University of Technology and Economics Budapest
Faculty of Electrical Engineering and Computer Science

Department of Automation and Applied Informatics

Embedded and Control Systems Laboratory

Laboratory Guide

Using the STM32 communication peripherals 1-2

Viktor Kovács (Kovacs.Viktor@aut.bme.hu)
Domokos Kiss (Kiss.Domokos@aut.bme.hu)
Zoltán Szabó (Szabo.Zoltan@aut.bme.hu)
Dr. Kristóf Csorba (kristof@aut.bme.hu)

Last updated: 2022.04.04. 10:48

Introduction

The aim of the laboratory session is to introduce common communication protocols to the students, such as RS-232 (UART), I2C, SPI and CAN.
In order to prepare for the laboratory exercise

- Read this guide
- Find answers to the questions found at the end of the guide. The exercise begins with a short test containing similar (not identical) questions.
- Review C programming language syntax and concepts.
- Review the materials from the Microcontroller Based Systems course.
- Download and familiarize yourself with the template used during the exercise. Notice which module is responsible for what purpose, where and how variables are defined. Familiarize yourself with the HAL library, especially the communication modules (uart, i2c, spi, can). Take a look at the defined types, constants and functions. Check which modules use the important global constants.
- Take a look at the sensor IC datasheets
  MCP9804 I2C temperature sensor:
  http://ww1.microchip.com/downloads/en/devicedoc/22203b.pdf
  LIS302DL SPI accelerometer:
  https://www.mouser.com/ds/2/389/lis302dl-954898.pdf
  https://www.st.com/content/ccc/resource/technical/document/application_note/e9/75/73/ca/41/5c/42/14/CD00098549.pdf/files/CD00098549.pdf/jcr:content/translations/en.CD00098549.pdf
- Familiarize yourself with the IDE (System Workbench for STM32), how code completion works (Ctrl+Space) for struct fields and other elements, navigation through the template source (F3) and the HAL library source etc.
- Do not miss the parts “Important notes about the laboratory exercise” and “Preparation for the exercise”.
- Read the guide again.

Two-part exercise

The communication exercise consists of two parts:

- UART + I2C
- SPI + CAN

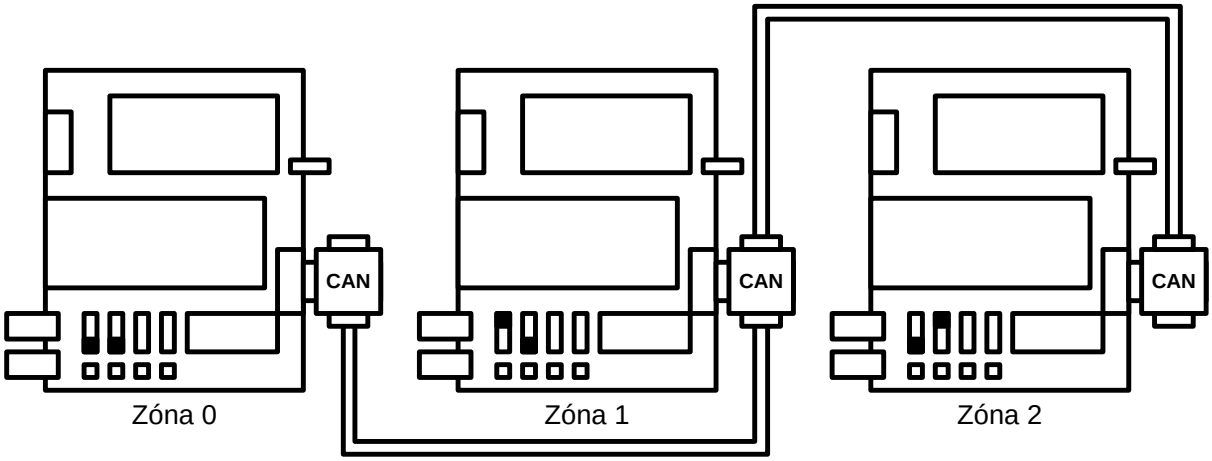
Both sessions are 4x45 minutes long. The first session deals with UART and I2C, the second deals with SPI and CAN. The guide handles the two exercises as a whole. Tests and tasks will contain only one set of communication protocols at a time. However, both sessions require skills in programming and overview of the tasks.

Overview of the exercise

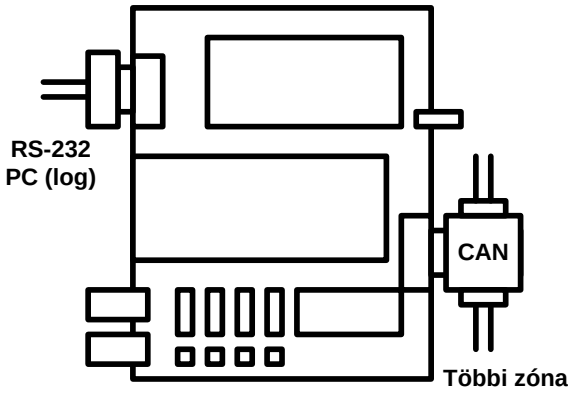
The laboratory tasks are related to a fictional multi-zone security system. The software template contains most of the functionality, however some missing parts of the code must be implemented. The system consists of:

- Multiple identical hardware devices (max 4), connected together over CAN bus

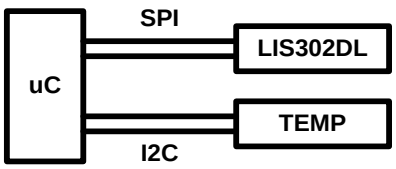
- Each device represents a zone (or room), each one measures temperature (fire alarm) and a MEMS accelerometer reports motion (tampering)
- By connecting up to 4 devices on a CAN bus, each device must have a unique (zone) ID associated (which room). This can be done by using **SW1** and **SW2** switches. The software reads the state of the switches only once after reset.



- There are two modes of the display based on the value of the **SW4** switch. Option one displays information as text: temperature and other values, or graphical display when the complete fictional floorplan with alarm events is drawn with alarm events.
- The device also transmits log information to a connected PC over the RS-232 (UART) port. The log level (amount of details) can be changed in the source code (normal/detailed).



- The device uses an I2C temperature sensor to measure temperature.
- The accelerometer is interfaced over SPI.



- **SW3** can be used for debugging purposes, a "test" alarm can be triggered. By setting **SW3** to ON state, UART and CAN communication can be tested, as the test alarm is logged over UART and sent over CAN network as well.

Structure of the embedded software, call sequences

The program consists of two main parts: initialization and the main loop. The following list shows the sequence from the main.c source.

MainLoop_Init(): entry point for the whole initialization
- **GlobalFunctions_Init()**: for the delay functions
- **Log_Init()**: UART initialization for the logging functions
  - **UART_Init()**: UART peripheral initialization
    - **MX_USART3_UART_Init()**: UART peripheral initialization
    - **HAL_UART_MspInit()**: GPIO init for the UART
- **LocalSensorProcessing_Init()**: initialization for the sensors
  - **Thermometer_Init()**: initialization for the temperature sensor
    - **I2C_Init()**: initialization of the I2C peripheral
      - **MX_I2C1_Init()**: I2C peripheral initialization
      - **HAL_I2C_MspInit()**: GPIO init for the I2C pins
    - (End of temperature sensor init)
  - **Accelerometer_Init()**: Initialization of the accelerometer sensor
    - **MX_SPI1_Init():** SPI initialization
      - **HAL_SPI_MspInit()**: pin setup for the SPI
    - (End of LIS302DL setup.)
  - (Reference temperature and local zone ID setting)
- **ButtonsSwitches_Init()**: init of buttons and switches
- **Graphics_Init()**: setup of graphics LCD
  - **GLCD_Init()**: GLCD library init
- **CAN_Init()**: CAN peripheral initialization
  - **MX_CAN1_Init()**: CAN peripheral initialization
  - **HAL_CAN_MspInit()**: pin setup for CAN

MainLoop_Go(): one iteration of the main loop
- **LocalSensorProcessing_Go()**: handling the sensors
  - **collectSensorData()**: collecting sensor data
    - **Thermometer_GetTemperature()**
    - **ButtonsSwitches_GetTestSwitchState()**
  - (Setting the alarms if any.)
- **CAN_Go()**: communication with the other devices if needed
  - (Send CAN alarm messages if the alarm state changed.)
- **Graphics_Go()**: update the display
  - (refresh the LCD screen)
- **Log_RepeatZoneData()**: Repeated logging of zone status info
- Processing incoming UART bytes if needed.

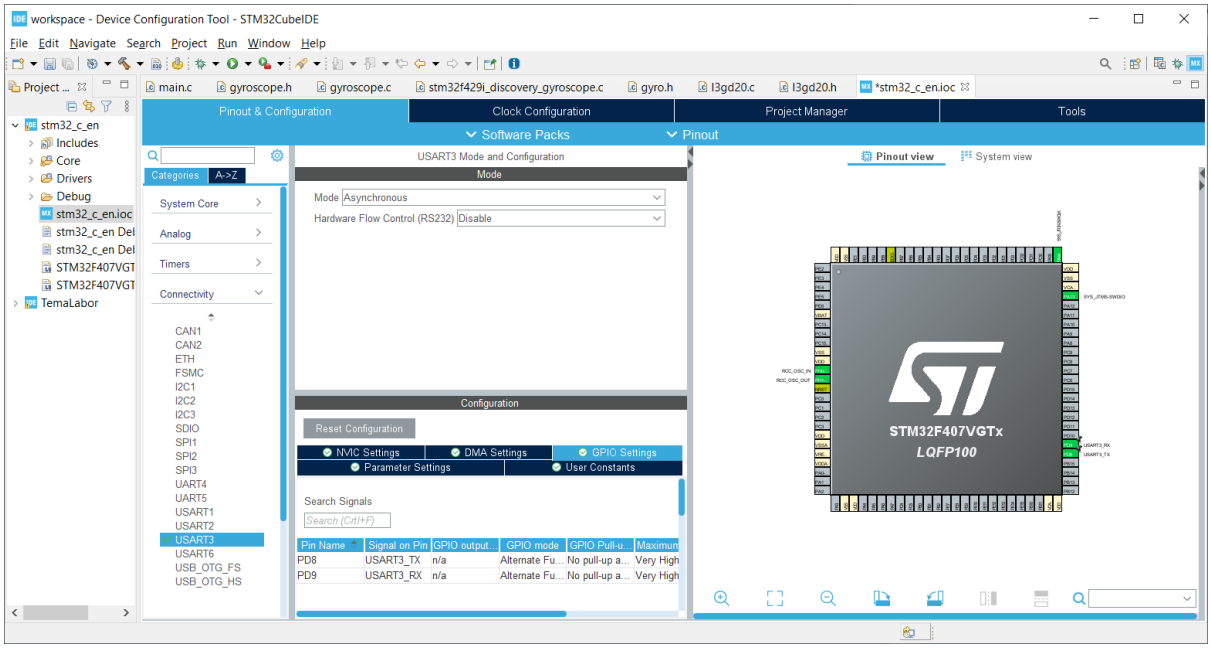
Tasks during the exercise

1 The measurement exercise starts by downloading and extracting the project template. The project should compile, but the communication parts (UART, CAN, I2C, SPI) are not yet implemented, only the test alarm (SW3) works. Flash the compiled project on the microcontroller and test the empty template software.

The rest of the tasks is about implementing different communication channels. UART communication is used between the PC and the device (**don't forget to plug in** the RS232 cable!), I2C and SPI is used on the board, CAN is used between devices. Parts of the code,

functions that need implementation is marked with “TODO” comments and task descriptions. The development environment shows these labels in the “Tasks” panel.

The development environment (STM32CubeIDE) provides a graphical user interface to set up peripherals and generates C code for the HAL library. This tool helps the configuration tremendously, however this does not mean that knowledge of the microcontroller and the HAL library capabilities and usage is not needed.



Laboratory session 1.

Task 1 – UART communication

UART is used to send logging information to a PC, optionally some control commands are sent from the PC to the device as well. For testing purposes, use "putty.exe" application to open a serial terminal: select Serial type for the session, COM1 as the port. More options need to be set in the "Serial" menu item: use 115200 baud rate, 8 data bits, no parity, 1 stop bit, no flow control.

2

The task description is found at the beginning of the bsp_uart.c:

> The main purpose of the functions is to offer implementation for the logging functions in log.c and also allow mainloop.c to receive bytes one by one.
>
> 3 In order to initialize the module, the GPIO pins must be initialized (alternate function, push-pull etc.), UART must be initialized (baud rate, stop bits etc.). A graphical user interface is available to configure the peripherals. The generated initializer function shall be called in the UART_Init() function.
>
> 4 The string sending is implemented in UART_SendString() function. This is not a blocking function, it uses interrupts in order not to block the rest of the system.
>
> 5 After implementing these functions, take a look at the UART frame using the oscilloscope (PD8 pin). Put a screenshot in the report where the beginning and the end of the frame is visible. Try to decode the meaning of each bit and the transmitted byte. Measure the baud rate with the cursor function of the oscilloscope. **Please pay extra attention not to short circuit any of the pins!** Use the RS-232 connector's screw as the ground point.

If the implementation was correct, you should see the messages of the device in the terminal window when either SW3 or SW4 is flipped.

Task 2 – Two-way UART communication

The PC can also send data over from the terminal to the microcontroller (STM32 RX line). In order to test this function one-byte commands can be implemented, such as:

Character	Command
g	Greeting, returns a hello message.
n	Normal, sets the log level to normal.
d	Detailed, sets the log level to detailed.
q	Query, returns the level of the logging.
v	Virtual test switch, use to apply a test alarm (remotely) without accessing the physical switches.
r	Repeat mode, turn on to get periodic state information of the zones (every 2sec or when states change).

6 The current level of the logging is described by the global *currentLogLevel* variable. The initializer constants *LOGLEVEL_NORMAL* and *LOGLEVEL_DETAILED* can be used to set a startup log level.

7

The virtual test switch state can be changed by calling the *ToggleVirtualTestSwitch()* function. The repeat mode is defined by the *repeatZoneDataEnabled* variable. Two constants are available as value: *REPEAT_ON* and *REPEAT_OFF*.
The task must be implemented in bsp_uart.c and mainloop.c source files.

bsp_uart.c

This task is about UART reception, the previous one only dealt with sending data. UART reception is implemented using interrupts. HAL library provides a uniform structure for handling interrupts. Most of the interrupt handling is implemented in HAL and provides callbacks for the user code that actually deals with the business logic.

First the interrupt event forces the microcontroller to save its state and jump to the address corresponding the event source in the interrupt vector table. The address is read from here, written to the PC. This address contains the **native** interrupt handler (such as USART3_IRQHandler()) which is independent from HAL. Check out USART3_IRQHandler() in stm32f4xx_it.c).

In order to utilize HAL, the programmer must call the HAL interrupt handler to take care of the interrupt (HAL_UART_IRQHandler(&huart3);). This function will call back to a user implemented functions depending on the situation. If a complete receive transaction finished, it calls the HAL_UART_RxCpltCallback() function. Reception must be allowed during the initialization in UART_Init(). In this task bytes are received one by one. The received byte is saved to lastReceivedUartCommand global variable, which is monitored by mainloop.c and handles it appropriately. This part of the task is implemented in mainloop.c.

mainloop.c

Reacts to the incoming command bytes, implemented in the MainLoop_Go() function. Rest of the task is implemented in uart.c, which is responsible for receiving data.

Task 3.1 - I²C communication

I²C is used to communicate with the temperature sensor: read in temperature values. The task is described at the beginning of the bsp_i2c.c source file.

Bsp_i2c.c provides low level function that are used by thermometer.c, that implements the protocol to communication with the thermometer IC using the functions provided by bsp_i2c.c.

In order to use the I2C function, the peripheral and the GPIO pins must be set up. This can also be done using the graphical user interface (but it is also possible to set it up manually in the code, the GUI generates such C code). Use the MARCOM.ioc file in the configurator and call the generated function in I2C_Init() function.

The I2C_ReadRegister() and I2C_WriteRegister() functions are responsible to read or write a register in the thermometer. HAL library provides a lot of help in this: it generates the start condition, I2C addressing, sending the address of the target register, reading or writing the register and the stop condition, all in the background.

In order to read a register from the thermometer, the following transaction must be executed:

START, thermometer address, write, register address, restart, thermometer address, read, [read the data], stop.

Note that even the register read starts by a write transaction, this set the address in the device where the next transaction shall read or write.

This is a common way to communicate with I2C devices, thus the HAL library has an implementation for it.

If the implementation is correct, when the temperature rises over two degrees compared to the temperature at reset, a small flame icon is drawn to the display. By setting to the display to show text information, the actual temperature is shown (use SW4!).
If the repeat mode is turned on, the temperature value is also sent over UART periodically.

Task 3.2 - I^2C inspection

Check the SCL and SDA lines using the oscilloscope. Try to understand what is going on!
SDA and SCL are located on PB6 and PB9.

Laboratory session 2.

Task 4.1 – SPI communication

SPI is used to connect the LIS302DL accelerometer to the STM32 microcontroller. The task description can be found in bsp_spi.c and bsp_accelerometer.c:

bsp_spi.c

SPI communication is implemented in this source, providing low level functions for the bsp_accelerometer.c that contains device specific functions and builds on top of the low level SPI driver. The initialization function (SPI_Init()) needs to be implemented here and all communication functions (SPI_EnableChip(), SPI_Send(), SPI_Receive() and SPI_SendReceive()) as well. These are used by bsp_accelerometer.c to communicate with the LIS302DL MEMS sensor.

Note that HAL allows reception and transmission at the same time, this is because SPI transaction is basically the exchange of data in two shift registers. During every SPI transaction, data transmission and reception occurs at the same time, however we might not care what is received (when transmission is intended) or transmitted (in case of a reception operation). That's why HAL driver provides distinct "Transmit" and "Receive" functions which hide the unnecessary data direction of the SPI transaction. There is also a TransmitReceive() function in HAL that handles both transmission and reception at the same time.

bsp_accelerometer.c

Device specific functions LIS302DL_Write() and LIS302DL_Read() are implemented here by using the low-level SPI functions. Higher level functions provided here

(Accelerometer_Init() és Accelerometer_GetTilt()) use these device specific functions to initialize and use the sensor.

WARNING! Notice that JP3 jumper near the right corner of the LCD must be open otherwise the microcontroller cannot communicate with the MEMS sensor.
In case of a correct implementation an icon of a person is shown when the device is tilted (being stolen ☺). This message is sent to the other devices as well.

Task 4.2 – SPI communication, data line inspection

Examine how the Accelerometer_GetTilt() function works. Notice that the first step of the communication is reading the "Who am I" register of the LIS302DL, in order to check whether the communication works well, and the device is actually the accelerometer. Examine what is sent over the bus (what byte, bytes are transferred which lines in what direction) and inspect these lines (SS – PE3, SCK – PA5, MISO – PA6, MOSI – PA7) using the oscilloscope! Think about which line should be used as trigger.

Task 5 – CAN communication

CAN communication is used to connect devices associated to different zones. Whenever an alarm is triggered in the zone, a message is sent over the network to alert the other devices (each one shows all the alerts on the display). When implementing this task, make sure that multiple devices are connected over the CAN bus. Notice that the CAN bus needs proper termination, so make sure that the two ends of the bus is terminated appropriately, while the center devices do not have the termination (terminating resistors can be connected using the JP5 jumper next to the CAN connector). Also make sure that each ZoneID is unique in the network (set by SW1-SW2, reset the microcontroller to update the zone ID). The task is described in bsp_can.c:

> The initialization of the CAN peripheral is similar to the others: GPIO pins and the embedded CAN module must be initialized. Use the configurator by opening MARCOM.ioc, call the generated initializer function from CAN_Init() This is complemented by setting up a CAN message filter. This filter describes which messages the peripheral listens to or ignores. As CAN uses message identifiers instead of device addressing, each device can filter which messages are relevant or not.
>
> For transmitting and receiving messages, use CanTxMsgTypeDef and CanRxMsgTypeDef type structures (global variables txMessage és rxMessage are already defined), and pass these structures during initialization. In order to send, only the fields of this structure must be filled and let the library know to send it, while receiving a message results in calling a callback function.
>
> Both sending and receiving is implemented using interrupts, in order not to block other functions and maximize performance and efficiency. Beyond that, we have to ensure to send (pass the message to the outgoing mailbox) only if CAN is ready and no other sending is in progress, and process the received message in the callback function.

Important notes about the laboratory exercise

This section contains information that might help understand the template code and allow to successfully implement the required code during the exercise.

Initialization process

During the tasks multiple communication channels and sensors must be initialized. There are three levels of this process. The lowest is the GPIO pins that allow physical connection between the microcontroller peripheral and the sensor itself. During this initialization phase parameters regarding the PGIO pins are set (input or output, push-pull or open drain etc.). The initialization of the communication peripheral consists of setting up the peripheral itself, providing information about bit rate, flow control, number of data bits etc. (in case of UART). Last there is the sensor initialization where the communication channel is used to modify parameters, setting of the sensor or even start the measurements (resolution, filters for the thermometer). These levels are separated in source files:

- Thermometer.c source contains Thermometer_Init() function, that sets up I2C communication by calling I2C_Init() and also sets up the thermometer.
- I2c.c source contains the I2C_Init() function that sets up the I2C peripheral by calling the MX_I2C1:Init() function that was generated by the GUI. This function fills in the correct values for the init structure (handle) and calls HAL_I2C_Init().
- Low level pin configurations are set up in HAL_I2C_MspInit(), which is also generated in our case.

The same process is used for all the communication channels.

Communication modes: polling, IT, DMA

HAL usually provides 3 different methods for handling communication:

- Polling mode: the code actively check whether something has happened or not and this information is returned (such as: has the transaction ended?, has the transaction ended?...). There is no automatic callback that notifies the code. This usually means that a transaction is implemented in a blocking mode (while loop is used to check the end of the transaction), so the processor actively waits (and does nothing else) until the communication finished. This means that all other activities (display refresh, using input handling, other communication events) are blocked and possibly may be missed. For example HAL_UART_Transmit() blocks and returns only after the transmission of bytes has finished. Keep in mind that communication is always several magnitudes slower than the processor.
- Interrupt based: after starting the transaction the processor starts to execute the following functions, it does not block execution. An interrupt is called when the transmission has finished. Keep in mind that the interrupt can interrupt any other executed functions, so data consistency must be taken care of. HAL_UART_Transmit_IT() does interrupt based transmission.
- DMA: (Direct Memory Access) similar to the interrupt based, the system uses the DMA peripheral, usually used to read or write multiple bytes to or from peripherals without the need for processor interaction. Such as: HAL_UART_Transmit_DMA().

During the tasks UART and CAN is handled using interrupts, I2C and SPI are used by blocking/polling mode.

The HAL ISR and callback system

HAL library significantly reduces work and programming mistakes at handling interrupts. In order to ask HAL library to handle the interrupt, the appropriate HAL interrupt handler (ie. CAN1_TX_IRQHandler) function needs to be called:

```
void CAN1_TX_IRQHandler(void)
{
	HAL_CAN_IRQHandler(&hcan);
}
```

This function handles the interrupt and calls back to the appropriate callback function (if applicable), such as HAL_CAN_RxCpltCallback(). The developer only needs to implement the callback function, that takes care of the "important" task, does not need to worry about the "infrastructure" code, checking, setting flags, etc: react to a CAN message etc.

Note that the native interrupt handler contains the specific periphery name, however the HAL function is general, can be applied to any periphery of the same type.

The handle variables

Each periphery is described or represented by a so called handle structure/variable. It contains all the information that is needed to handle and initialize the periphery.

Such as for UART, a global variable is created and the fields are filled for initialization:

```
UART_HandleTypeDef huart;
huart.Instance = USART3;
huart.Init.BaudRate = 115200;
huart.Init.HwFlowCtl = UART_HWCONTROL_NONE;
[…]
```

The Instance field (here USART3) points to the base address of the periphery register set. Register sets are organized the same way for the same types of peripherals (the base addresses are different). The handle (huart) contains all information, sometimes even function pointers that are needed to use the periphery. Everything neatly at the same place.

HAL_Status return values

HAL library uses the type *HAL_StatusTypeDef* as return values, which is an enum (C enumeration type) with different values: OK, ERROR, BUSY and TIMEOUT. It is suggested that these return values checked, as if there is some sort of error, it can be handled easiest, not just assuming that everything is ready and everything works.

HAL constants for initialization

Several settings in HAL structures have predefined macros or constants. The naming scheme always goes from general to specific, separated by underscores: HAL_I2C_[…]. Press Ctrl+Space in the IDE to see the list of know symbols. Symbols also allow the code to be much more readable.

Always use the constant symbols instead of literal values! This makes the code much more readable. Also if a setting represents the value "1", it does not mean the value 1 is used to describe such value (think of 1 stop bit setting for UART: value 0 describes stop bit to be "1".)

Predefined consts are always given in the following format: <periphery name>_<setting>_<value>, so it easy to find. Start typing: "UART_S", press Ctrl+Space to see the opcions.

TODO-s in the IDE

The “Tasks” window lets the developer easily access the parts of the code that contains TODO comments. It is advised to use this feature during the laboratory exercise as all the parts that need implementation are labeled by “TODO 1st UART task” comments.

Function descriptions

Eclipse-based IDE-s provide help to access code documentation. This laboratory guide does not contain detailed description of all the HAL functions, however the HAL code contains documentation. Move the mouse cursor over a function and the IDE will show the comment that is just before the function implementation. Use Ctrl+Space to access code completion. Use F3 to jump to the definition or the implementation of a function or a type.

Debugging: run, pause, call stack

The STM32 microcontroller provide advanced debugging features, such as breakpoints can be set, variable values can memory can be read out (by hovering the mouse cursor over a variable) or add watches. Code can be executed step by step (step into (F5) or over (F6) a function call), continue (F8), step return etc. Breakpoints can be set by double clicking on the left of the line.

Suspend can be used to stop (pause) the execution and see where the execution actually was. Whenever the execution is suspended, the call stack shows which functions were called and how the execution got there. In case the software is stuck in an infinite loop (waiting for a flag that never sets), it is easy to find where and how the code got stuck.

Keyboard and hotkeys

The keyboard language is set to Hungarian by default but English is also usable. Some keys and hotkeys do not work well in the IDE, in case modify them: Window->Preferences->General->Keys. Type “Ctrl+Alt+” in the filter text box. Test some of the keys used in C language: Ctrl+Alt+F for “[“, G for “]”, C for “&”, B for “{“, N for “}” etc. If hotkeys are associated, press unbind. Don’t forget to press Apply or Apply and Close when you finished.

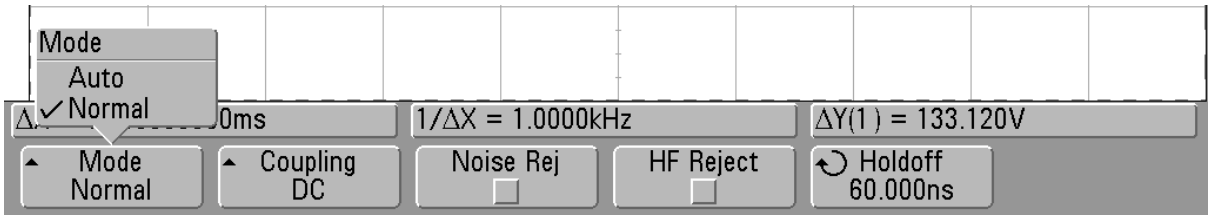
Logging, loglevel

Many systems provide runtime information for tracing behaviors, these logs can be saved or transmitted for later assessment. Based on how much runtime information is needed and how critical information is, log levels are introduced (info, warning, error, critical, fatal etc.). In the exercise example two levels are used. LOGLEVEL_NORMAL és LOGLEVEL_DETAILED. DETAILED level logs are shown only if the user requests them, normal log messages are always shown.

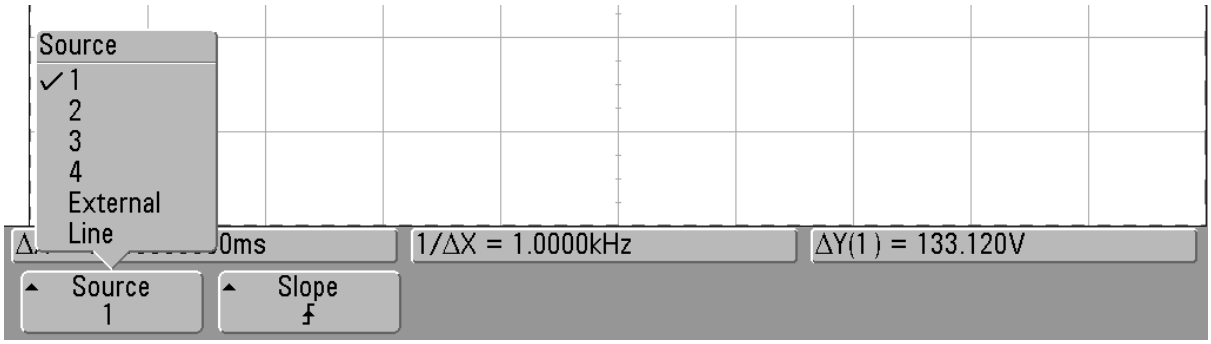
The oscilloscope: mode, trigger, cursor

During the laboratory exercise mostly the Mode/Coupling, Edge and Cursors menus are needed.

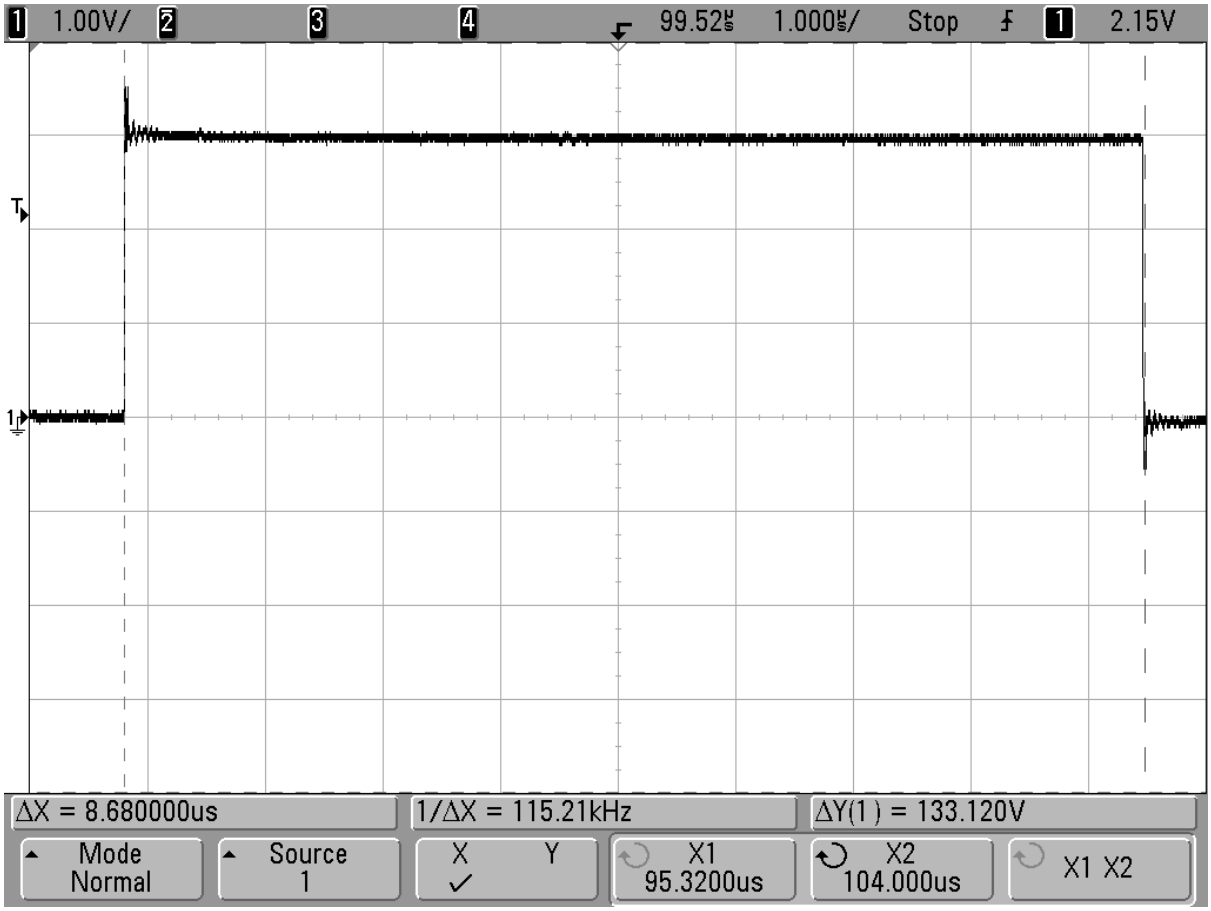
Mode/Coupling contains two trigger modes: auto creates virtual trigger events even if there was no real trigger event, in order to still show the signal. Normal only triggers if the trigger condition is set.



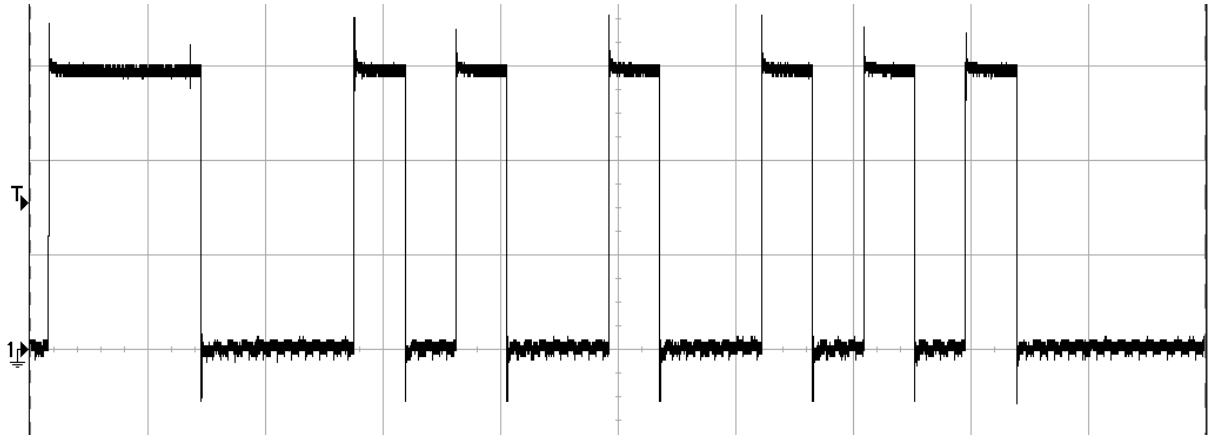
Use normal mode to “catch” UART frames. Set the trigger channel and slope according to the protocol standards. Auto mode automatically retriggers to update the screen frequently. This is useful if we are still “looking for” the signal and don’t yet know what to set the trigger for. Auto mode is not useful for catching UART frames.



Cursors menu is used to turn on horizontal (X) and vertical (Y) cursors pairs. Move each cursor to measure time/frequency or voltages.
In order to measure baud rate, move X1 and X2 cursors to the beginning and end of one bit. See the measured 115.21kHz as the reciprocal of the bit time (115200bps baud rate).



Run/Stop and Single buttons are also useful. Run turns on continuous refresh and triggering. Pressing it again freezes the sampling. In order to catch one UART frame, set up the trigger conditions and press Single. This will wait for the first trigger condition, sample a screen and stops sampling. This is useful for sampling one UART message as these bytes are not transmitted very often.



Preparation for the exercise

The following list contains preparation steps in order to successfully complete the laboratory exercise. It is important to come prepared as there is no time to get to know the template project, refresh C programming skills, etc.

Overview of the tasks, understand the template project

Before coming to the laboratory session, read the guide again, go through the template project, understand why and how the modules communicate. Read the comments in the code.

- Refresh your knowledge of UART, I2C, SPI and CAN if needed, you should know what each bit represents of an uart frame.
- Download the template and the IDE, compile the project and understand how it works. Follow how each function is called.

Refresh your C programming knowledge

There is no time to learn C during this exercise and C proficiency is assumed from previous studies.

- Basic debug functions (breakpoint, run, step into, step over, step out, call stack, watch window).
- Pointer usage: declaration, & operator, dereferencing, pointer arithmetics (incrementing, subtraction, addition)
- Memory allocation, string handling, stack vs heap, how to assign value to strings, how to iterate over string characters, what sprint is, how to use format strings. Syntax of switch-case, what extern means, how header files work.
- Importing a project into workspace or setting the hotkeys should not be a problem.

C_6: runtime error, text pointer doesn't point to anywhere and segment fault

C_7: assert function check a parameter, return if true and send error message if false and terminate. using call stack window

C_1: This shows through which functions we got to where we are right now. For example, you can find out exactly where the function we are in now called from. You can also jump there by double-clicking on its name.

Questions

There will be a short test at the beginning of the exercise, regarding the following topics (not necessary the same questions):

C programming, IDE

- What is the call stack window for? How can you use it to figure out what function call series led to the current PC position?
- What is the extern keyword used for for a global variable?
- What is the difference between & and &&? Show examples for both!
- What is the difference between “pData[3]” és a “*(pData+3)”?
- Implement a short function that calls “void send(const char c)” function for all the characters of the string passed as an argument.
- What is wrong with the following statements that tries to write the string “Hello” to text?

  ```
  char *text;
  strcpy(text, “Hello”);
  ```

- Your program runs to and endless loop. You suspend the execution and see it is while(1), the function names seems to relate to and assert function. How do you figure out what led to this state?
- What happens in the following code snippet? (How would you fix it to make sense?…)

  ```
  int first = 5;
  int second = 7;
  int something = 6;
  if (something== (first|| second)) printf(”Is it?”);
  ```

C_2: An external reference is in another file as a global variable and we want to use it.

C_3:
& the bit operation. Apply the AND operation bit by bit. 1010 & 1100 = 1000
&& is a logical operation. AND relationship between 2 boolean variables. Return 1 if both are non-zero and zero otherwise. 4 && 8 = 1

C_8: if (something == first || something == second)
(a||b) return 1

C_5:

```
char* ptr = string;
while (*ptr != "\0") {
    send(*ptr);
    ptr+=1;
}
```

I2C communication

- What features does I2C have? Point to point/bus, one or two direction, half/full duplex, serial/ parallel, synchronous/asynchronous, speed etc.?
- What is the SCL line?
- What is the SDA line?
- What is the role of a master or a slave device?
- What events happen during an I2C transaction? What denotes the beginning, the end of the transaction, etc?
- What is abbreviated as I2C?
- What is the ACK bit and which party drives it?
- What steps are used to read out a temperature value from an I2C temperature sensor?

SPI communication

- How many lines are needed to communicate on SPI? What are the names of the lines?
- Give the features of the SPI communication!
- What is MISO and MOSI?
- How are devices selected?
- Compare SPI to I2C!
- Why can the same function implement SPI reading and writing easily?

UART communication

- Name the parts of an UART frame!
- What is the TX and RX line?
- Give features of the UART protocol!

UART_1: Start bit
Data bits
It can also be a parity bit (not here)
Stop bit

UART_2: TX data transmission
RX data receiption

UART_3: duplex (half/full) 2 devices communicate with each other one at the time.
simplex 1 device send data, the other one receive

UART_5: Both the device sending the data and the receiver must send the data at a predetermined frequency the device to sample the incoming data at specified intervals. Before 2 devices to communicate the basic rules for the exchange of information between them need to be clarified. (Baud rate)

- What is the most important difference of UART vs I2C or SPI?
- How is the sender and the receiver synchronized?
- What is the maximum clock error between the sender and the receiver to ensure error-free transmission?

UART_4: uart is out of sync, no clock is sent, this allows the system to know where it is clocked

UART_6: Synchronization between transmitter and receiver requires very precise timing, so the same communication speed must be used on both sides of the bus. uart has 10 bits long so max 2 bit or 5 %

CAN

- Is there a master and multiple slaves on the bus?
- Are device addresses used to identify the sender and the recipients?
- What identifies the priority of the messages?
- Name a few parts of a CAN frame!

Frequent mistakes and solutions

- Wired communication needs wires to be connected. Check the cabling (RS-232, CAN). RS-232 and CAN both use DSUB-9 connectors, try not to exchange them by mistake.
- "&" character does not work in the IDE. Go to Window-Preferences-General-Keys, look for Ctrl-Alt-C in the bindings and unbind the command.
- A variable has a funny value. Always initialize values. Especially pointers! Pointers must point to allocated memory space!
- If the microcontroller is communicating with an other chip and something goes wrong, resetting the processor might not solve the issue as the other device may stuck in an unwanted state. Reset the other IC as well. First remove the USB cable, then the power, plugin in the power and the USB cable in this order.

I2C_1: The I2C two-wire synchronous data transmission system, developed to connect integrated circuits, consists of two wires: SCL (clock) and SDA (data).
Serial, 8-bit, two-way data traffic with a maximum speed of 100 kbit / s in standard mode and 400 kbit / s in fast mode.
The data is bidirectional, but only on one wire: half-duplex.

I2C_4: There is a simple master / slave connection between the communication devices. The master always initiates and controls the communication (the master provides the clock signal) and the master can act as both a transmitter and a receiver.

I2C_6: Inter-Integrated Circuit = between integrated circuits

I2C_7: The ACK bit allows the receiver to communicate to the transmitter that the byte was successfully received and another byte may be sent. Before the receiver can send an ACK, the transmitter must release the SDA line.

I2C_8: If the initialization is present, the data must be read from the register.
HAL_I2C_Mem_Read (& hi2c, deviceAddress, registerAddress, 1, pData, dataSize, i2c_timeout);
Read the "dataSize" byte from the "registerAddress" register of the "deviceAddress" device to the buffer shown by "pData"

SPI_1: 3 + x wire, where x is the number of devices. Synchronous serial system, three signal connections: DI (data input), DO (data output) and CLK (clock); and an enabling wire for each device.

SPI_2:
The SPI (Serial Peripheral Interface) bus is a two-way (duplex) synchronous serial implements communication between two devices.
Simultaneous two-way communication on two data lines, full duplex.

SPI_3:
MOSI (Master output, Slave input)
MISO (Master input, Slave output)

SPI_4: The Chip Select enable sign.

SPI_5:
- I2C is a protocol that can handle multiple master and multiple slave units, while SPI has a privileged tool in communication (master).
- The main advantage of the I2C bus is address-based communication. A device connected to each bus has a 7 (or 10) bit address that can be identified on the network, i.e. with the SPI in contrast, there is no need to connect a separate authorization cable to each device.
SPI: full duplex, faster but more wires (CS for everyone)
I2C: half duplex, slower but requires few wires

CAN_1: No, Unlike other protocols, such as I2C and SPI, CAN nodes do not have strict master/slave roles. Instead, each CAN node may operate as a transmitter or receiver at any time.

CAN_2: NO, each can controller will pick up a traffic on a bus, and using a combination of hardware filters and software, determining if the message in interesting or not.

CAN_3: An 11-bit ID field (message ID) of a message indicates the priority, the lower the number, the higher the priority.

CAN_4:
o Data frame
o Data request message (Remote frame)
o Error frame
o Overload frame message