with the received character (RXDATA)
  - See [03\\_EFM32\\_Reference\\_manual\\_EFM32GG-reference\\_manual.pdf](#) on page 481 (and next slide)
- Blocking can be avoided if we check the STATUS reg

# Non-blocking character reception

## 17.5.5 USARTn\_STATUS - USART Status Register

Offset	Bit Position																																																											
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																												
0x010																																																												
Reset																													R	0	R	0	R	0	R	0	R	0	R	0	R	0	R	0	R	0	R	0	R	1	R	0	R	0	R	0	R	0	R	0
Access																													R		R		R		R		R		R		R		R		R		R		R		R		R		R		R		R	
Name																													RXFULLRIGHT		RXDATAVRIGHT		TXBSRIGHT		TXBDRIGHT		RXFULL		<b>RXDATAV</b>		TXBL		TXC		TXTRI		RXBLOCK		MASTER		TXENS		RXENS							

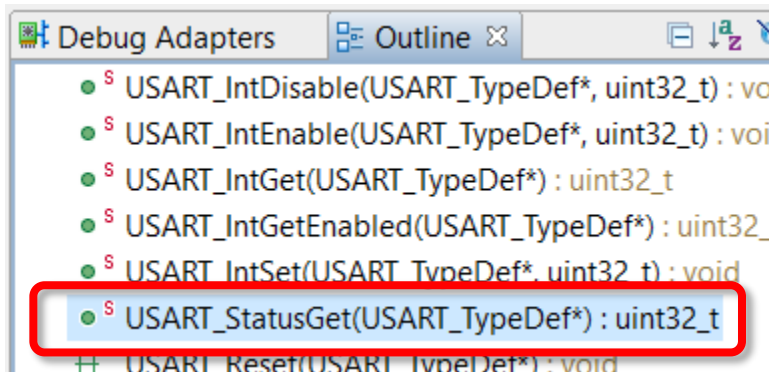
7	RXDATAV	0	R	<b>RX Data Valid</b>
---	---------	---	---	----------------------

Set when data is available in the receive buffer. Cleared when the receive buffer is empty.

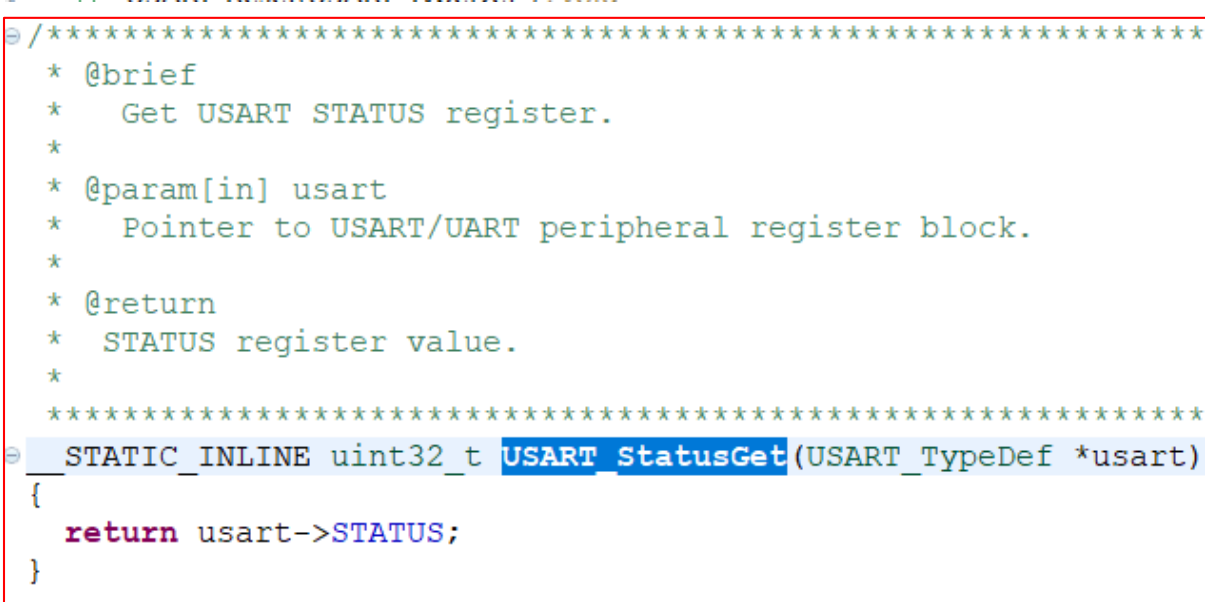
- Non-blocking solution: check STATUS reg. and call USART\_Rx() function only if incoming character is available

# Non-blocking character reception

- Search `em_usart.h` for a function that checks STATUS register (if available, hopefully it is):



```
Debug Adapters Outline
• USART_IntDisable(USART_TypeDef*, uint32_t) : void
• USART_IntEnable(USART_TypeDef*, uint32_t) : void
• USART_IntGet(USART_TypeDef*) : uint32_t
• USART_IntGetEnabled(USART_TypeDef*) : uint32_t
• USART_IntSet(USART_TypeDef*, uint32_t) : void
• USART_StatusGet(USART_TypeDef*) : uint32_t
+ USART_Reset(USART_TypeDef*) : void
```

```
/* *****
 * @brief
 *   Get USART STATUS register.
 *
 * @param[in] usart
 *   Pointer to USART/UART peripheral register block.
 *
 * @return
 *   STATUS register value.
 *
 * *****
 */
STATIC_INLINE uint32_t USART_StatusGet(USART_TypeDef *usart)
{
    return usart->STATUS;
}
```

# Non-blocking character reception

- Application of USART\_StatusGet() function:

```
/* Infinite loop */  
while (1) {  
    if (USART_StatusGet(UART0) & USART_STATUS_RXDATAV) {  
        USART_Tx(UART0, USART_Rx(UART0));  
    }  
}
```

- Even more elegant solution if we implement an own non-blocking function to receive characters

```
int USART_RxNonblocking(USART_TypeDef *usart)  
{  
    int retVal = -1;  
  
    if (usart->STATUS & USART_STATUS_RXDATAV) {  
        retVal = (int)(usart->RXDATA);  
    }  
  
    return retVal;  
}
```

Implementation of non-blocking function  
(put it before the main function)

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

Application of non-blocking function  
(put it in the main function)

# Non-blocking character reception

- Remark on USART\_Tx() function:
  - If data to be sent is too much even USART\_Tx() function can be blocking – have a look at USART\_Tx()

```
/* Infinite loop */
while (1) {
    //USART_StatusGet(USART_TypeDef *usart)
    if (USART_StatusGet(UART0) & USART_STATUS_RXDATAV) {
        USART_Tx(UART0, USART_Rx(UART0));
    }
}

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;
}
}

Press 'F2' for focus
```

- Clearly seen that blocking may happen but “less severe” → USART\_STATUS\_TXBL bit is checked in STATUS register



# Non-blocking character reception

- USART\_STATUS\_TXBL bit (TXBL may appear in other registers- be careful)

## 17.5.5 USARTn\_STATUS - USART Status Register

Offset	Bit Position																																														
0x010	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0															
Reset														R	0	R	0	R	0	R	0	R	0	R	0	R	0	R	1	R	0	R	0	R	0	R	0	R	0	R	0	R	0				
Access														R		R		R		R		R		R		R		R		R		R		R		R		R		R		R		R		R	
Name														RXFULLRIGHT	RXDATAVRIGHT	TXBSRIGHT	TXBDRIGHT	RXFULL	RXDATAV	TXBL	TXC	TXTRI	RXBLOCK	MASTER	TXENS	RXENS																					

Bit	Name	Reset	Access	Description
6	TXBL	1	R	<b>TX Buffer Level</b> Indicates the level of the transmit buffer. If TXBIL is cleared TXBL is set whenever the transmit buffer is empty, and if TXBIL is set, TXBL is set whenever the transmit buffer is half-full or empty.

See [03\\_EFM32\\_Reference\\_manual\\_EFM32GG-reference\\_manual.pdf](#) on page 481

# Non-blocking character reception



TX Register to load data into TX Buffer



TX Buffer to send data to comm. line

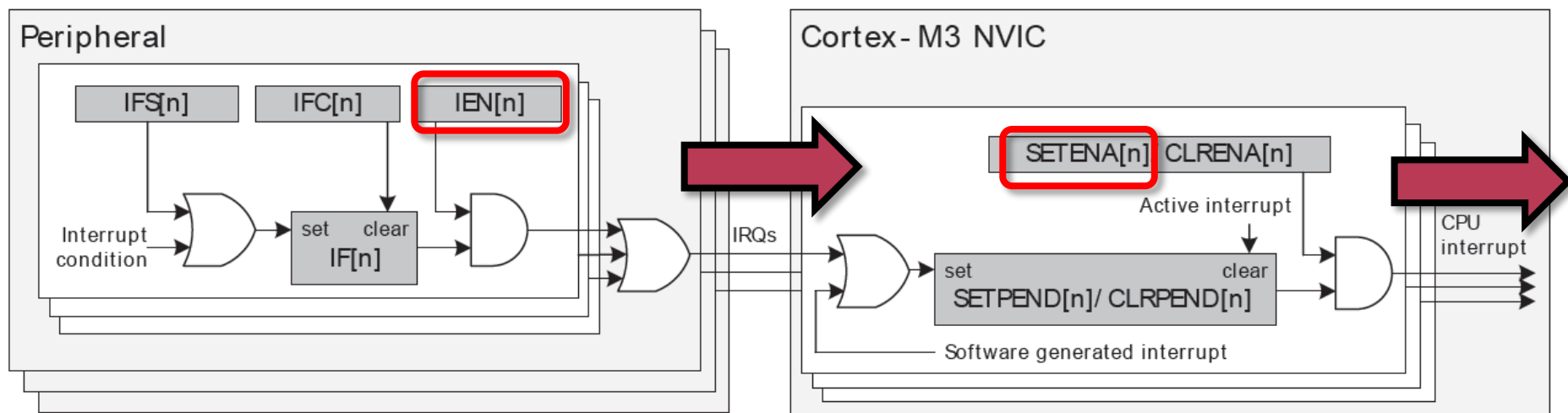
comm. line



- Operation of data transmission:
  - Generated data is loaded into TX Register only if TX Register is empty
    - Otherwise data in TX Register is overwritten and data loss may occur
  - If TX Buffer is empty data is loaded into it from TX Register
  - From TX Buffer data is sent out via the communication line (UART)
  - $R=115200\text{bps} \rightarrow 1\text{byte}$  needs  $70\mu\text{s}$
  - $T_{\text{clk}}=1/14\text{MHz}=70\text{ns} \rightarrow 1000\text{cycles}$  per byte!!!

# Interrupt-based character reception

- Problem with non-blocking character reception
  - If the main program executes a long-lasting task before repeated checking of character is done data loss may occur
  - To prevent that kind of data loss application of interrupt can be a solution



# IT initialization for a peripheral

- Initialization of IT in a general case:

- Enabling peripheral (turn perif. on, config., etc.)
- Determination of IT-handling function
- Clear of IT flag belonging to the certain IT
  - An IT request may be stuck from a previous state that can cause problem since after enabling IT a false interrupt can take action. A stuck IF can be the consequence of a non-initialized peripheral (e.g. IT occurs on a floating input)
- Enabling the IT of a certain peripheral
- Clearing of global IT flag (if needed)
- Enabling of global IT

DONE  
previously (UART\_init)

C  
O  
M  
E  
S  
  
N  
O  
W

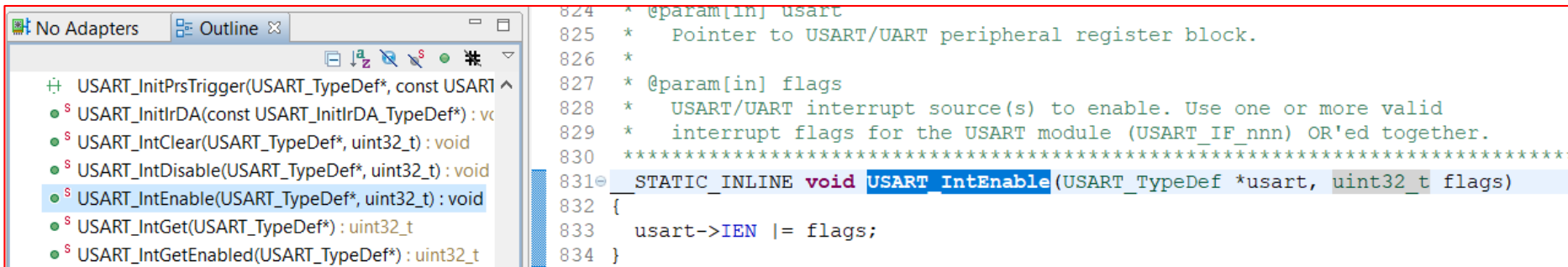
L  
A  
T  
E  
R

**NOTE: THIS SLIDE COMES FROM THE INTERRUPT TOPIC OF LECTURES  
USE THAT LECTURE AS A REFERENCE IF NEEDED**



# Interrupt-based character reception

- Check `em_usart.h` for interrupt enable function



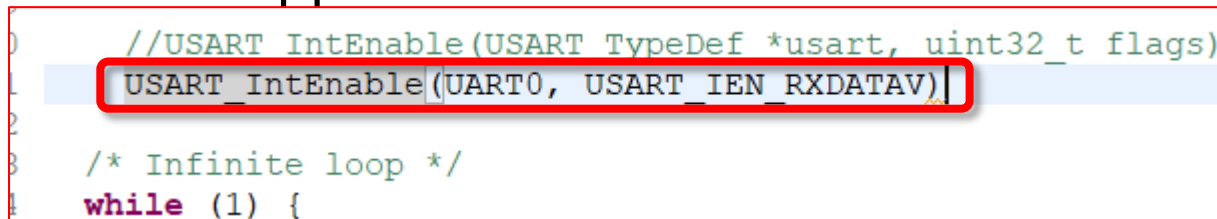
```
824 * @param[in] usart
825 *   Pointer to USART/UART peripheral register block.
826 *
827 * @param[in] flags
828 *   USART/UART interrupt source(s) to enable. Use one or more valid
829 *   interrupt flags for the USART module (USART_IF_nnn) OR'ed together.
830 *****
831 STATIC_INLINE void USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
832 {
833     usart->IEN |= flags;
834 }
```

- Insert `USART_IntEnable()` function  
flags = register content, here the 2<sup>nd</sup> bit is interesting  
(see previous slide)

- Check `efm32_gg_usart.h`

```
#define USART_IEN_RXDATAV (0x1UL << 2) /**< RX Data Valid Interrupt Enable */
```

- Code to be applied:



```
0 //USART IntEnable(USART_TypeDef *usart, uint32_t flags)
1 USART_IntEnable(UART0, USART_IEN_RXDATAV)
2
3 /* Infinite loop */
4 while (1) {
```

# Interrupt-based character reception

- Interrupts have to be cleared (all ITs) for UART
  - Check em\_usart.h for interrupt clear function

```

785 *
786 * @param[in] flags
787 *   Pending USART/UART interrupt source(s) to clear. Use one or more val
788 *   interrupt flags for the USART module (USART_IF_nnn) OR'ed together.
789 *****
790 _STATIC_INLINE void USART_IntClear(USART_TypeDef *usart, uint32_t flags)
791 {
792 #if defined (USART_HAS_SET_CLEAR)
793   usart->IF_CLR = flags;
794 #else
795   usart->IFC = flags;
796 #endif
797 }
    
```

## 17.5.19 USARTn\_IFC - Interrupt Flag Clear Register

Offset	Bit Position																																												
0x048	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													
Reset														0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
Access														W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1
Name														CCF	SSM	MPAF	FERR	PERR	TXUF	TXOF	RXUF	RXOF	RXFULL																						TXC

03\_EFM32\_Reference\_manual\_EFM32GG-reference\_manual.pdf

See page 489

# Interrupt-based character reception

- All bits in USARTn\_IFC register have to be cleared
  - A define can be found in efm32gg\_usart.h for that purpose:

```
#define USART_IFC_MASK 0x00001FF9UL /**< Mask for USART_IFC */
```

- Insert USART\_IntClear() function after UART init
- Code to be applied:

```
//USART IntClear(USART_TypeDef *usart, uint32_t flags)
USART_IntClear(UART0, USART_IFC_MASK);

//USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
USART_IntEnable(UART0, USART_IEN_RXDATAV);

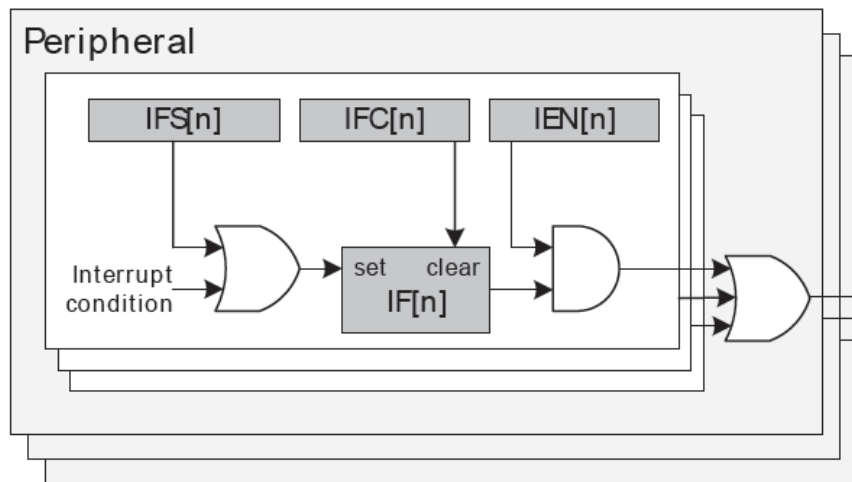
/* Infinite loop */
while (1) {
```

- This step is precarious: it is very probable that the program would work but in general, not clearing IT flags can cause a trouble

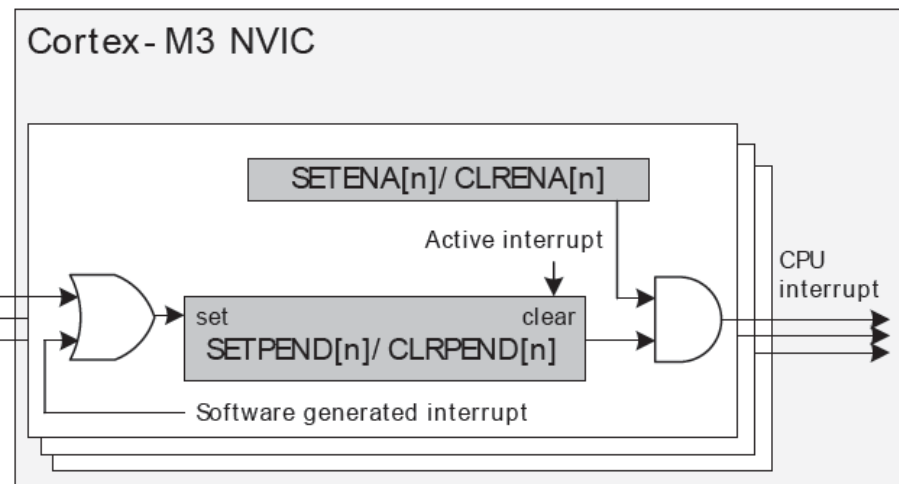


# Interrupt-based character reception

JUST DONE



COMING NEXT

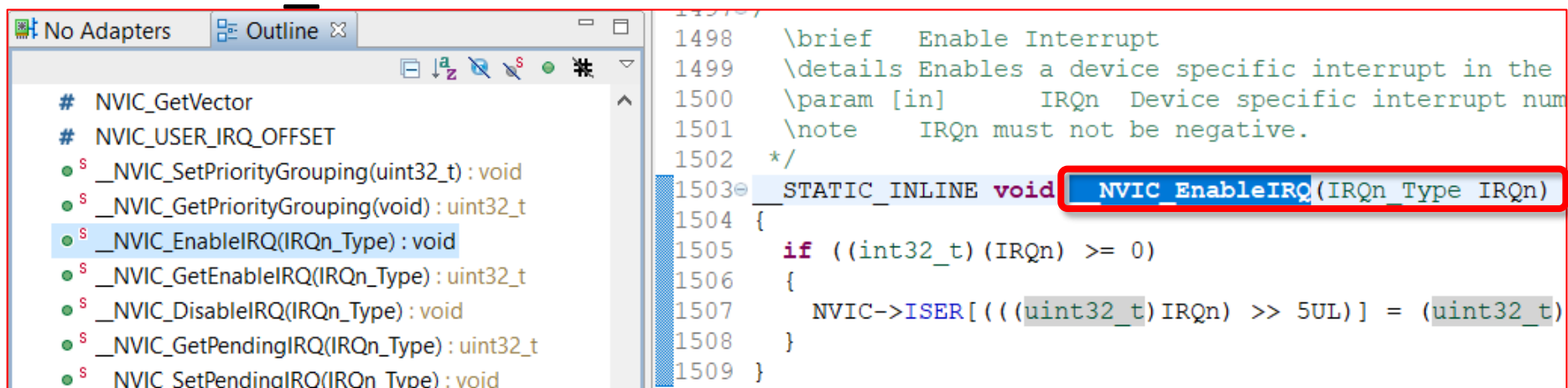


- So far UART peripheral-related IT has been dealt with
- From now let's see the core-related IT

# Interrupt-based character reception

- Core-related IT– IT for the UART has to be enabled
    - em\_decive.h + F3 (among included header files in at the top of the program)
      - > find in it efm32gg990f1024.h + F3
      - > find in it core\_cm3.h + F3
- NVIC functions are needed

- In core\_cm3.c search for



```
1498  \brief  Enable Interrupt
1499  \details Enables a device specific interrupt in the
1500  \param [in]      IRQn  Device specific interrupt num
1501  \note   IRQn must not be negative.
1502  */
1503  __STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
1504  {
1505      if ((int32_t) (IRQn) >= 0)
1506      {
1507          NVIC->ISER[(((uint32_t) IRQn) >> 5UL)] = (uint32_t)
1508      }
1509  }
```

The search results on the left show the following functions:

- # NVIC\_GetVector
- # NVIC\_USER\_IRQ\_OFFSET
- § \_\_NVIC\_SetPriorityGrouping(uint32\_t) : void
- § \_\_NVIC\_GetPriorityGrouping(void) : uint32\_t
- § **\_\_NVIC\_EnableIRQ(IRQn\_Type) : void**
- § \_\_NVIC\_GetEnableIRQ(IRQn\_Type) : uint32\_t
- § \_\_NVIC\_DisableIRQ(IRQn\_Type) : void
- § \_\_NVIC\_GetPendingIRQ(IRQn\_Type) : uint32\_t
- § NVIC\_SetPendingIRQ(IRQn\_Type) : void

# Interrupt-based character reception

- In `core_cm3.c` search for

- `void __NVIC_EnableIRQ(IRQn_Type IRQn)`

- `IRQn_Type IRQn` + F3 to check the possible ITs to find:

- `UART0_RX_IRQn = 20, /*!< 20 EFM32 UART0_RX Interrupt */`

- Code to be applied:

```
//USART_IntClear(USART_TypeDef *usart, uint32_t flags)
USART_IntClear(UART0, _USART_IFC_MASK);

//USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
USART_IntEnable(UART0, USART_IEN_RXDATAV);

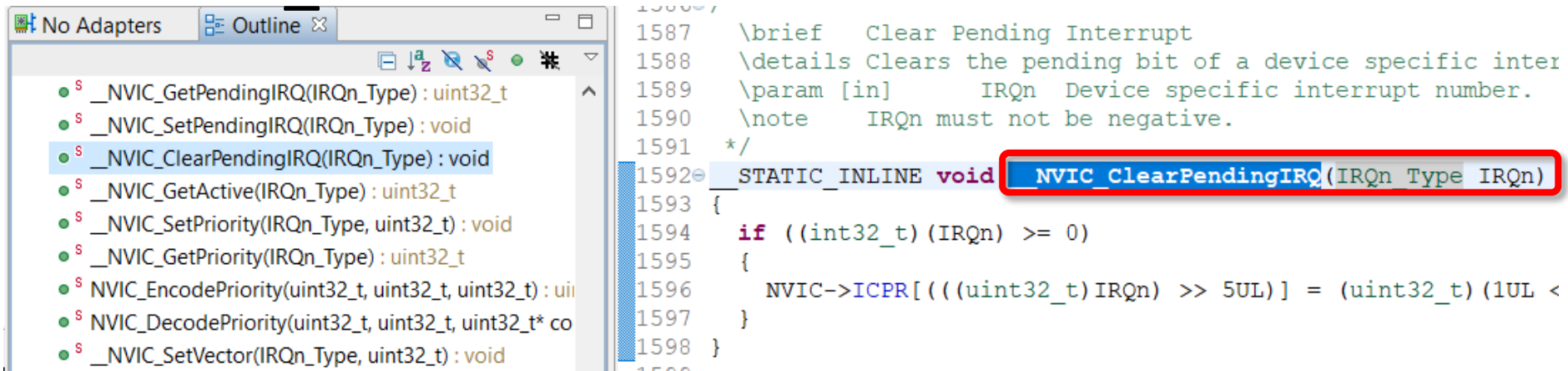
//void  NVIC_EnableIRQ(IRQn_Type IRQn)
__NVIC_EnableIRQ(UART0_RX_IRQn);

/* Infinite loop */
while (1) {
```

# Interrupt-based character reception

- Core-related IT– IT flags has to be cleared
    - em\_decive.h + F3 (among included header files in at the top of the program)
      - > find in it efm32gg990f1024.h + F3
      - > find in it core\_cm3.h + F3
- NVIC functions are needed

- In core\_cm3.c search for



```
1587  \brief  Clear Pending Interrupt
1588  \details Clears the pending bit of a device specific inter
1589  \param [in]      IRQn  Device specific interrupt number.
1590  \note           IRQn must not be negative.
1591  */
1592  STATIC_INLINE void NVIC_ClearPendingIRQ(IRQn_Type IRQn)
1593  {
1594      if ((int32_t) (IRQn) >= 0)
1595      {
1596          NVIC->ICPR[(((uint32_t) IRQn) >> 5UL)] = (uint32_t) (1UL <
1597      }
1598  }
```

# Interrupt-based character reception

## ○ In core\_cm3.c search for

- `void __NVIC_ClearPendingIRQ(IRQn_Type IRQn)`

- `IRQn_Type IRQn` + F3 to check the possible ITs to find:

```
UART0_RX_IRQn = 20, /*!< 20 EFM32 UART0_RX Interrupt */
```

## ○ Code to be applied:

```
//USART_IntClear(USART_TypeDef *usart, uint32_t flags)
USART_IntClear(UART0, _USART_IFC_MASK);
```

UART Perif. IT  
clear and enable

```
//USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
USART_IntEnable(UART0, USART_IEN_RXDATAV);
```

```
//void __NVIC_ClearPendingIRQ(IRQn_Type IRQn)
__NVIC_ClearPendingIRQ(UART0_RX_IRQn);
```

Proc. core IT  
clear and enable

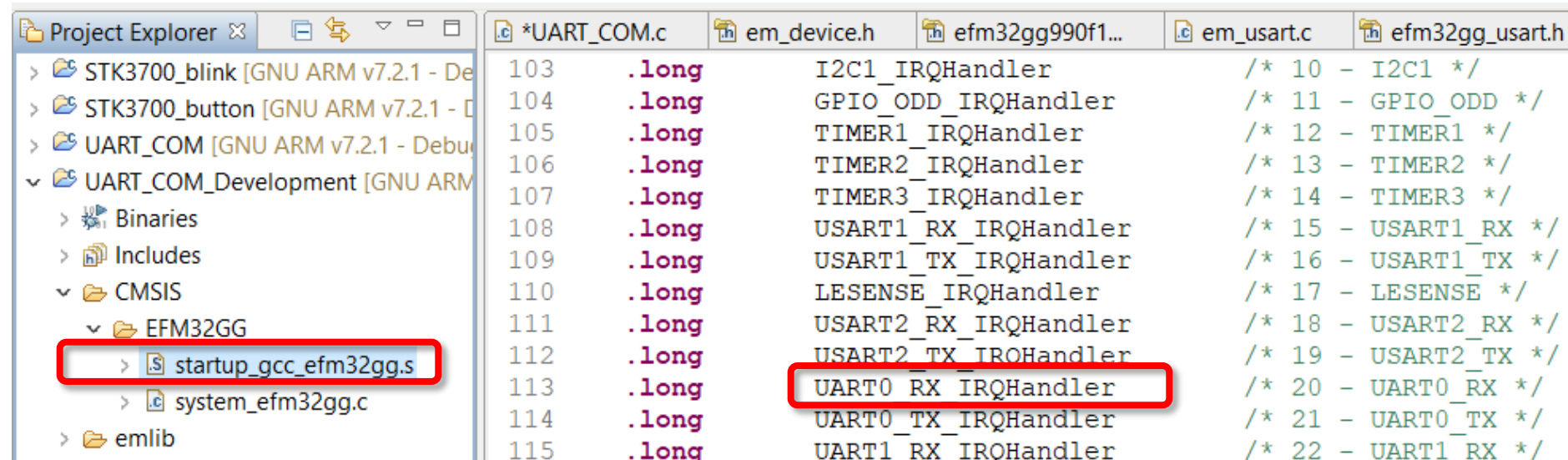
```
//void __NVIC_EnableIRQ(IRQn_Type IRQn)
__NVIC_EnableIRQ(UART0_RX_IRQn);
```

```
/* Infinite loop */
```

```
while (1) {
```

# Interrupt-based character reception

- ITs have just been correctly configured
  - When a character is received at UART0, IT is generated
- IT function has to be implemented
  - What should happen when IT event occurs
  - Check `startup_gcc_efm32gg.s` in Project Explorer



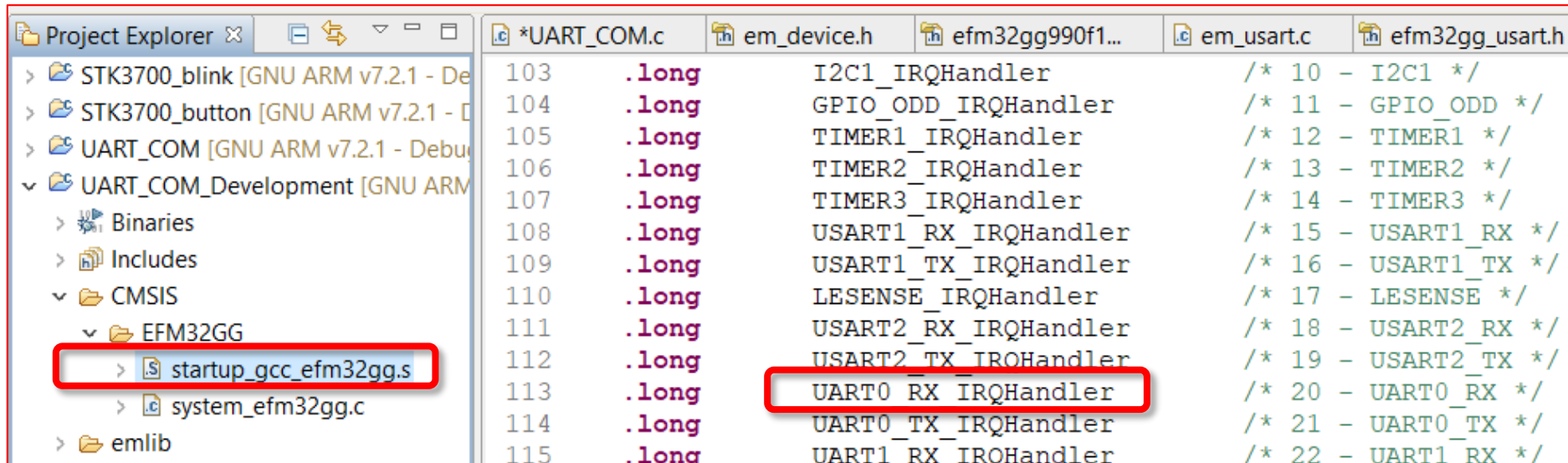
The screenshot shows the Project Explorer on the left and a code editor on the right. In the Project Explorer, the file `startup_gcc_efm32gg.s` is highlighted with a red box. The code editor displays the assembly file `*UART_COM.c` with the following content:

```
103 .long I2C1_IRQHandler /* 10 - I2C1 */
104 .long GPIO_ODD_IRQHandler /* 11 - GPIO_ODD */
105 .long TIMER1_IRQHandler /* 12 - TIMER1 */
106 .long TIMER2_IRQHandler /* 13 - TIMER2 */
107 .long TIMER3_IRQHandler /* 14 - TIMER3 */
108 .long USART1_RX_IRQHandler /* 15 - USART1_RX */
109 .long USART1_TX_IRQHandler /* 16 - USART1_TX */
110 .long LESENSE_IRQHandler /* 17 - LESENSE */
111 .long USART2_RX_IRQHandler /* 18 - USART2_RX */
112 .long USART2_TX_IRQHandler /* 19 - USART2_TX */
113 .long UART0_RX_IRQHandler /* 20 - UART0_RX */
114 .long UART0_TX_IRQHandler /* 21 - UART0_TX */
115 .long UART1_RX_IRQHandler /* 22 - UART1_RX */
```

The line `UART0_RX_IRQHandler` is highlighted with a red box in the code editor.

# Interrupt-based character reception

- Check startup\_gcc\_efm32gg.s in Project Explorer
- Search for UART0\_RX\_IRQHandler:



The screenshot shows the Project Explorer on the left with 'startup\_gcc\_efm32gg.s' selected under 'EFM32GG'. The main editor displays the code in 'em\_device.h' with 'UART0\_RX\_IRQHandler' highlighted in a red box. The code lists various interrupt handlers with their corresponding interrupt numbers and names.

```
103 .long I2C1_IRQHandler /* 10 - I2C1 */
104 .long GPIO_ODD_IRQHandler /* 11 - GPIO_ODD */
105 .long TIMER1_IRQHandler /* 12 - TIMER1 */
106 .long TIMER2_IRQHandler /* 13 - TIMER2 */
107 .long TIMER3_IRQHandler /* 14 - TIMER3 */
108 .long USART1_RX_IRQHandler /* 15 - USART1_RX */
109 .long USART1_TX_IRQHandler /* 16 - USART1_TX */
110 .long LESENSE_IRQHandler /* 17 - LESENSE */
111 .long USART2_RX_IRQHandler /* 18 - USART2_RX */
112 .long USART2_TX_IRQHandler /* 19 - USART2_TX */
113 .long UART0_RX_IRQHandler /* 20 - UART0_RX */
114 .long UART0_TX_IRQHandler /* 21 - UART0_TX */
115 .long UART1_RX_IRQHandler /* 22 - UART1_RX */
```

- UART0\_RX\_IRQHandler is a weak function so it can be overdefined in the program without causing any error:

```
/* Macro to define default handlers. Default handler
 * will be weak symbol and just dead loops. They can be
 * overwritten by other handlers.
 */

.macro def_irq_handler handler_name
.weak \handler_name
.set \handler_name, Default_Handler
.endm
```

# Interrupt-based character reception

- Implementation of IT function in the program code
- UART\_RX\_IRQHandler function has to be defined before the main function
  - During IT the received data has to be sent to UART

- Code to be applied:

```
uint8_t rx_data;  
  
void UART0_RX_IRQHandler(void) {  
    rx_data=USART_Rx(UART0);  
    USART_Tx(UART0, rx_data);  
    USART_IntClear(UART0, _USART_IFC_MASK);  
}  
  
int main(void)  
{
```

- Note: no input parameter and no return value  
-> **void func(void){**  
    what happen during IT;  
    clear IT flag; **}**



# Appendix: code – a working version

```
1 #include "em_device.h"
2 #include "em_chip.h"
3 #include "em_cmu.h"
4 #include "em_gpio.h"
5 #include "em_usart.h"
6 #include "em_core.h"
7 #include "em_emu.h"
8
9 uint8_t rx_data;
10
11 void UART0_RX_IRQHandler(void) {
12     rx_data=USART_Rx(UART0);
13     USART_Tx(UART0, rx_data);
14     USART_IntClear(UART0, _USART_IFC_MASK);
15 }
16
17 int main(void)
18 {
19     /* Chip errata */
20     CHIP_Init();
21
22     // Enable clock for GPIO
23     CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_GPIO;
24
25     // Set PF7 to high
26     GPIO_PinModeSet(gpioPortF, 7, gpioModePushPull, 1);
27
28     // Configure UART0
29     // (Now use the "emlib" functions whenever possible.)
30
```

# Appendix: code – a working version

```
30
31 // Enable clock for UART0
32 CMU_ClockEnable(cmuClock_UART0, true);
33
34
35 // Initialize UART0 (115200 Baud, 8N1 frame format)
36
37 // To initialize the UART0, we need a structure to hold
38 // configuration data. It is a good practice to initialize it with
39 // default values, then set individual parameters only where needed.
40 USART_InitAsync_TypeDef UART0_init = USART_INITASYNC_DEFAULT;
41
42 USART_InitAsync(UART0, &UART0_init);
43 // USART0: see in efm32ggf1024.h
44
45 // Set TX (PE0) and RX (PE1) pins as push-pull output and input resp.
46 // DOUT for TX is 1, as it is the idle state for UART communication
47 GPIO_PinModeSet(gpioPortE, 0, gpioModePushPull, 1);
48 // DOUT for RX is 0, as DOUT can enable a glitch filter for inputs,
49 // and we are fine without such a filter
50 GPIO_PinModeSet(gpioPortE, 1, gpioModeInput, 0);
51
52 // Use PE0 as TX and PE1 as RX (Location 1, see datasheet (not refman))
53 // Enable both RX and TX for routing
54 UART0->ROUTE |= UART_ROUTE_LOCATION_LOC1;
55 // Select "Location 1" as the routing configuration
56 UART0->ROUTE |= UART_ROUTE_TXPEN | UART_ROUTE_RXPEN;
57
```

# Appendix: code – a working version

```
58 //USART_IntClear(USART_TypeDef *usart, uint32_t flags)
59 USART_IntClear(UART0, _USART_IFC_MASK);
60
61 //USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
62 USART_IntEnable(UART0, USART_IEN_RXDATAV);
63
64 //void __NVIC_ClearPendingIRQ(IRQn_Type IRQn)
65 __NVIC_ClearPendingIRQ(UART0_RX_IRQn);
66
67 //void __NVIC_EnableIRQ(IRQn_Type IRQn)
68 __NVIC_EnableIRQ(UART0_RX_IRQn);
69
70 /* Infinite loop */
71 while (1) {
72     //USART_StatusGet(USART_TypeDef *usart)
73     //if (USART_StatusGet(UART0) & USART_STATUS_RXDATAV) {
74     //    USART_Tx(UART0, USART_Rx(UART0));
75     // }
76 }
77 }
```