Developer Manual
Blood Pressure Measuring Device With Stepper Motor Driven Piston Pump

Pascal Schüttengruber

March 2024

Contents

1 Introduction

This is a manual for the further development of the software from Pascal Schüttengruber's master thesis. Its a C# windows forms application that utilizes the USB-AD14f board by BMC and the hardware constructed in the thesis. To get the code ask Stefan Ropele (stefan.ropele@meduni-graz.at) or maybe it is available at GitHub (this will be determined after this document was created). For questions regarding the code contact Pascal Schüttengruber (p.schuettengruber@outlook.com).

The content is split into support classes and windows from classes. Support classes provide only function and windows form classes also provide a UI. This guide will not go into the details of UI development with visual studio, it will only tell about things that cannot be seen or clicked when in the visual builder. This manual assumes one has the code and UI builder opened.

2 Support Classes

2.1 BmcmInterface.cs

This class is used to better interface the BMC board. The interface that comes by the company it not very handy. For instance if one wants to set one digital port high there is a function "ad_discrete_out(...)" where one parameter represents how the whole state of the port should look like. If the value 4 is passed it sets pin 3 high because 0x04 corresponds to 0b00000100.

The BmcmInterface.cs class handles this by providing methods like "set_digital_output_high(int nr)". With this method one select one pin individually to be set high by only passing the pin number and the class handles the rest.

The class saves the state of the digital ports in the boolean arrays *digitalOutputs* and *digitalInputs*. The calculation between boolean array and uint is carried out by the methods "get_int_from_bool_array" and "get_bool_array_from_uint". The class also provides the utility to conduct a continuous reading of one ADC port, the so called "scans".

2.1.1 Methods

```
public BmcmInterface(string name)
```

The connection to the BMC board is opened and saved in the integer variable *adh*.

- **string name**
  Name of the BMC board, for instance "usbad14f".

```
~BmcmInterface()
```

This is a destructor, which is called when the garbage collector deletes its instance. It stops an ongoing scan and closes the connection.

```
public void set_digital_output_high(int nr)
```

Sets the desired pin of the digital output port high and saves the new state of the port in the variable *digitalOutputs*.

- **int nr**
  Number of the pin that should be set high. Value between 1 and 8.

```
public void set_digital_output_low(int nr)
```

Sets the desired pin of the digital output port low and saves the new state of the port in the variable *digitalOutputs*.

- **int nr**
  Number of the pin that should be set high. Value between 1 and 8.

```
public void set_analog_output(float value)
```

Sets the analog output pin to the desired voltage.

- **float value**
  Value the analog output should be set to. Value between 0 and 5.12.

```
public float get_analog_input(int nr)
```

Reads the analog voltage of the desired pin and returns the read value.

- **int nr**
  Number of the analog input pin that should be read. Number between 1 and 16.
- **returns float**
  Voltage of the desired pin.

```
public bool get_digital_input(int nr)
```

Reads the desired pin of the digital input port and returns its state.

- **int nr**
  Number of the digital input pin that should be read. Number between 1 and 8.
- **returns bool**
  State of the pin, either true or false.

```
public uint get_int_from_bool_array(bool[] boolArray)
```

Function that takes a bool array and calculates the representing uint number. Needed because an integer is required to control the IO channel. Always the whole IO port is read or set. For ease of access the state of the port is saved as bool array.

- **bool[] boolArray**
  Array of 8 booleans that represent a port.

- **returns uint**
  Integer representation of the bool array.

```
public bool[] get_bool_array_from_uint(uint value)
```

Function that takes an uint value and calculates the bool array representation. Needed because an integer is required to control the IO channel. Always the whole IO port is read or set. For ease of access the state of the port is saved as bool array.

- **uint value**
  Unsigned integer value that represents the state of a port.
- **returns bool[]**
  Boolean array representation of the integer value.

```
public void stop_scan()
```

Stops an ongoing scan.

```
public void start_scan(float sample_rate, int values_per_scan, int
    values_per_run, int how_many_runs)
```

Starts a scan.

- **float sample_rate**
  The rate in in which the port gets read (values/s). Number to max. 20 kHz. Recommended for this application is 500.
- **int values_per_scan**
  How many scans should be made as a whole. Take a very large number to run very long. Pretty much irrelevant, just equal or larger than how_many_runs. Eg. 2000.
- **int values_per_run**
  How many values should be read in one reading, e.a. by calling the method "get_values". Recommended number is 250, which means the values have to be read twice a second.
- **int how_many_runs**
  How many runs should done before the process gets restarted, such that it scans for ever. A number equal or lower than values_per_scan. Eg. 2000.

```
public float[] get_values(uint run_id)
```

Fetches the recorded values of a run during the scan. How many values it returns is set by the values_per_run parameter in the start_scan method.

- **uint run_id**
  Number that represents the run. Used as counter to know which data needs to be fetched and when to restart the scan. Start with 0 and increment by 1 each time the values are read.
- **returns float[]**
  An array that contains the measured values with the length defined by values_per_run in the start_scan method.

2.1.2 Suggested Improvements

- It would be possible to set more than one pin with a function like "set_digital_output_high()" by providing methods that can handle overloading of the argument or take an array as input.
- Scans are currently only possible at analog input 1! It was not necessary to support more channels in the thesis. Here one has to look at the LIBAD4 documentation what may be suitable for your application and re-implement to whole scan section.

2.2 AppConfig.cs

This class holds all the configuration and information that should be available throughout the program. It gets instanced right at the start of the program and all form classes used need this variable in their constructor. It reads the confi.csv file if available, otherwise it creates this file with standard values. Every value read is saved in one dictionary named *param*. The class gets instanced with:

```
AppConfig config = new AppConfig();
```

So the parameters are called like:

```
string NAME = config.param["NAME"]
```

Where NAME is something like sample_rate.

2.2.1 Methods

```
public AppConfig()
```

It trys to load the config.csv file. If it is not found the dictionary param gets filled manually and the config.csv file gets created. Also the pressure sensor gets initialized.

```
public void saveConfig()
```

Saves the dictionary param in a csv file.

```
public void loadConfig()
```

Loads the parameters from the config.csv file and saves it in the dictionary param.

```
public void InitPressureSensor()
```

Calculates the parameters of the linear curve calibration of the pressure sensor.

```
public float VoltageToMmHg(float voltage)
```

Calculates the mmHg value of a given voltage, by the before initialized linear curve of the pressure sensor.

- **float voltage**
  Voltage value to be converted into a mmHg value.
- **returns float**
  The corresponding mmHg value.

```
public float[] VoltageToMmHg(float[] voltage)
```

Same method as before, just for arrays.

- **float[] voltage**
  Voltage values to be converted into a mmHg values.
- **returns float[]**
  The corresponding mmHg values.

2.3 libad.cs

Provided class by BMC. Here is a link to their website, where programming examples are shown and the manual and source code can be found. It is never used directly, only via the $BmcmInterface$ class.

2.4 MovingAverage.cs

A class that provides a moving average filter. Make an instance like this:

```
MovingAverage mov_avg = new MovingAverage();
```

A new value is added by:

```
mov_avg.ComputeAverage(SAMPLE);
```

To retrieve the freshly calculated value:

```
float value = mov_avg.Average;
```

2.5 FilterButterworth.cs

Provides a Butterworth IIR filter of order 2. Highpass and lowpass are possible.

2.5.1 Methods

```
public FilterButterworth(float frequency, int sampleRate, PassType passType,
    float resonance)
```

Creates an instance of a Butterworth filter. Initializes the parameters of the filter.

- **float frequency**
  Cufoff frequency.
- **int sampleRate**
  Sample rate used by your signal
- **PassType passType**
  Either highpass or howpass like: FilterButterworth.PassType.Highpass
- **float resonance**
  Rez amount, from sqrt(2) to 0.1

```
public void Update(float newInput)
```

Calculates a new output of the filter.

- **float newInput**
  Input value that should be filtered.

Fetch the new output by calling something like:

```
float newOutput = filter.Value;
```

3 Windows Form Classes

The forms are designed with Visual Studio 2022 - Community Edition.

3.1 MainUI.cs

The whole application runs in this class. All other Form classes are opened as children, by loading the form in a *Panel* called panelChildForm. It has to be said that everything is in a *tableLayoutPanel* named tableLayoutPanel1 to provide scalability. It has two columns, where the left is fixed and the right scalable. The rest should be visible. Be sure to alter with care. It is not recommended to rescale anything, because every form has the right size to fit in the size of the childPanelForm.

3.1.1 Methods

```
public MainUI()
```

Instances the *AppConfig* class, checks if developer mode is activated and loads the help.rtf file.

```
private void openChildForm(Form childForm)
```

Closes the old form that was open in panelChildForm and displays a new one.

- **Form childForm**
  Input form to be displayed in the UI.

```
private void btnMeasure_Click(object sender, EventArgs e)
private void btnMeasure_MouseEnter(object sender, EventArgs e)
```

All buttons have click events that call the openChildForm method with their associated form. All button additionally have a hover effect because the default effect is too weak. The help button just closes the active form such that the help is visible again.

```
private void linkLabel1_LinkClicked(object sender,
    LinkLabelLinkClickedEventArgs e)
```

Opens the default mail program with a form to send an email with the developer as recipient.

```
private void checkDeveloper_CheckedChanged(object sender, EventArgs e)
```

Activates and deactivates the developer mode. The visibility of the developer button and the config changes regarding to the state.

3.2 Developer.cs

This form can only be opened when the developer checkbox is ticked. It can be quite useful for the developer because with that one can control every port of the BMC board exclusively. Again everything is tied to a tableLayoutPanel for scalability.

3.2.1 Methods

```
public Developer(AppConfig config)
```

Saves the config and opens a connection to the BMC board. Preemptively the BMC board gets set to a default state.

- **AppConfig config**
  Configuration of the app.

```
~Developer()
```

Destructor (gets called when the garbage collector deletes the instance). Disposes a timer that migth have been used.

```
private void toggleDigitalOutput(Label lbl, int nr)
```

Toggles a given digital output pin and displays its state on the given label.

- **Label lbl**
  Label that sould be altered.
- **int nr**
  Number of the pin. Values from 1 to 8.

```
private void btnDigitalOut1_Click(object sender, EventArgs e)
```

Gets calles when the DigitalOut1 button is clicked. Calles the toggleDigitalOutput method with the right parameters.

- Exists for every digital out button and does the same.

```
private void btnStartADC_Click(object sender, EventArgs e)
```

Displays the voltage on the analog input pin 1 in the UI.

```
private void btnDigitalInput1_Click(object sender, EventArgs e)
```

Displayes the state of the digital input pin 1 in the UI

- Exists for every digital input button and does the same.

```
private void btnNewDACValue_Click(object sender, EventArgs e)
```

Reads the value of the numDACValue UI element and sets the analog output.

```
private void btnStartScan_Click(object sender, EventArgs e)
```

Starts the scan, initializes the timer and calls the corresponding timer function "RepeatForEver".

```
private void btnStopScan_Click(object sender, EventArgs e)
```

Stops the currently running scan, disposes the timer and saves all collected data.

```
async void RepeatForEver()
```

When this method is called, the timer "timer" was started before. When "timer" elapses, the code in the while loop gets executed. It reads n values (eg. 1000) and prints just the first one. The values are getting stored in data.

3.3 Calibration.cs

On this form tasks should be made that don't depend on an individual measurement or patient. These tasks are calibrating the pressure sensor and the speed of the system as well as conducting a leaf-proof-test and saving these settings.

3.3.1 Methods

```
public Calibration(AppConfig config)
```

Saves the config and loads its entries in the corresponding UI element. It also opens the connection to the BMC board and sets it to the default state. Two BackgroundWorker are instanced here. These are needed to start a new thread for the speed calibration and the leak-proof-test.

- **AppConfig config**
  Configuration of the app.

```
private void btnLeakproofTestStart_Click(object sender, EventArgs e)
```

Starts the leak-proof-test by initializing the progress bar and opening a dedicated thread via the BackgroundWorker class. The DoWork method of the BackgroundWorker class starts the thread, with ProgressChanged one reports progress back to the main thread and RunWorkerCompleted gets called when the thread ended. These three methods are basically event handlers.

```
private string leakProofTest()
```

This is the method that runs on a second thread. It pumps initial air in the system with the membrane pump, stops it after a while and snapshots one voltage value. It then waits 5s, collects a second value and compares it.

- **returns string**
  Result of the leak-proof-test. Either "Result: Good!" or
  "Result: NOT LEAKPROOF!"

```
private void btnCalibrateSpeed_Click(object sender, EventArgs e)
```

Starts the speed calibration by initializing the progress bar and opening a dedicated thread via the BackgroundWorker class. The BackgroundWorker class has the same three events. This time only the progress bar gets manged and no value is returned.

```
private void calibrateSpeed()
```

This is the method that runs on a second thread. It starts the piston pump and a scan that reads 500 values every second. For reading the values the PeriodicTimer class is used. The property of the PeriodicTimer is that it runs on a different thread and it does not run out of sync, because its tracks when its code got executed and tries to reach the mean of its time given.

```
async void RepeatForEver()
```

Method that the PeriodicTimer that was initialized in the calibrateSpeed method uses. The pump steadily increases the pressure and every exactly one second the timer elapses. So every second the pressure values are read, averaged and saved. Always the last two values are compared. The difference has to be equal the desired speed plus-minus the given tolerance that the speed counts as calibrated. If this condition is not met the pump speed gets altered - faster if it was too slow or lower if it was to fast.

Progress is always reported and if the calibration cannot finish it will be stopped and a negative result will be reported by a message box. If the calibration is found the algorithm will also immediately stop and report a positive result via a message box.

```
private void numSampleRate_ValueChanged(object sender, EventArgs e)
private void numVoltLowEnd_ValueChanged(object sender, EventArgs e)
private void numVoltHighEnd_ValueChanged(object sender, EventArgs e)
private void numHgLowEnd_ValueChanged(object sender, EventArgs e)
private void numHgHighEnd_ValueChanged(object sender, EventArgs e)
private void numSpeedInflation_ValueChanged(object sender, EventArgs e)
private void numTolerance_ValueChanged(object sender, EventArgs e)
private void numStepSize_ValueChanged(object sender, EventArgs e)
```

These methods get called when their respective number selector get changed. They save their parameter in the config.param dictionary.

```
private void btnSaveSettings_Click(object sender, EventArgs e)
```

Saves all configurations that habe been made in the config.csv file. Configurations that are made and not explicitly saved are lost!

```
private void btnADC_Click(object sender, EventArgs e)
```

Reads the value of the analog input pin 1 and displays its value in the UI.

3.4 Measure.cs

This is the form where measurement and patient specific configurations are made and the actual measurement is conducted. Specific configurations are the start pressure and the start position waiting time (this defines the start position of the piston, it should be higher if the system is leaky). Also the name and the location of the log file where the data is saved.

The radio buttons "Normal Mode" and "Dynamic Mode" define what parameters the measurement process gathers. In dynamic mode only the MABP is evaluated, in normal mode also systolic and diastolic blood pressure are detected.

The **simulation checkbox** is only active when the developer mode is activated.

When simulation is ticked, pre-recorded data is loaded in the algorithm and no data is collected. This is very handy to test the algorithm.

3.4.1 Structs

```
public readonly struct Measurement
```

Is used to store the values of each run with the corresponding time and direction.

```
public struct Peaks
```

Is used to store the peaks such that one can interpolate afterwards.

3.4.2 Methods

```
public Measure(AppConfig config)
```

Saves the config and loads its entries in the corresponding UI element. Opens a connection to the BMC board and initializes all variables and filters.

```
private void rdoNormalMode_CheckedChanged(object sender, EventArgs e)
```

Not used.

```
private void rdoDynamicMode_CheckedChanged(object sender, EventArgs e)
```

Changes the visibility of the labels that are not used when this mode is active.

```
private void btnSaveLog_Click(object sender, EventArgs e)
```

Opens a SaveFileDialog and then declares the directory where the data should be saved. Also saves preemtively all parameters in the config.csv file.

```
private void btnStart_Click(object sender, EventArgs e)
```

Starts the measurement procedure. Makes the "Stop" and "Trigger" buttons visible. Starts a timer with a 50 ms intervall.

```
async void StartTimer()
```

Skipped when in simulation mode. Brings the piston in the right start position and then activates the membrane pump. When the start pressure is reached, the actual measurement is started.

First the valves are brought in the right state and stepper mode is set to the speed maximum. Then the motor gets activated, going in the inflating direction. Every 50 ms it is checked if the piston presses the limiting-switch. When its pressed the direction of the motor gets changed and the motor gets reactivated.

Then the piston moves in the other direction until the "start position waiting time" has passed. Then the system is at the right position to start. The motor stops and the membrane pump gets activated. It pumps until the "start pressure" is reached. The timer gets disposed, the piston pump deflates the system and the measurement begins.

```
async void RepeatForEver()
```

The method that constantly fetches the data from the BMC board. If simulation is activated it reads it from a file. The method saves the data in a list of the measurement struct - this is the one that gets saved in the end. The "data_work" list is used to compare afterwards - it is easier and faster to search in this list than in a list of structs. "data_period" is the data that is used in "DataAnalysis()".

```
private void btnStop_Click(object sender, EventArgs e)
```

Stops the ongoing measurement. Saves the data, disposes the timer, stops the motor and opens the emergency valve.

```
private void DataAnalysis()
```

Highpass filtering and peak detection part.

Stops the measurement procedure here when a too low pressure is recognized. Too low pressure typically means that no blood pressure parameter is found.

First the highpass filter is applied. The transient response has to be waited for, before the peak detection can start. Minima and maxima are detected. When a minimum is found, it will be added to the next maximum and is saved in the list of structs "Peaks". Everytime a peak has been found the method "linearPeakInterpolation()" is called.

How the peak detection works: It always looks at a window of the data. When after a full window length no further minimum is found the first entry of the list is declared as the minimum. Then a maximum will be searched for with the same method. When a maximum is found it searches for a minimum again and so on.

```
private void linearPeakInterpolation()
```

Heart rate calculation, interpolation with simultaneous envelop calculation and peak

detection of said envelop part.

The first peak is just saved and a time correction of the filter delay is made. From the second peak onwards the heart rate calculation can be conducted. The time difference of 7 peaks are calculated and the average is built. The result is converted to match pulses per minute and displayed in the UI.

Also from the second peak onwards the linear interpolation between the peaks is conducted. The resulting curve is lowpass filtered to smoothen the curve (getting rid of outliers). Then another peak detection of the resulting curve is made. The maximum of this curve corresponds to the MABP. Minima are not regarded this time. It can happen, that not all maxima found are true maximas. This is better handled in the analysis section. Time should be saved hence this is not perfect here (The window length should be longer).

When the MABP is found then one other BP parameter can be found right away depending on the direction. This parameter will only be calculated in normal mode. Otherwise it will be skipped and the next MABP value gets searched, by changing the direction. In normal mode the "foundBP()" gets called when a peak was found as long as the other BP parameter is found.

```
private int nearestPeak(int position)
```

If a significant point (maximum, systole or diastole) in the envelop is detected, the corresponding peak has to be found. This function finds the corresponding peak by finding the x value of the lowest distance.

- **int position** X value of the significant position found.
- **return int** X value of the nearest peak from the significant point.

```
private void findBP()
```

In normal mode, when MABP is detected, one BP parameter cannot be detected yet, depending on inflation or deflation cycle. It has to be tried every time, if the parameter can be found. If its found the direction of the motor is changed and the algorithm start from a new.

```
private void changeDir()
```

Changes the direction if the motor, which means changing from inflation to deflation and vice versa.

```
private void btnTrigger_Click(object sender, EventArgs e)
```

Trigger means, that the data gets a timestamp, when for instance the MRI measurement started. It just sets a flag and when the next data is collected the timestamp is made.

```
private void numStartPressure_ValueChanged(object sender, EventArgs e)
private void numWaitTime_ValueChanged(object sender, EventArgs e)
```

These methods get called when their respective number selector get changed. They save their parameter in the config.param dictionary.

3.5 Analysis.cs

This form can be used after the measurement to analyse it. It is possible to load csv files and txt files. Txt files saved the actual voltage value, therefore the calibration data has to entered first via the number selectors at the bottom left. The signal can be viewed in every stage of the signal analysis chain. The calculated BP parameters are also shown. The plot class used is the ScottPlot library.

3.5.1 Methods

```
public Analysis(AppConfig config)
```

Saves the config, resets the BMC board and initializes the filters and the plot.

```
private void btnGo_Click(object sender, EventArgs e)
```

Starts the analysis process.

It follows the same algorithm that is used during the measurement, it is only implemented such that the whole data runs through each step individually and always the whole data is processed (not in junks). The only difference is that the last peak detection is made more secure, as stated in the measurement.cs part. The implementation here is not efficient but each step is better understandable and tweekable this way.

```
public int closestPeak(int index)
```

Finds the peak that is closest to the given index.

- **int index** X value of the significant position found.
- **return int** X value of the nearest peak from the significant point.

```
private float[] VoltageToMmHg(float[] voltage)
```

Calculates the mmHg value of a given range of voltages. (Same as in config but different values for k and d are used!)

```
private void checkSignal_CheckedChanged(object sender, EventArgs e)
private void checkHighpass_CheckedChanged(object sender, EventArgs e)
private void checkPeaks_CheckedChanged(object sender, EventArgs e)
private void checkEnvelop_CheckedChanged(object sender, EventArgs e)
private void checkLowpass_CheckedChanged(object sender, EventArgs e)
```

These methods display or remove the selected signal from the plot.

```
private void butXSmall_Click(object sender, EventArgs e)
private void butYSmall_Click(object sender, EventArgs e)
private void butYLarge_Click(object sender, EventArgs e)
private void butXLarge_Click(object sender, EventArgs e)
```

These methods handle the scaling of the plot. Making them either larger or smaller. The plot library used is ScottPlot.