

EMQ Framework

QCS-Introduction course

Lesson 1

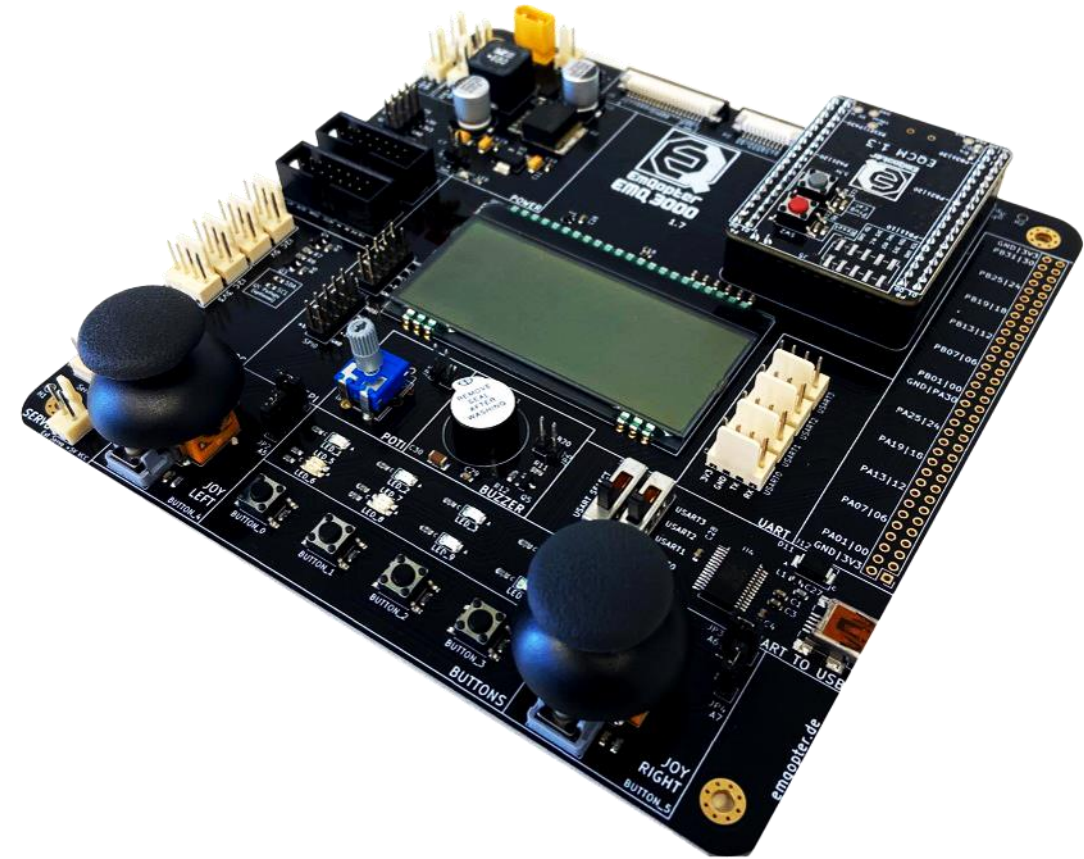


Content



- Embedded
- The development board EMQ3000
- Basics for programming
- The software library
- Display DOGM204
- Exercises

Time scope: approx. 2-3 hours





What does embedded mean?

- „**Embedded**“ = embedded
- A processor / computer / calculator is embedded in a process or sequence
- Examples:
 - Washing machine: Computer controls washing program
 - Cola vending machine: Computer controls order
 - Printer: Computer controls printing process
 - Smartphone: Computer controls calls (a.o.)
 - **Drone**: Computer controls flight
- Embedded is found everywhere these days



Examples of the embedded system



Embedded – special features on the drone example:

- Real-time: Drone must complete its tasks (e.g. position control) **on time**, otherwise it crashes
- Scarcity of resources: only limited resources (such as memory, computing power) available, as the computer needs to fly with it
- „**Black-Box**“: it is not directly possible to see what the computer is doing because there is no monitor (e.g. as with a PC)
- Hardware close to programming: more freedom, but also more room for errors
- Typical hardware: Micro Controller Unit = **MCU**
Microprocessor: Small processors vs. **Intel CPU** or **ORNL** supercomputer
- Typical programming language: **C/C++**

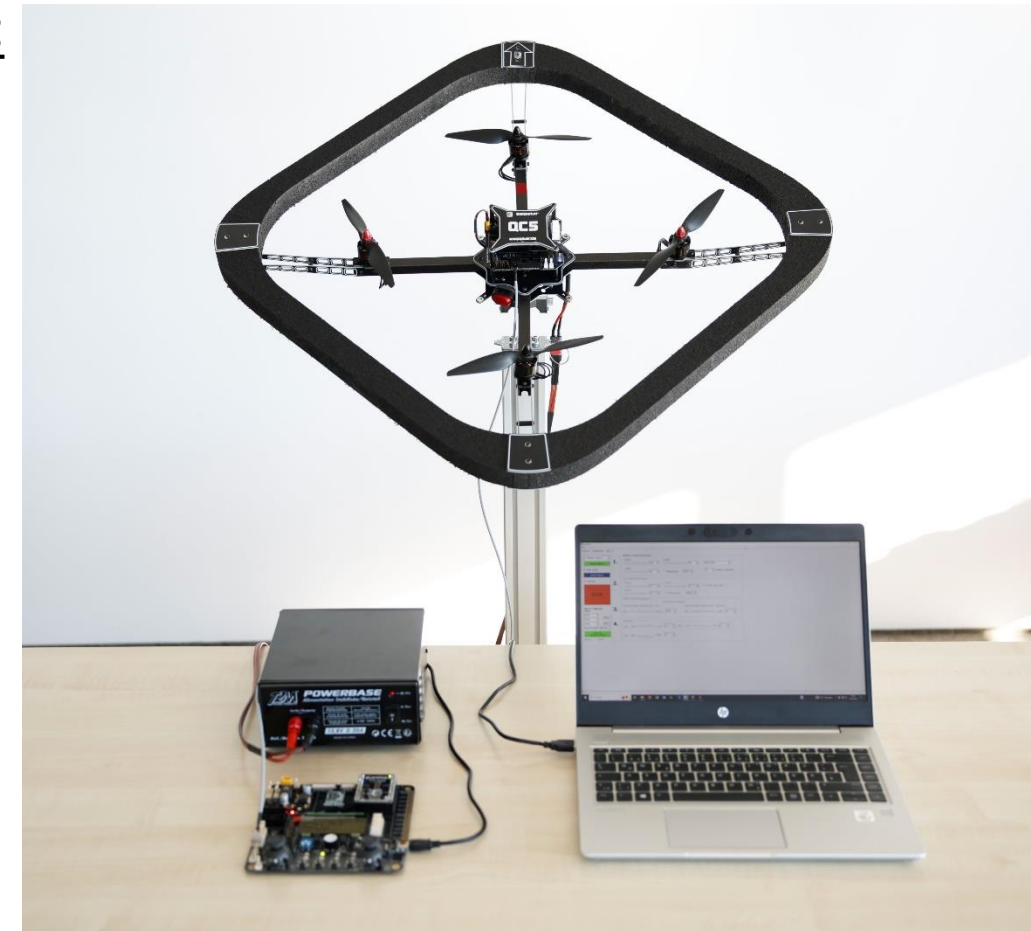


ORNL supercomputer: a bit big for drones



Embedded – special features on the drone example:

- Program starts as soon as power is switched on
- Power supply/Plug in battery → Program runs immediately → Motors can theoretically switch on at any time
- Program is permanently stored in the so-called flash memory on the drone
- If modifications need to be carried out on the program (e.g. after programming), it must be **compiled & flashed** again





Embedded – special features on the drone example:

- Procedure consists of four steps: programming, compiling, flashing & executing
- **Compiling** -> Translating the program code into machine code:
People understand program code (e.g.: C, Java, Python) -> high-level language
Machines speak **ONLY** machine code (01010110) -> binary code
High-level language -> understandable for people, has „*grammar*“ and „*sentence structure*“
Binary code -> unreadable for people, incomprehensible
- **Flashing** -> Copying machine code from the computer into flash memory:
We use a USB-based bootloader for this. When the reset button is released, the microcontroller restarts and checks whether the boot pin is pressed. In this case, the bootloader is executed instead of the main program and the new program is written to the flash memory.



Flash-memory:

- Digital memory (e.g. USB stick)
- Non-volatile, which means that stored Information is retained permanently (even without power supply)
- **UC3A1512** (CPU of the EQCM) has **512 kilobytes** of flash memory (that's a large memory capacity for an **MCU**)

Functionality (simplified):

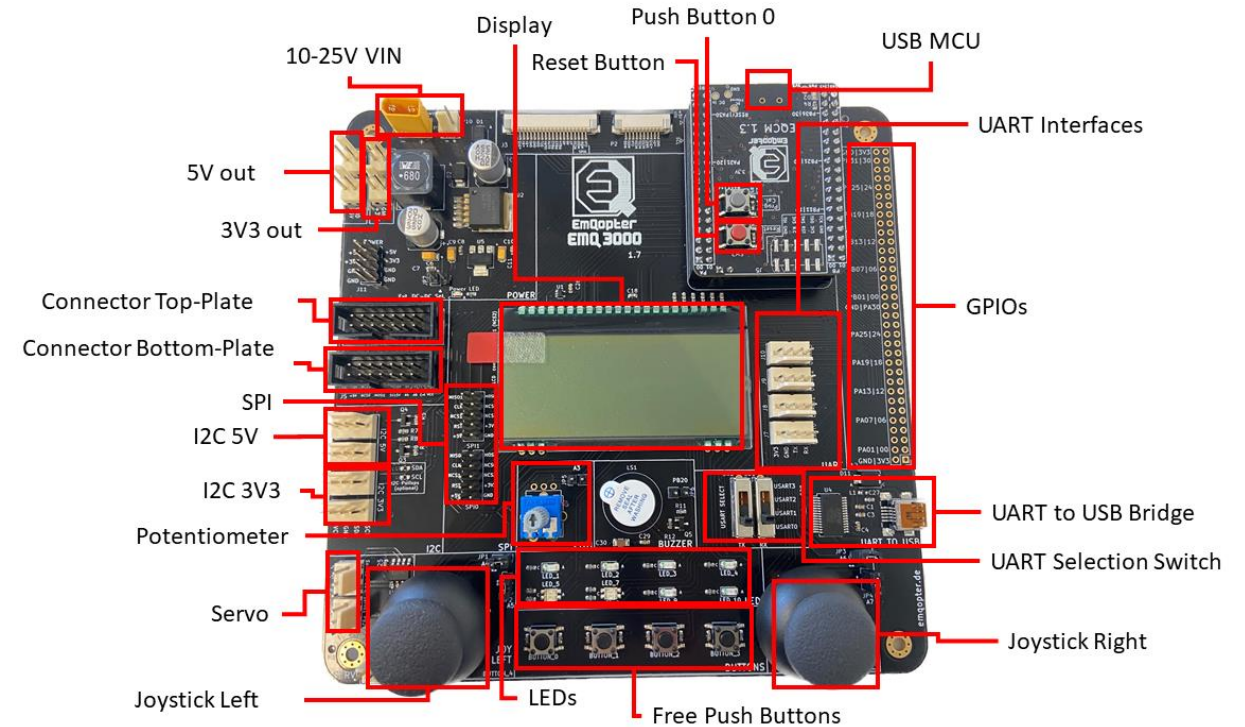
- Data = charges on the floating gate of a **MISFET**: Metal-Insulator-Semiconductor Fieldeffect Transistor
- Floating gate insulated by dielectric -> no discharge of charge
-> in simple words: isolated transistor (**transistor** = switch, here as memory)
- Nowadays: **Multi-Level-Cell** memory cells

The development board EMQ3000



Development board EMQ3000:

- Everything is easily programmable
- Extensive peripherals
Peripherals = everything which is not directly CPU
- MCU / microprocessor name:
32UC3A1512-U



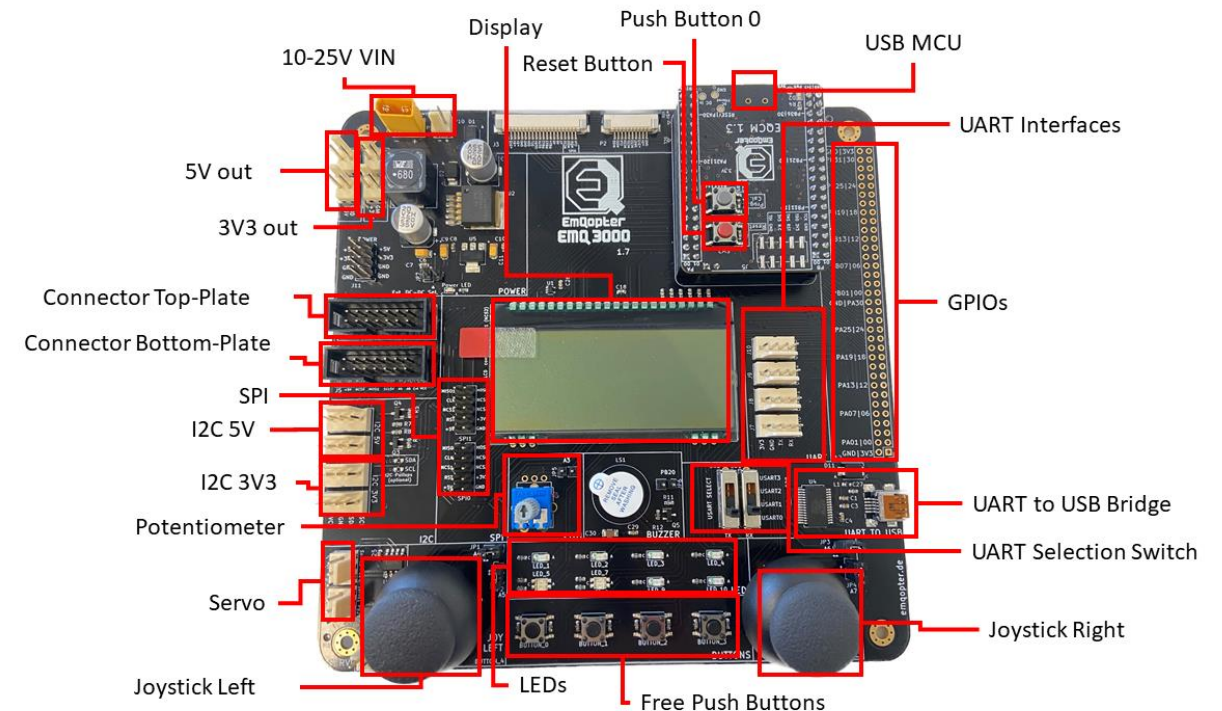
The development board EMQ3000



Development board EMQ3000:

Peripherals:

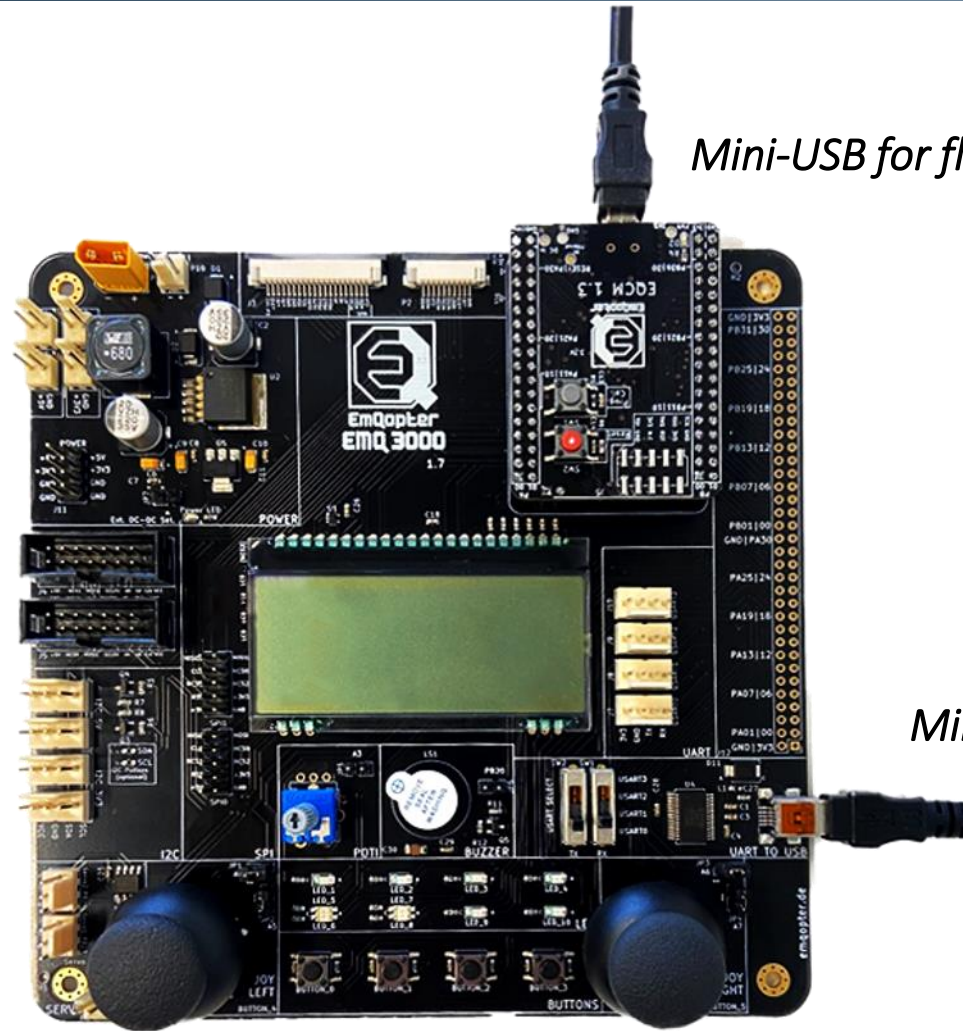
- USART = Serial communication RS232, simple predecessor of USB
- PINs = Contacts
- **GPIOs** (General Purpose Input Output)
- Classic interfaces:
 - **I²C** or **TWI**
 - **SPI**
- Buttons for inputs
- Display for indications (**DIP** abbreviated)



The development board EMQ3000



Connection to PC:



Mini-USB for flashing and power supply

Mini-USB for USART communication



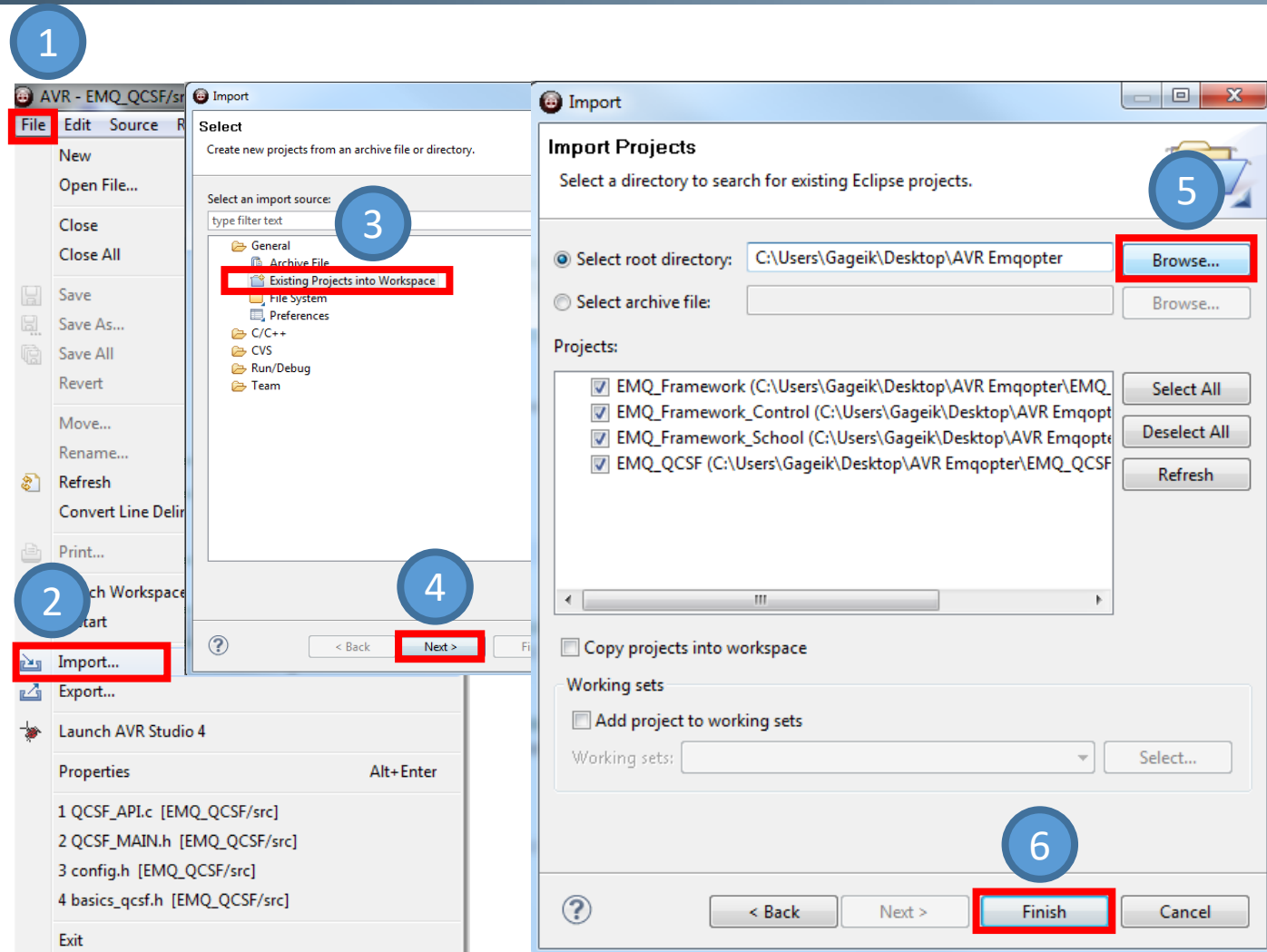
Working with the template

Import project:

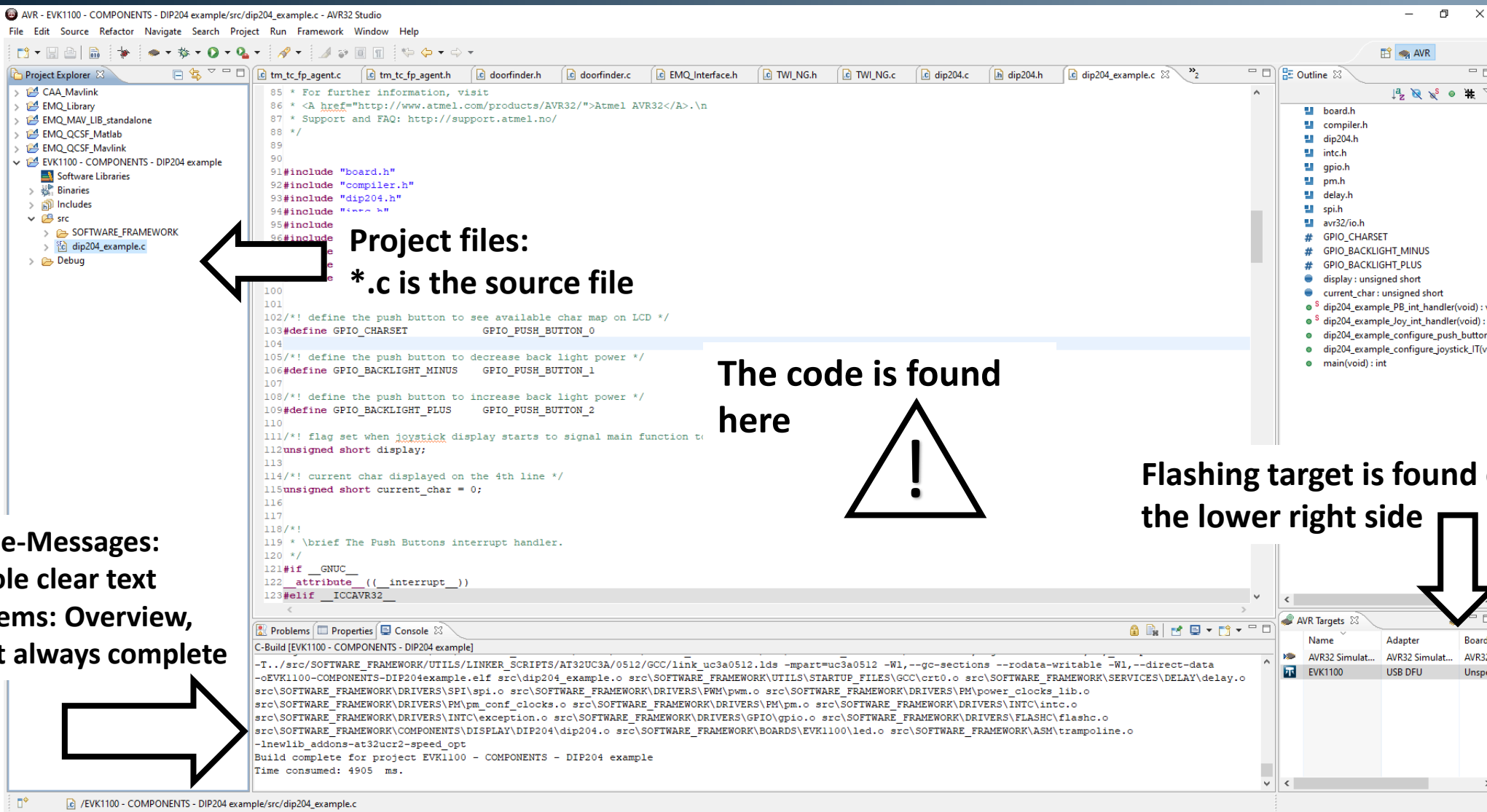
1. Select in the menu: „**File**“
2. -> „**Import...**“
3. Select „**General**“ -> „**Existing Projects into Workspace**“
4. Click on „**Next >**“
5. Click on „**Browse**“ (next to „**Select root directory**“) and select the following folder from the desktop: „**AVR Emqopter**“
6. Click on „**Finish**“

The projects should now be visible in the project explorer

Note: Only one project with the same name can be imported at a time.



Basics for programming

Project files:
*.c is the source file

The code is found here

Flashing target is found on the lower right side

Compile-Messages:
- Console clear text
- Problems: Overview, but not always complete

AVR Targets

Name	Adapter	Board
AVR32 Simulat...	AVR32 Simulat...	AVR32...
EVK1100	USB DFU	Unspec

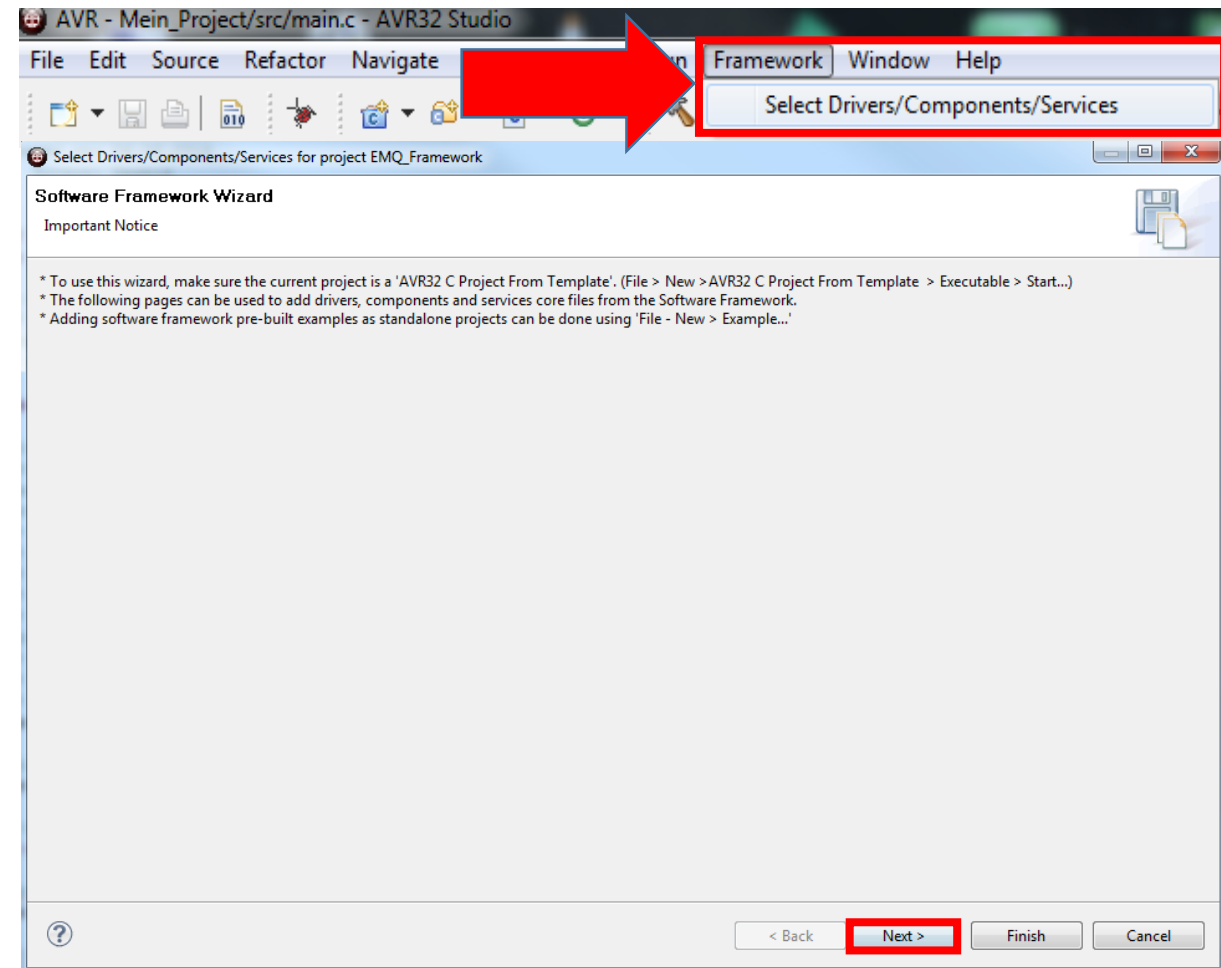


Add driver – Once for each project:

- Select the project in the project explorer
- Select: Framework -> „**Select Drivers/Components**“
- By using the EMQ library, the drivers and services are already configured correctly
- Click on „**Finish**“, with this the required „**newlib-addon speed**“ is added to the project

Note:

In case of compiler errors in connection with newlib, the driver settings should be checked.

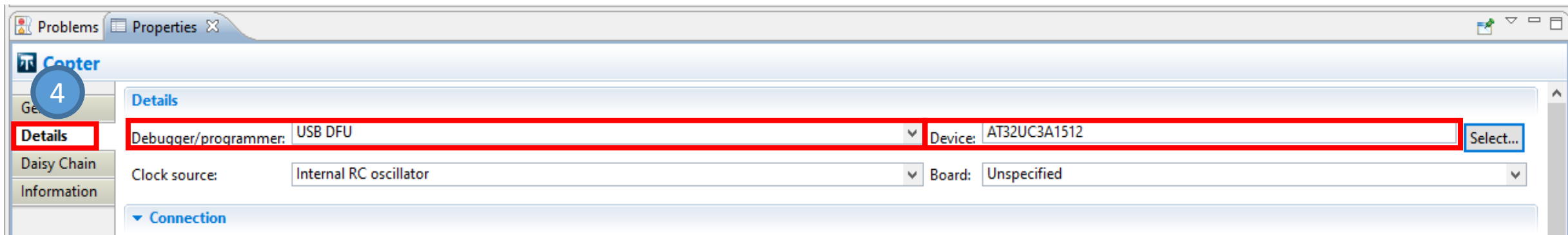
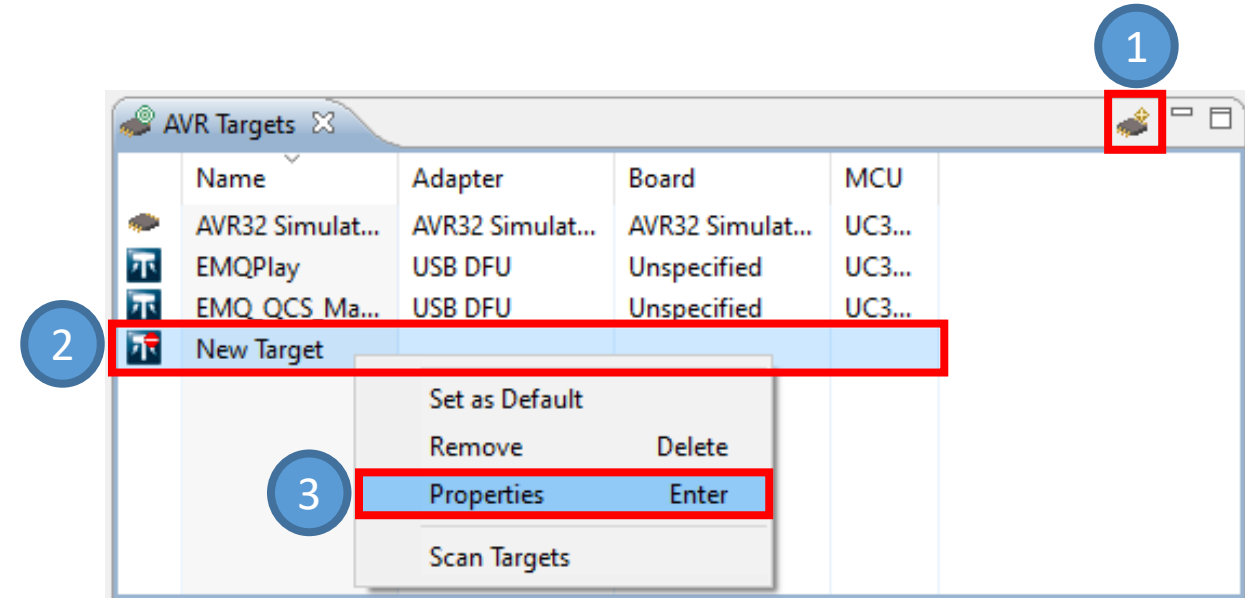


Basics for programming



Add target

1. Left-click on the „*Create a new target*“ icon
A „*New Target*“ appears in the list
2. Right-click on „*New Target*“
3. Left-click on „*Properties*“
4. Set the following in „*Details*“:
Debugger/programmer: **USB DFU**
Device: **AT32UC3A1512**



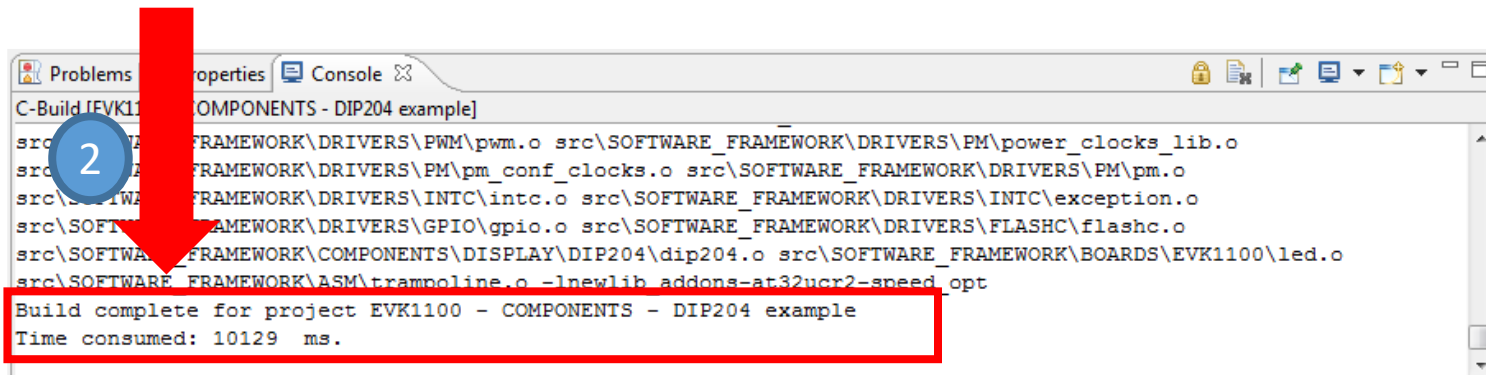
Basics for programming



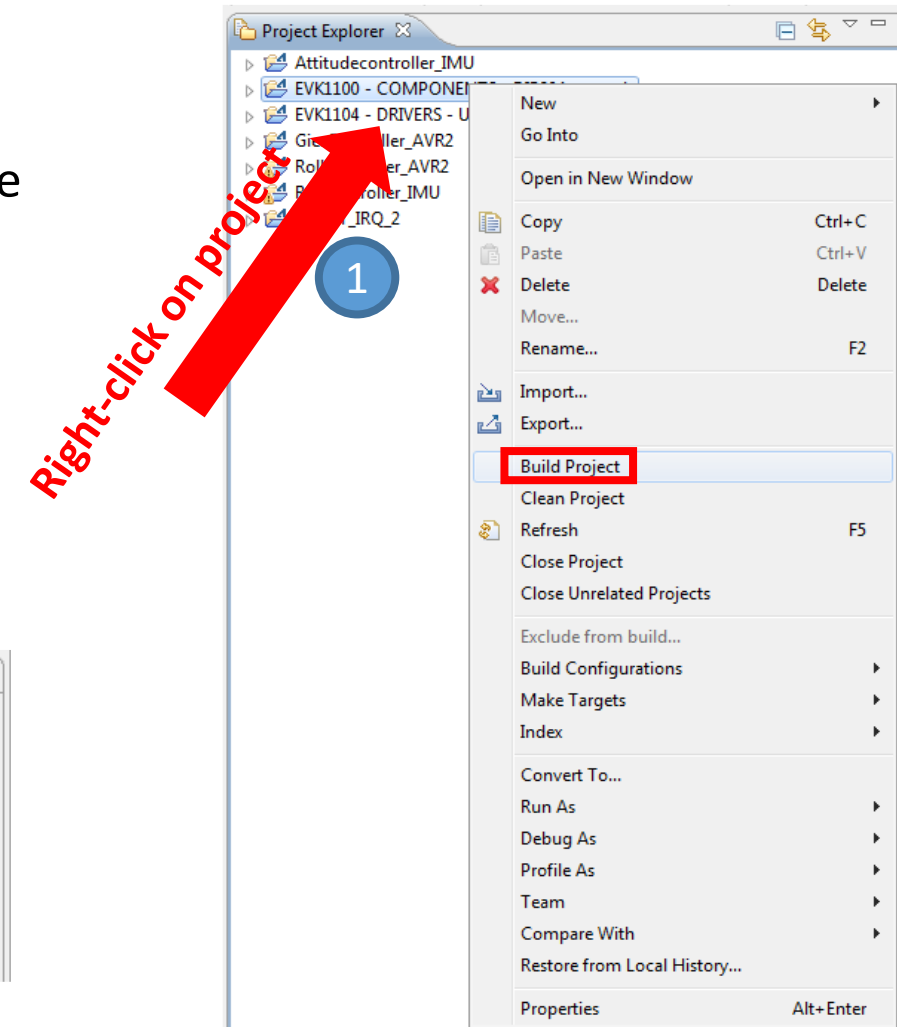
Flashing / programming in 4 steps

1. Switch on power:
Board must have power: Connect to PC via USB and bring the switch to USB position (green LED on)
2. Code must be compiled correctly:
Compile project code using „**Build Project**“ or „**Strg + B**“
Always check for success after compiling!

Check for successful compilation



```
C-Build [EVK1100 - COMPONENTS - DIP204 example]
src\SOFTWARE_FRAMEWORK\DRIVERS\PM\pwm.o src\SOFTWARE_FRAMEWORK\DRIVERS\PM\power_clocks_lib.o
src\SOFTWARE_FRAMEWORK\DRIVERS\PM\pm_conf_clocks.o src\SOFTWARE_FRAMEWORK\DRIVERS\PM\pm.o
src\SOFTWARE_FRAMEWORK\DRIVERS\INTC\intc.o src\SOFTWARE_FRAMEWORK\DRIVERS\INTC\exception.o
src\SOFTWARE_FRAMEWORK\DRIVERS\GPIO\gpio.o src\SOFTWARE_FRAMEWORK\DRIVERS\FLASHC\flashc.o
src\SOFTWARE_FRAMEWORK\COMPONENTS\DISPLAY\DIP204\dip204.o src\SOFTWARE_FRAMEWORK\BOARDS\EVK1100\led.o
src\SOFTWARE_FRAMEWORK\ASM\trampoline.o -lnewlib add-ons-at32ucr2-speed_opt
Build complete for project EVK1100 - COMPONENTS - DIP204 example
Time consumed: 10129 ms.
```



Basics for programming

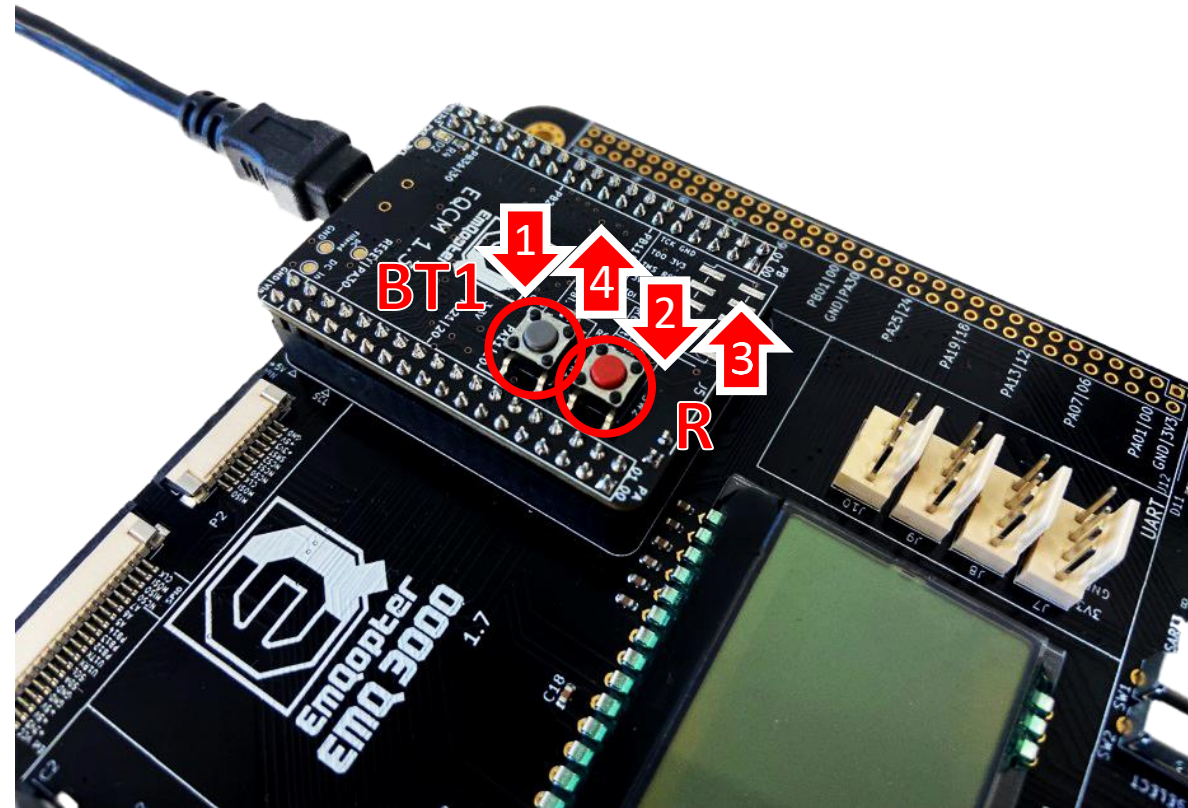


Flashing / programming in 4 steps

3. Flash-Mode: Prepare board for flashing, i.e. press „**BT1**“ button and „**R**“ reset button as follows:

- Press and hold „**BT1**“
- Then press „**R**“
- Release „**R**“
- Then release „**BT1**“

If MCU stops, i.e. display and LED1 to LED6 go out, then we are in the correct mode

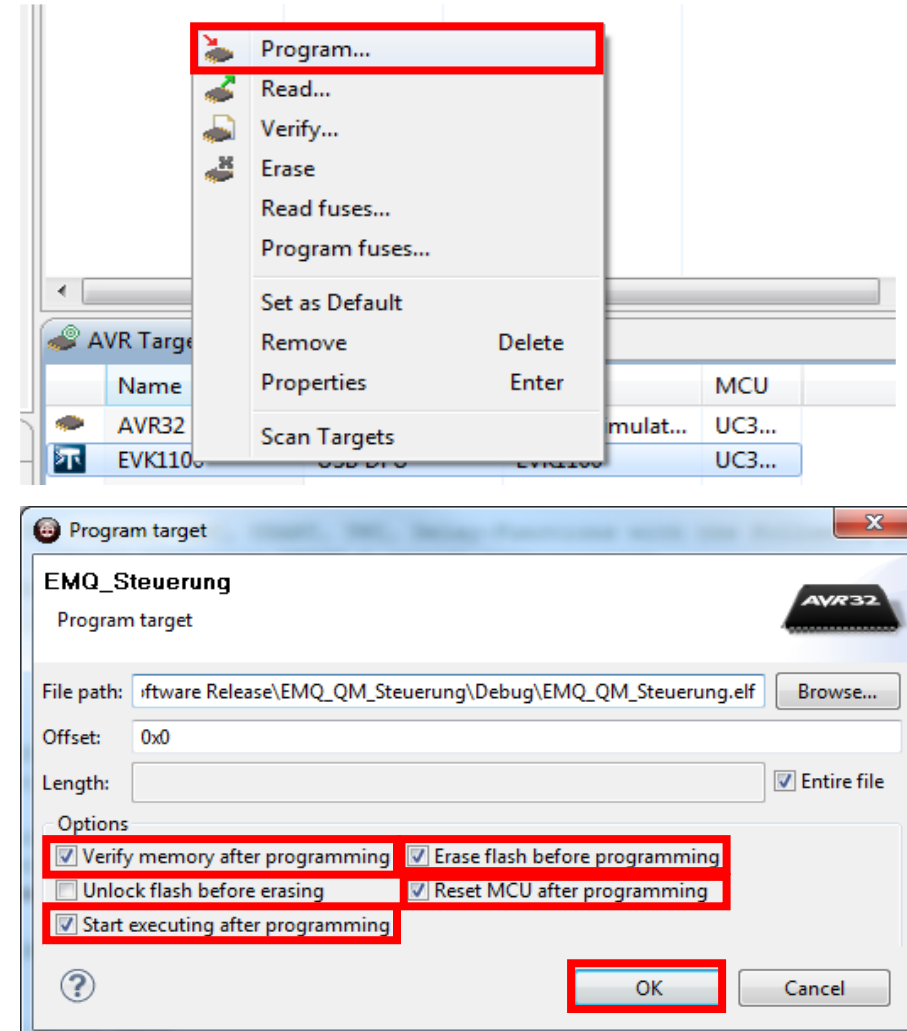




Flashing / programming in 4 steps

4. The actual flashing

- Now we want to save the program in the flash memory to enable its execution
- Right-click on the target -> Select „**Program**“
- Select binary (***.elf file**) from the following folder:
AVR_Workspace / “Project name” / Debug
- Tick as follows:
 - Verify memory -> Check after copying
 - Start executing -> Start afterwards
 - Erase flash -> Delete old program
 - Reset MCU -> Restart program
- Press **OK**



Basics for programming

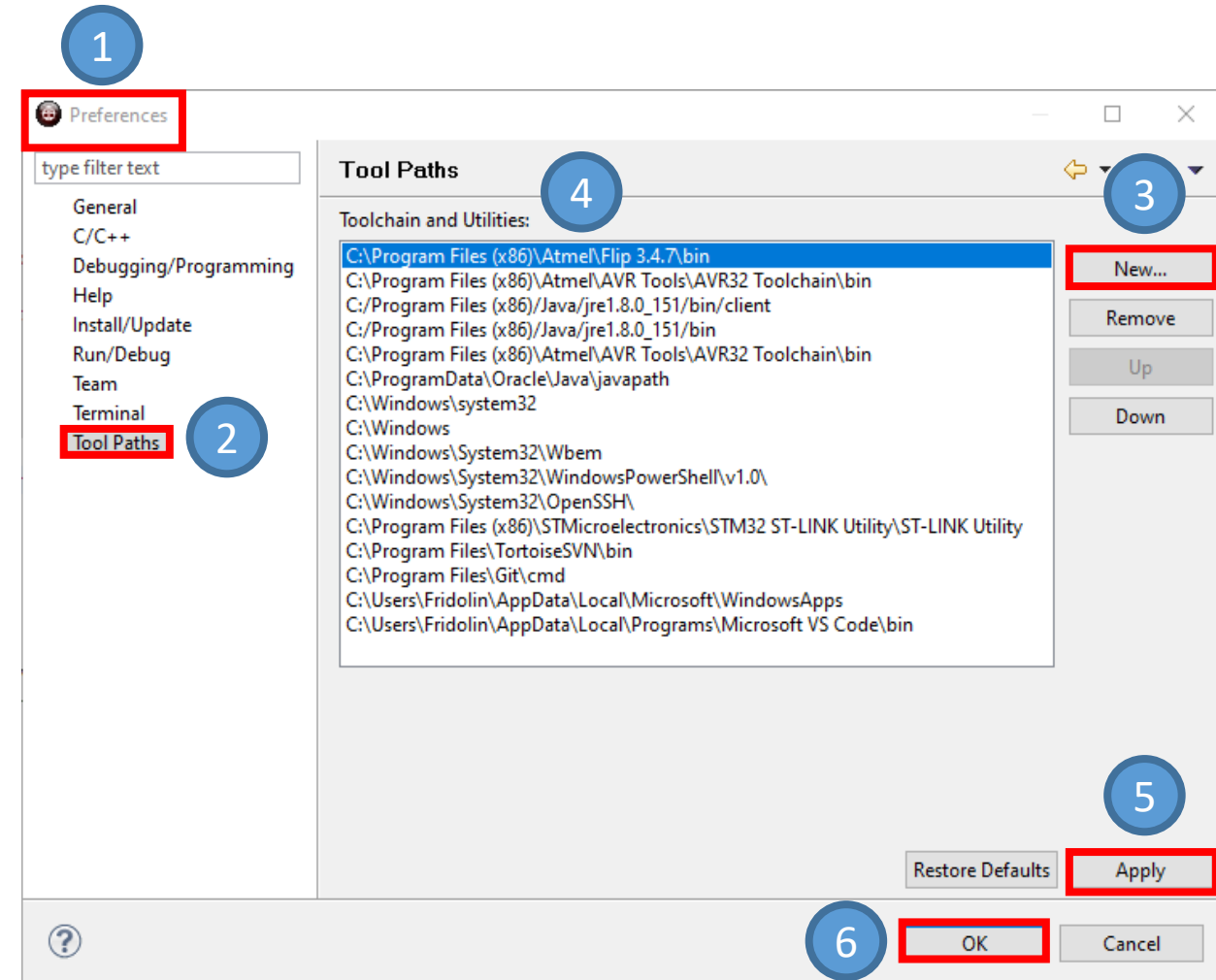


Batchisp error:

- The batchisp error is an error that occurs if the driver path is missing
- It occurs only once in the **IDE**
- If it occurs and flashing fails, it can be solved as follows

Batchisp error solution:

- Select „**Preferences**“ and then „**Tool Paths**“
- Add the following folder via "New":
C:\Program Files (x86)\Atmel\Flip 3.4.7\bin
or (depending on the Windows operating system)
C:\Program Files\Atmel\Flip 3.4.7\bin





Moving on to the first program - Hello World

Changing the information on the display to „*Hello Qopter*“

- Open the file *"my_display.c"* in the *"Framework_Basic"* project in the *"Display"* folder
- Right-click on the left edge of the window -> select *„Show-Line Number“*
- Go to line 19
- Insert the following two lines:
snprintf(line_d,20,"Hello Qopter");
write_to_display(1,line_d);
- Compile and flash: Repeat steps 1 to 4 from before
- Look at the display to see what has happened

```
9#include "../QCSF_MAIN.h"
10
11unsigned int last_DP_time;           // Timer for Display
12short dip_index = 0;                // Index for row to write
13char line_d[150] = "                "; // String to write to display
14
15// Exercise Solution Function
16// Function to write to display according to exercises, executed once
17void my_write_to_display(){
18    /*      ----      Add your code here      ----      */
19    snprintf(line_d,20,"Hello Qopter");
20    write_to_display(1,line_d);
21}
22
23// Exercise Solution Function
24// Function to write to display according to exercise, executed in a loop
25void my_renew_display(){
26    /*      ----      Add your code here      ----      */
27}
```



Overview of projects

5 projects are supplied, which differ only in the extent of the solutions they contain. This means that modules are missing from the frameworks, i.e. there are gaps to be filled. On this basis, the teacher can add solutions as required to reduce the workload or delete parts:

- The „**EMQ_QCSF**“ project contains the complete solutions and is initially intended ONLY for the teacher. The solutions can be looked up here. It can also be used for flying.
- The „**Framework**“ project contains cloze texts, i.e. the solutions (see EMQ_QCSF) must be entered / worked out by the students. This is also the minimum version provided by us.
- In the "**Framework_Control**" project, only the Control module is missing, which would have to be provided. The other solutions are already included.
- The "**Framework_Basic**" project corresponds to Framework with a few (minor) aids and simplifications.
- The "**EMQ_QM_Steuerung**" project contains the SBUS-based commanding, e.g. of a Pixhawk flight controller, in addition to the EMQ_QCSF software



Overview of projects

Project \ Module	COM	Automation	Control	Display	IMU	Quaternion	AutoPilot
EMQ_QCSF	+	+	+	+	+	+	0
Framework	-	0	-	-	-	0	0
Framework_Control	+	0	-	+	+	+	0
Framework_Basic	-	0	-	-	-	0	0
EMQ_QM_Steuerung	+	+	+	+	+	+	+

Explanation of symbols:

- + Module with example solution
- Module with cloze text
- 0 Module not included



Overview Framework

EMQ Framework Basic - Code explained

Files / Modules:

▪ COM	->	Communication	Exercise part / Module
▪ Control	->	Control	Exercise part / Module
▪ Display	->	Display	Exercise part / Module
▪ IMU	->	Sensors / IMU	Exercise part / Module
▪ Libs	->	Library / Interfaces	
▪ Quaternion	->	Quaternions	Exercise part / Module
▪ basics_qcs.h	->	Basic settings	
▪ QCSF_API	->	Application connection level, connects main functions with drivers	
▪ QCSF_MAIN	->	Main program, top level	



EMQ Framework Basic

Libs / Library / Library functions:

- Located in the „**Libs**“ folder
 - Contains ready-made functions and templates
- „**Data types**“ (EMQ_Interface_Data):
 - Contains the data types of the solution or our recommendation. If the data types are used in combination with functions of the library, the data types must not be changed
- „**Functions**“ (EMQ_Interface.h)
 - Functions permanently implemented in the library that can be used directly as given
- „**libemq.a**“ is the library supplied by us, already compiled, where these functions are implemented

Basics for programming



basics_qcs.h / Basic settings

- The motors can be activated with „**ENGINE_ACTRL_ON**“
-> deactivated by default for safety reasons

Try it out:

- | | |
|-------------------|-------------------|
| 1.) Comment in | 5.) Comment out |
| 2.) Clean + Build | 6.) Clean + Build |
| 3.) Flash | 7.) Flash |
| 4.) Press PB0 | 8.) Press PB0 |
- Further settings:
 - The model and SW version are shown here
 - The **LED-PIN-Out** depends on the board:
 - No FLUG = External control via **EMQ3000**
 - FLUG = Controlled by **EQCM onboard**
 - For us here: Irrelevant

EMQ Framework Basic

```
basics_qcs.h
/*
 * basics.h
 *
 * Created on: 11.05.2017
 * Author: Sageik
 */

#ifndef BASICS_H_
#define BASICS_H_

// #define ENGINE_ACTRL_ON // Aktiviert Motoren und Regelung

#define MODELL_NAME "I QCS Basic |\n"
#define MODELL_VERSION "Rel. 01_18|\n"

// --- Select Hardware Configuration here
#define NEW_HW_CONFIG // Activate, if hardware Config released after 28/11/2017
// New HW Config = all with EQCM such as EMQ3000 and Quadrokopter Mainboard with EQCM

#define USED_BOARD_ID GROUNDSTATION_BOARD

#endif /* BASICS_H_ */
```

EMQ QCSF

```
// #define FLUG // Compile without Display
// For Flying-Config better

#ifndef FLUG
#define ENGINE_WITH_SWITCH_DEBUG // Set Motor
#define DISPLAY_ON // Activate I
#define USED_BOARD_ID GROUNDSTATION_BOARD
#else
#define USED_BOARD_ID ON_BOARD
#endif
```



Excursus - Preprocessor instructions

- Compiler instructions that are executed before compilation
- These are used to set configurations / parameters (program settings, see above)
- **#define KENNUNG** Defines an identifier that can be queried as follows, for example:

#ifdef KENNUNG

...

#endif

The code (...) is then only executed if the identifier is defined.

- **#define BEZEICHNER ERSATZ** Preprocessor replaces BEZEICHNER (identifier) everywhere with ERSATZ (replacement)
- E.g.: **#define PI 3.14159**
double radius = 1;
double circumference = 2 * PI * radius -becomes -> double circumference = 2 * 3.14159 * radius;
double area = PI * radius * radius; -becomes -> double area = 3.14159 * radius * radius;



Contains data types in EMQ_Interface_Data, for:

PID(-Controller)-parameter	P, I, D
(3D) vector	X, Y, Z
IMU-Data	RPY, RPY_RATE, quaternion
IMU-Raw data	Vectors for gyrometer and accelerometer
Motor control values	4 bytes for the 4 motors
Control data	engine_on: activates motors calibrate_on: shows status of the RC calibration flying: Indicates whether the Qopter is flying blctrl_init: Indicates whether the brushless controllers need to be initialised

```

14// PID Parameter
15typedef struct {
16    double p;
17    double i;
18    double d;
19} pid_parameter;
20
21// 3D Vector
22typedef struct{
23    double x;
24    double y;
25    double z;
26} vector;
27
28// IMU Data: RPY Angle, RPY Angular Rate, Qu
29typedef struct {
30    vector rpy;
31    vector rpy_angular_rate;
32
33    double q0;
34    double q1;
35    double q2;
36    double q3;
37} imu_data;
38
39// IMU Raw Data
40typedef struct {
41    vector gyro;
42    vector acc;
43} imu_data_raw;
44
45// Motor Values
46typedef struct {
47    unsigned char M1;
48    unsigned char M2;
49    unsigned char M3;
50    unsigned char M4;
51} motor_data;
  
```



■ Remote control (RC), 6 channels, 7 values

Remote Data	Remote Data Raw (Raw value)	Explanation
gas	throttle	Throttle stick (left stick in up + down direction)
soll_angle_roll	roll	Roll stick (right stick in left + right direction)
soll_angle_pitch	pitch	Pitch stick (right stick in up + down direction)
soll_angle_yaw	yaw	Yaw stick (integrative) (left stick in left + right direction)
angle_yaw	yaw	Yaw stick (left stick in left + right direction)
active	-	0, if no active connection (e.g. no timeout), otherwise 1
calibrated	-	1 if RC successfully calibrated, otherwise 0
power_sw	gear_switch	Power switch -> Starts motors (A Switch)
height_sw	gear_switch	Height switch for height control, if available (H switch)
automation_state	flightmode_switch	Additional function (B Switch)
automation_state2	flightmode_switch	Additional function (G Switch)



Dx6E



- **Control Data:** Set values for roll, pitch and yaw as well as motor throttle value
- **TWI packet frame** for communication via **TWI / I²C**
The standard prescribes that first the TWI address, then the command and then the data should be placed on the bus.
The framework expects the **7bit TWI adress** and adds the read or write bit on its own.
 - chip: TWI slave address (7bit)
 - addr: Command (usually register address)
 - addr_length: Length of the command in bytes (usually 1)
 - buffer: Pointer to the data: Read / write
 - length: Data length: Number of bytes that are read or written
- **Datenstrom** dataStream
 - data: Pointer to the data size
 - size: Number of data in bytes
- **ggaData:** Data from the GPS receiver (AddOn): Time, position (3D), quality, number of satellites



Functions of the library „EMQ Interface“

Contains ready-made functions that make life easier!

- **Hardware_Init()**
Initialises the hardware (CPU, IRQ, TC, RC, ADC, USART, TWI, Delay) according to default settings
- **Motors_Init()**
Initialises the motor communication
- **Motors_run(motor_data *m)**
Controls the motors (4 control values as parameters)
- **Motors_Start()**
Starts the motors, i.e. enables them -> sets enable flag (safety)
- **Motors_Stop()**
Stops the motors, i.e. removes enable -> clears enable flag, sets initialisation flag (ESCs)



Functions of the library „*EMQ Interface*“

- **Wait(int milliseconds)**

Wait a number of milliseconds, has a blocking effect! Never use in flight
→ otherwise crash

- **USART_write(char* line)**

Sends a string via serial USART interface. Use with care and caution, as it is relatively time-consuming!

- **Remote_calibrate()**

Calibrates the RC remote control

- **Remote_run(remoteData *remote_data)**

Reads new remote control data from the RC DMA (Direct Memory Access) and writes it to remote_data.

- **Remote_get_raw_data(remoteData_raw *remote_raw)**

As a by-product, Remote_run() also generates new raw data from the remote control commands. These can be retrieved with this function if desired.



Functions of the library „*EMQ Interface*“

- **ADC_run()**
Performs the analogue-to-digital conversion. This is required to generate new data for modules that are connected via ADC. These are currently the add-ons: IR sensors, voltage measurement
- **Jitter_Check()**
Checks the timing of the main loop and issues an error message (once).
- **Filtering_Init()**
Initialises the filters: average value + stray filter
- **Steer_Init(steerData s)**
Initialises the control data (steerData), required before flying (see code)
- **LED_Control(char led, char command)**
Controls the LEDs: The first parameter *led* specifies the identifier, the second parameter *command* specifies the command (e.g. ON, OFF, TOGGLE).



Functions of the „*EMQ Interface*“

- **Push_Button_1_Pressed()**
Returns the value "1" if Push Button 1 has been pressed since the last function call, otherwise the value "0".
- **Push_Button_2_Pressed()**
Analogue function like "*Push_Button_1_Pressed()*" for button 2.
- **set_pin(unsigned int pin)**
Sets a GPIO PIN to high (1). Use with caution! Can cause a short circuit, for example!
- **clr_pin(unsigned int pin)**
Sets a GPIO PIN to low (0). Use with caution! Can cause a short circuit, for example!
- **sign(double a)**
Returns the prefix: 1 or -1



Functions of the library „*EMQ_Interface*“

- **minimum(double a, double b)**
Returns the minimum of two numbers
- **saturate(double x, double grenze)**
Saturates (limits the amount of) a number x so that $|x| < \text{limit}$
Example: `saturate(-305.7,256.0) = -256.0`
- **read_from_twi(const twi_package_t *twi, int error_id)**
Reads data via TWI interface. Error_ID is a freely selectable parameter to recognise errors.
- **write_to_twi(const twi_package_t *twi)**
Writes data via TWI interface.
- **write_to_display(char dip_index, char* line)**
Writes a line to the display in the dip_index column. A maximum of 20 characters (per line) are permitted.



Functions of the library „EMQ_Interface“

- **Display_Init()**
Initialises the display: This function must be executed before the display can be used.
- **Start_Button_Init()**
Initialises the push button interrupts (PB0, PB1, PB2)
- **USART0_schreiben_double_6(double value1, value2, value3, value4, value5, value6)**
Sends 6 floating-point numbers to the serial interface (USART 0). Used for debugging. Use with care!
- **USART0_schreiben_double_6_hs(...)**
Analogue function as above. However, the output is in high resolution, i.e. with more decimal places.



Functions of the library „*EMQ_Interface*“

- **usart_read()**
Reads a data stream from the serial interface (USART 0). The data is stored in the **DMA** (=Direct Memory Access) buffer and can be processed directly from there. After the data has been processed, it **must** be released with the `usart_release()` function so that the memory area of the **DMA** can be used for new data. Otherwise there will be problems! This procedure is more efficient than copying beforehand.
- **usart_release()**
Releases the previously read memory area of the **DMA**. First a data stream must be read with `usart_read()` and processed as required, then the memory area must be released with `usart_release()`. If the memory is not released, the **DMA's** buffer will fill up and problems will occur that need to be avoided.



Further USART functions

(with x=1 for USART1 or x=3 for USART3):

- USARTx_init(unsigned int usart_baudrate);
- USARTx_schreiben(char* line);
- USARTx_schreiben_binary(char* data,int len);
- USARTx_schreiben_char(char x);
- usartx_read();
- usartx_release();

Initialise USART with baud rate

Send string

Send string of fixed length

Send a byte / character

Receive data stream

Empties receive buffer, necessary
after usart1_read()

Display and buttons (DIP)



Display:

- Type **DOGM204**
- Controlled via **SPI**
- 20 columns x 4 rows (80 characters)
- Display of information and values (variables)
- Example code:

```
char line[20];  

int wert = 3;  

sprintf(line, 20 „Wert ist :%i “, wert);  

write_to_display(1,line);
```
- Sprintf parameter as with printf:
see table

%	Meaning	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
F	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
E	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
G	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65

Good (safe) alternative:

***snprintf**(line, 20, „value ist :%i “, value);
 max. 20 characters long.*

Display and buttons (DIP)



Buttons

- The **EMQ3000** has 4 push buttons and 2 joystick buttons
- Buttons generate interrupts
- Drivers for the push buttons can be found in the library:
Push-Button PA07 is reserved for switching the motors on/off.
There are functions for buttons **PB15** and **PB16**
- Alternatively, the sample code of the AVR framework can be viewed, e.g. also for controlling the joysticks (not yet supported by our library):
Select DIP 204: -> **AT32UC3A1512** -> EVK 1100 COMPONENTS
DIP204 example
Interrupt Service Routine (ISR) for joystick:
dip204_example_Joy_int_handler()

```
/*!
 * \brief The joystick interrupt handler.
 */
#ifdef __GNUC__
    __attribute__((__interrupt__))
#elif __ICCAVR32__
    __interrupt
#endif
static void dip204_example_Joy_int_handler(void)
{
    if (gpio_get_pin_interrupt_flag(GPIO_JOYSTICK_UP))
    {
        dip204_set_cursor_position(19,1);
        dip204_write_data(0xDE);
        display = 1;
        /* allow new interrupt : clear the IFR flag */
        gpio_clear_pin_interrupt_flag(GPIO_JOYSTICK_UP);
    }
    if (gpio_get_pin_interrupt_flag(GPIO_JOYSTICK_DOWN))
    {
        dip204_set_cursor_position(19,3);
        dip204_write_data(0xE0);
        display = 1;
        /* allow new interrupt : clear the IFR flag */
        gpio_clear_pin_interrupt_flag(GPIO_JOYSTICK_DOWN);
    }
    if (gpio_get_pin_interrupt_flag(GPIO_JOYSTICK_LEFT))
    {
        dip204_set_cursor_position(18,2);
        dip204_write_data(0xE1);
        display = 1;
        /* allow new interrupt : clear the IFR flag */
        gpio_clear_pin_interrupt_flag(GPIO_JOYSTICK_LEFT);
    }
    if (gpio_get_pin_interrupt_flag(GPIO_JOYSTICK_RIGHT))
    {
        dip204_set_cursor_position(20,2);
        dip204_write_data(0xDF);
        display = 1;
        /* allow new interrupt : clear the IFR flag */
        gpio_clear_pin_interrupt_flag(GPIO_JOYSTICK_RIGHT);
    }
    if (gpio_get_pin_interrupt_flag(GPIO_JOYSTICK_PUSH))
    {
```



Necessary hardware:

- EMQ3000
- QCS / QCSF
- Mini-USB cable for power and flashing

Necessary software:

- AVR Studio 32 installed (with toolchain and flip driver)
- Project code (framework + solution) and library:
EMQ_Framework_Basic



Help and background:

The tasks are to be solved with the help of the **"Framework_Basic"** project. Add your solutions to the Display folder in the ***my_display.c file***. The following two functions are used for this purpose: The function ***my_write_to_display()*** is executed once, the function ***my_renew_display()*** is executed recurrently (every 10ms)

Exercise 1:

First, we write the text **"Hello world"** to the display in line 1 using the functions ***write_to_display(char dip_index, char* line)*** and ***sprintf(...)*** or ***snprintf(...)***. You can use slide 38 to help you with this. Insert the code into the function ***my_write_to_display()***.

Exercise 2:

Create a variable and assign it the value 3, for example. Now write the value of this variable on the display. Compare slide 38.

Exercises



Exercise 3:

Turn on LED_1 and LED_2 on the **EMQ3000** board. Use the ***LED_Control(char led, char command)*** function for this. Slide 32 serves as an aid.

Exercise 4a:

LED_4 flashes on the **EMQ3000** by default. Switch this off.

Exercise 4b:

Switch LED_3 and LED_4 on the **EMQ3000** to flashing. LED_3 should flash quickly and LED_4 should flash slowly. To do this, use the function ***LED_Control(char led, char command)*** and the help from slide 32. Remember that ***"flashing"*** is a recurring process: on-off-on-off or toggle-toggle-toggle.

Exercise 4c:

What happens if you now undo the solution from task 4a? Can you explain the behaviour?



Exercise 5:

Program a three-sided display. A different value should be displayed on each page. It should be possible to change the page number using the **PB15** button. Page 3 should be followed by page 1. The **PB16** button should increment the value of the current page.

Help and background:

The ***Push_Button_1_Pressed()*** function can be used to query whether the **PB15** button has been pressed. It returns **"1"** if the **PB15** button has been pressed since the last function call. The ***Push_Button_2_Pressed()*** function works in the same way for button **PB16**.



Typical error messages:

- **cannot find -lemq**
The library **libemq.a** is missing and must be copied to the **src/Libs** folder.
- **cannot find -lnewlib_addons-at32ucr2-speed_opt**
The configuration data is missing or incorrect. Please follow the solution on slide 13.
- **Batchisp Error**
An error that occurs if the path to the flip driver is missing. Please execute the solution from slide 21.