Prepare your Environment

Create an ASDD directory (e.g. ASDD_Module), where all ASDD related materials (workspaces, projects, documents, APIs and so on) will be stored. Because the ASDD environment is not available until now, use the GIT eclipse installation for the Lab01 and prepare an intermediate environment as follows:

- Create directories
  - for the workspace within the ASDD directory (e.g. ASDD_Module\workspace).
  - for the IOWarrior API within the ASDD directory (e.g. ASDD_Module\IOWarriorAPI).
- Start eclipse and set the workspace to the newly created workspace directory (e.g. ASDD_Module\workspace).
- Create a new project (C++ Managed build project -> GIT project or – if you have another installation – C++ hello world project with MinGW GCC toolchain). The project (e.g. Lab01_prep) shall be stored into the newly created workspace, so *Use default location* must be checked. Remove the generated cpp file from the source code folder.
- Download the Moodle content from *Lab01* and store the files:
  - Unzip *IOWarriorAPI.zip* into the directory for the IOWarrior API within the ASDD directory (e.g. ASDD_Module\IOWarriorAPI).
  - Unzip source code files (*Lab01_start_code.zip*) into the new projects source code folder. If they do not appear in the eclipse project explorer, klick on the project and press F5.
  - *IOWarriorStartDoc.zip*: You may unzip/copy the *doc* folder into the new projects directory (e.g. ASDD_Module\workspace\Lab01_prep\doc).
- Open the properties of your newly created project and make the following changes:
  - C/C++ build -> Settings -> GCC C++ Compiler -> includes
    - include paths: path of the IOWarrior API (e.g. ASDD_Module\IOWarriorAPI)
  - C/C++ build -> Settings -> MinGW C++ Linker -> Libraries
    - Libraries: *iowkit*
    - Library search paths: path of the IOWarrior API (e.g. ASDD_Module\IOWarriorAPI)
- Build the project. If this does not work, check the configuration from the last step.
- Run the program. Running fails, because iowkit.dll is not found. A quick and dirty way to solve the problem (in the ASDD environment it will be solved more generally):
  - copy *iowkit.dll* into the directory that contains the exe file (e.g. Lab01_prep.exe is stored in ASDD_Module\workspace\Lab01_prep\debug).

Because the IOWarrior is not connected, the program should show all information on the eclipse console.

Prepare the Lab01 – IOWarrior button control and amplitude meter

The code, you write to solve the following tasks, shall be built without any syntax or build error. Save your source code files on an USB stick and bring it to the lab.

1. Add a new method for the button input to CIOWarrior. It looks, if the test button on the IOWarrior evaluation board is currently pressed. If not, it immediately returns *false*. Otherwise, it waits for the button to be released by the user and returns *true*.

Hints:
The button is connected to Port 0 Pin 0 (P0.0). The internal pullup resistor of an IOWarrior port pin is activated by writing 1 to the pin (see *IOWarriorDataSheet.pdf* chapter 5 for more information). This is done for all pins of all ports by the constructor of *CIOWarrior*. Once the pullup is activated, the pin may be used as an input pin. Therefore, the pressed button causes P0.0 to be low. Otherwise, P0.0 is high.

Because we want to use the method to stop and resume audio playback, the method must not block the program (e.g. getch() is a blocking function that waits for the user to enter a key and does not return until the user enters a key). Choose the right API function to fulfill this requirement (see API documentation in *IowKit_V15_API.pdf*).

2. The current implementation of the ampere meter class *CAmpMeter* is given in the source code files *CAmpMeter.h* and *CAmpMeter.cpp*.
   a. Comment the code of the methods that are already implemented in *CAmpMeter.cpp*.
   b. Complete the implementation of *CAmpMeter*. Pay attention to the comments given in the source code files.
   c. Write another test function in *main.cpp* to test your implementation (for screen visualization only) – e.g. visualize the values of the running light on the screen.

CAmpMeter
-m_scmode:SCALING_MODES=SCALING_MODE_LIN -m_scMax:float=0. -m_thresholds:float[8]=0. -m_iowdev:CIOWComm*=NULL
+CAmpMeter() +init(min:float, max:float, scmod:SCALING_MODES, logScaleMin:int, iowdev:CIOWComm*):void +write(databuf:float*, databufsize:unsigned long):bool +write(data:float):bool +print(databuf:float*, databufsize:unsigned long):void +print(data:float):void -_getBarPattern(float data):unsigned char -_getValueFromBuffer(float* databuf, databufsize:unsigned long):float

Hints:
Linear scaling: The Amplitude meter shows the absolute data values (amplitudes) between 0 and the scale maximum (*m_scMax*), which is calculated according to the range of the data values to be visualized (see comments in *CAmpMeter.h*).

Logarithmic scaling: The Amplitude meter shows the data values in dB between a user-defined minimum and 0 dB. The minimum is a negative dB value that is passed to the *init*-method (see comments in *CAmpMeter.h*). Before calculating the dB value for visualization, the (linear) data value shall be divided by the linear scale maximum (*m_scMax*) to adjust the highest value to 0 dB (this is called peak normalization).

All methods have to make sure that they work with valid parameter values and that the object is in the right state to fulfill the task the method has to do. Implement an appropriate error handling, which returns immediately if the method cannot handle the problem on its own.

Do not duplicate code – whenever it is possible, call a method that does the job (e.g. call _getBarPattern(…) whenever you need a bar pattern).