NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

PROSJEKTOPPGAVE

Kandidatens navn: Stian Orø Moen

Fag: Teknisk kybernetikk

Oppgavens tittel (norsk): Laveffekts akustisk telemetribøye

Oppgavens tittel (eng.): Low power acoustic telemetry buoy

Oppgavens tekst:

Automatiske lyttebøyer har stor betydning for anvendeligheten av akustisk telemetri innen storskala studier av fisks og andre dyrs atferd i havet. Lyttebøyene er batteridrevne enheter som foretar automatisk registrering av identitet og andre typer data fra merkede individer som måtte befinne seg innenfor bøyenes deteksjonsradius, dvs. uten behov for tilstedeværelse eller noen direkte medvirkning fra operatør. Plassering av flere bøyer i strategiske posisjoner gjør det dermed mulig å overvåke atferdsmønstre og vandringer over større geografiske områder med minimal bruk av ressurser.

Det stilles imidlertid strenge krav til telemetribøyenes egenskaper i forhold til bl.a. batterilevetid og robust mottak av telemetridata. Formålet med denne oppgaven er å se nærmere på designprinsipper for den akustiske mottakeren i bøya som ivaretar krav til levetid og robust deteksjon, samtidig som det gis fleksibilitet med hensyn til konfigurasjon av mottakerparametre/kodeskjema og utbygging av bøya med tilleggsfunksjoner (f.eks. mobilkommunikasjon). Oppgaven omfatter følgende punkter:

- Utarbeidelse av overordnet krav- og funksjonsspesifikasjon for telemetribøye
- Utforske egenskapene til forskjellige mottakerløsninger med ulike grader av digital signalbehandling langs signalkjeden, spesielt sett i forhold til krav til effektforbruk, robust deteksjon og fleksibilitet
- Velge hensiktsmessige komponenter og foreta sammenstilling, implementasjon og utprøving mottakerprototyp(er)
- Dokumentasjon og diskusjon av resultater

Oppgaven gitt: 18. august 2008

Besvarelsen leveres: 20. februar 2009

Utført ved Institutt for teknisk kybernetikk

Veiledergruppe: Jo Arve Alfredsen (ITK, NTNU)
Jan Eyolf Bjørnsen, Åge Grønningsæter, Bård Holand (Thelma AS)

Trondheim, 18. august 2008

Jo Arve Alfredsen
Faglærer

Preface

This report is a result of the autumn project carried out in the 5th and final year of a master study at the institute of Engineering Cybernetics, IME faculty at the Norwegian University of Science and Technology. It is meant to be both a project in itself, but should also serve as a basis for the master thesis in the following semester.

The project assignment is provided by Thelma AS in collaboration with the institute of Engineering Cybernetics. Thelma provided the specification and several meetings have been held as the project has evolved.

The project is influenced by the writers experience working at Atmel Norway with AVR microcontrollers both before and during the project. The experience, the access to programming tools and samples and the fact that AVR is one of the leading low power microcontroller series puts Atmel product in focus.

I would like to thank my teaching supervisor Jo Arve Alfredsen for his great effort and guidance throughout the project. I would also like to express gratitude towards Jan Eyolf Bjørnsen and Bård Grønningsæter at Thelma for their help and support.

Finally I would like to thank Atmel Norway and John Olav Horrigmo at the institutes' component service for providing equipment, components and tools.

Trondheim
20th of February 2009

Stian Orø Moen

Summary

The goal of this project was to determine if it is possible to design a low-power acoustic telemetry receiver with digital signal processing, powered by a single battery. The receiver should operate for at least a year and be capable of at least 100 000 receptions.

Alternative modulation schemes have been studied to prepare the design for new fish-tags using more modern form of communication.

This project focuses on the digital part of the acoustic system. To find the ultimate low power MCU several microcontrollers have been compared.

A prototype was designed in order to implement digital signal processing and perform power measurements. The prototype was designed around the ATxmegaA1 microcontroller. A program was constructed that performs sampling, storing, digital filtering and signal level calculation.

The measurements are compared to the AVR32 AT32UC3B microcontroller power calculations. Using datasheet numbers and benchmark tests of the DSP library for the AVR32 series it was found that the AT32UC3B would provide a much better result than the ATxmegaA1. The AT32UC3B microcontroller should therefore be used for further design

Power calculations are done for two types of reception:

- With an external triggering the microprocessor to start receiving data.
- Without an external trigger sampling cautiously and measuring the signal level.

If a good external trigger for reception can be constructed this will provide the lowest energy consumption for the digital system. The system should though incorporate digital filtration to allow compatibility with various frequencies and modulation schemes.

The final conclusion is that with the AT32UC3B microcontroller it should be possible to construct a telemetry receiver with a lifetime operation of one year, both with and without an external trigger.

Contents

1. Introduction

1.1 Project definition

Please refer to page I for the formal project definition. The main goal of the project is to answer the question:
Is it possible to design a low-power acoustic telemetry receiver using digital signal processing?

The telemetry receiver should meet the following specifications:

- Store at least 100 000 receptions with individual time stamps
- Operating time of at least one year using one D-cell Tadiran 19Ah lithium battery [10]
- Handle acoustic receptions of modulated signals with carrier frequencies up to 100kHz

In addition to this it is desired to consider alternative modulation schemes than the ones used today. By doing this it is possible to design the system to handle alternative modulation schemes for future fish tags.

1.2 Background

Today Thelma AS has a manual operated telemetry receiver. This is used to manually observe tagged fish for research purposes. In many situations it is neither practical nor possible to have people operating such a device. This is typically research where records are done in a period of several weeks or months. It is desired to have a stand-alone device that can be left for a longer period of time that records the presence of tagged fish. These devices are referred to as *acoustic telemetry buoys*. The buoys are typically placed in a matrix to record the position of one or several fish over time. Some solutions exist, but the device must typically be fetched to retrieve the recorded information. Thelma wishes to develop their own product with more advanced functions such as GSM communication, GPS and more.

1.3 Available solutions

There are currently two other vendors of telemetry buoys; VEMCO and SONOTRONICS. The latest telemetry buoy from each vendor is the VR2W and the SUR respectively. The following will provide a short description of these products.

VR2W
The VEMCO produced VR2W have the following features:

- 8 MB of memory
- 1 000 000 detections
- 15 months operating time
- Receiver frequency: 69.0 kHz
- Bluetooth communication
- Real time clock

SUR – Submersible Ultrasonic Receiver
The SONOTRONICS produced SUR have the following features:

- 1 MB flash memory
- 100 000 detections
- 7-12 months operating time using two batteries
- 15 selectable frequencies ranging 30 kHz – 150 kHz
- RS-232 communication
- Real time clock
- Ping and response function to check if the SUR have any data

Note that the SUR will not listen to all 15 frequencies at the same time. It will listen for a particular frequency for two seconds, then it will power down for one second, power up and listen to the next frequency. In addition a one second delay is added after all 15 frequencies have been scanned. As a result a particular frequency will be checked every 46 seconds. With these delays the receiver will have an operating time of seven months. It is possible to increase the delays to achieve up to twelve months operating time.

2. Theory

This chapter is the result of an initial theoretical study describing general aspects related to constructing an *ultra low power mixed-signal electronic system, more specifically, an acoustic telemetry buoy.* The chapter covers low-power design techniques and considerations, methods for current measurement, underwater acoustics and a short discussion of relevant modulation schemes.

2.1 Low-power design

This section describes some of the factors that must be considered when designing a low-power application. A basic overview of low power considerations is found in chapter 2.1.1, the rest of this section tries to focus on more special considerations that are often overlooked, special software techniques for minimizing power consumption in an MCU and ways of estimating this power consumption.

2.1.1 General low-power design considerations

In addition to choosing the component with the lowest power consumption there are several other factors that must be taken into consideration. The highest level of integration usually results in the lowest power consumption. In an embedded system this often means choosing a microcontroller with many of the needed features inbuilt. Connecting “components” in an internal circuit instead of on a PCB will in most cases result in less leakage, provide the possibility of running the components closer to the voltage limits and a higher grade of optimization.

In most cases there is a need for special external components in addition to the microcontroller. The ability to disable these devices when not needed will in most cases result in lower average power consumption. Some devices have this function inbuilt, but this feature can be implemented externally by using a port pin and a transistor. A transistor with low leakage is of course preferred. A FET or Darlington transistor will provide high power efficiency.

The most fundamental low-power requirement for any embedded system is that the system must be interrupt driven. This will allow the processor to sleep whenever the CPU is not needed for computation. Delay routines of the type

```
for(int i = 0; i < 200; i++){
//do nothing
}
```

are therefore strongly prohibited in a low-power application. The solution is to use a timer with an associated interrupt to wake the processor when the delay has expired. Modules that can relieve the CPU will also contribute to more time in sleep. This can for instance be a communication module, DMA controller, timer, etc.

Computation prohibits the CPU or MCU from being in a sleep mode. It is therefore vital that the computation time is at a minimum. This requires not only a high frequency and a powerful CPU instruction set, but that the program is constructed and optimized for the specific CPU. To achieve this, great knowledge about the instruction set and the compiler is required. Optimization settings for the compiler must be set, but without knowledge about the CPU the program will never be optimal. An example is using floating-point numbers on a CPU without hardware support for this. All computation will therefore have to be done in software which will lead to a large increase in computation time.

Oscillators will also contribute to the overall power consumption. The designer must not only consider the frequency, but also the type of oscillator. The typical oscillators are internal RC oscillator, external clock, external crystal and external resonator. Factors that must be considered are

- Power consumption
- Start-up time
- Stability (jitter)
- Accuracy

Studying for instance the XMEGA manual [3] we find that the internal RC oscillator has the shortest start-up time and is in many microcontrollers the oscillator that requires the least amount of power. The downsides are that an RC oscillator typically will be more inaccurate and have a grater amount of jitter.

Since a real-time clock often is required in an embedded system many microcontrollers feature an ultralow-power, low frequency oscillator designed especially for 32.768 kHz quartz crystals. The low frequency usually excludes the possibility of using this oscillator as a system clock. On AVR microcontrollers it can though be used as reference to do runtime calibration of the internal RC oscillator to achieve greater accuracy.

Most microcontrollers have the ability of connecting an external crystal, external clocks are therefore seldom used. The external clock option can though be used if there are several other devices that require an oscillator. The devices can then share the same clock.

Unwanted oscillations may also contribute to an increase in power consumption. It is therefore important to use appropriate decoupling and insure a stable digital level on unused inputs. A high input resistance is also important to ensure that a minimum of current is consumed.

2.1.2 General battery considerations

A rather new consideration in low-power design is the non-ideal behaviour of the power source when using a battery. An explanation of the most important aspects is given by Oklobdzija [1]. In many cases, it is implicitly assumed that the power supply provides a constant voltage and delivers a fixed amount of energy. This assumption is not valid in the case of battery-operated devices.

The main non-idealities of real-life battery cells include:

1. The battery output voltage depends nonlinearly on its state of charge.
2. The capacity of the battery depends on the current load. A high current will result in a larger loss of efficiency than a low current.
3. The battery acts as low-pass filter and therefore the “frequency” of the discharge current will affect the amount of charge the battery can deliver.
4. Batteries have some recovery capacity when discharged at high current loads. If the battery is allowed to rest with low current for a while after a high current load, the output voltage increases again. This is known as *the recovery effect*.

5. Nominally equal battery cells may exhibit significant differences in internal resistance, output voltage and voltage vs. current characteristics. It is therefore not recommended to directly connect batteries in parallel.

2.1.2.1 System design techniques

In *Low-Power Electronics Design* Oklobdzija [1] specifies that in a battery application it is necessary to use a DC-DC converter to ensure a good supply-voltage. This will introduce an added power loss, but can result in a lower total current when using a buck converter. This is due to less current consumption of components when using a lower voltage. This will though depend on the specific application.

As described in section 2.1.2, a battery will be less efficient with high currents. It is therefore important to consider the peak currents in the application. One way of limiting the peak currents is to reduce the frequency of the processor. This will lead to a longer computation time, but will reduce the peak current. Whether or not this will result in an overall increase of the lifetime of the battery will depend on the electrical and chemical conditions at the given time.

When implementing dynamic power management (DPM) in a system with relatively high currents, the system can in many cases utilize the recovery phenomenon, described in section 2.1.2, to achieve an extended battery life. By scheduling high current peaks in such a way that the battery gets time to recover in-between peaks, a higher level of capacity can be utilized. Oklobdija [1] refers to this as *Battery-Aware Task Scheduling*.

Another phenomenon that is important to notice is that changing between different modes (off, sleep, active, etc.) might cause current peaks. In a system where the mode changes often, it is important that these currents are taken into consideration. This is referred to as *open-loop DPM.*

Some applications can operate at different qualities or levels of functionality and thus different current requirements. As an example, an mp3 player will often lower the sound quality when the battery is at a low level. This will stretch the lifetime of the battery when a long lifetime is needed, but will maintain the quality in most cases. In applications where the system can operate at different qualities/current consumptions, it can be exploited by measuring the battery voltage and lowering the quality and thereby the current consumption at low battery voltages. This is referred to as *closed-loop DPM.*

As an example of the latter, the Atmel AVR XMEGA microcontrollers can operate at up to 32MHz working at a VCC level of 2.7 – 3.6 V. But when operating at voltages below 2.7V the device requires a lower system clock frequency to ensure correct operation. The relation is shown in Figure 1. If the voltage is measured using the internal analogue to digital converter (ADC) the system clock frequency can be prescaled in run-time to meet the specifications given in the datasheet [19]. This will enable the system to operate down to a lower voltage and will therefore better utilize the battery capacity.

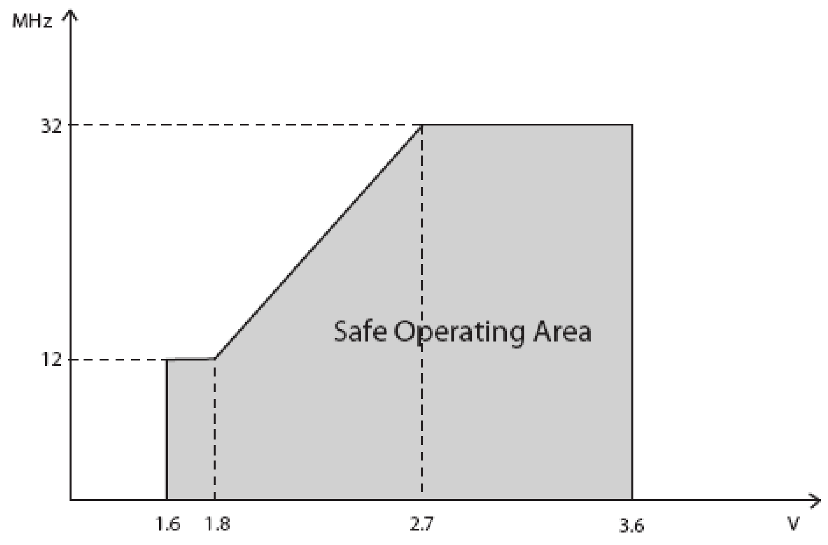


Figure 1 - XMEGA system clock frequency vs. VCC

2.1.3 Specific battery considerations

Section 1.1 specifies that the battery to be used should be a D-cell Tadiran 19Ah lithium battery [10]. It is important that the electrical characteristics of the battery are studied in order to design a system with optimal battery lifetime. The most important electrical specification for this application is the capacity and voltage.

Capacity
Nominal capacity is 19 Ah. It is though important to notice that this only applies if the battery is discharged continuously with a current of 4mA down to 2V at 25̊C. Studying the capacity vs. current and temperature uncovers that 4mA and 25̊C is the optimum operating conditions. It follows that discharging the battery at other current and temperatures will result in a considerable decrease in the total amount of capacity provided by the battery.

The telemetry buoy should function in cold environments. The curve for 0̊C will therefore be considered. The application is specified to operate at least one year. Studying the curve for 0̊C in Figure 2 gives the average current to be approximately

$$I_{avg} = \frac{13Ah}{24h \cdot 365} = 1.48mA \tag{2.1}$$

This will not be perfectly accurate since the function for the curve is unknown. But it shows that the battery must be considered to be about 13mA and not 19mA as in the nominal specifications.

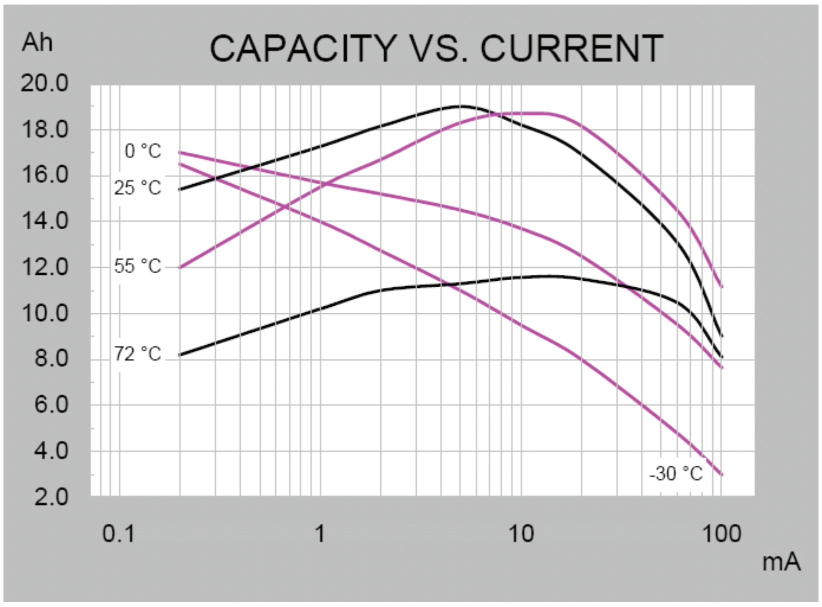


Figure 2 - TL-5930 lithium battery capacity vs. voltage and temperature

Voltage

The voltage will vary with load, temperature and the remaining capacity. Studying Figure 3 shows that the voltage will drop excessively in cold environments and with higher loads. It is therefore important to know that the voltage typically will vary and drop considerably in periods with high peak loads. Figure 4 shows that with a discharge current of 4mA the system must be able to handle voltages down to 2V to be able to utilize the last one thousand hours of operation. With an average current of about 1.48mA, as calculated earlier, the operation time will increase with $1000 hours \cdot 4/1.48 \approx 2700 hours \approx 112 days$ if the system can handle voltages down to 2V instead of 3.5V. This accounts for 31% of the total capacity. It is therefore vital that the system is able to operate at a voltage lower than 3.6V. The system should therefore operate at 2V - 3.7V in order to utilize the capacity of the battery. If the system fulfils this requirement the average current of 1.48A and the capacity of 13mA are valid numbers.

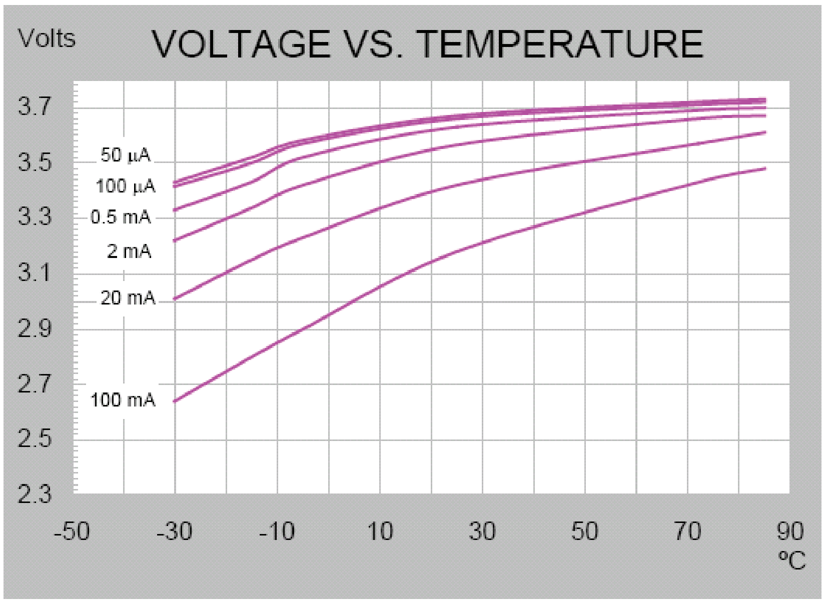


Figure 3 - TL-5930 lithium battery voltage vs. Temperature

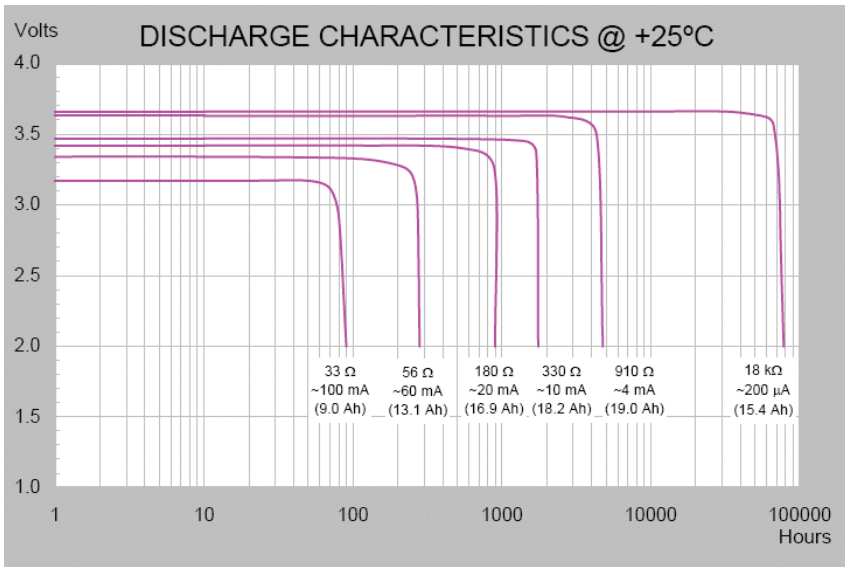


Figure 4 - TL-5930 Lithium battery discharge characteristics

2.1.4 Processor current measurement and estimation

This section presents different techniques to measure and estimate the current, and thereby the power consumption of an embedded processor. The information is based on [2] Chatherine H. Gebotys chapter: *Low-Power Software Techniques*.

2.1.4.1 Estimation models for power consumption

The current drawn by a processor is caused by two factors, leakage and gate switching. Previous methods for estimating power at a high level using gate-level or architecture-level simulations were either inaccurate or too time consuming. For an embedded designer a more practical approach is needed. An easy model to predict the current is to use estimated computation time and the average current. However this introduces large inaccuracies.

Several models for estimating the power consumption has been developed. The different models present different complexities and accuracy. Gebotys categorizes the models into three different types depending on the type of data used to calculate the estimated power. The models with accuracy results are summarized in Table 1**.**

Table 1 - Software power prediction models

Model	**Estimation factors**	**Maximum Error**	**R^2**
Algorithm level	Number of additions, multiplications	3.7%	0.63
Instruction level	Number and type of assembly instructions	2.7%	0.78
Switching level	No. of loads, program address bus access, subtracts, instruction register access, data bus access.	2.1%	0.89

Note that the *Maximum Error* is a value based on tests. The *algorithm* type of model is based on the number of additions and multiplications only. This is an old-fashion way of predicting the execution time and will typically present a larger error due to hardware multipliers. The *instruction* type calculates the current based on tests made for each assembly instruction. This will give better results, but the actual execution of the program can in most cases not be predicted. The most complex way of predicting the current is the *switching* type. This method estimates the current based upon the average switching in registers and busses. However, this requires an instruction-level simulator with switching activity to which embedded system designers typically do not have access.

Typically we find that the smaller the compiled code, the shorter execution time and thereby the overall power consumption.

2.1.4.2 Methods for measuring average current consumption

There are mainly two different ways of measuring the average current consumption of a processor:

1. Multimeter / amperemeter
2. Oscilloscope and series resistor

These are intuitive methods and it is clear that using an oscilloscope provides more detailed information than using a multimeter. Gebotys [2] also recommends using decoupling capacitors. In some cases it can also be a good idea to utilize a current mirror to avoid having a series resistor with the CPU and thereby improving noise conditions. The importance of running the same routine in a loop to get more accurate results is also pointed out.

2.1.5 Specific power consumption considerations

This section describes considerations, specified by the manufacturer, for minimizing power consumption in a microcontroller. The devices studied are limited to Atmel AVR XMEGA, Atmel AVR32 UC3 and Texas Instruments MSP430.

Sleep modes enables the microcontroller to shut down unused modules to save power. When the device enters sleep mode, program execution is stopped. Interrupts or reset is used to wake the device again. Moreover, the individual clock to unused peripherals can be stopped during normal operation to save power.

The wake-up time for the device is dependent on the sleep mode and the frequency of the main clock source. The start-up time for the system clock source must be added to the wake-up time for sleep modes where the clock source is stopped. The ability to get into and out of the low-power modes and process data quickly is crucial because current is wasted by the CPU waiting for the clock to become stable [5]. Some MCUs have a two-stage clock wake-up providing a low-frequency clock to the CPU while a high-frequency clock is being stabilized. On these devices the CPU may be operational in a short time, but running inefficient due to the low frequency.

For the XMEGA [3] the power reduction register provides a method to stop the clock to individual peripherals. This can be used in sleep modes to reduce overall power consumption significantly. On some Texas Instruments devices [5] the peripherals have the ability to disable themselves automatically when not in use. For more complex devices like the AVR32 UC3 [6] the ability to control the clock is at a much more detailed level. The synchronous clock generator can adjust the performance of the system, according to the current requirements, by switching between three different clock sources; internal RC-oscillator, PLL0 and oscillator0. Depending on the developers design it may be better to scale the clock instead of switching the source. This can be done in most MCUs "on-the-fly". In addition to scaling and switching the synchronous clock for the CPU, the clock for each of the internal buses can be scaled down when the bus is not utilized completely. The DMA controller allows the bus to work at a different speed then the CPU. Hence having a DMA can lower the power consumption both in active and sleep mode [5].

Leakage current is sometimes overlooked when choosing a low-power MCU, but it must be considered for the most demanding low-power applications [5]. For the UC3 [6] all pins that are not connected externally to pull-ups, pull-downs, ground or power should be left as inputs, but with the internal pull-up enabled. This will ensure a stable digital level while reducing the input current with the internal resistor and thereby ensure the lowest possible power consumption

The XMEGA Manual [3] lists a few tips and tricks for minimizing power consumption. These are summarized in the following:

- In order to make use of the sleep modes the system must be event driven. Use interrupts instead of polling to be able to sleep until an event occurs.
- Whenever the application has to wait for an event, set the device in idle sleep mode or a deeper sleep mode.
- Instead of using a delay-loop, use the sleep function in combination with a timer. This will allow the CPU to sleep instead of counting.

The specific power consumption of the modules on the different devices is of course also a very important factor. This are covered in chapter *3*.

Available microcontrollers.

2.2 Underwater acoustic communication and signal modulation

Underwater acoustic signals may be modulated in several ways. In fish telemetry the main underwater communication challenges that must be overcome are

- Signal attenuation
- Multipath propagation
- Noise
- Transmission collision
- Low-power transmission

There are several ways of modulating the signal. In fish tags the most common modulation scheme is pulse position modulation (PPM). This project is therefore focused on PPM, but it is important to consider alternative methods when designing the prototype to include compatibility with future products.

Kilfoyle and Baggeroer states in an IEEE report [8] that the frequency content of an underwater telemetry signal remains largely contained within its original band whereas the amplitude and phase of the signal can vary widely in both time and space. This observation naturally concludes that modulation schemes using pulse position or frequency are of the most interest. For this reason this section will cover pulse position, binary and multiple frequency shift keying and spread spectrum technology.

2.2.1 Transmission protocol for Vemco fish-tags

Existing digital acoustic fish tags use pulse position modulation (PPM) to send information. The package includes an ID and can include specific information such as temperature, depth, salinity, heart rate, checksum and more.

The most common format that is used is defined by Vemco. The description is provided by Thelma and is in essence as follows

- Differential pulse-position modulated signal, see chapter 2.2.2 for a description.
- Pulse length T_p = 10ms for all pulses
- Timeslot T_s = 20ms, pulses occur in the middle of the timeslot
- There are M = 4 bits coded by each pulse, this gives 2^4 = 16 time slots
- A transmission starts with two synchronization pulses. The difference in time between the pulses T_{sync} corresponds to the amount of data that will be transmitted i.e. the number of expected pulses.
- A guard time T_g it added between all pulses to compensate for any multipath or echo distortions.
- A transmission ends with an 8-bit CRC checksum.
- All pulses are detected on a rising edge.
- Timing is always referred to the rising edge of a pulse.

Figure 5 gives an example of a transmission of the data 30 and checksum 5. In addition to the listed specifications the following applies for the example:

- T_g = 380ms, guard time to suppress multipath distortion
- T_{sync} = 360ms, it is assumed that this means that one byte + CRC will be transmitted

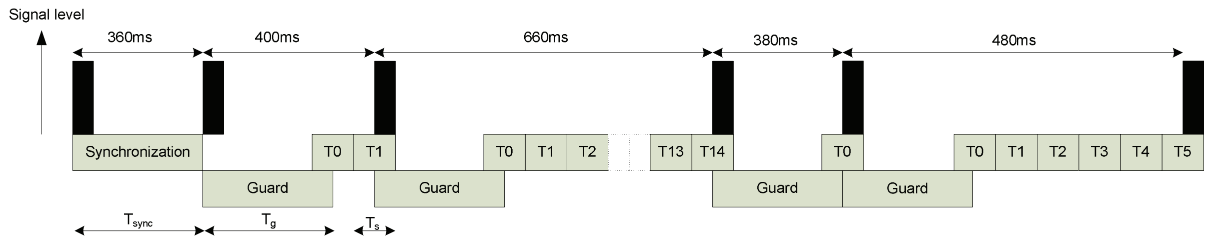


Figure 5 - Example of a Vemco DPPM packet

The time delays between the pulses are calculated by the transmitter as in Table 2.

Table 2 - Composition of a DPPM Vemco signal

Delay between	Delay components	Information	Delay
Pulse 1-2	Synchronization time $T_{sync.}$	Defined by protocol, specifies a total of 6 pulses i.e. 1 byte data	360ms
Pulse 2-3	Guard time T_g + 1 x Timeslot T_s,	Data[7:4] = 0001_2	400ms
Pulse 3-4	Guard time T_g + 14 x Timeslot T_s, data:	Data[3:0] = 1110_2	660ms
Pulse 4-5	Guard time T_g + 0 x Timeslot T_s	CRC[7:4] = 0000_2	380ms
Pulse 5-6	Guard time T_g + ¨5 x Timeslot T_s	CRC[3:0] = 0101_2	480ms
		SUM	2280ms

2.2.2 Pulse-position modulation

In pulse position modulation (PPM) the signal is coded by the position of a series of pulses [9]. A transfer starts with a synchronization pulse followed by a series of timed pulses. A message section of M bits is encoded by transmitting a single pulse in one of 2^M possible time-shifts. Several message sections can be used with the same synchronization pulse to lengthen the message. An example with M = 3 where the binary message 011 is sent is given in Figure 6. The fact that the second pulse is in timeslot four after the synchronization pulse shows that the transmitted code is 011.

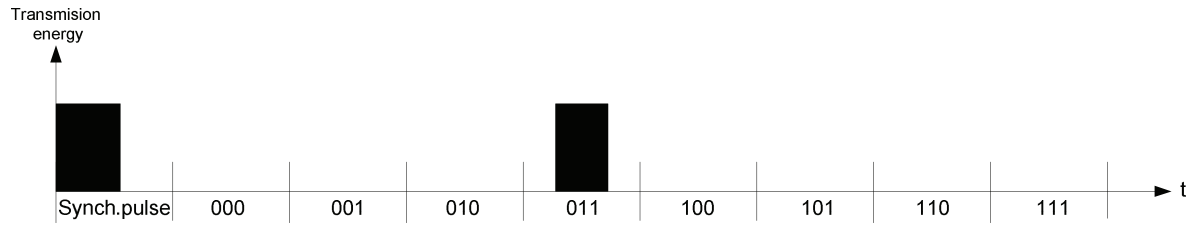


Figure 6 - PPM example sending 011

If more than three bits are to be sent in the example above the timeslots for data will be repeated N/M times where N is the total number of bits in one package.

The average data rate can be improved by always using the last pulse as a new synchronization pulse. This form of PPM is known as differential pulse-position modulation (DPPM) and is the type of modulation used by Vemco and Thelma's fish tags. A basic example with N=6 and M = 3, transmitting the code 010011 is given in Figure 7.

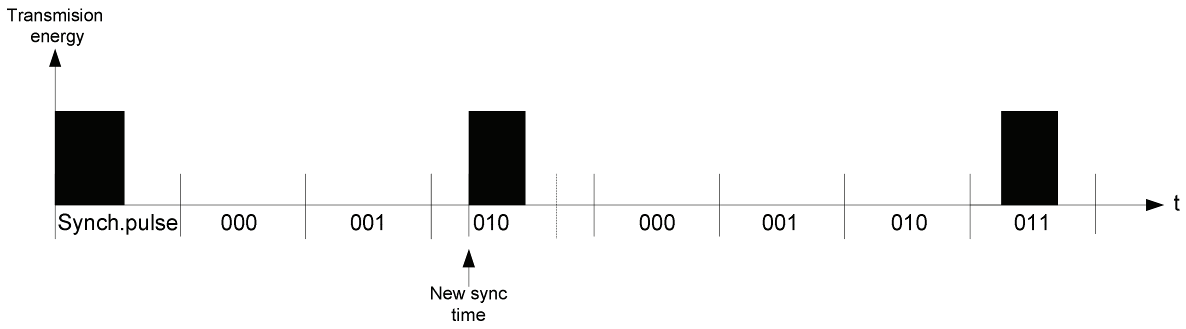


Figure 7 - DPPM example, transmitting 010011

The receiver must know how many bits that will be transmitted. This can wither be fixed or determined by the first pulses. The latter is the case with VEMCO fish tags.

2.2.3 Frequency shift keying modulation

Frequency shift keying (FSK) use distinct tonal pulses to denote digitized information [7]. In binary FSK there are two frequencies, one for logical 0 and one for logical 1. Figure 8 shows an example of BFSK. Multiple frequencies can be used to decode k bits per frequency using 2^k different frequencies. This is referred to as multiple FSK (MFSK).

For acoustic telemetry this modulation form can provide a higher throughput than PPM due to its better immunity against multipath distortion if the same frequency is not repeated too frequently. It will not add much complexity for the fish tag, but the receiver must be able to separate the different frequencies which will add complexity to the digital or analogue signal processing.

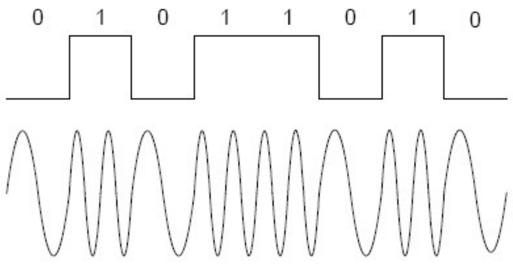


Figure 8 - Example of binary shift keying

2.2.4 Spread spectrum technology

The spread spectrum technique was initially developed for military and intelligence requirements [7]. The idea is to modulate the signal so as to increase the bandwidth of the signal to be transmitted. The main goals are to make jamming and interception more difficult and improve reception in wireless communication. In an acoustic telemetry system for fish tags this is relevant to improve noise immunity, typically in situations with rainfall and situations where many fish tags are transmitting at the same time. There are two different types of spread spectrum:

- Frequency-hopping spread spectrum
- Direct sequence spread spectrum

In addition to these methods, spread spectrum often incorporates a technique called *Code division multiple access (CDMA).*

Several benefits can be gained using spread spectrum including

- Immunity from various kinds of noise and multipath distortion.
- Encryption of signals. The receiver must know the spreading code to recover the information.
- Several users can independently use the same bandwidth with very little interference.

In fish telemetry spread spectrum is not widely used, but there is ongoing research to implement this technique. The main disadvantage is added complexity and therefore an increase in power consumption. In systems where long battery life is not the main priority, but signal robustness is important, spread spectrum may be a good choice.

2.2.4.1 Frequency-hopping spread spectrum

With frequency-hopping spread spectrum (FHSS), the signal is broadcast over a seemingly random series of frequencies, hopping from frequency to frequency at fixed intervals. Typically there are 2^k carrier frequencies forming 2^k channels. The transmitter operates in one channel at a time for a fixed interval and then changes to another channel according to the spreading code.

For transmission the data are fed into a modulator using some digital-to-analogue encoding scheme, such as frequency shift keying (FSK) or binary phase shift keying (BPSK). The signal is then multiplied with the frequency given by the spreading code. If we consider Figure 9, we have an FSK modulated signal as in equation (2.2).

$$s_d(t) = A\cos(2\pi f_0 + 0.5(b_i + 1)\Delta f)t), \quad for\ iT < t < (i+1)T \tag{2.2}$$

Where

A = amplitude of signal

f_0 = base frequency

b_i = value of the ith bit of data (+1 for binary 1, -1 for binary 0)

Δf = frequency separation

T = bit duration; data rate = 1/T

Multiplying this signal with the spread spectrum code frequency

$$c(t) = \cos(2\pi f_i t) \tag{2.3}$$

and using the trigonometric identity $\cos(x)\cos(y) = 0.5(\cos(x+y) + \cos(x-y))$ gives us the signal

$$p(t) = 0.5A[\cos(2\pi(f_0 + 0.5(b_i + 1)\Delta f + f_i)t) + \cos(2\pi(f_0 + 0.5(b_i + 1)\Delta f - f_i))t] \tag{2.4}$$

Applying a bandpass filter to block the difference-frequency and pass the sum-frequency gives the resulting signal

$$s(t) = 0.5A\cos(2\pi(f_0 + 0.5(b_i + 1)\Delta f + f_i)t) \tag{2.5}$$

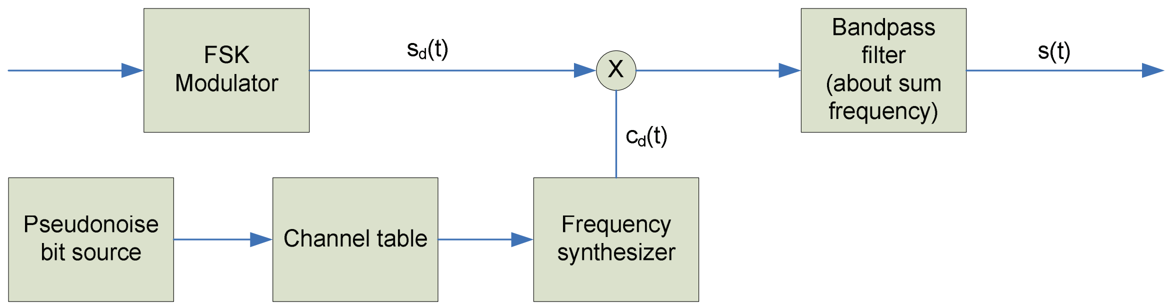


Figure 9 - FSK modulated spread spectrum transmitter

Using the same derivation method for Figure 10 gives us the equation for the received signal described by (2.6).

$$p(t) = 0.25A\cos(2\pi(f_0 + 0.5(b_i + 1)\Delta f)t) \tag{2.6}$$

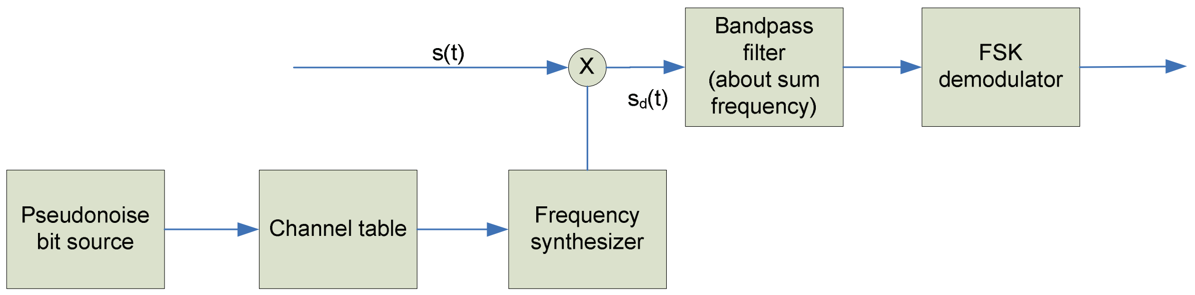


Figure 10 - FSK modulated spread spectrum receiver

Frequency hopping spread spectrum is commonly used with multiple frequency shift keying (MFSK). An example is illustrated in Figure 11 where

- W_m is the bandwidth for each frequency in the MFSK modulation using 2^2 channels, coding two bits per frequency.
- W_d is the bandwidth for each of the spread spectrum frequency hop frequencies
- W_s is the total bandwidth of the spread spectrum signal.

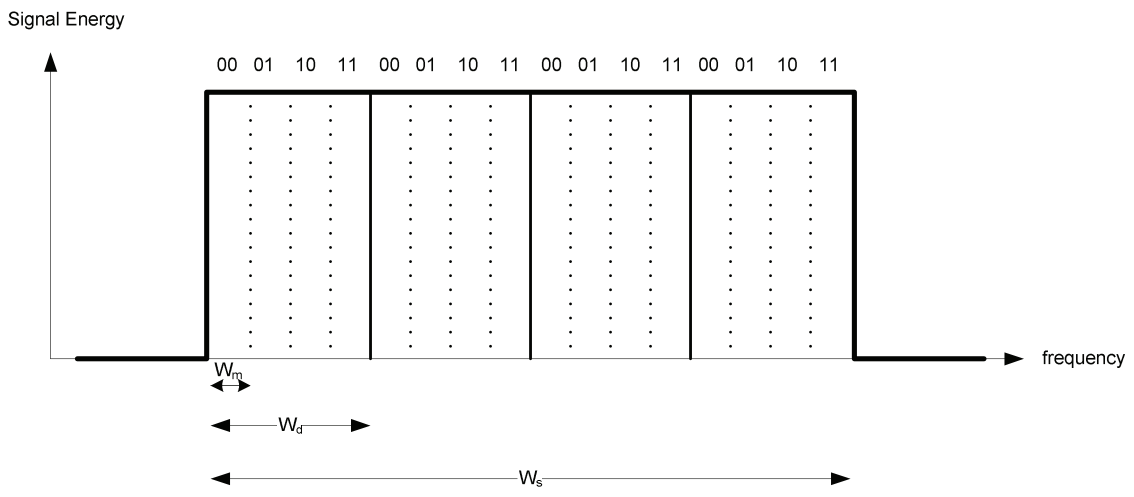


Figure 11 - Spread spectrum with MFSK

2.2.4.2 Direct sequence spread spectrum

With direct sequence spread spectrum (DSSS), each bit in the original signal is represented by multiple bits in the transmitted signal, using a spreading code. As a consequence, the signal is more immune against jamming, but the technique introduces a rather high level of redundancy in the transmitted signal. Since more information is transmitted, this technique will lead to a substantial increase in power consumption. This is not acceptable in most fish-telemetry applications and this technique will therefore not be discussed further.

2.2.4.3 Code division multiple access

Code division multiple access (CDMA) is a technique used with spread spectrum. As with DSSS, this technique also introduces redundancy in the transmitted signal. Instead of sending a single bit the transmitter sends a specific code. This enables the receiver to separate nodes when several nodes are communicating at the same time. Since this will result in a substantial increase in the time that the fish-tag will have to send information and the computing time of the receiver, this technique will not be discussed further.

2.3 Digital signal processing

It is desired to implement as much as possible of the signal processing in a digital form. There are several reasons why digital processing is preferred. The most important factor is that it is more flexible and the application specifications can easily be altered by updating the software instead of changing the hardware. Secondly it will result in fewer electronic components and may therefore be cheaper and the probability of failure is decreased. In addition, using a digital approach will provide immunity against external noise once digitalized. The downsides include introduction of time-delay, added power consumption and quantization error.

2.3.1 Sampling

Preserving all the information in a signal requires that the signal is sampled according to the Nyquist theorem

$$F_s > 2F_{\max} \tag{2.7}$$

Where
F_s = sampling frequency
F_{max} = Maximum frequency of the sampled signal

This implies that there must be some kind of analogue low-pass filtration to ensure that frequencies above F_{max} are not sampled. This is to avoid aliasing.

If the valid information in the signal is limited to a known frequency band, aliasing can be exploited. If the analogue signal is band-pass filtered so that minimum lower frequency F_L and maximum higher frequency F_H are known a lower sampling frequency than the Nyquist frequency can be used. Harris refers to this as an *aliasing digital down converter using subsampling* in *Multirate signal processing for communication systems* [16].

The sampling frequency must comply with [17]:

$$\frac{1f_H}{k} \leq f_s \leq \frac{2f_L}{k-1} \tag{2.8}$$

Where k is an integer satisfying the condition:

$$1 \leq k \leq \left\lfloor \frac{f_H}{f_H - f_L} \right\rfloor \tag{2.9}$$

To minimize the cost of the analogue band-pass filter the signal band is arranged to alias to one fourth of the selected sample-rate. Aliasing to the quarter-sample rate maximizes the separation between the positive frequency and the negative frequency, which permits the maximum transition bandwidth of the analogue band-pass filter. Aliasing the centre frequency f_C to the quarter sample rate during the sampling process is assured if the sample rate satisfy (2.10). The k + ¼ option aliases the signal to the positive quarter-sample rate while the k – ¼ option aliases the signal to the negative quarter sample rate. The use of two options simply introduces more possible sample rates [16].

$$f_C = k \cdot f_S \pm \frac{1}{4} f_S \tag{2.10}$$

3. Available microcontrollers

The telemetry buoy application will include some kind of digital processing unit, due to the low power requirements this will be a microcontroller instead of a digital signal processor. This chapter evaluates several microcontrollers with respect to low-power, ADC and processing features. The goal is to get a general overview of some of the products to be able to choose the most suitable microcontroller (MCU).

The most important peripherals the microcontroller for this project should include are shown in Figure 12. Some of the modules can of course be external, but internal modules are preferred due to power and size requirements. The optional peripherals or functionalities are drawn with a dotted line. The *"performance enhancement modules"* block can include features such as DMA, event system, independent timers, etc. These modules are not directly required, but must be considered since they typically increase the overall performance and allow the CPU more time in sleep modes.

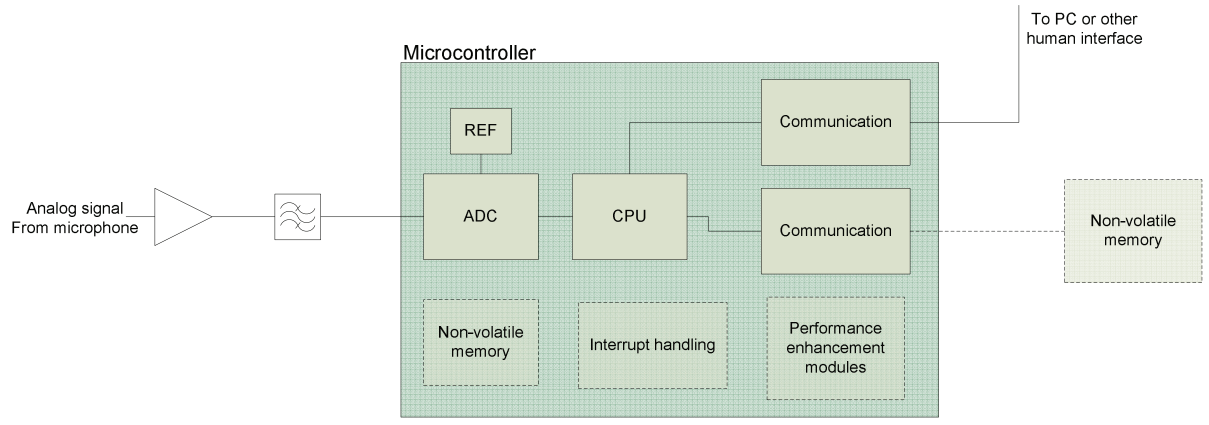


Figure 12 - Simplified application functionality

The focus is on Atmel AVR, AV32 and the MSP430. This is due to their market leading place as the ultra low power microcontroller.

3.1 Atmel AVR ATmega and ATtiny series

Atmel AVR picopower devices are preferred when low-power and low-cost are the main priorities for the application. To provide full flexibility it is desired that the microcontroller samples the analogue signal and performs some kind of digital signal processing to retrieve the transmitted data. The AVR mega and tiny devices provides an internal ADC with a maximum sampling frequency of $f_s = 77$ kHz. According to the Nyquist sampling theorem this limits the input signal bandwidth to a maximum of 38.5 kHz. Today's fish tags typically send information at 69 kHz. The internal ADC of the ATmega and ATtiny devices is therefore not suitable for direct Nyquist sampling of the signal.

Since low-cost is not a priority for this project the focus will be on AVR XMEGA and AVR32 UC3 series which typically provide more performance while still providing ultra low power features.

3.2 Atmel AVR XMEGA series

The only Atmel AVR XMEGA devices available today is the ATxmegaA1 series. The XMEGA series is profiled as a second generation picopower device. It is based on the AVR core, but has several new features, operates at a higher frequency and a lower supply voltage. The CPU performance is greatly improved by introducing special function registers such as dedicated set bit, clear bit and toggle bit registers for ports. This reduces the read-modify-write operation to a single modify operation and hence the performance is greatly improved. The following chapter will give a more detailed description of the new features of the XMEGA A1 series.

3.2.1 ATxmegaA1 features and specifications

The XMEGA series is based on the 8-bit AVR MEGA series of microcontrollers. The basic features of the AVR MEGA are still present, but more features are added. The ATxmegaA1 features most relevant to this project are [19]:

- 1.6 – 3.6 V operation
- Direct Memory Access controller (DMAC), four channels
- Real time clock (RTC)
- Serial peripheral interface (SPI)
- Universal Synchronous and asynchronous serial receiver and transmitter (USART)
- Infrared communication module (IrCOM)
- Two 8-channel Analogue to digital converters (ADC), up to 2 MSPS, 12 bit resolution.
- Digital to analogue converter (DAC), up to 2 MSPS
- Analogue comparator (AC)
- External bus interface supporting SRAM and SDRAM

Some features are of special interest. These are the DMA, ADC, event system, clock distribution system and the power management module.

DMAC

The DMAC will greatly reduce the overall current consumption by allowing the CPU to sleep while data from the ADC is stored in memory. In the idle sleep mode the MCU will draw a current of about 7.1mA (3V, 32MHz) instead of 18mA (3V, 32MHz) in active mode. This reduces the current by a factor of 2.5 when transferring data.

ADC

The ADC on the XMEGA can operate at up to 2000 MSPS with a resolution of 12-bits. It can therefore, according to the Nyquist theorem, sample analogue signals of up-to 1 MHz. The internal ADC is therefore capable of both normal sampling and burst oversampling of the 69kHz acoustic signal of VEMCO standard fish tags.

Event system

The event system provides CPU and DMA independent inter-peripheral communication. It enables the possibility for a change state in one peripheral to automatically trigger actions in one or more other peripherals. All events from all peripherals are routed into the event routing network as shown in Figure 13. This consists of eight multiplexers where each can be configured in software to select which event to be routed into that event channel. All eight event channels are connected to the peripherals that can use events, and each of these peripherals can be configured to use events from one or more event channels to automatically trigger a software selectable action.

Utilizing the event system will allow the CPU more time in sleep modes and a faster system response time. It is therefore one of the most important power saving features of the XMEGA series.

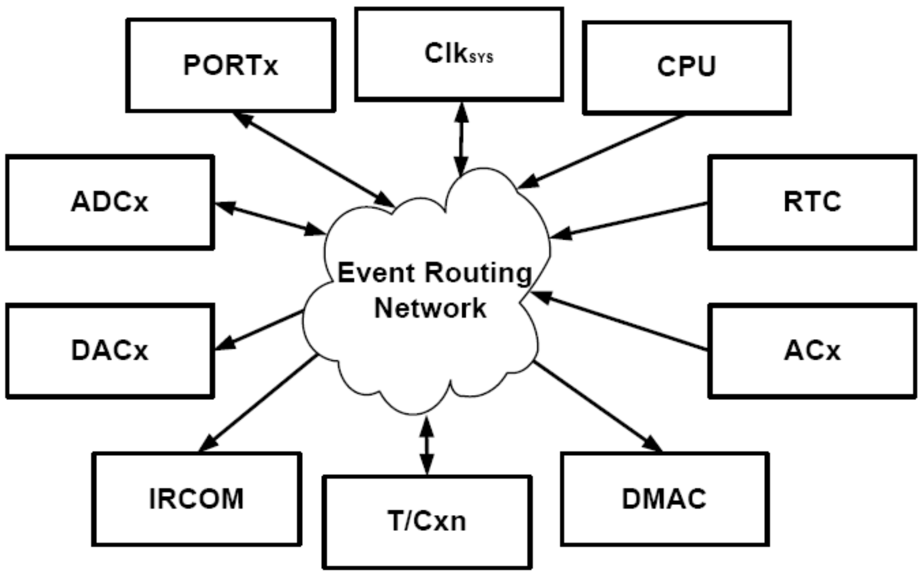


Figure 13 - Event system block diagram

System clock
The ATxmegaA1 provides the possibility of using several internal or external oscillators. Compared to the AVR mega series, the XMEGA has several new features including the ability to change system clock source during run-time. This allows the programmer to combine the advantage of a high accuracy crystal oscillator with the low power feature of an internal RC oscillator. The device also provides automatic run-time calibration of the internal 32 MHz RC oscillator with an external 32.768 kHz crystal as reference. This crystal can also be used as the source to the internal Real Time Clock (RTC). An ultralow power 32 kHz internal oscillator can be used if the RTC does not require the high accuracy of a quartz crystal.

The XMEGA, as most other AVR devices, provides the ability of prescaling the clock source. The designer can thereby tailor the system clock frequency in run-time to achieve optimal performance at different supply voltages. Please refer to section *2.1.2.1 System design techniques* for details.

Power management and sleep modes
The ATxmegaA1 provides five different sleep modes. These are listed and described in Table 3. In addition the XMEGA provides a power reduction register which enables the programmer to individually disable internal peripherals in run-time.

Table 3 - XMEGA sleep modes

Sleep mode	**Description**	**Current @ 3V, 32 MHz** [(1)]
Idle	In idle mode the CPU and the non-volatile memory are stopped, but all peripherals are running. Interrupt requests from all enabled interrupts will wake the device.	8.15mA
Power-down	In power-down mode all system clock sources, and the asynchronous RTC clock source, are stopped. This allows the device to use a minimum of power. Only external pin-change interrupts and Two-wire address match interrupt can wake the device.	2uA[(2)]
Power-save	Power-save mode is identical to power-down with one exception; If the RTC is enabled, it will keep running during sleep. The device can also wake up from RTC interrupts.	TBD
Standby	Standby mode is identical to power-down with the exception that all enabled system clock sources are kept running, while CPU, peripheral and RTC are stopped. This reduces the wake-up time when external crystals or resonators are used.	TBD
Extended standby	Extended standby mode is identical to standby mode with the exception that the RTC is enabled.	TBD

Note 1: All numbers are typical characteristics with external clock
Note 2: All functions disabled

The current consumption numbers origin form the ATxmegaA1 datasheet [3]. Since the device is new, not all sleep modes are characterized and are marked with TBD.

3.3 Atmel AVR32 UC3 series

The Atmel AVR32 UC3 is a 32-bit microcontroller featuring DSP instructions and achieve up to 83 DMIPS at 66 Mhz and an energy efficiency of 1.3mW/MHz [12].

3.3.1 Comparing the UC3A and the UC3B

There are two types: UC3A and UC3B. The UC3A datasheet [13] and the UC3B datasheet [14] lists typical power consumption under the following conditions:

1. CPU running from flash
2. CPU clocked from PPL0 at the frequency given in Table 4
3. Voltage regulator is on.
4. XIN0: external clock
5. XIN1 stopped. XIN32 stopped
6. PLL0 running
7. All peripheral clocks activated
8. GPIOs on internal pull-up
9. JTAG unconnected with external pull-up

A comparison of UC3A and UC3B is given in Table 4. Figure 14 shows that the UC3A consumes a considerable more amount of power than the UC3B in active mode.

Table 4 - power consumption for AVR32 UC3A and UC3B

Mode	Frequency [MHz]	Consumption, typical		Unit
		UC3A	UC3B	
Active	12	9	5,5	*mA*
Active	24	16	10	*mA*
Active	36	23	14,5	*mA*
Active	50	31,5	19,5	*mA*
Active	60	37	23,5	*mA*
Static	N/A	25	15,5	*uA*

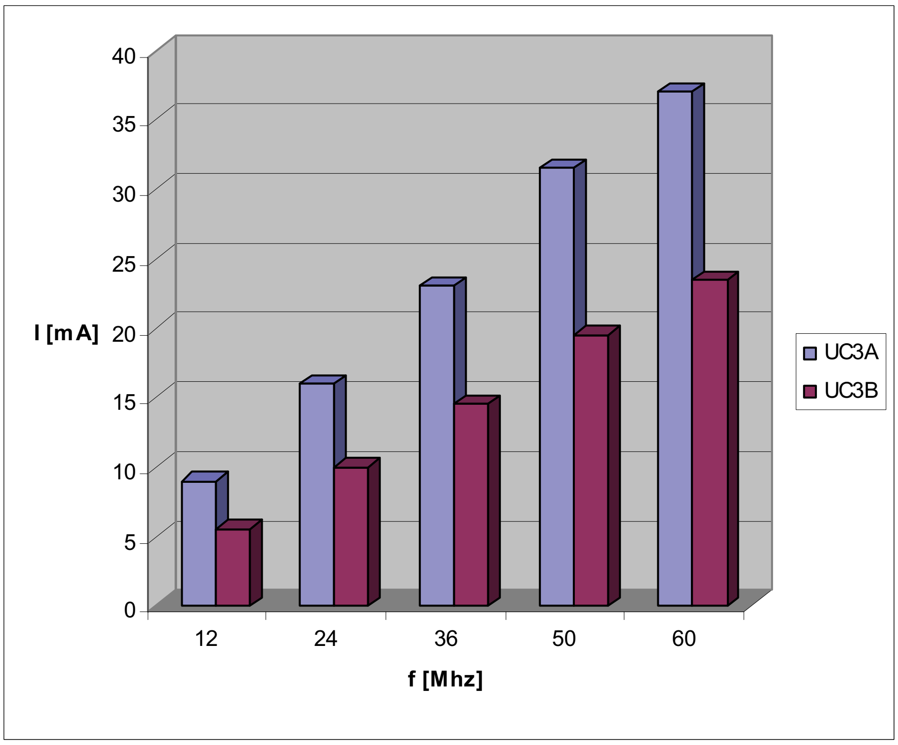


Figure 14 - Power consumption of AVR32 UC3A and UC3B in active mode

The datasheets also report power consumption numbers for each peripheral module in active mode. This also shows that the AC3A will, in average, consume more power than the UC3B.

The main difference between the UC3A and the UC3B with respect to features is that the AC3A has got an inbuilt support for an Ethernet MAC 10/100 Mbps interface. The UC3A also has some extra I/O capabilities like an extra SPI and an extra USART. Low-power consumption will be the main priority of this project and the following section will therefore focus on the UC3B.

3.3.2 UC3B features and specifications

The most relevant features for the Atmel AVR32 UC3B [14] are

- Up to 75 Dhrystone MIPS at 60 Mhz
- Seven Peripheral DMA channels
- Single cycle execution
- Universal Serial Bus (USB 2.0)
- Three USARTs

- SPI
- 8-channel 10-bit ADC working at up to 384kSPS, consuming 90uA
- Operating voltage: 2.7 – 3.6V

The UC3 microcontroller has got good performance; the main disadvantage is that the MCU requires a boost regulator to operate at voltages below 2.7V. Section 2.1.3 specifies that in order to fully utilize the energy of the battery the system must operate on voltages as low as 2.0V. The power loss of such a regulator must therefore be taken into account when calculating the total system current consumption.

3.4 Other vendors

The ultralow power market is dominated by Atmel with its AVR devices and by Texas Instruments with its MSP430 series. There are many other vendors on the market such as Freescale, Holtek, Maxim and MicroChip, but these consume a substantial higher current then the AVR and the MSP430.

3.4.1 Texas Instruments MSP430 series

The MSP430 series is by many regarded as the first ultralow power MCU series. Texas instruments have taken a good market share with the MSP430 due to its ultralow power features at such an early stage as February 1992.

The MSP430 is a 16-bit RISC MCU. It features a von Neumann architecture. Since the device is 16-bits it will have an advantage over the AVR when using mainly 16-bit numbers. The drawback is the von Neumann design which reduces the MIPS per MHz radio considerably. The AVR core is designed using Harvard architecture and is therefore capable of achieving a 1:1 ratio of MIPS per MHz.

3.4.1.1 MSP430 series features and specifications

The most resent addition to the MSP430 series is the MSP430F543xA. The features, found in the MSP430F543xA datasheet [19], most relevant to this project are

- 1.8V – 3.6V operation
- Power management system
- Unified clock system
- UART
- IrDA Encoder and Decoder
- SPI
- I^2C
- 12-bit, 12-channel ADC
- Three channel DMA
- Internal RTC

Four features are of special interest; the Power management system (PMS), the Unified clock system, the DMA and the ADC. These will be described in the following [21].

Power management system and sleep modes

The MCU includes an integrated low-dropout voltage regulator which can be programmed to four different voltages. Reducing the core voltage will reduce the ability to work at high frequencies as shown in Figure 15. This enables the designer to program the MCU to use which ever voltage found suitable at the given situation. If full CPU performance is not required the core voltage can be reduced to save power.

The MCU supports five different sleep modes, LPM0 – LPM4 turning off different modules and clock sources. Please refer to the *MSP430x5xx Family User's Guide* [21] for details.

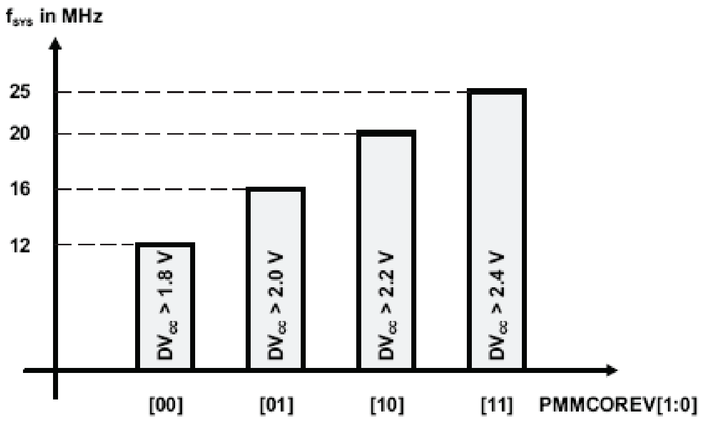


Figure 15 - System frequency and core voltage

Unified clock system

Using three internal clock signals, the user can select the best balance of performance and low power consumption at any given time. The clock system can be configured to operate with internal and/or external oscillators. The Unified clock system supports the following as inputs:

- 32kHz low-power external crystal oscillator
- 12kHz very low-power internal oscillator
- Internal calibrated oscillator with 32kHz as typical frequency
- Internally digitally controlled oscillator
- External crystal oscillator

The Unified clock system provides three outputs:

- Auxiliary clock. An individually scalable clock for each peripheral module.
- Master clock. A scalable clock used by the CPU and system.
- Sub-system master clock. An individually scalable clock for each peripheral module.

Allowing different clock sources to be selected and scaled for each peripheral ensures that the balance between power consumption and performance is optimal at any given time.

DMA

The direct memory access (DMA) controller transfers data from one address to another, without CPU intervention. Using the DMA controller can increase the throughput of peripheral modules. It can also reduce system power consumption by allowing the CPU to remain in a low-power mode without having to awaken to move data to or from a peripheral.

ADC

The ADC features a 200ksps maximum conversion rate with a 12-bit resolution. The internal ADC is therefore capable of Nyquist sampling the 69kHz acoustic signal of VEMCO standard fish tags. Burst/oversampling cannot be done since the sampling rate is limited to just above the Nyquist sampling frequency of $69 \cdot 2kHz = 138Khz$. It is specified in section 1.1 that the system should be able to handle frequencies of up to 100 kHz. The ADC meets this criterion only if we assume an ideal filter or use undersampling as described in section 2.3.1 on page 16.

3.5 Conclusion

The most important criteria for this project is low power. The AVR XMEGA features new and innovative CPU hardware design with single cycle execution and 1.6V operation. The clock source and prescaling can be changed during run-time to optimize power vs. performance at any given time. The DMA and the Event system will allow the CPU and other modules more time in sleep modes than with any other microcontroller. A great number of sleep modes and the ability to disable unused modules will allow the microcontroller to use the minimum amount of power when idle and the internal 32kHz RTC will eliminate the need for an external circuit consuming power.

These features combined with an high performance internal ADC leads to the choice of the ATxmegaA1 as the system microcontroller for the project.

4. Digital signal processing

Using digital signal processing introduces many possibilities. For the telemetry buoy project it includes the ability to

- Use the same HW platform with several different acoustic tags with different carrier frequencies.
- Develop performance enhancing software upgrades.
- Develop new acoustic modulation schemes without having to alter the HW and thereby making today's telemetry buoy compatible with fish tags of the future.

General benefits of digital signal processing (DSP) are given in section 2.3 on page 16.

4.1 Designing a digital band pass filter

DSP covers advanced frequency analysis, power analysis, adaptive filters and much more. Since the algorithm in this project is to be designed for an 8-bit microcontroller and optimized for low-power, it is important that the algorithm is kept simple to allow the microcontroller to execute the algorithm as fast as possible. It is also important that the algorithm easily can be changed to support other frequencies. A band pass filter will provide a relatively simple algorithm while having the flexibility of being used on several frequencies by changing the coefficients. The design is therefore concentrated on designing a digital second order band pass filter [25].

4.1.1 Derivation

A general second order continuous time filter has the form of

$$H(s) = \frac{sG}{(s - j\omega_{0,1})(s - j\omega_{0,2})} \tag{4.1}$$

We place the poles at

$$\begin{aligned} j\omega_{0,1} &= -\pi B + j2\pi f_0 \\ j\omega_{0,2} &= -\pi B - j2\pi f_0 \end{aligned} \tag{4.2}$$

Where
B : Desired band width
f_0: Desired centre frequency
G : Filter gain

Multiplying the divisor of (4.1) gives us

$$H(s) = \frac{sG}{s^2 - s(j\omega_{0,1} + j\omega_{0,2}) + j\omega_{0,1} \cdot j\omega_{0,2}} \tag{4.3}$$

By using (4.2) we get

$$H(s) = \frac{sG}{s^2 - ps + q} \tag{4.4}$$

Where

$$p = j\omega_{0,1} + j\omega_{0,2} = -2\pi B$$
$$q = j\omega_{0,1} \cdot j\omega_{0,2} = \pi^2 B^2 + 4\pi^2 f_0^{\,2} \tag{4.5}$$

We get the discrete version by applying the bilinear transform [15]

$$s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} \tag{4.6}$$

This gives the discrete transfer function

$$H\left(\frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}\right) = \frac{\dfrac{2G}{T(\frac{4}{T^2}+\frac{2p}{T}+q)} - \dfrac{2G}{T(\frac{4}{T^2}+\frac{2p}{T}+q)} z^{-2}}{1 + \dfrac{2q-\frac{8}{T^2}}{\frac{4}{T^2}+\frac{2p}{T}+q} z^{-1} + \dfrac{\frac{4}{T^2}-\frac{2p}{T}+q}{\frac{4}{T^2}+\frac{2p}{T}+q} z^{-2}} \tag{4.7}$$

Comparing (4.7) with the general expression for a second order IIR filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} \tag{4.8}$$

Gives the coefficients

$$\begin{aligned} b_0 &= \frac{2G}{T(\frac{4}{T^2}+\frac{2p}{T}+q)} \\ b_1 &= 0 \\ b_2 &= \frac{2G}{T(\frac{4}{T^2}+\frac{2p}{T}+q)} \\ a_0 &= 1 \\ a_1 &= \frac{2q-\frac{8}{T^2}}{\frac{4}{T^2}+\frac{2p}{T}+q} \\ a_2 &= \frac{\frac{4}{T^2}-\frac{2p}{T}+q}{\frac{4}{T^2}+\frac{2p}{T}+q} \end{aligned} \tag{4.9}$$

The relationship between the frequency variables in the continuous and the discrete domain is highly nonlinear when using the bilinear transform. The relation is given in (4.10) found in [15]

$$\omega = 2\tan^{-1}\left(\frac{\Omega T}{2}\right) = 2\tan^{-1}\left(\frac{2\pi f T}{2}\right) \tag{4.10}$$

Thereby in order to achieve a centre frequency of f_d and a bandwidth of B_d, the filter coefficients should be calculated using (4.11).

$$\begin{aligned} f_0 = f_a &= \frac{\tan(\pi f_d T)}{\pi T} \\ B = B_a &= \frac{\tan\left(\pi T\left(f_d + \frac{B_d}{2}\right)\right)}{\pi T} - \frac{\tan\left(\pi T\left(f_d - \frac{B_d}{2}\right)\right)}{\pi T} \end{aligned} \tag{4.11}$$

This is referred to as prewarping. G can be calculated to provide unity gain at the centre frequency. The gain is given by

$$\begin{aligned} H(s) &= \frac{sG}{s^2 - ps + q} \\ H(j\omega) &= \frac{j\omega G}{(j\omega)^2 - jp\omega + q} \\ |H(j2\pi f_a)| &= \frac{2\pi f_a G}{\sqrt{(q-(2\pi f_a)^2)^2 + (p2\pi f_a)^2}} \end{aligned} \tag{4.12}$$

Using (4.12) , (4.5) and selecting $|H(j2\pi f_a)| = 1$ we get the expression for the value of G that provides unity gain at the centre frequency

$$G = \frac{B_a}{2f_a}\sqrt{\pi^2 B_a^{\ 2} + 16\pi^2 f_a^{\ 2}} \tag{4.13}$$

Where f_a and B_a are given by (4.11).

The relation between the discrete transfer function and the difference equation is given by (4.14).

$$H(z) = \frac{\sum_{i=0}^{P} b_i z^{-i}}{\sum_{j=0}^{Q} a_j z^{-j}} \Leftrightarrow y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right) \tag{4.14}$$

By using (4.14) we find the difference equation for (4.8) to be

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \tag{4.15}$$

Where

x : filter input

y : filter output

b_0, b_1, b_2, a_1, a_2 : Constants given by (4.9)

4.1.2 MATLAB implementation and simulation

MATLAB was used to calculate the coefficients and analyze the filter. The complete code is found in appendix 3 and 5. The filter is analysed with the following parameters

$$\begin{aligned} f_s &= 250kHz \\ f_0 &= 69kHz \\ B &= 500Hz \end{aligned} \tag{4.16}$$

The centre frequency f_0 is based on existing VEMCO tags. The bandwidth B should be as small as possible. The narrower the filter the smaller the coefficients. The constraint is therefore that the smallest coefficients must be large enough to be stored in an 8-bit fixed point number. Please refer to section 4.2 and 5.3.1 for details and argumentation. The sample frequency is based on the XMEGA's clock options. These parameters results in the following coefficients

$$\begin{aligned} b_0 &= 0.0062 \\ b_1 &= 0 \\ b_2 &= -0.0062 \\ a_1 &= 0.3233 \\ a_2 &= 0.9875 \end{aligned} \tag{4.17}$$

The frequency response of the filter is shown in Figure 16.

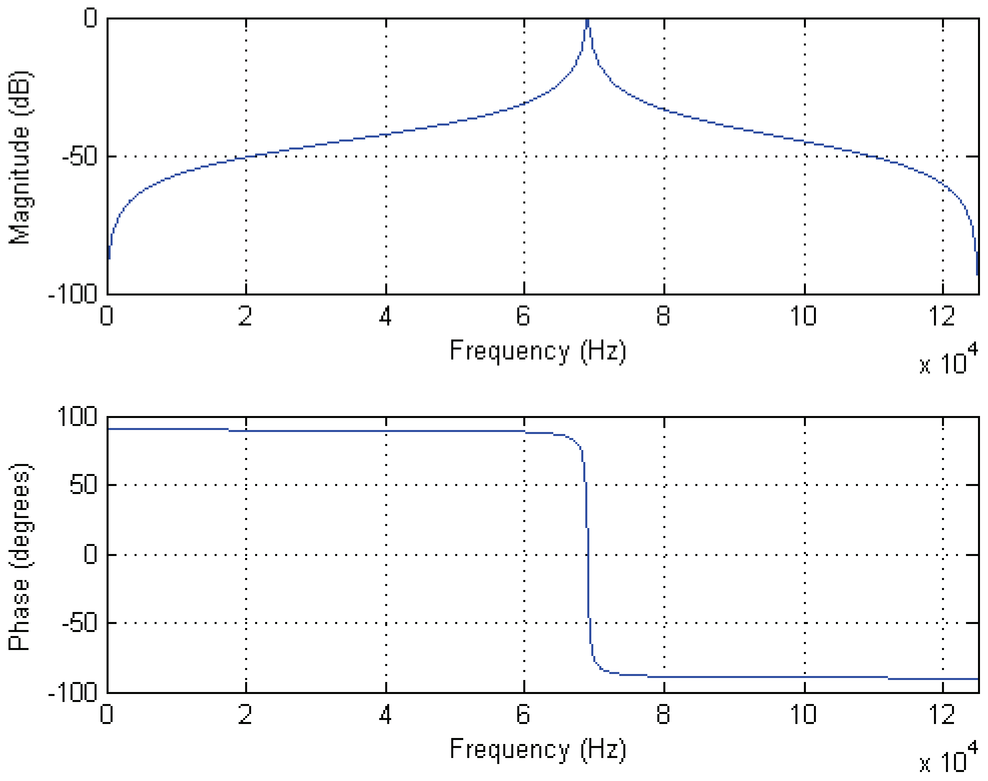


Figure 16 - Filter frequency response with acurate coefficients

Simulating a fish tag signal of 69 kHz combined with Gaussian noise with an SNR = 30dB shows that the algorithm clearly filters out unwanted noise as in Figure 17. The simulation uses N=1000 samples.

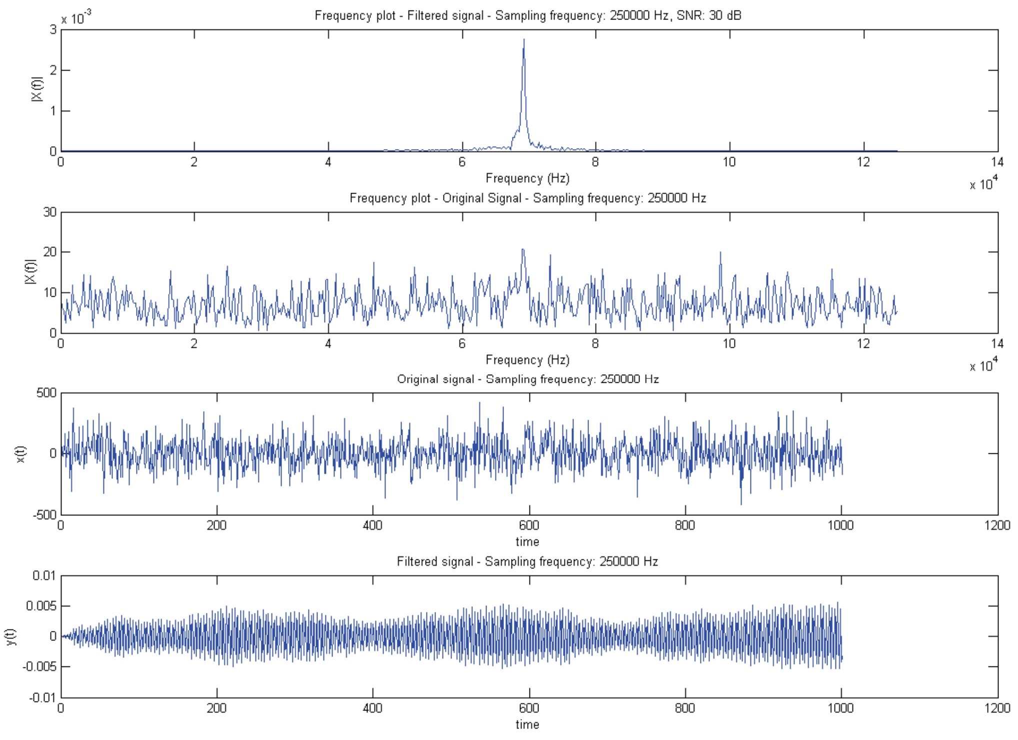


Figure 17 - Filter simulation using precise coefficients

4.2 Number representation

To achieve optimal performance it is vital to use the correct number representation. The standard representation of numbers or variables in the programming language C are listed in Table 5.

Table 5- Variables in C

Type	**Description**	**Size**
char	Signed integer	1 byte
short int	Signed integer	2 bytes
int	Signed integer	4 bytes
long int	Signed integer	8 bytes
float	Signed floating number	4 bytes
double	Signed floating number	8 bytes

The AVR is an 8-bit architecture and does not have hardware support for floating numbers. It is therefore important to avoid using floating numbers whenever possible. Appendix 1 gives some examples of multiplying and dividing different variable types. It is demonstrated that multiplying two float type numbers takes about 44 times longer time than multiplying to numbers of char type. The implementation therefore uses mainly Q1.7 and Q2.6 fixed point representation which can be stored in a single char variable. The details of these formats are covered in the following section.

4.2.1 Q1.7 representation

Q1.7 numbers can represent fixed-point numbers ranging from -1 to 0.9921875 in increments of 0.0078125 (-1 to 1-1/128) [22]. The 8-bit Q1.7 number bit weighting is shown below. The decimal place is between bits 6 and 7.

Table 6 - Q1.7 format

Bit:	7	6	5	4	3	2	1	0
Value	-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128

4.2.2 Q2.6 representation

Eight bit Q2.6 format represent fixed-point numbers ranging from -2 to 1.984375 in increments of 0.015625 (-2 to 2-1/64). The Q2.6 representation bit weighting is shown below. The decimal place is between bit 5 and 6.

Table 7 - Q2.6 format

Bit:	7	6	5	4	3	2	1	0
Value	-2	1	1/2	1/4	1/8	1/16	1/32	1/64

4.2.3 Q3.13 representation

16-bit Q3.16 format represent fixed-point numbers ranging from -4 to 3.9998779296875 in increments of 0.0001220703125. The Q3.13 format bit weighting is shown below.
The decimal place is between bit 12 and 13.

Table 8 - Q3.13 format

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	-4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048	1/4096	1/8192

5. Prototype design

A prototype was designed in order to do testing on the actual hardware. The main purpose is to measure current consumption, running time and the correctness of the output for the individual algorithms. All algorithms are tested on an ATmegaA1. The hardware documentation is found in appendix 4. The assembled prototype is shown in Figure 18.

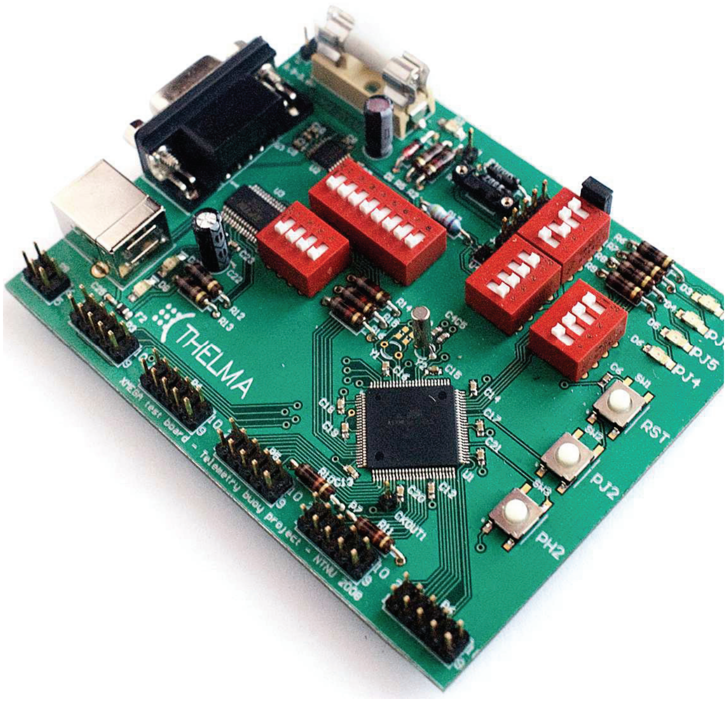


Figure 18 - Picture of assembled prototype

Current consumption measurement
There are several ways of measuring current consumption. The two most usual ways are described in section *2.1.4.2 Methods for measuring average current consumption* on page 9. The prototype is prepared for both series resistor measurement with an oscilloscope and connecting an ampere meter in series. To allow good signal strength for the oscilloscope at low currents a current shunt amplifier is used. Several different series resistor values can be selected using dip-switches. All other components besides the XMEGA which can draw current are connected through switches. It follows that these devices can be fully disconnected when measuring current consumption.

Execution time measurement
Execution time is measured by simply setting an I/O-pin high before executing the routine or algorithm and setting the I/O-pin low when done. This will allow accurate time readings with the oscilloscope. The time it takes to actually set and clear the I/O pin should be taken into consideration when the execution time of the algorithm is short. AVR Studio provides the ability to calculate execution time and the number of clock cycles in simulation mode. This method is applicable when single lines of code or short algorithms are to be tested.

Correctness of the output of the algorithm
The ATxmegaA1 includes a debug interface which allows the programmer to read out variables at any given time using AVR Studio however, when the amount of data is larger then a few variables and require post processing to be able to do analysis, the debug interface

is not suitable. Therefore the prototype includes both an RS232 serial connection and a USB connection. This will allow large datasets to be sent to a computer for processing and analysis.

Other features

In addition to the features mentioned, the prototype includes

- Protection circuitry for high voltage, wrong polarity and short circuit
- LEDs for indication
- Switches for debugging purposes
- Pin headers for programming, debugging, analogue input/output, GPIO and communication busses such as TWI (I^2C compliant), UART and SPI.

5.1 Schematics

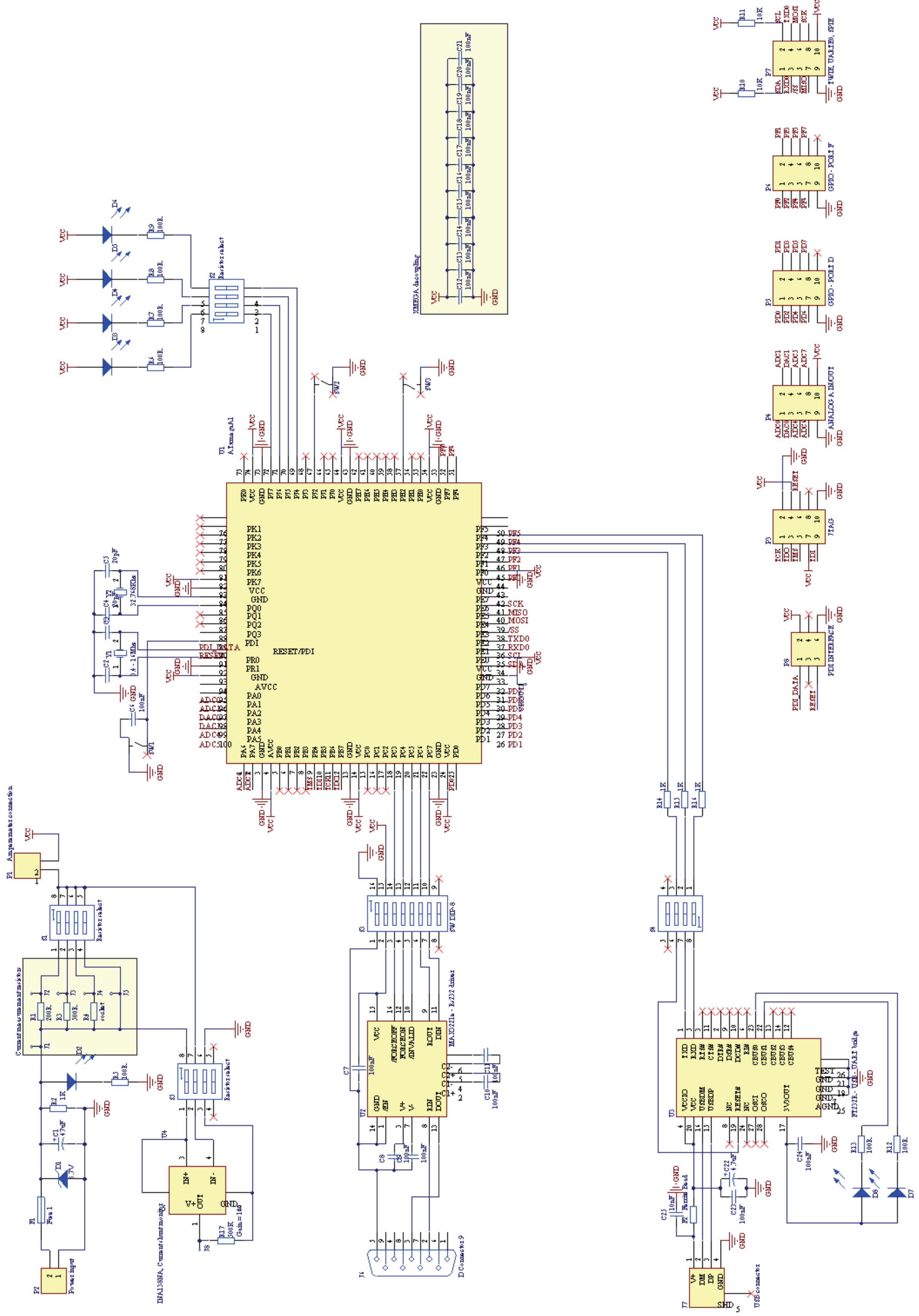


Figure 19 - Protype schematics

5.2 Layout

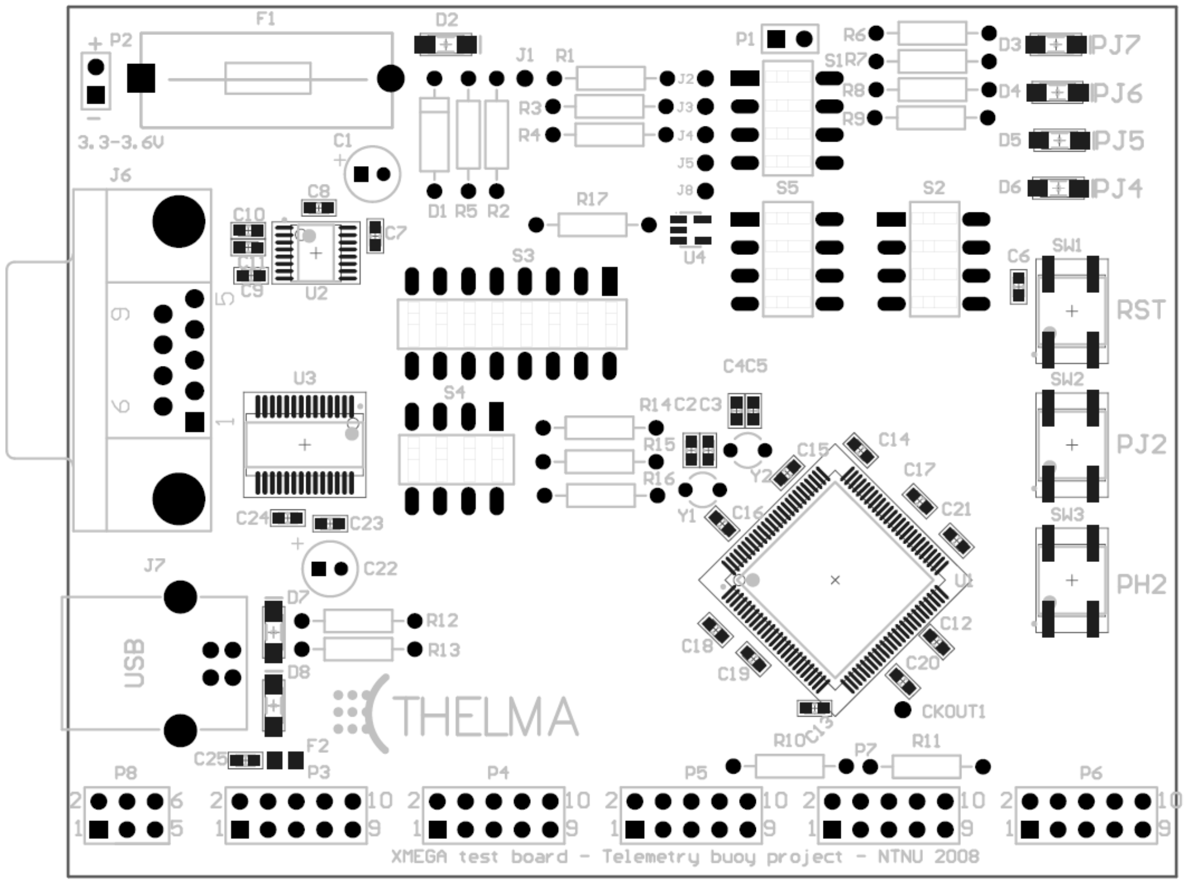


Figure 20 - Prototype component placement

5.3 Program code

The program is structured as in Figure 21. To let the CPU sleep for as long as possible the DMA controller is used to transfer the ADC results to internal SRAM. The DMA is programmed to transfer a fixed number of results. An interrupt is generated when this number is reached, waking the microcontroller. To minimize CPU intervention the ADC is used in free running mode with a downscaled clock. In this mode the ADC will have a sample frequency equal to the ADC frequency of 250 kHz.

To minimize the power consumption the processor runs on the 32MHz internal oscillator. When all N samples have been stored, the system clock prescaler will be set to one to allow maximum CPU frequency when running the digital band-pass filter. The float diagram is shown in Figure 21.

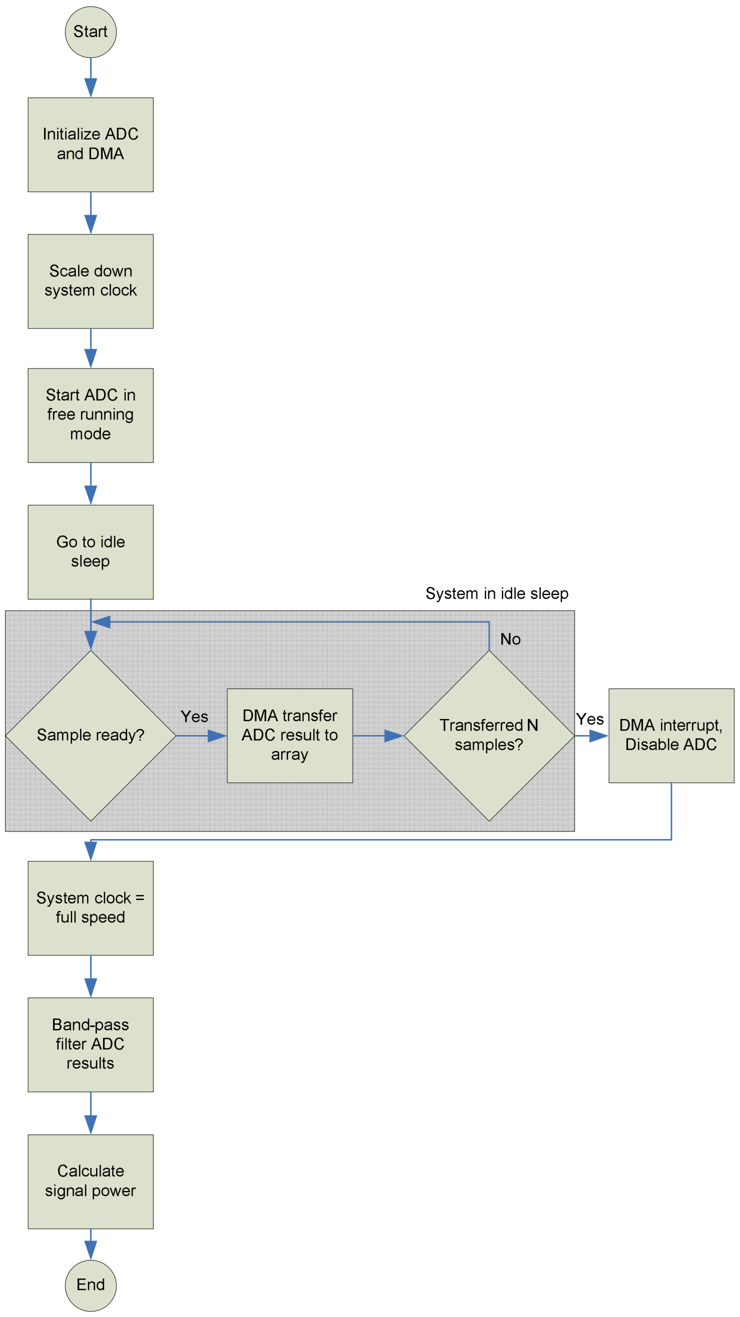


Figure 21 - Program float diagram

5.3.1 Band-pass filter implementation

As described in section 4.2 it is important to avoid using floating numbers and minimize the use of representations larger than 8-bit. It is therefore desirable to store all filter coefficients in an 8-bit fashion. The filter coefficients to be implemented are calculated in section *4.1* where the smallest value is 0.0062 and the largest is 0.9875. When choosing the number representation it is important that it is possible to represent the largest value without sacrificing too much accuracy to represent the smallest number. Using Q1.7 representation, as described in section 4.2.1 on page 31, will present a resolution of 0.0078125 and a maximum of 0.9921875. For these filter coefficients this will provide sufficient accuracy. It is important to notice that this problem must be addressed again if the filter coefficients are changed. Using for instance Q3.13 format will provide high accuracy and provide a more general implementation, but the cost is a much more inefficient program.

The coefficients are stored as shown below

```
#define FloatToQ7(a) (a*128 + 0.5)
signed char b0 = FloatToQ7(0.0062);
signed char b1 = FloatToQ7(0.0);
signed char b2 = FloatToQ7(-0.0062);
signed char a1 = FloatToQ7(0.3233);
signed char a2 = FloatToQ7(0.9875);
```

This will of course introduce some quantization error, but using MATLAB to simulate the filter with the new quantizised coefficients gives a result almost equal to the original. The results are shown in Figure 22 and Figure 23.

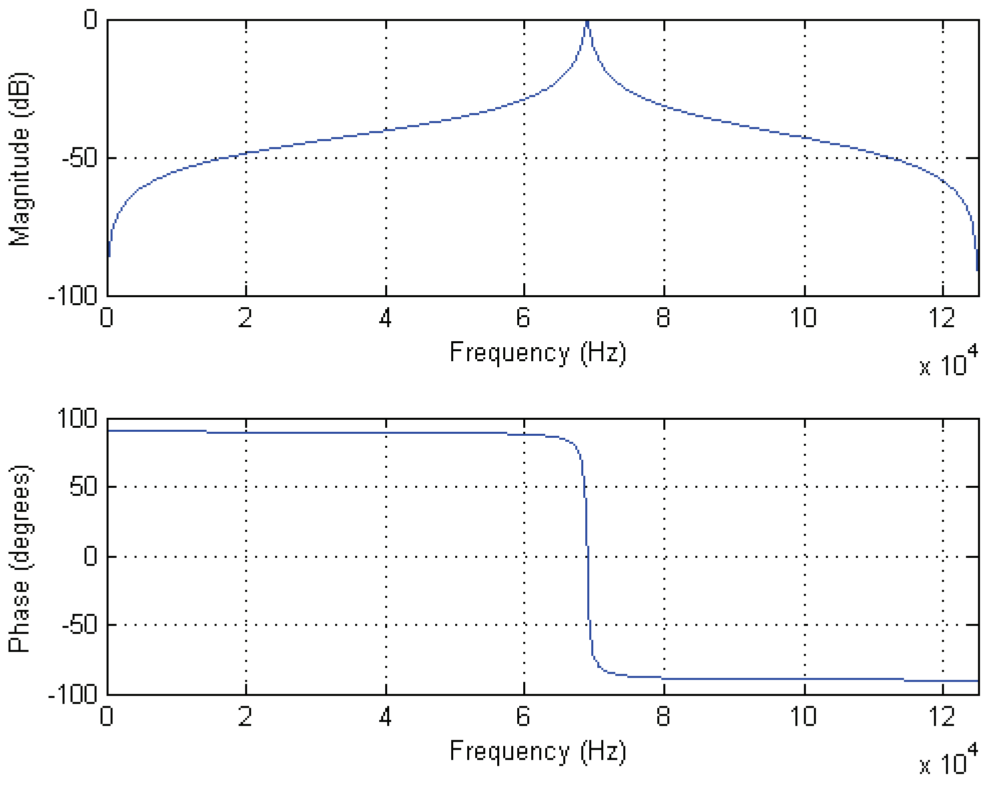


Figure 22 - Filter frequency response with quantizised Q1.7 coefficients

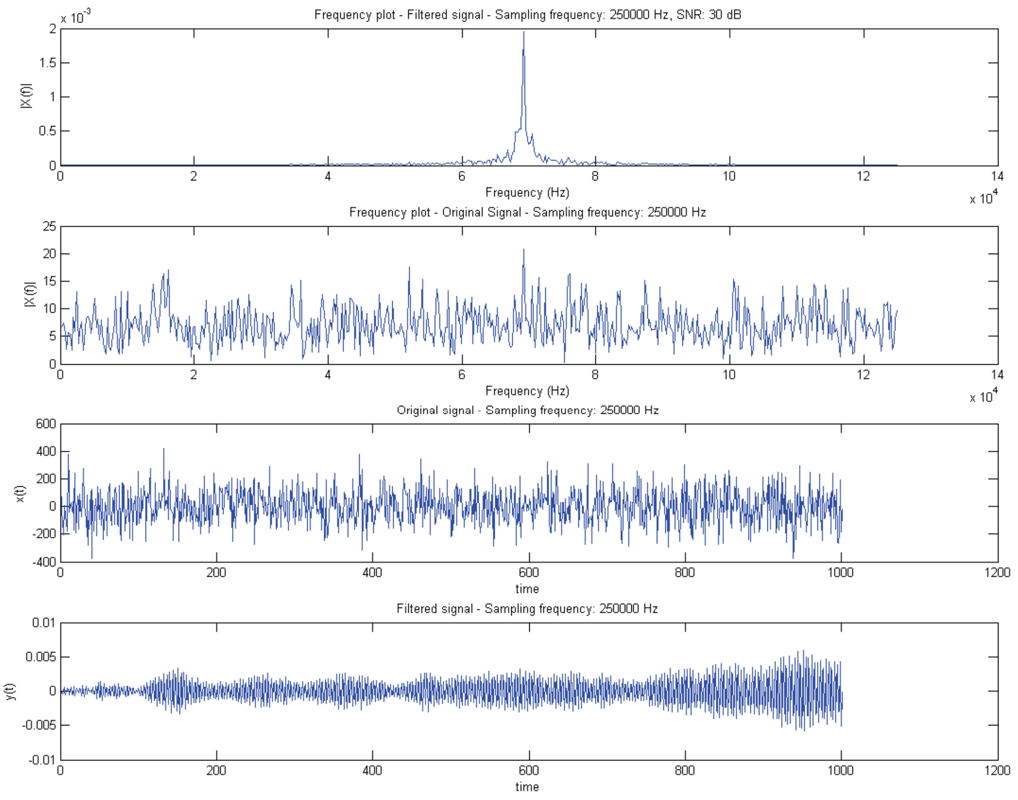


Figure 23 - Simulation with actual quantizised Q1.7 coefficients

When using scaled numbers such as the Q1.7 and the Q3.13 format it is important to scale down the result when using multiplications. For instance multiplying a Q1.7 number with a Q2.6 number will result in a Q3.13 format. The most effective way to do this scaling in an AVR processor is to use right shifts. To avoid some of these shifts and castings the result array is in a Q3.13 format.

The band pass algorithm implementation is shown below.

```
/*********************************************************
Name: filter_array_optimized
Type: void
Function: 2. order bandpass filters input signal arrav
requires coeffisients a1, a2, b0, b1, b2
Input: array of 8-bit signed numbers (x)
Output: Array of 16-bit signed numbers (y)
**********************************************************/
void filter_array_optimized(signed char * x, signed int * y,unsigned int size)
{
        unsigned int n;
        //initialize y[0] -> y[1]
        y[0] = ((int)b0*x[0]);
        y[1] = ((((long int)-a1*y[0])>>7) + (int)b0*x[1] + (int)b1*x[0]);
        //run filter
        for (n=2; n < size; n++)
        {
                y[n] = ((((long int)-a1*y[n-1])>>7) - (((long int)a2*y[n-2])>>7)+ b0*x[n] + b1*x[
                n-1] + b2*x[n-2]);
        }
}
```

The complete c-code is given in appendix 2 and 5.

6. Test results

This chapter presents test measurements for the prototype. The test-code is provided in appendix 2 and 5. Only the results and accuracy are presented here. For a discussion, please refer to chapter 7.

6.1 Accuracy of the results

The test is carried out using a 1Ω series resistor and a current shunt resistor amplifier with an amplification of 100. A TENMA 72-7235 150MHz oscilloscope is used for all measurements. The current is passed through a 100mA fuse introducing a series resistance of 21.2Ω. The voltage will therefore vary with the current consumption. It is therefore stated for every measurement.

The tolerance of the measurements is calculated as follows:
Series resistor: 1Ω, 1%
Resistor in current shunt amplifier circuit: 499kΩ, 1%
Current shunt amplifier total output error: 2%.
Oscilloscope vertical accuracy: 3%

The worst case scenario error is calculated in Table 9.

Table 9 - Current measurement error calculation

Device	Error	Comment
Series resistor	-1%	
Current shunt resistor	492.01/500 - 1 = -1.6%	100 gain is for 500k resistor
Current shunt amplifier output error	-2%	
Oscilloscope measurement error:	-3%	
SUM	**-7.6%**	

If we assume that noise and thermal effects are small and can be neglected, current measurements have a tolerance of 7.6%. For voltage measurements the tolerance is only dependent on the oscilloscope and is 3%. Time measurements are done by toggling a port pin. This will introduce a slight delay, but can be neglected due to the relatively long execution time of the routines/algorithms. The oscilloscope will introduce a tolerance of 0.01% for time measurements. A summary of the tolerances is given in Table 10.

Table 10 - Measurement tolerances

Measurement	Tolerance
Current	7.6%
Voltage	3%
Time	0.01%

6.2 Measurement results

The measurements are carried out in a routine by routine fashion. The results are shown in Table 11 and Table 12. Some of the routines have different number of samples. This is due to technical reasons with the oscilloscope setup when measuring average voltage.

Table 11 - Prototype measurements

Routine	Voltage [V]	Avarage current [A]	Time [s]	Samples
Sampling, ADC with DMA	2,89	0,00472	0,0004899	85
Remove DC offset	2,73	0,0198	0,0000512	85
Filter	2,77	0,017	0,001	110
Power-calcualtion	2,76	0,0194	0,0001101	110
System clock sample mode	2,92	0,0000928	0,00035	
System clock running mode	2,99	0,00282	0,00001262	

Table 12 - Prototype energy calculations

Routine	Samples	Time pr sample [s]	Energy pr Burst [J]	Energy pr sample [J]
ADC with DMA	85	5,76353E-06	6,68263E-06	7,86192E-08
Remove DC	85	6,02353E-07	2,76756E-06	3,25596E-08
Filter	110	9,09091E-06	0,00004709	4,28091E-07
Power-calcualtion	110	1,00091E-06	5,89519E-06	5,35927E-08
System clock sample mode			9,48416E-08	
System clock running mode			1,06409E-07	

The time for the routines include enabling and disabling the used module.

7. Evaluation

This chapter discusses the measurement values in chapter 6.2. The evaluation is based on energy calculations. Two principles of detecting and receiving information will be covered. It is important to notice that the energy consumption of the microcontroller will vary with supply voltage. There will therefore be some deviation in the energy calculations, but the results will provide a good estimation.

An example from the datasheet is shown in Figure 24. If a voltage regulator is used to lower the voltage, the microcontroller energy consumption will decrease, but the energy loss of the regulator must be added. Also note that when operating at voltages below 2.7V the XMEGA cannot run at full speed, but requires the system clock to be scaled as in Figure 1 on page 6.

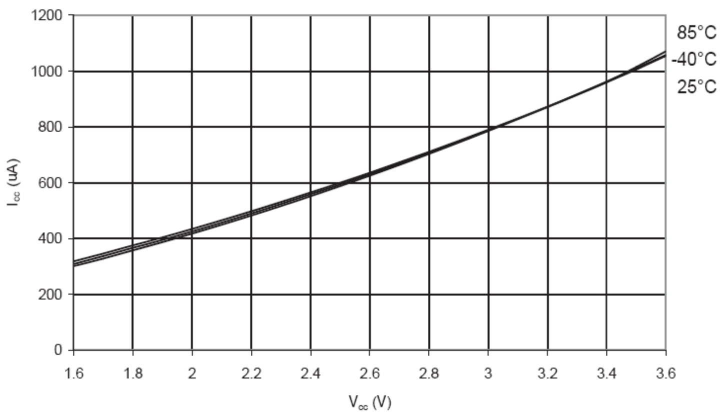


Figure 24 - Active supply current vs. Vcc (f = 1.0Mhz)

7.1 Energy calculations

The battery specifications are covered in section 2.1.3 on page 6. The total amount of energy of the battery is

$$\begin{aligned} E &= 13Ah \cdot 3.5V \\ E &= 13 \cdot 3.5 \cdot 3600Ws = 163800J \end{aligned} \tag{7.1}$$

This is the maximum energy consumption of the complete system, including analogue and digital electronics. This project only considers the digital signal processing part of the system. Assuming that the sampling and filtering uses 30% of the total energy this leaves the DSP functionality with

$$E_{DSP} = 163800 \cdot 0,30 \approx 50000J \tag{7.2}$$

By using the results from chapter 6.2 we can calculate total energy consumption of a complete burst. The example below is an example scenario using 100 samples per burst. The *ADC with DMA, Remove DC, filter,* and *power-calculation* routines have a running time growth function $T(n) = \Theta(n)$. The *time per sample* in Table 12 is therefore used to calculate the execution time for the mentioned routines.

Table 13 - Calculation of burst energy

	Energy pr burst [J]	Time pr burst [s]	% of total Energy	% of total Time
ADC with DMA	7,86192E-06	0,04899	13,2	29,6
Remove DC	3,25596E-06	0,00512	5,5	3,1
Filter	4,28091E-05	0,1	72,0	60,4
Power-calcualtion	5,35927E-06	0,01101	9,0	6,7
System clock sample mode	9,48416E-08	0,00035	0,2	0,2
System clock running mode	1,06409E-07	0,00001262	0,2	0,0
SUM	**5,94875E-05**	**0,16548262**		

By using the result that one burst of samples requires 5,94875E-05 Joule of energy we can calculate the total number of 100-sample bursts k restrained by (7.2).

$$k = \frac{50000}{5.94875 \cdot 10^{-5}} = 840512949 \qquad (7.3)$$

This result will be used in the preceding section.

Note that the tasks performed by the CPU consume most of the energy. The *Filter* Routine, *Remove DC* routine and *power-calculation* routine contributes to 86,5% of the total energy consumption. The CPU power consumption should therefore be the main concern if a different microcontroller should be considered.

7.2 Detecting and decoding a signal

Two principles for detecting a signal will be presented. The first method relies only on digital processing to determine if a signal is present. The second method relies on an external signal to initiate a reception. Both methods are based on detecting the example VEMCO signal given in section 2.2.1 on page 11.

Sampling every 5 ms

By sampling every 5ms the MCU can determine if a signal pulse is present by comparing the signal power of two and two bursts. Since the pulse-length is 10ms and the timeslots are 20ms, sampling every 5ms is sufficient for detecting a pulse flank and synchronising the clock. The calculation is given in Table 14.

Table 14 - Sampling every 5ms

Bursts pr battery	840 512 949
bursts pr second	200
seconds	4 202 565
days	**49**

With the previous assumptions this method will only provide operation for 49 days.

Sampling only when a signal is present

This method relies on external circuitry to trigger the device when a transmission is detected. This circuitry is not the scope of this project, but could be a flank detector constructed with operational amplifiers or a phase locked loop. This circuit would of course introduce added power consumption, but it is assumed that this is not a problem.

The synchronisation is based on the external circuitry and a higher density burst rate to detect the second synchronisation pulse. The detection is based on the fact that we need to determine if a flank is present in the given timeslot. We need two signal power calculations to determine a flank. There are 16 timeslots per nibble. In a worst case scenario we get a pulse in the 16th time slot for every nibble, thus $16 \cdot 2 \cdot 2 = 64$ bursts and signal power calculations is adequate to decode the signal. We assume one byte of data and one byte CRC as in Figure 5 on page 12. We get the results given in Table 15.

Table 15 - Sampling only when receiving signals

Bursts pr battery	840 512 949
Bursts pr byte	64
Bursts pr sync	20
Bursts pr CRC	64
Bytes pr transmission	1
Bursts pr transmission	148
Transmissions	**5 679 142**

We notice that this number is fairly high. Using the demand of a minimum of 100 000 correct readings per battery/year we achieve an allowable failure rate of

$$f = \frac{5679142}{100000} = 56.8 \tag{7.4}$$

Where f is the number of failed detections per non-failed detection. This will allow the external circuitry to be far from perfect, supporting the assumption that it is possible to construct such a circuitry with low power consumption.

This result also shows that, for the receiver, it should be possible to implement a more complex and robust modulation scheme such as spread spectrum or frequency shift keying as described in section 2.2 *Underwater acoustic communication and signal modulation* on page 11.

7.3 Comparison with AVR32 UC3B

It is a difficult task to choose the correct microcontroller at the start of a project. This task becomes especially difficult when the device is not fully characterized as with the ATxmegaA1. This chapter will therefore compare the measurement results for the ATxmegaA1 with calculations based upon characterization data for the AVR32 UC3B device [14] and the AVR32 DSP library application note [24].

The AVR32 UC3B is described in section 3.3.2 on page 22. Its 32-bit processor runs at up to 60 Mhz. This will provide a higher throughput than the ATxmegaA1, but the cost will be a higher power-consumption when the processor is running. It is though possible that the total current consumption will be lower due to faster computation.

As described in section 3.3.1 on page 21 the UC3B will draw a current of 23.5 mA @ 60 Mhz and 3.3V. Compared to the test results of the ATxmegaA1 this is only 6.5 mA higher. To compare the two microcontrollers the following scenario is assumed:

- 72-point ADC burst
- Energy calculations of 3.3V with the UC3B is comparable with the measurement results for the ATxmegaA1.
- Power loss due to an external voltage regulator can be neglected.

For the ATxmegaA1, based on the results from section 6.2, we get the following result.

Table 16 - ATxmegaA1 energy consumption for 72 samples

	Energy pr burst [J]	**Time pr burst [s]**	**% of total Energy**	**% of total Time**
ADC with DMA	5,66058E-06	0,0352728	13,2	29,6
Remove DC	2,34429E-06	0,0036864	5,5	3,1
Filter	3,08225E-05	0,072	71,9	60,4
Power-calcualtion	3,85867E-06	0,0079272	9,0	6,6
System clock sample mode	9,48416E-08	0,00035	0,2	0,3
System clock running mode	1,06409E-07	0,00001262	0,2	0,0
SUM	**4,28873E-05**	**0,11924902**	**100**	**100**

Based on the datasheet characterization [23] and the DSP application note benchmark results, we get the following result for the UC3B.

The filter has the specifications

- 2. order
- FIR type
- 16-bit Q1.15 number representation
- Execution time: 47.9us

Executing the filter will require the energy

$$E_f = 47.9 \cdot 10^{-6} s \cdot 23.5 \cdot 10^{-3} A \cdot 3.3V = 3.7146 \cdot 10^{-6} J \tag{7.5}$$

The energy ratio is thereby

$$r = \frac{3,08225 \cdot 10^{-5}}{3.7146 \cdot 10^{-6}} = 8.3 \tag{7.6}$$

Assuming the same relationship for the other routines in active mode these routines consume

$$E_r = \frac{6.40421 \cdot 10^{-6}}{8.3} = 7.71592 \cdot 10^{-7} J \tag{7.7}$$

The internal ADC of the UC3B does not have any free running mode in sleep modes. The microcontroller will therefore have to be in active mode when sampling continuously. To achieve a sample frequency of 250kHz the ADC clock need to be $250 \cdot 13kHz = 3.25Mhz$. Assuming that the CPU works at 12MHz and there is no need for a DMA due to the active mode, the energy consumption for the sampling and store becomes

$$\begin{aligned} t_s &= \frac{1}{250 \cdot 10^3} \cdot 72 = 2.88 \cdot 10^{-4} s \\ I &= 5.5 \cdot 10^{-3} A \\ E_s &= V \cdot I \cdot t_s = 3.3 \cdot 5.5 \cdot 10^{-3} \cdot 2.88 \cdot 10^{-4} = 5.2272 \cdot 10^{-6} J \end{aligned} \tag{7.8}$$

We assume that the overhead of starting a conversion can be neglected. The number might be reduced if the CPU can spend some of the time in a sleep mode using a timer or the ADC complete flag to wake the device.

The total amount of energy for one burst adds up to be

$$E_T = E_f + E_r + E_s = 9,71339 \cdot 10^{-6} J \tag{7.9}$$

This shows that the UC3B will in theory provide a more energy economic solution for sampling and filtering. Unfortunately the energy consumption for sleep modes cannot be compared for the ATxmegaA1 and the AT32UC3B due to absence of power characterization for the different sleep modes of the AT32UC3B. But assuming that these are similar for the two devices the AT32UC3B will provide a solution that is a factor of f_p times more energy economic.

$$f_p = \frac{4.2873 \cdot 10^{-5}}{9.71339 \cdot 10^{-6}} = 4.42 \tag{7.10}$$

By using (7.9) the same calculations as in section 7.2 is done for the AT32UC3B.

Table 17 - Sampling every 5ms

Energy pr burst	9,71339E-06
Joule from battery	50 000
Bursts pr battery	5 147 533 456
bursts pr second	200
seconds	25 737 667
days	**298**

This result shows that is may be possible to achieve a fully digital solution with sampling every 5ms to detect a signal, given that 72 samples is enough. To meet the requirement of one year operation we can either tune the number of samples per burst, tune the MCU frequency, introduce concurrency by filtering the signal while the ADC is converting, or allow the sampling and filtering function 38% of the battery capacity instead of 30%. It may also be possible to utilize sleep modes while converting.

Table 18 - Sampling only when receiving signals

Joule from battery	50 000
Bursts pr battery	5 147 533 456
Bursts pr byte	64
Bursts pr sync	20
Bursts pr CRC	64
Bytes pr transmission	1
Bursts pr transmission	148
Transmissions	**34 780 631**

As with the ATxmegaA1 there should be no problem to construct a system with an external trigger to initiate a reception. The AT32UC3B will obviously require less energy for the same amount of receptions.

It is important to notice that the AT32UC3B requires a buck-boost regulator to operate on battery power. This regulator must have good power efficiency in order to meet the requirements.

8. Conclusion

The goal of this project was to determine if it is possible to design a low-power acoustic telemetry buoy with digital signal processing that meets the requirements given in section 1.1 on page 1. Alternative modulation schemes have been studied to prepare the design for new fish-tags using more modern form of communication.

The test results in chapter 6 are based on the implementation of sampling, filtering and signal-power estimation on the ATxmegaA1. An example is given where 100 samples pr burst is used, the DSP part of the system is assigned 30% of the total amount of energy of the battery and some form of external reception-start trigger is assumed available. With these assumptions it is found that the system can handle over 5 million receptions. This number is way above the requirement of 100 thousand receptions.

By calculating the power consumption for the AT32UC3B it is found that this controller will consume about 4.4 times less energy than the ATxmegaA1 prototype results. This is based on typical power consumption numbers in the datasheet and the benchmark test results of the DSP-lib for AVR32. Some deviation will occur, but it is likely that it is possible to design a system both with and without an external trigger to initiate a reception using the AT32UC3.

It has also been proven that it is not possible to achieve one year operation without an external reception trigger using the ATxmegaA1.

The final conclusion is therefore that the AT32UC3B microcontroller should be used for further design. With this controller it should be possible to design an acoustic telemetry buoy that meets the requirements given in section 1.1. If a good external trigger for reception can be constructed this will provide the lowest energy consumption. The system should though incorporate digital filtration to allow compatibility with various frequencies and modulation schemes.

9. Appendix list

1. Efficiency-test of AVR multiplication and division
2. C-code for AVR XMEGA DSP implementation
    a. main.c
    b. dsp.c
    c. dsp.h
3. MATLAB code for filter analysis.
4. Prototype documentation
    a. Schematic
    b. Assembly drawing
    c. Copper planes
        i. Top
        ii. Bottom
    d. Bill of materials
5. CD-ROM
    a. Complete C-code for XMEGA implementation
    b. Matlab code
    c. Report
    d. Gerber files for prototype
    e. Schematic for prototype

10. Bibliography

[1] Vojin Oklobdzija, edited by Christian Piguet, *Low-Power Electronics Design* Chapter VII: Battery Cells, Sources of Energy, and Chip Cooling.

[2] Vojin Oklobdzija, edited by Christian Piguet, *Low-Power Electronics Design* Chapter 32: Low-power Software Techniques.

[3] Atmel datasheet*: XMEGA Manual,* 8077B-AVR-06/08

[4] Atmel XMEGA web site, http://www.atmel.com/products/AVR/default_xmega.asp

[5] Texas Instruments application report: *Choosing An Ultralow-Power MCU,* SLAA207 – June 2004

[6] Atmel application note: *AVR32739: AVR32 UC3 Low power software design,* Rev 32093B-AVR32-05/08

[7] William Stallings, *Data and Computer Communications, seventh Edition,* 2004

[8] Daniel B. Kilfoyle and Arthur B. Baggeroer, IEEE article, *The state of the Art in Underwater Acoustic telemetry, January 2000*

[9] Jan Eyolf Bjørnsen, Master Thesis: *Utvikling av intelligent hydrofonbøye for PINPOINT II, 10th june 2002*

[10] Tadiran datasheet for MODEL TL-5930, Rev. B 01/06

[11] Atmel application note: *AVR120 Characterization and Calibration of the ADC on an AVR,* Rev. 2559D-AVR-02/06

[12] Atmel AVR32 website: *www.atmel.com/AVR32*

[13] Atmel datasheet *AT32UC3A Series*, Preliminary, Rev. 32058F-AVR32-08/08

[14] Atmel datasheet *AT32UC3B Series*, Preliminary, Rev. 32059G-AVR32-04/08

[15] John G. Proakis and Dimitris G. Manolakis, *Digital Signal Processing – Principles. Algorithms and Applications, Fourth Edition*

[16] Fredric J. Harris, *Multirate signal processing for communication systems,* May 2004

[17] Hiroshi Harada and Ramjee Prasad, *Simulation and software radio for mobile communications*

[19] Atmel datasheet: *XMEGA A1 preliminary*, Rev 8067G-AVR-11/08

[20] Texas Instruments datasheet: *MSP430F543xA datasheet,* SLAS612-September 2008

[21] Texas Instruments user guide: *MSP430x5xx Family User's Guide*

[22] Obestar Consulting whitepaper by Eric L. Oberstar: *Fixed-Point Representation & Fractional Math*, Rev. 1.2, August 30, 2007

[23] Texas instruments overview paper: *MSP430 Overview and Key Application,* TI developer conference 21-30 May 2008

[24] Atmel AVR32 application note: *AVR32718: AT32UC3 Series Software Framework DSPLib*, Rev 32076A-AVR32-11/07

[25] Erick L. Oberstar and Michael J. Bauch, *Narrow Band Filter implementation On A low Cost Microcontroller, Issues and Performance,* 2001

Appendices

1. Efficiency-test of AVR multiplication and division

Efficiency-test of AVR multiplication and division

By Stian O. Moen

This document presents the number of cycles required to perform a multiplication/division of two numbers. All functions takes in two numbers of the type specified by the function name and returns either the product or the quotient. Finally two example-functions are given. The function simply returns a scaled version of the input. The actual routines are given below. Note that the result will be the same for both functions, but the required calculation time is quite different. The user of any microcontroller should be aware of the large added complexity of the program and computation time when operating with floating numbers and variables larger then the processor register-size.

```
//MULTIPLICATION
unsigned char result = char_multiply(13,16);      //40 cycles
int result2 = int_multiply(13,16);                //66 cycles
float restult3 = float_multiply(13,16);           //1754 cycles
double restult4 = double_multiply(13,16);         //1754 cycles

//DIVISION
unsigned char result5 = char_divide(130,16);      //117 cycles
int result6 = int_divide(130,16);                 //288 cycles
float restult7 = float_divide(130,16);            //1271 cycles
double restult8 = double_divide(130,16);          //1271 cycles

//EXAMPLES
unsigned char position = percent_slow(122);       //3227 cycles
unsigned char position2 = percent_fast(122);      //283 cycles (11,4 times faster)


//EXAMPLE ROUTINES:
unsigned char percent_slow(unsigned char joystick){
return joystick*0.78;
}
unsigned char percent_fast(unsigned char joystick){
return (joystick*78)/100;
}
```

29.09.2008

2. C-code for AVR XMEGA implementation

```c
#include "dma_driver.h"
#include "avr_compiler.h"
#include "adc_driver.h"
#include "clksys_driver.h"
#include "avr/sleep.h"
#include "dsp.h"
#include "usart_driver.h"
#include "stdio.h"


#define USART USARTF0


//GLOAL VARIABLES:

// Global declared status for interrupt routine
volatile bool gInterruptDone;
volatile bool gStatus;

//array for storing samples
volatile signed char ADC_samples[NUMBER_OF_SAMPLES] = {0};
volatile long unsigned int signal_power = 0;
volatile long unsigned int noise_power = 0;
volatile float ratio = 0;
//PROTOTYPES:

bool BlockMemCopy( const void * src, void * dest, uint16_t blockSize,
                   volatile DMA_CH_t * dmaChannel );
void init_ADC( void );
void init_DMA( void );
void init_USART( void );
void init_ports( void );
void enable_all_pullups(volatile PORT_t * port);
void disable_unused_modules(void);
void go_to_sleep(SLEEP_SMODE_t sleep_mode);
void system_clock_sample_mode( void );
void system_clock_running_mode( void );
void USART_putc(char data);
void disable_all_modules();
void wait_for_button();

int main( void )
 {

    volatile unsigned char filtered_array[NUMBER_OF_SAMPLES] = {0};
    volatile signed int test[NUMBER_OF_SAMPLES] = {0};

    //Initialize
    init_DMA();
    init_ADC();
    init_ports();
    disable_unused_modules();
    system_clock_running_mode();


/*** CURRENT MEASUREMENT ***/

    //change clock to sample mode
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    system_clock_sample_mode();
    PORTD.OUTCLR = PIN7_bm;

    //Do sampling in sleep mode
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    ADC_Enable(&ADCA);
    ADC_Wait_8MHz(&ADCA);
    ADC_FreeRunning_Enable(&ADCA);
    go_to_sleep(SLEEP_SMODE_IDLE_gc);
    disable_all_modules();
    PORTD.OUTCLR = PIN7_bm;


    //change system clock frequency
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    system_clock_running_mode();
    PORTD.OUTCLR = PIN7_bm;


    //remove DC offset
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    unsigned int n;
    for(n=0; n < NUMBER_OF_SAMPLES; n++)
    {
    ADC_samples[n] -= DC_OFFSET;
    }
    PORTD.OUTCLR = PIN7_bm;


    //Filter samples
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    filter_array_optimized((signed char*) ADC_samples, (int*)test,sizeof(ADC_samples));
    PORTD.OUTCLR = PIN7_bm;


    //Calculate signal power
    wait_for_button();
    PORTD.OUTSET = PIN7_bm;
    signal_power = calculate_signal_power(test);
    PORTD.OUTCLR = PIN7_bm;

    while(true);
}


//INTERRUPT HANDLERS:

/*! DMA CH0 Interrupt service routine. Clear interrupt flags after check. */
//intterupt occurs when all adc samples has been stored in array
ISR(DMA_CH0_vect)
{
    //bust samling done, disable ADC
    ADC_FreeRunning_Disable(&ADCA);
    ADC_Disable(&ADCA);


    //check for errors
    if (DMA.CH0.CTRLB & DMA_CH_ERRIF_bm) {
        DMA.CH0.CTRLB |= DMA_CH_ERRIF_bm;
        gStatus = false;
    } else {
        DMA.CH0.CTRLB |= DMA_CH_TRNIF_bm;
        gStatus = true;
    }
    gInterruptDone = true;
}


//FUNCTIONS:

/*********************************************************
Name:       BlockMemCopy
Type:       bool
Function:   Sets upt the DMA to tranfer data from the ADC
            one byte at a time. icreasing the target address
            after each byte
Input:      pointer to source data (src),
            pointer to destination array (dest)
            size of each block (blockSize)
            which DMA channel to use (dmaChannel)
Output:     always true
**********************************************************/
bool BlockMemCopy( const void * src,
                   void * dest,
                   uint16_t blockSize,
                   volatile DMA_CH_t * dmaChannel )
{
    DMA_EnableChannel( dmaChannel );
    DMA_SetupBlock( dmaChannel,
                    src,
                    DMA_CH_SRCRELOAD_BURST_gc,
                    DMA_CH_SRCDIR_INC_gc,
                    dest,
                    DMA_CH_DESTRELOAD_NONE_gc,
                    DMA_CH_DESTDIR_INC_gc,
                    blockSize,
                    DMA_CH_BURSTLEN_1BYTE_gc,
                    1,
                    false );

    DMA_EnableSingleShot(dmaChannel); //tranfer burst istead of block when triggered
    DMA_SetTriggerSource(dmaChannel, DMA_CH_TRIGSRC_ADCA_CH0_gc);
    //enable interrupt when complete:
    //clear interrupt flags
    DMA_SetIntLevel(dmaChannel, DMA_CH_TRNINTLVL_MED_gc, DMA_CH_ERRINTLVL_OFF_gc);

    return true;
}



/**********************************************************
Name:       init_ADC
Type:       void
Function:   Initializes the ADC to convert at PORTA pin0,
            free running mode, sweep only CH0,
            prescaler = DIV8, 8-bit resolution
Input:      none
Output:     none
***********************************************************/
void init_ADC( void )
{
    //uint8_t offset;

    /* Move stored calibration values to ADC A. */
    ADC_CalibrationValues_Set(&ADCA);

    /* Get offset value for ADC A. */
    //offset = ADC_Offset_Get(&ADCA);

    /* Set up ADC A to have unsigned conversion mode and 8 bit resolution. */
    ADC_ConvMode_and_Resolution_Config(&ADCA, false, ADC_RESOLUTION_8BIT_gc);

    /* Sample rate is CPUFREQ/8. Allow time for storing data. */
    ADC_Prescaler_Config(&ADCA, ADC_PRESCALER_DIV8_gc);
    /* Set referance voltage on ADC A to be VCC-0.6 V.*/
    ADC_Referance_Config(&ADCA, ADC_REFSEL_VCC_gc);

    /* Setup channel 0, 1, 2 and 3 to have single ended input. */
    ADC_Ch_InputMode_and_Gain_Config(&ADCA.CH0,
                                     ADC_CH_INPUTMODE_SINGLEENDED_gc,
                                     ADC_CH_GAIN_1X_gc);



    /* Set input to the channels in ADC A to be PIN 4, 5, 6 and 7. */
    ADC_Ch_InputMux_Config(&ADCA.CH0, ADC_CH_MUXPOS_PIN0_gc, ADC_CH_MUXNEG_PIN0_gc);


    /* Setup sweep of all four virtual channels. */
    ADC_SweepChannels_Config(&ADCA, ADC_SWEEP_0_gc);

    /* Enable low level interrupts on ADCA channel 0, on conversion complete. */
    ADC_Ch_Interrupts_Config(&ADCA.CH0, ADC_CH_INTMODE_COMPLETE_gc, ADC_CH_INTLVL_OFF_gc
);
}

/**********************************************************
Name:       disable_unused_modules
Type:       void
Function:   Disables all modules except the ADC and the DMA
Inout:      none
```

```
Output:      none
************************************************************/
void disable_unused_modules(void)
{
    //disable all peripherals except ADFC and DMA
    PR_PR   = (PR_EBI_bm |  PR_RTC_bm | PR_EVSYS_bm /*| PR_DMA_bm*/);
    PR.PRPA = (PR_DAC_bm /*| PR_ADC_bm*/ | PR_AC_bm);
    PR.PRPB = (PR_DAC_bm | PR_ADC_bm | PR_AC_bm);
    PR.PRPC = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPD = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPE = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPF = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
}


/***********************************************************
Name:        go_to_sleep
Type:        void
Function:    Puts the MCU to sleep
Input:       sleepmode defined in device io header file
             (sleep_mode)
Output:      none
************************************************************/
void go_to_sleep(SLEEP_SMODE_t sleep_mode)
{
    SLEEP.CTRL = sleep_mode | SLEEP_SEN_bm;
    sleep_cpu();
    sleep_disable();
}

/***********************************************************
Name:        init_DMA
Type:        void
Function:    Initilizes DMA to trancer data from ADCA.CH0
             to array ADC_samples. Enable interrupt when
             NUMBER_OF_SAMPLES bytes have been transferred
Input:       none
Output:      none
************************************************************/
void init_DMA( void )
{
    volatile DMA_CH_t * Channel;
    Channel = &DMA.CH0;

    DMA_Enable();
    gStatus = BlockMemCopy((void*)&ADCA.CH0.RES, (void*) ADC_samples, NUMBER_OF_SAMPLES,
 Channel);

    //enable all level interrupts:
    PMIC.CTRL |= PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm | PMIC_HILVLEN_bm;
    sei();
}

/***********************************************************
Name:        system_clock_sample_mode
Type:        void
Function:    Sets the system clock to 32MHz internal
             oscillator. Prescales the clock down to 2 MHz
Input:       none
Output:      none
************************************************************/
void system_clock_sample_mode( void )
{
    //change oscillator to 32MHz internal RC and choose prescaler.
    CLKSYS_Enable( OSC_RC32MEN_bm );
    CLKSYS_Prescalers_Config( CLK_PSADIV_16_gc, CLK_PSBCDIV_1_1_gc );
    do {} while ( CLKSYS_IsReady( OSC_RC32MRDY_bm ) == 0 );
    CLKSYS_Main_ClockSource_Select( CLK_SCLKSEL_RC32M_gc );
}

/***********************************************************
Name:        system_clock_running_mode
```

```
Type:        void
Function:    Sets the system clock to 32MHz internal
             oscillator. Prescaler = DIV1
Input:       none
Output:      none
************************************************************/
void system_clock_running_mode( void )
{
    CLKSYS_Prescalers_Config( CLK_PSADIV_1_gc, CLK_PSBCDIV_1_1_gc );
}


/**********************************************************
Name:        USART_putc
Type:        void
Function:    Function created to enable compatibility
             to fdevopen
Input:       none
Output:      none
************************************************************/
void USART_putc(char data)
{
    while(!USART_IsTXDataRegisterEmpty(&USART));
    USART_PutChar(&USART, data);
}

/**********************************************************
Name:        init_USART
Type:        void
Function:    Initializes USART for communication with the
             USART <--> USB converter. uses fdevopen() to
             link printf to USB
Input:       none
Output:      none
************************************************************/
void init_USART( void )
{
    /* This PORT setting is only valid to USARTC0 if other USARTs is used a
     * different PORT and/or pins is used. */
    /* PIN3 (TXD0) as output. */
    PORTF.DIRSET = PIN3_bm;

    /* PF2 (RXD0) as input. */
    PORTF.DIRCLR = PIN2_bm;

    /* USARTC0, 8 Data bits, No Parity, 1 Stop bit. */
    USART_Format_Set(&USART, USART_CHSIZE_8BIT_gc, USART_PMODE_DISABLED_gc, false);

    /* Set Baudrate to 9600 bps:
     * Use the default I/O clock fequency that is 2 MHz.
     * Do not use the baudrate scale factor
     *
     * Baudrate select = (1/(16*(((I/O clock frequency)/Baudrate)-1)
     *                 = 12
     */
    USART_Baudrate_Set(&USART, 12 , 0);

    /* Enable both RX and TX. */
    USART_Rx_Enable(&USART);
    USART_Tx_Enable(&USART);

    fdevopen(USART_putc,0);
}


/**********************************************************
Name:        init_ports
Type:        void
Function:    Initializes all port pins to input with pullup
             except the ADC input pin.
Input:       none
Output:      none
************************************************************/
void init_ports( void )
{
    //debug
```

```c
    PORTD.DIRSET = PIN7_bm;
    PORTJ.PIN2CTRL = PORT_OPC_PULLUP_gc;

    //enable pullups for all unused pins
    enable_all_pullups(&PORTA);
    PORTA.PIN0CTRL = 0;
    enable_all_pullups(&PORTB);
    enable_all_pullups(&PORTC);
    enable_all_pullups(&PORTD);
    enable_all_pullups(&PORTE);
    enable_all_pullups(&PORTF);
    enable_all_pullups(&PORTH);
    enable_all_pullups(&PORTJ);
    enable_all_pullups(&PORTK);
    enable_all_pullups(&PORTQ);

}


/***********************************************************
Name:       enable_all_pullups
Type:       void
Function:   Enable pullups on all port pins.
Input:      pointer to the port to set pullups
Output:     none
************************************************************/
void enable_all_pullups(volatile PORT_t * port)
{
    port->PIN0CTRL = PORT_OPC_PULLUP_gc;
    port->PIN1CTRL = PORT_OPC_PULLUP_gc;
    port->PIN2CTRL = PORT_OPC_PULLUP_gc;
    port->PIN3CTRL = PORT_OPC_PULLUP_gc;
    port->PIN4CTRL = PORT_OPC_PULLUP_gc;
    port->PIN5CTRL = PORT_OPC_PULLUP_gc;
    port->PIN6CTRL = PORT_OPC_PULLUP_gc;
    port->PIN7CTRL = PORT_OPC_PULLUP_gc;
}

/***********************************************************
Name:       disable_all_modules
Type:       void
Function:   Disables all modules except the CPU to save power
Input:      none
Output:     none
************************************************************/
void disable_all_modules()
{
    PR_PR   = (PR_EBI_bm |  PR_RTC_bm | PR_EVSYS_bm | PR_DMA_bm);
    PR.PRPA = (PR_DAC_bm | PR_ADC_bm | PR_AC_bm);
    PR.PRPB = (PR_DAC_bm | PR_ADC_bm | PR_AC_bm);
    PR.PRPC = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPD = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPE = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
    PR.PRPF = (PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm |
PR_TC1_bm | PR_TC0_bm);
}

/***********************************************************
Name:       wait_for_button
Type:       void
Function:   Waits for one button push on PJ2.l Includes
            debounce filter
Input:      none
Output:     none
************************************************************/
void wait_for_button()
{
    do
    {
        while((PORTJ.IN & PIN2_bm));//wait for user initiation
        _delay_ms(100);             //wait for bounce to run out
    }
    while(PORTJ.IN & PIN2_bm);
    while(!(PORTJ.IN & PIN2_bm)); //wait for release

}
```

```c
/*************************************************************
This file includes several different implementation of
the second order band-pass filter. The most effective and the
one that has provided the test restults is
filter_array_optimized(). Arrray lookups typically require
more time than reading single variables.
filter_semi_array_optimized() was implemented to check if
storing the values in single variables would give a better
result. This was not the case. It most certainly is the case
for zero optimization compiler settings, but with optimizations
the compiler produces faster code for the array type.
Author: Stian O. Moen
*************************************************************/
#include "dsp.h"


/*************************************************************
Convert coefficients to Q7 numbers and store.
*************************************************************/
signed char b0 = FloatToQ7(0.0062);
signed char b1 = FloatToQ7(0.0);
signed char b2 = FloatToQ7(-0.0062);
signed char a1 = FloatToQ7(0.3233);
signed char a2 = FloatToQ7(0.9875);


/************************************************************
Name:        filter_array_optimized
Type:        void
Function:    2. order bandpass filters input signal array
             requires Q7 coeffisients a1, a2, b0, b1, b2
Input:       array of 8-bit signed numbers (x)
Output:      Array of 16-bit signed numbers (y)
*************************************************************/
void filter_array_optimized(signed char * x, signed int * y,unsigned int size)
{
    unsigned int n;
    //initialize y[0] -> y[1]
    y[0] = ((int)b0*x[0]);
    y[1] = ((((long int)-a1*y[0])>>7) + (int)b0*x[1] + (int)b1*x[0]);

    //run filter
    for (n=2; n < size; n++)
    {
      y[n] = ((((long int)-a1*y[n-1])>>7) - (((long int)a2*y[n-2])>>7)+ b0*x[n] + b1*x[
n-1] + b2*x[n-2]);
    }
}

/************************************************************
Name:        filter_semi_array_optimized
Type:        void
Function:    2. order bandpass filters input signal array
             requires Q7 coeffisients a1, a2, b0, b1, b2
Input:       array of 8-bit signed numbers (x)
Output:      Array of 16-bit signed numbers (y)
*************************************************************/
void filter_semi_array_optimized(signed char * x, signed int * y, unsigned int size)
{
    volatile signed int y_0 = 0;
    volatile signed int y_1 = 0;
    volatile signed int y_2 = 0;
    volatile signed char x0 = 0;
    volatile signed char x1 = 0;
    volatile signed char x2 = 0;
    unsigned int i;

    for (i=0; i < size; i++)
    {
        x0 = x[i];
        y_0 = ((((long int)-a1*y_1)>>7) - (((long int)a2*y_2)>>7) + b0*x0 + b1*x1 + b2*
x2);
        y_2 = y_1;
        y_1 = y_0;
        x2 = x1;
        x1 = x0;
        y[i] = y_0;
    }
}


/**********************************************************
Name:       calculate_signal_power
Type:       unsigned int
Function:   returns the mean absolute value of the input
            signal array.
Input:      pointer to array of 8-bit signed numbers
            (array)
Output:     16 bit unsigned number
***********************************************************/
unsigned int calculate_signal_power(signed int * array)
{
    unsigned long int signal_power = 0;
    unsigned int n;
    for (n=0;  n < NUMBER_OF_SAMPLES; n++)
    {
        if (array[n] < 0)
            signal_power -= array[n];
            else
                signal_power += array[n];
    }
    return (unsigned int)(signal_power / NUMBER_OF_SAMPLES);
}


/**********************************************************
Name:       filter_array_optimized_Q13
Type:       void
Function:   2. order bandpass filters input signal array
            requires Q13 coeffisients a1, a2, b0, b1, b2
            This routine has higher accuracy then the Q7
            version and is therefore more general. The
            cost is a higher execution time.
Input:      array of 8-bit signed numbers (x)
Output:     Array of 16-bit signed numbers (y)
***********************************************************/
void filter_array_optimized_13(signed char * x, signed int * y,unsigned int size)
{
    signed int ADC_samplesQ14[NUMBER_OF_SAMPLES];

    unsigned int n;
    for(n=0; n < NUMBER_OF_SAMPLES; n++)
    {
    ADC_samples[n] -= DC_OFFSET; //subtract DC offset
    ADC_samplesQ14[n] = x[n] << 8; //change from Q6 to Q14
    }

    //initialize y[0] -> y[1]
    y[0] = ((long int)b0*ADC_samplesQ14[0]) >> 13;
    y[1] = ((long int)-a1*y[0] + (long int)b0*ADC_samplesQ14[1] + (long int)b1*
ADC_samplesQ14[0]) >> 13;

    //run filter
    for (n=2; n < NUMBER_OF_SAMPLES; n++)
    {
      y[n] = ((long int)-a1*y[n-1] - (long int)a2*y[n-2] + (long int)b0*ADC_samplesQ14[n
] + (long int)b1*ADC_samplesQ14[n-1] + (long int)b2*ADC_samplesQ14[n-2]) >> 13;
    }
}
```

```
#ifndef __DSP_H__
#define __DSP_H__ 1


#define FloatToQ7(a)    (a*128 + 0.5)
//the offset of the input signal
#define DC_OFFSET       0x80
//the number of samples in one sample burst
#define NUMBER_OF_SAMPLES 110

//filter Coefficients
extern signed char b0;
extern signed char b0;
extern signed char b1;
extern signed char b2;
extern signed char a1;
extern signed char a2;

//prototypes:

void filter_array_optimized(signed char * x, signed int * y,unsigned int size);
void filter_semi_array_optimized(signed char * x, signed int * y, unsigned int size);
unsigned int calculate_signal_power(signed int * array);
void filter_array_optimized_13(signed char * x, signed int * y,unsigned int size);

#endif
```

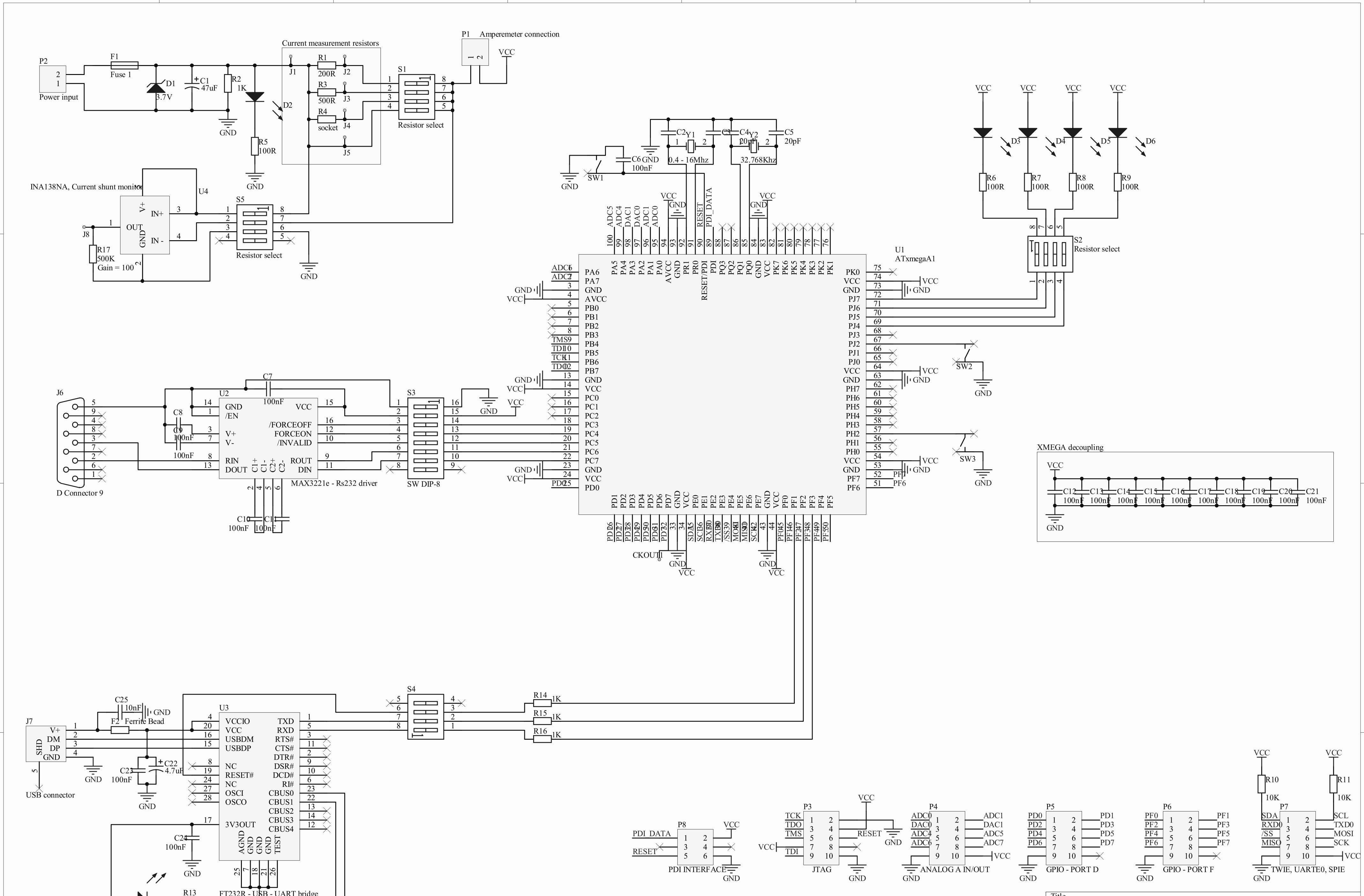
3. MATLAB code for filter analysis

```
close all
clear all
hold off
clc

% Digital filter
fd = 69000;     % Filter center frequency (Hz)
fs = 250000;    % Sampling frequency
T = 1/fs;       % Sampling interval
Bd = 500;       % Filter band width (Hz)

% Prewarping
fa = tan(pi*fd*T)/(pi*T);
famin = (tan(pi*(fd-(Bd/2))*T))/(pi*T);
famax = (tan(pi*(fd+(Bd/2))*T))/(pi*T);
Ba = famax - famin;
G = (Ba/(2*fa))*sqrt(pi^2*(Ba^2 + 16*fa^2));

% Filter Coefficients
%1/128, 41/128 and 126/128 are the values used by the microcontroller
%use the commented expressions to calculate new coefficients
p = 2*pi*Ba;
q = pi^2*Ba^2 + 4*pi^2*fa^2;
b0 = 1/128%2*G/(T*(4/T^2 + 2*p/T + q))%1/128
b1 = 0
b2 = -b0
a1 = 41/128 %(2*q - 8/T^2)/(4/T^2 + 2*p/T + q)
a2 = 126/128%(4/T^2 - 2*p/T + q)/(4/T^2 + 2*p/T + q)
y1 = 0;
y2 = 0;
x1 = 0;
x2 = 0;


%create signal
N = 1000; % Number of Samples
f = 69000; % Frequency of Sine Wave
n = 0:1:N;
t = n/fs;
snrdB = 30;

x = sin(2*pi*f*n/fs).*10^(snrdB/20); %create sinus signal
noise = fs/2000*randn(1,N+1); %create gaussian noise
x = x +  noise; %add random noise to signal

sigPowerdB = 10*log10(x*x'/length(x));
noisePowerdB = 10*log10(noise*noise'/length(noise));
SNR = sigPowerdB - noisePowerdB

%Filter
for k=1:N+1
    y(k) = (-a1*y1 - a2*y2 + b0*x(k) + b1*x1 + b2*x2);
    y2 = y1;
    y1 = y(k);
    x2 = x1;
    x1 = x(k);
```

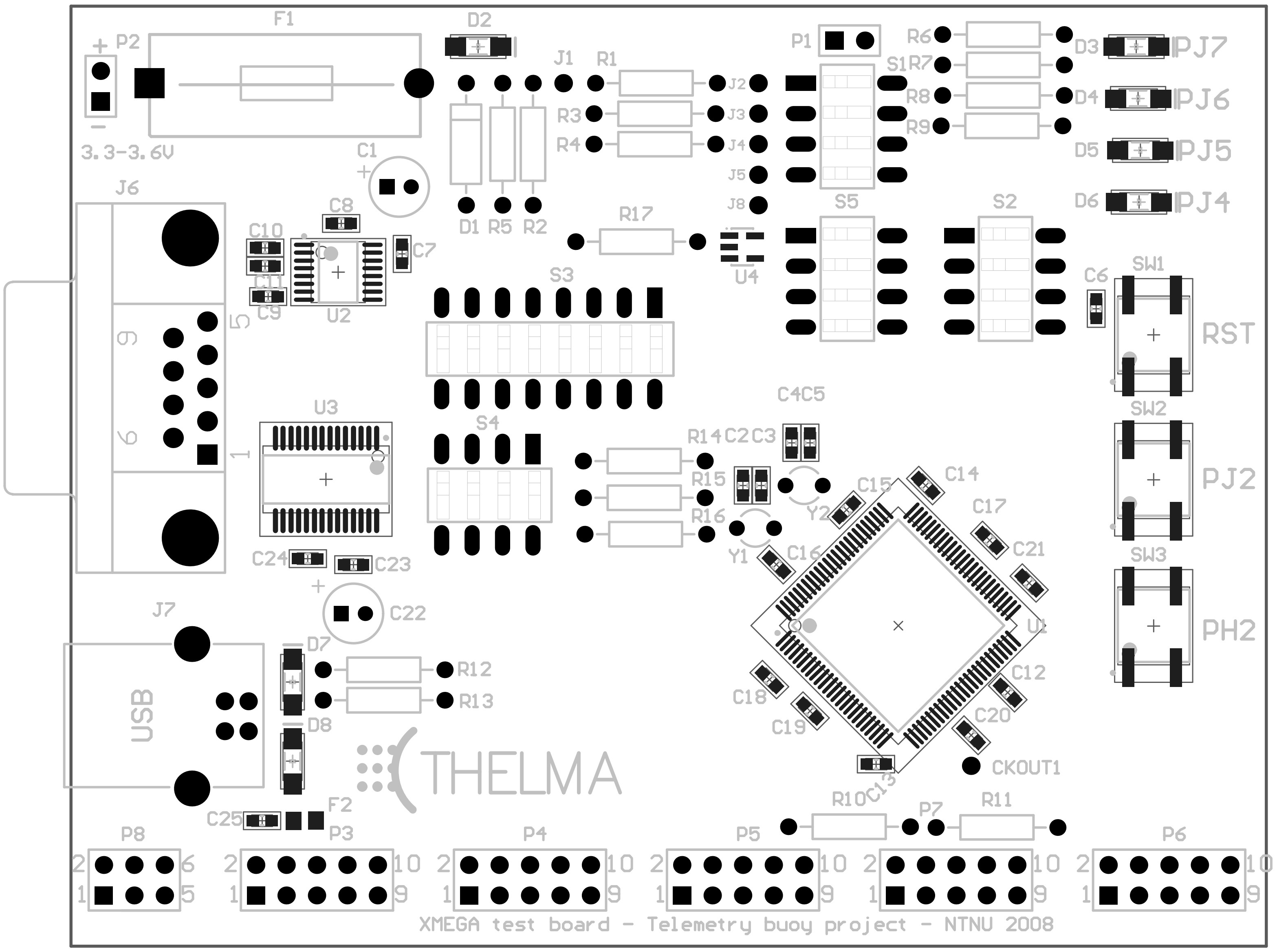
```
end


%FFT plot

plot(t,x)
hold on
plot(t,y,'r')

%original signal
NFFT = 2^nextpow2(N+1); % Next power of 2 from length of y
X = fft(x,NFFT)/(N+1);
f = fs/2*linspace(0,1,NFFT/2);

%filtered signal
Y = fft(y,NFFT)/(N+1);

% Plot single-sided amplitude spectrum.
xmin=0;
xmax=100000;
ymin=0;
ymax=1;

subplot(4,1,1), plot(f,2*abs(Y(1:NFFT/2)))
%axis([xmin xmax ymin ymax])
title(sprintf('Frequency plot - Filtered signal - Sampling frequency:%d Hz, SNR: %d↙
dB', fs, snrdB));
xlabel('Frequency (Hz)')
ylabel('|X(f)|')

subplot(4,1,2), plot(f,2*abs(X(1:NFFT/2)))
title(sprintf('Frequency plot - Original Signal - Sampling frequency: %d Hz', fs));
%axis([xmin xmax ymin ymax])
xlabel('Frequency (Hz)')
ylabel('|X(f)|')

xmax=N;
ymax=3.5;
ymin=-ymax;
subplot(4,1,3), plot(x)
title(sprintf('Original signal - Sampling frequency: %d Hz', fs));
%axis([xmin xmax ymin ymax])
xlabel('time')
ylabel('x(t)')

ymax=0.5;
ymin=-ymax;
subplot(4,1,4), plot(y)
title(sprintf('Filtered signal - Sampling frequency: %d Hz', fs));
%axis([xmin xmax ymin ymax])
xlabel('time')
ylabel('y(t)')



% Plot frequency response for filter
```

```
figure(2)
[H,W,S] = freqz([b0 b1 b2],[1 a1 a2],1024,fs);
S.xunits = 'hz';
freqzplot(H,W,S);
```

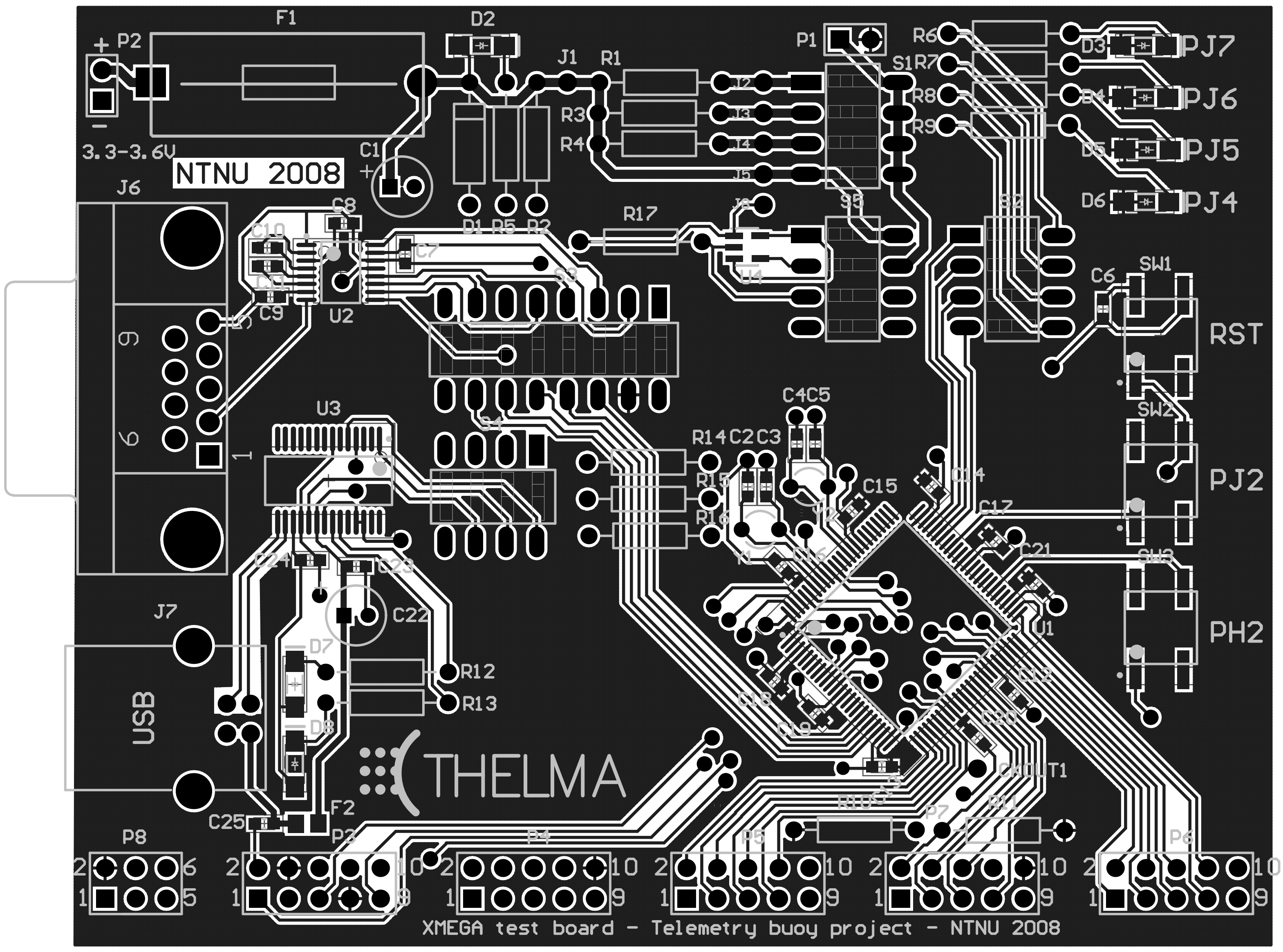
4. Prototype documentation

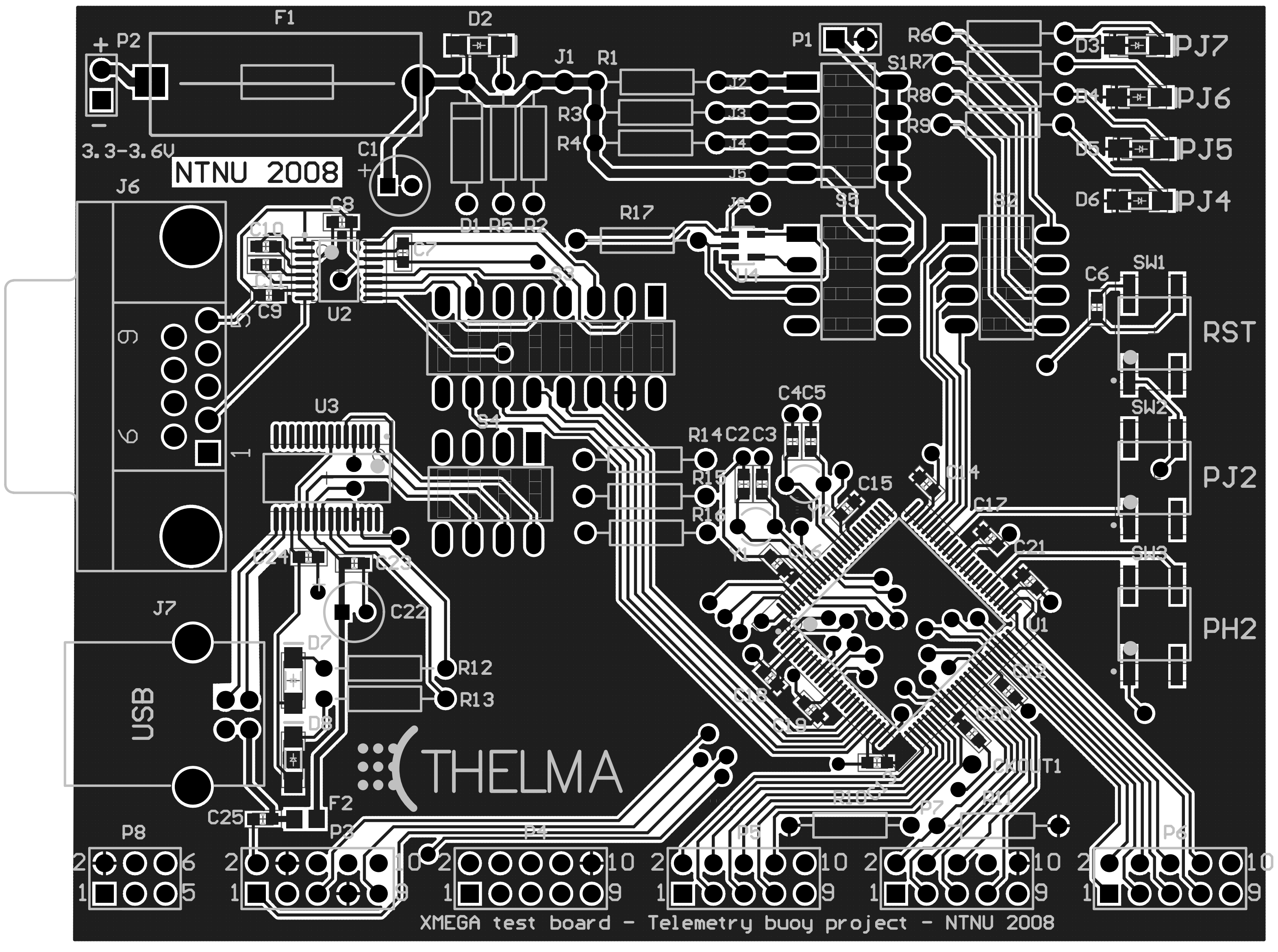


Title		
Size	Number	Revision
A3		
Date:	15.02.2009	Sheet of
File:	C:\Mine Dokumenter\..\ATxmegaA1 testkort\schematic.SchDoc	Drawn By:



F1
D2
P2
+
−
3.3-3.6V
J1
R1
P1
R6
S1
R7
R8
R9
D3
PJ7
D4
PJ6
D5
PJ5
D6
PJ4
J2
J3
J4
J5
J8
R3
R4
C1
D1
R5
R2
R17
U4
S5
S2
J6
C8
C10
C7
C11
C9
U2
S3
C6
SW1
RST
5
9
6
1
U3
S4
C4C5
R14
C2
C3
R15
R16
Y2
Y1
C15
C14
C17
C21
C16
SW2
PJ2
SW3
PH2
C24
C23
C22
J7
USB
D7
D8
R12
R13
U1
C12
C18
C19
C20
CKOUT1
THELMA
F2
C25
R10
C13
P7
R11
P8
P3
P4
P5
P6
2
1
6
5
10
9
XMEGA test board - Telemetry buoy project - NTNU 2008


P2
F1
D2
P1
R6
S1R7
R8
R9
D3
PJ7
D4
PJ6
D5
PJ5
D6
PJ4
J1
R1
R3
R4
3.3-3.6V
NTNU 2008
C1
J6
C8
C10
C7
C11
C9
U2
D1 R5 R2
R17
U4
S5
S2
C6
SW1
RST
U3
S4
C4C5
R14 C2 C3
R15
R16
C15
C14
C17
C21
SW2
PJ2
SW3
PH2
U1
C24
C23
C22
J7
USB
D7
R12
R13
D8
THELMA
CKOUT1
C25
F2
P3
P4
P5
R10
P7
R11
P6
P8
XMEGA test board - Telemetry buoy project - NTNU 2008



F1
D2
P2
3.3-3.6V
NTNU 2008
P1
J1
R1
R3
R4
C1
S1
R6
R7
R8
R9
PJ7
PJ6
PJ5
PJ4
D3
D4
D5
D6
J6
C8
C10
C7
U2
R17
U4
S5
S2
C6
SW1
RST
SW2
PJ2
SW3
PH2
U3
S3
S4
C4C5
R14 C2 C3
R15
R16
C15
C14
C17
C21
C24
C23
C22
J7
USB
R12
R13
D7
D8
F2
C25
THELMA
CKOUT1
U1
R10
R11
P7
P8
P3
P4
P5
P6
XMEGA test board - Telemetry buoy project - NTNU 2008

Bill of materials

Description	LibRef	Designator	Quantity	Value
Polarized Capacitor (Radial)	Cap Pol1	C1	1	47uF
Capacitor	Cap	C2, C3, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, C20, C21, C23, C24	20	100nF
Polarized Capacitor (Radial)	Cap Pol1	C22	1	4.7uF
Capacitor	Cap	C25	1	10nF
Capacitor	Cap	C4, C5	2	20pF
Header, 1-Pin	Testpin	CKOUT1, J1, J2, J3, J4, J5, J8	7	
Zener Diode	D Zener	D1	1	
SMD LED	LED	D2, D3, D4, D5, D6, D7, D8	7	
Fuse	Fuse 1	F1	1	100mA
Ferrite Bead	Fer2	F2	1	
Receptacle Assembly, 9 Position, Right Angle	D Connector 9	J6	1	
USB connector	USB connector	J7	1	
Header, 2-Pin	Header 2	P1, P2	2	
Header, 5-Pin, Dual row	Header 5X2	P3, P4, P5, P6, P7	5	
Header, 3-Pin, Dual row	Header 3X2	P8	1	
Resistor	Res2	R1	1	200R
Resistor	Res2	R10, R11	2	10K
Resistor	Res2	R17	1	500K
Resistor	Res2	R2, R14, R15, R16	4	1K
Resistor	Res2	R3	1	500R
Resistor	Res2	R4	1	socket
Resistor	Res2	R5, R6, R7, R8, R9, R12, R13	7	100R
DIP Switch, 4 Position, SPST	SW DIP-4	S1, S2, S4, S5	4	
DIP Switch, 8 Position, SPST	SW DIP-8	S3	1	
Push button	Button	SW1, SW2, SW3	3	
Atmel AVR RISC Microcontroller	ATxmegaA1	U1	1	
MAX3221e - Rs232 driver	MAX3221e - Rs232 driver	U2	1	
FT232R - USB - UART bridge	FT232R - USB - UART bridge	U3	1	
Current Chunt amplifier	INA138NA, Current shunt monitor	U4	1	
Crystal Oscillator	XTAL	Y1, Y2	2	0.4 - 16Mhz, 32.768Khz