I would like to thank my thesis supervisor, Mr. Jan Fischer, for his guidance, support and advice throughout the year. I have been extremely lucky to have a supervisor who cared about my work so much and responded to my questions very quickly.

I would also like to thank Ms. Anjana Jain, for generating my love for electronics and computer science during high school.

Finally, I would like to thank my family for their patience, encouragment and support during my studies.

Czech technical university in Prague
Faculty of electrotechnics

department of measurement

BACHELOR THESIS

Design of embedded instruments using STM32 microcontrollers

Author: Tomáš Svítil

Thesis supervisor: Doc. Ing. Jan Fischer, CSc.

Prague, 2016

Název práce: Návrh vestavných přístrojů za použití mikrokontrolérů STM32

Autor: Tomáš Svítil

e-mail: svititom@fel.cvut

Katedra (ústav): Katedra měření

Vedoucí bakalářské práce: Doc. Ing. Jan Fischer, CSc.

e-mail vedoucího: fischer@fel.cvut.cz

Abstrakt Tato práce se zabývá možnostmi implementace funkce tradičních přístrojů např. voltmetru, měřiče frekvence či osciloskopu jako vestavných přístrojů realizovaných pomocí mikrokontrolérů. Tyto vestavné přístroje byly implementovány pomocí online IDE mbed a frameworku od ST: STM32 Cube. Pro demonstraci možností impementace byly vytvořeny terminálově ovládané aplikace pro měření napětí, frekvence (přímo i nepřímo), generování napětí i PWM signálů. Pomocí mbedu byla také vytvořena samostatná měřící aplikace se zobrazením na LCD. Pomocí Cube byla vytvořena aplikace včetně PC frontendu pro 3 kanálové měření napětí, 1 kanálové nepřímé měření frekvence a střídy, jednoduché čítání pulzů, generování 2 PWM signálů a generování napětí a také osciloskopická aplikace včetně PC frontendu. Nakonec byla prozkoumána omezení takto vytvořených přístrojů. Jako nejpřesnější periferie se ukázaly časovače. Na AČP všech mikrokontrolérů byl přítomen šum, ale bylo možné jej potlačit průměrováním.

Klíčová slova: mikrokontrolér, mbed, osciloskop, vestavný přístroj, STM32Cube

Title: Design of embedded instrumets using STM32 microcontrollers

Author: Tomáš Svítil

e-mail: svititom@fel.cvut.cz

Department: Department of measurement

Supervisor: Doc. Ing. Jan Fischer, CSc.

Supervisor's e-mail address: fischer@fel.cvut.cz

Abstract This thesis explores how to implement the functions of traditional instruments such as voltmeters, frequency meters and oscilloscopes as embedded systems using microcontrollers. The instruments were implemented with mbed, an online framework and an STMicroelectronics framework: STM32 Cube. To demonstrate the implementation possibilities, applications for voltage, frequency (direct and reciprocal), and duty cycle measurement, voltage and pwm signal generation were created. All of these applications have been implemented with serial interface control. Using mbed, an autonomous measurement demo with an LCD display was created. Using cube an application with 3 channel voltage measurement, single channel recpirocal frequency measurement and duty cycle measurement, simple pulse counting, PWM signal generation and voltage generation, including a PC frontend, was created. Furthermore an oscilloscope, also with a PC frontend was created. Finally the the limitations of such instruments were explored. The timers were found to be the most accurate perihperals. The ADCs were riddled with noise, but averaging suppressed it.

Keywords: microcontroller, mbed, oscilloscope, embedded instrument, STM32Cube

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
Katedra měření

Akademický rok **2015-16**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Tomáš Svítil**
Studijní program:	**Kybernetika a robotika**
Obor:	**Senzory a přístrojová technika**
Název tématu česky:	**Návrh vestavných přístrojů za použití mikrokontrolérů ST32**
Název tématu anglicky:	**Design of Embedded Instruments Using ST32 Microcontrollers**

Pokyny pro vypracování:

Navrhněte metodu realizace vestavných přístrojů s využitím mikrořadičů řady STM32Fxxx. Tyto vestavné přístroje mají být použitelné pro funkci vnitřního testování a ovládání elektronických funkčních bloků. Bude se jednat o měření napětí a záznam jeho průběhu, měření frekvence signálu, nastavování napětí a generování signálu. Prakticky prověřte možnost realizace vestavných přístrojů s využitím mikrořadičů STM32Fxxx. Určete obvodová omezení i meze použitelnosti takto realizovaných vestavných přístrojů.

Seznam odborné literatury:

[1] Yiu, J.: The Definitive Guide to the ARM Cortex-M0, Elsevier 2011
[2] Yiu, J.: Definitive Guide to ARMR CortexR-M3 and CortexR-M4 Processors
[3] STMicroelectronics: RM0091 Rev 8 Reference manual

Vedoucí bakalářské práce:	doc. Ing. Jan Fischer, CSc.
Datum zadání bakalářské práce:	14. prosince 2015
Platnost zadání do[1]:	30. září 2017

[redacted]

Doc. Ing. Jan Holub, Ph.D.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická

L. S.

v

[redacted]

Prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 12. 2015

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER

ONE

INTRODUCTION

Traditional measurement instruments[1] have been available for a long time. Ever since, they have been greatly improved in all aspects, including precision, size, communication interfaces and others, but there are still some reasons why they are not very viable for the wide public, and especially students. The two main reasons are high price and large size.

Currently there is an abundance of microcontrollers which have complex peripherals to implement such traditional instrument functions. These microcontrollers have limitations, the largest one low accuracy and speed, however, since their benefits include very small size, and very low price, microcontrollers fit the role of being low cost measurement instruments perfectly, especially considering the fact, that multiple instruments can be implemented on a single microcontroller.

Thanks to their low price, size and plethora of peripherals, microcontrollers can be embedded into virtually any device, ranging from laboratory demos to products like power supplies. There they could be used, among others, as onboard diagnostics. The main benefit being, that if correctly done (i.e. separated from the actual device with resistors, own power supply from USB etc.), these embedded systems are independent of the device they are embedded in, and thus can provide relevant information even when the device they are embedded in is failing.

This thesis will explore the methods of implementation of traditional instrument functions with microcontrollers, and will analyze the limitations of these embedded instruments.

[1]A traditional instrument is self contained measurement instrument, which measures that it has the all required sensors and signal processing, as well as a user interface built in.

CHAPTER
TWO

CURRENT SITUATION AND GOALS

2.1. Current situation

If somebody would like to start learning about electronics today, they would have to invest a significant amount of money into measurement instruments. At the very least, they would need a multimeter, a power supply, and ideally a signal generator and an oscilloscope. A multimeter can be bought for some $20, but the rest of the instruments are much more expensive, especially the oscilloscope which starts at $300. An alternative to all these instruments, would be to buy a proprietary measurement system[1], which has all the required instruments built in, such as the NI ELVIS. Unfortunately, they are also very expensive (hundreds of dollars).

The high cost of traditional instruments poses a problem for those who want to learn electronics. This is especially true for high school students, as usually neither they, nor their high schools[2] can afford such instruments. High school students are a very important group that should be exposed to electronics. This is because high schools are a place where students decide which field to study in university, especially in the case of techincal fields, even their carrer path.

Furthermore, the large size of traditional instruments can be an inconvenience in labs where the precision and versatility of the traditional instruments is not necessary. Imagine a lab that will teach the student about a sensor , but not how to control the instruments. Depending on the sensor, one could need a signal generator, a multimeter, and an oscilloscope, and all of these instruments take up a lot of space. All of this functionality can be built into a single microcontroller and controlled from a computer, which are abundant these days.

There are many companies including ST, NXP and Texas Instrumets, who have a large variety of microcontrollers and in particulary development boards[3]. These development boards have all the necessary circuitry for the microcontroller to run and connect to measured circuitry. Thus one can simply upload a program to the board, and have the board perform the required functions such as measuring voltage, generating pwm signals, or sending the data back to the computer. The main benefits of instruments built on these evaluation boards are:

[1]A measurement system consists of one or more instruments which can communicate externally, such as over a serial connection, or USB.

[2]With the exception of technical schools, because working with the hardware is a big part of their curricullum. There is however, never enough of measurement instruments.

[3]ST has 3 series of development boards: nucleo, discovery and eval. Only the first two are interesting for this thesis, as eval boards are for the industry and as such, they have a high price.

- very low price (a STM32F303RE nucleo boards costs a little more than \$10), and
- small size (roughly the size of a smart phone).

Currently there are public projects which implement traditional instrument functions on these development boards and microcontrollers. For example Tomas Ostrowsky created an oscilloscope [13], which has many features including an arbitrary signal generator or FFT signal analysis. Unfortunately, to find these projects one has to know what he is searching for, and very often the documentation requires previous experience with electronics. Therefore getting the project up and running can take long hours of trial and error. Ultimately, many people would be much more motivated to explore the field of electronics if their first projects were easy and 'just work'[4].

2.2. Goals

This bachelors thesis will explore two ways to implement an embedded instrument, which can be used instead of the aforementioned traditional instruments. These instruments will be implemented on STM32 microcontrollers, and the limitations of such instruments will also be explored.

Signal measurement with traditional instruments can be divided into static and dynamic measurement. Static measurement deals with the parts of the signal that are constant, such as frequency or RMS voltage of a periodic harmonic signal (power outlet), thus usually a single number i.e. a 4 digit display can represent the measurement. Dynamic measurement deals with the changing parts of a signal, e.g. how the actual sine wave in the power outlet looks, or measuring the characteristics of a transient. Dynamic measurements are usually presented in the form of a graph (such as an oscilloscope or a logic analyzer).

This thesis will discuss the methods of implementation and limitations of the functions described in the following subsections on STM32F3 and STM32F0 series microcontrollers. The F3 series microcontrollers are based on the ARM Cortex-M4 core, and the F0 series are based on the ARM Cortex-M0 core. Both cores are 32-bit but they have a different architecture and the Cortex-M0 has a limited instruction set. These two series have been chosen as they represent what is available on the market today in terms of computing power, and choice of peripherals quite well.

2.2.1. Static voltage measurement:

Measuring voltage is arguably one of the most important functions of an embedded instrument, since it is the most common type of measurement. This functionality is provided by Analog to Digital Converters, or ADCs, which are builtin peripherals on most microcontrollers.

It is expected, that the embedded system will need to measure voltages in multiple ranges. Typical ranges are

- CMOS logic levels: $0\,\mathrm{V} \sim 3.3\,\mathrm{V}$, TTL logic levels: $0\,\mathrm{V} \sim 6\,\mathrm{V}$, and
- bipolar voltages for measurement on operational amplifiers (op amps): $-5\,\mathrm{V} \sim +5\,\mathrm{V}$, or even $-15\,\mathrm{V} \sim +15\,\mathrm{V}$.

[4]This was beautifully demonstrated by Arduino. It is development platform that has many examples which are well documented and easy to use, and it is now very widely used.

Most microcontrollers and their perihperals, including the ADC, are built with CMOS logic, which means that they only have a limited input range of 0 V ∼ 3.3 V or 0 V ∼ 5 V (a higher voltage could destroy them). Therefore a signal conditioning block needs to be added before the microcontroller input if one needs to work with other voltages. This block would reduces the voltage range, and in case of the bipolar voltages also shift the voltage to start at $0V$. This can be done with a resistor voltage divider, or an op amp with negative feedback.

Another limitation of the builtin ADCs is noise and interference. Even though ADCs have their own analog power and ground, they will still experience interference and noise from the rest of the microcontroller, and also from the omnipresent 50Hz sinusoid in powerlines. Fortunately though, many times this noise and interference can be eliminated by using averaging from a large enough sample set, and the 50Hz interference can be removed by averaging from sample sets that cover whole periods of the 50 Hz signal (20ms).

Most microcontrollers use successive aproximation ADCs, which require a sample and hold (S/H). As will be explained later, the current draw of these S/Hs grows with the sampling frequency. This in turn means, that at slow sampling rates, there is a low current draw that doesn't affect the measured voltages too much, at higher frequencies though, which is what an oscilloscope needs, the current draw is very high. For example, the ADC on an F303 processor, will current draw roughly $1\mu A$ at at 200kHz sampling rate. This would cause a 100mV drop on a $100k\Omega$ resistor which is not negligible. Since with dynamic voltage measurement the highest sampling rates are desirable, this limitation should be taken into account.

Finally, all ADCs can measure voltage only upto their supply voltage i.e. with different supply voltages, the whole measurement gets shifted. This means that the ADC has to be calibrated to compensate for the voltage shift to ensure that it is measuring correctly. For this purpose, all ADCs have an internal reference voltage, which can be used to calibrate the ADC.

2.2.2. Controlled voltage source:

The easiest way to generate voltage is using a digital to analog converter (DAC), which is an builtin peripheral in some microcontrollers. It often supports DMA transfers, thanks to which it can be used as an arbitrary function generator, however that is out of the scope of this thesis.

The DACs in the F303 series microcontrollers have built in buffers. These buffers decrease output impedance, and allow driving external loads directly, however they limit the maximum and minimum output voltages of the DAC. Therefore rail to rail voltages on the output cannot be achieved. It is possible to disable the buffers, and thus get full rail to rail output voltage, but at the cost of higher output impedance and lower maximum output current. This can be fixed by using an external buffer with better parameters, such as an opamp voltage follower.

Like in the case of the ADC, DACs have selectable resolution. In many cases the maximum is 12bits and lower resolutions such as 8bits can be selected.

DACs are not as ubiquitous as ADCs, so there may be cases where a voltage source is needed, but a DAC is not available. There are two other ways with which a controlled voltage source can be implemented:

1. directly with the digital pins of the microcotroller, or
2. using a PWM signal.

The benefit of using digital pins is that they very simple to set up, however they only only two voltages. Using the PWM signal on the other hand requires a timer peripheral [5](see section 2.2.5), but can provide a higher output voltage resolution.

2.2.3. Dynamic voltage measurement:

Many times we need to see how a signal changes in time, thus we measure the dynamic properties of the signal. The approach is similar to that of static voltage measurement with averaging. The voltage is sampled at a certain frequency, usually much faster though, and instead of averaging those measured values, they are saved to a buffer, and then graphed against time. Since higher sampling frequencies are used (on microcontrollers hundreds of kSps[6] are achievable, on traditional oscilloscopes, hundreds of MSps, even units of GSps), DMA is used to get the highest possible speeds. Also, because using DMA frees up the processor (it doesn't need to facilitate the data transfer between the ADC and the buffer), the processor can perform additional tasks such as triggering and communication.

Since we are interested in the dynamic properties of the signal, often the resolution is less important than the measurement speed. To get the highest speeds, lower resolution, usually 8 bits, is used, since it offers the best compromise between speed and resolution.

All the other limitations that were mentioned for static voltage measurement apply, but since higher sampling frequencies are usually used, the problem with current draw at higher sampling frequencies is more important. Furthermore, at high sampling frequencies, the capacitor in the sample and hold might not have enough time to fully charge to the measured value, and thus the measured voltage will be different. Finally, because the data needs to be plotted in a graph, a frontend which can process and plot the data is required.

2.2.4. Counter, frequency and duty cycle measurement

Counter

A counter simply counts the number of pulses. The easiest way to implement it, is using the timer perihperal and setting the pulse source as the timers clock source. There are two limitations:

1. the maximum counting frequency, which is typically half of the timer clock speed, and
2. the maximum counter size. In the STM32 line microprocessors, there can be zero to two 32bit ($2^{32} \approx 4 \cdot 10^6$) counters and all the others are 16bit ($2^{16} \approx 65 \cdot 10^3$). Usually though, when using the counter as a counter (as opposed to DFM or RFM discussed in further ahead), it is used only at a low speed, and as such, neither the speed nor counter size are a problem.

Direct frequency measurement (DFM)

We can measure frequency directly by gating[7] the counter. A simplified model of a DFM instrument can be seen on fig. 2.1. For example on fig. 2.2: if we gate the counter for a period $T = 1s$ and it counts 5 pulses, the frequency is 5 Hz, since frequency is defined as the rate of something happening in a second.

[5]It cas also be done by switching digital pins with software, this is also know as bitbanging.
[6]The ADCs can actually sample at MSps, but the data trasmission speed from the microcontroller is the limit.
[7]Gating means turning the counter on and off for a certain period of time.

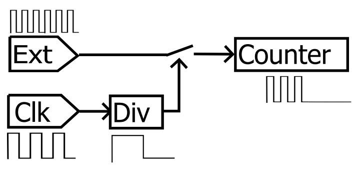


Figure 2.1.: DFM instrument model

The resolution Res_{DFM} is inversly proportional to the frequency of the measured signal f_{ms}, i.e.

$$Res_{\mathrm{DFM}} = \frac{1}{f_{\mathrm{ms}}} \tag{2.1}$$

For example with a measured frequency of 1kHz, the accuracy is 10^{-3}. Conveniently, this also means that with a gating time of 1 s the resolution is 1 Hz. To get good accuracy, the measured frequency should be much higher than the gating frequency, at least 1000 times larger.

In the past, the gating time was decadic multiple of 1 s, such as 0.1 s or 0.01 s because the counter was directly connected to a display. Thanks to this, the frequency at a different gating time was 'calculated' by moving the decimal point on the display. Microcontrollers, have a lot of processing power, so it is easily possible to set any gating time and then recalculate the frequency. This has one downside though, the resolution can become non integer, e.g. 1.2 Hz.

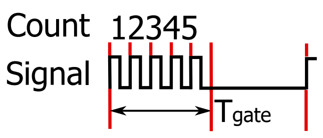


Figure 2.2.: Direct Frequency Measurement

As with all time related measurements, the biggest limitations are caused by

- the accuracy of the clock source. The HSI (high speed internal clock) which is selected as default is very innacurate (0.1 %), but if an external crystal is used for the HSE, even units or fractions of a ppm can be achieved. The second limitation is
- the size of the timer (16 or 32bit). Fortunately, even if only 16bit timers are available, it is possible to chain timers and use the overflow of one as a carry into the other, effectively making 32bit or larger timers.

Reciprocal frequency measurement (RFM)

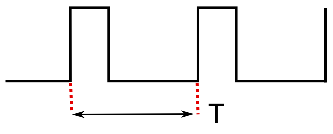


Figure 2.3.: Reciprocal Frequency Measurement

Reciprocal frequency measurement is very similar to DFM. Instead of gating the counter from an internal clock and counting the number of the pulses from the measured source, the pulse signal is used to gate a counter which measures the number of pulses from the internal clock source. Therefore, while the resolution of DFM was given by the measured signal i.e. ±1 pulse of the measured signal, the resolution of RFM Res_{RFM} is ±1 internal clock cycle T_{ic}, which is contsant i.e.

$$Res_{\mathrm{RFM}} = T_{\mathrm{ic}} = \frac{1}{f_{\mathrm{ic}}}$$

Since RFM resolution is set by the internal clock cycle, to measure the signal with high precision the signals should have a much lower frequency than the internal clock. If the clock is 72MHz, then to get a resolution of at least 10^{-3}, the measured signal has to be lower or equal to 72kHz. Since RFM measures the period of the signal T, as seen on fig. 2.3, the frequency has to be calculated using equation 2.2.

$$f = \frac{1}{T} \tag{2.2}$$

Measuring the period (RFM) is more complicated than direct frequency measurement (DFM), because the counter is not enabled by a 'known' signal, but by the measured signal itself. This means that we have to detect where the period starts and where it ends. This can be done either by:

1. having the timer measure the clock pulses at each rising edge, and then calculate the period from the difference of these two signals, or

2. have the first rising edge start the timer, and the other rising edge record the time. This gives us the period directly.

DFM has a large benefit over RFM in that it measures average frequency by definition (pulses/time), however with RFM, only a single period is captured and converted, therefore if the measured signal is not completely periodic, significant errors can be introduced[8].

Even though RFM is used mainly for measuring lower frequencies, the lowest frequency is limited by the timer clock. This is especially the case when a 16bit timer is used. To be able to measure a lower frequency, either a counter with more bits has to be emulated, i.e. use the counter overflow as a carry into another counter, or the clock speed has to be lowered, this of course lowers the maximum frequency that can be measured.

The same limitations apply as with DFM - clock percision (HSI vs. HSE) and counter size (32bit vs 16bit).

Duty cycle measurement

Duty cycle measurement can be done as an extension of indirect frequency measurement, by measuring the pulse width T_{pw} (e.g. how long the period is high), in addition to the period T, as seen on fig. 2.4. The duty cycle $\%DC$ can be the calculated using equation 2.3.

$$\%DC = \frac{T_{\text{pw}}}{T} \cdot 100 \tag{2.3}$$

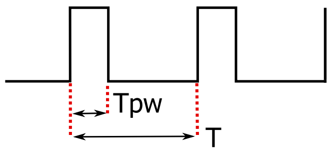


Figure 2.4.: Duty Cycle Measurement

2.2.5. PWM signal generation

PWM or pulse width modulation signals are important signals with a wide variety of usage. PWM signals can be used to control motor speed, LEDs, emulate quadrature signals etc. They can also be used instead of a DAC.

PWM signal generation can be done with the timer peripheral using output compare.

[8]This can be however be corrected by averaging, and since the length of each period is know, the fluctuation of the signal can be analysed.

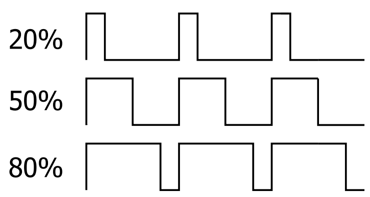


Figure 2.5.: PWM Signal Duty Cycles

As seen on fig. 2.5, PWM works by switching an output high and low at a specified ratio - the duty cycle, at a high frequency. The average output voltage is calculated using equation 2.4 where V_{high} is the voltage on the pin when it is high, and $V_{\text{low}}$is the voltage on the pin when it is low (this is typically 0 though).

$$V_{\text{out}} = \frac{\%\text{DC}}{100} \cdot (V_{\text{high}} + V_{\text{low}}) \tag{2.4}$$

Output compare is the timers ability to trigger an output without interrupting the processor, thanks to this, it is possible to set the timer i.e. set the frequency and duty cycle, and the timer will generate the set pwm signal until it is requested to do something else. Because the frequency and duty cycle are set with a counter, similar to RFM, the same limitations apply, i.e. the counter size (16bit or 32bit) and maximum clock speed limit the frequency and duty cycle resolutions at high speed and the frequency range.

2.3. Approaches to developing embedded instruments

There are many situations in which an embedded system is developed. These situations can be very roughly grouped into three development groups:

- Fast - the system is required in a very short time, and it is expected that the microcontroller will not be used to its full potential. This can be done with the online IDE and framework - **mbed**. It provides a highly abstract api, which allows high code portability (even between different vendors) and virtually no knowledge of the microcontroller for which the code is written. This approach is good for beginners with microcontrollers, because it doesn't require deep studies of the microcotroller and many working, simple to understand programs are provided. This approach is discussed in chapter 3.

- Maximum utilization - it is desired to get out the most of the microcontroller, e.g. an oscilloscope. This would normally mean implementing the system with a very light framework, or without one altogether, because frameworks create an overhead. It would also mean writing to the microcontroller registers directly, in C language or even in assembler. This approach also requires rigorous study of the microcontroller documentation, datasheet and errata. A light framework suitable for this approach is **STM32 Perihperal Libraries**, of course, it is only usable microcontrollers from the STM32 line. It is also possible to only use the memory definitions from the framework and write even more low level code. However, since ST is moving away from the preipheral libraries, they will not be discussed in this thesis, and their successor - STM32Cube will be discussed.

- A compromise between the previous two. This means there is code abstraction with a framework and it is possible to utilize and completely control all the features of the microcontroller, but it is not possible to get the most from the microcontroller. **STM32Cube** is precisely for this. It also has a GUI for generating peripheral initialization code, which is usually the hardest part in microcontroller programming. This approach is discussed in chapter 4.

CHAPTER

THREE

QUICK DEVELOPMENT USING MBED

3.1. Overview

In microcontroller development, usually the hardest part is initializing the peripherals, that is enabling the peripherals clock, actually setting up the peripheral itself, setting up the GPIO pins correctly, and the debugging to see that the functionality was actually set up correctly. All of this requires a good understanding of the peripherals and what one is doing, and while all the functionality is described in the reference manuals, datasheets and erratas, setting up the peripherals can be very time consuming.

Unfortunately, the complexity and time requirements of initializing the microcontroller peripherals make microcontroller development very hard for beginners. This can pose a problem even for experienced users if they need a program up and running fast. For these and other reasons, Mbed was created.

Mbed is an online IDE and framework, which allows writing highly abstract code for supported ARM[1] based microcontrollers. All the complex initialization is done with single line commands, and there are many examples and tutorials. For example dimming an LED with PWM is almost trivial[2] - first a `PWMOut(PinName pwm)` object is created, which initializes all the peripherals required for PWM signal generation, and then the `write(float dutyCycle)` method can be used to set the duty cycle. One of the benefits of such abstract code is that it is very portable onto other microcontrollers, even to those from other vendors (this is hardly, if at all possible when writing at a lower abstraction level).

Of course mbed has its cons, because of the abstraction level, one cannot use the full potential of the microprocessor[3], and mbed can only be used with supported boards. As mentioned earlier though, these cons are completely outweighed in some situations. As mbed enables very fast prototyping and also has a very low learning curve, it is ideal for students, beginners or even experienced users.

[1] ARM develops the ARM architecture and licenses it to other companies who make their own products and add their peripherals, such as TI, NXP or ST.

[2] Normally, these two lines of code would translate to enabling clocks to the timer and gpio peripherals, setting up the timer in output compare mode and calculating the appropriate autoreload and capture compare values, and setting up the gpio in correct alternate function mode.

[3] With mbed it is possible to directly acces the registers, but it requires very good knowledge of the microcontrollers and thus won't be discussed.

There are two key documents when working with mbed - the handbook[11], and the cookbook [10]. The handbook contains information about individual libraries, their APIs and examples on how to use them, and the cookbook contains full tutorials, including client and server side programs where relevant.

Even though mbed supports a plethora of microcontrollers, some libraries, especially user contributed ones, don't support all microcontrollers. Many libraries support only LPC series microcontrollers from NXP, so before using a library, do look into the handbook whether your microcontroller is supported.

Important! Each pin supports only some functions, for example on the F303RE nucleo, `AnalogOut` (digital to analog converter) is supported only on pins D13 (PA_5) and A2 (PA_4). To find out which functions are on which pins, check the pinout of the board on the mbed site. For example the pinout for the F303RE nucleo can be found at [16]. If classes are incorrectly map to pins (such as PWMOut to PB_3), the program will not start and only "pinmap not found" will be sent over uart.

3.2. Ways to control an embedded instrument

We can divide individual embedded instruments by how the user can interact with them. There are four main ways of controlling them:

1. Not using any control. The device is plugged in and just 'works'. This could be a signal or voltage generator where changes to the output signal are not required. As this method is very simple, it won't be discussed further.

2. Controlling with manual reset i.e. singleshot. After connecting the device to a power supply, it generates a set signal, say 100 periods of a PWM signal and stops. To get another set of the signal, the user simply resets the device either by a builtin reset button (many microcontroller boards have these) or by unplugging and plugging back in the USB cable. This is a rather crude approach but is great when one needs to quickly test something. A visualization of this can be seen on fig. 3.1. Like the previous case, this method is very simply and won't be discussed any further.

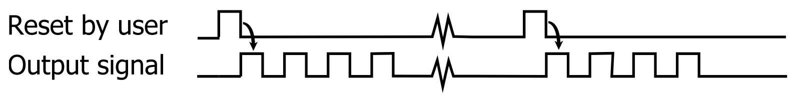


Figure 3.1.: Singleshot control

3. Controlling with buttons and LEDs. Buttons can be used to cycle through preset programs, e.g. signal frequencies, or shapes, to start and stop generation etc., and LEDs can be used as simple status displays. It is discussed in section 3.2.1. A possible use of this approach can be seen in fig. 3.2, where a button is used to select 1 of 3 freqeuncies.

4. Using a display, such as an LCD to communicate data to the user. The main difference between this approach and the other 4 is that it requires an external library which must

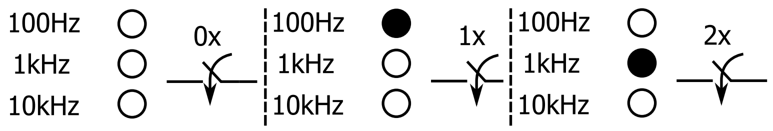


Figure 3.2.: Control with buttons and LEDs

be imported. The huge benefit over external control is that the instrument is self contained while providing data in text format. Buttons can be added for interactivity. This approach is discussed in more detail in sec. 3.2.2

5. The last method is external digital control, i.e. control over a serial connection, USB, I2C etc. With a serial, or USB connection, the system can be controlled from a computer terminal application, or a dedicated PC application. This method is very flexible, since full text strings can be displayed to the user and sent back to the device. It also allows further data processing. This approach is discussed in section 3.2.3

3.2.1. Control with buttons and LEDs

There are cases where controlling the instrument from a computer would be more complicated than useful and only a few presets or simple settings are needed. In these cases the instrument can be controlled with just buttons and LEDs. This approach is great for signal generators, and simple digital (on or off) controls and displays.

For example, signal generators with only a few frequencies can be controlled very easily with one button and optionally a few LEDs. The button cycles through the frequency presets and the leds can show the selected preset. An example can be seen in fig. 3.1.

If one would like to save on buttons, holding a button down can be used as another operation. If one would like to save LEDs dimming them can be used to show the selected presets (e.g. 0%, 25%, 50%, 75%, 100%). This can be done with the DAC or a PWM signal, which are discussed in sections 3.4 and 3.10 respectively. An example of change the brightness of an LED can be seen in Alg. B.6.

Controlling the instrument with buttons and leds is quite convenient when working with mbed. One can buy a prototype shield[4], seen on fig. 3.3, solder onto it the required LEDs or buttons and plug the shield in to create a standalone embedded instrument.

A huge benefit of this method is that it saves space (as opposed to using a computer), while offering an intuitive way to control the system, however if more than the built in controls are required (typically one button and 2 LEDs), extra hardware and all the costs that come with it are incurred.

To work with digital (2 state) devices such as buttons or LEDs one can use the

- `DigitalIn(PinName input)` and `DigitalOut(PinName output)`objects, and their
- `int read()` and `write(int value)` methods for reading and setting them.

An example of using a button to control an LED can be seen in Alg. 3.1. The main benefit of using the `DigitalIn` and `DigitalOut` classes is that they work with all pins.

[4]Shields are boards which are placed on or under the main board to extend its functionality. A prototype shield is a prototype PCB with a layout to fit the development board.

Algorithm 3.1 Mbed controlling an LED

```
DigitalIn button(PC_13);
DigitalOut led(LED1);
int main(){
    while(1){
        //One can simply assign the button value to the LED
        led = button.read();
        //Or use a conditional
        if(button){
            led = 1;
        }else {led = 0;}
    }
}
```

3.2.2. Using an LCD display

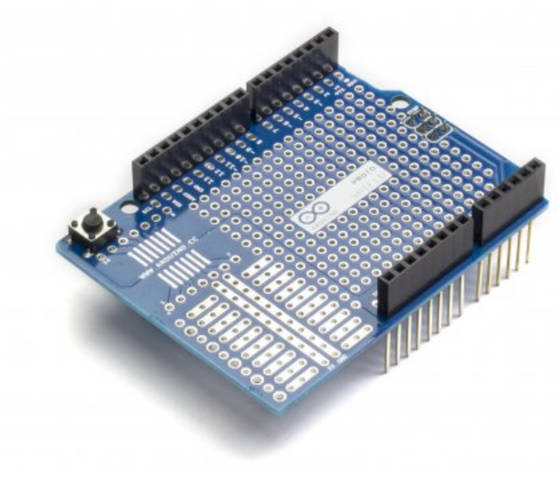


Figure 3.3.: Prototype shield with Arduino headers

When creating selfcontained instruments LCDs can be very useful, as they provide much more information than simple LEDs, but don't need a computer attached. Since controlling LCDs directly is not very viable (too many pins), most LCDs have a controller.

One of the most widely used cotnrollers is the HD44780. In the minimum configuration it requires 4 data lines and 2 control lines, thus taking up 6 GPIO ports. Although the control protocol is not very simple, mbed has multiple libraries just for the HD44780, so one doesn't need any knowledge of the controller except its pinout. I have used the HD44780 library [5] to create an example that measures voltage on a potentiometer (A), a reflective optical sensor (B), and measures the number of passes through an optical gate (C and count). A picture of it in action can be seen in fig. 3.4.

You might notice that there isn't a nucleo board, but just a microcontroller on a breakout board. The program was developed for the STM32F042 nucleo board, and the flashed onto the STM32F042 microcontroller seen in the picture. Please refer to Apendix A

3.2.3. Control using a serial interface (UART)

Many development boards, especially low cost ones, have very limited control and display capabilities, and limited memory and processing power. While there are cases where this does

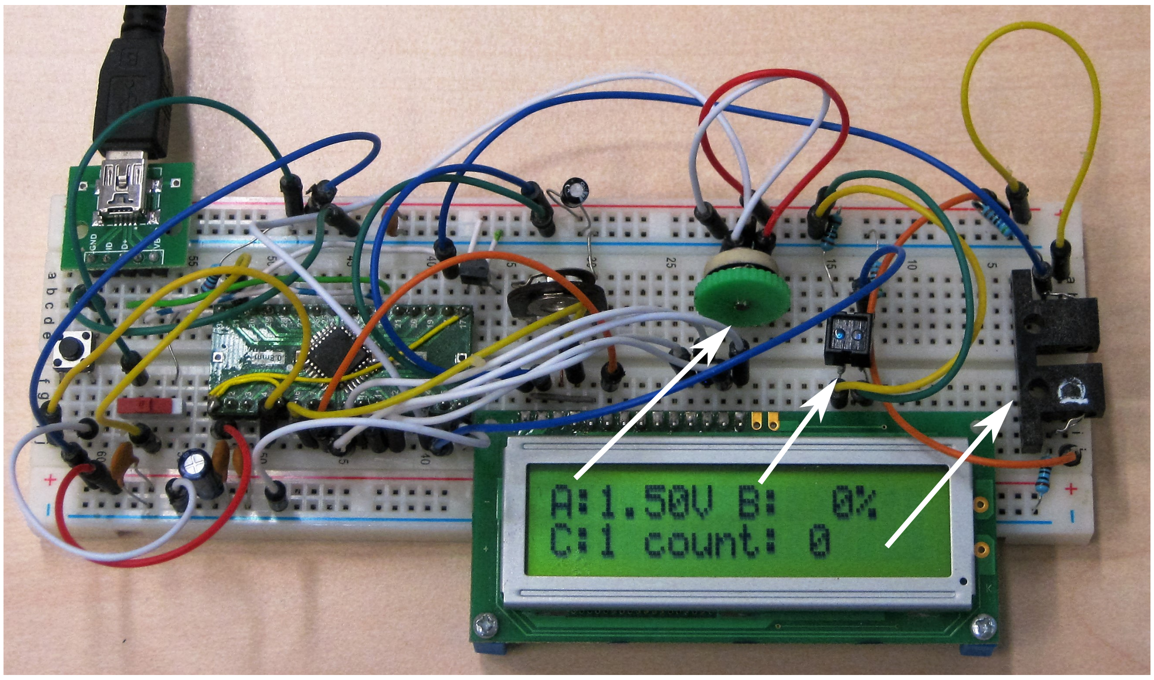


Figure 3.4.: Using an LCD Display

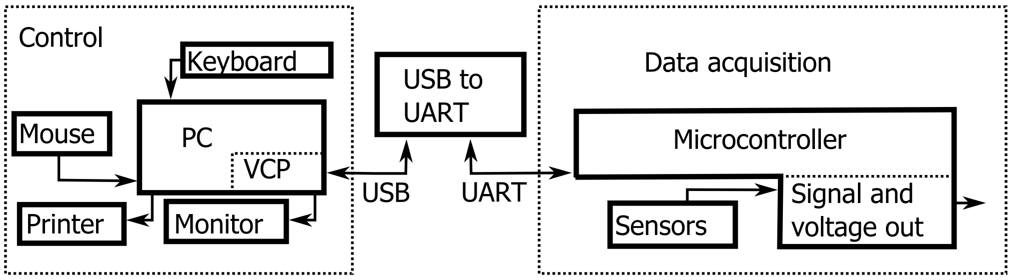


Figure 3.5.: Model of external control with a PC

not matter (such as function and voltage generators with presets), there are other cases in which we would like more control and feedback. Of course, more hardware can be added to facilitate this, such as a displays and keyboards, but since the boards that are supported by mbed typically have a VCP, controlling the board over a serial connection from a computer is much simpler. A VCP or virtual com port is a software interface that enables applications, such as a terminal or a full control GUI application, to access a USB device as if it were a serial device. Typically a VCP serves as a bridge between USB and serial device.

For its simplicity, asynchronous serial communication is a widely spread, and most microcontrollers have UART peripherals which facilitate it directly[5]. Typically, data is transmitted at baud rates[6] which are multiples of 9600. Baud rates higher than 115200 are not recommended.

[5]The reader might be confused at this moment. Most development boards have two processors, one is the actual microcontroller for which we are designing the embedded instrument, and another which can work as a debugger, a virtual com port .

[6]The baud rate is the number of bits transferred per second, including control bits.

In the basic setting, a data frame consists of 2 control bits - the start and stop bits, and 8 data bits. This frame can be seen in fig.3.6. If there is a need, a different number of data and stop bits can be set, and extra features such as a parity bit and flow control can be turned on. Hardware or software flow control can be used to limit data flow if the receiver cannot cope with the speed of the transmitter[4].

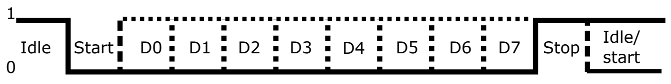


Figure 3.6.: Typical UART data frame

Adding control over serial communication to an application using mbed is very simple, as med has a builtin library for that[8]. To use it, a `Serial(PinName tx, PinName rx)` object is created and then the `char getc()` and `void putc(char c)` methods can be used to read and write a single byte. `int readable()` can be used to see whether a byte is available and `printf(char *format,...)` and `scanf(char_*format,...)` can be used for full string operations. A very simple example of can be seen in Alg. 3.2.

Algorithm 3.2 Mbed simple command processing

```
#include "mbed.h"
Serial pc(USBTX, USBRX);
int main() {
    while(1) {
        /*By using readable() we don't overwhelm the
        system, and other commands can be executed*/
        if(pc.readable() && pc.getc() == '?') {
            //If we receive a ?, return a hello world.
            pc.printf("Hello world!");
        }
    }
}
```

Furthermore, mbed has a simple implementation of interrupts[7], which are a better way of handling serial communication. Alg. B.7 shows a simple interrupt based command handler.

Note that like with every object (e.g. Serial) in mbed, to get more information on the available methods, simply look up the object in the mbed handbook. All the mentioned features including baud rate, flow cotrol or data format can be changed by calling the appropirate method on the object.

3.3. Static voltage measurement

As was already mentioned in the goals, static voltage measurement is the most common type of measurement. It is done with an analog to digital converter (ADC) which is built into most microcontrollers and is reasonably accurate.

In mbed, the ADC is abstracted into the AnalogIn class. To use it, an `AnalogIn(PinName in)` object is created, and its `float read(void)` method is used to read the voltage.

The voltage is measured as a fraction of the ADC supply voltage, therefore the range of the returned value is from 0 to 1. The `read()` method can be used implicitly. Assuming that the system is controlled using a serial connection, an example of a simple 2 input voltmeter can be seen in alg. B.1

Since the input voltage of the ADC is limited, in the case of the the STM32F303 and STM32F042 series microcotrollers to 0 to 3.3V, it can be desirable to shape the inpt signal so that one can measure a larger range of voltages. To learn more about this, refer to section 5.1.

It is possible that the input signal will be noisy and riddled with intereference. This could cause errors in the measurement. An in depth analysis of the causes is in section 5.3. In many cases averaging from a large enough sample set (at least hundreds of samples), which is taken over whole periods of $20ms$ (for 50Hz intereference) is the easiest way to remove most noise and interference.

By setting up a for loop with all the operations for reading the ADC for say 1000 cycles, we can measure the sampling period. For a cycle which only adds the current reading to a variable, which will later by divided to get the average, a period of $8.2\mu s$ has been measured. The minimum sample set size n_s can be calculated with equation 3.1, where T_cycle is the period of a single cycle.

$$n_\text{s} = \frac{0.02}{T_\text{cycle}} \tag{3.1}$$

Algorithm 3.3 Mbed measuring the adc conversion time

```
#include "mbed.h"
Serial pc(USBTX, USBRX);
Timer t;
AnalogIn adc(A0);
t.start();
for(int i = 0; i < 1000; i++) {

    data += adc.read();

}
t.stop();
printf("\nTotal time: %dus\t",t.read_us());
```

3.4. Controlled voltage source

The easiest way to have a fully controllable voltage source is to use a digital to analog converter (DAC). On many boards however, there is either no DAC or it has only a few channels. The STM32F303 has two DAC channels, which are on pins PA4 and PA5, while the STM32F042 microcontroller has none, so sometimes another method has to be used.

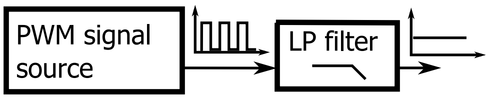


Figure 3.7.: Voltage source using PWM

With mbed, to use the DAC an `AnalogOut(PinName out)` object is created, and by assigning values to it, the output voltage is set. An example of its usage can be seen in alg. B.6.

Roughly in the middle of the code it can be seen that the dac is set with a single line - `extLED = ((float)state) / 4;`.

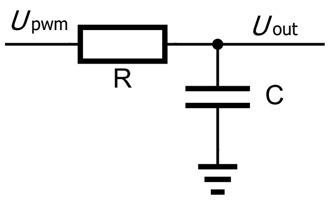


Figure 3.8.: RC circuit as LPF

If two voltage levels are sufficient a digital pin can be used. In cases where only the mean output voltage matters, a It is possible to use a PWM signal with a low pass filter[1] as seen on fig. 3.7. For example when controlling LEDs, or controlling DC motors (of course with a driver between the motor and the microcontroller). For generating a PWM signal source, see section **3.10.**

To smooth the PWM signal, a low pass filter can be added as seen in fig. 3.7. However some ripple $\%Rip$ in the output will be present. If an RC circuit as in fig. 3.8 is used as the LP filter, the ripple as a percentage of the supply voltage can be calculated with eq. 3.2, where $f_{\text{pwm}}$is the frequency of the PWM signal, and R and C are the values of the RC resistor and capacitor respectively. After a bit of t, it can be seen that with increasing α the ripple increases, therefore to decrease the ripple the PWM signal frequency, resistance or capacitance of the LP filter have to be increased.

$$\%Rip = \frac{1-e^{\alpha}}{1+e^{\alpha}} \cdot 100,\ \alpha = \frac{1}{f_{\text{pwm}} \cdot 2 \cdot R \cdot C} \tag{3.2}$$

With a bit of algebra, eq. 3.2 can be rearranged to express the parameters directly as seen in eq.3.3.

$$f_{\text{pwm}} \cdot C \cdot R = \frac{1}{2ln(\frac{1+\%Rip/100}{1-\%Rip/100})} \tag{3.3}$$

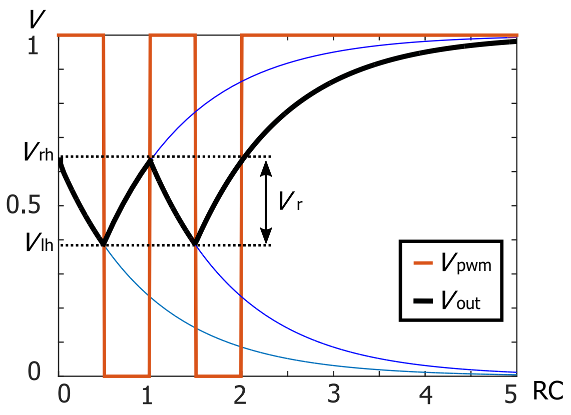


Figure 3.9.: Ripple on PWM Signal with a LPF

To derive the ripple equation, we first assume that the PWM signal has a 50% duty cycle[7], ripple has already stabilized, and the circuit is currently in the situation seen in fig. 3.9. The lowest and highest voltages of the ripple V_{rl}, V_{rh}will occur at the edges of the signal. Since the signal has a 50% duty cycle, the highest voltage can be written as:

$$V_{\text{rh}} = \frac{V_{\text{cc}}}{2} + \frac{V_{\text{r}}}{2} \tag{3.4}$$

where V_{r} is the peak to peak ripple voltage. The lowest voltage will appear on the next rising edge, which occurs after half a period $T_{\text{pwm}}$of the PWM signal. As the voltage on the capacitor is given by an exponential and the RC time constant τ, V_{rl} is calculated with eq. 3.5. Furthermore the ripple voltage is calculated with eq. 3.6.

$$V_{\text{rl}} = V_{\text{rh}} e^{\frac{-t}{\tau}},\ t = \frac{T_{\text{pwm}}}{2} \tag{3.5}$$

[7]This simplifies the calculations and also, at 50% duty cycle the ripple is the highest.

$$V_{\mathrm{r}} = V_{\mathrm{rh}} - V_{\mathrm{rl}} \tag{3.6}$$

Therefore by substituting eq. 3.5 into eq. 3.6 we get eq. 3.7.

$$V_{\mathrm{r}} = V_{\mathrm{rh}}(1 - \mathrm{e}^{\frac{-T_{\mathrm{pwm}}}{2\cdot\tau}}) \tag{3.7}$$

Then by expressing V_{rh} from eq. 3.7 we can set it equal to eq. 3.4 and get to eq. 3.8.

$$\frac{V_{\mathrm{r}}}{2} + \frac{V_{\mathrm{cc}}}{2} = \frac{V_{\mathrm{r}}}{1 - \mathrm{e}^{\frac{-T_{\mathrm{pwm}}}{2\cdot\tau}}} \tag{3.8}$$

Finally, equation 3.8 can be simplified to 3.9.

$$V_{\mathrm{r}} = V_{\mathrm{cc}}\frac{1 - \mathrm{e}^{\alpha}}{1 + \mathrm{e}^{\alpha}},\ \alpha = \frac{T_{\mathrm{pwm}}}{2\cdot\tau} \tag{3.9}$$

The RC Filter design tool [3] can be used to aid one in the design of a suitable LP filter.

If a two state voltage sourceis sufficient, a digital pin can be used. An example of this can be seen in alg. 3.1. After creating the `DigitalOut(PinName out)`, the output is set by writing to it: `led = button;`.

Do note that all pins on the microcontroller have a limit of roughly 20 mA. This means that if a larger output current is required, a transistor or driver needs to be used.

3.5. Dynamic voltage measurement

An oscilloscope won't be implemented because it is not practical with mbed. Normally the most important properties of an oscilloscope are its bandwidth and sampling frequency, and the best possible solution created with mbed will be inferior to an implementation with a lower level (less abstract) framework like STM32Cube or even the ST Standandard Peripheral Libraries (there are implementations for mbed which reach speeds of ~70kSps, with peripherals, 100kSps can be reached easily without optimalizations). Therefore an oscilloscope is implmented only in section **4.**

I have tested that it is possible to directly read the data from the ADC at roughly 120 kSps, however this is only reading data out of the ADC, not sending it to the pc or even processing it. Therefore the actual speed when something useful is required will be lower. Furthermore if the data is sent directly over the VCP, the baud rate is the bottleneck. I have tested that by setting the baud rate to 1000000, data can be directly transmitted at roughly 80 kSps. Data transmision at such a high baud rate could be prone to errors and also, all the data processing would have to be done on the PC.

3.6. Simple pulse counter

In microcontrollers, pulse counters usually aren't a dedicated peripheral, but instead a timer (a builtin peripheral) is used. The pulse source is connected to the timer clock and all the other functionality of the timer is disabled.

Currently mbed doesn't have a counter nor a timer class which could be used to control the counter and direct access to the timer is possible only by accessing the registers. For this reason the counting has to be done on the software side with interrupts. This software overhead will unfortunately cause errors in the measurement, and the performance won't be as high as it could be when using the hardware. Since this chapter explores mbed as either a very quick or a very simple development platform though, it will not be a problem.

External interrupts are facilitated with the `InterruptIn` class. To use it, an

- `InterruptIn(PinName in)` object is created, and then a callback method is attached, with the
- `rise(void(*fptr)(void))` or `fall(void(*fptr)(void))`methods to the rising or falling edge respectively.

The complex signature simply means that a pointer to a method with no returns and no parameters should be used as the argument. An example of a counter can be seen in alg. 3.4.

Algorithm 3.4 Mbed pulse counter using interrupts

```
InterruptIn pSource(PC_13);
int pCount = 0;
//Just increment the counter with each high
void incrementCount(){pCount++;}
int main(){
    //Attach the count method to the rising edge of the pulse
    pSource.rise(&incrementCount);
    while(1){/*Send data ...*/}
}
```

The two limitations of this approach were already stated. Since it interrupts are used instead of the hardware timer, this approach cannot reach frequencies higher than 200 kHz (the hardware timer is limited by the clock speed, which is in the range of tens of MHz) and also because the pulse is transferred with an interrupt, the propagation time is not known, introducing error. Furthermore, at higher frequencies the interrupts can limit the processing time allocated to other parts of the program, or even completely choke the program.

The speed limitation and direct software processing does however remove the counter size problem, since a software variable is used and it can always be 32 bit.

3.7. Direct frequency measurement

In chapter 2 it was explained that frequency can be measured directly by gating the counter for a certain period. If the counter from section 3.6 is used, timer interrupts can be added to start and stop the counter.

These timer interrupts are implemented with the ticker class. The ticker is initialized by creating a

- `Ticker(void)` object and attaching a callback method. The callback is attached with the
- `attach(void(*fptr)(void), float seconds)` or `attach_us(void(*fptr)(void), float t microseconds)`metods which also specify how long between each callback. To stop the callbacks, the callback has to be detached with

- `detach()` method.

An example of DFM implemented in mbed can be seen in alg. B.2.

DFM implemented with a timer has three main benefits over RFM. It has a much higher maximum measurable frequency, lower error as DFM uses averaging by definition, and simpler implementation. But since interrpts are used, the maximum frequency is severly limited.

I have tested that it is possible to measure frequencies upto 200 kHz accurately, with an accuracy of 5 ppm at 200 kHz. This is an astonishing result as the accuracy is extremely high considering the counting is done on the software side.

The accuracy of measurement fell very fast at higher frequencies, at 400 kHz the accuracy was roughly 1000 ppm and over 410 kHz the accuracy was completely off (random). The program stopped responding at 740 kHz.

3.8. Reciprocal frequency measurement

The method of repciprocal frequency measurement was explained in section 2.2.4.

From the previous sections we know that the only way to access the timer peripheral with mbed is to directly access the registers and this is not interesting in the scope of thi thesis. However, mbed it can directly drive interrupts from pins with the `InterruptIn` class. The time between two interrputs of the same polarity has to be measured to get the desired period T. This can be done with the `Timer` class which is initalized by creating the `Timer(void)`object and it is controlled with the `start()`, `stop()`, `reset()`, and `read_us()` methods. An example of RFM implemented using mbed can be seen in alg. B.3.

The are main downside to this approach is the same like in the cases of the counter and DFM: all the 'triggering' is done on the software side, which doesn't guarantee any timing and could choke the system at higher speeds. Furthermore, RFM calculates the frequency from a single period, which could introduce significant error if the signal is not perfectly periodic, or some delays occur in the program itself. The degree of this error could be lessend by measuring multiple periods instead of just one.

Unfortunately RFM has reasonable accuracy only upto 15 kHz, at higher frequencies the the output fluctuated between the actual value and another value that was more than 1000 ppm higher. It is interesting that the program started to freeze already at frequencies higher than 90 kHz. Thats 7 times less than in the case of DFM.

3.9. Duty cycle measurement

Duty cycle can be measured by slightly modifying indirect frequency measurement. Since we have defined duty cycle as the ratio of the period when the signal is high, and RFM measures the complete period of the signal. Therefore the pulsewidth T_{pw} has to be measured and the duty cycle can be calculated as $\%DC = \frac{T_{\text{pwm}}}{T}$.

This can be done by starting another timer for the pulsewidth together with the period counter, and then stopping it on the falling edge of the signal (assuming the frequency is measured on the rising edges). The code for this can be found in alg. B.4.

3.10. PWM Signal generation

A PWM (pulse width modulation) signal is useful in many cases, it can be used for motor and servo control, by adding a simple lowpass RC filter with a time constant roughly the same as the PWM signal frequency an 'analog' test signal can be created, or, as described in section 3.4, by adding a lowpass filter with a very long time constant, it can be used as a voltage source. A PWM signal can be seen in figures 2.5 and 2.4, T is the PWM period and T_{pw} is the PWM pulsewidth.

The default library for PWM in mbed is `PWMOut`, and in many cases it is sufficient: it allows for setting the period and pulsewith with a double in seconds, and an integer in milliseconds and microseconds. If higher frequencies are neede, the `FastPWM` library can be used. It allows setting the period and pulsewidth in doubles in all ranges (seconds, milliseconds and microseconds), and for less overhead, it even allows setting the number of clock ticks per period and pulsewidth and the clock prescaler.

To use the PWM classes, a

- `PWMOut(PinName out)`object is created, and then its
- `period(float seconds)` and `pulsewidth(float seconds)` methods, or for more precise control,
- `period_ms(int miliseconds)`,`period_us(int microseconds)`,`pulsewidth(int milliseconds)` and `pulsewidth(int microseconds)` methods are used to set the period and pulsewidth..

Furthermore the implicit `write(float value)` method sets the duty cycle directly. A simple example of pwm generation can be seen in alg. **B.5**.

CHAPTER

FOUR

FULLY FEATURED DEVELOPMENT WITH STM32CUBE

If one needs to create a fully featured embedded instrument, which can utilize most of the power of the microcontroller, but doesn't wish to spend long times learning the intricate details of a specific microcontroller, STM32 Cube is a great choice. Among others it allows easy portability to other microcontrollers from the STM32 line and very simple adding and initialization of new peripherals.

STM32 Cube [21] is a software platform from STMicroelectronics which consists of a graphical tool for setting up the required microcontroller and generating initialization code - STM32 CubeMX, and embedded libraries, including HAL (Hardware Abstraction Layer) and middlewares such as RTOS.

One of the most time consuming processes in microcontroller development is writing code for initializing peripherals. This is because one has to learn how to operate registers of the peripheral[1] and then program them. CubeMX allows setting up the peripherals, assigning pins and generating a project for the most common IDEs[2], and others, all from a nice GUI. It also allows changing the setup, including adding peripherals and changing clocks, and then regenerating the whole project while keeping the developers code intact.

It should be noted that one has to understand how peripherals work, otherwise CubeMX will not be very helpful. Also, debugging a microcontroller program will ultimately require examining the peripheral registers during run time, however, with the bulk of the code generated by CubeMX it is much simpler to find what one is looking for in the reference manual.

All the code that uses peripherals initialized with CubeMX, must be placed after the calls to the initialization functions. These are currently named as MX_<peripheral>x_Init, where x is the number of the peripheral.

The two most important documents when working with microcontrollers are their datasheets and reference manuals. For the F0303 they are [15, 20] and for the F042 [18, 19].

[1]This information can be found in the reference manual, and the microcontroller specifications can be fonud in the datasheet. Just to give you an idea, the reference manual for STM32F3xx line has more than 1100 pages.

[2]Integrated develpment environment, it usually consists of a code editor, build tools and a debugger. For example KEIL uVision, TrueStudio, or Coocox

4.1. Working with cube

Cube is a huge framework, and explaining what each of its function does is completely out of the scope of this thesis. Therefore rather than doing that, this chapter aims to give an idea of how to work with cube, and where and what to look for when trying to implment something.

When working with cube it is highly advisable to have the reference manual at hand. Even though cube is getting better and more verbose with each version, the most information about the settings of the peripherals and microcontroller can be found in the reference manual.

It is also a good idea to inspect the code generated by CubeMX and the actual HAL functions. This is great especially for beginners and intermediate developers, as they can learn the naming conventions and the 'philosophy' of the framework. Furthermore, it helps understanding what the functions actually do and help debug the program.

For each discovery, nucleo and eval borad, there are packages with examples. While they do not contain the CubeMX file, and may need some work to work on another board e.g. PWMOutput on a 303E Eval board might not work on a 303K8 nucleo, they give an idea of how its supposed to be done.

Each HAL driver set i.e. F3, F4, L1 etc. has a user manual, which can be found with the download links for the cube itself. Personally I have not found it very useful, since it contains mostly the same information as the comments in the code.

Finally, use the internet. Many problems have been faced by other developers and have already solved on the ST forums and stackexchange.

4.2. Overview of CubeMX

Cube consists of a set of libraries, and a GUI tool to generate initialization tool - CubeMX. This section aims to give a brief overview of the tool and its use.

After starting, the user can load a project, or create a new one. If a new project is created, the MCU/board selection screen is displayed (fig. 4.1 on page 24). Here the user can select a microcontroller or a develpoment board from the STM32 series. The user can filter the micro-cotrollers and boards by available peripherals, series and package among others.

After selecting the microcontroller or board, the pinout screen is displayed (fig. 4.2 on page 25). Here the peripherals can be enabled, and their basic settings set. If the peripherals need to use an external pin, it can also be set on the right side. While not seen in the picture, the middlewares can also be set in this screen. If an external crystal or clock source is used, it must first be enabled on this screen in the RCC selection.

The clock configuration tab is self explanatory. The clock speeds of peripherals and busses can be set either manually by setting the respective sources, or the automatic selection utility can be used by entering the required frequency into the frequency field and pressing enter.

The configuration tab is a continuation of the pinout tab. Seen in fig. 4.3 on page 26, the left side it shows an overview of the currently enabled peripherals and their basic settings. On the right side is a selection of the available detailed settings, which then opens a configuration window seen for example in fig. 4.5 on page 28. In the parameter settings tab the settings of the peripheral are set. The tab User Constants allows, as the name suggests, to define constants. In the NVIC Settings tab, the interrupts available for the peripheral can be enabled and disabled. In the DMA tab the DMA controller can be set up to work with the peripheral if it is available and in the GPIO Settings tab the GPIO settings such as pull-up mode and output speed can be configured.

A very important screen is found in Project->Settings (alt+p). Here the toolchain for which the project will be generated can be selected. Furthermore in the Code Generator tab, under the template settings one can choose the destination folder.

4.3. Control over UART with Cube

It has already been explained in the mbed chapter why controlling the instrument using serial communication is great. It allows to utilize the versatility of the PC to control the instrument from either a terminal, or a fully featured application, and then use the processing power of the PC to process the received data. For example, I created a frontend with Labwindows CVI which uses UART for communication with the device. It can be seen in fig. 4.4.

Assuming that computers with serial ports almost don't exist, the easiest way to add a serial communication channel between the embedded instrument and PC is to use a VCP (USB to UART bridge).

4.3.1. Using the UART to USB bridge on nucleo boards

All STM32 nucleo boards have a uart to usb bridge, therefore one only needs to initialize the UART on the microcontroller and everything is ready on the instrument side. Do note that many of the STM32 microcontrollers have multiple UARTs, and **on nucleo boards, only UART2 is connected to the UART to USB bridge**.

HAL offers three modes of transmiting and receiving data over UART:

- Blocking mode: the processor moves a byte into the UART data register, and once it is transmitted, it moves in another byte. All of this is done in a while loop, so the processor is effectively stuck until either all the data is transmitted, or a timeout occurs

- Nonblocking mode with interrupts: it is an improvment upon the previous method. The processor still moves the data into the UART data register, however, once the data is moved, the processor is free to continue in what it was doing. Once the byte is transmitted, an interrupt is thrown and the processor moves another byte.

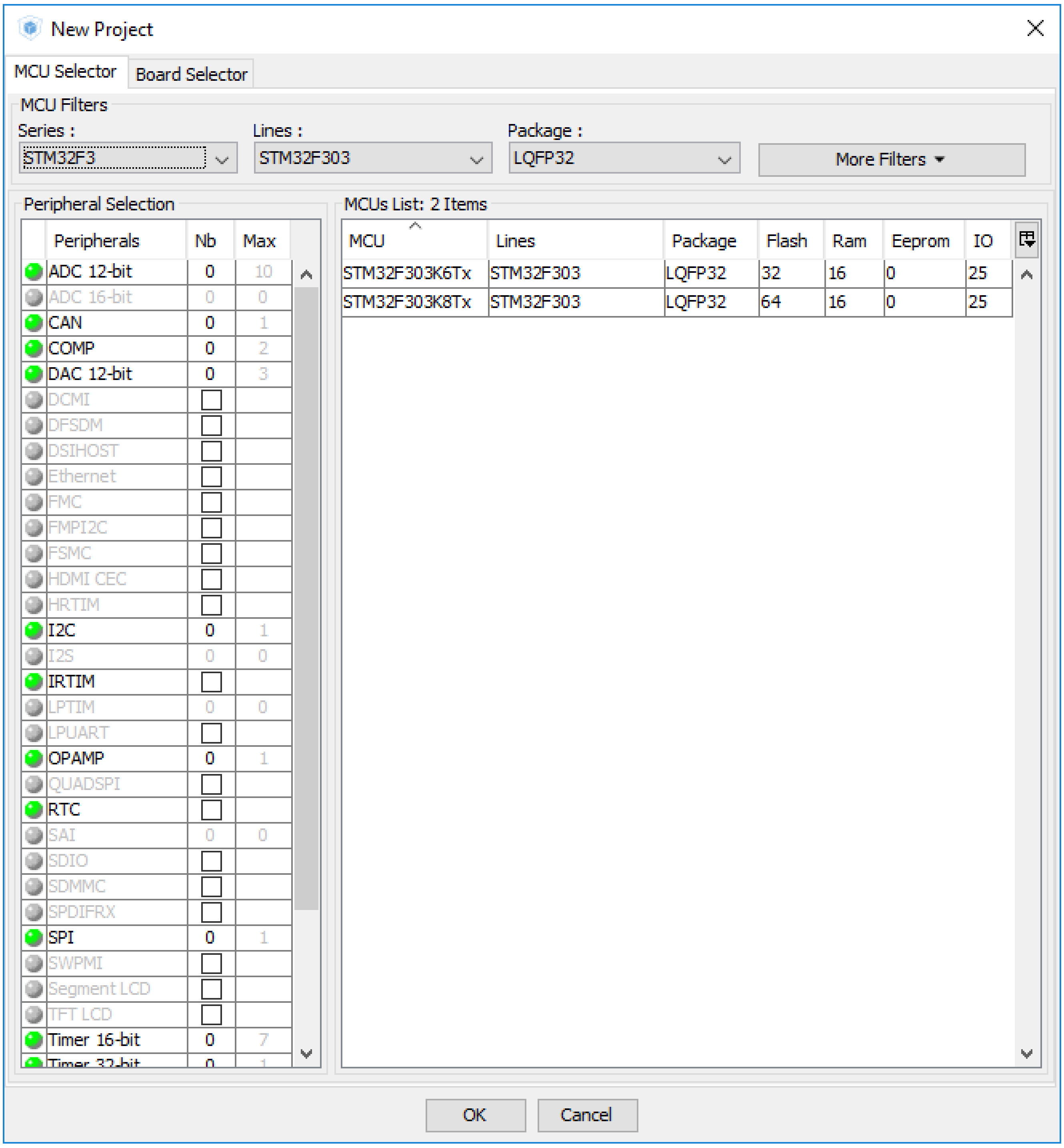
- Nonblocking mode with DMA: this modes uses the direct memory access controller to move the data from an array into the UART data register, and the processor doesn't have to do anything. Do note that to use this method, the DMA controller has to be initiated separately.

Since the processor is free to do other things in nonblocking modes, it needs to be notified somehow that the transfer is complete. This is done with callbacks, and their method signatures can be found in the IO operation functions section of the uart library. A callback is used simply by implementing a function with its signature. The most important callbacks are:

- `void HAL_UART_TxCpltCallback()` - transfer complete callback

- `void HAL_UART_RxCpltCallback()` - receive complete callback

- `void HAL_UART_ErrorCallback()` - error callback



New Project

MCU Selector | Board Selector

MCU Filters

Series : STM32F3 | Lines : STM32F303 | Package : LQFP32 | More Filters

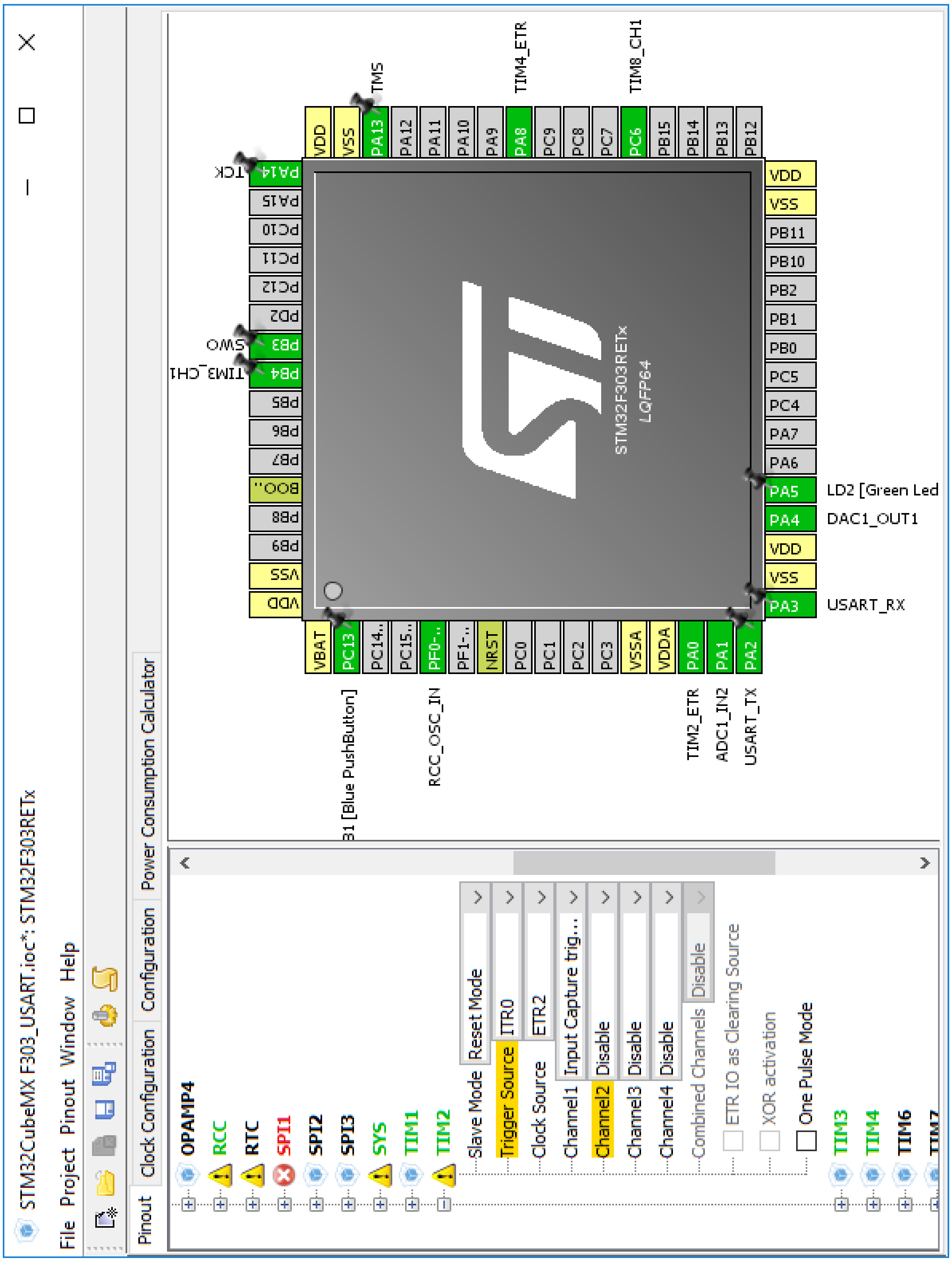
Peripheral Selection

	Peripherals	Nb	Max
🟢	ADC 12-bit	0	10
⚪	ADC 16-bit	0	0
🟢	CAN	0	1
🟢	COMP	0	2
🟢	DAC 12-bit	0	3
⚪	DCMI	☐	
⚪	DFSDM	☐	
⚪	DSIHOST	☐	
⚪	Ethernet	☐	
⚪	FMC	☐	
⚪	FMPI2C	☐	
⚪	FSMC	☐	
⚪	HDMI CEC	☐	
⚪	HRTIM	☐	
🟢	I2C	0	1
⚪	I2S	0	0
🟢	IRTIM	☐	
⚪	LPTIM	0	0
⚪	LPUART	☐	
🟢	OPAMP	0	1
⚪	QUADSPI	☐	
🟢	RTC	☐	
⚪	SAI	0	0
⚪	SDIO	☐	
⚪	SDMMC	☐	
⚪	SPDIFRX	☐	
🟢	SPI	0	1
⚪	SWPMI	☐	
⚪	Segment LCD	☐	
⚪	TFT LCD	☐	
🟢	Timer 16-bit	0	7
🟢	Timer 32-bit	0	1

MCUs List: 2 Items

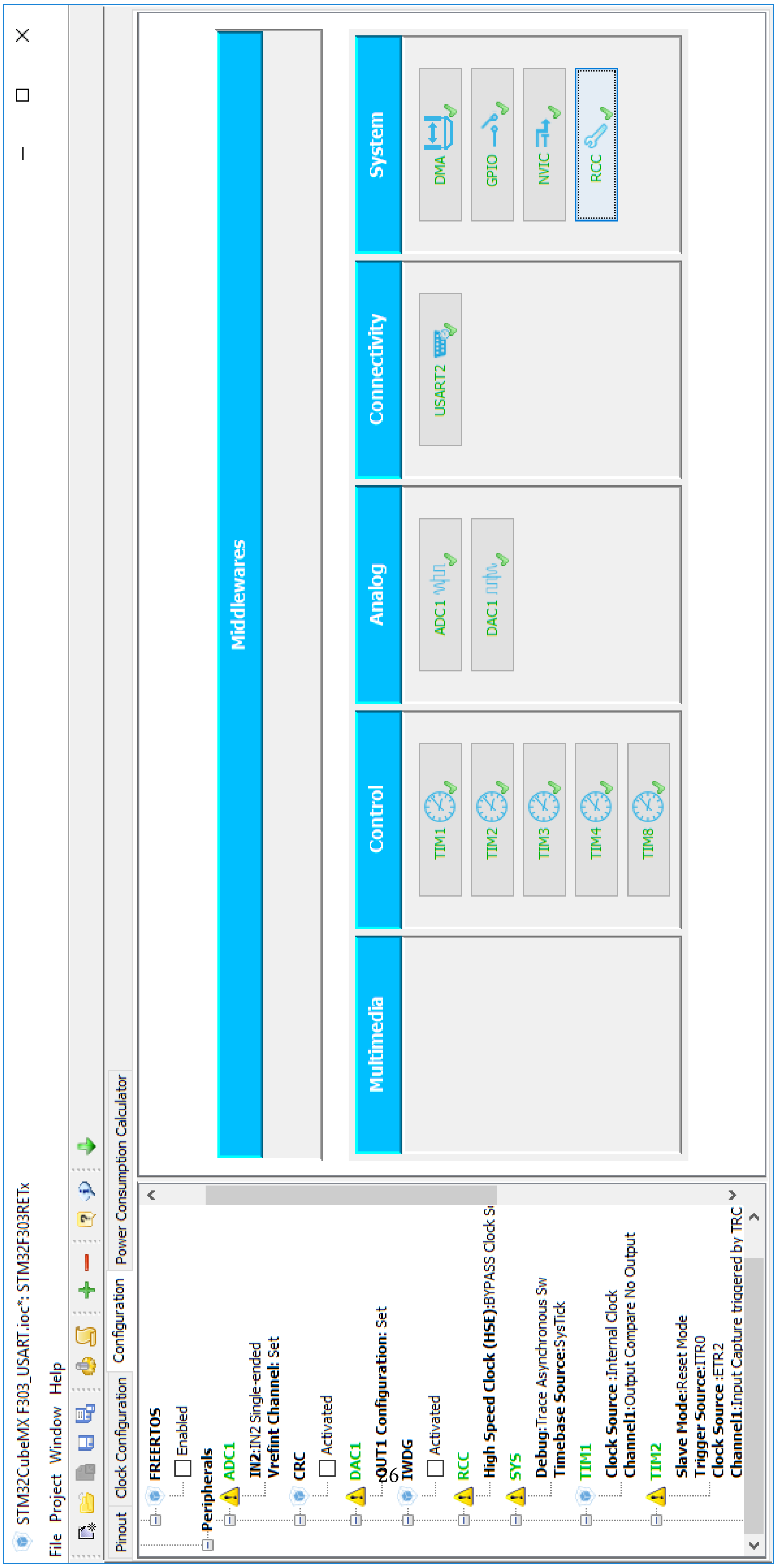
MCU	Lines	Package	Flash	Ram	Eeprom	IO
STM32F303K6Tx	STM32F303	LQFP32	32	16	0	25
STM32F303K8Tx	STM32F303	LQFP32	64	16	0	25

Figure 4.1.: CubeMX MCU/Board selection screen



Slave Mode: Reset Mode
Trigger Source: ITR0
Clock Source: ETR2
Channel1: Input Capture trig...
Channel2: Disable
Channel3: Disable
Channel4: Disable
Combined Channels: Disable
☐ ETR IO as Clearing Source
☐ XOR activation
☐ One Pulse Mode

Figure 4.2.: CubeMX pinout screen


STM32CubeMX F303_USART.ioc*: STM32F303RETx

File Project Window Help

Pinout | Clock Configuration | Configuration | Power Consumption Calculator

- FREERTOS
  - ☐ Enabled
- Peripherals
  - ADC1
    - **IN2:**IN2 Single-ended
    - **Vrefint Channel:** Set
  - CRC
    - ☐ Activated
  - DAC1
    - **OUT1 Configuration:** Set
  - IWDG
    - ☐ Activated
  - RCC
    - **High Speed Clock (HSE):**BYPASS Clock S
  - SYS
    - **Debug:**Trace Asynchronous Sw
    - **Timebase Source:**SysTick
  - TIM1
    - **Clock Source** :Internal Clock
    - **Channel1:**Output Compare No Output
  - TIM2
    - **Slave Mode:**Reset Mode
    - **Trigger Source:**ITR0
    - **Clock Source** :ETR2
    - **Channel1:**Input Capture triggered by TRC

Figure 4.3.: CubeMX configuration tab

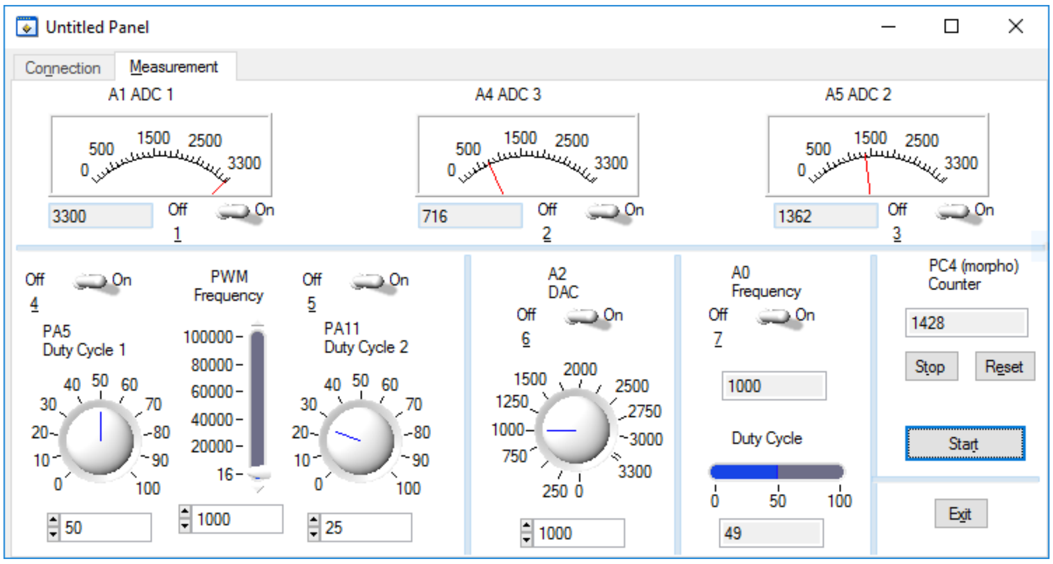


Figure 4.4.: Measurement frontend I implemented

The transmit and receive methods have a similar name convention (arguments are not included because they are too long):

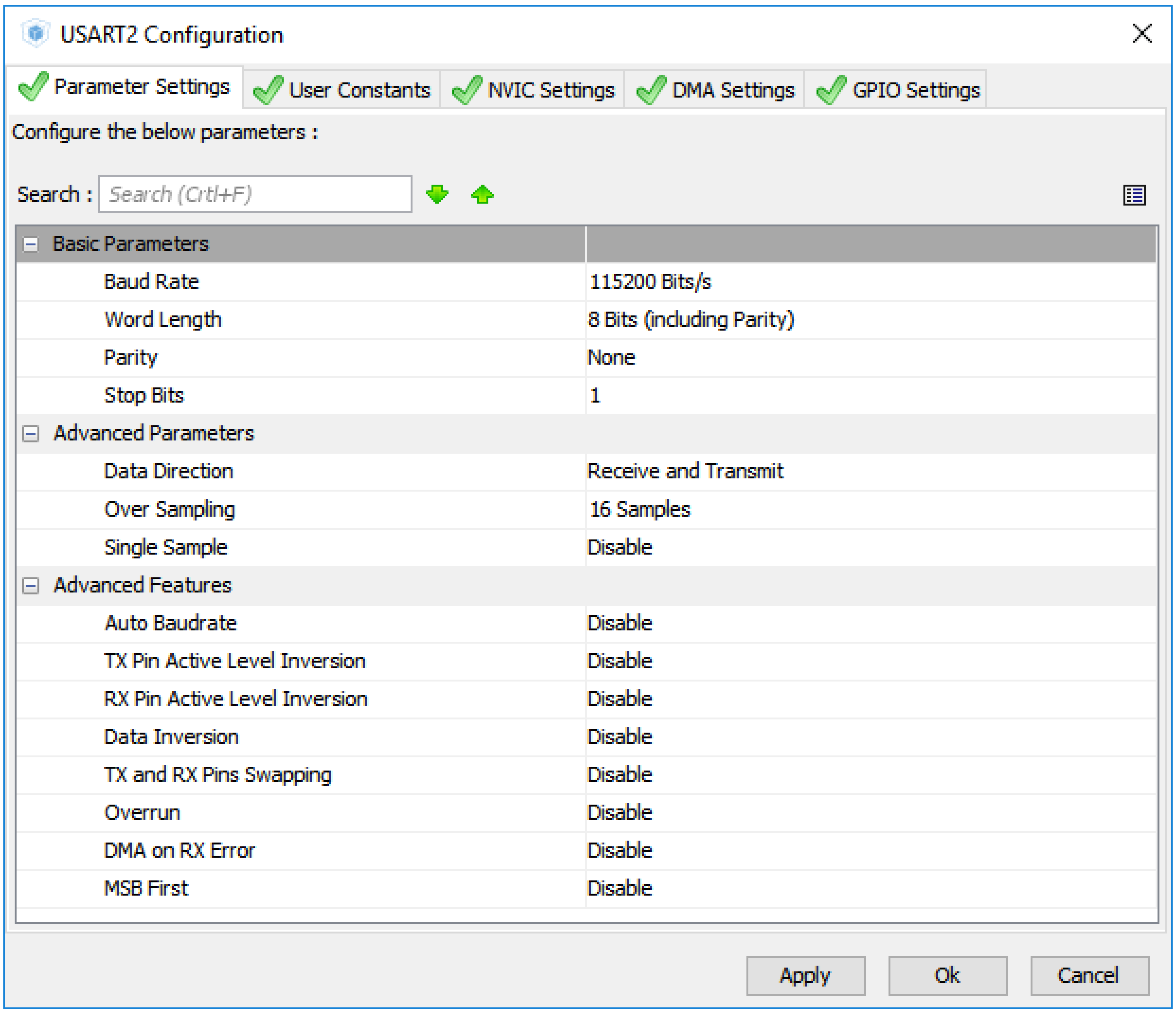
- `HAL_UART_Transmit(...)` and `HAL_UART_Receive(...)` sends and receives data in blocking mode,
- `HAL_UART_Transmit_IT(...)` and `HAL_UART_Receive_IT(...)` sends and receives data in nonblocking mode with interrupts, and
- `HAL_UART_Transmit_DMA(...)` and `HAL_UART_Receive_DMA(...)` sends and receives data in nonblocking mode with DMA.

The detail configuration in CubeMX can be seen in fig. 4.5. The basic parameters are set up the same way as described in mbed section 3.2.3. The advanced features shouldn't be enabled unless the user knows what they do and wants to use them.

In order to use the nonblocking mode with DMA, the DMA controller has to be set up. This can be done in the DMA Settings tab, by adding a channel for the required directions (USART2_TX for transmit and USART2_RX for receive). Since data is sent over UART byte by byte, the data width will be a single byte and to allow the DMA to write data to an array, memory address incrementation has to be enabled, but the mode has to be normal (not circular).

A simple example of working with the uart driver in blocking mode can be seen in alg. C.1

Unfortunately HAL is one of the frameworks where if you wish to do something different than what was intended you will have a very hard time. This is the case of controlling the instrument with commands that do not have uniform length, e.g. SCPI (all commands can have a different length). This is because HAL allows setting the number of characters to be received, but not the termination character, therefore one would have to build up the command string character by character, or bypass HAL altogether. Of course, if the command length is the same, this does not pose a problem and allows for a very simple implementation.



Basic Parameters	
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable
Advanced Features	
Auto Baudrate	Disable
TX Pin Active Level Inversion	Disable
RX Pin Active Level Inversion	Disable
Data Inversion	Disable
TX and RX Pins Swapping	Disable
Overrun	Disable
DMA on RX Error	Disable
MSB First	Disable

Figure 4.5.: Setting up the VCP UART in STM32CubeMX

4.3.2. Using the microcontroller itself as a UART to USB bridge

If an external UART to USB bridge is not available, like in the case of Discovery boards, or just processors by themselves, it is possible to implement the bridge on the microcontroller itself. Of course, this is only possible on those microcontrollers that have a USB interface (for example the STM32F042).

Implementing the USB to UART bridge with CubeMX is actually very easy, as CubeMX will generate all the necessary files into the project. The necessary steps in CubeMX:

1. in Peripherals -> USB enable Device,
2. in Configuration -> MiddleWares -> USB_DEVICE select Communication Device Class for FS IP,
3. generate the project.

In the projcet itself:

1. include usbd_cdc_if.h into main.c (generetad by CubeMX into Application/User),
2. **add a delay after the `MX_USB_DEVICE_Init()`**, I have found out that 500ms sufficies. You can use HAL_Delay(uint32_t ms). If you do not have a delay before starting the USB and trying to send data over it, your microcontroller will not be recognized by your PC!
3. You can use the `CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)` method to send data to the PC, and
4. to receive data you can update the `CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)` method in `usbd_cdc_if.c` to suit your needs. Personally I found the easiest send the `Buf` and `Len` to a new function in main.c. Also, do note that the `Buf` is not cleared, so you must use `Len`.

Don't get confused by the IN and OUT endpoints, they are actually reversed because the microcontroller works as a slave device. Therefore data is received through the OUT endpoint, and sent through the IN endpoint.

It is also possible to implement the VCP yourself from scratch, but that is out of the scope of this thesis. You can find some information at [2].

4.4. Static voltage measurement using Cube

As was already explained in chapter 2, static voltage measurement is one of the most common types of measurement. This section will examine the settings of the ADC and implementations of static voltage measurement.

In the STM32 line of microcontrollers we can usually find one to four 12 bit successive aproximation ADCs. Each of the ADCs has a relatively large number of input channels which are multiplexed onto the ADC input, as seen on fig. 4.6. On the STM32F303x microcontrollers, each ADC can sample with a rate of upto 5.1MSps, but this only interesting when measuring dynamic events for example with an oscilloscope.

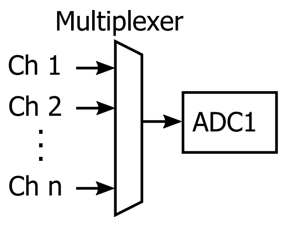


Figure 4.6.: ADC channel multiplexing

When the channels are multiplexed, the sampling speed per channel will decrease as the time is distributed between the channels. For static voltage measurement this doesn't pose a problem though.

Typically ADCs only have one data register to store the measured data, so after each conversion the measurement has to be read out of the data register. It is possible to read it in the main program loop, but that is wasteful and it is better to use the Direct Memory Access (DMA) controller. The DMA transfers data from the ADC to a user specified buffer behind the back of the processor, thus freeing it up.

To set the ADC in cube you will need to:

- enable the required channels in single or differential mode in the pinout tab (single mode measures the signal voltage against ground, differential against another signal voltage),

- in the parameter settings tab, depending on the microcontroller, set the number of regular conversions to the number of channels,
- set the sampling time (if you don't know, just select the highest),
- enable scan conversion mode (scan mode converts a single value from one channel, then moves to the next one) and continuous conversion mode.
- finally enable DMA continous requests and in the DMA Settings tab add a DMA channel in Circular mode (once the DMA reaches the end of the buffer, it starts filling the buffer again from the beginning).

After generating the project, you can use the `HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t* pData, uint32_t Length)` method to start he ADC and DMA. Unless you have a specific need, I highly suggest using the DMA as its very easy, and simplifies everything.

The limitations of the ADC, and their solutions are discussed in chapter 5.

4.5. Controlled voltage source

The controlled voltage is mostly the same as described mbed. Either the DAC, a PWM signal or the digital pins can be used. The main difference is that since the DAC peripheral can be controlled completely, it is possible to set whether the output buffer will be used. With the DMA controller the DAC can be used as an arbitrary function generator for example with Direct Digital Synthesis (DDS). The latter will not be discussed as it is out of the scope of this thesis.

The output buffer typically an opamp voltage follower which on one side allows higher current draws and therefore lower impedance loads, but on the other side doesn't allow rail to rail voltage. The limit is roughly 60mV from Gnd and 40mV from Vss. The effect of the buffer is discussed can be seen in section 5.5 on page 47.

With CubeMX the setup is almost trivial. The required DAC channel is enabled in the pinout tab, and if needed the the output buffer can be turned off in the Configuration tab under the DAC. After generating the project, the DAC is controlled with the

- `HAL_DAC_Start(DAC_HandleTypeDef* hdac, uint32_t Channel)` and the
- `HAL_DAC_SetValue(..., uint32_t alignment, uint32_t data)` methods.

Usually the alignment would be set to `DAC_ALIGN_8B_R` or `DAC_ALIGN12B_R` depeding on whether 8 bit or 12 bit resolution is required.

Do note that similarly to the ADC, the DAC output voltage is a ratio of $V\text{ref}_{+} - V\text{ref}_{-}$, and unless connected to a calibrated source, $V\text{ref}_{+}$is connected to V_{cc} i.e. the power supply of the microcontroller, and $V\text{ref}_{-}$is connected to the ground. Therefore to get a precise output, the DAC has to be adjusted for the $V\text{ref}_{+}$and $V\text{ref}_{-}$. This can be done directly by measuring the DAC output with the ADC (which has been adjusted using the internal reference 5.4), or indirectly by measuring the supply voltage on the ADC and then adjusting the DAC output in both cases.

4.6. Simple pulse counter

The pulse counter has already been described in chapters 2 and 3. Thanks to the fact that Cube is designed to allow access to all features and peripherals of the microcontroller, it is possible to use the timer peripheral to count pulses.

A huge leap in performance over mbed is that the maximal counter speed is limited only by its clock speed. For example in the F303 series, this is upto 144 MHz on selected timers and 72 MHz on the others. This is 3.5 orders of magnitude faster than mbed which maxed out at 200 kHz. This is not very interesting for the pulse counter itself, but it is a key aspect for direct frequency measurement.

To setup the timer in counter mode, in CubeMX select any timer that has a dropdown for the clock source, and select **ETR2** (External Clock Mode 2).

You can find out more about the clock modes in the reference manual. CubeMX will automatically assign a GPIO pin and set it up. In the Configuration tab, select the corresponding timer e.g. TIM2, and in the parameter settings tab of the Configuration window, set the counter period to the maxmimum possible ($0xFFFF$ for most counters, $0xFFFFFFFF$ for TIM2).

When the counter reaches the counter period (autoreload), it will reset its count to 0, which means that **if the counter period is left at 0, the counter will not count.** Usually it is desirable to set the maximal possible value, but there are cases, i.e. if we want to count an exact number of pulses. In the latter case, the counter period is set to the required pulse count -1, and the overflow interrupt is enabled. Once the required pulse count is reached, an interrupt is generated and further actions can be taken.

After generating the project, the timer must be started by `HAL_TIM_Base_Start(TIM_HandleTypeDef *)` where `TIM_HandleTypeDef *` is the pointer to the handle of the counting timer e.g. `&htim2`. The counter can be read with `__HAL_TIM_GetCounter(TIM_HandleTypeDef *)`.

4.7. Direct frequecy measurement

Direct frequency measurement is the perfect choice when measuring very high frequencies. This is because the resolution of DFM is ±1 count of the measured signal, and it can be setup in a manner such that the only limit is the maximum input speed of the GPIO pins, for example 72 MHz in the case of the F303. Dedicated DFM units would usually have a counter which is gated for a decimal multiple of one second, thus the frequency could be displayed directly from the counter. Since microcontrollers have a lot of processing power though, it is possible to use an arbitarary gating period and calculate the frequency.

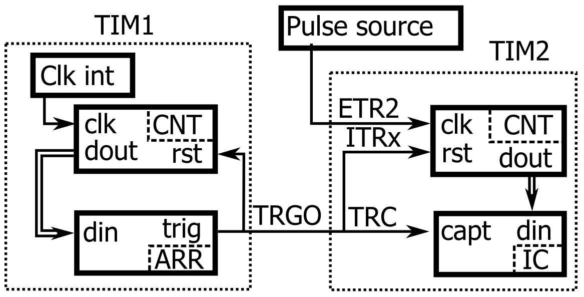


Figure 4.7.: Setup of DFM with 2 timers

Unfortunately the F303 and F042 both have only one 32 bit timer, and that is also the case of many other microcontrollers. Since DFM allows the highest precision and a range from kilohertz to tens of megahertz, as opposed to RFM which is usable only to 100 kHz, it is highly advisable to use the 32 bit timer for DFM.

There are two ways to implement DFM, in both cases a pulse counter is used (such as the one described in the previous section) and it is gated either with another hardware timer, or with software interrupts. Out of all measurements that can be done, frequency measurement can be done with the highest precision, therefore it makes sense to use two hardware timers and achieve the highest precision. Interrupts do not have an exact time of execution, but rather a range during which it will be executed. This is most problematic in interrupt heavy programs and is a source of significant error. Furthermore, an external crystal should be used (HSE) because it can be a few orders of magnituted more precise than the internal (HSI).

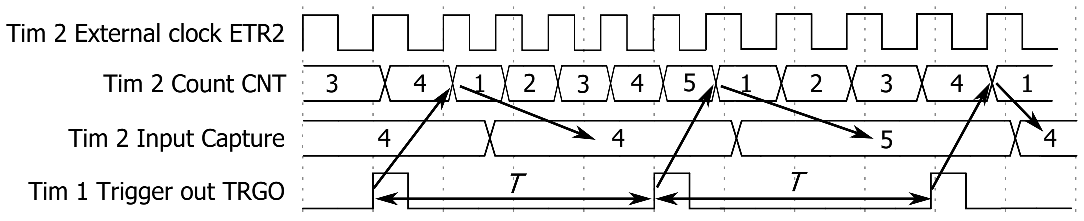


Figure 4.8.: Timing diagram of DFM with 2 timers

The timers on the STM32 microcontrollers can be gated by other timers in slave mode. The best slave mode for this purpose is Reset mode, as instead of starting and stopping the timer (Gated mode), it starts the timer and also resets it to zero in every gate. If the timer won't stop after every capture, an input capture unit needs to be used to store the frequency. A simplified diagram of this setup can be seen in fig.4.7, where CNT, ARR and IC are the counters, autoreload unit (counter period) and input capture unit, and rst are the reset ports, din/dout are the data ports and trig is the trigger output from the gating timer.

To set up the pulse counter, for example the 32bit TIM2:

1. set clock mode to ETR2,
2. set slave mode to Reset,
3. select the appropriate ITRx[3] for the master trigger in,
4. enable Input Capture triggered by TRC ,
5. in the configuration, set the counter period to maximum .

To set up the gating timer, e.g. TIM1 if ITR0 was selected:

1. select the internal clock as clock source,
2. enable Output Compare No Output on channel 1,
3. in the configuration, select Comparue Pulse (OC1) as the Trigger Event Selection TRGO in Trigger Output Parameters,
4. set the prescaler and counter period .

[3]This can be found in the TIMx internal trigger connection in the reference manual . In the example code I control TIM2 form TIM1, therefore ITR0 is selected.

The gating frequency f_g is calculated with equation 4.1. For example, if the timer clock speed f_{cs} is 72 MHz, and we wish to gate every second, prescaler PSC can be set to 719 and the counter period T_{Cnt} to 9999. When setting the parameters, do hold in mind that the values have to be 16bit.

$$f_g = \frac{f_{cs}}{(\mathrm{PSC} + 1) \cdot (T_{Cnt} + 1)} \tag{4.1}$$

To **enable the HSE crystal**, expand the RCC menu in the Pinout tab and select HSE->Bypass if the microcontroller doesn't have its own crystal, such as the nucleo series. If there is a dedicated crystal or ceramic resonator select HSE->Crystal/Resonator. You can set the HSE frequency in the Clock Configuration tab. **The bypass HSE doesn't have to be available even if a crystal is present**, so consult the development board user manual to see which solder bridges to update.

After generating the project, the pulse counter needs to be started in input capture mode i.e.

- `HAL_TIM_Base_Start(TIM_HandleTypeDef *htim)`, and
- `HAL_TIM_IC_Start(TIM_HandleTypeDef *htim, uint32_t Channel)` methods must be called.

The possible arguments for the Channel can be found in the method documentation. Since it is on channel 1, it will be `TIM_CHANNEL_1`.

Finally, the frequency can be read from the input capture unit using the `HAL_TIM_ReadCapturedValue(TIM_HandleTypeDef *htim, uint32_t Channel)`method.

Important: many signal generators have a zero offset by default, seen on the left side in fig. 4.9. This means that the output voltages is negative for half the time, and as was explained in chapter 4, this can destroy the microcontroller. For example a sine with an amplitude of 1V and offset of 0 will generate voltages from −1 V to 1 V, however by setting the offset equal to the amplitude the signal will not enter the negative region. This can be seen on the right side of fig. 4.9, where the both the offset and the ampluted are 1, and the signal is only positive.

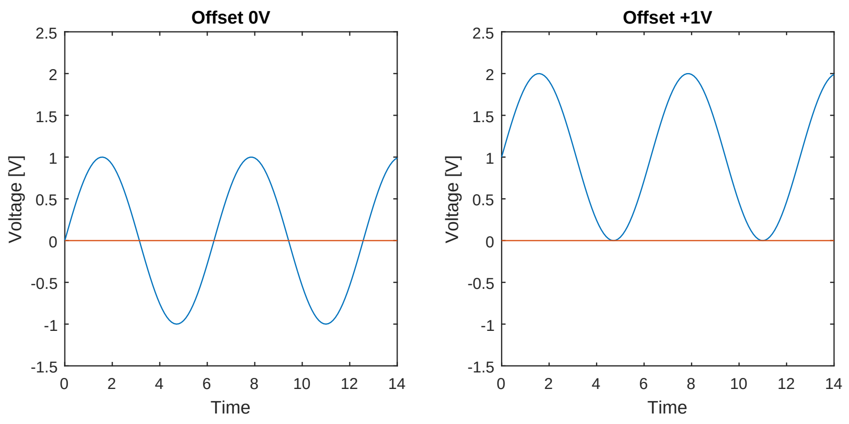


Figure 4.9.: Offset in function generators, amplitude 1V

4.8. Reciprocal frequency measurement

As opposed to DFM, RFM si better suited for measuring lower frequencies. This is because the resolution is constant - ±1 clock cycle of the internal clock, therefore the lower the frequency, the higher the precision.

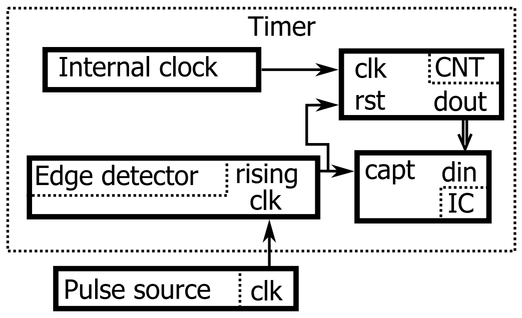


Figure 4.10.: Setup of RFM with a single timer

On microcontrollers, RFM can be implemented using timers very similarly to DFM. The main difference is that the clock source and input capture trigger are switched, i.e. the internal clock is used as the clock source, and the input capture is triggered by the rising or falling edges of the external pulse. However, since this approach only stores the relative time, i.e. how long the period was since the last input capture, it is better to set the timer in reset slave mode. This mode resets the counter everytime a rising or falling edge is deteceted, therefore the absolute length of the period is stored. The setup of this mode can be seen in fig. 4.10.

The measurable frequency range completely depends on the selected clock frequency and counter size. The clock frequency sets the maximum measurable frequency and the conuter size the lowest. For example if a clock frequency of 1MHz is selected, and the counter is 16 bit, using equation 4.2 we get 15 Hz. However, at frequencies higher than 10 kHz the error is 1 %, and over 100 kHz it is 10 % (since at 10 kHz 100 pulses are measured and at 100 kHz only 10 pulses are measured).

$$f_{lowest} = \frac{f_{clk}}{counter\_size} \tag{4.2}$$

FM calculates the frequency from a single period, and the measurement instrument would typically display the frequency at a very low refresh rate (units of Hertz). In the case of a signal that is not perfectly periodical, the displayed value can be strongly distorted. There are two ways that this could be solved:

1. Divide the incoming clock. This is the hardware equivalent of averaging, as each division increases the number of periods which the input capture captures.

2. Average the value. Of course, to keep a usable upper frequency limit this has to be done using the DMA controller to move captured periods into an intermediate buffer and process it on demand with the main processor. Unfortunately with the current version of CubeMX, when using the PWMInput mode it is only possible to use the rising edge IC with the DMA. As a workaround it is possible to either set the timer manually, or add the dma channel in code.

4.9. Duty cycle measurement

Duty cycle measurement is very closely related to RFM, as it is measures 2 periods instead of one. The concept was explained well enough in chapter 2 so it will not be discussed further.

This is a rather common measurement, but setting it up when one doesn't know exactly what he is doing can be rather problematic. Fortunately HAL has a function for this - `PWMInput`.

It sets the timer up in reset slave mode, with 2 input capture units. The reset trigger and one input capture is set on the rising edge of the measured signal, and the second input capture unit is set to the falling edge of the signal. With the values from the input capture units the frequency f can be calculated using equation 4.3, where $IC_{\text{fall}}$is the value stored in the IC unit set to the falling edge and IC_{rise} is the value stored in the IC unit set to the rising edge.

$$f = \frac{1}{IC_{\text{rise}}}, \%DC = \frac{IC_{\text{fall}}}{IC_{\text{rise}}} \tag{4.3}$$

To set a timer in `PWMInput` mode in CubeMX, in the Pinout screen, select a timer and in the Combined Channels dropdown list select PWM Input on Channel x.

Then go to the Configuration tab, select the timer, and set the counter period to 0xffff (or 0xffffffff if a 32 bit timer is used).

Once the project is generated, start the timer and enable the input capture with the

- `HAL_TIM_Base_Start(TIM_HandleTypeDef *handle)`, and

- `HAL_TIM_IC_Start(TIM_HandleTypeDef *tim, uint32_t Channel)` methods.

The IC start has to be called on both channels. Finally the periods can be read from the input capture with the `uint32_t HAL_TIM_ReadCapturedValue(TIM_HandleTypeDef *tim, uint32_t Channel)` method.

4.10. PWM Signal generation

The uses of PWM signals was explained in previous chapters. Using them with cube is simple, but may be a bit confusing when done for the first time.

After enabling a timer in PWM Output mode from the pinout tab, the following parameters must be set (in the configuration tab or in code):

- counter period - it sets how long one full period is. By substituting the required pwm frequency f_{pwm}, prescaler PSC and timer clock frequency f_{cs} into equation 4.4 the counter period T_{cnt} can be calculated,

- pulse - it is used to set the duty cycle. The duty cycle $\%DC$ is calculated with eq. 4.5, where T_{pw} is the pulse,

$$T_{\text{cnt}} = \frac{f_{cs}}{f_{\text{pwm}} \cdot (PSC + 1)} \tag{4.4}$$

$$\%DC = \frac{(T_{\text{pw}} + 1)}{(T_{\text{cnt}} + 1)} \cdot 100 \tag{4.5}$$

- optionally, the prescaler can be set. It is used when lower frequencies are generated, but only a 16bit timer is used.

For example, the STM32F303RE has a base clock of 72 MHz, and we wish to generate a 1 kHz PWM signal.. If a 16 bit timer is used, it is impossible to set the required frequency with the counter period alone (since it would have to be 71999, but the highest number in 16 bits is 65535). Therefore the prescaler has to be used. I find it convenient to prescale the clock to 1 MHz, therefore $PSC = 71$, since $\frac{72}{71+1}$ MHz = 1 MHz and the counter period has to divide this 1 MHz to 1 kHz i.e. $T_{\text{cnt}} = 999$. Finally, to get a 50 % duty cycle, set $T_{\text{pw}} = 499$.

4.11. Oscilloscope

All the measurement instruments described in the previous sections output a single number to represent the measured signal. In many cases though, it is also important to see how the signal itself looks like. An oscilloscope is a device that does exactly this: it plots the measured signal against time.

On the microcontroller side, the implementation is actually rather simple, and very similar to static voltage measurement with averaging. Typically DMA will be used to copy data from the ADC to a buffer, but instead of averaging the data in the buffer, it is sent to the PC to be displayed.

Furthermore higher sampling speeds are used, as it is desirable to see fast signals, and also some kind of trigger is required. "The trigger makes repetitive waveforms appear static on the oscilloscope display by repeatedly displaying the same portion of the input signal. Imagine the jumble on the screen that would result if each sweep started at a different place on the signal!"[22]

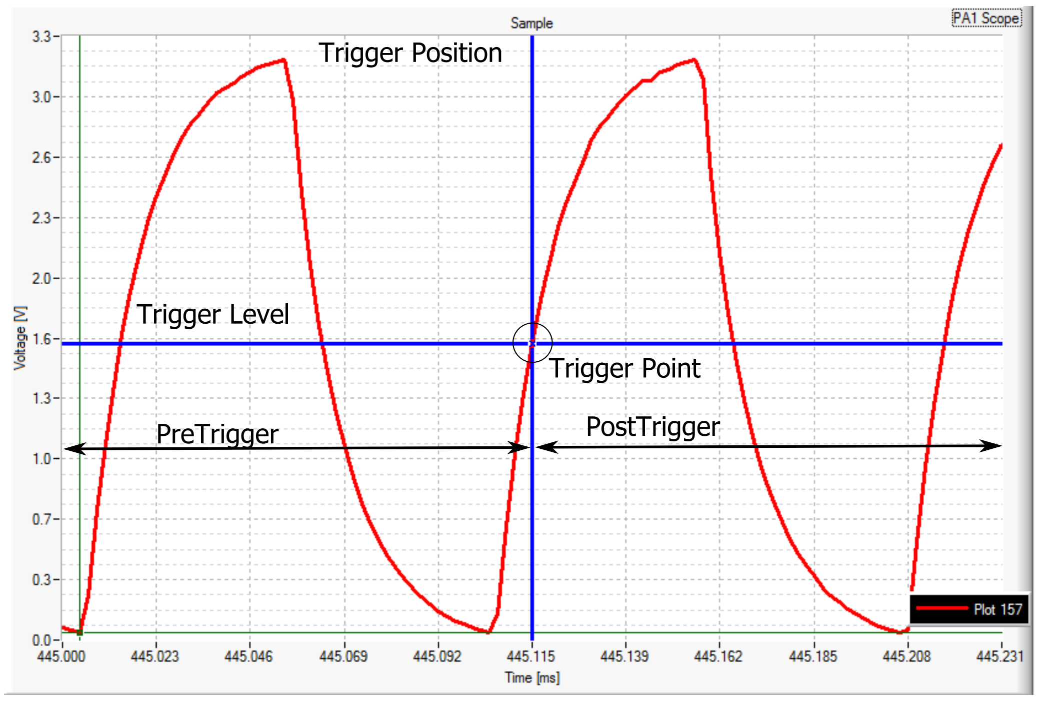


Figure 4.11.: Rising edge oscilloscope triggering

The most basic trigger consists of three parameters:

- trigger edge, usually rising, or falling (the signal has to be greater, or lower than the trigger level),
- trigger level i.e. the threshold the signal has to exceed to cause the triggering, and
- trigger position (horizontal).

The most basic setting of the trigger is 50% trigger level, rising edge and middle trigger position, as seen in fig. 4.11. This setting allows one to see what happened before and after the trigger and is sufficient for many signals.

The triggering method I implemented is a slight variation of the one explained by J. Hladik[6]. In his method, the DMA copies data from the ADC to a circular buffer, and the microcontroller reads the last value written by the buffer i_{DMA}. If the trigger conditions are met, the DMA is set to transfer additional samples to satisfy the trigger position.

Since I only implemented a half full trigger with unadjustable position or level, the triggering method could be simplified. As seen in fig. 4.12, instead of reading the current buffer value i_{DMA}, the value "opposite" of it i_{trig} is read (i.e. half of the length of the buffer before the last DMA write). If the trigger conditions are met, the DMA and ADC are stopped, and the data is sent over to the PC.

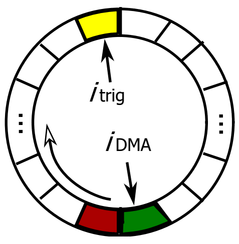


Figure 4.12.: Circular buffer triggering

Of course, if i_{trig} is calculated simply (half the buffer length before i_{DMA}), depending on the speed of the microcontroller and DMA transfers, i_{trig} will lag behind the actual index, and thus the exact trigger position needs to be found. For simplicity, I do this in the PC frontend by iterating backwards from the middle of the buffer and looking for the first sample that does not satisfy the trigger conditions.

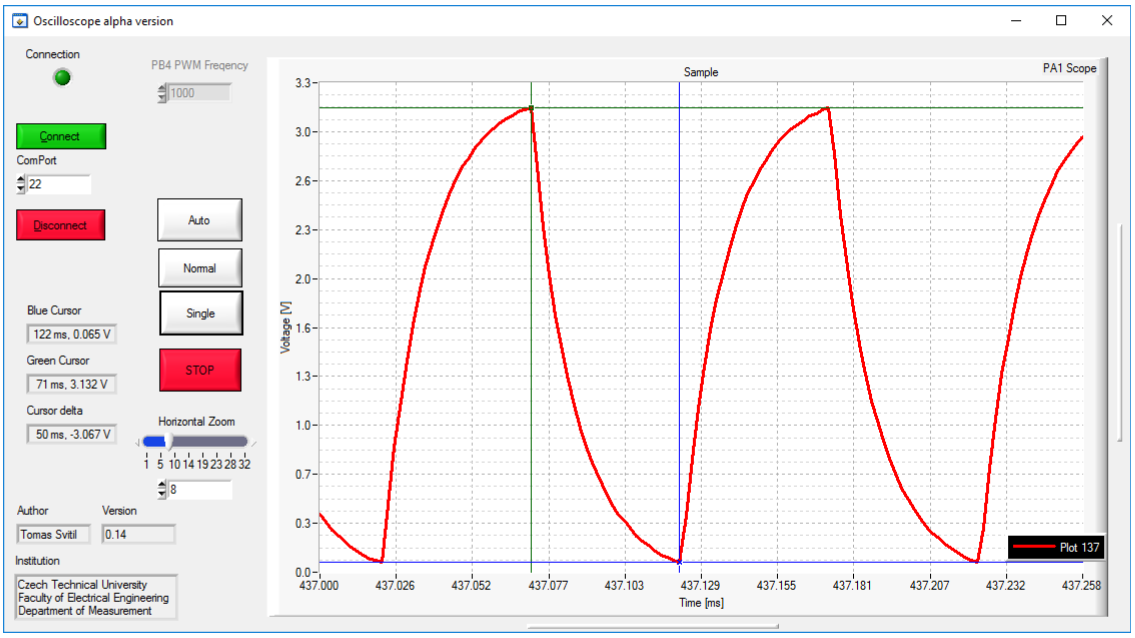
The oscilloscope is slightly harder to implement than the other functions, because it requires a PC frontend. This means that the developer either has to have a complete frontend to which he'll only send data, or he'll have to create one. I have created a frontend with Labwindows CVI as seen in fig. 4.13

Note, to create an 'analog' signal on the microcontroller, simply add a low pass filter (RC circuit) after a PWM signal. To get nice exponentials, the RC time constant τ should be roughly the same as period of the pwm signal i.e. $R \cdot C = \tau = T_{\mathrm{pwm}}$. For example if the PWM signal has a frequency of 10 kHz, the period $T_{\mathrm{pwm}}$is 100 μs, thus a 1 kΩ resistor and 100 nFcapacitor could be used. The LP filter would be connected like in fig. 3.8 and the V_{out} could be connected to the oscilloscope. This is exactly how the signal in figs. 4.11 and 4.13 was generated.

4.12. Code portability with Cube

One of the main benefits of Cube is that code is portable between different microcontrollers and even microcontroller series. This is achieved with the HAL libraries which have the same[4] API for all series.

[4]There are some differences, for example the F411 doesn't support `hal_adcex_calibrate`, and sometimes the naming conventions differ e.g. `hadc` and `hadc1`.



PB4 PWM Freqency: 1000
ComPort: 22
Blue Cursor: 122 ms, 0.065 V
Green Cursor: 71 ms, 3.132 V
Cursor delta: 50 ms, -3.067 V
Horizontal Zoom: 8
Author: Tomas Svitil
Version: 0.14
Institution: Czech Technical University Faculty of Electrical Engineering Department of Measurement

Figure 4.13.: Oscilloscope frontend

Unfortunately CubeMX doesn't support migrating projects from one microcontroller series to another, but this is understandable since the peripherals are different. It only allows for migration between microcontrollers of the same series. This means that the peripherals have to be set again for the new series in CubeMX, and code has to be manually copied and pasted in the C code. Of course since the series can have differently numbered peripherals, this has to be reflected in the copied code.

To demonstrate this, I have created a simple program which uses the measures voltage on 1 channel, frequnecy, generates a PWM signal, and a voltage on the DAC if applicable, and is controlled over UART. The voltage is corrected for gain (i.e. the input voltage is measured, and the internal reference channel is measured together using DMA), and averaged. Frequency is measured with DFM on 32 bit timers where available.

The programs have been ported from the F303 to the F411, L053 and later to the F042. Porting to the first two took me roughly 4 hours with all debugging (mainly because of silly mistakes), however porting to the F042 took me only 30 minutes (it was done another day).

I had to consult the datasheet and reference manual to find the type and memory address of the adc voltage reference, and to find the 32 bit timers (which were not available in the L053) in the microcontrollers. The address is important, as accessing an 'incorrect' address will case a hardfault.

Important: the st-link master clock out, or MCO (HSE bypass), does not have to be connected to the main microcontroller. You should consult the development board manual to find if and which solder bridges should be soldered or removed in order to connect the MCO.

CHAPTER

FIVE

MEASUREMENT LIMITATIONS IN F3XX AND F0XX SERIES MICROCONTROLLERS

In chapter 2, limitations of the ADC and timer were listed. This chapter will explore those limitations in more detail and offer some solutions to them. Application notes [14, 17] have been used as a starting point for this chapter.

5.1. ADC Input Signal Shaping

Most, if not all microcontrollers, are created with CMOS technology. As seen on fig. 5.1, there diodes D1 and D2 connect the input pin to Vcc and Gnd. If the voltage Vcin is higher than the power supply Vcc, current will flow through D1 (red arrow) and similarly if Vcin is lower than Ground current will flow through the D2 into the input (blue arrow).[1] For this reason the input voltage cannot exceed the supply voltages of the CMOS logic, which are 3.3V and 0V in the case of STM32F3xx and STM32F0xx microcontrollers. Note that diode D1 isn't always in there, typically 5V tolerant pins would not have it and the protection is handled in another way.

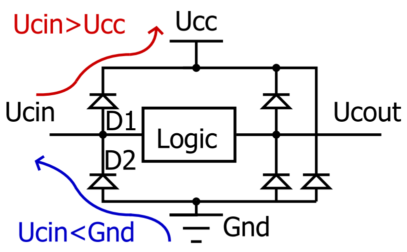


Figure 5.1.: CMOS input diodes

However, there are times when this voltage range will not be sufficient. For example when we wish to work with TTL logic levels or equivalent (0 V to 5 V), CD series logic (0 V to 15 V), or operational amplifiers (−15 V to 15 V). External circuitry has to be used to shape the signal.

There are 2 approaches that can be taken:

1. A resistor divider can be used. A 2 resistor divider will divide the voltage, and a 3 resistor divider will also shift it, which allows us to connect negative voltages.The main benefit of using resistor dividers is that it they only require a few resistors. They will however decrease the input impedance of the "voltmeter".

2. An opamp can be used. It can be used to divide the range in the basic noninverting configuration and shift it by adding a few mor resistors. The main benefit of using opamps

[1]This also means that supply voltage applied to any CMOS devices must never be inverted, as that would destroy the chip.

is that the input impedance will be very high (theoretically infinite). This approach however requires more components.

You can find more information about using the opamp to also shift voltage at [12].

5.1.1. Resistor voltage divider

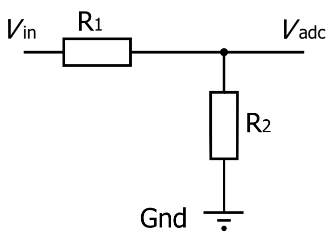


Figure 5.2.: Simple resistor voltage divider

For asymetric voltages (those that start at 0V), the simplest solution is to use a resistor voltage divider, seen on fig. 5.2. The calculation of output voltage can be seen on eq. 5.1.

$$V_{adc} = \frac{R_2}{R_1 + R_2} \cdot V_{in} \tag{5.1}$$

For example, if we would like to measure TTL logic - 0 V to 5 V, it is a good idea to have the limit levels higher than the actual maximum measured voltage, so for simplicity we choose a voltage of 6.6V. Therefore we need a ratio $V_{\text{adc}} = \frac{1}{2} \cdot V_{\text{in}}$ and thus $R_2 = R_1$. We can choose a resistance of 1 kΩ for both resistors.

5.1.2. Opamp in noninverting configuration

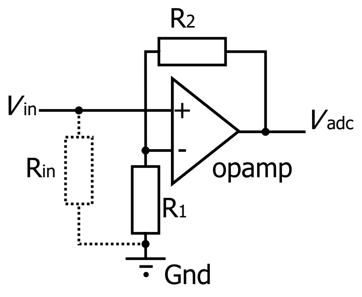


Figure 5.3.: Noninverting opamp configuration

In some cases it is desirable to have very high input impedance of the divider, so that very little current passes through it. An op amp in noninverting configuration, seen on fig. 5.3, has a very high input impedance.

With an ideal opamp, the input impedance of this configuration would be infinite, however because ideal op amps do not exist, the input impedance is as high as the internal resistance of the opamp. This is typically more than a megaohm. If the optional R_{in} is connected, the input impedance is R_{in}. R_{in} is connected only so that an output voltage is defined if no input is connectedf to V_{in}.

The voltage on the output of the divider can be calculated with equation **5.2.**

$$V_{\text{adc}} = (1 + \frac{R_2}{R_1}) \cdot V_{\text{in}} \tag{5.2}$$

5.1.3. Shifting resistor voltage divider

If we wish to measure a negative voltage, a 3 resistor divider as seen on fig. 5.4 can be used. The V_{in} to V_{adc} transformation can be calculated with node voltages, the result can be seen in equation 5.3. The whole calculalation follows:

$$\frac{V_{\text{adc}} - V_{\text{cc}}}{R_{s1}} + \frac{V_{\text{adc}} - V_{\text{in}}}{R_{\text{in}}} + \frac{V_{\text{adc}}}{R_{S2}} = 0$$

$$V_{\text{adc}} \cdot (\frac{1}{R_{s1}} + \frac{1}{R_{s2}} + \frac{1}{R_{\text{in}}}) = \frac{V_{\text{in}}}{R_{\text{in}}} + \frac{V_{cc}}{R_{s1}}$$

This is however very impractical for calculations, so let us use substitutions $R_{\text{in}} = R$, $R_{s1} = \frac{c}{R}$ and $R_{s2} = \frac{g}{R}$, where c and g are constants. With this the equation can be simplified to:

$$V_{\text{adc}} \cdot (\frac{c+g+1}{R}) = \frac{V_{\text{in}}}{R} + c \cdot \frac{V_{\text{cc}}}{R}$$

$$V_{\text{adc}} = \frac{V_{\text{in}} + c \cdot V_{\text{cc}}}{c+g+1} \tag{5.3}$$

We now substitute the required minimum and maximum voltages for V_{in}, the required transformed ADC voltage V_{adc} and supply voltage for V_{cc} into equation 5.3. With that we get a system of two equations with two variables, which we can solve. To find the required resistors, a value for Rin is selected, e.g. $10\,\text{k}\Omega$ and the other voltages are calculated as $R_{s1} = \frac{R_{in}}{c}$ and $R_{s2} = \frac{R_{in}}{g}$.

As an example, let us calculate the values required for an opamp with output voltages of $V_{\text{inmin}} = -15\,\text{V}$ which will be transformed to $V_{\text{adcmin}} = 0\,\text{V}$, $V_{\text{inmax}} = 15\,\text{V}$ which will be transformed to $V_{\text{adcmax}} = 3.3\,\text{V}$ and the supply voltage is 3.3 V.

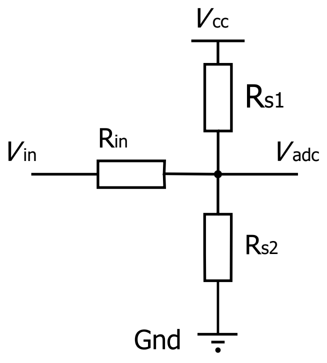


Figure 5.4.: Shifting resistor voltage divider

$$V_{\text{adcmin}} = \frac{V_{\text{inmin}} + c \cdot V_{\text{cc}}}{c+g+1}, \quad V_{\text{adcmax}} = \frac{V_{\text{inmax}} + c \cdot V_{\text{cc}}}{c+g+1}$$

$$0 = \frac{-15 + c \cdot 3.3}{c+g+1}, \quad 3.3 = \frac{15 + c \cdot 3.3}{c+g+1}$$

$$u \cong 4.54, \quad g \cong \quad 3.55$$

Assuming we select $R_{in} = 10\,\text{k}\Omega$, then $R_{s1} = \frac{10000}{4.54} = 2.2\,\text{k}\Omega$ and $R_{s2} = \frac{10000}{3.55} = 2.8\,\text{k}\Omega$. Do note, that this means that upto the device will source[2]$\sim$ 1.2 mA and sink upto 1.5 mA into the device. The sourced and sinked current can be calculated with equation 5.4.

$$I_{\text{sunk}} = \frac{V_{\text{inmin}} - V_{\text{adcmin}}}{R_{\text{in}}}, \quad I_{\text{sourced}} = \frac{V_{\text{inmax}} - V_{\text{adcmax}}}{R_{\text{in}}} \tag{5.4}$$

5.2. 50/60Hz interference handling (rejection)

In an ideal world, the voltage measured by the ADC could be taken as it is. In the real world though, noise and interference is omnipresent. This section will examine a way of correcting 50/60Hz power line interference[3] appearing on the input.

For any sinusoid $V \cdot sin(\omega \cdot t)$, equation 5.5, i.e. the integral of one whole period is equal to zero holds true. Since the 50/60 Hz interference can be modeled as $V_{\text{if}} \cdot sin(\omega \cdot t)$, and it superimposes itself on the actual voltage V_{act} i.e. $V(t) = V_{\text{act}} + V_{\text{if}} \cdot sin(\omega \cdot t)$, equations 5.6

[2]Current flows from devices that source it and into devices that sink it.

[3]An easy way to see the 50/60Hz interference is to touch a oscilloscope probe with hands and set the time division to 10ms/div. The interferring sinusoid will appear on the screen.

and 5.7 shows that the interference can be removed by integrating over whole periods of the interference signal. This can be done in hardware with an integrating ADC by setting its integration period to a whole multiple of the interference period.

$$\int_0^T V \cdot sin(\omega \cdot t)dt = 0 \tag{5.5}$$

$$\frac{1}{T}\int_0^T (V + V_{\text{if}} \cdot sin(\omega \cdot t))dt = \frac{1}{T}\int_0^T V dt + \frac{1}{T}\int_0^T V_{\text{if}} \cdot sin(\omega \cdot t)dt \tag{5.6}$$

$$= \frac{1}{T}\int_o^T V \cdot dt = \frac{1}{T}[V]_0^T = \frac{1}{T}V \cdot T = V \tag{5.7}$$

Many microcontrollers do not have integrating ADCs though, and the correction has to be done after sampling. As the discrete equivalent of integration is summation, the continuous time equation 5.6 can be converted into discrete time as shown in equation 5.8 (assuming that the sampling will be at a constant rate). The sum over T samples divided by T is the average of all the samples. Thus the 50/60Hz interference can be removed by averaging over a sample set which contains whole periods of the interference. The calculation for 50Hz can be seen in equation 5.9.

$$\frac{1}{T}\sum_{k=0}^{T}(V + V_{if} \cdot sin(\omega \cdot k) = \frac{1}{T}\sum_{k=0}^{T} V + \frac{1}{T} \cdot V_{if} \cdot \sum_{k=0}^{T} sin(\omega \cdot k) = \frac{1}{T} \cdot V \cdot T = V \tag{5.8}$$

$$n \cdot T_{sampling} = n \cdot \frac{1}{f_{sampling}} \rightarrow T_{sampling} = n \cdot \frac{1}{50} = n \cdot 0.02s \tag{5.9}$$

5.3. Noise handling

As mentioned in the previous section, noise and intereference is everywhere. In many cases this noise isn't periodic like the 50/60Hz intereference described in the previous section, but it is random.

The best case would be that the noise would be completely random, because as such it could be easily eliminated by averaging from a large sample set. This is not the case, as the autocorrelation function seen in fig. 5.6 shows a relatively strong correlation every 18.2ms at 1V (1055 samples at a 58kHz sampling rate) and 15.7ms (910 samples) for 2V and 3V. All samples were taken directly from a 10kS ADC buffer, which was serviced by the DMA controller.

Figure 5.5 shows the histograms of the measured data. A possible explanation of the fact that the data is distributed in two hills around the mean rather than a single hill is that there is alternating interference, such as the 50/60Hz. The means and relative errors can be seen in table 5.1.

Expected voltage [mV]	1000	2000	3000
Mean voltage [mV]	999.58	2001.67	3003.13
Relative error [%]	0.042	0.083	0.10

Table 5.1.: ADC Averaged measurements and relative errors

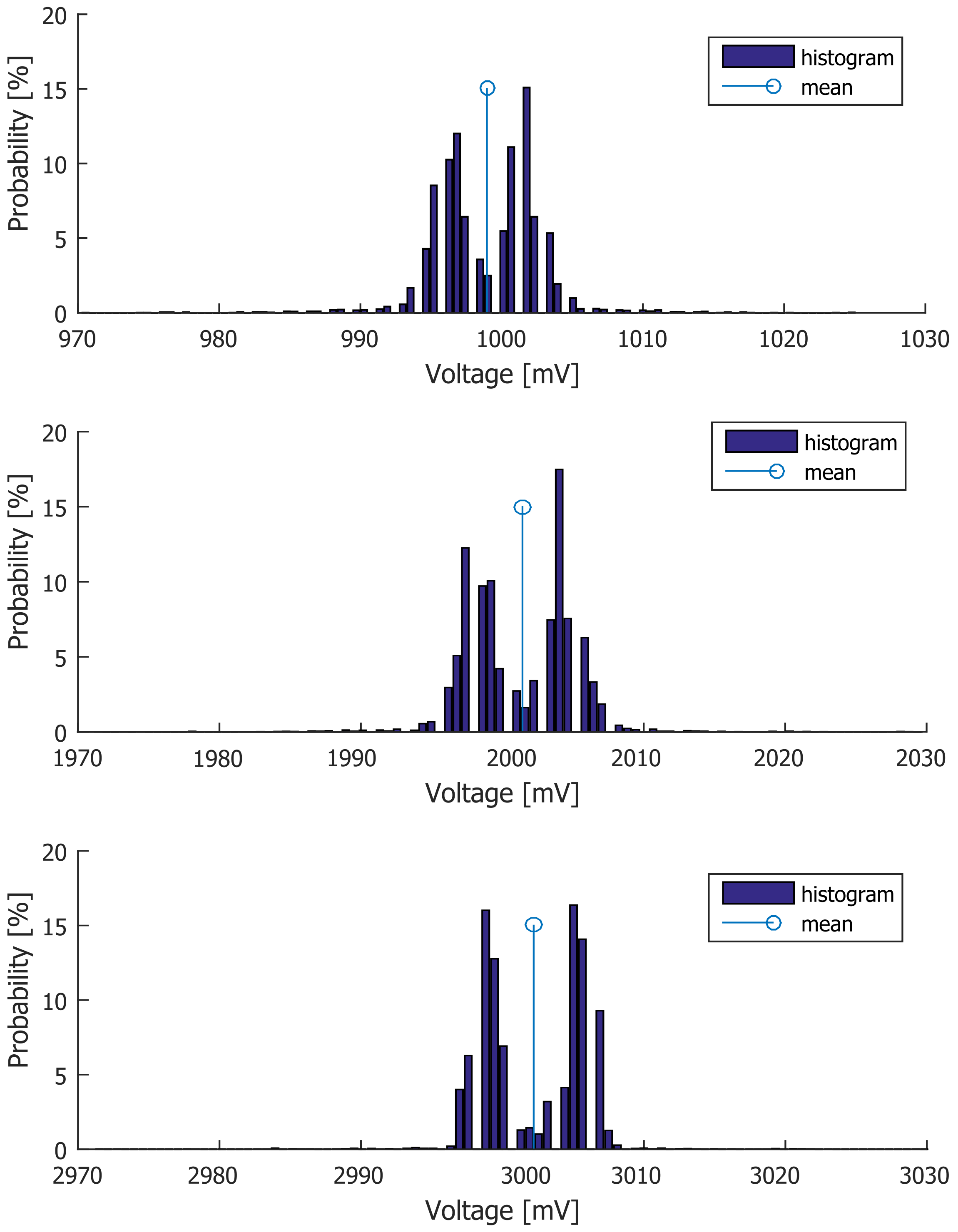


Figure 5.5.: Histogram of ADC measurements

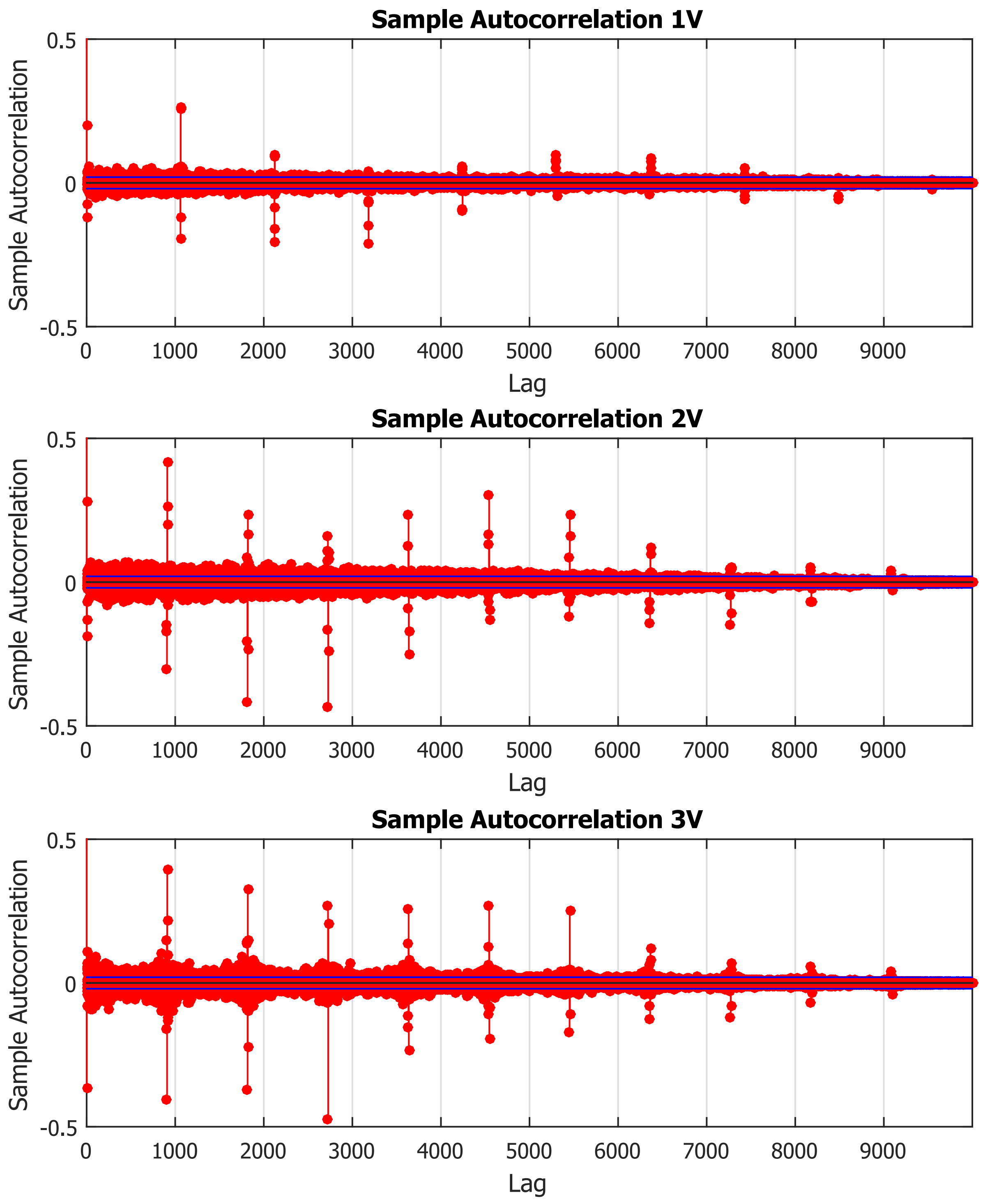
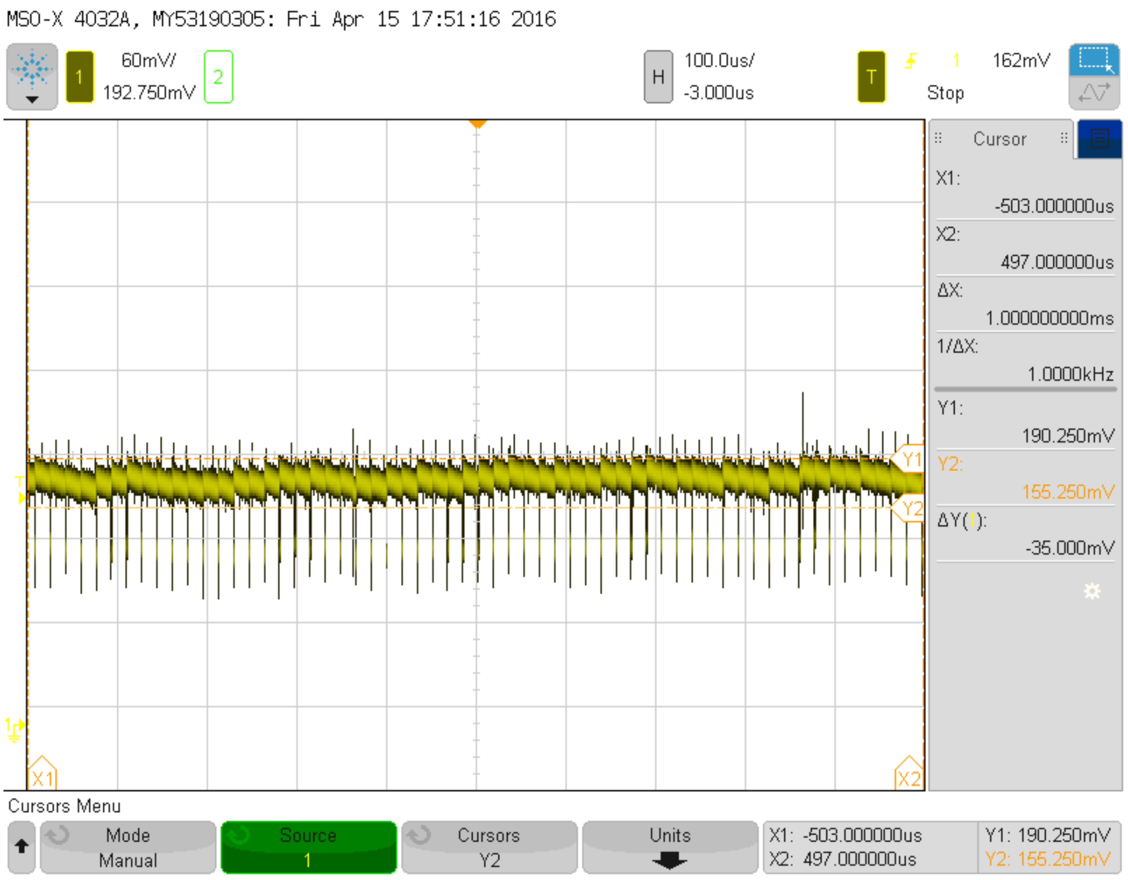


Figure 5.6.: Autocorrelation of ADC measurements

Finally, even though the UART is a strong source of interference, it does not affect the signal strongly. This can be seen on figs. 5.7 and 5.8. The first figure is representative of when the DAC is connected to the ADC with a set voltage, both in the case of the UART sending and recieving data and in the case of being idle. The second figure on the other hand shows what happens when the ADC input is left floating. In both cases a 10kB transfer test at 115200baud was used to test whether the UART affects the measurement.



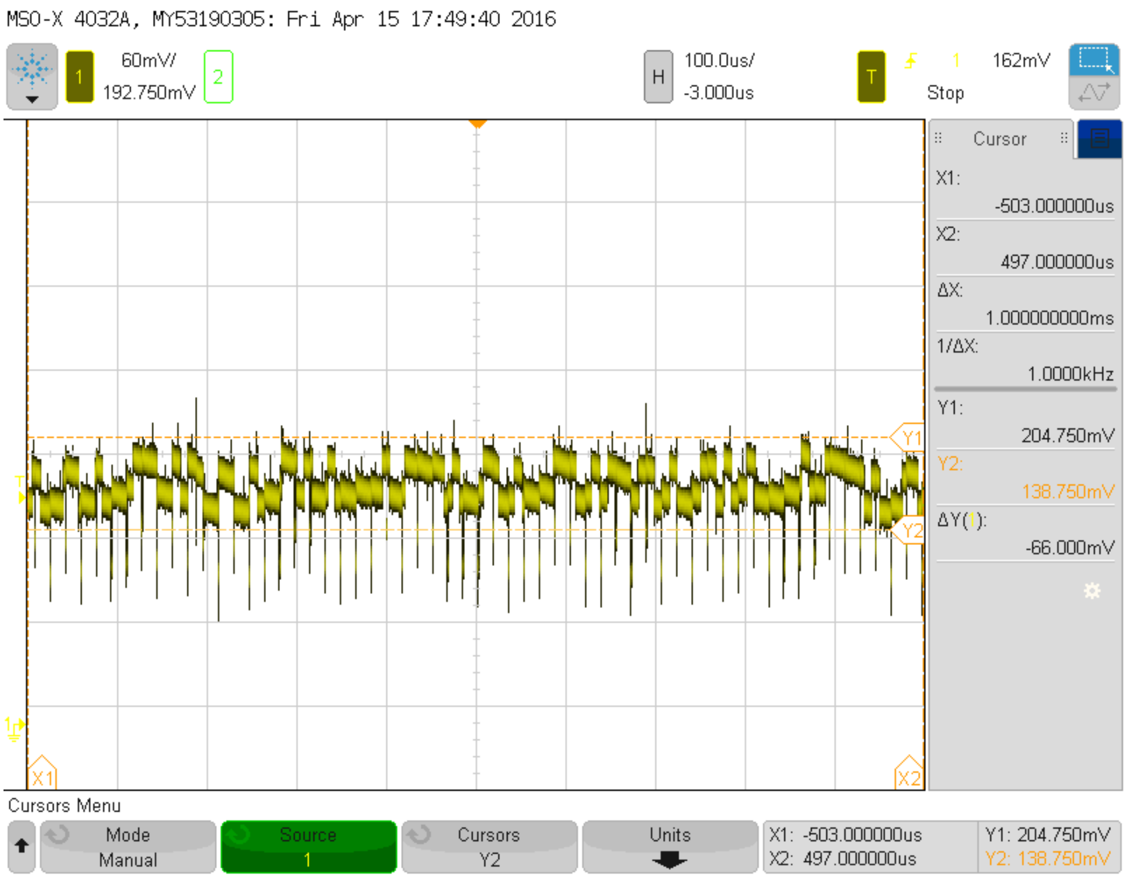
X1: -503.000000us
X2: 497.000000us
ΔX: 1.000000000ms
1/ΔX: 1.0000kHz
Y1: 190.250mV
Y2: 155.250mV
ΔY(1): -35.000mV

Figure 5.7.: Fluctuations on DAC set to 2V

From the data we can conclude that averaging from a large enough sample set (1000 samples are ideal, but even 100 yielded satisfactory results) most of the noise can be removed. Furthermore, if the ADC input pin is left floating it will catch interference, and thus it should always be connected, at least with a pullup.

5.4. Correcting ADC gain error with internal reference

The ADC measures voltage relatively to the the reference voltages Vref+ and Vref-, which are typically connected to the supply voltage Vcc and Ground. This means that if the supply voltages are not known, the value read from the ADC cannot be correctly converted to a voltage.



MSO-X 4032A, MY53190305: Fri Apr 15 17:49:40 2016

1 60mV/ 192.750mV 2

H 100.0us/ -3.000us

T 1 162mV Stop

Cursor

X1: -503.000000us

X2: 497.000000us

ΔX: 1.000000000ms

1/ΔX: 1.0000kHz

Y1: 204.750mV

Y2: 138.750mV

ΔY(1): -66.000mV

Cursors Menu

Mode Manual | Source 1 | Cursors Y2 | Units

X1: -503.000000us X2: 497.000000us

Y1: 204.750mV Y2: 138.750mV

Figure 5.8.: Fluctuations on DAC set to high impedance

The voltage on the ADC is converted with equation 5.10[4], where $Full\_Scale$ is the ADC resolution, e.g. 256 is 8 bit mode, and 4095 in 12 bit mode and $ADCx\_DATA$ is the measured value from the ADC. For example, if the supply voltage is 3.2V and the ADC measures the value 127 in 8bit mode (one half), then the input voltage is 1.6V, if the supply voltage is 3V however, the same reading would mean the input voltage is only 1.5V. If the supply voltage of the ADC is not known precisely, the measured value will not be precise. This means that if one has two identical microcontrollers, running the same program, but with a different power supply, they will both measure different voltages.

$$V_{\text{Channelx}} = \frac{V_{\text{ccA}} \cdot ADCx\_DATA}{Full\_Scale} \tag{5.10}$$

This limitation can be easily fixed using the internal reference voltage. Usually either its voltage is given, such as in the F1xx series, or its calibration value measured for each microcontroller, whic his true for the STM32F3xx and STM32F0xx lines. The analog supply voltage V_{ccA} for the latter two is calculated from the internal reference voltage $V_{\text{REFINT\_DATA}}$ channel and the factory calibration value $V_{\text{REFINT\_CAL}}$ using equation 5.11. Therefore by substitut equa-

[4]The calculation shown in this section works only with some microcontrollers, such as the F042, F303 and F411. On others the reference is done differently, therefore it is highly advisable to check the reference manual before implemneting it.

tion 5.11 into equation 5.10, we get the final equation 5.12 for calculating the correct voltage on the ADC.

$$V_{\mathrm{DDA}} = \frac{3.3\,\mathrm{V} \cdot V_{\mathrm{REFINT\_CAL}}}{V_{\mathrm{REFINT\_DATA}}} \tag{5.11}$$

$$V_{\mathrm{Channelx}} = \frac{3.3\,\mathrm{V} \cdot V_{\mathrm{REFINT\_CAL}} \cdot ADCx\_DATA}{V_{\mathrm{REFINT\_DATA}} \cdot Full\_Scale} \tag{5.12}$$

Unfortunately I have not found a method in the HAL libraries to access the $V_{\mathrm{REFINT\_CAL}}$, so the value address has to be addressed directly i.e.

```
const uint16_t * vrefint_cal_addr = 0x1ffff7ba;
unint16_t VREFINT_CAL = *vrefint_cal_addr;
```

The address can be found in the datasheet and is different for every microcontroller series, maybe even for every microcontroller line (I have not checked this.)

Important: reading data at other than allowed addresses will lead to a hard fault and the program will not run at all. For example if you write a program for a F303, and set the vrefint_cal address for a F042, the program will end in a hard fault.

5.5. ADC input current

Another source of error when measuring voltage with an ADC is its current input. Many microcontrollers, including STM32 series microcontrollers, use successive aproximation ADCs and since these ADCs work by comparing the input voltage to an internal DAC a sample and hold circuit (S/H) is required.

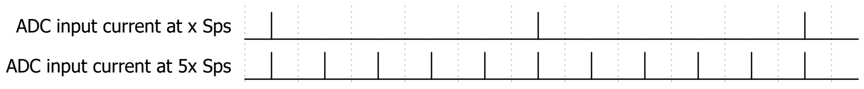


Figure 5.9.: ADC input current at different sampling frequencies

The S/H connects to the measured circuit momentarily to charge its capacitor, as seen in fig. 5.10. The charging time of the S/H capacitor is called sampling time and usually can be set to a selection of values. The choice of this time is relevant to the quality of the measurement, as with lower sampling time the input current grows. The input current of the ADC is given by eq. 5.13, where T is the sampling period, V_{cc} the supply voltage and C_{sh} is the sample and hold capacitance (typically $5 \sim 7\,\mathrm{pF}$) .

$$I_{cc} = f \cdot V_{cc} \cdot \sum C = \frac{V_{cc} \cdot \sum C}{T} \tag{5.13}$$

This current draw can be problematic because it will cause a voltage drop on the measured load. For example the F303 has a 5 pF S/H capacitor; assuming a sampling time of 20 μs (200 kHz) is used and the measured voltage is 1 V, then the current is $2 \cdot 10^5\,\mathrm{Hz} \cdot 1\,\mathrm{V} \cdot 5 \cdot 10^{-12}\,\mathrm{F} = 1 \cdot 10^{-6}\,\mathrm{A}$. Thus if the load impedance is 10 kΩ, the voltage drop will be 10 mV, which is significant, especially considering that the resolution of the ADC at 12bits is 0.8 mV.

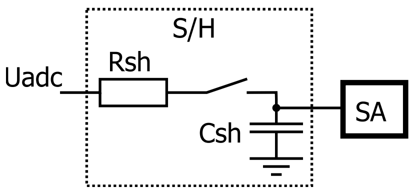


Figure 5.10.: ADC Sample and hold

At higher sampling speeds (the F303RE has a sampling frequency of upto 5.1 MHz) another problem arises: if the load impedance is too high, the capacitor will not be able to charge (or discharge) fully thus distorting the measurement even more.

The maximum recomended impedances for each sampling time are in the datasheet in the Maximum ADC Rain. For example at the lowest sampling time, the maximum recommended impedance is only18 Ω.

- The S/H draws the current in pulses, as seen in fig. 5.9, however the rest of the time there is virtually no current draw. For this reason it is possible to lessen the error caused by the voltage drop by adding a low pass filter before the ADC input (effectively a capacitor in parallel with the S/H internal capacitor). This additional capacitor will provide the required charge to the internal S/H capacitor, and thus the current draw from the load will be lower. This also means that the load can have an impedance higher than recomendded, and the measurement will still be correct.

- Another solution is to simply increase the sampling time. In most cases the ultra fast sampling times are required only when the instrument is used as an oscilloscope or logic analyzer.

The ADC input current draw can be seen in figure 5.7, where each 'needle' is the sampling, and in figure 5.11 it can be seen that the sampling causes a voltage drop, which is then compensated by the DAC.

5.6. Results from the ADCs

This section will explore the measured characteristicts of the ADCs of the F042K6, F303RE, and the L053R8 and F411RE microcontrollers. All the test were done with averaging from a 50Hz rejecting sample set.

The voltage was supplied from an Agilent E3632A, and was measured by an Agilent 34401A multimeter for reference.

5.6.1. F303

To show that the ADC input current does make a difference, the F303 dataset contains measurements with and without an RC lowpass filter (10 nF, 470 Ω) which acts as a 'quick power bank' for the ADC. All the other ADCs were measured only with the LP filter . The data was also measured with the voltage reference correction applied.

From the %Error Average it s clear that just adding the low pass filter increases the accuracy. The F303 was measured at 58kHz, therefore at higher frequencies the error without any correctinos would be even higher. By adding both the LP filter and correcting the gain, the accuracy increase over 3.3 times.

The default voltage of the ADC (can be found by connecting a capacitor in parallel with the internal S/H capacitor) is ~ 210 mV, however just by activating the internal reference (not even measuring on it), this voltage shifts to ~ 500 mV.

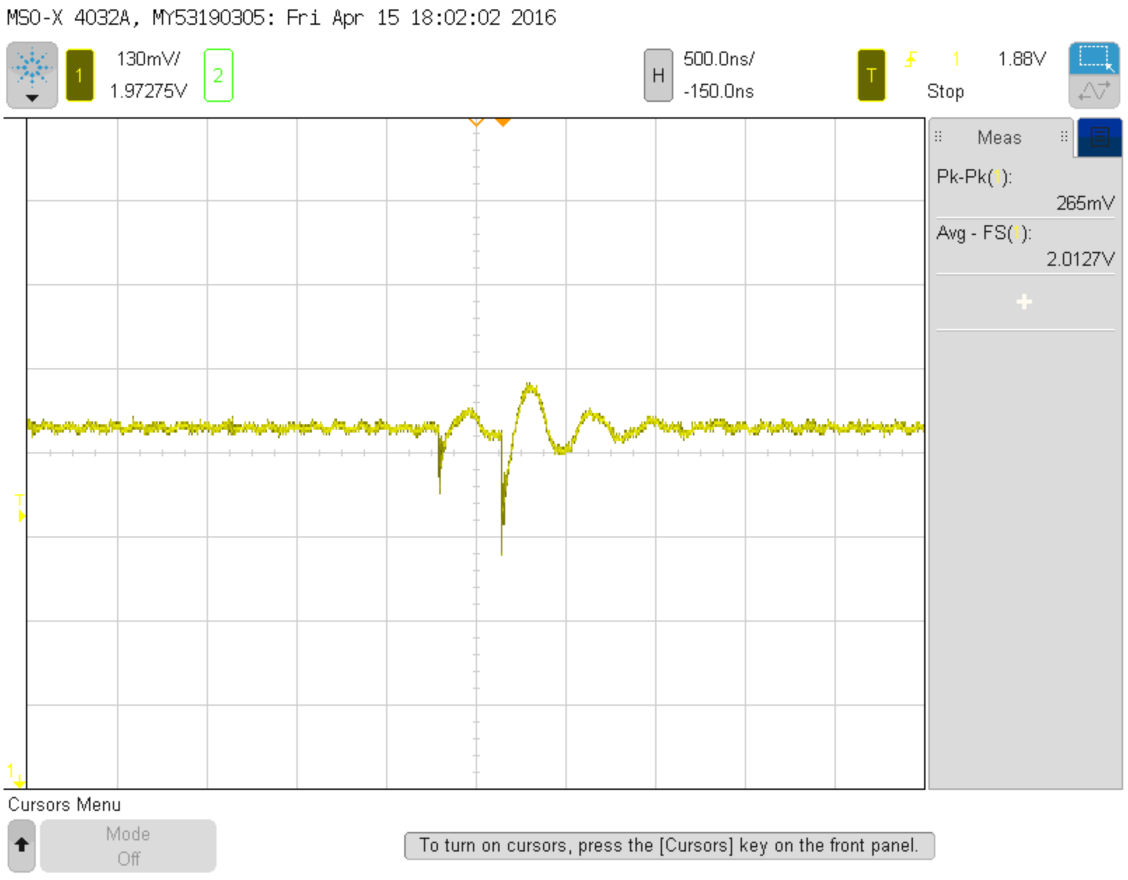

MSO-X 4032A, MY53190305: Fri Apr 15 18:02:02 2016
1
130mV/
1.97275V
2
H
500.0ns/
-150.0ns
T
1
1.88V
Stop
Meas
Pk-Pk(1):
265mV
Avg - FS(1):
2.0127V
Cursors Menu
Mode
Off
To turn on cursors, press the [Cursors] key on the front panel.

Figure 5.11.: DAC reaction to ADC current draw

5.6.2. F042

As seen in table 5.3, the F042 ADC is interesting, because the voltage on the ADC is shifted by 55mV, it doesn't even measure any voltage until 55mV. However in the measured range the results were almost perfect - at worst the voltage was 1mV less (assuming a correction of the voltage shift). However because of the shift, the gain correction breaks the measurement. While the %Error average is better with the gain correction, it is clear the measurements without it are much better.

Very interestingly, in these test the F042 ADC seems to be the most accurate, even though the F303 ADC should be the best. This is most probably because the F303 ADC is better in 'stressful situations'.

5.6.3. F411 and L053

The L053 has the internal reference implemented differently than the other three microcontrollers, therefore I did not implement or measure it.

5.7. Limitations when measuring frequency

Supply	**Corrected gain and low pass added**		**Low pass added**		**Corrected gain**		**No corrections**	
Voltage	Voltage	%Error	Voltage	%Error	Voltage	%Error	Voltage	%Error
100	97	3.00	97	3.00	94	6.00	94	6.00
200	197	1.50	196	2.00	194	3.00	193	3.50
400	397	0.75	395	1.25	394	1.50	392	2.00
600	598	0.33	595	0.83	595	0.83	592	1.33
800	799	0.13	794	0.75	796	0.50	791	1.13
1000	999	0.10	994	0.60	997	0.30	991	0.90
1200	1199	0.08	1193	0.58	1197	0.25	1190	0.83
1400	1400	**0.00**	1392	0.57	1397	0.21	1390	0.71
1600	1600	**0.00**	1591	0.56	1597	0.19	1589	0.69
1800	1800	**0.00**	1791	0.50	1798	0.11	1788	0.67
2000	2000	**0.00**	1990	0.50	1998	0.10	1988	0.60
2200	2201	0.05	2190	0.45	2199	0.05	2187	0.59
2400	2402	0.08	2389	0.46	2400	**0.00**	2387	0.54
2600	2603	0.12	2589	0.42	2601	0.04	2587	0.50
2800	2803	0.11	2788	0.43	2801	0.04	2786	0.50
2900	2903	0.10	2887	0.45	2901	0.03	2885	0.52
3000	3004	0.13	2988	0.40	3002	0.07	2986	0.47
3100	3104	0.13	3087	0.42	3102	0.06	3085	0.48
3200	3204	0.13	3187	0.41	3203	0.09	3185	0.47
3300	3305	0.15	3287	0.39	3303	0.09	3285	0.45
%Error Average		**0.34**		**0.75**		**0.67**		**1.14**

Table 5.2.: STM32F303 ADC measured data

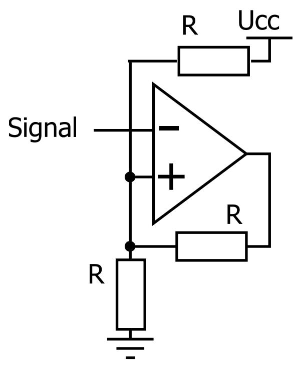


Figure 5.12.: Saturated inverting opamp

There are three main limitations that should be taken care of when measuring frequency:

1. the input signal must not exceed the voltage range of the logic, typically 0 to 3.3V. See warning 4.7 on page 33.
2. The signal should be square. This is a limitation mainly for recpirocal frequnecy measurement and duty cycle, because if the signal is not a square, the start and stop voltage levels do not have to be identical. Thus if a non-square signal is measured, either DFM can be used, or an opamp in saturation as seen in fig. 5.12 should be added before the timer input. (Do note that it will invert the signal).
3. To get a reasonable accuracy an external crystal must be used instead of the HSI. On the discovery and nucleo boards, this can also be done via the bypass which uses the clock from the debugger.

The accuracy when using just the HSI can be seen in table 5.5[5]. For DFM the accuracy is 400 to 1000ppm (0.04%~0.1%), but the measured frequencies are very low. By introducing the HSE, the accuracy increases significantly. In the case of the F303, as seen in fig. 5.6, the accuracy

[5]I have implmented DFM in 2 ways. With a second timer for gating and with systick interrupts for gating.

Supply	**Corrected gain and**		**Low pass added**	
Voltage	Voltage	%Error	Voltage	%Error
100	48	52.00	45	55.00
200	153	23.50	145	27.50
400	365	8.75	345	13.75
600	577	3.83	544	9.33
800	789	1.38	745	6.88
1000	1001	0.10	945	5.50
1200	1213	1.08	1145	4.58
1400	1426	1.86	1345	3.93
1600	1635	2.19	1545	3.44
1800	1845	2.50	1744	3.11
2000	2058	2.90	1945	2.75
2200	2270	3.18	2144	2.55
2400	2483	3.46	2345	2.29
2600	2696	3.69	2545	2.12
2800	2909	3.89	2745	1.96
3000	3122	4.07	2945	1.83
3100	3227	4.10	3045	1.77
3200	3335	4.22	3144	1.75
3300	3441	4.27	3245	1.67
%Error Average		**6.89**		**7.98**

Table 5.3.: STM32F042 ADC measured data

L053			F411				
Supply	Low pass added		Supply	Corrected gain and Low pass		Low pass added	
Voltage	Voltage	%Error	Voltage	Voltage	%Error	Voltage	%Error
500	498	0.40	500	501	0.20	498	0.40
1000	994	0.60	1000	1001	0.10	996	0.40
1500	1491	0.60	1500	1502	0.13	1493	0.47
2000	1988	0.60	2000	2004	0.20	1992	0.40
2500	2489	0.44	2500	2506	0.24	2491	0.36
3300	3280	0.61	3300	3306	0.18	3286	0.42
%Error Average		0.54	%Error Average		0.18		0.41

Table 5.4.: STM32L053 and STM32F411 ADC measured data

increased more than 100 times to 3ppm. Also it might seem that with systick there is lower %Error than with 2 timers. However, the data with HSE was measured only for the two timer DFM. Thus the %Error with systick DFM actually grows to 0.1%, while the %Error drops to 3ppm with 2 timer DFM.

Source	**DFM with 2 timers**		**DFM with Systick**		**RFM**	
Frequency	Frequency	%Error	Frequency	%Error	Frequency	%Error
1000	999	0.10	1000	0.00	999	0.10
2000	1999	0.05	2001	0.05	2000	0.00
5000	4997	0.06	5002	0.04	5000	0.00
10000	9995	0.05	10004	0.04	10000	0.00
12000	11994	0.05	12006	0.05	12000	0.00
15000	14993	0.05	15008	0.05	15030	0.20
20000	19989	0.06	20009	0.05	20050	0.25
30000	29982	0.06	30013	0.04	30125	0.42
40000	39977	0.06	40017	0.04	40223	0.56
50000	49980	0.04	50026	0.05	50320	0.64
65000	64963	0.06	65036	0.06	65454	0.70
%Error Average		**0.06**		**0.04**		**0.26**

Table 5.5.: Timer accuracy with HSI

Source Frequency [MHz]	MeasuredFrequency [MHz]	Error PPM
1	1.000,003	3.00
2	2.000,006	3.00
5	5.000,015	3.00
10	1.000,0032	3.20
12	12.000,038	3.17
15	15.000,042	2.80

Table 5.6.: Timer accuracy with HSE

The accuracies for the microcontrollers with HSE in bypass and measuring 10MHz with DFM were the following:

- STM32F411: 10ppm
- STM32F042: 6ppm
- STM32F303: 3ppm

Unfortunately the L053 doesn't have a 32bit timer, and thus I did not mesure the accuracy of DFM.

CHAPTER

SIX

CONCLUSION

In this thesis I have explored a few methods how to implement basic measurement functions using STM32 microcontrollers.

I have discussed the various methods of controlling microcontrollers as measurement instruments and created working examples, templates and programs. The complete programs I created with mbed for the STM32F042 and STM32F303 nucleo boards ar:

- a standalone (no PC required) demo for measuring passes through an optical gate, evaluating the output of an reflexive optical sensor and reading voltage, with an HD44780 LCD display,
- frequency measurement programs, most notably the DFM one, and a two channel voltmeter, all with UART communication.

The complete programs I created with STM32Cube are:

- an oscilloscope with a PC frontend for the STM32F042 and STM32F072, which implements a USB to UART bridge in the microcontroller,
- static signal measurement and generation programs for the STM32F042, STM32F303, STM32F411 and STM32L052 nucleo boards which communicate over UART, and a
- static signal measurement and generation program with SCPI communication protocol and a PC frontend for the STM32F303 nucleo.

I have found out that the microcontrollers have reasonable voltage measurement capabilities, and in the case of the F042 an accuacy within 2LSB. Furthermore I have found out that the timers of the microcontrollers are very precise and by using the master clock out from the debugger (since it has a crystal) on discovery and nucleo boards, DFM can achieve an astonishing accuracy of 3ppm and high frequencies (tested to 15 MHz with calibrated signal generator). Unfortunately mbed cannot utilize the accuracy of the timers without directly programming them through registers, however software side DFM worked for frequencies upto 200 kHz with a top accuracy of 5ppm.

I have explored methods of compensating the measurement limitations of the perihperals, such as the limited input voltage and unwanted noise.

I have also shown the possibilities of porting the programs created with STM32Cube and mbed to other microcontrollers of different series such as the F411 and L053. With mbed it was a matter of changing the target and recompiling, with cube a new project had to be created and the perihperals reconfigured. However, compared to porting programs created with STM32Peripheral libraries or even directly in assembly[1], it is very simple.

At the beginning of the thesis I also created some programs for the F042 with the standard peripheral libraries from ST. I felt that they are not suitable for this thesis though (the usability of the programs was roughly the same as with Cube, but took much more effort to create), and thus they are not discussed and only the programs are attached.

6.1. What could be improved

To increase accuracy of reciprocal frequency measurement, either the input signal could be divided with another timer, or the input capture data could be transferred with DMA into a buffer, and processed later like it is done with static voltage measurement. If the 16bit size of the timer size were limiting, two timers could be 'chained' to create a 32bit timer if one isn't available. Furthermore, if the DMA method is used, it would be possible to plot the data to see how the frequency is fluctuating in time.

Both frontends were created with NI Labwindows CVI, which is not very practical since it is not freely available. Moreover the static measurement frontend (F303RE Frontend) uses the VISA library for UART communications, which means the installer has an additional 120MB just to install VISA. On the other hand the scope uses the RS232 library which is very small, but doesn't have a builtin function for listing available COM ports. If I were to develop a new application for the same target group, I would most probably use either the .NET framework, or QT as they are free and widely used.

Unfortunately Cube does not provide a function to read from the UART until a certain termination character, but only a function to read a certain amount of characters. Since SCPI is based on separators (':', ' ', '\n', etc.) and the length of each command can be different, I had to build the strings character by character and parse the termination character myself. Next time it would be better to choose another protocol, or use another library to avoid this problem (unless ST adds the functions).

The oscilloscope is only 8bit, while the microcontrollers have 12bit ADCs. It would be interesting to use the full 12 bits, albeit with possibly slower sampling rates. Since the oscilloscope was created rather as a proof of concept, a full text protocol has been used and no transfer speed optimizations have been done. If it were desired to reach the full potential of the microcontroller, a new VCP driver could be created and the protocol should be minimized.

[1]Porting programs created in Perihperals or assembly was not shown or discussed in this thesis.

BIBLIOGRAPHY

[1] David M. Alter. Using pwm output as a digital-to-analog converter, September 2008. URL http://www.ti.com/lit/an/spraa88a/spraa88a.pdf.

[2] Jan Axelson. Implementing a vcp. URL http://janaxelson.com/usb_virtual_com_port.htm.

[3] Okawa Electric Design. Rc low-pass filter design for pwm, 2008. URL http://sim.okawa-denshi.jp/en/PWMtool.php.

[4] Frank Durda. Bsd serial and uart tutorial, April 2014. URL http://www.freebsd.cz/doc/en/articles/serial-uart/.

[5] Simon Ford. Hd44780 mbed library, January 2014. URL https://developer.mbed.org/components/HD44780-Text-LCD/.

[6] Jiri Hladik Jan Fischer. Seminar at the department of measurement, faculty of electrical engineering, czech technical university, educational platform with nucleo modules, 2016.

[7] Andy Kirkham. Interrupts and contexts, March 2011. URL https://developer.mbed.org/users/AjK/notebook/regarding-interrupts-use-and-blocking/.

[8] Mbed. Mbed serial handbook, . URL https://developer.mbed.org/handbook/Serial.

[9] Mbed. Automatic clock configuration, . URL https://developer.mbed.org/teams/ST/wiki/Automatic-clock-configuration.

[10] Mbed. Mbed cookbook, . URL https://developer.mbed.org/cookbook/Homepage.

[11] Mbed. Mbed handbook, . URL https://developer.mbed.org/handbook/Homepage.

[12] Multiple. Analog voltage level conversion (level shift), 2012. URL http://electronics.stackexchange.com/questions/30719/analog-voltage-level-conversion-level-shift.

[13] Tomas Ostrowsky. Miniscope. URL http://tomeko.net/miniscope_v2c/index.php?lang=en.

[14] SiLabs. An118 improving adc resolution by oversampling and averaging, July 2013. URL https://www.silabs.com/Support%20Documents/TechnicalDocs/an118.pdf.

[15] STMicroelectronics. Stm32f303re datasheet, December 2015 . URL http://www2.st.com/resource/en/datasheet/stm32f303re.pdf.

[16] STMicroelectronics. Nucleo stm32f303re pinout, . URL https://developer.mbed.org/platforms/ST-Nucleo-F303RE/.

[17] STMicroelectronics. An1636 understanding and minimising adc conversion errors, June 2003. URL http://www.st.com/web/en/resource/technical/document/application_note/CD00004444.pdf.

[18] STMicroelectronics. Stm32f042k6 datasheet, December 2015. URL http://www2.st.com/resource/en/datasheet/stm32f042k6.pdf.

[19] STMicroelectronics. Stm32f0xx reference manual, July 2015. URL http://www2.st.com/resource/en/reference_manual/dm00031936.pdf.

[20] STMicroelectronics. Stm32f3xx reference manual, May 2016. URL http://www2.st.com/resource/en/reference_manual/dm00043574.pdf.

[21] STMicroelectronics. Cube, 2016. URL http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1897?icmp=tt2930_gl_pron_oct2015&sc=stm32cube-pr14.

[22] Tektronix. Triggering fundamentals, 2011. URL http://www.tek.com/dl/55W_17291_6_0.pdf.

APPENDIX

A

USING THE USB BOOTLOADER TO LOAD MBED OR CUBE PROGRAMS TO STANDALONE MICROCONTROLLERS

This apendix will show how to convert a binary (.bin) or hex file to a DFU which can be loaded to an ST microcontroller which has the USB bootloader, such as the STM32F042.

Since the development boards compatible with mbed use the same microcontrollers which can be bought directly, such as the STM32F042 nucleo, and mbed should set the clock dynamically, the mbed binaries can be used on 'solo' microcontrollers.

I have created a simple program that is controlled with push buttons, displays data on an HD44780 display, counts pulses and voltage. This program was developed using an STM32F042 nucleo board, and then simply uploaded to the microcontrollers.

A.1. Considerations

It has been mentioned multiple times that frequency measurement requires a precise clock. Therefore it is important to either use an external clock source or a crystal. Mbed claims that it tries to connect to the various clock sources in the following order[9]:

1. external bypass clock (from the debugger Master Clock Output on development boards), then
2. it tries to start the crystal, and if that is unsuccessful,
3. it switches to the HSI

Unfortunately I have tested this on a STM32F042 microcontroller that had a crystal connected and the **crystal did not start** (nothing seen on an oscilloscope with a 10:1 probe and removing the crystal did not cause any problems in the program execution). When the exact same setup was programmed with a Cube program, the crystal did start, therefore it can be assumed that there wasn't a problem with the hardware.

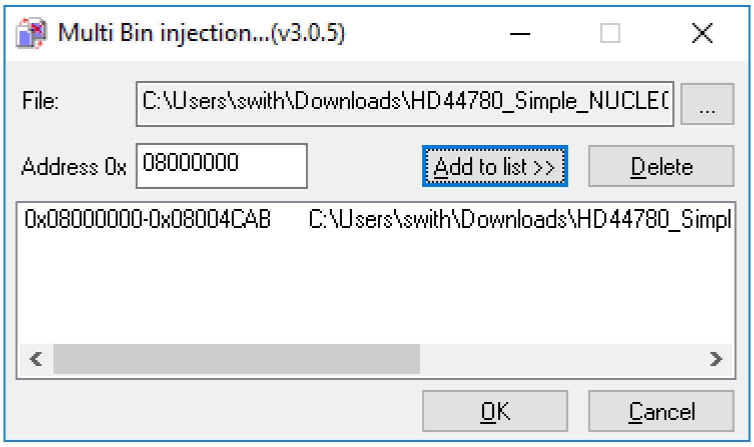
It is possible that this is a bug just in mbed F042 implementation, however this is out of the scope of this thesis and thus will be left only with the warning that the crystal does not start.

A.2. Creating a dfu file and flashing the microcontroller

From the ST website download the DFU File Manager and the DFuSe demo.

When you have your bin or hex file ready, open the DFU File Manager. For a hex file simply click hex, choose it and click generate. For a bin,

- click multi-bin, a window as seen in fig. A.1 pops up,
- select your bin file, and set the address to 0x08000000 (Flash memory),
- add to list, click ok and generate



File: C:\Users\swith\Downloads\HD44780_Simple_NUCLE(

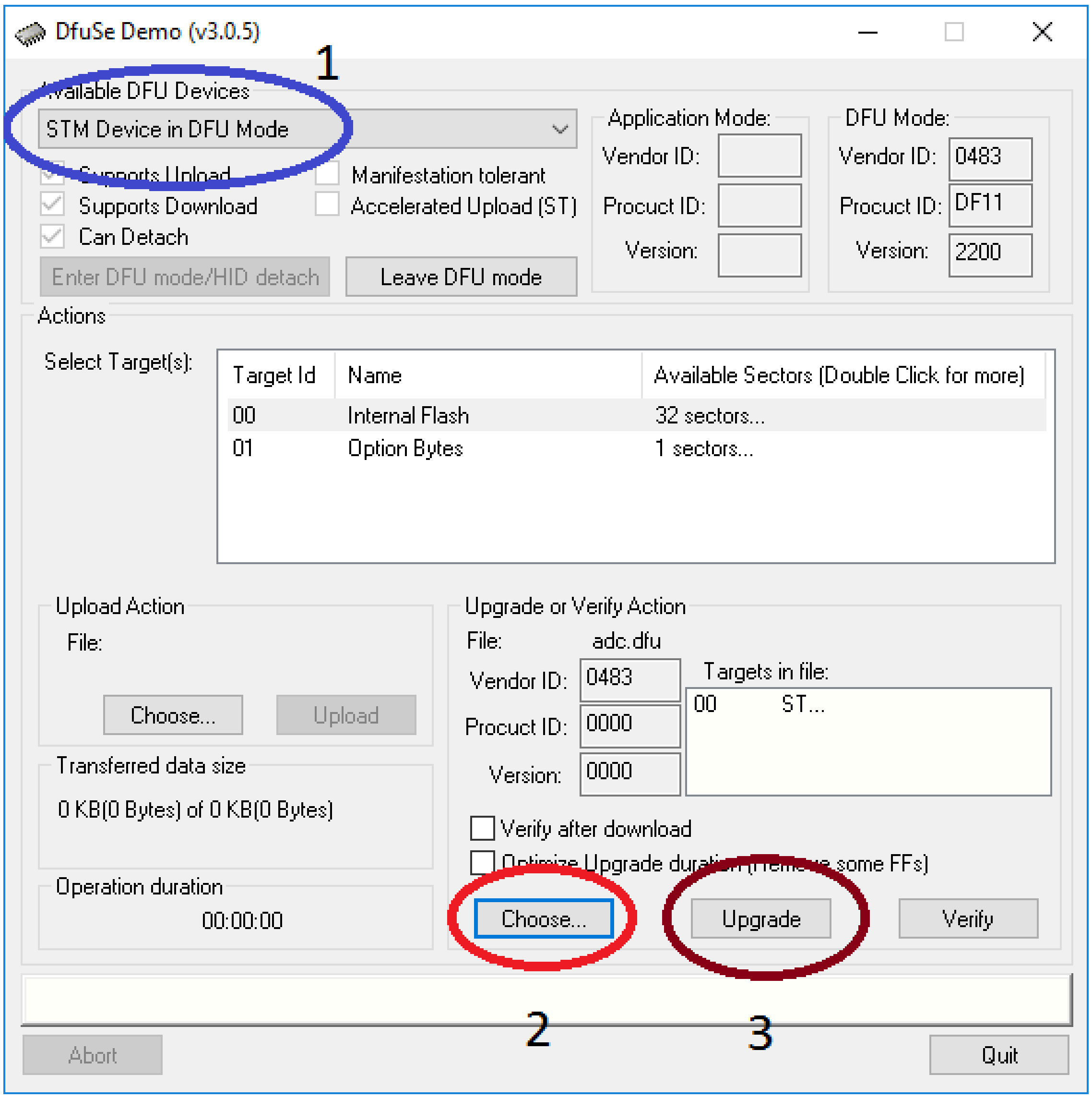
Address 0x 08000000

0x08000000-0x08004CAB   C:\Users\swith\Downloads\HD44780_Simpl

Figure A.1.: DFU memory setup for bin files

To flash the DFU file to the microcontroller, open the DFuSe demo and connect your microcontroller in DFU mode[1]. You will know that it is connected because DFuSe will immediately show it up (1 in fig. A.2). Then select the dfu file generated in the previous steps and upload it via the upgrade button (2 and 3 in fig. A.2). Finally put the microcontroller back into normal mode and your microcontroller is flashed.

[1]This most probably varies by micocrocorntroller. On the STM32F042 the Boot memory selection (pin 31 on LQFP32 and pin 1 on TSSOP20) pin has to be pulled low during reset to enter DFU mode.



DfuSe Demo (v3.0.5)

Available DFU Devices: STM Device in DFU Mode

- [✓] Supports Upload
- [✓] Supports Download
- [✓] Can Detach
- [ ] Manifestation tolerant
- [ ] Accelerated Upload (ST)

Application Mode: Vendor ID: Procuct ID: Version:

DFU Mode: Vendor ID: 0483 Procuct ID: DF11 Version: 2200

Actions

Select Target(s):

Target Id	Name	Available Sectors (Double Click for more)
00	Internal Flash	32 sectors...
01	Option Bytes	1 sectors...

Upload Action

File:

Transferred data size: 0 KB(0 Bytes) of 0 KB(0 Bytes)

Operation duration: 00:00:00

Upgrade or Verify Action

File: adc.dfu

Vendor ID: 0483

Procuct ID: 0000

Version: 0000

Targets in file: 00 ST...

- [ ] Verify after download
- [ ] Optimize Upgrade duration (Remove some FFs)

Figure A.2.: DFuSe demo main window

APPENDIX

B

MBED SAMPLE CODES

Algorithm B.1 Mbed 2 channel voltmeter

```
#include "mbed.h"
Serial pc(USBTX, USBRX);
AnalogIn channel1(A0);
AnalogIn channel2(A1);
int main() {
     double voltage1;
     double voltage2;
     while(1) {

         //Both channel1 and 2 are objects, but printf requires a
         //plain old data type (POD), therefore the data needs to be read into a plain variable
         voltage1 = channel1*3.3;
         //or we can call the read explicitly
         voltage2 = channel2.read() * 3300;
         //%x.yf formats the float with x digits before and y after the decimal point
         pc.printf("A0: %1.2fV, A1: %4.1fmV\r", voltage1, voltage2);
         wait_ms(100);
     }
}
```

Algorithm B.2 Mbed Direct frequency measuremnet

```
#include "mbed.h"
InterruptIn pSource(A0);
Serial pc(SERIAL_TX, SERIAL_RX);
//Test pwm signal
PwmOut pwm(D5);
uint32_t pCount = 0;
//Just increment the counter with each high
void incrementCount() {
    pCount++;
}
int main() {
    //Attach the count method to the rising edge of the pulse
    pSource.rise(&incrementCount);
    pSource.mode(PullDown);
    pwm = 0.5;
    pwm.period_us(2);
    pwm.pulsewidth_us(1);
    while(1) {
        /*Send data ...*/
        printf("Freq: %d\r\n", pCount);
        //clear counter
        pCount = 0;
        //interrupts override wait, so we can use it as a delay
        wait_ms(1000);
        }
}
```

Algorithm B.3 Mbed Reciprocal Frequency Measurement

```
#include "mbed.h"
Serial pc(SERIAL_TX, SERIAL_RX);
InterruptIn pSource(A0);
PwmOut pwm(D5);
Timer period;
Timer pulseWidth;
double frequency;
int dutyCycle;
void calculateFrequency() {
    /*If period != 0, the measurement
    was already started*/
    if(period.read_us() == 0) {
        //Start the period and duty cycle timers
        period.start();
    } else {
        period.stop();
        // f = 1/T
        //don't forget to convert T to seconds
        frequency = 1/(period.read_us()*0.000001);
        period.reset();
    }
}
int main() {
    //Attach the callback to the rising edge on pSource
    pSource.rise(&calculateFrequency);
    while(1) {
        printf("Freq: %f\r\n", frequency);
        /*Do something with the frequency*/
    }
}
```

Algorithm B.4 Mbed measuring duty cycle

```
#include "mbed.h"
Serial pc(SERIAL_TX, SERIAL_RX);
InterruptIn pSource(A0);
PwmOut pwm(D5);
Timer period;
Timer pulseWidth;
double frequency;
int dutyCycle;
void measureDutyCycle() {
    //A falling edge was detected
    if(pulseWidth.read_us() != 0) {

        //And we're measuring duty cycle, so stop the timer
        pulseWidth.stop();
    }
}
void calculateFrequency() {
    if(period.read_us() == 0) {
        //Start the period and duty cycle timers
        period.start();
        pulseWidth.start();
    } else {
        period.stop();
        // f = 1/T
        //but don't forget to convert T to seconds
        frequency = 1/(period.read_us()*0.000001);
        //Calculate the duty cycle, %DC = T_pw / T
        dutyCycle = ((float) pulseWidth.read_us()) /
        ((float) period.read_us()) * 100;
        period.reset();
        pulseWidth.reset();
    }
}
int main() {
    //Attach the callback to the rising edge on pSource
    pSource.rise(&calculateFrequency);
    pSource.fall(&measureDutyCycle);
    while(1) {
        printf("Freq: %f\r\n", frequency);
        /*Do something with the frequency*/
    }
}
```

Algorithm B.5 Mbed PWM Signal Generation

```
/*Function changeFrequency cycles through 3 frequencies
100Hz, 1kHz and 10kHz, and changeDC cylces through 3 duty cycles
25%, 50% and 75%*/
//Initializes the output with a 50Hz frequency and 50% duty cycle
PWMOut pwm(PB_4);
//starting frequency will be 100Hz (10000us)
int period_us = 10000;
float dutyCycle = 0.5f;
void changeFrequency(){
     if(period_us > 100){
         period /= 10;
     } else {period = 10000;}
     pwm.period_us(period_us);
}
void changeDC(){
     //float is terrible, so let us be sure it doesn't overflow
     if(dutyCycle < 0.7f){
         dutyCycle += 0.25f;
     }else {dutyCycle = 0.25;)
     //dutyCycle setting is implicit
     pwm = dutyCycle;
}
```

Algorithm B.6 Mbed LED as multilevel display

```
#include "mbed.h"
InterruptIn button(PC_13);
//An external LED needs to be connected to PA_4
AnalogOut extLED(PA_4);
int state;
void buttonPressed() {
     /*If the button was pressed,
     cycle through states 0-3*/
     if(state < 4) {
         state += 1;
     } else {
         state = 0;
     } //Update the LED
     /*The LED has a ~1.5V drop, so the
     applied voltage has to be at least
     1.5V for it to light up*/
     extLED = 0.5+((double)state)/10;
}
int main() {
     /*Using the interrupt in instead of
     directly reading the button is
     great for debouncing*/
     button.rise(&buttonPressed);
     while(1) {
         switch(state) {
             case 0: /*State 0 activity*/ break;
             case 1: /*State 1 activity*/ break;
             /*...*/
             case 4: break;
             default: /*error ...*/
         }
     }
}
```

Algorithm B.7 Mbed serial command interrupt handler

```
#include "mbed.h"
#include <string.h>
Serial pc(USBTX, USBRX);
int voltage; //Variable where the measured voltage is stored
string command;
char c;
void processCommand(void); //Function prototype
void commandHandler(){

     //One can never be safe enough
     if(pc.readable()){

         //Read the character
         c = pc.getc();
         if(c == ’\n’){

             //If its a newline, process the command
             processCommand(command);
             //After processing, clean the string to receive a new one
             command.clear();

         }
         //Otherwise just append it to the command string and continue
         else {command += c;}

     }

}
int main(){


     //Calls commandHandler() whenever a byte is received on the serial
     pc.attach(&commandHandler);
     while(1){

         //Do stuff

     }

}
void processCommand(string cmd){

     //Working with strings in mbed is simpleL
     if(cmd == “Hello”){

         pc.printf(“Hello!\n”);

     } else if(cmd == “MEAS:VOLT?”){

         pc.printf(“Voltage: %d\n”, voltage);

     } else{

         pc.printf(“Command not recognized\n”);

     }

}
```

APPENDIX

C

CUBE SAMPLE CODES

Algorithm C.1 Using UART with HAL

```
#include <string.h>
//autogenerated
UART_HandleTypeDef huart2;
uint8_t rxBuffer[1];
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

     if(rxBuffer[0] ==’?’){/*do stuff in response to ? being recieved*/}
}
int main(){
     //Array to store the received command
     uint8_t rxBuffer[1];
     //Array to store the message to be sent back
     char result[10];
     //autogenerated, initializes uart
     MX_USART2_UART_Init();
     while(1){
         //Try to receive a single byte, with a 100ms timeout
         if(HAL_UART_Receive(&huart2, (uint8_t *)&aRxBuffer, 1, 100) == HAL_OK){

             /*Process command stored in aRxBuffer ... output in result*/
             HAL_UART_Transmit(&huart2, (uint8_t *)&result, strlen(measurement));
         }
     }
}
```