Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Filière Systèmes industriels

Infotronics

Projet de diplôme 2014

Aurélien Merz

Low Energy Bluetooth & Algorithms

Professeur Medard Rieder

Sion, le 11 juillet 2014

HES-SO Valais-Wallis • rte du Rawyl 47 • C.P. • 1950 Sion 2
+41 27 606 85 11 • info@hevs.ch • www.hevs.ch

☒ FSI ☐ FTV	Année académique / *Studienjahr* **2013/14**	No TD / *Nr. DA* **it/2014/69**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*	Etudiant / *Student* **Aurélien Merz** Professeur / *Dozent* **Medard Rieder**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja [1] ☒ non / nein	Expert / *Experte (données complètes)* **Solioz Baptiste** SOPROD SA | Rue de la Blancherie 61 | 1950 Sion	

Titre / *Titel*

Low Energy Bluetooth and Algorithms

Description / *Beschreibung*

The goal of this diploma thesis is to develop a Bluetooth Low Energy (BLE) based device executing some complex algorithm onboard. The algorithm has to be tuned for low power. As a demonstrator, a guitar tuner system will be implemented. The system consists of a BLE device that is plugged or integrated into the guitar. It analyzes the played tone and sends this information to a second BL enabled device that acts as a human interface device. As an option, the played tone can be sent as samples to the second BLE device.

Objectifs / *Ziele*

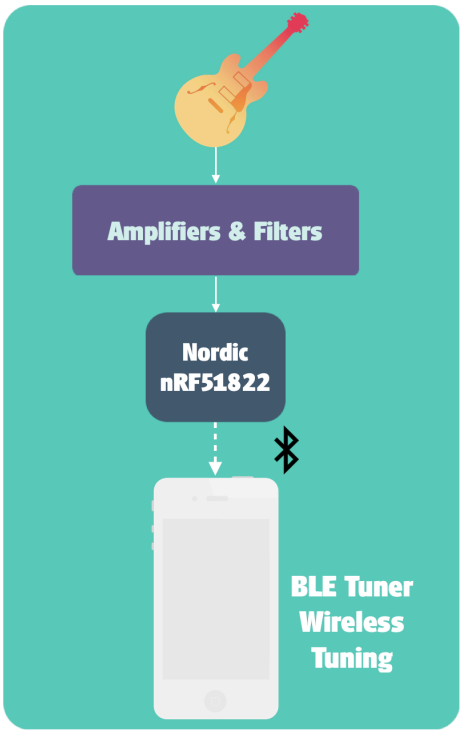
Analysis work has been done during the semester project and has produced promising results. It is therefor the goal of the diploma work to continue doing the design, implementation and test of the guitar tuner device prototype.

- Select the option to implement: Onboard data analysis or fast data transmission
- Implement and test the code of the complete solution using the Nordic development kit
- Design the electrical schematic of the tuner devices
- Build and test the hardware of the tuner devices
- Deploy and test software on the tuner devices
- Optimize hardware and software
- Establish technical documentation
- Establish a final report

Signature ou visa / *Unterschrift oder Visum* Responsable de l'orientation *Leiter der Vertiefungsrichtung:*..... [1] Etudiant / Student :	Délais / *Termine* Attribution du thème / *Ausgabe des Auftrags:* 12.05.2014 Remise du rapport / *Abgabe des Schlussberichts:* 11.07.2014, 12:00 Expositions / *Ausstellungen der Diplomarbeiten:* 27 – 29.08.2014 Défense orale / *Mündliche Verfechtung:* Semaine | Woche 36

[1] *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.*
Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....



Low Energy Bluetooth & Algorithms

Graduate

Aurélien Merz

Objectives

Develop a Bluetooth Low Energy (BLE) based device that analyses the tone produced by an electric guitar and sends this information to another BLE device operating like a human interface.

Methods | Experiences | Results

The Nordic Semiconductor chip nRF51822 is used to detect the frequency of a signal produced by an electric guitar using an adaptive counting algorithm. For a given number of samples converted from the Analog/Digital converter, the nRF51822 sends the number of times the signal has crossed a certain value, the AD middle range for instance.

The algorithm was first tested with generated signals, directly injected into the chip AD converter. Then a hardware prototype was built to test the system with an electric guitar.

An iOS application was developped in order to collect the data transfered from the nRF51822. This application operates the tuning algorithm as well as the display for the user.

Bachelor's Thesis
| 2014 |

Degree course
Systems Engineering

Field of application
Major

Supervising professor
Medard Rieder
medard.rieder@hevs.ch

Contents

THANKS
Special thanks to the following persons:
Medard Rieder
Thierry Hischier
Gilbert Maître
François Corthay
Thibaud Rossini

July 11, 2014

1 Introduction

The goal of this diploma thesis is to develop a Bluetooth Low Energy device that can tune an electric guitar using the Nordic Semiconductor nRF51822 chip. The system will transfer data converted from the string tone to another BLE device acting like a human interface.
Research has been made during the semester project on how capable the Nordic Semiconductor nRF51822 chip was to deal with complex mathematical algorithms and data treatments in relation to audio signals which vary in the range of 80[Hz] to 340[Hz]. The options to study are:

1. Compute Fast Fourier Transform directly on the chip,
2. Remote computing (fast data transmission),
3. Frequency detection by adaptive counting

The results from the semester project analysis showed that the chip, built on a Cortex M0 architecture, wasn't suitable for complex digital signal processing such as FFT. This diploma thesis goal is to continue working on the design, the implementation of new solutions and testing on the final hardware device.

2 Agenda

Here is the agenda for the thesis:

- Select option to implement: Onboard data analysis or fast data transmission,
- Implement and test the code of complete solution on the Nordic development kit,
- Design the electrical schematic of the tuner device,
- Build and test the hardware of the tuner device,
- Deploy and test software on the tuner device,
- Optimize hardware and software,
- Establish technical documentation

3 From Vibrations to Sounds

Before going into the details of the project, here is a brief summary about how an electric guitar works:
Electric guitars are built with pickups that transform mechanical vibrations into electricity. There are two types of pickups: active and passive. The difference between the two is that active pickups need a power source to operate their integrated preamplifier whereas passive pickups do not. Active pickups have also more undeniable characteristics:

- Bigger bandwidth implying more acute harmonics,
- Flat frequency response,
- Lower impedance reducing less probability of signal loss and static noise,
- Higher output level than passive pickups,
- Cleaner sound

Pickups are essentially made of magnets and coils. As strings are made of steel, and as they come back and forth (oscillatory movement), they interact with the magnetic field created by the magnets and then produce an electric polarised signal that goes through a jack into an amplifier [1].
Here are the fundamental frequencies produced by guitar strings:

Guitar Frequencies Range			
Note	Frequency [Hz]	Piano	String Number
E	82.41	E2	6
A	110.0	A2	5
D	146.83	D3	4
G	196.00	G3	3
B	246.94	B3	2
E	329.63	E4	1

Table 1: Guitar Frequencies Chart

NB: An octave is formed by 12 tones which compose the chromatic scales: A, $A^{\#}$,B, C, $C^{\#}$, D, $D^{\#}$, E, F, $F^{\#}$, G, $G^{\#}$.

4 Technical Details

4.1 Hardware

1. Nordic Semiconductor nRF51822 with:
   - ARM® Cortex™-M0 32 bit processor, single-cycle multiplier, 3-stage pipeline,
   - 16 [MHz] Clock Frequency,
   - Memory: 256 kB or128 kB embedded flash program, 16 kB RAM,
   - 8/9/10 bit ADC - 8 configurable channels,
   - Supply voltage range 1.8 V to 3.6 V,
   - S100 series SoftDevice ready
2. Nordic Semiconductor nRF51822 Evaluation Kit with:
   - Seggers Debug Chip
3. Apple iPad Air, 128 GB Memory

4.2 Software

Here are listed the software used for development:

1. Nordic Semiconductor nRF51822 with:
   - nRFgo Studio,
   - nRF51 Software Development Kit (SDK),
   - Keil ARM project files,
   - S110 nRF51822 SoftDevice,
   - S110 SoftDevice programming tools
   - $\mu Vision4$
2. Apple XCode 5.1 SDK
3. LightBlue iPhone Application

5 Improvements From Previous Design

5.1 Sampling Frequency

An important fact was highlighted at the end of the semester project: The sampling frequency was uneven and not fully controlled due to analog conversions performed on an irregular timing. The problem is later solved by using a timer to trigger AD conversions.

The nRF51822 provides a system based on events and tasks. This system is called Programmable Peripheral Interface (PPI) enabling different peripherals to interact autonomously with each other using tasks and events and without having to use the CPU [2]. Here is the way this system is used:

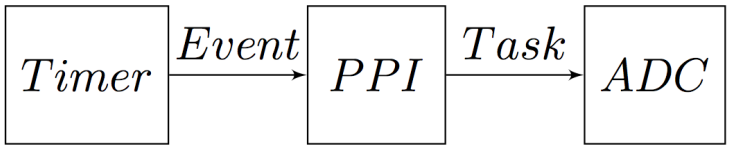


Figure 1: AD Conversion Triggering

The AD conversion task is triggered by an event coming from the timer. It is actually an overflow event. The timer can be configured in two different modes: Timer mode and Counter mode. The Timer mode enables to choose at which frequency the timer will increment its internal counter register. The timer frequency is derived from the chip's internal clock HFCLK:

$$f_{timer} = \frac{HFCLK}{2^{PRESCALER}} \tag{1}$$

The highest conversion resolution provided by the nRF51 is a 10 bit resolution. According to the datasheet, it takes approximately $68\mu s$ to complete a conversion. Choosing a PRESCALER of 7 corresponds to an $8\mu s$ overflow period. Thus counting 9 overflows takes $72\mu s$ letting a $4\mu s$ margin.

The taken sampling period for a 10 bit conversion using Timer and PPI is $80\mu s$. This corresponds to a frequency of 12.5 [kHz]. First tests were made with this frequency but the final implementation uses a sampling frequency of 4.5[kHz].

Figure 2 shows oscilloscope screenshots proving that the AD conversion and the timer overflow are synchronized.

A pin on the development board is toggled every time the ADC interrupt handler is called. The timer starts counting as soon as the device is connected to another BLE device:

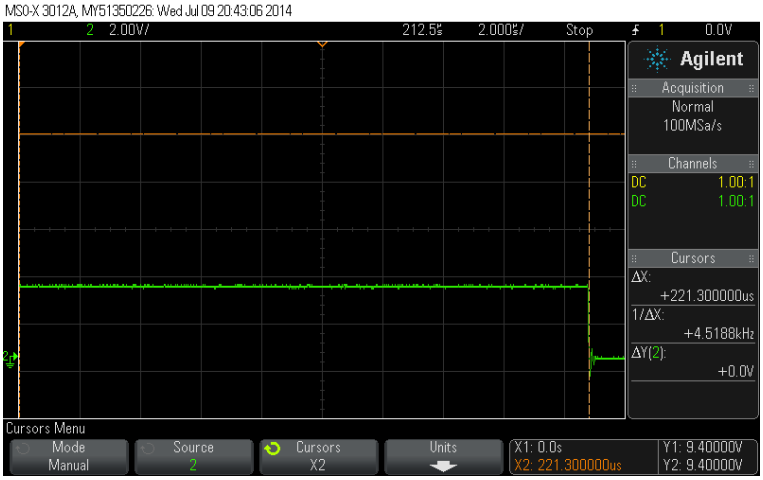


Figure 2: 4.5 kHz Timer Period Scope

The figure 2 shows the period is close to the theoretical one. Indeed, the Δx representing the value of the period which is 221.3 μs, the theoretical one is 222.222 μs.

5.2 Data Compression

Compressing data allows to send more information using less bits than the original representation. There are two types of compression: Lossy and lossless compression. Lossless compression optimizes the number of bits by identifying statistical redundancy and is typically used for important data files like bank records. On the other hand, lossy compression uses inexact approximation to represent the encoded data. It is used for multimedia data like images, audio streaming and internet telephony and many more applications. Lossless compression is the one used for the application [3].

The previous design implemented a 20 data bytes transmission per frame. This implementation allowed to send 10 measures packed into 2 bytes (10 bit coded). As the values were only 10 bit, a loss of memory space was created since the last 4 bits were actually zeros. Therefore, a compression algorithm was used to pack a maximum amount of measures in the given space.

Using this algorithm allows to send a maximum of 16 values per frame as it can hold 20 data bytes (160 bits). This decreases the transmission time in a non-negligible way:

Transmission Time for N = 1024	
First Design	Second Design
535 [ms]	330 [ms]

Table 2: Transmission Time

The transmission is about 38% faster than the first design. Another compression algorithm could be used as improvement: the Run-Length Encoding RLE. This algorithm consists of counting the number of time a data value appears and sent this number followed by the value. Here's an example:

$$AAAAAEEEDDDDDDAA$$
$$5A3E6D2A$$

Figure 3: Run-Lenght Encoding

This algorithm is simple to implement, allows to send data much quicker than the previous algorithm and is lossless [4].

6 Practical Implementation

6.1 Data Debit Difficulties

Even after being improved, the system presents some difficulties in terms of transmission. The previous section showed it takes approximately 330 [ms] to transfer 1024 samples, or an average of 3100 samples per seconds. However, this reveals an important problem: some data is lost. Indeed, using a $80\mu s$ sampling period, the ADC provides 12'500 samples per seconds and only 3100 samples are transmitted:

$$n = \frac{12500}{3100} \simeq 4 \tag{2}$$

This means 4 times less data is transferred. It undeniably leads to data loss. This could be compared to a barrel filled with water(representing the data) with a bigger pipe in the entry than in the exit. If the discharge isn't important enough at the exit, the barrel fills up until it overflows(data loss).

6.2 Sampling Test

To ensure the nRF51822 is sampling correctly, a test using the Universal Asynchronous Receiver Transmitter UART is executed: Every AD conversions, taken at a chosen frequency, are transferred to a terminal using UART in order to be saved into a file. This file is then used into Matlab to plot the signal and compute the frequency. The test is realized under the following conditions:

1. Sampling frequency: 4[kHz],
2. UART Baudrate: 38400 [Bd],
3. 8 data bits, 1 stop bit, no parity bit,
4. 8 bit AD conversion

Three signals coming from a waveform generator were tested: 253[Hz], 400[Hz] and 1022[Hz] signals. A Matlab script is written for the test and basically computes a FFT and finds the frequency from it. The Matlab script for this test can be found in attachment A. The following image shows the data coming from the UART and shown in CoolTerm software.

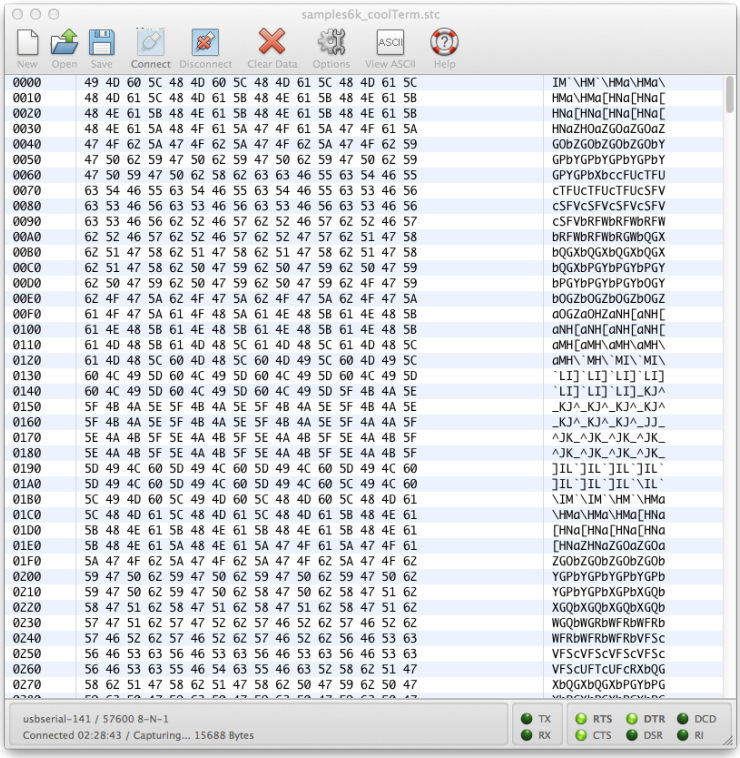


Figure 4: CoolTerm Terminal with AD Conversions

The result from the algorithm are convincing, as shown in figure 5:

Figure 5: Matlab Test Results

These results prove that the nRF51822 samples correctly.

6.3 Limits of Data Transmission

One solution to the transmission problem is to execute a downsampling before sending the data. Downsampling reduces the data rate of signal [5]. It is commonly used in audio processing and influences the lowpass filter implementation in such way that cut-off frequency is not be divided by 2 ($f_c = \frac{f_s}{2}$) but, in this case, by 8 meaning $f_s \simeq 1.6kHz$. Here is how the process is executed:

Only the sum of the last 4 sampled values are transferred instead of sending an average. Indeed, summing 10 bits variables will give a result coded with more bits, implying better resolution since dividing causes bit loss. Figure 6 illustrates the thing:

```
  1111111111
  1111111111
  1111111111
+ 1111111111
------------
 111111111100
```

Figure 6: 10 bit Numbers Sum

Adding one more bit to a binary number implies its range is doubled. Adding more bits to the signal implies the filter to attenuate more since the signal gain is 6 [dB] bigger per bit. Thus, 12 bits lead to an attenuation of 72 [dB]. On the other hand, dividing is performed by right shifting operations. The MSB bits are lost since they are replaced by zeros.

Nevertheless, this does not solve the problem entirely. This being said, a new solution to the problem is needed to avoid massive data transmission: Frequency detection by adaptive counting. This new solution is presented in the following section.

7 Frequency Detection by Adaptive Counting

As the data transmission is not fast enough at all because of the massive amount of data to transfer, the new solution is to perform an analysis directly on the chip but not a FFT since it has already been proven not to work. The principle of this technique is to count the number of value occurrences during a specific period of time, knowing the sampling period. The number of occurrences during this period allows to compute the input signal frequency. The numbers of occurrences is sent over the BLE, transmitting less data than the previous implementations.

7.1 Finite State Machine Algorithm

The first implementation of this frequency detection uses a finite state machine and two functions. Each time a AD conversion is done, the AD value is given as a parameter to a first function that will update a state machine. Then another function is called with the current state as parameter. Depending on the state, the function increments or reset a counter. The value of the counter is transferred with the BLE to another device like an iPhone for instance. This value represents a scalar of the desired frequency compared to the sampling frequency.
Figure 7 illustrates the finite state machine diagram:

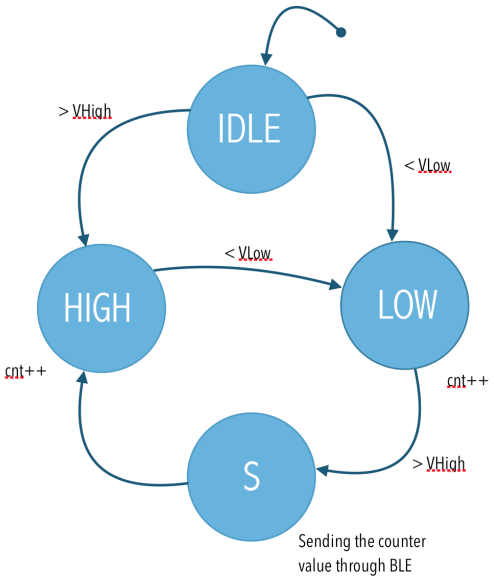


Figure 7: Finite State Machine

The following figure shows how the algorithm's behavior in time domain:

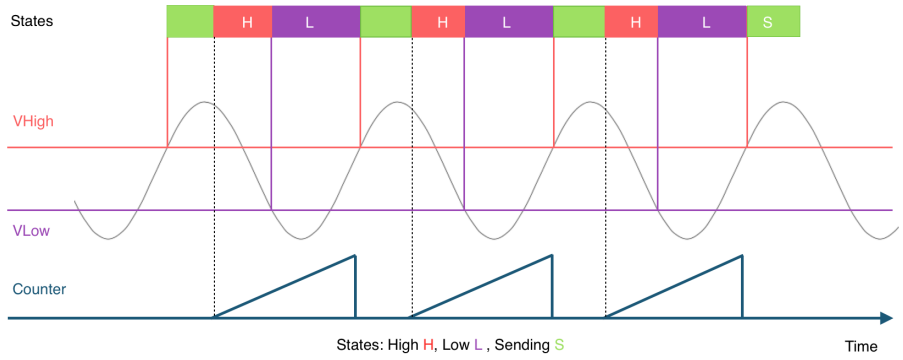


Figure 8: FSM Algorithm Behavior in Time Domain

The advantages and disadvantages of this method are listed below:

1. Advantages:
   - Very little time into the ADC interrupt,
   - Very little data to transmit (counter value only),
   - Security: Using hysteresis avoiding perturbations to disturb the algorithm,
   - Easy to implement,
2. Disadvantages:
   - Insufficient precision due to nature of value as integer type, even if counting with float
   - Variation in values is too high

This being taken in consideration, tests are made and discussed in the discussion section.

7.2 Adaptive Counting Algorithm

Correct tuning is normally done using Cent which is a precise unit of musical intervals based on logarithmic scales. A cent is defined as the 100^{th} of a semitone. An octave is made of 1200 cents as the chromatic scale is made of 12 semitones [6]. Thus tuning implies to make the report of a frequency to a reference:

$$c = 1200 \cdot \log_2(\frac{f_1}{f_2}) \tag{3}$$

To avoid using this mathematical equation and as samples transmission is not giving great results, another method is used.

This method consists in counting the number of time a reference value is crossed for a certain number of samples. The algorithm compares the previous and the actual value of the AD conversion to the reference: If the previous value is higher and the actual lower than the reference, then a counter is incremented. This means, the count is performed only on the falling edge of the signal. As there's a sampling period between each samples, for a chosen number of samples corresponds a certain duration:

$$T_{total} = \frac{N_{samples}}{f_{sampling}} \tag{4}$$

Knowing this duration and the numbers of time the signal crosses the reference, the frequency of the analyzed signal can be found:

$$Frequency = \frac{N_{pass}}{T_{total}} \tag{5}$$

Here are the advantages and disadvantages of this method:

1. Advantages:
   - Only uses incremented variables,
   - Easy implementation,
   - Efficient and fast
2. Disadvantages:
   - Much more data to transfer: number of crossings and counter,
   - Insecure reference: It is not safe enough to use only one reference because the number of crossings is not correct if the signal is disturbed within a period.

Using the data coming from the UART, a Matlab script is written to prove the algorithm works before implementing it on the chip. The script can be found in attachment B. Here are the results for the same signals and UART configuration from section 6.2:

Figure 9: Adaptive Counting Matlab Test Results

Figure 9 shows the solution is working without needing any FFT or complex mathematical algorithms. These results are satisfactory.

The advantages of both technics lies in the fact they don't perform mathematical operations using floats on the nRF51822. These operations are executed on the other BLE device.

7.3 Transmission to Another BLE Device

After showing the adaptive counting algorithm is working, the data transmission is performed. There are two options that can be used concerning transmission:

1. Send frequency only:
   As the value is of the type $float32$, it can't be sent without being type casted as the nRF51 data frames are only made of bytes of type $uint8$. Then the frequency has to be casted before the transmission. This affects the precision of the value in the way that the decimals are lost. Another thing to mention is it takes many CPU cycles to compute the frequency since it uses $float32$ types.

2. Send counter value only:
   As mentioned above, float computation takes a lot of CPU cycles, especially division operation. This affect the performance of the algorithm and system in terms of speed and reactivity. To avoid computing with float, only the number of passes and the number of samples are transmitted. The other BLE device computes the float operation instead

of the nRF51. This implicates sending more bytes, 8 bytes actually (2 *uint*32 values), than only one but this does not affect the precision of the future frequency.

The second option is used for the final implementation.

8 Tuner Device Hardware

This section explains how the hardware is designed and realized.

8.1 Guitar Output Signal

All the first tests made with the ADC were done using precise, not to say perfect, signals generated from a waveform generator. A few tests are done with an electric guitar to see how the real signals look like and how they behave.
There are a lot of different ways to use guitar pickups depending on how they are combined, but for analyzing the signal form, it does not play a big role except if a frequency analysis is done.

Here are the voltages output for each strings using passive pickups.

Guitar Voltage Outputs	
Note	Voltage [mV]
E	$\sim$ 165
A	$\sim$ 125
D	$\sim$ 100
G	$\sim$ 60
B	$\sim$ 84
E	$\sim$ 108

Table 3: Passive Pickups Voltage Outputs

The following figures are scopes of E (82.41 Hz) and A (110 Hz) strings with their respective octaves using passive pickups:

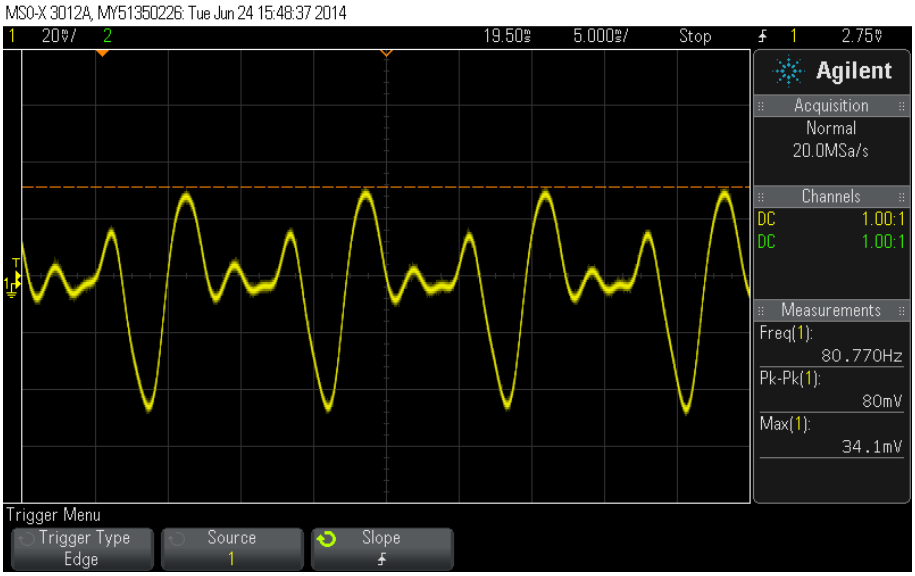


Figure 10: E String Output Signal

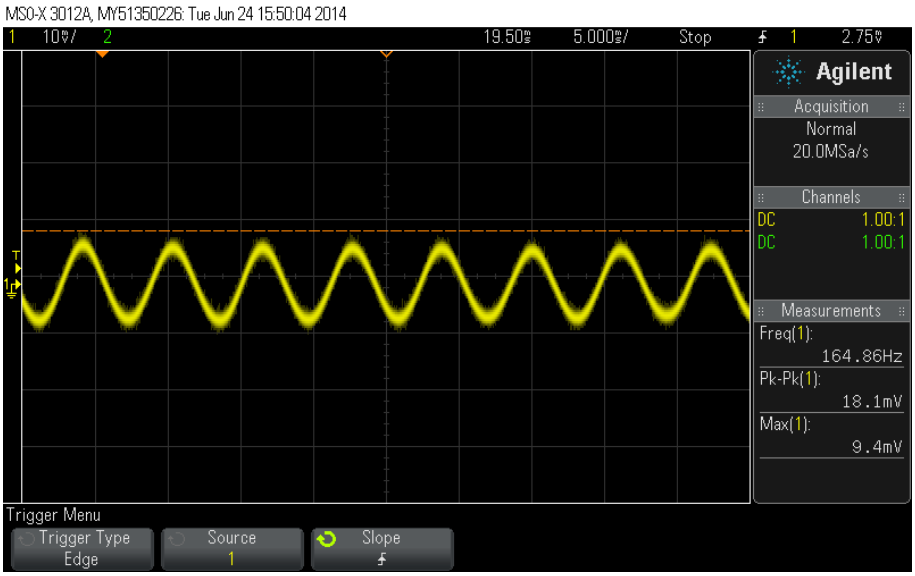


Figure 11: E String Output Octave Signal

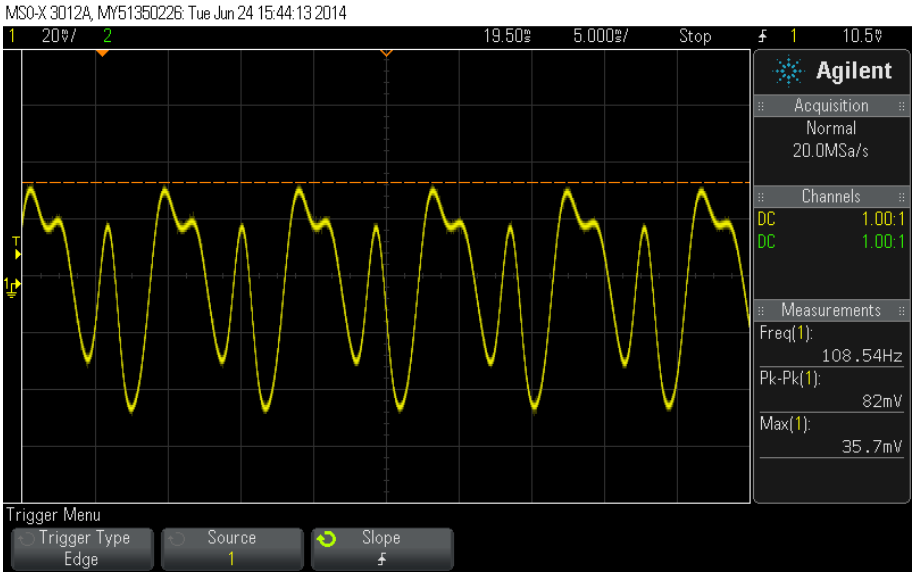


Figure 12: A String Output Signal

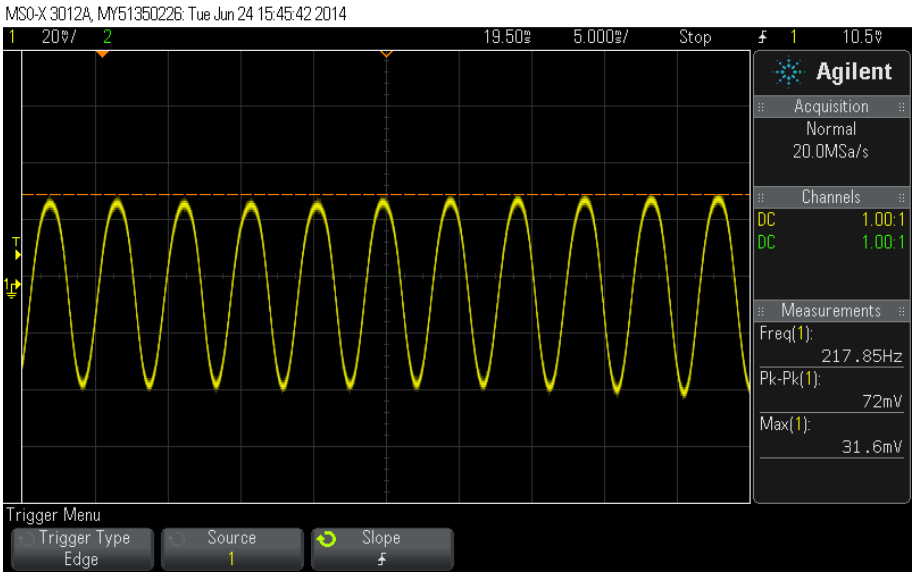


Figure 13: A String Output Octave Signal

The previous figures show frequencies are close to theoretical ones, especially when playing the octave. It is interesting to note that the signal gets much more accurate when playing the octave than the fundamentals.

Another test is performed using a 5 string bass guitar built with both active and passive pickups. Here are the output voltage:

Active & Passive Voltage Outputs			
Note	Passive	Active	String Number
C	> 320[mV]	> 511[mV]	5
E	300[mV]	> 511[mV]	4
A	$250 \rightarrow 300$ [mV]	> 511[mV]	3
D	$370 \rightarrow 405$ [mV]	> 511[mV]	2
G	312 [mV]	> 511[mV]	1

Table 4: Passive & Active Bass Pickups Voltage Outputs

The oscilloscope constantly show a voltage output bigger than 511[mV] for the active pickups. A test with a multimeter is also done but it can not refresh the value faster enough to see the peak.

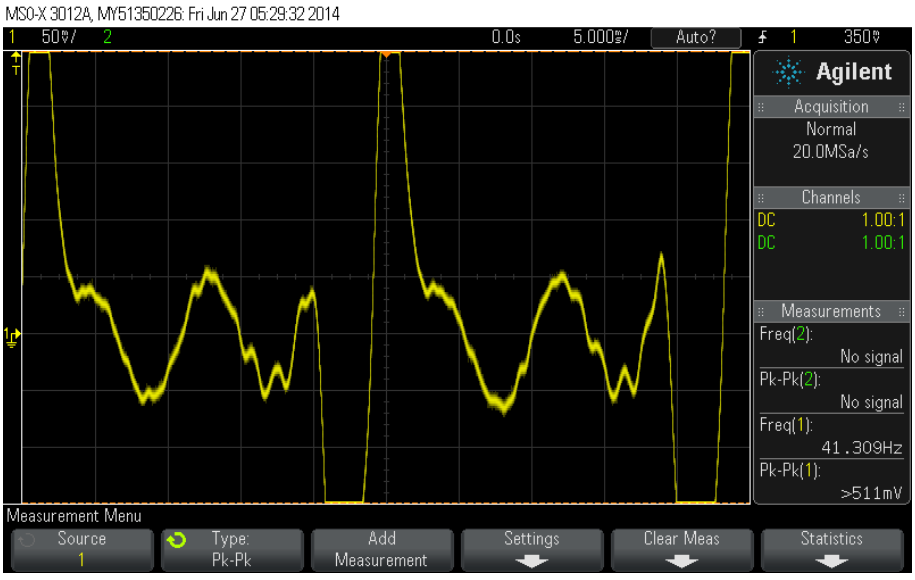


Figure 14: Active Pickups Output Signal

These tests undeniably show that the signal needs to be conditioned (amplification and filtering) before being analyzed because there are too many interfering elements like the vibration of other strings.

8.2 Amplification

The output signal out of the guitar is polarised. It varies from -U [mV] to +U [mV]. The ADC range goes from 0 [V] to 1.2 [V], it is actually the VGB internal reference of the chip itself. Using a 10bit conversion, this means the middle of the range, corresponding to 600[mV], has a value of 512. The final adaptive counting algorithm uses the value of 512 as a trigger for a counter. If the signal is under this value, the algorithm does not work because of its too small quantized value. This being said, the input voltage has to be amplified in order to fit the ADC scale and the algorithm limit.

The best way to ensure the algorithm works, is to make a 600[mV] offset and amplify the signal to make it clip into the ADC. It doesn't matter if it doesn't get the full signal because only the passing to the 512 middle range value. Figure 15 illustrate the circuit:

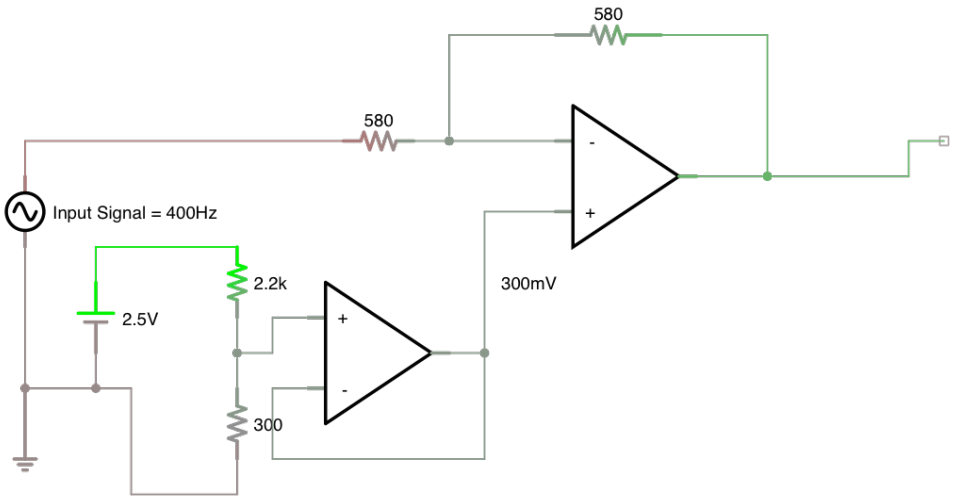


Figure 15: Amplifier with Offset Second Prototype

This stage has a gain of 2. The 300[mV] is applied to the positive input in order to be amplified twice (report of the resistors). This insures a constant 600[mV] input value to the ADC. The circuit schematics can be found in attachment C.

8.3 Anti-aliasing Filtering

Before sampling an analog signal, its bandwidth needs to be restricted in order to conform to the sampling theorem. The system in charge to realize this operation is called an anti-aliasing filter: Its job is to filter the signal frequencies that do not contain pertinent informations or informations that are not relevant for the application. It is also used to avoid the aliasing effect created by the sampling. The type of filter used are lowpass filters, which allows a certain band of frequencies to pass before attenuating them.

The filter has to have precise specifications:

1. Cut-off frequency: $f_c < \frac{f_s}{2}$, f_s as sampling frequency,
2. Gain in the pass band in dB,
3. Ripple margin in dB,
4. Attenuation in the cut band in dB,
5. Type: Butterworth, Chebyshev, Elliptic or Bessel

Here are the different types of filters and their frequency responses:

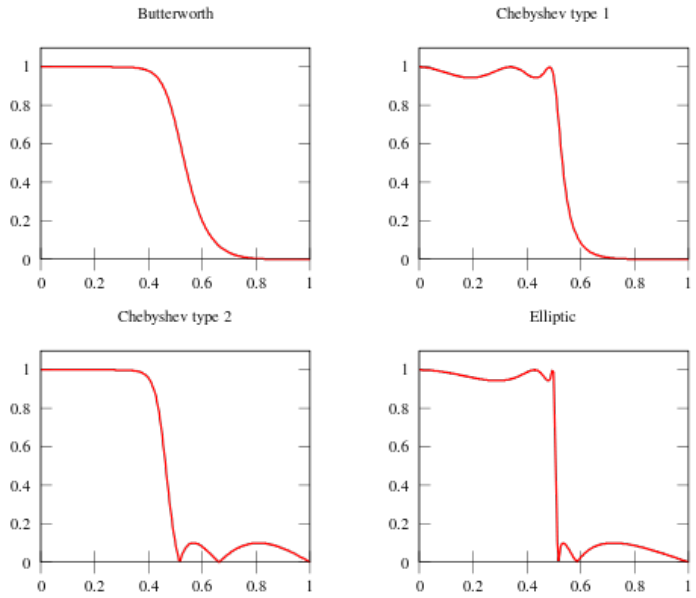


Figure 16: Electronics Linear LP Filters

The highest frequency to analyze is about 330 Hz. In order to make a good frequency analysis, all frequencies up to the octave are taken in consideration. So the final specifications are the following:

1. Cut-off frequency: 800 [Hz],
2. Unity gain,
3. Ripple margin to 800 [Hz]: 1 [dB] $\leq$ A $\leq$ 3 [dB] ,
4. Attenuation at 1.6 [kHz]: $A_s = 80$ [dB],

After simulating several filters on Matlab, the Chebyshev Type 1 fits the best the application, giving an 8th order filter, the lowest in this case. As observed, the cut-off frequency is much lower than the possible sampling frequencies the nRF51822 can provide. This filter was actually designed because of the data transmission problem cited in a previous section.
In order to realize an 8th order lowpass filter, cascading 2nd order filters is the solution. The Sallen-Key topology is used to implement these filters.

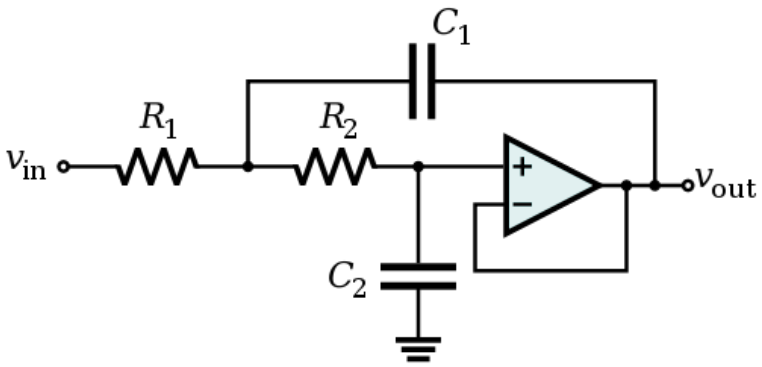


Figure 17: Sallen-Key Lowpass Filter Topology

Each stages has a unity gain and has the same type of transfer function:

$$H(s) = \frac{1}{1 + C_2(R_1 + R_2)s + (C_1C_2R_1R_2)s^2} \tag{6}$$

Stages also have their own specific quality factor Q and frequency scaling factor FSF. According to Texas Instruments filter design, here are the corresponding Q and FSF for each stage:

Filter Order	Stage 1		Stage 2		Stage 3		Stage 4	
8	FSF	Q	FSF	Q	FSF	Q	FSF	Q
	0.2228	1.0558	0.5665	3.0789	0.8388	6.8302	0.9870	22.8481

Table 5: 3-dB Chebyshev Filter Table for 8th Order

These two following equations allows to compute the value of the analog components using Q and FSF:

$$\begin{aligned} FSF \cdot f_c &= \frac{1}{2\pi CR \cdot \sqrt{mn}} \\ Q &= \frac{\sqrt{mn}}{m+1} \end{aligned} \tag{7}$$

Assuming that for chosen R and C:

$$\begin{aligned} R_1 &= mR \\ R_2 &= R \\ C_1 &= nC \\ C_2 &= C \end{aligned} \tag{8}$$

The Matlab script for the computing as well as the Texas Instrument filter design sheet are in attachments D and E.

The transfer function are disposed in a certain way, from the lowest Q to the highest to help avoid saturation.

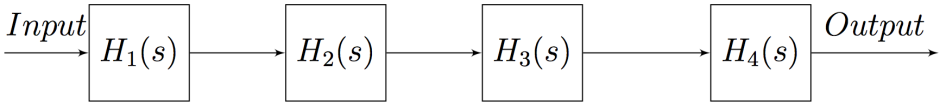


Figure 18: Transfer Functions Blocs

Here's the frequency response for the whole filter:

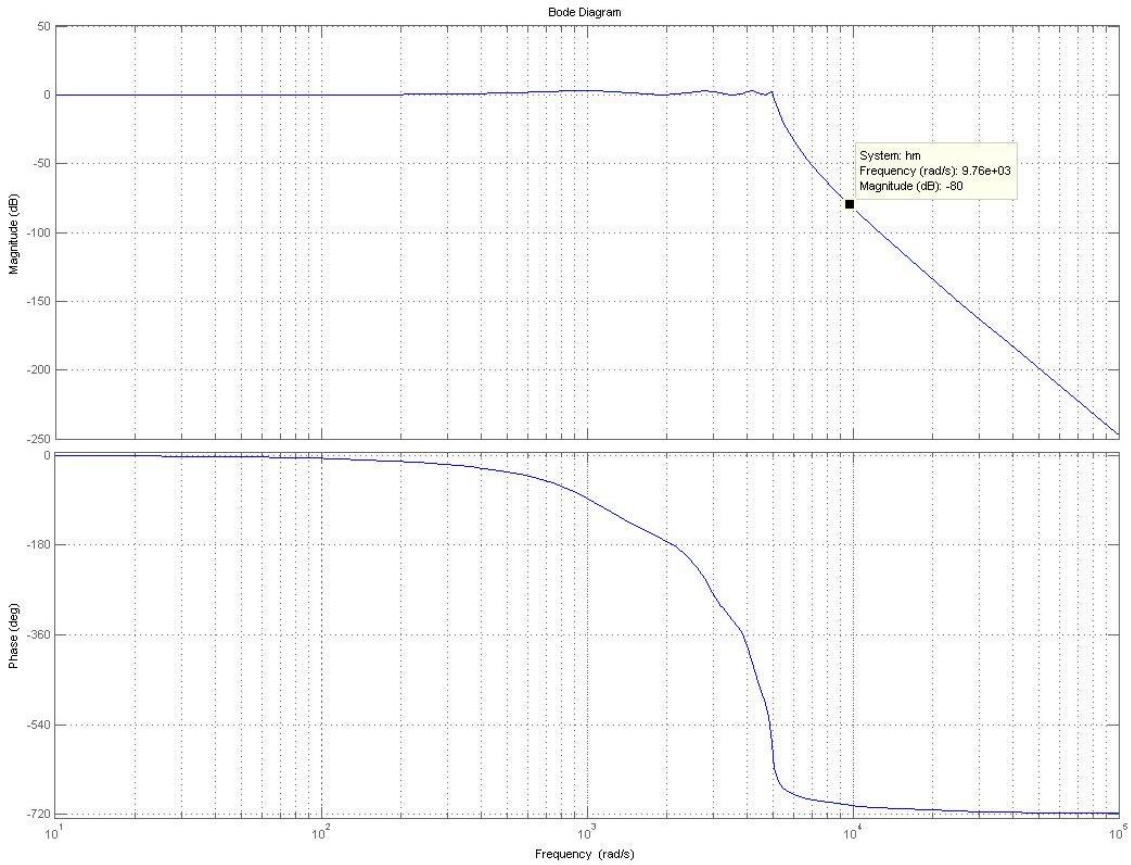


Figure 19: Chebyshev Type 1, 8th Order Lowpass Filter

The frequency responses for all the stages are in attachments as well as the schematics of the whole circuit. For better visualization, scopes of a filtered signal can be found in attachment F.

9 Software Architecture

Wireless data transfer is at the heart of the project. This section presents the two software developments done during this project: The programming part of the nRF51822 chip and the iOS application.

9.1 Bluetooth Low Energy

In the Bluetooth 4.0, devices communication is based on a client-server architecture. Peripherals (servers) are devices that have data needed by other devices. Centrals (clients) use the data provided by the peripherals to achieve tasks. The data is collected into structures called services. Data is called characteristic [7].

There are a lot of different services like the Battery Service or Health Thermometer to name a few. A custom service is made for the project and holds two characteristics:

- Crossings: The number of time the middle range is crossed
- Counter: The number of samples

These two values represent the two important parameter of the adaptive counting algorithm. They are sent to avoid computing with floats on the nRF51822 and computing them on the iOS device instead. As previously said, computing with floats is better in terms of precision. Loosing decimals can produce errors on the result.

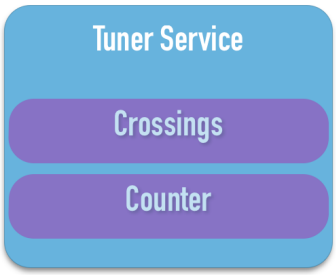


Figure 20: Tuner Service

9.2 nRF51822 Programming

The program is developed using KEIL μVision 4 and coded in C. The algorithm used to detect the frequency is directly performed on the chip. The following flowchart shows how the program operates:

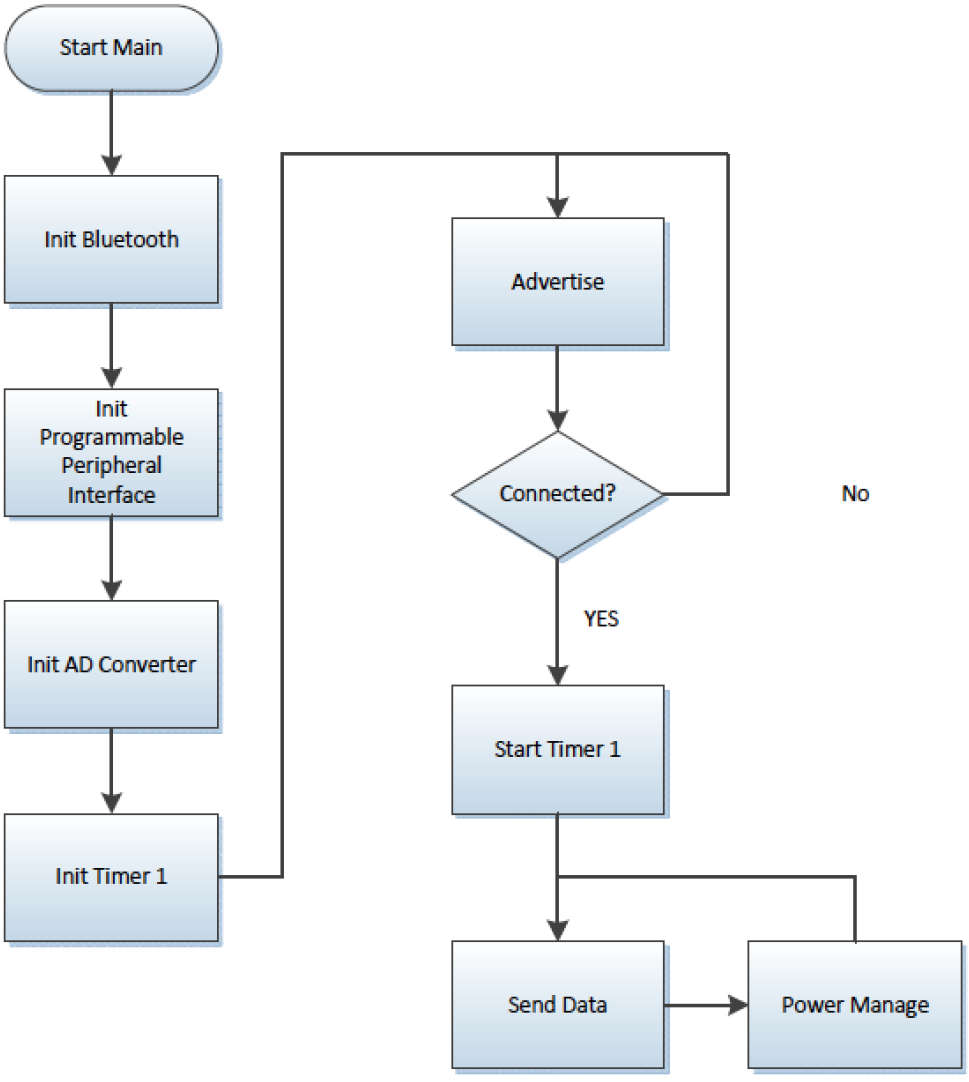


Figure 21: Main Function Flowchart

The algorithm is performed into the AD interrupt handler implying that it is updated each time an AD conversion is done.

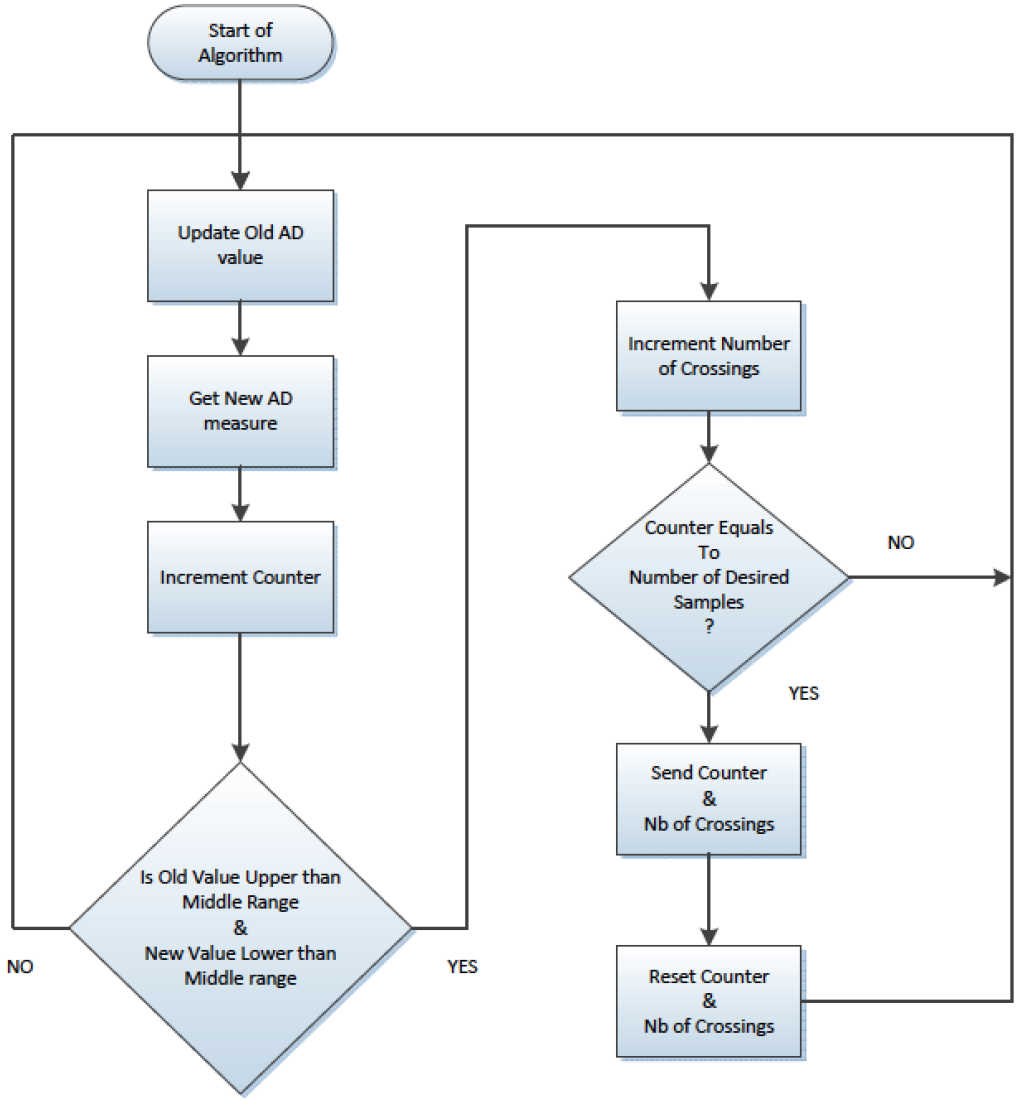


Figure 22: Algorithm Flowchart

Here's the code of algorithm corresponding figure 22. The full code can be found in attachment G.

9.3 iOS Application

The iOS application development is the final part of the project. The application collects the data transferred by the nRF51822 and operates like a human interface. It also computes the frequency from the data and displays information about the actual tone. This helps the user to correctly tune its instruments.

iOS provides the *CoreBluetooth* framework which is used for the application. This framework is needed for the device to communicate with other devices which implement the low power wireless communication protocol.

Here is the class diagram for the Objective-C software:

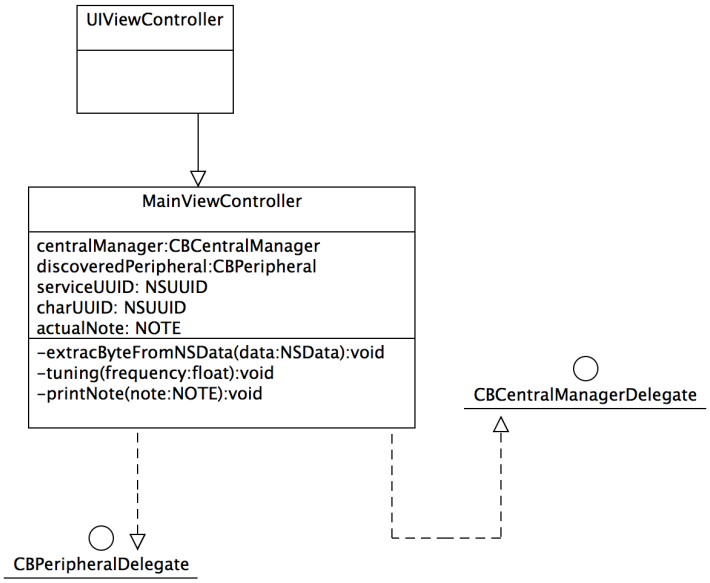


Figure 23: Objective-C Class Diagram

The MainViewController class implements the CBCentralManagerDelegate and CBPeripheralDelegate which are interfaces from the *CoreBluetooth* framework. Delegation is a pattern in which an object from a program can act on behalf or in coordination with another object. The delegating object has a reference to the other object, the delegate, in order to send it messages. The delegate is alerted about the delegating object handling events. This pattern is a kind of Observer pattern. These interfaces, called Protocols in Objective-C, provide methods for playing both roles of Peripheral and Central, as well as connection management, data transmission and reception

[7].

Once the connection between the iOS device and the nRF51822 is made, the nRF51822 starts sampling the input signal. The communication between the two devices is based on notifications: The nRF51822 notifies the iPad each time its characteristics are updated. The iPad then takes the new characteristics data in order to work with it. It actually computes the frequency.

Figure 24 illustrates this mechanism:

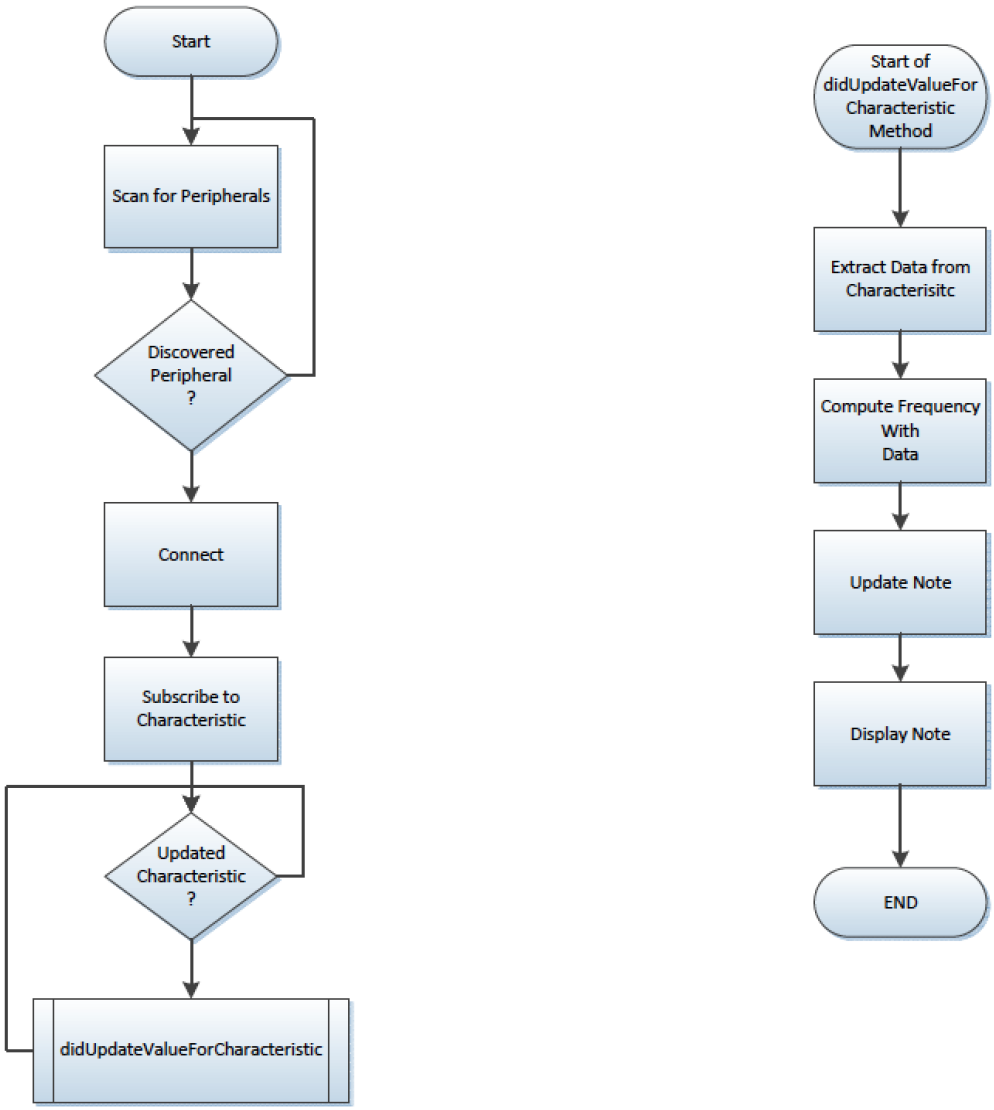


Figure 24: iOS Application Flowchart

The iOS application also manages the case of peripheral disconnection. If it is the case, the application restarts scanning for others peripherals to connect to.

The following figure shows how both devices interact with each other:

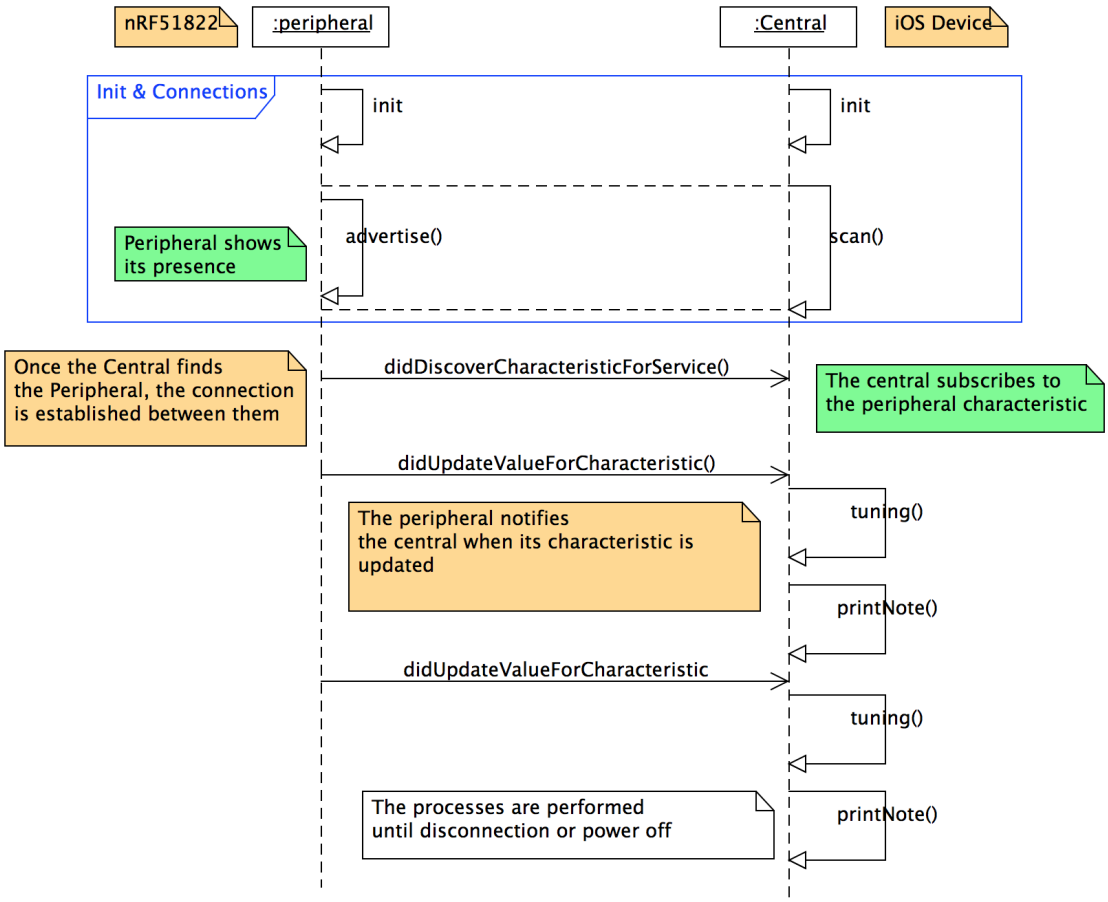


Figure 25: Communication Sequence Diagram

The tuning process uses two methods: *tuning* and *printNote*. The application analyses the computed frequency value and updates the display to help the user. As the frequency value is updated very fast, an average is computed to make the tuning easier. Gaps are used to determine the note. The full code is in attachment H.

The following figure shows test version of the application:

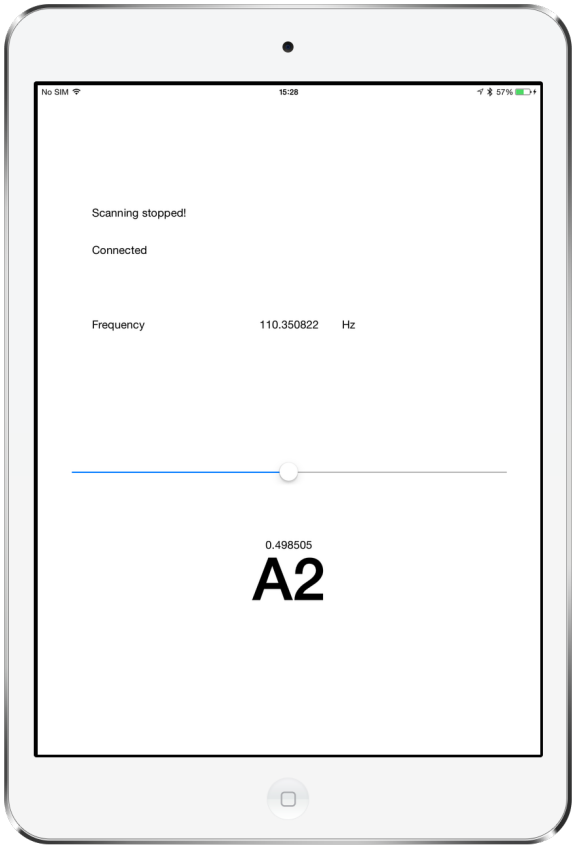


Figure 26: iOS Application Tuning v1.0

The application uses a slider to help the user know if the string is too tight or not enough to be tuned correctly. The slider constantly updates its maximas to fit a certain scale. The slider maximas scale is always corresponding to a ± 10[Hz] tolerance relative to a note. For instance when playing a E2 note which is 82.83[Hz], the maximas scale goes from 72[Hz] to 92[Hz].

In order to determine if the string is correctly tuned, a second scale is used. This time, the second scale has a ± 2[Hz] tolerance. For the E2 note example, if the computed frequency is between 80[Hz] and 84[Hz], the note is considered as correct. This way is the easiest one to perform a relative correct tuning.

10 Results

10.1 Hardware Performance & Energy Consumption

After working on the hardware design, a prototype is built. The prototype is a mezzanine plugged on the development kit with two 1/4 inch jack connectors (input and output) and all the electronic. Two versions of the prototype have been developed:

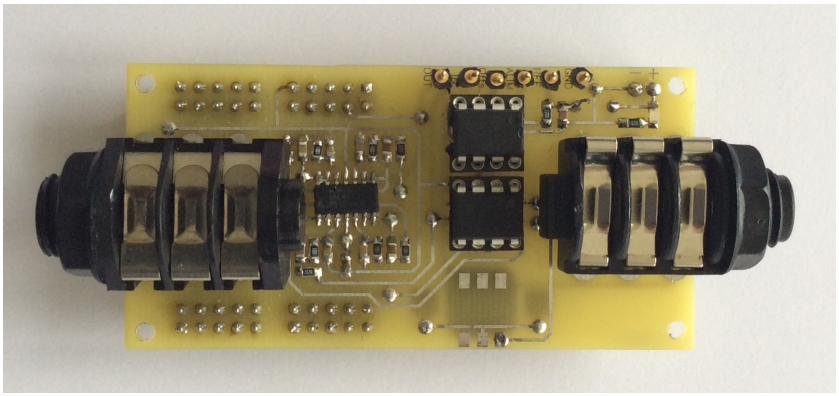


Figure 27: First Version Prototype

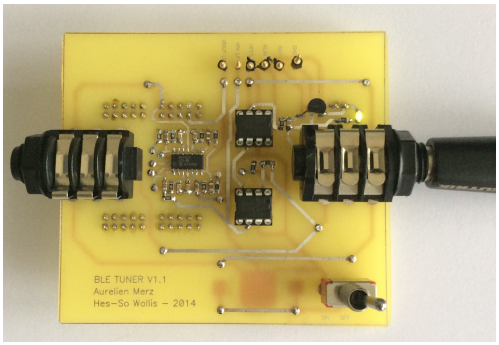


Figure 28: Second Version

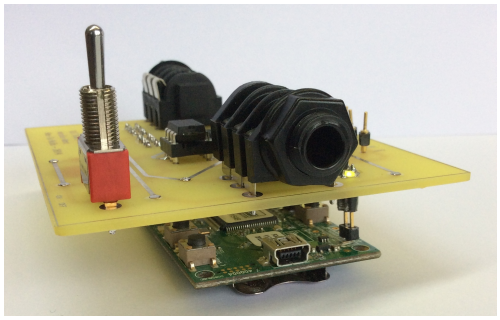


Figure 29: Complete Hardware

The only difference between the two prototypes lies in the in fact that the second one has a different amplification stage than the first one. The amplification is different in the way the first version makes the signal clip and the second one do not. The circuit schematics can be found in attachments C.

The mezzanine powers the whole system using a 3[V] battery. The power can be turned off when the device is not used. The following enumerations show the different consumptions for the first prototype:

1. Mezzanine only: 12.14[mA] $\rightarrow$ 12.20[mA]
2. Mezzanine with jack plugged: $\sim$ 11.50[mA]
3. Mezzanine and Development kit together: 16.63[mA] $\rightarrow$ 16.75[mA]
4. Mezzanine and Development kit with jack plugged: 15.83[mA] $\rightarrow$ 16[mA]

The systems consumption is about 14.62[mA] in average, this is a little too much for low energy systems. The second prototype consumption is not showed as the components are the same.

10.2 Tuning

The tuning algorithm is the final part of the project. Section 7 presented two ways to make frequency detection by adaptive counting. The results of both algorithms are listed below. The following tables show the computed frequencies without any correction:

1. First algorithm:
   As explained in a previous section, the algorithm computes the desired frequency using a counter and the sampling frequency with help of state machines.
   For instance, using this algorithm for a 400[Hz] signal sampled at 4[kHz], the theoretical counter value will be 10:

$$n = \frac{4000[Hz]}{400[Hz]} = 10 \tag{9}$$

   The algorithm has been tested and shows that the counter value varies between 9 and 10. A value of 9 gives a frequency of 444.444[Hz]. Making an average of those 2 possible values over N returns values in a certain range $9.45 \leq m \leq 9.85$.

2. Second algorithm:
   Signal frequencies from the waveform generator:

Note Frequency	Computed Frequency	Computed Average Frequency
82.41 [Hz]	93.75 ± 1[Hz]	94 ± 1[Hz]
110 [Hz]	125.43 ± 1[Hz]	125 ± 1[Hz]
146.83 [Hz]	160.13 ±1[Hz]	161 ± 1[Hz]
196 [Hz]	202 ± 1[Hz]	205 ± 1[Hz]
246.94 [Hz]	246 ± 2[Hz]	246 ± 1[Hz]
329.63 [Hz]	336 ± 1[Hz]	341 ± 1[Hz]

Table 6: Computed Frequencies

Table 6 highlights particular offsets except for the 246 [Hz] value. The chosen number of samples is 3000 and the sampling frequency is 4.5[kHz]. These two parameters allow to compute the acquisition time for the 3000 samples:

$$\begin{aligned} t_{acquisition} &= \frac{N_{samples}}{f_{sampling}} \\ &= \frac{3000}{4500} \\ &= \frac{2}{3} \end{aligned} \tag{10}$$

Thus the number of crossings can be computed with the following equation:

$$N_{crossings} = frequency \cdot t_{acquisition} \tag{11}$$

The following table shows number of crossings for the A2, D3 and the E4 notes:

Note Frequency [Hz]	Theoretical Crossings	Practical Crossings	Δ
110 (A2)	66.66	83.33	16.66
146.83 (D3)	97.33	106.667	9.36
329.63 (E4)	219.75	227.73	7.58

Table 7: Theoretical and Practical Number of Crossings

These offsets come from a problem in the adaptive counting algorithm. Indeed, using the debugger, the counter value appears not to always be equal to the desired numbers of samples which is 3000 in this case. Its value is always bigger.

The following section discusses the results.

11 Discussions

11.1 Adaptive Counting Problem

As described in the previous section, the signal is generated by a waveform generator which outputs a high quality signal. Signals produced by the guitar are irregular in terms of amplitude and power. Thus, the algorithm needs certain conditions to work properly:

1. Time:
   The signal is only stable and regular after an average of 5 seconds. This period of time is too long to make a correct analysis because of the absorption. The tuner starts to give better results as the signal gets stable but the signal gets too low to keep performing the algorithm correctly. Normal tuning needs an average time of 1 or 2 seconds before the musician strums the string again.

2. Signal regularity:
   As the algorithm only uses the 600[mV] ADC middle range for counting the number of time crossings, this number is not correct because of the uneven signal coming out of the guitar. The following figure shows this problem:

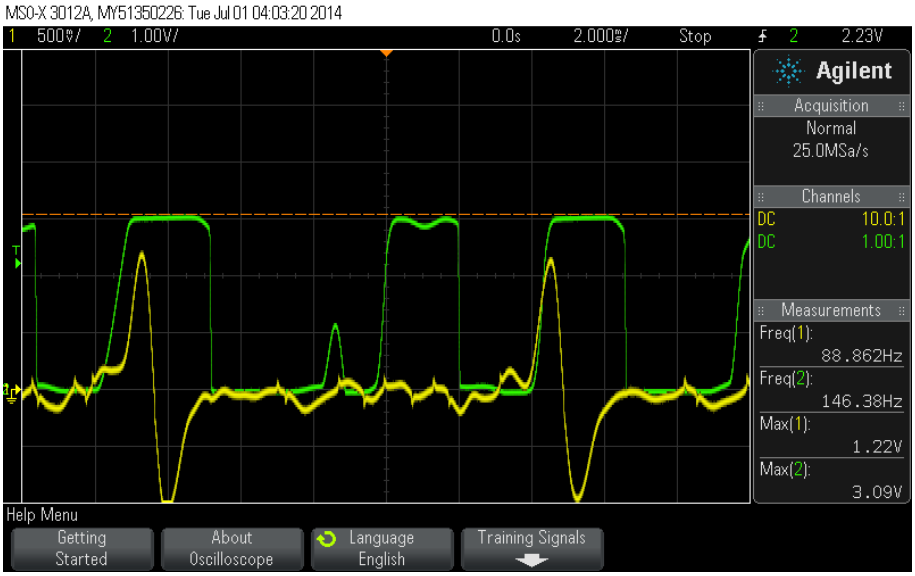


Figure 30: Glitches and Irregularity

The green signal represents the filtered signal and the yellow one represents the input. Figure 30 shows how irregular the signals are compared to the signals out of the waveform generator in figure 31:

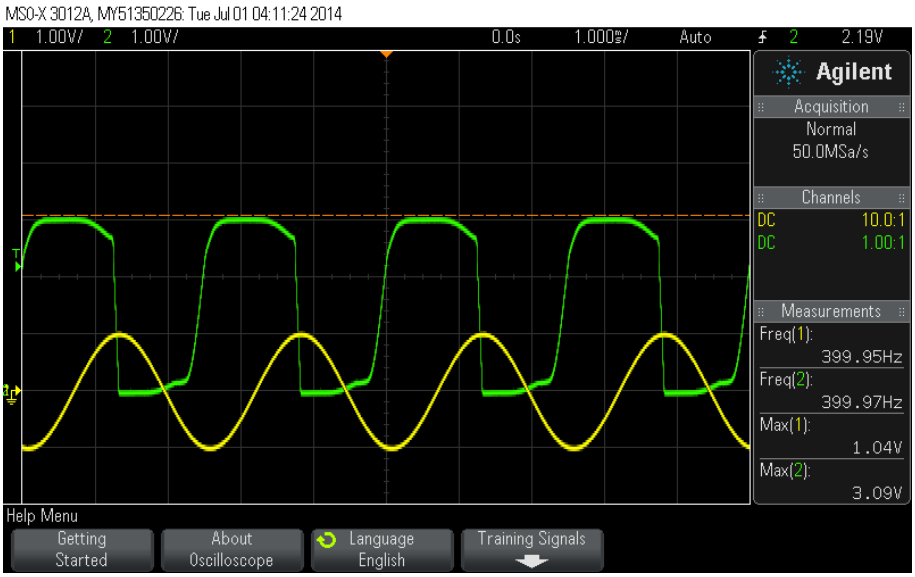


Figure 31

The peaks and glitches in figure 30 make the algorithm return incor-

rect result. Even if the hardware makes the signal clipping, the ADC detects more amplitude variations than a regular signal.

The problem lies in the fact that it is not secure enough to only have one trigger to detect the crossings. The 600[mV] middle range trigger was chosen because the input signal was conditioned to clip into the AD converter. As the signal is fully amplified, all the glitches and peaks are also amplified and appear clipping into the AD. This implies that the algorithm does not see a regular periodic signal and then gives fake results.

These two conditions show that the first algorithm using finite state machine is better than the second one concerning irregular signals. The two trigger values ensure not having to deal with the problem explained above.

11.2 Offset Problem

As noticed in practice and mentioned above, offsets appear when computing the frequency. This has to deal with the counter, the one that counts until it reaches the desired number of samples. The theoretical time to acquire 3000 samples using a 4.5[kHz] sampling frequency is:

$$t = 3000 \cdot \frac{1}{4500[Hz]} = 666.66[ms] \tag{12}$$

Measures performed using pins toggling show this time is actually bigger in practice. It corresponds to a time of 686 [ms] in average. This is 20 [ms] longer and terribly long.

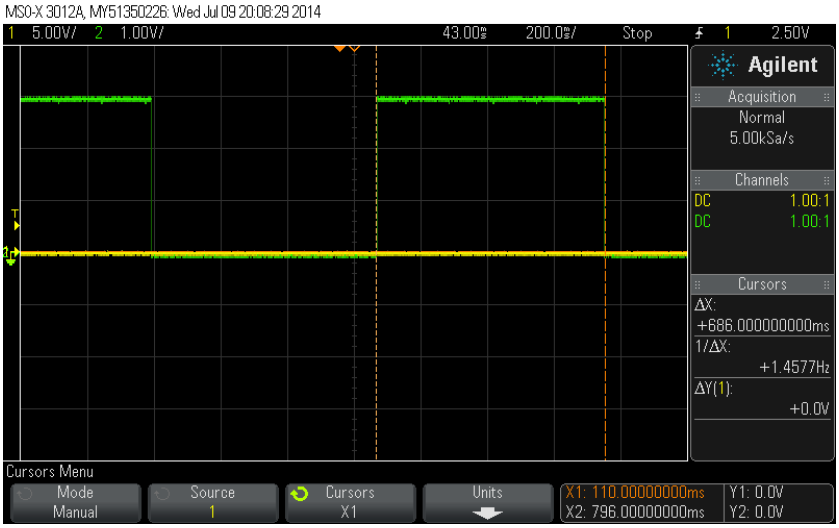


Figure 32: Samples Acquisition Time

To know what causes such time stretching, some reflexions and analysis are undertaken. Transmission time is very small, it takes a maximum time 4[μs] to transfer the data. Thus it is clearly not the transmission that causes the problem. The sampling frequency is measured to see if it could be a source of the problem. The measure shows the frequency is not equal to the desired one. Looking at the periods, the desired one is equal to 222.222[μs] and the measured one is equal to 221.3[μs] which corresponds to a frequency of 4.5188[kHz].

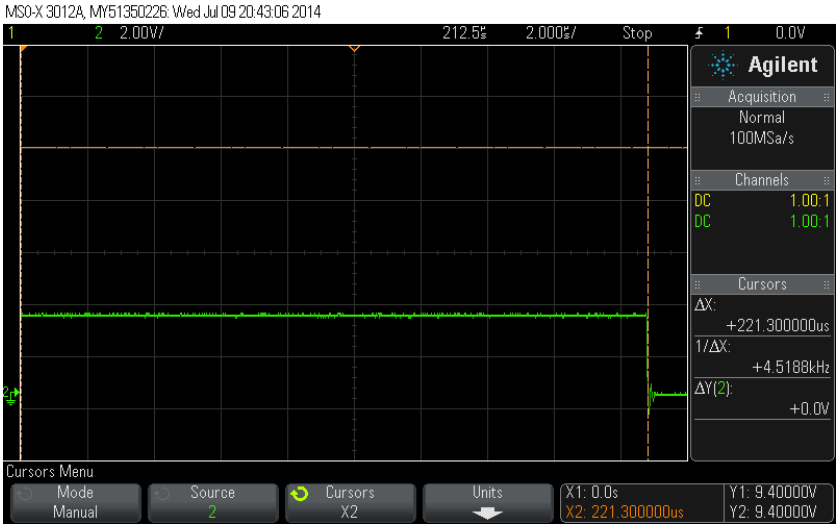


Figure 33: Sampling Frequency

The error is:

$$error = 100 - (\frac{4500}{4518.8} \cdot 100) = 0.416\% \tag{13}$$

Equation (13) highlights a small error between the frequencies but it could have bad repercussions if analyzed upon a long period of time. The way the frequency is computed can have consequences on the result whether it is computed on the nRF51822 or on the iOS device. The two devices are different in terms of hardware and compilers. Thus it is hard to know how the data is casted on each platform. The loss of the decimals can produce, as well, undeniable errors when computing .

Another thing to mention is that the algorithm was first performed into the interrupt handler of the AD conversion. As it is better to go as fast as possible out of the interrupt, the algorithm was performed into the main function, using *extern* variables. Implementation outside the interrupt reveals better results than previously presented but still have reduced offset.

Despite the lack of time, the performed tests have allowed to better target the source of the problem and suggest it comes from the implementation of the algorithm itself and not from the nRF51822.

12 Improvements

12.1 Hardware

The main improvement to be undertaken is about energy consumption. The nRF51822 chip is made for low power applications so the whole systems needs to consume less power for a better battery life. This can be improvement by choosing bigger resistors values and low consumption operational amplifiers.

Improving the second prototype would be necessary because the few tests done with it reveal the signals need to be amplified further. The adaptive counting algorithm works difficulty. The amplification has to better fit the ADC scale but without clipping. This could be done by regulation or in software.
Another problem to be resolved concerns the shunt regulator which is supposed to give a regular voltage reference for the 600[mV] AD middle range value. This regulator, and this for both prototypes, does not work at all even for a low power supply.

12.2 Software

A few things can be improved concerning the softwares (KEIL and iOS):

1. Dynamic Schmitt Trigger:

   Instead of using only one trigger for the crossing detection, using a dynamic Schmitt-Trigger would be a better solution. Indeed, as the signals are not regular enough in terms of amplitude, having a gap of values is safer in way that it keeps away the effects of glitches. Obviously the signal will need to be conditioned to better fit the AD scale.

2. Parallelism in iOS application:

   The tuning algorithm is performed each time a new frequency is computed and lies into the didUpdateValueForCharacteristic method. This implies sequential behavior. Creating a class specialized into the tuning would be better especially because if this class inherit from the *NSObject* class, its object would be a thread itself, implying parallelism.

13 Conclusion

This diploma thesis has made it possible to develop a prototype of a wireless tuner based on research made during the semester project and actual thesis. Although the prototype does not work perfectly and is not completely successful as regards tuning, this project has allowed to demonstrate the technical problems related to the processing and analysis of audio signals. It was also interesting to see how signals from an electric guitar behave using the two types of pickups (active and passive).

An algorithm for detecting the frequency of a signal has been implemented without using FFT. Although FFTs contain more informations about the signal, the algorithm has proved effective and fast behavior compared to the samples transmission option. The transfer of data also revealed the limits of low power transmission. Indeed, during the transfer of samples to the second device, the transmission is not strong enough to carry a significant number of incoming data (converted samples) implying data loss. Hence the need for a dow nsampling in such cases.

The nRF51822 is designed for systems with low energy consumption but is not really used for this purpose because the system consumes too much power. The project focused instead on the nRF51822 Analog/Digital converter and also focused on its limits and performances.

Some results are still not completely demonstrated, particularly with the offsets when calculating frequencies. The problem is not the hardware but the software certainly. The implementation of the algorithm is probably one of the causes of this problem. The accuracy of the timer that triggers the conversion is not 100% accurate, which can lead to significant errors during long periods of time. To better target the source of the problem should be analyzed in more detail what happens between two AD conversions.

However, the application globally does work great with generated signals. The iOS tuning application can display the frequency as well as the current note without any problems. It is not the case with the guitar signals since they are too irregular and not conditioned in the way the should be. Once the conditioning and the offset problems are resolved, the application will work without any issues.

References

1 Aurélien Alexandre Merz. Bluetooth Low Energy & Algorithms. Semester Work, University of Applied Sciences of Western Switzerland, Sion, May 2014.

2 nRF51 Manual References v1.1, Nordic Semiconductor, March 2013.

3 Wikipedia. Data Compression. `http://en.wikipedia.org/wiki/Data_compression`.

4 Wikipedia. Run-Lenght Encoding. `http://en.wikipedia.org/wiki/Run-length_encoding`

5 Wikipedia. Downsampling. `http://en.wikipedia.org/wiki/Downsampling`

6 Wikipedia. Cent et Savart. `http://fr.wikipedia.org/wiki/Cent_et_savart`

7 Apple Developpers. `https://developer.apple.com`

List of Figures

List of Tables

Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Filière Systèmes industriels

Infotronics

Projet de semestre 2014

Aurélien Merz

Low Energy Bluetooth & Algorithms

Professeur Medard Rieder

Sion, le 5 mai 2014

HES-SO Valais-Wallis • rte du Rawyl 47 • C.P. • 1950 Sion 2
+41 27 606 85 11 • info@hevs.ch • www.hevs.ch

Contents

1 Introduction

The goal of this semester project consists in finding out if the Nordic Semiconductor nRF51822 processor is capable to compute complex algorithms for signal processes such as Fast Fourier Transform (FFT) and also determine if the *Bluetooth Low Energy (BTLE)* transmission bandwidth is big enough to send relatively big amount of datas to another devices.
All these researches are aimed to lately build a devices that will tune an electric guitar.

2 Electric guitar: How it works

What's the big difference between acoustic and electric guitars? Electric guitars don't produce as much sound as an acoustic guitars but they produce another physical phenomena: Electricity. Electric guitars are built with elements called pickups that are going to transform mechanical vibrations into electricity.
Pickups are essentially made of magnets and coils.The physical process has to deal with Electro-magnetic field. As strings are made of steel, and as they come back and forth (oscillatory movement), they interact with the magnetic field produced by the magnets. This create variations of the field and so a current is induced into the coils. This voltage varies from - U [mV] and + U [mV] at a frequency corresponding to the string vibration.

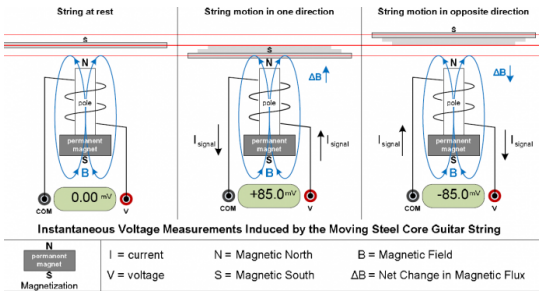


Figure 1: Basic Electric Guitar Pickups

The induced current then goes through the jack directly to an amplifier that will boost the signal. Then the signal is transmitted to the speakers to

create the sound.[4]

2.1 Sounds & Frequencies

Sounds are physical waves supported by the air. Human ears can detect a certain range of this sounds that goes from 20[Hz] to 20[kHz]. Here are the fundamental frequencies produced by the strings vibrations from a guitar and a bass guitar. The main frequency reference is 440[Hz] produced by the A note localized on the fourth octave of a piano keyboard [8].

Guitar Frequencies Range			
Note	Frequency [Hz]	Piano	String Number
E	82.41	E2	6
A	110.0	A2	5
D	146.83	D3	4
G	196.00	G3	3
B	246.94	B3	2
E	329.63	E4	1

Table 1: Guitar Frequencies Chart

Bass Frequencies Range				
Note	Frequency [Hz]	4 Strings	5 Strings	6 Strings
B0	30.868		5	6
E1	41.204	4	4	5
A1	55	3	3	4
D2	73.416	2	2	3
G2	98	1	1	2
C3	130.813			1

Table 2: Bass Frequencies Chart

3 BTLE Communication Protocol

Bluetooth Low Energy (BTLE) is a specification of *Bluetooth 4.0* in the way that it intends to reduce the power consumption during connections and data transfers. Nowadays most of mobile operating systems such as *iOS*, *Android* or *Windows Phone* integrate BTLE.

BTLE devices use Generic Attribute (GATT). The battery life time is highly longer than normal Bluetooth.

3.1 Centrals & Peripherals

Centrals and peripherals communication is based on a client-server architecture. Peripherals are devices that have datas needed by other devices. Centrals use the datas provided by the peripherals to achieve tasks.

3.1.1 Discover and Connection

Peripherals broadcast small bundles of datas representing advertising packets. These small bundles contain informations about what the peripheral has to provide, such as services or peripheral's name. This is a particularity of Bluetooth Low Energy. Advertising is the way peripherals show they are present.

A central is a device that listen to peripheral's advertising packet. I can chose to make a connection to the ones that may interest it. When they connect, a peripheral and central form what we call a piconet. The central become the Master and the peripheral the Slave.

3.2 Bluetooth Low Energy Data Structure

Datas needed by a central are structure in a certain way. Datas are structured into a shape we call Services.

3.2.1 Services & Characteristics

Services are collection of characteristics. Characteristics are themselves datas providing further informations about services. After the connection is made, the central can read or even write values of a service's characteristics.

3.2.2 UUID

Centrals discover a certain range of services after making the connection. They use UUIDs to know which services they're interacting with. UUID stands for Universally Unique Identifier. UUIDs are 128 bits coded value that identifiers services. A reduced 16 or 32 bits value is normally used because a range of these bits have been set by Bluetooth Special Interest Group. The Bluetooth UUDIs are of the form xxxxxxxx-0000-1000-8000-00805F9B34FB, with x representing the effective ID. [5]

4 Researches & Equipment

Here are listed the hardware component and software tools used for developing the prototype.[2]

4.1 Hardware

1. Nordic Semiconductor nRF51822 with:
   - ARM® Cortex™-M0 32 bit processor, single-cycle multiplier, 3-stage pipeline,
   - 16 [MHz] Clock Frequency,
   - Memory: 256 kB or128 kB embedded flash program, 16 kB RAM,
   - 8/9/10 bit ADC - 8 configurable channels,
   - Supply voltage range 1.8 V to 3.6 V,
   - S100 series SoftDevice ready
2. Nordic Semiconductor nRF51822 Evaluation Kit with:
   - Seggers Debug Chip
3. Apple iPhone 4S, 32GB Memory

4.2 Software

Here are listed the software used for development:

1. Nordic Semiconductor nRF51822 with:
   - nRFgo Studio,
   - nRF51 Software Development Kit (SDK),
   - Keil ARM project files,
   - S110 nRF51822 SoftDevice,
   - S110 SoftDevice programming tools
   - $\mu Vision4$
2. Apple XCode 5.1 SDK
3. LightBlue iPhone Application

4.3 Signal Processing & Researches

Fast Fourier Transform (FFT) is an algorithm used to compute the discrete Fourier transform (DFT) of an analog signal. The algorithm transforms discrete data from the time domain into the frequency domain. [7]

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \ , \ k = 0,...,\text{N-1} \tag{1}$$

It could assumed that only taking the fundamental frequencies would be enough to make a sufficient tuning but to make a better measure and with better precision, harmonics are necessary. One relatively critical point is the sampling frequency of the Analog/Digital Converter (ADC) because it will determine the frequency resolution of the FFT but also will determine how to build an efficient analog anti-aliasing filter. In fact the more the sampling frequency is big, the better because the transfer function would be much easier to built with analog component.
What is exactly the frequency resolution? It is the report between the sampling frequency and the considered number of samples while computing the FFT.

$$\triangle f = \frac{f_s}{N} \ , \ \begin{cases} f_s &= \text{sampling frequency} \\ N &= \text{number of samples acquired} \end{cases} \tag{2}$$

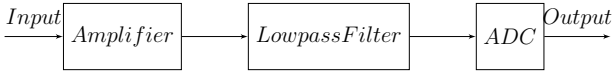
$\triangle f$ is solely determined by the acquisition time. Improves as the acquisition time increases. A big sampling frequency allows analyzing signal that have higher frequency. The FFT number of sample N is and has to be an 2^n value.

5 Analysis

This section shows the different steps and conditions in order to make a correct system for the application. Indeed, the system has to have a certain of important component such as an amplifier and a lowpass filter.

5.1 Conditioning

This figure shows where the input signal will pass before being sampled and prepared for a FFT:



5.1.1 Amplification

As the voltage range coming from the guitar coils is in [mV], it needs to be amplified. The amplification needs to fit the ADC voltage range in order to measure relevant values. The nRF51822 allows different voltage prescaling references and inputs such as:

1. Inputs Voltage Prescaling:
   - $\frac{1}{3} \cdot Input$,
   - $\frac{2}{3} \cdot Input$,
   - $1 \cdot Input$,
2. Voltage References Prescaling:
   - $\frac{1}{3} \cdot VDD$,
   - $\frac{1}{2} \cdot VDD$,
   - External references (AREF0 or AREF1),
   - VBG, Internal 1.2 V Reference

In the testing phase, the $\frac{1}{3}$ prescaling and the VBG reference were used. There are different classes of electronic amplifier which have their proper outputs signal but the dimensioning isn't discussed here.

5.1.2 Filtering

After being amplified, the signal needs to be filtered by an anti-aliasing filter. This is necessary in term of after-sampling signal quality. This filter is dependent of the ADC sampling frequency.

$$f_c < \frac{f_s}{2} \text{ , } f_s \text{ sampling frequency} \tag{3}$$

The nRF51822 provides three AD conversion modes with their corresponding time to convert a single sample [2] :

- 10 bit mode: $t_{10bit} = 68\mu s \Rightarrow f_{10bit} = 14.7$[kHz]
- 9 bit mode: $t_{9bit} = 36\mu s \Rightarrow f_{9bit} = 27.77$[kHz]
- 8 bit mode: $t_{8bit} = 20\mu s \Rightarrow f_{8bit} = 50$[kHz]

Here are the 3 possible cut frequencies for the lowpass filter:

f_c	Frequency[kHz]
$f_{c,10bit}$	7.35
$f_{c,9bit}$	13.88
$f_{c,8bit}$	25

Table 3: Lowpass Filter Cut Frequencies

One of the aspect to consider is the filter capacity to attenuate the signal at the frequency $\frac{f_s}{2}$. This attenuation has to be "sufficient", in another words, it depends on the application constraints. For instance, an 80 dB attenuation is considered as sufficient for phone audio signals but insufficient for a Hi-Fi quality.

Using a Butterworth filter could be a good option because of its linear phase and its non signal distortion. But in this case, only the amplitude matters. Another way would be to use Elliptic filters because they allow to realize the smallest filter order for a certain transition speed in the bandwidth and the cut bandwidth.

5.2 Sampling Frequency

The nRF51822 sampling frequency from the ADC is a relatively critical point. In fact, it will determine the order of the anti-aliasing filter and also will set the frequency resolution of the FFT:

$$\triangle f = \frac{f_s}{N} \tag{4}$$

Here are the possible frequency resolutions using the nRF51822 ADC:

Resolution	N = 256	N = 512	N = 1024	N = 2048
$\triangle f_{10bit}$	57.42[Hz]	28.71[Hz]	14.35[Hz]	7.17[Hz]
$\triangle f_{9bit}$	108.47[Hz]	54.23[Hz]	27.11[Hz]	13.55[Hz]
$\triangle f_{8bit}$	195.313[Hz]	97.65[Hz]	48.82[Hz]	24.41[Hz]

Table 4: Frequency Resolutions

The lowest frequency to analyze is 30.868[Hz] and the highest is 329.63[Hz]. These frequencies correspond to piano keys B0 and E4 respectively. Shannon's rule says:

$$f_s > 2 \cdot f_{max} \Leftrightarrow f_s > 659.26 \tag{5}$$

In this case, the ADC sampler is suitable for the application because the nRF51 lowest sampling frequency is 14.7[kHz] [2].

5.3 Data Transmission through BTLE

Bluetooth low energy is a whole new approach regarding wireless technologies. Original Bluetooth was created for continuous streaming applications like transmitting voice and made its place into the market. Even if the BTLE is limited for some applications, it has some new benefits (low energy consumption) and is used in severals domain like medical for example.
Another solution to our problem is to transmit the sampled values by the BTLE radio transmission. Datas are transferred to a remote device that would compute the FFT. BTLE cannot stream information as the other Bluetooth protocols and a frame can hold 20 bytes of data. So the goal is to pack the maximum of values coming from the ADC into the data frame. This data frame will be seen as a characteristic to other devices. See annexe 2 for flowchart.
As the conversion result is 10 bits long, only the first two LSB are necessary. This way allows the array to contain 10 measured values. When the remote device receives the 20 bytes, it will have to cast each measures into a float variable and fill the MSB with 0x00 to be used by the FFT.
A good optimization would be to check if the last 8 bits of the MSB are equal to 0. In that case, we could add only one byte into the array. This solution would add a bit of complexity when analyzing the array because it's

hard to know if the i^{th} element is only a one byte value or if the $i^{th}-1$ is part of this value. Another optimization would be to use a special ARM library for compressing data without losing information.

6 Results & Discussions

The best way to know how much time it takes to complete the algorithm is to measure. In the way to choose the best option for a good FFT, it is necessary to know which options we have using the nRF51822 ADC.

6.1 FFT Computing Time Results

A first small program was implemented to see how much time would take the algorithm to compute a complete FFT with different numbers of samples. The system crashes for N higher than 1024 samples. It goes into the HardFault Handler, meaning that the memory capacity has been overloaded. Flowchart is provided in annexes 1 for better visualization.

In practice, the test was realized with a 440[Hz] sine wave, with an amplitude of 85[mV_{pp}] and 600[mV] offset to fit to the ADC voltage reference (1.2 [V]):

FFT Computing Time	
Nb of Samples N	Time t [ms]
256	81
512	177.5
1024	377

Table 5: FFT Computing Time in debug mode

These results come from a non-optimized FFT algorithm coming from a scientific computing book [9]. A second test was implemented using a free DSP library for Cortex M0. This library was provided by Cortex Microcontroller Software Interface Standard (CMSIS) [6]. Here are the new results:

FFT Computing Time	
Nb of Samples N	Time t [ms]
128	9.2
256	21.5
512	47.5

Table 6: CMSIS Optimized FFT Computing Time

Memory is overloaded for a number of samples higher than 512 but computing time decreases undeniably.
FFT computing time have also been measured on the iPhone 4S: It takes in average 1.3[ms] for the iPhone to compute a 1024 samples FFT.

6.2 Data Transmission Results

As implemented in the code, the measures are performed continuously in the main and since the buffer containing these measures is full, the chip sends it. The nRF51822 works under the $Notify$ state meaning that it sends the data as soon as the characteristic is updated. Transmitting 1024 samples from the nRF51822 to the iPhone takes in average 535[ms]. This time is doubled for each 2^N samples. In this interval of time are performed a certain number of actions such as:

- starting AD conversion,
- filling the array,
- send the new characteristic,
- power management

Flowchart is provided in annexes.

As the processor makes all of these actions, it can't measure the whole input range so it misses some points. After collecting a hundred of samples, plot and FFT of the discrete signal were computed to see how it looked like [3]:

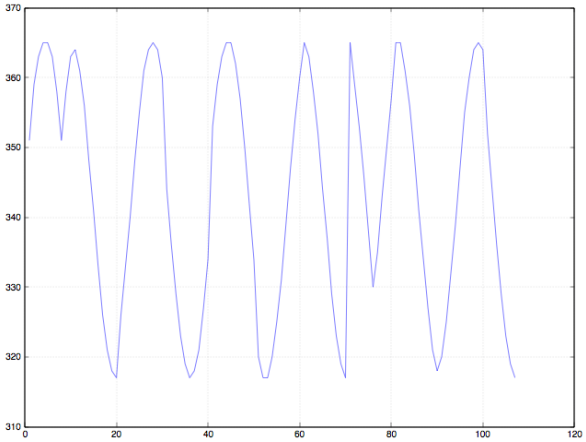


Figure 2: Discrete Signal in Time Domain

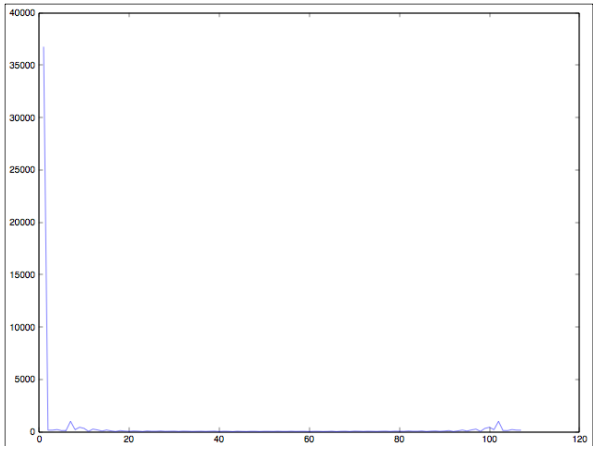


Figure 3: Discrete Signal FFT

In the Figure 2, X-Axis represent time in [ms] and Y-Axis represents amplitude value. In Figure 3, X-Axis represents frequencies in Hertz and Y-Axis the amplitude value.

It can be seen on Figure 2 that the sampled signal isn't regular at all. Amplitude jumps are present because the CPU cannot manage getting new samples while sending the previous ones over the BTLE connection so it misses measures as previously said.

Optimization and improvements are explained in the next section.

6.3 Discussions & Improvements

At this point of the project, it's necessary to make a bilan and see what are the advantages and disadvantage of using the nRF51822 for signal processing as well as showing the improvements.

6.3.1 Sampling & Computing

Here are listed the advantages and disadvantages for signal processing part. Only the 10-bit and 8-bit conversion are discussed because they are the extremes.

Conversion	Advantages	Disadvantages
10-bit	Highest precision for samples	Harder to realize analog filter because of the attenuation slope
8-bit	1 byte = 1 sample, perfect for sending multiple samples over BTLE transmission (20 samples/frame).Easier way to build analog filter, faster sampling (higher than HQ 48[kHz])	Non sufficient precision, 8 bit is not enough for a good analysis

The choice of the conversion depends on how much precise the application has to be: In this case, the 10-bit option is taken.

As the Nordic nRF51822 is built on a Cortex-M0 architecture, it's quiet limited by its computing capacity. In fact, FFT requires a lot of mathematical operations depending on the number of samples that Cortex M3 or M4 would better fit the application. The nRF51 memory also limits the application: A float array of 2^N samples requires a lot of memory as well as a second one needed to hold the complex numbers (result of FFT).

6.3.2 Transmitting Samples

Transmission is part of the critical points concerning real-time. A frame is sent each time the characteristic is updated so the time could increase depending on how the code is written. Another thing to mention is that BTLE can't stream data and this case makes the application unsuitable for real-time.
A way to improve the transmission could be to make data compression using ARM optimized algorithms. But this won't solve the real-time problem because compression takes time as well as collecting enough measures.

6.3.3 Improvements

As the project is in development, some little issues were noticed such as lack of memory and non regular discrete transmitted signal. So a few improvements need to be brought in order to provide better performances:

1. Nordic Semiconductors New Chip Generation:

   Concerning the chip, the new generation of Nordic Semiconductor nRF82 will be built on Cortex-M3 architecture and softdevices(stack) will allow simultaneous transmission and computing. Using these new chips would be a great solution in the way they will increase the performances.

2. Using Independent Timer for Sampling:

   The last day of the project, a relevant point was mentioned: Depending on the conversion mode, it takes a certain time to make an AD conversion. As mentioned previously, the main loop makes these listed operations:

   - starting AD conversion,
   - filling the array,
   - send the new characteristic / compute FFT,
   - power management

   Each and every operations listed above takes a certain time to make what they have to do. So this means that the sampling frequency isn't controlled because the conversion is only performed after the other tasks (filling array, sending, etc) are done. Indeed, for a 10 bit conversion, the time between 2 samples won't be the ideal wanted $68\mu s$ (see section 5.1.2) but will be:

   $$t = t_{sampling} + t_{fillingArray} + t_{sending/FFT} \tag{6}$$

   This influences the quality of the signal in a undeniable way.
   So the solution to that problem is to work with timers. The nRF51 series provide a system called Programmable Peripheral Interconnect (PPI) that enables different peripherals to interact autonomously with each other using tasks and events and without having to use the CPU. The mechanism behind this is that tasks in a certain peripheral are automatically triggered as a response to an event coming from another peripheral. These tasks listen to a PPI channel in order to get the events they are connected to. In this way, a timer based on the nRF main clock (HFCLK = 16MHz) will count the desired sampling period (with a certain margin) and will trigger the ADC conversion START event when it comes to overflow. Reference to annexe 3.
   The timer frequency is computed with the following formula:

   $$f_{TIMER} = \frac{HFCLK}{2^{PRESCALER}} \tag{7}$$

   This solution will make the sampling frequency completely controlled and this without using the CPU. [1]

3. Using a READ & a WRITE buffers:

   Instead of filling only one array of samples, wait till it's full and finally send it, using a second array for the transmission would improve performances in the way that while transmitting an array, a second one can be filled with new datas. This could be done with the new nRF generations.

4. Compressing Data:

   As the samples are quite greedy in terms of memory, a good improvement would be to use an ARM optimized library for compressing data without losing information. In that way, it could be possible to send a lot more values in order to improve real-time.

7 Conclusion

This semester project demonstrated how the nRF51822 is capable to deal with digital signal processing. The researches showed that it is necessary to make compromises in terms of measures, precision and real-time behavior:

- Less samples: Faster transmission but weak FFT,
- Many samples: Slower transmission but better FFT,

The way the nRF51822 can't make simultaneous transmission and computation makes it limited for the application as well as the BTLE protocol can't allow data streaming like the original Bluetooth. This project gave me a better understanding of signal processing, how Fast Fourier Transforms really work and what are their memory cost. I also saw how much the performances increase when using optimized mathematical algorithms for a given processor. The nRF51822 is a great deal for low energy consumption but in terms of performance, especially for digital signal processing, it's not the best solution. Using the next generation of nRF built on Cortex-M3 architecture will make sens for this type of application.

References

[1] *nRF51 Manual References v1.1.* Nordic Semiconductor, March 2013.

[2] *nRF51822 Product Specification v1.1.* Nordic Semiconductor, March 2013.

[3] GNU Octave. GNU Octave. `https://www.gnu.org/software/octave/`.

[4] Kurt Prange. Guitar Pickups Explained. `http://www.guitarsite.com/news/features/Basic-Electric-Guitar-Circuits-Pickups/`.

[5] Alexandre Sierro. *Bluetooth Low Energy.* PhD thesis, University of Applied Sciences of Western Switzerland, Sion, July 2012.

[6] Cortex Microcontroller Software Interface Standard. CMSIS DSP Software Library. `http://www.keil.com/pack/doc/cmsis/DSP/html/index.html`.

[7] Wikipedia. Discrete-time Fourier transform. `http://en.wikipedia.org/wiki/Discrete-time_Fourier_transform`.

[8] Wikipedia. Guitar Tunings. `http://en.wikipedia.org/wiki/Guitar_tuning`.

[9] William T. Vetterling Brian P. Flannery William H. Press, Saul A. Teukolsky. *Numerical Recipes in C*, volume 2. Cambridge University Press, 2 edition, February 2002.

Industrial Systems
Infotronics

Diploma Thesis
2014

Aurélien Alexandre Merz

ATTACHEMENTS

July 11, 2014

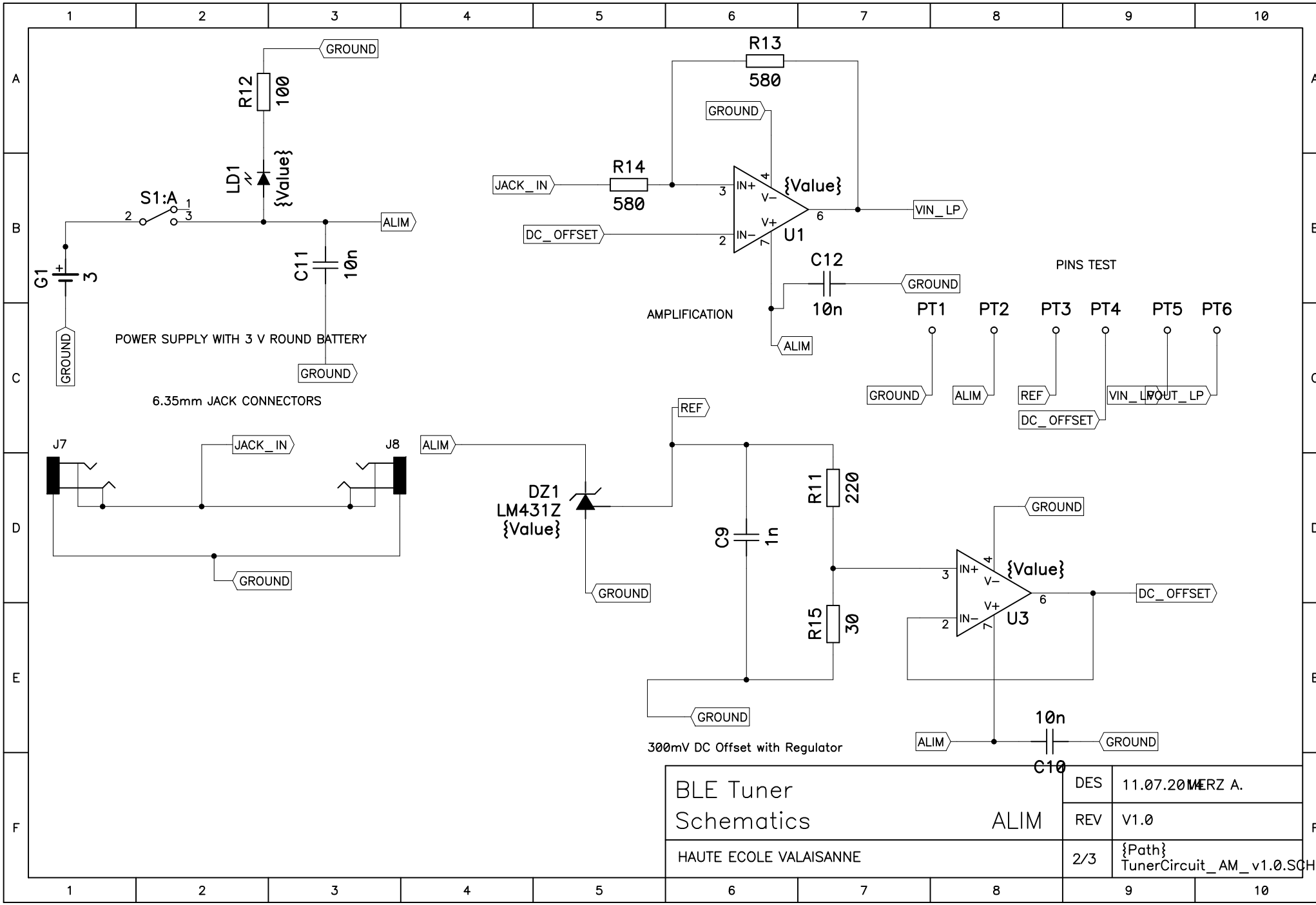
1 ATTACHEMENT A: UART MATLAB SCRIPT

```
%------------------------
%-- UART MATLAB SCRIPT  --
%------------------------

Fs = 4000;
L = 300;
N = dlmread('samplesDEC4k.txt')
M = dlmread('1022DEC.txt');
E = dlmread('253DEC.txt');

%figure(1)
%plot(M)

x_fft = fft(M);
m_fft = fft(N);
e_fft = fft(E);

%figure(2)

[maxValue,indexMax] = max(abs(fft(M-mean(M))));
[maxValue1,indexMax1] = max(abs(fft(N-mean(N))));
[maxValue2,indexMax2] = max(abs(fft(E-mean(E))));

%COMPUTING THE FREQUENCIES

frequency_test400 = indexMax1 * Fs / L
frequency_test253 = indexMax2 * Fs / L
frequency_test1022 = indexMax * Fs / L


%plot(abs(x_fft))
```

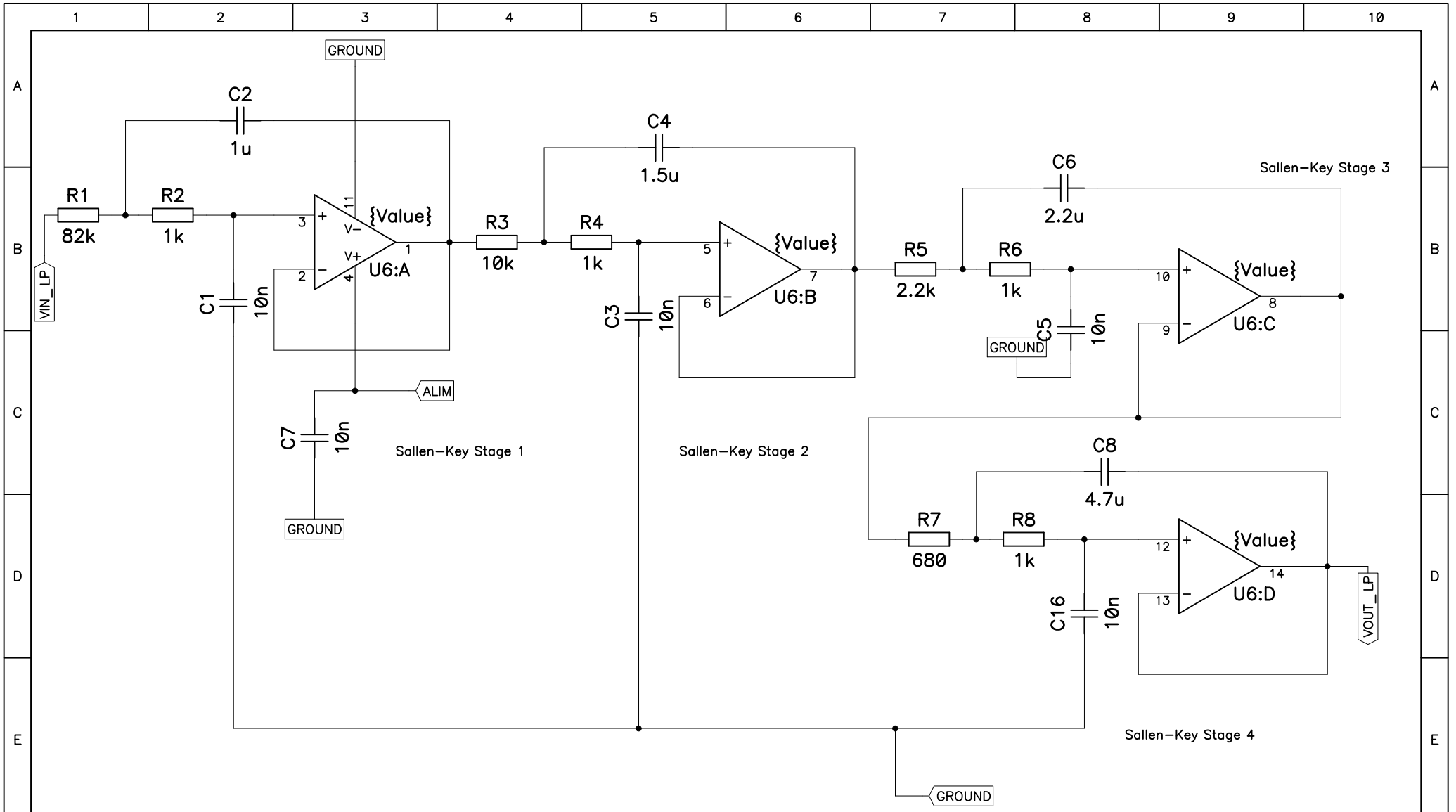
2 ATTACHEMENT B: ADAPTIVE COUNTING ALGORITHM SCRIPT

```
%-------------------------------------------------------
%-- FREQUENCY DETECTION BY ADAPTIVE COUNTING ALGORITHM  --
%-------------------------------------------------------
Fs = 4000;
L  = 300;

%-- FILES CONTAINING THE DATA POINTS  --

M  = dlmread('253DEC.txt');
M2 = dlmread('samplesDEC4k.txt');
M3 = dlmread('1022DEC.txt');

%-- ACTUAL AND PREVIOUS DATA POINT VALUES --

x  = 0;
x2 = 0;
x3 = 0;
x_old = 0;
x_old2 = 0;
x_old3 = 0;

%-- AVERAGE OF THE DATA POINTS  --

avg = 0;
avg2 = 0;
avg3 = 0;
avg21 = 0;
avg22 = 0;
avg23 = 0;

%-- NUMBER OF TIME THE SIGNAL PASSES THE TRIGGER FROM UP TO DOWN  --

pass = 0;
pass2 = 0;
pass3 = 0;

i = 1;
j = 1;
t_total = 0;
for i = 1: 1

    for j = 1:300

        x_old  = x;
       % x_old2 = x2;
       % x_old3 = x3;

        x  = M(i,j);

       % x2 = M2(i,j);
       % x3 = M3(i,j);

        avg = (0.85*avg + 0.15*x);

        %figure(2)


        avg2 = (0.50*avg2 + 0.50*avg);

        figure(3)
```

3 ATTACHEMENT C: SCHEMATICS

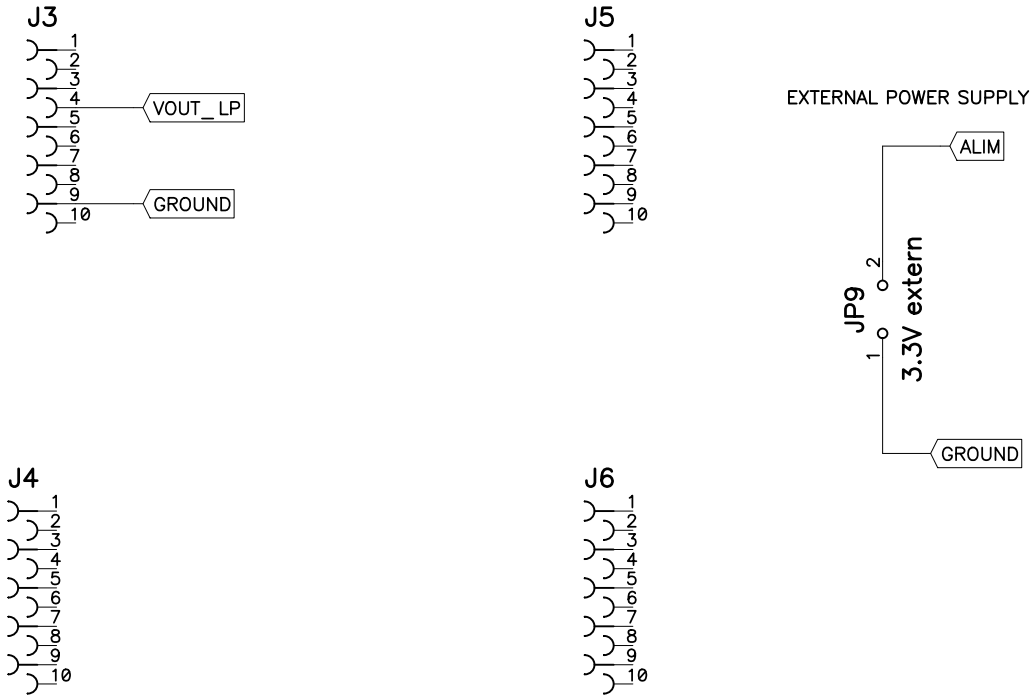


BLE Tuner Schematics — ALIM	DES	11.07.20MERZ A.
	REV	V1.0
HAUTE ECOLE VALAISANNE	2/3	{Path} TunerCircuit_AM_v1.0.SCH

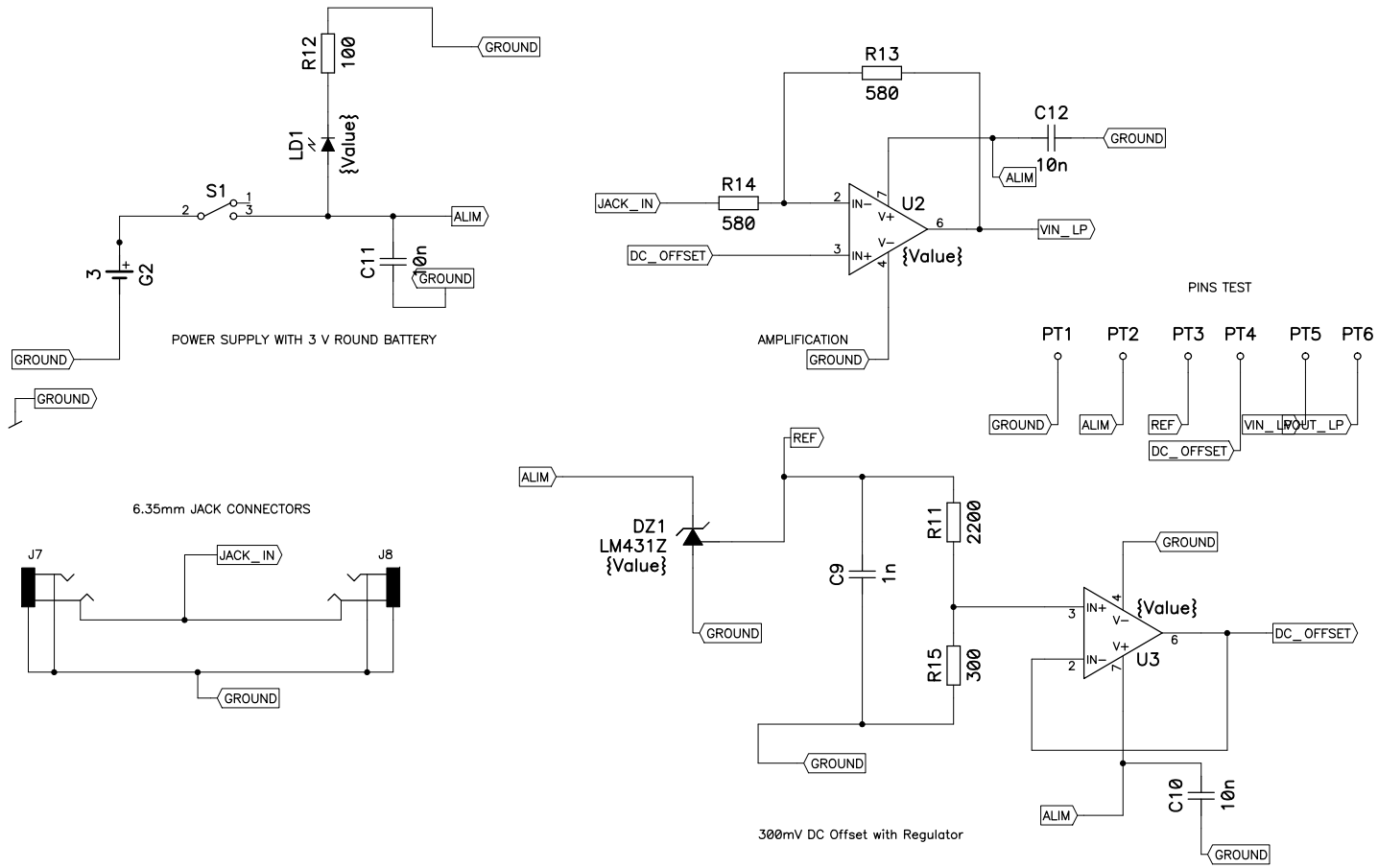


BLE Tuner Schematics — Lowpass	DES	11.07.2014 Aurélien Merz
	REV	V1.0
HAUTE ECOLE VALAISANNE	1/3	{Path} TunerCircuit_AM_v1.0.SCH

GPIO Headers



BLE Tuner Schematics nRFConnection	DES	11.07.2014 MERZ A.
	REV	V1.0
HAUTE ECOLE VALAISANNE	3/3	{Path} TunerCircuit_AM_v1.0.SCH



BLE Tuner		DES	11.07.2014 MERZ A.
Schematics	Sheet1	REV	v1.1
HAUTE ECOLE VALAISANNE		1/1	{Path} bite.sch

4 ATTACHEMENT D: TEXAS INSTRUMENTS FILTER DESIGN

Active Low-Pass Filter Design

Jim Karki *AAP Precision Analog*

ABSTRACT

This report focuses on active low-pass filter design using operational amplifiers. Low-pass filters are commonly used to implement antialias filters in data-acquisition systems. Design of second-order filters is the main topic of consideration.

Filter tables are developed to simplify circuit design based on the idea of cascading lower-order stages to realize higher-order filters. The tables contain scaling factors for the corner frequency and the required Q of each of the stages for the particular filter being designed. This enables the designer to go straight to the calculations of the circuit-component values required.

To illustrate an actual circuit implementation, six circuits, separated into three types of filters (Bessel, Butterworth, and Chebyshev) and two filter configurations (Sallen-Key and MFB), are built using a TLV2772 operational amplifier. Lab test data presented shows their performance. Limiting factors in the high-frequency performance of the filters are also examined.

Contents

List of Figures

List of Tables

1 Introduction

There are many books that provide information on popular filter types like the Butterworth, Bessel, and Chebyshev filters, just to name a few. This paper will examine how to implement these three types of filters.

We will examine the mathematics used to transform standard filter-table data into the transfer functions required to build filter circuits. Using the same method, filter tables are developed that enable the designer to go straight to the calculation of the required circuit-component values. Actual filter implementation is shown for two circuit topologies: the Sallen-Key and the Multiple Feedback (MFB). The Sallen-Key circuit is sometimes referred to as a voltage-controlled voltage source, or VCVS, from a popular type of analysis used.

It is common practice to refer to a circuit as a Butterworth filter or a Bessel filter because its transfer function has the same coefficients as the Butterworth or the Bessel polynomial. It is also common practice to refer to the MFB or Sallen-Key circuits as filters. The difference is that the Butterworth filter defines a transfer function that can be realized by many different circuit topologies (both active and passive), while the MFB or Sallen-Key circuit defines an architecture or a circuit topology that can be used to realize various second-order transfer functions.

The choice of circuit topology depends on performance requirements. The MFB is generally preferred because it has better sensitivity to component variations and better high-frequency behavior. The unity-gain Sallen-Key inherently has the best gain accuracy because its gain is not dependent on component values.

2 Filter Characteristics

If an ideal low-pass filter existed, it would completely eliminate signals above the cutoff frequency, and perfectly pass signals below the cutoff frequency. In real filters, various trade-offs are made to get optimum performance for a given application.

Butterworth filters are termed maximally-flat-magnitude-response filters, optimized for gain flatness in the pass-band. the attenuation is –3 dB at the cutoff frequency. Above the cutoff frequency the attenuation is –20 dB/decade/order. The transient response of a Butterworth filter to a pulse input shows moderate overshoot and ringing.

Bessel filters are optimized for maximally-flat time delay (or constant-group delay). This means that they have linear phase response and excellent transient response to a pulse input. This comes at the expense of flatness in the pass-band and rate of rolloff. The cutoff frequency is defined as the –3-dB point.

Chebyshev filters are designed to have ripple in the pass-band, but steeper rolloff after the cutoff frequency. Cutoff frequency is defined as the frequency at which the response falls below the ripple band. For a given filter order, a steeper cutoff can be achieved by allowing more pass-band ripple. The transient response of a Chebyshev filter to a pulse input shows more overshoot and ringing than a Butterworth filter.

3 Second-Order Low-Pass Filter – Standard Form

The transfer function H_{LP} of a second-order low-pass filter can be express as a function of frequency (f) as shown in Equation 1. We shall use this as our standard form.

$$H_{LP}(f) = -\frac{K}{\left(\frac{f}{FSF \times fc}\right)^2 + \frac{1}{Q}\,\frac{jf}{FSF \times fc} + 1}$$

Equation 1. Second-Order Low-Pass Filter – Standard Form

In this equation, f is the frequency variable, fc is the cutoff frequency, FSF is the frequency scaling factor, and Q is the quality factor. Equation 1 has three regions of operation: below cutoff, in the area of cutoff, and above cutoff. For each area Equation 1 reduces to:

- $f << fc \Rightarrow H_{LP}(f) \approx K$ – the circuit passes signals multiplied by the gain factor K.
- $\frac{f}{fc} = FSF \Rightarrow H_{LP}(f) = -jKQ$ – signals are phase-shifted 90° and modified by the Q factor.
- $f >> fc \Rightarrow H_{LP}(f) \approx -K\left(\frac{FSF \times fc}{f}\right)^2$ – signals are phase-shifted 180° and attenuated by the square of the frequency ratio.

With attenuation at frequencies above fc increasing by a power of 2, the last formula describes a second-order low-pass filter.

The frequency scaling factor (FSF) is used to scale the cutoff frequency of the filter so that it follows the definitions given before.

4 Math Review

A second-order polynomial using the variable s can be given in two equivalent forms: the coefficient form: $s^2 + a_1 s + a_0$, or the factored form; $(s + z_1)(s + z_2)$ – that is: $P(s) = s^2 + a_1 s + a_0 = (s + z_1)(s + z_2)$. Where $-z_1$ and $-z_2$ are the locations in the s plane where the polynomial is zero.

The three filters being discussed here are all pole filters, meaning that their transfer functions contain all poles. The polynomial, which characterizes the filter's response, is used as the denominator of the filter's transfer function. The polynomial's zeroes are thus the filter's poles.

All even-order Butterworth, Bessel, or Chebyshev polynomials contain complex-zero pairs. This means that $z_1 = Re + Im$ and $z_2 = Re - Im$, where Re is the real part and Im is the imaginary part. A typical mathematical notation is to use z_1 to indicate the conjugate zero with the positive imaginary part and z_1^* to indicate the conjugate zero with the negative imaginary part. Odd-order filters have a real pole in addition to the complex-conjugate pairs.

Some filter books provide tables of the zeros of the polynomial which describes the filter, others provide the coefficients, and some provide both. Since the zeroes of the polynomial are the poles of the filter, some books use the term poles. Zeroes (or poles) are used with the factored form of the polynomial, and coefficients go with the coefficient form. No matter how the information is given, conversion between the two is a routine mathematical operation.

Expressing the transfer function of a filter in factored form makes it easy to quickly see the location of the poles. On the other hand, a second-order polynomial in coefficient form makes it easier to correlate the transfer function with circuit components. We will see this later when examining the filter-circuit topologies. Therefore, an engineer will typically want to use the factored form, but needs to scale and normalize the polynomial first.

Looking at the coefficient form of the second-order equation, it is seen that when $s << a_0$, the equation is dominated by a_0; when $s >> a_0$, s dominates. You might think of a_0 as being the break point where the equation transitions between dominant terms. To normalize and scale to other values, we divide each term by a_0 and divide the s terms by ωc. The result is:

$P(s) = \left(\frac{s}{\sqrt{a_0} \times \omega c}\right)^2 + \frac{a_1 s}{a_0 \times \omega c} + 1$. This scales and normalizes the polynomial so that the break point is at $s = \sqrt{a_0} \times \omega c$.

By making the substitutions $s = j2\pi f$, $\omega c = 2\pi fc$, $a1 = \frac{1}{Q}$, and $\sqrt{a_0} = FSF$, the equation becomes:

$P(f) = -\left(\frac{f}{FSF \times fc}\right)^2 + \frac{1}{Q}\,\frac{jf}{FSF \times fc} + 1$, which is the denominator of Equation 1– our standard form for low-pass filters.

Throughout the rest of this article, the substitution: $s = j2\pi f$ will be routinely used without explanation.

5 Examples

The following examples illustrate how to take standard filter-table information and process it into our standard form.

5.1 Second-Order Low-Pass Butterworth Filter

The Butterworth polynomial requires the least amount of work because the frequency-scaling factor is always equal to one.

From a filter-table listing for Butterworth, we can find the zeroes of the second-order Butterworth polynomial: $z_1 = -0.707 + j0.707$, $z_1^* = -0.707 - j0.707$, which are used with the factored form of the polynomial. Alternately, we find the coefficients of the polynomial: $a_0 = 1$, $a_1 = 1.414$. It can be easily confirmed that $(s + 0.707 + j0.707)(s+0.707-j0.707)=s^2+1.414s+1$.

To correlate with our standard form we use the coefficient form of the polynomial in the denominator of the transfer function. The realization of a second-order low-pass Butterworth filter is made by a circuit with the following transfer function:

$$H_{LP}(f) = \frac{K}{-\left(\frac{f}{fc}\right)^2 + 1.414\,\frac{jf}{fc} + 1}$$

Equation 2. Second-Order Low-Pass Butterworth Filter

This is the same as Equation 1 with FSF = 1 and $Q = \frac{1}{1.414} = 0.707$.

5.2 Second-Order Low-Pass Bessel Filter

Referring to a table listing the zeros of the second-order Bessel polynomial, we find: $z_1 = -1.103 + j0.6368$, $z_1^* = -1.103 - j0.6368$; a table of coefficients provides: $a_0 = 1.622$ and $a_1 = 2.206$.

Again, using the coefficient form lends itself to our standard form, so that the realization of a second-order low-pass Bessel filter is made by a circuit with the transfer function:

$$H_{LP}(f) = \frac{K}{-\left(\frac{f}{fc}\right)^2 + 2.206\,\frac{jf}{fc} + 1.622}$$

Equation 3. Second-Order Low-Pass Bessel Filter – From Coefficient Table

We need to normalize Equation 3 to correlate with Equation 1. Dividing through by 1.622 is essentially scaling the gain factor K (which is arbitrary) and normalizing the equation:

$$H_{LP}(f) = \frac{K}{-\left(\frac{f}{1.274fc}\right)^2 + 1.360\,\frac{jf}{fc} + 1}$$

Equation 4. Second-Order Low-Pass Bessel Filter – Normalized Form

Equation 4 is the same as Equation 1 with FSF = 1.274 and $Q = \frac{1}{1.360 \times 1.274} = 0.577$.

5.3 Second-Order Low-Pass Chebyshev Filter With 3-dB Ripple

Referring to a table listing for a 3-dB second-order Chebyshev, the zeros are given as $z_1 = -0.3224 + j0.7772$, $z_1^* = -0.3224 - j0.7772$. From a table of coefficients we get: $a_0 = 0.7080$ and $a_1 = 0.6448$.

Again, using the coefficient form lends itself to a circuit implementation, so that the realization of a second-order low-pass Chebyshev filter with 3-dB of ripple is accomplished with a circuit having a transfer function of the form:

$$H_{LP}(f) = \frac{K}{-\left(\frac{f}{fc}\right)^2 + 0.6448\,\frac{jf}{fc} + 0.7080}$$

Equation 5. Second-Order Low-Pass Chebyshev Filter With 3-dB Ripple – From Coefficient Table

Dividing top and bottom by 0.7080 is again simply scaling of the gain factor K (which is arbitrary), so we normalize the equation to correlate with Equation 1 and get:

$$H_{LP}(f) = \frac{K}{-\left(\frac{f}{0.8414fc}\right)^2 + 0.9107\,\frac{jf}{fc} + 1}$$

Equation 6. Second-Order Low-Pass Chebyshev Filter With 3-dB Ripple – Normalized Form

Equation 6 is the same as Equation 1 with FSF = 0.8414 and $Q = \frac{1}{0.8414 \times 0.9107} = 1.3050$.

The previous work is the first step in designing any of the filters. The next step is to determine a circuit to implement these filters.

6 Low-Pass Sallen-Key Architecture

Figure 1 shows the low-pass Sallen-Key architecture and its ideal transfer function.

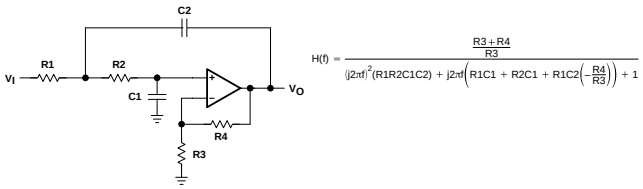


Figure 1. Low-Pass Sallen-Key Architecture

At first glance, the transfer function looks very different from our standard form in Equation 1. Let us make the following substitutions: $K = \frac{R3 + R4}{R3}$, $FSF \times fc = \frac{1}{2\pi\sqrt{R1R2C1C2}}$, and $Q = \frac{\sqrt{R1R2C1C2}}{R1C1 + R2C1 + R1C2(1-K)}$, and they become the same.

Depending on how you use the previous equations, the design process can be simple or tedious. Appendix A shows simplifications that help to ease this process.

7 Low-Pass Multiple-Feedback (MFB) Architecture

Figure 2 shows the MFB filter architecture and its ideal transfer function.

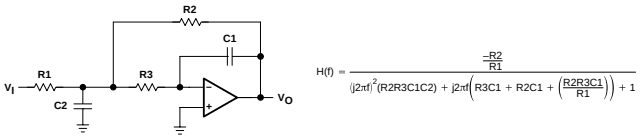


Figure 2. Low-Pass MFB Architecture

Again, the transfer function looks much different than our standard form in Equation 1. Make the following substitutions: $K = \frac{-R2}{R1}$, $FSF \times fc = \frac{1}{2\pi\sqrt{R2R3C1C2}}$, and $Q = \frac{\sqrt{R2R3C1C2}}{R3C1 + R2C1 + R3C1(-K)}$, and they become the same.

Depending on how you use the previous equations, the design process can be simple or tedious. Appendix A shows simplifications that help to ease this process.

The Sallen-Key and MFB circuits shown are second-order low-pass stages that can be used to realize one complex-pole pair in the transfer function of a low-pass filter. To make a Butterworth, Bessel, or Chebyshev filter, set the value of the corresponding circuit components to equal the coefficients of the filter polynomials. This is demonstrated later.

8 Cascading Filter Stages

The concept of cascading second-order filter stages to realize higher-order filters is illustrated in Figure 3. The filter is broken into complex-conjugate-pole pairs that can be realized by either Sallen-Key, or MFB circuits (or a combination). To implement an n-order filter, n/2 stages are required. Figure 4 extends the concept to odd-order filters by adding a first-order real pole. Theoretically, the order of the stages makes no difference, but to help avoid saturation, the stages are normally arranged with the lowest Q near the input and the highest Q near the output. Appendix B shows detailed circuit examples using cascaded stages for higher-order filters.

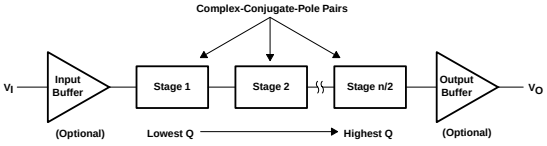


Figure 3. Building Even-Order Filters by Cascading Second-Order Stages

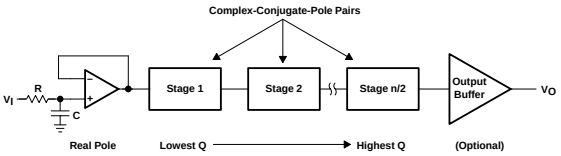


Figure 4. Building Odd-Order Filters by Cascading Second-Order Stages and Adding a Single Real Pole

9 Filter Tables

Typically, filter books list the zeroes or the coefficients of the particular polynomial being used to define the filter type. As we have seen, it takes a certain amount of mathematical manipulation to turn this information into a circuit realization. Although this work is required, it is merely a mechanical operation using the following relationships: frequency scaling factor, $FSF = \sqrt{Re^2 + |Im|^2}$, and quality factor $Q = \frac{\sqrt{Re^2 + |Im|^2}}{2Re}$, where Re is the real part of the complex-zero pair, and Im is the imaginary part. Tables 1 through 4 are generated in this way. It is implicit that higher-order filters are constructed by cascading second-order stages for even-order filters (one for each complex-zero pair). A first-order stage is then added if the filter order is odd. With the filter tables arranged this way, the preliminary mathematical work is done and the designer is left with calculating the proper circuit components based on just three formulas.

For a low-pass Sallen-Key filter with cutoff frequency fc and pass-band gain K, set

$K = \frac{R3 + R4}{R3}$, $FSF \times fc = \frac{1}{2\pi\sqrt{R1R2C1C2}}$, and $Q = \frac{\sqrt{R1R2C1C2}}{R1C1 + R2C1 + R1C2(1-K)}$ for each second-order stage. If an odd order is required, set $FSF \times fc = \frac{1}{2\pi RC}$ for that stage.

For a low-pass MFB filter with cutoff frequency fc and pass-band gain K, set

$K = \frac{-R2}{R1}$, $FSF \times fc = \frac{1}{2\pi\sqrt{R2R3C1C2}}$, and $Q = \frac{\sqrt{R2R3C1C2}}{R3C1 + R2C1 + R3C1(-K)}$ for each second-order stage. If an odd order is required, set $FSF \times fc = \frac{1}{2\pi RC}$ for that stage.

The tables are arranged so that increasing Q is associated with increasing stage order. High-order filters are normally arranged in this manner to help prevent clipping.

Table 1. Butterworth Filter Table

FILTER ORDER	Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
	FSF	Q	FSF	Q	FSF	Q	FSF	Q	FSF	Q
2	1.000	0.7071								
3	1.000	1.0000	1.000							
4	1.000	0.5412	1.000	1.3065						
5	1.000	0.6180	1.000	1.6181	1.000					
6	1.000	0.5177	1.000	0.7071	1.000	1.9320				
7	1.000	0.5549	1.000	0.8019	1.000	2.2472	1.000			
8	1.000	0.5098	1.000	0.6013	1.000	0.8999	1.000	2.5628		
9	1.000	0.5321	1.000	0.6527	1.000	1.0000	1.000	2.8802	1.000	
10	1.000	0.5062	1.000	0.5612	1.000	0.7071	1.000	1.1013	1.000	3.1969

Table 2. Bessel Filter Table

FILTER ORDER	Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
	FSF	Q	FSF	Q	FSF	Q	FSF	Q	FSF	Q
2	1.2736	0.5773								
3	1.4524	0.6910	1.3270							
4	1.4192	0.5219	1.5912	0.8055						
5	1.5611	0.5635	1.7607	0.9165	1.5069					
6	1.6060	0.5103	1.6913	0.6112	1.9071	1.0234				
7	1.7174	0.5324	1.8235	0.6608	2.0507	1.1262	1.6853			
8	1.7837	0.5060	2.1953	1.2258	1.9591	0.7109	1.8376	0.5596		
9	1.8794	0.5197	1.9488	0.5894	2.0815	0.7606	2.3235	1.3220	1.8575	
10	1.9490	0.5040	1.9870	0.5380	2.0680	0.6200	2.2110	0.8100	2.4850	1.4150

Table 3. 1-dB Chebyshev Filter Table

FILTER ORDER	Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
	FSF	Q	FSF	Q	FSF	Q	FSF	Q	FSF	Q
2	1.0500	0.9565								
3	0.9971	2.0176	0.4942							
4	0.5286	0.7845	0.9932	3.5600						
5	0.6552	1.3988	0.9941	5.5538	0.2895					
6	0.3532	0.7608	0.7468	2.1977	0.9953	8.0012				
7	0.4800	1.2967	0.8084	3.1554	0.9963	10.9010	0.2054			
8	0.2651	0.7530	0.5838	1.9564	0.5538	2.7776	0.9971	14.2445		
9	0.3812	1.1964	0.6623	2.7119	0.8805	5.5239	0.9976	18.0069	0.1593	
10	0.2121	0.7495	0.4760	1.8639	0.7214	3.5609	0.9024	6.9419	0.9981	22.2779

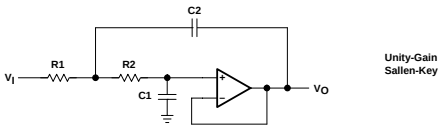
Table 4. 3-dB Chebyshev Filter Table

FILTER ORDER	Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
	FSF	Q	FSF	Q	FSF	Q	FSF	Q	FSF	Q
2	0.8414	1.3049								
3	0.9160	3.0678	0.2986							
4	0.4426	1.0765	0.9503	5.5770						
5	0.6140	2.1380	0.9675	8.8111	0.1775					
6	0.2980	1.0441	0.7224	3.4597	0.9771	12.7899				
7	0.4519	1.9821	0.7920	5.0193	0.9831	17.4929	0.1265			
8	0.2228	1.0558	0.5665	3.0789	0.8388	6.8302	0.9870	22.8481		
9	0.3559	1.9278	0.6503	4.3179	0.8716	8.8756	0.9897	28.9400	0.0983	
10	0.1796	1.0289	0.4626	2.9350	0.7126	5.7012	0.8954	11.1646	0.9916	35.9274

10 Example-Circuit Test Results

To further show how to use the above information and see actual circuit performance, component values are calculated and the filter circuits are built and tested.

Figures 5 and 6 show typical component values computed for the three different filters discussed using the Sallen-Key architecture and the MFB architecture. The equivalent simplification (see Appendix A) is used for each circuit: setting the filter components as ratios and the gain equal to 1 for the Sallen-Key, and the gain equal to –1 for the MFB. The circuits and simplifications are shown for convenience. A corner frequency of 1 kHz is chosen. The values used for m and n are shown. C1 and C2 are chosen to be standard values. The values shown for R1 and R2 are the nearest standard values to those computed by using the formulas given.



R1=mR, R2=R, C1=C, C2=nC, and K=1 result in: $FSF \times fc = \frac{1}{2\pi RC\sqrt{mn}}$, and $Q = \frac{\sqrt{mn}}{m+1}$

FILTER TYPE	n	m	C1	C2	R1	R2
Butterworth	3.3	0.229	0.01 µF	0.033 µF	4.22 kΩ	18.2 kΩ
Bessel	1.5	0.42	0.01 µF	0.015 µF	7.15 kΩ	14.3 kΩ
3-dB Chebyshev	6.8	1.0	0.01 µF	0.068 µF	7.32 kΩ	7.32 kΩ

Figure 5. Sallen-Key Circuit and Component Values – fc = 1 kHz

R2=R, R3=mR, C1=C, C2=nC, and K=1 results in: $FSF \times fc = \frac{1}{2\pi RC\sqrt{mn}}$, and $Q = \frac{\sqrt{mn}}{1+2m}$

FILTER TYPE	n	m	C1	C2	R1 & R2	R3
Butterworth	4.7	0.222	0.01 µF	0.047 µF	15.4 kΩ	3.48 kΩ
Bessel	3.3	0.195	0.01 µF	0.033 µF	15.4 kΩ	3.01 kΩ
3-dB Chebyshev	15	10.268	0.01 µF	0.15 µF	9.53 kΩ	2.55 kΩ

Figure 6. MFB Circuit and Component Values – fc = 1 kHz

The circuits are built using a TLV2772 operational amplifier, 1%-tolerance resistors, and 10%-tolerance capacitors. Figures 7 through 10 show the measured frequency response of the circuits. Figure 11 shows the transient response of the filters to a pulse input.

Figure 7 compares the frequency response of Sallen-Key and MFB second-order Butterworth filters. The frequency response of the filters is almost identical from 10 Hz to about 40 kHz. Above this, the MFB shows better performance. This will be examined latter.

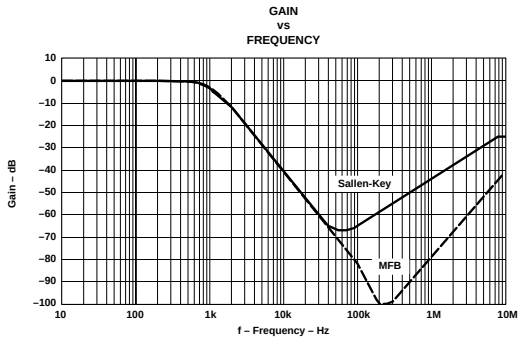


Figure 7. Second-Order Butterworth Filter Frequency Response

Figure 8 compares the frequency response of Sallen-Key and MFB second-order Bessel filters. The frequency response of the filters is almost identical from 10 Hz to about 50 kHz. Above this, the MFB has superior performance. This will be examined latter.

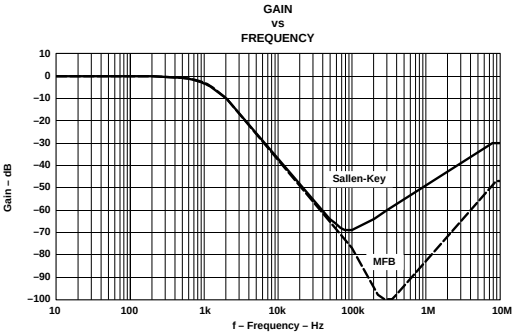


Figure 8. Second-Order Bessel Filter Frequency Response

Figure 9 compares the frequency response of Sallen-Key and MFB second-order 3-dB Chebyshev filters. The frequency response of the filters is almost identical from 10 Hz to about 50 kHz. Above this, the MFB shows better performance. This will be examined shortly.

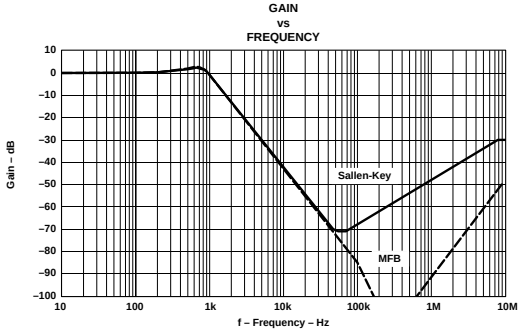


Figure 9. Second-Order 3-dB Chebyshev Filter Frequency Response

Figure 10 is an expanded view of the frequency response of the three filters in the MFB topology, near fc (the Sallen-Key circuits are almost identical). It clearly shows the increased rate of attenuation near the cutoff frequency, going from the Bessel to the 3-dB Chebyshev.

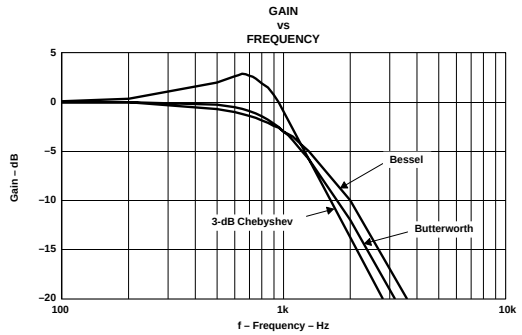


Figure 10. Second-Order Butterworth, Bessel, and 3-dB Chebyshev Filter Frequency Response

Figure 11 shows the transient response of the three filters using MFB architecture to a pulse input (the Sallen-Key circuits are almost identical). It clearly shows the increased overshoot going from the Bessel to the 3-dB Chebyshev.

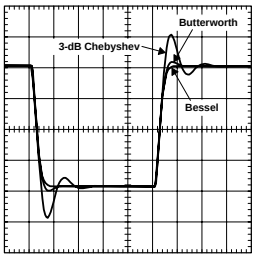


Figure 11. Transient Response of the Three Filters

11 Nonideal Circuit Operation

Up to now we have not discussed nonideal operation of the circuits. The test results shown in Figures 7 through 9 show that at high frequency, where you expect the response to keep attenuating at –40 dB/dec, the filters actually turn around and start passing signals at increasing amplitudes. We will now examine why this happens.

11.1 Nonideal Circuit Operation – Sallen-Key

At frequencies well above cutoff, simplified high-frequency models help show the expected behavior of the circuits. Figure 12 is used to show the expected circuit operation for a second-order low-pass Sallen-Key circuit at high frequency. The assumption made here is that C1 and C2 are effective shorts when compared to the impedance of R1 and R2 so that the amplifier's input is at ac ground. In response, the amplifier generates an ac ground at its output, limited only by its output impedance Zo. The formula shows the transfer function of this model.

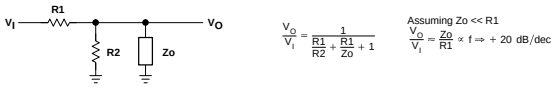


Figure 12. Second-Order Low-Pass Sallen-Key High-Frequency Model

Zo is the closed-loop output impedance. It depends on the loop transmission and the open-loop output impedance zo: $Zo = \frac{zo}{1 + a(f)\beta}$, where a(f)β is the loop transmission. β is the feedback factor set by resistors R3 and R4 and is constant over frequency, but the open loop gain a(f) is dependant on frequency. With dominant-pole compensation, the open-loop gain of the amplifier decreases at –20 dB/dec over the usable frequencies of operation. Assuming that zo is mainly resistive (usually a valid assumption up to 100 MHz), Zo increases at a rate of 20 dB/dec. At high frequencies the circuit is no longer able to attenuate the input and begins to pass the signal at a 20-dB/dec rate, as the test results show.

Placing a low-pass RC filter at the output of the amplifier can help nullify the feed-through of high-frequency signals. Figure 13 shows a comparison between the original Sallen-Key Butterworth filter and one using an RC filter on the output. A 100-Ω resistor is placed in series with the output, and a 0.047-µF capacitor is connected from the output to ground This places a passive pole in the transfer function at about 40 kHz that improves the high-frequency response.

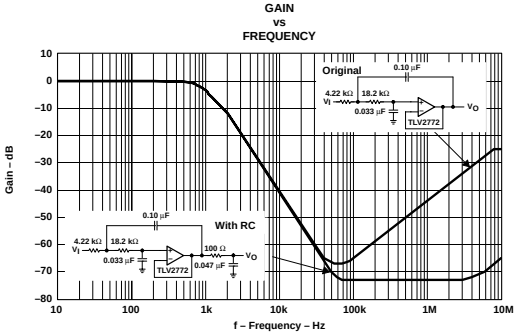


Figure 13. Sallen-Key Butterworth Filter With RC Added in Series With the Output

11.2 Nonideal Circuit Operation – MFB

The high-frequency analysis of the MFB is very similar to the Sallen-Key. Figure 14 is used to show the expected circuit operation for a second-order low-pass MFB circuit at high frequency. The assumption made here is that C1 and C2 are effective shorts when compared to the impedance of R1, R2, and R3. Again, the amplifier's input is at ac ground, and generates an ac ground at its output limited only by its output impedance Zo. Capacitor Cp represents the parasitic capacitance from V_I to V_O. The ability of the circuit to attenuate high-frequency signals is dependent on Cp and Zo. The impedance of Cp decreases at –20 dB/dec and Zo increases at 20 dB/dec. The overall transfer function turns around at high frequency to 40 dB/dec as seen in the laboratory data. Spice simulation shows that as little as 0.4 pF will produce the high-frequency feed through observed.

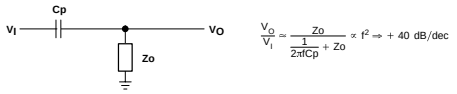


Figure 14. Second-Order Low-Pass MFB High-Frequency Model

Care should be taken when routing the input and output signals to keep capacitive coupling to a minimum.

Placing a low-pass RC filter at the output of the amplifier can help nullify the feed through of high-frequency signals. Figure 15 below shows a comparison between the original MFB Butterworth filter with one using an RC filter on the output. A 100-Ω resistor is placed in series with the output, and a 0.047-µF capacitor is connected from the output to ground. This places a passive pole in the transfer function at about 40 kHz that improves the high-frequency response.

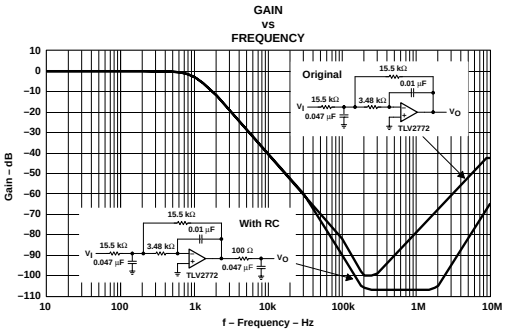


Figure 15. MFB Butterworth Filter With RC Added in Series With the Output

12 Comments About Component Selection

Theoretically, any values of R and C which satisfy the equations might be used, but practical considerations call for certain guidelines to be followed.

Given a specific corner frequency, the values of R and C are inversely proportional to each other. By making C larger R becomes smaller, and vice versa.

Making R large may make C so small that parasitic capacitors cause errors. This makes smaller resistor values preferred over larger resistor values.

The best choice of component values depends on the particulars of your circuit and the tradeoffs your are willing to make. Adhering to the following general recommendations will help reduce errors:

- **Capacitors**
  - Avoid values less than 10 pF
  - Use NPO or COG dielectrics
  - Use 1%-tolerance components
  - Surface mount is preferred.
- **Resistors**
  - Values in the range of a few-hundred ohms to a few-thousand ohms are best.
  - Use metal film with low-temperature coefficients.
  - Use 1% tolerance (or better).
  - Surface mount is preferred.

13 Conclusion

We have investigated building second-order low-pass Butterworth, Bessel, and 3-dB Chebyshev filters using the Sallen-Key and MFB architectures. The same techniques are extended to higher-order filters by cascading second-order stages for even order, and adding a first-order stage for odd order.

The advantages of each filter type come at the expense of other characteristics. The Butterworth is considered by a lot of people to offer the best all-around filter response. It has maximum flatness in the pass-band with moderate rolloff past cutoff, and shows only slight overshoot in response to a pulse input.

The Bessel is important when signal-conditioning square-wave signals. The constant-group delay means that the square-wave signal is passed with minimum distortion (overshoot). This comes at the expense of a slower rate of attenuation above cutoff.

The 3-dB Chebyshev sacrifices pass-band flatness for a high rate of attenuation near cutoff. It also exhibits the largest overshoot and ringing in response to a pulse input of the three filter types discussed.

The Sallen-Key and MFB architectures also have trade-offs associated with them. The simplifications that can be used when designing the Sallen-Key provide for easier selection of circuit components, and at unity gain, it has no gain sensitivity to component variations. The MFB shows less overall sensitivity to component variations and has superior high-frequency performance.

Tables 5 and 6 give a brief summary of the previous trade-offs.

Table 5. Summary of Filter Type Trade-Offs

FILTER TYPE	ADVANTAGE(s)	DISADVANTAGE(s)
Butterworth	Maximum pass-band flatness	Slight overshoot in response to pulse input and moderate rate of attenuation above fc
Bessel	Constant group delay – no overshoot with pulse input	Slow rate of attenuation above fc
3-dB Chebyshev	Fast rate of attenuation above fc	Large overshoot and ringing in response to pulse input

Table 6. Summary of Architecture Trade-Offs

ARCHITECTURE	ADVANTAGE(s)	DISADVANTAGE(s)
Sallen-Key	Not sensitive to component variation at unity gain	High-frequency response limited by the frequency response of the amplifier
MFB	Less sensitive to component variations and superior high-frequency response	Less simplifications available to ease design

5 ATTACHEMENT E: MATLAB FILTERS

```
%--------------------------
% FUNCTION TO ANALYZE FILTER
%--------------------------

% PARAMS: PASSING FREQUENCY, STOP FREQUENCY, RIBBLE IN PASSING BAND,
    ATTENUATED FREQUENCY

function analyze_filter(f_pass,f_stop,R_pass,R_stop)

[N,Wp] = ellipord(2*pi*f_pass,2*pi*f_stop,R_pass,R_stop,'s')
[B,A] = ellip(N,R_pass,R_stop,Wp,'s')
freqs(B,A,2*pi*[0:1:12.5e3])
```

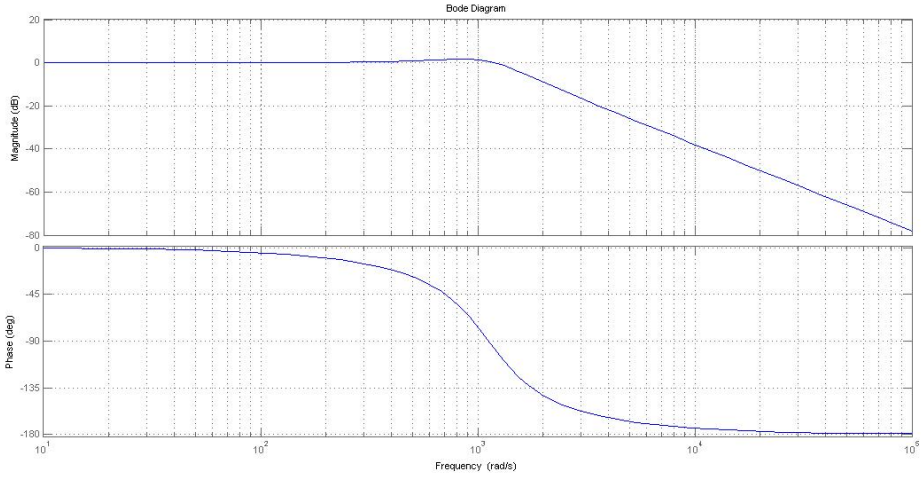


Figure 1: Stage 1

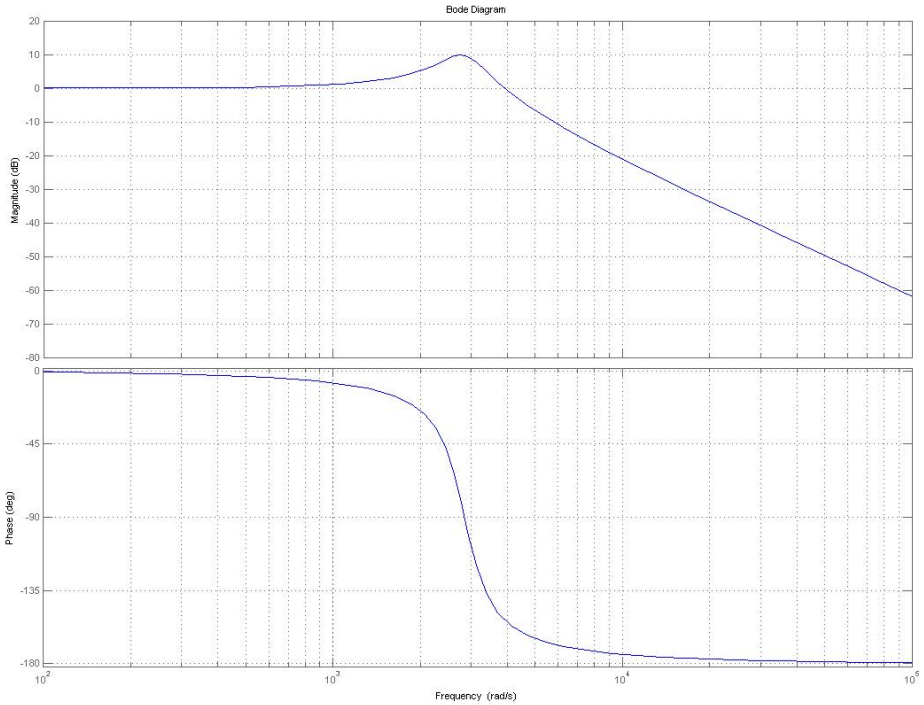


Figure 2: Stage 2

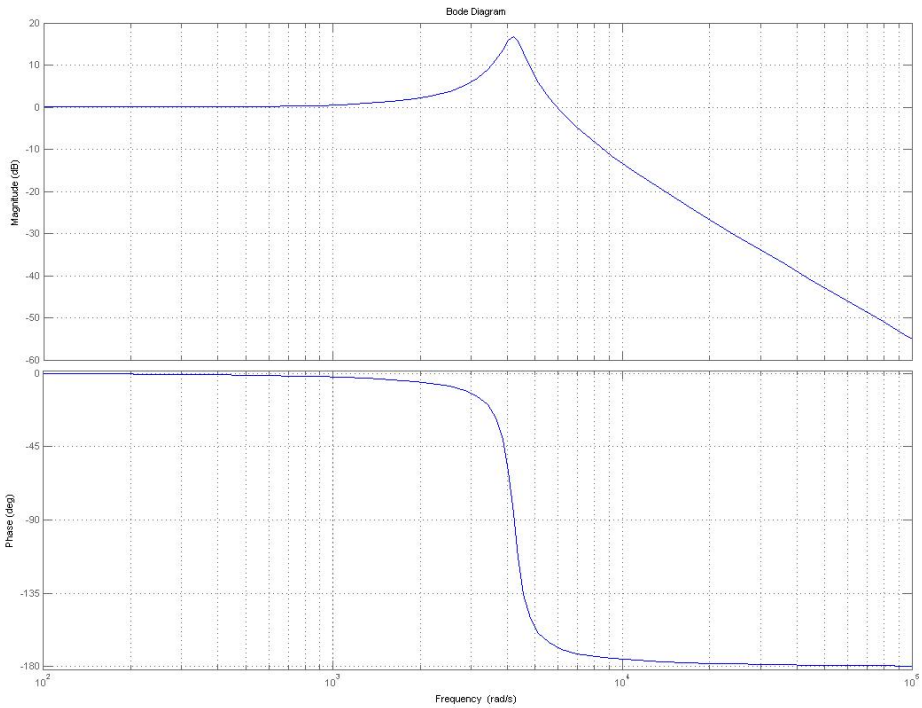


Figure 3: Stage 3

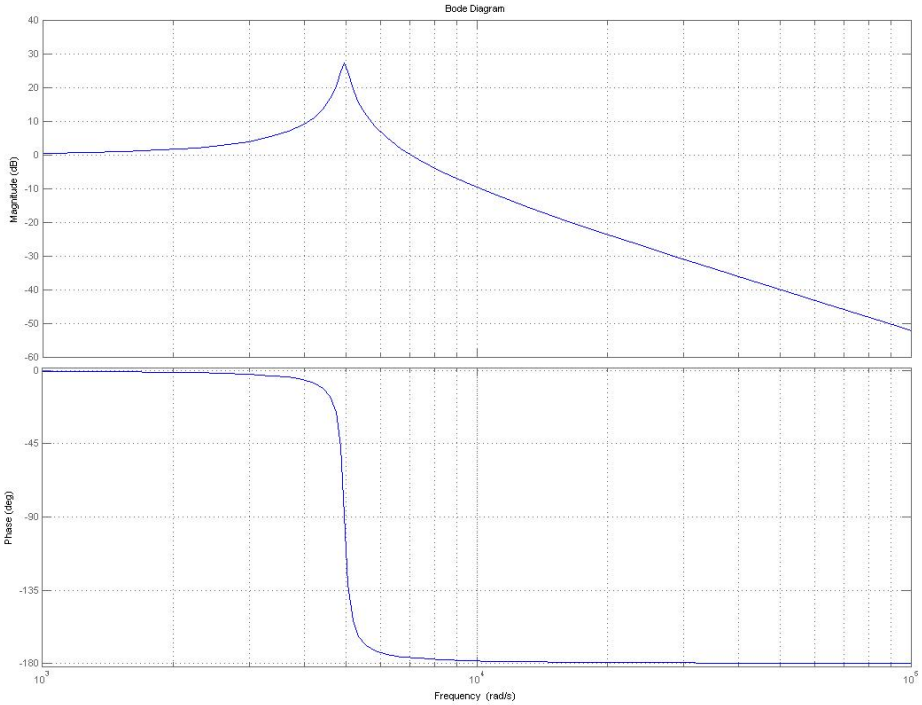
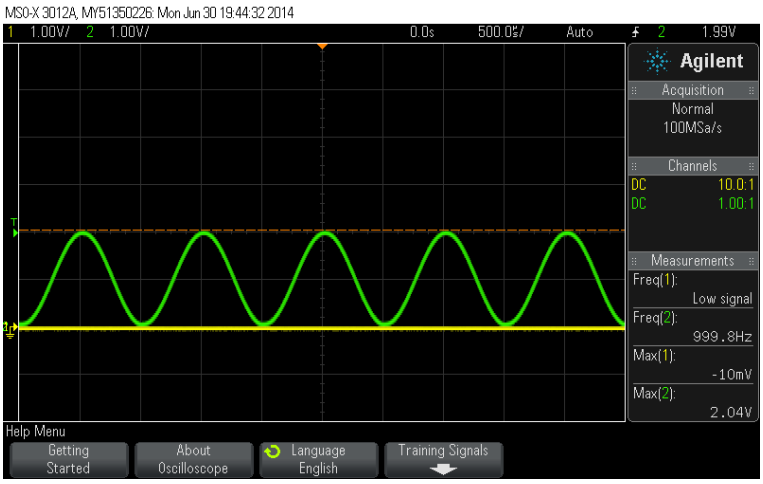
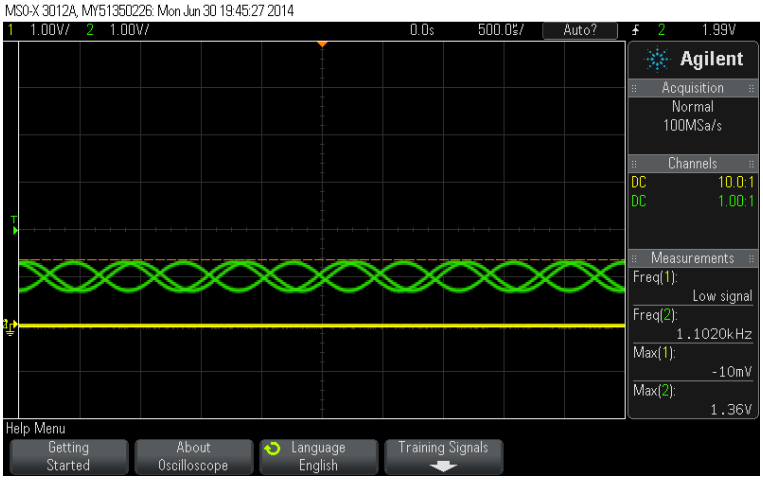


Figure 4: Stage 4

6 ATTACHEMENT F: FILTERED SIGNAL





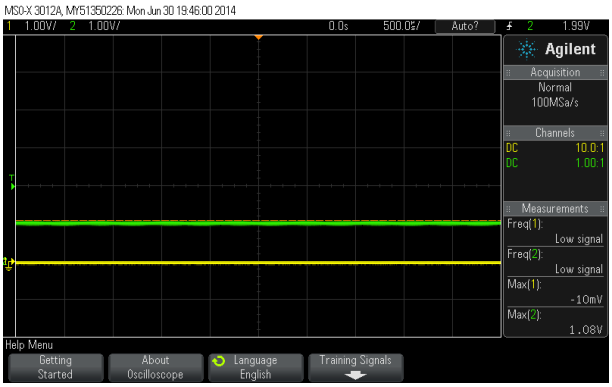


Figure 5: Filtered Signal

7 ATTACHEMENT G: KEIL CODE

```c
/**
 * @file   adc.c
 * @brief  Contains all necessary functions for the ADC
 *
 *
 * @author Thierry HISCHIER
 * @version    1.0
 * @date   November 2013
 */

#include "adc.h"
#include "gpio.h"
#include "arm_math.h"
#include "uart.h"


#define NB_SAMPLES 3000
/**
 *
 * VARIABLE'S DECLARATION
 *
 */

    uint16_t adc_value;
    uint16_t old_adc_value;
    uint32_t pass2send;
    uint32_t counter2;
    bool done;
    bool didFinishConversion;

    float32_t         t_total;



/**@brief ADC interrupt handler.
 *
 * @details    This function will fetch the conversion result from ADC, convert
     the value into
 *         milivolt and send it to peer (battery measure) or stock them in an
     measure table (pressure measure).
 */
void ADC_IRQHandler( void )
{

    if( NRF_ADC->EVENTS_END != 0 )
    {

        old_adc_value = adc_value;

        NRF_ADC->EVENTS_END      = 0;
        adc_value = (uint16_t)NRF_ADC->RESULT;
        NRF_ADC->TASKS_STOP      = 1;

        didFinishConversion = true; // Flag to start the adaptive counting
            algorithm
    }

}
void adc_FFT_start()
{
```



```c
}
/**
 * @brief Function to make the ADC start a battery level conversion.
 */
void adc_bas_start( void )
{
    NRF_ADC->TASKS_STOP = 1;
    // Setup the ADC for Battery Service
    adc_init( ADC_CONFIG_RES_8bit,
              ADC_CONFIG_INPSEL_AnalogInputOneThirdPrescaling,
              ADC_CONFIG_REFSEL_VBG,
               ADC_CONFIG_PSEL_AnalogInput3,
              ADC_CONFIG_EXTREFSEL_None );

    // Starts a measurement
    NRF_ADC->TASKS_START  = 1;
}

/**
 * @brief  Configure the ADC
 */
void adc_init( uint32_t resolution,
               uint32_t reference_prescaler,
               uint32_t reference,
               uint32_t analog_input,
               uint32_t extrefsel )
{
    uint32_t err_code;

    // Configure ADC to convert measure
    NRF_ADC->INTENSET =   ADC_INTENSET_END_Msk;
    NRF_ADC->CONFIG   =  ( resolution           << ADC_CONFIG_RES_Pos )    |
                         ( reference_prescaler  << ADC_CONFIG_INPSEL_Pos ) |
                         ( reference            << ADC_CONFIG_REFSEL_Pos ) |
                         ( analog_input         << ADC_CONFIG_PSEL_Pos )   |
                         ( extrefsel            << ADC_CONFIG_EXTREFSEL_Pos );

    NRF_ADC->EVENTS_END = 0;
    NRF_ADC->ENABLE       = ADC_ENABLE_ENABLE_Enabled;

    // Enable ADC interrupt
    err_code = sd_nvic_ClearPendingIRQ( ADC_IRQn );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }

    err_code = sd_nvic_SetPriority( ADC_IRQn, NRF_APP_PRIORITY_LOW );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }

    err_code = sd_nvic_EnableIRQ( ADC_IRQn );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
```

```
        // Stop any running conversions
        NRF_ADC->EVENTS_END = 0;
    }
```

```
/**
 * @file   main.c
 * @brief  Main file for the project
 *
 * @author Aurelien MERZ
 * @version   1.0
 * @date   July 2013
 */

#include "config.h"
#include "adc.h"
#include "advertise.h"
#include "apptimer.h"
#include "gpio.h"
#include "services.h"
#include "timerConstants.h"
#include "simple_uart.h"

#include "arm_math.h"


//#include <stdio.h>

/**
 *
 * VARIABLE'S DECLARATION
 *
 */

//#define NDATA 1024

// PPI defines
#define PPI_CHAN0_GPIO_P002                0
            /**< The PPI channel that connects CC0 compare event to the
    GPIOTE to controle MOSFET transistor. */
#define PPI_CHAN1_TO_MEASURE_ADC        1
          /**< The PPI channel that connects CC1 compare event to the ADC task
    that take a new ADC measure. */
#define PPI_CHAN2_GPIO_P002                2
            /**< The PPI channel that connects CC1 compare event to the
    GPIOTE to controle MOSFET transistor. */

// GPIO defines
#define GPIO_P0_2                       2
          /**< To drive the P0.02 */

//GPIOTE defines
#define GPIOTE_CHAN_GPIO_P0_02_TASK        0
            /**< The GPIOTE channel used to perform write operation on the
    pin P0.02. */
#define GPIOTE_CHAN_GPIO_P0_02_ENDTASK     2
            /**< The GPIOTE channel used to perform write operation on the
    pin P0.02. */

// TIMER defines
#define CAPTURE_COMPARE_0_VALUE            0x0232
            /**< Timer Delay in milli-seconds | 0x1E8480 => 2s, 0xF4240 =>
    1s, 0xF424 => 500ms, 0x7A12 => 250ms, 0x30D4 => 100ms, 0x0232 => 4.5ms */
#define CAPTURE_COMPARE_1_VALUE            0x0271
            /**< Timer Delay in milli-seconds | 0x1E8480 => 2s, 0xF4240 =>
```



```
    1s, 0xF424 => 500ms, 0x7A12 => 250ms, 0x30D4 => 100ms, 0x0271 => 5ms */
#define TIMER_PRESCALER                    7
             /**< TIMER prescaler value -> Prescaler = 7 => 1 tick = 8us. */
#define CAPTURE_COMPARE_VALUE           0x1F   //0xA for 80us

#define TEST_LENGTH_SAMPLES 256
#define SAMPLING_FREQ 14700
#define PDU_LENGTH 2
#define CHAR_LEN 8

#define UART_BUFFER_SIZE 10
#define GPIO_UART_TXD 9
#define GPIO_UART_RXD 11

/* ------------------------------------------------------------------
* FINITE STATE MACHINE STATES & FUNCTIONS DECLARTIONS
* ------------------------------------------------------------------ */
// Trigger Values
#define V_HIGH 530
#define V_LOW  330
#define NB_SAMPLES 3000

uint16_t periodCount = 0;
float32_t freq = 0.0;
extern uint16_t adc_value;

// FSM STATES
typedef enum{
          TRIGGER_STATE_H,
          TRIGGER_STATE_L,
          SENDING_STATE,
          IDLE_STATE
} SM_STATES;

typedef enum{
          IDLE,
          HIGH_STATE,
          LOW_STATE,
          ARMED_STATE,
          TEMP_STATE
}SM2_STATES;

SM_STATES actualState;
SM2_STATES sm2_state;
// Global variables
static ble_gap_sec_params_t
    m_sec_params;                              /**< Security requirements for
    this application. */
uint16_t                                m_conn_handle =
    BLE_CONN_HANDLE_INVALID;    /**< Handle of the current connection. */

// // UART

// Packer from data compression,  Holds the pointer to the buffer to
// write the data to and the actual position inside the buffer.
//BitPacker packer;
uint8_t buffer[PDU_LENGTH];
uint8_t charact[CHAR_LEN];

extern uint32_t pass2send;
```

```
extern uint32_t counter2;
extern float32_t         t_total;
uint32_t freq2send;
float32_t frequency;
float32_t samplingPeriod;
extern bool done;

uint8_t LSB;
uint8_t MSB;

bool connected = false;
extern bool didFinishConversion;
extern uint16_t adc_value;
extern uint16_t old_adc_value;
uint32_t counter = 0;

uint32_t err_code;
float32_t avg = 512;
uint32_t pass = 0;

/**
 *
 * PUBLIC METHODS
 *
 */

/* ----------------------------------------------------------------------
* Sending the period value to another BLE device
* ----------------------------------------------------------------------*/
void sendPeriod_BLE(void){

    uint32_t err_code;
    // Send over BLE
    buffer[0] = periodCount;
    buffer[1] = periodCount >> 8;

    err_code = ble_sns_attr_adcVal_send(&m_sns, buffer, sizeof(buffer));
    if( ( err_code != NRF_SUCCESS ) &&
        ( err_code != NRF_ERROR_INVALID_STATE ) &&
        ( err_code != BLE_ERROR_NO_TX_BUFFERS ) &&
        ( err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING ) )
    {
        ASSERT( false );
    }
}
/*----------------------------------------------------------------------
* Updates the state machine depending on the adc value
----------------------------------------------------------------------*/
void update_fsm(uint16_t adc_value){

    if((adc_value < V_LOW ) && (actualState == TRIGGER_STATE_H))
    {
        actualState = TRIGGER_STATE_L;
  }
    else if((adc_value > V_HIGH ) && (actualState == TRIGGER_STATE_L))
    {
        actualState = SENDING_STATE;
  }
    else if((adc_value > V_HIGH) && (actualState == IDLE_STATE))
    {
```



```
        actualState = TRIGGER_STATE_H;
  }
    else if((adc_value < V_LOW) && (actualState == IDLE_STATE))
    {
        actualState = TRIGGER_STATE_L;
    }

}
/*----------------------------------------------------------------------
* Count the period reccurence
* ----------------------------------------------------------------------*/
void countPeriod(SM_STATES state){

    switch(state)
    {
        case TRIGGER_STATE_H:
                                periodCount++;
                                break;

        case TRIGGER_STATE_L:
                                periodCount++;
                                break;

        case SENDING_STATE:
                                sendPeriod_BLE();
                                //freq = 4000 / periodCount;
                                periodCount = 0;
                                actualState = TRIGGER_STATE_H;
                                break;

        case IDLE_STATE: break;
    }

}


/**
 * @brief Error handler function, which is called when an error has occurred.
 *
 * @param[in] error_code  Error code supplied to the handler.
 * @param[in] line_num    Line number where the handler is called.
 * @param[in] p_file_name Pointer to the file name.
 */
void app_error_handler(uint32_t error_code, uint32_t line_num, const uint8_t *
    p_file_name)
{
    // Copying parameters to static variables because parameters are not
       accessible in debugger.
    static volatile uint8_t  s_file_name[128];
    static volatile uint16_t s_line_num;
    static volatile uint32_t s_error_code;

    strcpy((char *)s_file_name, (const char *)p_file_name);
    s_line_num   = line_num;
    s_error_code = error_code;
    UNUSED_VARIABLE(s_file_name);
    UNUSED_VARIABLE(s_line_num);
    UNUSED_VARIABLE(s_error_code);

    for (;;)
```

```
    {
        // Loop forever. On assert, the system can only recover on reset.
    }
}

/**
 * @brief Assert macro callback function.
 *
 * @details This function will be called if the ASSERT macro fails.
 *
 * @param[in]   line_num   Line number of the failing ASSERT call.
 * @param[in]   file_name  File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * file_name)
{
    // Copying parameters to static variables because parameters are not
       accessible in debugger
    static volatile uint8_t  s_file_name[128];
    static volatile uint16_t s_line_num;

    strcpy((char *)s_file_name, (const char *)file_name);
    s_line_num = line_num;
    UNUSED_VARIABLE(s_file_name);
    UNUSED_VARIABLE(s_line_num);

    for (;;)
    {
        // Loop forever. On assert, the system can only recover on reset
    }
}

/**
 * @brief GAP initialization.
 *
 * @details This function shall be used to setup all the necessary GAP
    (Generic Access Profile)
 *          parameters of the device. It also sets the permissions and
    appearance.
 */
static void gap_params_init( void )
{
    uint32_t              err_code;
    ble_gap_conn_params_t gap_conn_params;
    ble_gap_conn_sec_mode_t   sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN( &sec_mode );

    err_code = sd_ble_gap_device_name_set( &sec_mode, ( const uint8_t * )
        DEVICE_NAME, strlen( DEVICE_NAME ) );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }

    err_code = sd_ble_gap_appearance_set( BLE_APPEARANCE_GENERIC_KEYRING );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
```



```
    memset( &gap_conn_params, 0, sizeof( gap_conn_params ) );
    gap_conn_params.min_conn_interval   = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval   = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency    = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set( &gap_conn_params );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }

    // Set TX Power
    err_code = sd_ble_gap_tx_power_set( TX_POWER_LEVEL );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
}


/**
 * @brief Initialize security parameters.
 */
static void sec_params_init( void )
{
    m_sec_params.timeout      = SEC_PARAM_TIMEOUT;
    m_sec_params.bond         = SEC_PARAM_BOND;
    m_sec_params.mitm         = SEC_PARAM_MITM;
    m_sec_params.io_caps      = SEC_PARAM_IO_CAPABILITIES;
    m_sec_params.oob          = SEC_PARAM_OOB;
    m_sec_params.min_key_size = SEC_PARAM_MIN_KEY_SIZE;
    m_sec_params.max_key_size = SEC_PARAM_MAX_KEY_SIZE;
}


/**
 * @brief Connection Parameters module error handler.
 *
 * @param[in]  nrf_error  Error code containing information about what went
     wrong.
 */
static void conn_params_error_handler( uint32_t nrf_error )
{
    ASSERT( false );
}


/**
 * @brief Initialize the Connection parameters
 */
static void conn_params_init( void )
{
    uint32_t              err_code;
    ble_conn_params_init_t    cp_init;

    memset( &cp_init, 0, sizeof( cp_init ) );

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay  = FIRST_CONN_PARAMS_UPDATE_DELAY;
```

```
    cp_init.next_conn_params_update_delay   = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle     = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail              = true; /*false*/
    cp_init.evt_handler                     = NULL; /*on_conn_params_evt*/
    cp_init.error_handler                = conn_params_error_handler;

    err_code = ble_conn_params_init( &cp_init );
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
}


/**
 * @brief Application's BLE Stack event handler.
 *
 * @param[in]  p_ble_evt  Bluetooth stack event
 */
static void on_ble_evt( ble_evt_t * p_ble_evt )
{
    uint32_t                         err_code;
    static ble_gap_evt_auth_status_t m_auth_status;
    ble_gap_enc_info_t *             p_enc_info;

    switch( p_ble_evt->header.evt_id )
    {
        case BLE_GAP_EVT_CONNECTED:
            connected = true;
            NRF_TIMER1->TASKS_START = 1;
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;

            break;

        case BLE_GAP_EVT_DISCONNECTED:
            connected = false;
            m_conn_handle = BLE_CONN_HANDLE_INVALID;
          apptimer_stop_adc_timer();
            advertising_start();
            break;

        case BLE_GAP_EVT_SEC_PARAMS_REQUEST:
            err_code = sd_ble_gap_sec_params_reply( m_conn_handle,
                                                    BLE_GAP_SEC_STATUS_SUCCESS,
                                                    &m_sec_params );
            break;

        case BLE_GATTS_EVT_SYS_ATTR_MISSING:
            err_code = sd_ble_gatts_sys_attr_set( m_conn_handle, NULL, 0 );
            break;

        case BLE_GAP_EVT_AUTH_STATUS:
            m_auth_status = p_ble_evt->evt.gap_evt.params.auth_status;
            break;

        case BLE_GAP_EVT_SEC_INFO_REQUEST:
            p_enc_info = &m_auth_status.periph_keys.enc_info;
            if( p_enc_info->div == p_ble_evt->evt.gap_evt.params.
                sec_info_request.div )
```



```
            {
                err_code = sd_ble_gap_sec_info_reply( m_conn_handle, p_enc_info,
                    NULL );
            }
            else
            {
                // No keys found for this device
                err_code = sd_ble_gap_sec_info_reply( m_conn_handle, NULL, NULL
                    );
            }
            break;

        case BLE_GAP_EVT_TIMEOUT:
            if( p_ble_evt->evt.gap_evt.params.timeout.src ==
                BLE_GAP_TIMEOUT_SRC_ADVERTISEMENT )
            {
                // Go to system-off mode (this function will not return; wakeup
                    will cause a reset)
                //err_code = sd_power_system_off();
            }
            break;

        default:
            break;
    }
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
}


/**
 * @brief Dispatechs a BLE stack event to all modules with a BLE Stack event
     handler.
 *
 * @param[in]  p_ble_evt  Bluetooth stack event
 */
static void ble_evt_dispatch( ble_evt_t * p_ble_evt )
{
    ble_conn_params_on_ble_evt( p_ble_evt );

    // Service event handlers
    ble_bas_on_ble_evt( &m_bas, p_ble_evt );
    ble_sns_on_ble_evt(&m_sns, p_ble_evt);
    ble_srws_on_ble_evt(&m_srws, p_ble_evt);

    // General Bluetooth stack event handler function
    on_ble_evt( p_ble_evt );
}


/**
 * @brief BLE stack initialization.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init( void )
{
    // The Low Frequency clock is generated from a external crystal
```

```
    BLE_STACK_HANDLER_INIT( NRF_CLOCK_LFCLKSRC_XTAL_20_PPM,
                            BLE_L2CAP_MTU_DEF,
                            ble_evt_dispatch,
                            false );
}


/**
 * @brief Power management.
 */
static void power_manage( void )
{
    uint32_t err_code = sd_app_event_wait();
    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }
}

/*****************************************************************/
void initADC(void)
{
    adc_init( ADC_CONFIG_RES_10bit,
              ADC_CONFIG_INPSEL_AnalogInputNoPrescaling,//
                 ADC_CONFIG_INPSEL_AnalogInputOneThirdPrescaling,
              ADC_CONFIG_REFSEL_VBG,
               ADC_CONFIG_PSEL_AnalogInput4,
              ADC_CONFIG_EXTREFSEL_None );
}

/* @brief PPI initialisation function.
 *
 * @details This function will initialise Programmable Peripheral Interconnect
     peripheral. It will
 *          configure the PPI channels as follows –
 *              PPI Channel 0 – Connecting CC0 Compare event to GPIOTE Task to
    toggle the LED state
 *          This configuration will feed a PWM input to the LED thereby making
    it flash in an
 *          interval that is dependent on the TIMER configuration.
 */
static void ppi_init( void )
{
    uint32_t err_code;

    // Assign PPI Channel 1 to start ADC task
    err_code = sd_ppi_channel_assign( PPI_CHAN1_TO_MEASURE_ADC,
                                  &( NRF_TIMER1->EVENTS_COMPARE[ 1 ] ),
                                  &( NRF_ADC->TASKS_START ) );

    if( err_code != NRF_SUCCESS )
    {
        ASSERT( false );
    }

    // Enable PPI channel 1
    err_code = sd_ppi_channel_enable_set(PPI_CHEN_CH1_Msk);
    if(err_code != NRF_SUCCESS)
    {
        ASSERT(false);
```



```
    }
}

/**@brief TIMER1 Initialization.
 *
 * @details    Initialize TIMER1 module with all necessary
     configurations(Prescaler, CC0 value, etc.).
 *
 * Initializes Timer 1 peripheral, creates event and interrupt every 222us.
*/
static void timer1_init( void )
{
    // Configure timer1 to overflow every 222 ms
  // SysClk = 16Mhz
  // BITMODE = 16 bit
  // PRESCALER = 7
  // As
    // Clear TIMER1
    NRF_TIMER1->TASKS_CLEAR = 1;

    // Configure TIMER1 for compare[0] event every 222us
    NRF_TIMER1->PRESCALER = 7;          // Prescaler 7 results in 1 tick == 8
        us
    NRF_TIMER1->CC[ 0 ] = 0x0;
    NRF_TIMER1->CC[ 1 ] = COMPARE_VALUE_4k5Hz; //
    NRF_TIMER1->MODE = TIMER_MODE_MODE_Timer;
    NRF_TIMER1->BITMODE = TIMER_BITMODE_BITMODE_16Bit;
    NRF_TIMER1->SHORTS = ( TIMER_SHORTS_COMPARE1_CLEAR_Enabled <<
        TIMER_SHORTS_COMPARE1_CLEAR_Pos );
}

void sendChar_BLE(uint32_t pass, uint32_t counter){

    uint32_t err_code;
    // Send over BLE

    charact[0] = counter;
    charact[1] = counter >> 8;
    charact[2] = counter >> 16;
    charact[3] = counter >> 24;

    charact[4] = pass;
    charact[5] = pass >> 8;
    charact[6] = pass >> 16;
    charact[7] = pass >> 24;

    err_code = ble_sns_attr_adcVal_send(&m_sns, charact, sizeof(charact));
//  err_code = ble_sns_attr_adcVal_send(&m_sns, &periodCount,
    sizeof(periodCount));
    if( ( err_code != NRF_SUCCESS ) &&
        ( err_code != NRF_ERROR_INVALID_STATE ) &&
        ( err_code != BLE_ERROR_NO_TX_BUFFERS ) &&
        ( err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING ) )
    {
        ASSERT( false );
    }

}

/**
```

```
 * @brief Application main function
 */
int main( void )
{
    // Initialize
    apptimer_init();
    ble_stack_init();
    gap_params_init();
    services_init();
    advertising_init();
    conn_params_init();
    sec_params_init();
    gpio_init();

    initADC();

    actualState = IDLE_STATE;
    sm2_state = IDLE;
    periodCount = 0;

    timer1_init();  // using timer to generate events every 222us
    ppi_init();     // use ppi to redirect the event to timer start/stop tasks


//  gpiote_init();

    // Start execution
    advertising_start();
    nrf_gpio_pin_dir_set(GPIO_OUT_PIN5_Pos, 1);


    NRF_TIMER1->TASKS_START = 1;

        for(;;)
        {

            if(didFinishConversion)
            {
                 // ADAPTIVE COUNTING ALGORITHM
                counter++;

                if((old_adc_value > avg) && (adc_value < avg))
                {

                    pass++;

                    if(counter >= NB_SAMPLES)
                    {
                      sendChar_BLE(pass,counter);
                        pass = 0;

                        counter = 0;

                        didFinishConversion = false;

                    }
                    didFinishConversion = false;

                }
```



```
                }
                power_manage();
        }

}
```

8 ATTACHEMENT H: OBJECTIVE-C CODE

```
//
//  MainViewController.h
//  BLE_iPad
//
//  Created by Aurél on 10.06.14.
//  Copyright (c) 2014 Aurelien Merz. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <CoreBluetooth/CoreBluetooth.h>
#import "bitpacker.h"
#import "TransferService.h"
#import "timerConstants.h"
#import "samplingPeriods.h"

#define NB_SAMPLES 1000
#define SAMPLING_PERIOD 0.000250
#define N_AVG 100

/*****************************************
 *
 *****************************************/

#define E2_UPPER_LIMIT 84
#define E2_LOWER_LIMIT 80

#define A2_UPPER_LIMIT 112
#define A2_LOWER_LIMIT 108

#define D3_UPPER_LIMIT 148
#define D3_LOWER_LIMIT 144

#define G3_UPPER_LIMIT 198
#define G3_LOWER_LIMIT 190

#define B3_UPPER_LIMIT 248
#define B3_LOWER_LIMIT 245

#define E4_UPPER_LIMIT 331
#define E4_LOWER_LIMIT 327

/*****************************************
 *
 *****************************************/

#define DELTA_E2 12
#define DELTA_A2 15
#define DELTA_D3 15
#define DELTA_G3 9
#define DELTA_E4 11


/*****************************************
 *
 *****************************************/

#define E2_H_LIMIT 96
#define E2_L_LIMIT 92

#define A2_H_LIMIT 127
```



```
#define A2_L_LIMIT 120

#define D3_H_LIMIT 163
#define D3_L_LIMIT 160


#define G3_H_LIMIT 207
#define G3_L_LIMIT 202

#define E4_H_LIMIT 343
#define E4_L_LIMIT 340

/*****************************************
 *  SLIDERS MAXIMA
 *****************************************/

#define E2_SLIDER_UPPER_LIMIT 92
#define E2_SLIDER_LOWER_LIMIT 72

#define A2_SLIDER_UPPER_LIMIT 100
#define A2_SLIDER_LOWER_LIMIT 120

#define D3_SLIDER_UPPER_LIMIT 156
#define D3_SLIDER_LOWER_LIMIT 136

#define G3_SLIDER_UPPER_LIMIT 206
#define G3_SLIDER_LOWER_LIMIT 186

#define B3_SLIDER_UPPER_LIMIT 256
#define B3_SLIDER_LOWER_LIMIT 236

#define E4_SLIDER_UPPER_LIMIT 339
#define E4_SLIDER_LOWER_LIMIT 319

@interface MainViewController : UIViewController <CBCentralManagerDelegate,
    CBPeripheralDelegate>
{
    NSUUID *serviceUUID;
    NSUUID *charUUID;

}

@property (nonatomic, strong) NSString   *connected;
@property (nonatomic, strong) NSString   *dataADC;
@property (nonatomic, strong) NSString   *manufacturer;
@property (nonatomic, strong) NSString   *peripheralData;

// Instance method to get the heart rate BPM information
- (void) getADCData:(CBCharacteristic *)characteristic error:(NSError *)error;

// Instance methods to grab device Manufacturer Name, Body Location
- (void) getManufacturerName:(CBCharacteristic *)characteristic;

@end
```

```
//
//  MainViewController.m
//  BLE_iPad
//
//  Created by Aurél on 10.06.14.
//  Copyright (c) 2014 Aurelien Merz. All rights reserved.
//

#import "MainViewController.h"

#define UIColorFromRGB(rgbValue) [UIColor colorWithRed:((float)((rgbValue &
    0xFF0000) >> 16))/255.0 green:((float)((rgbValue & 0xFF00) >> 8))/255.0
    blue:((float)(rgbValue & 0xFF))/255.0 alpha:1.0]

@interface MainViewController ()

@property (strong, nonatomic) CBCentralManager *centralManager;
@property (strong, nonatomic) CBPeripheral *discoveredPeripheral;
@property (strong, nonatomic) NSMutableData *data;
@property (strong, nonatomic) CBCharacteristic *aChar;
@property (strong, nonatomic) CBCharacteristic *batteryChar;

@property(strong, nonatomic) NSData  *charVal;

@property(strong, nonatomic) NSMutableArray *samples;

@property (strong, nonatomic) IBOutlet UILabel *scanningState;
@property (strong, nonatomic) IBOutlet UIActivityIndicatorView *
    connectionActivity;
@property (strong, nonatomic) IBOutlet UILabel *foundPeripheral;
@property (strong, nonatomic) IBOutlet UILabel *connectionState;
@property (strong, nonatomic) IBOutlet UIActivityIndicatorView *
    scanningActivity;
@property (strong, nonatomic) IBOutlet UISwitch *connection_sw;
@property (strong, nonatomic) IBOutlet UILabel *measuredVal;
@property (strong, nonatomic) IBOutlet UILabel *amplitudeVal;
@property (strong, nonatomic) IBOutlet UILabel *batteryLevel;

@property (strong, nonatomic) IBOutlet UIBarButtonItem *sideBarButton;
@property (strong, nonatomic) IBOutlet UISlider *tuningValueSlider;

@property (strong, nonatomic) IBOutlet UILabel *note;
@property (strong, nonatomic) IBOutlet UISlider *sliderTone;
@property (strong, nonatomic) IBOutlet UILabel *sliderValue;

@end

@implementation MainViewController

NSData *adcValue;
NSData *val;
NSTimer *timer;
UIAlertView *scanningAlert;

NSMutableIndexSet *indexes;

uint8_t buffer[8];
float freq_avg = 0.0f;
Float32 count_avg = 0;
```



```
Float32 frequency = 0.0f;

uint16_t receivedFreq;

int n_avg = 0;
float actual_avg = 0;
uint16_t n = 0;


int nb_samples = 1000;
uint32_t pass;
uint32_t count;
float t_total = 0.0f;

typedef enum
{            E2 ,
             A2 ,
             D3 ,
             G3 ,
             B3 ,
             E4,
             NO_NOTE
}NOTE;

NOTE actualNote;

uint16_t periodCount = 0;

#pragma mark - View Lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];


    [self preferredStatusBarStyle];
    self.data = nil;

    self.view.backgroundColor = UIColorFromRGB(0x22313F);
    [self.sliderTone setThumbTintColor:UIColorFromRGB(0xc0392b)];

    [self setSliderMaximas:100.0:0];
    self.sliderTone.value = 50.0;

    // Start up the CBCentralManager

    _centralManager = [[CBCentralManager alloc] initWithDelegate:self queue:
       nil];

    //And somewhere to store the incoming data

    _data = [[NSMutableData alloc] init];
    _aChar =[[CBCharacteristic alloc] init];

    [_sliderTone addTarget:self action:@selector(sliderValueChanged:)
       forControlEvents:UIControlEventValueChanged];

    actualNote = NO_NOTE;
```

```
    _charVal = [[NSData alloc]init];

    t_total = nb_samples * t_4kHz;

}

- (IBAction)sliderValueChanged:(UISlider *)sender {
    [self.sliderValue setText:[NSString stringWithFormat:@"%f",[self.
       sliderTone value]]];
}

-(void)viewWillDisappear:(BOOL)animated
{
    // Don't keep it going while we're not showing
    //[self.centralManager stopScan];
    //NSLog(@"Scanning stopped");

    [super viewWillDisappear:animated];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.


}

-(UIStatusBarStyle)preferredStatusBarStyle
{
    return UIStatusBarStyleLightContent;
}

#pragma mark - Central Methods

/
    ****************************************************************************
    ******************************************
 *
    *
 *    SCANNING FOR PERIPHERALS FUNCTION
 *
 *    Scan for peripherals - specifically for our service's 128bit CBUUID
    *

    ****************************************************************************
    ******************************************/

- (void)scan
{
    [self.scanningActivity startAnimating];

    [self.centralManager scanForPeripheralsWithServices:@[[CBUUID
       UUIDWithString:TRANSFER_SERVICE_UUID]]
                                        options:@{
                                       CBCentralManagerScanOptionAllow
                                       DuplicatesKey : @YES }];
    //    [self.centralManager scanForPeripheralsWithServices:nil
    //
       options:@{ CBCentralManagerScanOptionAllowDuplicatesKey : @YES }];
```



```
    //

    [self.scanningState setText:@"Scanning for peripherals..."];


    [self.connectionActivity startAnimating];
    NSLog(@"Scanning started");
}
/
    ****************************************************************************
    ******************************************
 *
    *
 *    CENTRAL MANAGER DID UPDATE STATE
    *
 *
    *

    ****************************************************************************
    ******************************************/

-(void)centralManagerDidUpdateState:(CBCentralManager *)central
{


    // You should test all scenarios
    if ((central.state == CBCentralManagerStatePoweredOn)) {
        // Scan for devices, nil used for any devices
        NSLog(@"CoreBluetooth BLE hardware is powered on and ready");

        [self scan];
    }

    if(central.state == CBCentralManagerStateUnknown){
        NSLog(@"CoreBluetooth BLE state is unknown");
        return;
    }

    if(central.state == CBCentralManagerStateResetting){
        NSLog(@"Reset State");
        return;
    }

    if(central.state == CBCentralManagerStateUnsupported){
        NSLog(@"CoreBluetooth BLE hardware is unsupported on this platform");
        return;
    }

    if(central.state == CBCentralManagerStateUnauthorized)
    {
        NSLog(@"CoreBluetooth BLE state is unauthorized");
        return;
    }

    if(central.state == CBCentralManagerStatePoweredOff)
    {
        NSLog(@"CoreBluetooth BLE hardware is powered off");
        return;
    }
```

```
}
/
    ********************************************************************************
    ****************************************
 *  This callback comes whenever a peripheral that is advertising the
     TRANSFER_SERVICE_UUID is discovered.
 *  We check the RSSI, to make sure it's close enough that we're interested in
     it, and if it is, we start the connection
 *  process.
 *  This is called with the CBPeripheral class as its main input parameter.
 *  This contains most of the information there is to know about a BLE
     peripheral.

     ********************************************************************************
     ****************************************/


-(void)centralManager:(CBCentralManager *)central didDiscoverPeripheral:
    (CBPeripheral *)peripheral advertisementData:(NSDictionary *)
    advertisementData RSSI:(NSNumber *)RSSI
{
    // Reject any where the value is above reasonable range
    /* if(RSSI.integerValue > -15)
     {
     return;
     }

     // Reject if the signal strenght is too low to be close enough (Close is
         around -22dB)
     if(RSSI.integerValue < -35)
     {
     return;
     }*/

    //[self.connectionActivity stopAnimating];

    [scanningAlert dismissWithClickedButtonIndex:0 animated:YES];

    NSLog(@"Discovered %@ at %@", peripheral.name, RSSI);
    [self.scanningState setText:@"Discovered peripheral"];

    // Ok, it's in range - have we already seen it?
    if(self.discoveredPeripheral != peripheral)
    {
        // Save a local copy of the peripheral, so Corebluetooth doesn't get
            rid of it
        self.discoveredPeripheral = peripheral;

        // And connect
        NSLog(@"Connecting to peripheral %@", peripheral);
        [self.scanningState setText:@"Connecting to peripheral"];

        [self.centralManager connectPeripheral:peripheral options:nil];
    }
}


/** If the connection fails for whatever reason, we need to deal with it
 */
```



```
-(void)centralManager:(CBCentralManager *)central didFailToConnectPeripheral:
    (CBPeripheral *)peripheral error:(NSError *)error
{
    NSLog(@"Failed to connect to %@. (%@)", peripheral, [error
        localizedDescription]);
    [self.scanningState setText:@"Failed to connect to peripheral"];

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Connection
        failed!" message:@"Failed to connect to peripheral" delegate:nil
        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];

    [alert show];
    //[self.scanningActivity stopAnimating];
    [self cleanup];
}


/** We've connected to the peripheral, now we need to discover the services
    and characteristics to find the 'transfer2
 characteristic.
 // method called whenever you have successfully connected to the BLE
     peripheral
 */

-(void)centralManager:(CBCentralManager *)central didConnectPeripheral:
    (CBPeripheral *)peripheral
{
    NSLog(@"Peripheral connected");


    [self.scanningActivity stopAnimating];

    [self.connectionState setText:@"Connected"];

    self.connected = [NSString stringWithFormat:@"Connected: %@", peripheral.
        state == CBPeripheralStateConnected ? @"YES" : @"NO"];

    NSLog(@"%@", self.connected);

    // Stop scanning
    [self.centralManager stopScan];
    [self.scanningState setText:@"Scanning stopped!"];
    NSLog(@"Scanning stopped");

    // Clear the data that we may already have
    [self.data setLength:0];

    //Make sure we get the discovery call back
    peripheral.delegate = self;

    //Search only for services that match our UUID
    [peripheral discoverServices:@[[CBUUID UUIDWithString:
        TRANSFER_SERVICE_UUID]]];//,[CBUUID
        UUIDWithString:BLE_UUID_BATTERY_SERVICE]]];
    //[peripheral discoverServices:@[[CBUUID
        UUIDWithString:BLE_UUID_BATTERY_SERVICE]]];
}


/** The service was discovered
```

```
 */
-(void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:(NSError *)
    error
{
    if (error) {
        NSLog(@"Error discovering services: %@",[error localizedDescription]);
        [self cleanup];
        return;
    }

    //Discover the characteristic we want...
    // Loop through the newly filled peripheral.services array, just in case
       there's more than one.

    for (CBService *service in peripheral.services) {
        NSLog(@"Discovered service: %@", service.UUID);

        [peripheral discoverCharacteristics:@[[CBUUID UUIDWithString:
           TRANSFER_CHARACTERISTIC_UUID]]
                                 forService:service];

        //        [peripheral discoverCharacteristics:@[[CBUUID
           UUIDWithString:BLE_UUID_BATTERY_LEVEL_STATE_CHAR]]
        //                                 forService:service];

    }
}


/** The tranfer characteristic was discovered
 *  Once this found, we want to subscribe to it, which lets the peripheral
     know we want the data it contains
 */

-(void)peripheral:(CBPeripheral *)peripheral
    didDiscoverCharacteristicsForService:(CBService *)service error:(NSError *)
    error
{
    if(error){
        NSLog(@"Error discovering characteristics: %@", [error
           localizedDescription]);
        [self cleanup];
        return;
    }

    // Again we loop through the array, just in case
    for(CBCharacteristic *characteristic in service.characteristics)
    {
        // And check if it's the right one
        if([characteristic.UUID isEqual:[CBUUID UUIDWithString:
           TRANSFER_CHARACTERISTIC_UUID]])
        {
            //If it is, subscribe to it
            self.aChar = characteristic;
            [peripheral setNotifyValue:YES forCharacteristic:characteristic];


            NSLog(@"Subscribed to characteristic");
            //NSLog(@"Characteristic value: %@",[characteristic value]);
        }
```



```
    }

    // Once this is complete, we just need to wait for the data to come in.
}

// Invoked when you retrieve a specified characteristic's value, or when the
    peripheral device notifies your app that the characteristic's value has
    changed.
- (void)peripheral:(CBPeripheral *)peripheral didUpdateValueForCharacteristic:
    (CBCharacteristic *)characteristic error:(NSError *)error
{
    if (error) {
        NSLog(@"Error changing notification state: %@", error.
           localizedDescription);
    }
    //uint16_t freq = 0;
    //NSLog(@"Char Value: %@",[characteristic value]);



    [self extractByteFromNSData:[characteristic value]];




    t_total = count / 4500.0;
    frequency = pass / t_total;

    n_avg++;


    freq_avg += frequency;

    if (n_avg == N_AVG)
    {

        freq_avg /= n_avg;

        if((freq_avg > E2_L_LIMIT) && (freq_avg < E2_H_LIMIT))
        {
            [self setSliderMaximas:E2_SLIDER_UPPER_LIMIT :
              E2_SLIDER_LOWER_LIMIT];
            freq_avg -= DELTA_E2;
            [self.sliderTone setValue:freq_avg];
        }

        else if((freq_avg > A2_L_LIMIT) && (freq_avg < A2_H_LIMIT))
        {
            [self setSliderMaximas:A2_SLIDER_UPPER_LIMIT :
              A2_SLIDER_LOWER_LIMIT];
            freq_avg -= DELTA_A2 ;
            [self.sliderTone setValue:freq_avg];
        }
        else if((freq_avg > D3_L_LIMIT) && (freq_avg < D3_H_LIMIT))
        {
            [self setSliderMaximas:D3_SLIDER_UPPER_LIMIT :
              D3_SLIDER_LOWER_LIMIT];
            freq_avg -= DELTA_D3;
```

```
            [self.sliderTone setValue:freq_avg];
        }
        else if((freq_avg > G3_L_LIMIT) && (freq_avg < G3_H_LIMIT))
        {
            [self setSliderMaximas:G3_SLIDER_UPPER_LIMIT :
              G3_SLIDER_LOWER_LIMIT];
            freq_avg -= DELTA_G3;
            [self.sliderTone setValue:freq_avg];
        }
        else if((freq_avg > E4_L_LIMIT) && (freq_avg < E4_H_LIMIT))
        {
            [self setSliderMaximas:E4_SLIDER_UPPER_LIMIT :
              E4_SLIDER_LOWER_LIMIT];
            freq_avg -= DELTA_E4;
            [self.sliderTone setValue:freq_avg];
        }


        [self.measuredVal setText:[NSString stringWithFormat:@"%f",freq_avg]];


        [self tuning:freq_avg];
        [self printNote:actualNote];


        n_avg = 0;
    }


    // Notification has started
    if (characteristic.isNotifying) {

    }

}

- (void)peripheral:(CBPeripheral *)peripheral
    didUpdateNotificationStateForCharacteristic:(CBCharacteristic *)
    characteristic error:(NSError *)error{

    if (error) {
        NSLog(@"Error changing notification state: %@", error.
           localizedDescription);
    }

    // Notification has started
    if (characteristic.isNotifying) {

    }
}

/** Once the disconnection happens, we need to clean up our local copy of the
    peripheral
 */
-(void)centralManager:(CBCentralManager *)central didDisconnectPeripheral:
    (CBPeripheral *)peripheral error:(NSError *)error
{
    NSLog(@"Peripheral Disconnected");
    self.discoveredPeripheral = nil;
    [self.connectionState setText:@"Disconnected"];
```



```
    [self.measuredVal setText:@"" ];

    //We're disconnected, so start scanning again
    [self scan];
}



/** Call this when things either go wrong, or you're done with connection.
 *  This cancels any subscriptions if there are any, or straight disconnects
     if not.
 */

-(void)cleanup
{
    // Don't do anything if we're not connected
    if(self.discoveredPeripheral.state == 0)
    {
        return;
    }

    // See if we are subscribed to a characteristic on the peripheral
    if(self.discoveredPeripheral.services != nil)
    {
        for(CBService *service in self.discoveredPeripheral.services)
        {
            if (service.characteristics != nil)
            {
                for(CBCharacteristic *characteristic in service.
                  characteristics)
                {
                    if([characteristic.UUID isEqual:[CBUUID UUIDWithString:
                      TRANSFER_CHARACTERISTIC_UUID]])
                    {
                        if (characteristic.isNotifying) {
                            //It is notifying, so unsubscribe
                            [self.discoveredPeripheral setNotifyValue:NO
                                                          forCharacteristic:
                                                        characteristic];
                            // And we're done
                            return;
                        }
                    }
                }
            }
        }
    }
    // If we've got this far, we're connected, but we're not subscribed, so we
       just disconnect
    [self.centralManager cancelPeripheralConnection:self.discoveredPeripheral]
       ;
}



#pragma mark - CBCharacteristic helpers

// Instance method to get the heart rate BPM information
- (void) getADCData:(CBCharacteristic *)characteristic error:(NSError *)error
{
```

```
    NSData *adc = [characteristic value];
    //const uint8_t *reportData = [adc bytes];

    NSString *s;
    s = [NSString stringWithFormat:@"%@",adc];

    [self.measuredVal setText:s];

    //    if((characteristic.value) || !error)
    //    {
    //        NSLog(@"Characteristic Value is ok");
    //
    //    }
}



// Instance method to get the manufacturer name of the device
- (void) getManufacturerName:(CBCharacteristic *)characteristic
{
    NSString *manufacturerName = [[NSString alloc] initWithData:characteristic
       .value encoding:NSUTF8StringEncoding];  // 1
    self.manufacturer = [NSString stringWithFormat:@"Manufacturer: %@ :)",
       manufacturerName];    // 2
    return;
}

/
   ********************************************************************************
   ********************************************
 *  This function will extract the  data out of the bytes sent from the nRF51

    ********************************************************************************
    *******************************************/

-(void)extractByteFromNSData:(NSData *)data
{

    [data getBytes:&buffer length:8];

    count |= buffer[3];
    count = count << 24;
    count |= buffer[2];
    count |= count << 16;
    count |= buffer[1];
    count = count << 8;
    count |= buffer[0];

    pass |= buffer[7];
    pass = pass << 24;
    pass |= buffer[6];
    pass |= pass << 16;
    pass |= buffer[5];
    pass = pass << 8;
    pass |= buffer[4];

}
//-(void)extractCountFromNSData:(NSData *)data
//{
```



```
//
//
//    [data getBytes:&periodCount length:2];
//
//    NSLog(@"periodCount: %d", periodCount);
//}
//
//-(void)extractFrequencyFromNSData:(NSData *)data
//{
//    [data getBytes:&receivedFreq length:2];
//    //NSLog(@"Freq received: %d", receivedFreq);
//}


/*
 * This function determines the actual note
 */
-(void)tuning:(float)frequency
{
    if((frequency > E2_LOWER_LIMIT) && (frequency < E2_UPPER_LIMIT))
    {
        actualNote = E2;
    }

    else if((frequency > A2_LOWER_LIMIT) && (frequency < A2_UPPER_LIMIT))
    {
        actualNote = A2;
    }
    else if((frequency > D3_LOWER_LIMIT) && (frequency < D3_UPPER_LIMIT))
    {
        actualNote = D3;
    }
    else if((frequency > G3_LOWER_LIMIT) && (frequency < G3_UPPER_LIMIT))
    {
        actualNote = G3;
    }
    else if((frequency > B3_LOWER_LIMIT) && (frequency < B3_UPPER_LIMIT))
    {
        actualNote = B3;
    }
    else if((frequency > E4_LOWER_LIMIT) && (frequency < E4_UPPER_LIMIT))
    {
        actualNote = E4;
    }

}
// Displays the actual note on screen and update slider for visualisation
-(void)printNote:(NOTE)note
{
    switch (note) {

        case E2:
            [self.note setText:@"E2"];
            self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
            [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
           // NSLog(@"E2");
            break;

        case A2:
            [self.note setText:@"A2"];
```

```
                self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
                //NSLog(@"A2");
                break;

            case D3:
                [self.note setText:@"D3"];
                self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
                //NSLog(@"D3");
                break;

            case G3:
                [self.note setText:@"G3"];
                self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
                //NSLog(@"G3");
                break;

            case B3:
                [self.note setText:@"B3"];
                self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
                //NSLog(@"B3");
                break;

            case E4:
                [self.note setText:@"E4"];
                self.view.backgroundColor = UIColorFromRGB(0x2ecc71);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0x27ae60)];
                //NSLog(@"E4");
                break;

            default:
                [self.note setText:@""];
                self.view.backgroundColor = UIColorFromRGB(0x22313F);
                [self.sliderTone setThumbTintColor:UIColorFromRGB(0xc0392b)];
                break;
        }
    }

    -(void)setSliderMaximas:(float)max : (float)min
    {
        self.sliderTone.maximumValue = max;
        self.sliderTone.minimumValue = min;
    }

    @end
```

```
/**
 * Copyright (C) Hes-so VALAIS/WALLIS, HEI, Infotronics. 2014
 * Created by Aurélien Merz (merz.aurel@me.com)
 *
 * @file   timerConstants.h
 * @brief  nRF51822 Timer1 definitions of COMPARE VALUES for some sampling
    frequencies.
 *          TIMER MODE using prescalar of 7 making a tick every 8us.
 *         All values corresponds to the number of tick, you have to count to
    obtain the
 *         desired sampling frequency.
 *
 *
 * @author Aurélien MERZ
 * @version    1.0
 * @date   June 2014
 */

#define SAMPLING_FREQ 4000

#define COMPARE_VALUE_1kHz   0x7D
#define COMPARE_VALUE_1k5Hz  0x53
#define COMPARE_VALUE_2kHz   0x3E
#define COMPARE_VALUE_2k5Hz  0x32
#define COMPARE_VALUE_3kHz   0x29
#define COMPARE_VALUE_3k5Hz  0x24
#define COMPARE_VALUE_4kHz   0x1F
#define COMPARE_VALUE_4k5Hz  0x1C
#define COMPARE_VALUE_5kHz   0x29
#define COMPARE_VALUE_5k5Hz  0x16
#define COMPARE_VALUE_6kHz   0x15
#define COMPARE_VALUE_6k5Hz  0x13
#define COMPARE_VALUE_7kHz   0x11
#define COMPARE_VALUE_8kHz   0xD
#define COMPARE_VALUE_9kHz   0xC
#define COMPARE_VALUE_10kHz  0xD
#define COMPARE_VALUE_11kHz  0xB
#define COMPARE_VALUE_12kHz  0xA
```

9 ATTACHEMENT I: MCP601 DATASHEET

MCP601/1R/2/3/4

2.7V to 6.0V Single Supply CMOS Op Amps

Features

- Single-Supply: 2.7V to 6.0V
- Rail-to-Rail Output
- Input Range Includes Ground
- Gain Bandwidth Product: 2.8 MHz (typical)
- Unity-Gain Stable
- Low Quiescent Current: 230 µA/amplifier (typical)
- Chip Select ($\overline{CS}$): **MCP603 only**
- Temperature Ranges:
  - Industrial: -40°C to +85°C
  - Extended: -40°C to +125°C
- Available in Single, Dual, and Quad

Typical Applications

- Portable Equipment
- A/D Converter Driver
- Photo Diode Pre-amp
- Analog Filters
- Data Acquisition
- Notebooks and PDAs
- Sensor Interface

Available Tools

- SPICE Macro Models
- FilterLab® Software
- Mindi™ Simulation Tool
- MAPS (Microchip Advanced Part Selector)
- Analog Demonstration and Evaluation Boards
- Application Notes

Description

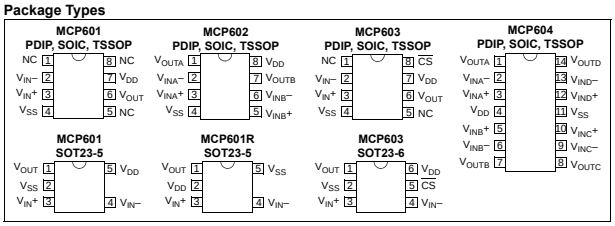
The Microchip Technology Inc. MCP601/1R/2/3/4 family of low-power operational amplifiers (op amps) are offered in single (MCP601), single with Chip Select ($\overline{CS}$) (MCP603), dual (MCP602), and quad (MCP604) configurations. These op amps utilize an advanced CMOS technology that provides low bias current, high-speed operation, high open-loop gain, and rail-to-rail output swing. This product offering operates with a single supply voltage that can be as low as 2.7V, while drawing 230 µA (typical) of quiescent current per amplifier. In addition, the common mode input voltage range goes 0.3V below ground, making these amplifiers ideal for single-supply operation.

These devices are appropriate for low power, battery operated circuits due to the low quiescent current, for A/D convert driver amplifiers because of their wide bandwidth or for anti-aliasing filters by virtue of their low input bias current.

The MCP601, MCP602, and MCP603 are available in standard 8-lead PDIP, SOIC, and TSSOP packages. The MCP601 and MCP601R are also available in a standard 5-lead SOT-23 package, while the MCP603 is available in a standard 6-lead SOT-23 package. The MCP604 is offered in standard 14-lead PDIP, SOIC, and TSSOP packages.

The MCP601/1R/2/3/4 family is available in the Industrial and Extended temperature ranges and has a power supply range of 2.7V to 6.0V.

Package Types



1.0 ELECTRICAL CHARACTERISTICS

Absolute Maximum Ratings †

$V_{DD} - V_{SS}$ 7.0V
Current at Input Pins .....±2 mA
Analog Inputs ($V_{IN}+$, $V_{IN}-$) †† ..... $V_{SS} - 1.0V$ to $V_{DD} + 1.0V$
All Other Inputs and Outputs ..... $V_{SS} - 0.3V$ to $V_{DD} + 0.3V$
Difference Input Voltage ..... $|V_{DD} - V_{SS}|$
Output Short Circuit Current .....Continuous
Current at Output and Supply Pins .....±30 mA
Storage Temperature.....–65°C to +150°C
Maximum Junction Temperature (T_J) .....+150°C
ESD Protection On All Pins (HBM; MM) ..... ≥ 3 kV; 200V

† Notice: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at those or any other conditions above those indicated in the operational listings of this specification is not implied. Exposure to maximum rating conditions for extended periods may affect device reliability.

†† See **Section 4.1.2 "Input Voltage and Current Limits"**.

DC CHARACTERISTICS

Electrical Specifications: Unless otherwise specified, T_A = +25°C, V_{DD} = +2.7V to +5.5V, V_{SS} = GND, $V_{CM} = V_{DD}/2$, $V_{OUT} = V_{DD}/2$, $V_L = V_{DD}/2$, and R_L = 100 kΩ to V_L, and $\overline{CS}$ is tied low. (Refer to Figure 1-2 and Figure 1-3).

Parameters	Sym	Min	Typ	Max	Units	Conditions
Input Offset						
Input Offset Voltage	V_{OS}	-2	±0.7	+2	mV	
Industrial Temperature	V_{OS}	-3	±1	+3	mV	T_A = -40°C to +85°C **(Note 1)**
Extended Temperature	V_{OS}	-4.5	±1	+4.5	mV	T_A = -40°C to +125°C **(Note 1)**
Input Offset Temperature Drift	$\Delta V_{OS}/\Delta T_A$	—	±2.5	—	µV/°C	T_A = -40°C to +125°C
Power Supply Rejection	PSRR	80	88	—	dB	V_{DD} = 2.7V to 5.5V
Input Current and Impedance						
Input Bias Current	I_B	—	1	—	pA	
Industrial Temperature	I_B	—	20	60	pA	T_A = +85°C **(Note 1)**
Extended Temperature	I_B	—	450	5000	pA	T_A = +125°C **(Note 1)**
Input Offset Current	I_{OS}	—	±1	—	pA	
Common Mode Input Impedance	Z_{CM}	—	10^{13}||6	—	Ω||pF	
Differential Input Impedance	Z_{DIFF}	—	10^{13}||3	—	Ω||pF	
Common Mode						
Common Mode Input Range	V_{CMR}	$V_{SS} - 0.3$	—	$V_{DD} - 1.2$	V	
Common Mode Rejection Ratio	CMRR	75	90	—	dB	V_{DD} = 5.0V, V_{CM} = -0.3V to 3.8V
Open-loop Gain						
DC Open-loop Gain (large signal)	A_{OL}	100	115	—	dB	R_L = 25 kΩ to V_L, V_{OUT} = 0.1V to V_{DD} – 0.1V
	A_{OL}	95	110	—	dB	R_L = 5 kΩ to V_L, V_{OUT} = 0.1V to V_{DD} – 0.1V
Output						
Maximum Output Voltage Swing	V_{OL}, V_{OH}	V_{SS} + 15	—	V_{DD} – 20	mV	R_L = 25 kΩ to V_L, Output overdrive = 0.5V
	V_{OL}, V_{OH}	V_{SS} + 45	—	V_{DD} – 60	mV	R_L = 5 kΩ to V_L, Output overdrive = 0.5V
Linear Output Voltage Swing	V_{OUT}	V_{SS} + 100	—	V_{DD} – 100	mV	R_L = 25 kΩ to V_L, A_{OL} ≥ 100 dB
	V_{OUT}	V_{SS} + 100	—	V_{DD} – 100	mV	R_L = 5 kΩ to V_L, A_{OL} ≥ 95 dB
Output Short Circuit Current	I_{SC}	—	±22	—	mA	V_{DD} = 5.5V
	I_{SC}	—	±12	—	mA	V_{DD} = 2.7V
Power Supply						
Supply Voltage	V_{DD}	2.7	—	6.0	V	**(Note 2)**
Quiescent Current per Amplifier	I_Q	—	230	325	µA	I_O = 0

Note 1: These specifications are not tested in either the SOT-23 or TSSOP packages with date codes older than YYWW = 0408. In these cases, the minimum and maximum values are by design and characterization only.

2: All parts with date codes November 2007 and later have been screened to ensure operation at V_{DD}=6.0V. However, the other minimum and maximum specifications are measured at 1.4V and/or 5.5V.

AC CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, T_A = +25°C, V_{DD} = +2.7V to +5.5V, V_{SS} = GND, V_{CM} = V_{DD}/2, V_{OUT} = V_{DD}/2, V_L = V_{DD}/2, and R_L = 100 kΩ to V_L, C_L = 50 pF, and $\overline{CS}$ is tied low. (Refer to Figure 1-2 and Figure 1-3).

Parameters	Sym	Min	Typ	Max	Units	Conditions
Frequency Response						
Gain Bandwidth Product	GBWP	—	2.8	—	MHz	
Phase Margin	PM	—	50	—	°	G = +1 V/V
Step Response						
Slew Rate	SR	—	2.3	—	V/µs	G = +1 V/V
Settling Time (0.01%)	t_{settle}	—	4.5	—	µs	G = +1 V/V, 3.8V step
Noise						
Input Noise Voltage	E_{ni}	—	7	—	μV_{P-P}	f = 0.1 Hz to 10 Hz
Input Noise Voltage Density	e_{ni}	—	29	—	nV/√Hz	f = 1 kHz
	e_{ni}	—	21	—	nV/√Hz	f = 10 kHz
Input Noise Current Density	i_{ni}	—	0.6	—	fA/√Hz	f = 1 kHz

MCP603 CHIP SELECT ($\overline{CS}$) CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, T_A = +25°C, V_{DD} = +2.7V to +5.5V, V_{SS} = GND, V_{CM} = V_{DD}/2, V_{OUT} = V_{DD}/2, V_L = V_{DD}/2, and R_L = 100 kΩ to V_L, C_L = 50 pF, and $\overline{CS}$ is tied low. (Refer to Figure 1-2 and Figure 1-3).

Parameters	Sym	Min	Typ	Max	Units	Conditions
$\overline{CS}$ Low Specifications						
$\overline{CS}$ Logic Threshold, Low	V_{IL}	V_{SS}	—	0.2 V_{DD}	V	
$\overline{CS}$ Input Current, Low	I_{CSL}	-1.0	—	—	µA	$\overline{CS}$ = 0.2V_{DD}
$\overline{CS}$ High Specifications						
$\overline{CS}$ Logic Threshold, High	V_{IH}	0.8 V_{DD}	—	V_{DD}	V	
$\overline{CS}$ Input Current, High	I_{CSH}	—	0.7	2.0	µA	$\overline{CS}$ = V_{DD}
Shutdown V_{SS} current	I_{Q_SHDN}	-2.0	-0.7	—	µA	$\overline{CS}$ = V_{DD}
Amplifier Output Leakage in Shutdown	I_{O_SHDN}	—	1	—	nA	
Timing						
$\overline{CS}$ Low to Amplifier Output Turn-on Time	t_{ON}	—	3.1	10	µs	$\overline{CS}$ ≤ 0.2V_{DD}, G = +1 V/V
$\overline{CS}$ High to Amplifier Output High-Z Time	t_{OFF}	—	100	—	ns	$\overline{CS}$ ≥ 0.8V_{DD}, G = +1 V/V, No load.
Hysteresis	V_{HYST}	—	0.4	—	V	V_{DD} = 5.0V

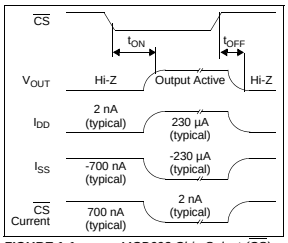


FIGURE 1-1: *MCP603 Chip Select ($\overline{CS}$) Timing Diagram.*

TEMPERATURE CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, V_{DD} = +2.7V to +5.5V and V_{SS} = GND.

Parameters	Sym	Min	Typ	Max	Units	Conditions
Temperature Ranges						
Specified Temperature Range	T_A	-40	—	+85	°C	Industrial temperature parts
	T_A	-40	—	+125	°C	Extended temperature parts
Operating Temperature Range	T_A	-40	—	+125	°C	**Note**
Storage Temperature Range	T_A	-65	—	+150	°C	
Thermal Package Resistances						
Thermal Resistance, 5L-SOT23	θ_{JA}	—	256	—	°C/W	
Thermal Resistance, 6L-SOT23	θ_{JA}	—	230	—	°C/W	
Thermal Resistance, 8L-PDIP	θ_{JA}	—	85	—	°C/W	
Thermal Resistance, 8L-SOIC	θ_{JA}	—	163	—	°C/W	
Thermal Resistance, 8L-TSSOP	θ_{JA}	—	124	—	°C/W	
Thermal Resistance, 14L-PDIP	θ_{JA}	—	70	—	°C/W	
Thermal Resistance, 14L-SOIC	θ_{JA}	—	120	—	°C/W	
Thermal Resistance, 14L-TSSOP	θ_{JA}	—	100	—	°C/W	

Note: The Industrial temperature parts operate over this extended range, but with reduced performance. The Extended temperature specs do not apply to Industrial temperature parts. In any case, the internal Junction temperature (T_J) must not exceed the absolute maximum specification of 150°C.

1.1 Test Circuits

The test circuits used for the DC and AC tests are shown in Figure 1-2 and Figure 1-2. The bypass capacitors are laid out according to the rules discussed in **Section 4.5 "Supply Bypass"**.

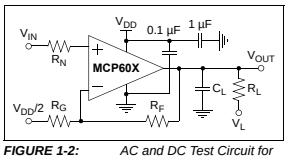


FIGURE 1-2: *AC and DC Test Circuit for Most Non-Inverting Gain Conditions.*

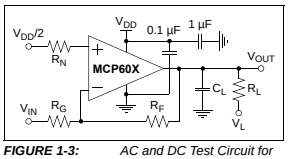


FIGURE 1-3: *AC and DC Test Circuit for Most Inverting Gain Conditions.*

10 ATTACHEMENT J: LM431Z DATASHEET

Shunt Regulator

General Description


TO-92


SOT-23


SOT-23-5


SOP-8


SOT-89

- The LM431 Series ICs are three-terminal programmable shunt regulators with guaranteed thermal stability over a full operation range. These monolithic ICs voltage reference operate as a low temperature coefficient Zener which is programmable from Vref to 36V with two external resistors. These devices exhibit a wide operating current range of 1.0 to 100mA with a typical dynamic impedance of 0.15 to 0.22Ω. The characteristics of these references make them excellent replacements for Zener diodes in many applications such as digital voltmeters, power supplies and op amp circuitry The 2.5V reference makes it convenient to obtain a stable reference from 5.0V logic supplies, and since the LM431 series operates as a shunt regulator, it can be used as either a positive or negative voltage reference.
- The output voltage of both types can be set to any value between V_{REF} (2.5V) and the corresponding maximum cathode voltage.

.

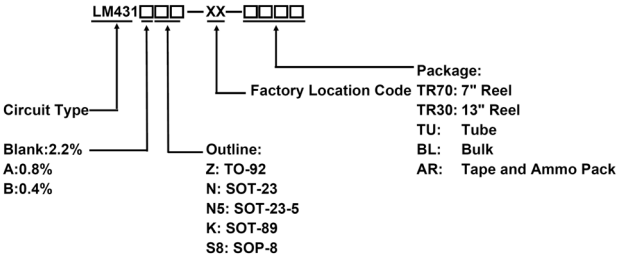
Features

- Programmable Precise Output Voltage from 2.5V to 36V
- High Stability under Capacitive Load
- Low Temperature Deviation
- Can be used in either positive or negative Voltage reference
- Low Dynamic Output Impedance: 0.15 to 0.22Ω Typical
- Operating Current from 1.0mA to 100mA
- Low Output Noise Voltage
- RoHS Compliance

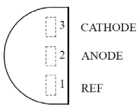
Applications

- Charger
- Voltage Adapter
- Switching Power Supply
- Graphic Card
- Precision Voltage Reference

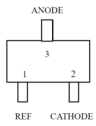
Ordering Information



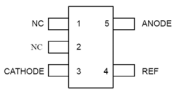
Pin Configuration



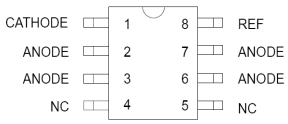
Outline: Z
TO-92



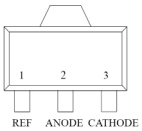
Outline: N
SOT-23



Outline: N5
SOT-23-5

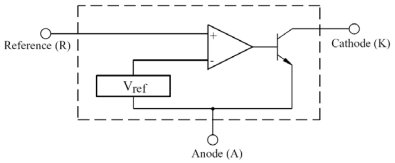


Outline: S8
SOP-8

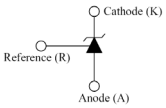


Outline: K
SOT-89

Block Diagram



Symbols



Absolute Maximum Ratings (Ta=25° C)

(Full operating ambient temperature range applies unless otherwise noted.)

Symbol	Description		LM431	LM431A	LM431B	Unit
V_{KA}	Cathode Voltage		36			V
I_{KA}	Cathode Current Range (Continuous)		-100 ~ 150			mA
I_{REF}	Reference Input Current Range (Continuous)		-0.05 ~ 10			mA
T_J	Operating Junction Temperature		150			° C
T_{OPR}	Operating Temperature Range		-40 ~ 85	-40 ~ 125		° C
T_{STG}	Storage Temperature Range		-65 ~ 150			° C
P_D	Power Dissipation	Z:TO-92	700	770		mW
		N:SOT-23	-	370		
		N5:SOT-23-5	-	370		
		K: SOT-89	800	770		
		S8: SOP-8	-	770		
Rth_{JA}	Package Thermal Impedance	Z:TO-92	180	150		° C/W
		N:SOT-23	-	330		
		N5:SOT-23-5	-	250		
		K: SOT-89	160	50		
		S8: SOP-8	-	150		

Electrical Characteristics *(T_A=25°C, unless otherwise specified)*

Symbol	Description		LM431 Min.	LM431 Typ.	LM431 Max.	LM431A Min.	LM431A Typ.	LM431A Max.	LM431B Min.	LM431B Typ.	LM431B Max.	Unit	Test Circuit	Conditions	
V_{REF}	Reference Input Voltage		2.440	2.495	2.550	2.480	2.500	2.520	2.490	2.500	2.510	V	Fig.1	V_{KA}=V_{REF},I_{KA}=10mA	
ΔV_{REF}	Reference Input Voltage Deviation	0-70° C	-	-	-	-	4.8	8	-	4.5	8	mV	Fig.1 (Note1)	V_{KA}=V_{REF},I_{KA}=10mA	
		-40-85 °C	-	7.0	30	-	4.5	10	-	4.5	10				
$\Delta V_{REF}/\Delta V_{KA}$	Ratio of Change in Reference Input Voltage to Change in Cathode to Anode Voltage		-	-1.4	-2.7	-	-1.0	-2.7	-	-1.0	-2.7	mV/V	Fig.2	IKA= 10mA	ΔV_{KA}=10V~ V_{REF}
			-	-1.0	-2.0	-	-0.5	-2.0	-	-0.5	-2.0				ΔV_{KA}=36V~ 10V
I_{REF}	Reference Input Current	Ta=25° C	-	1.8	4.0	-	0.7	4.0	-	0.7	4.0	μA	Fig.2	I_{KA}=10mA, R1=10KΩ, R2=∞	
		Ta=Topr	-	-	6.5	-	-	-	-	-	-				
ΔI_{REF}	Reference Input Current Deviation		-	0.8	2.5	-	0.4	1.2	-	0.4	1.2	μA	Fig.2	I_{KA}=10mA, R1=10KΩ, R2=∞, -40 ~ 85° C	
I_{MIN}	Min. Cathode Current For Regulation		-	0.5	1.0	-	0.4	1.0	-	0.4	1.0	mA	Fig.1	V_{KA}=V_{REF}	
I_{OFF}	Off-State Cathode Current		-	2.6	1000	-	50	1000	-	50	1000	nA	Fig.3	V_{KA}=36V, V_{REF}=0V	
Z_{KA}	Dynamic Impedance		-	0.22	-	-	0.15	0.5	-	0.15	0.5	Ω	Fig.1 (Note2)	V_{KA}=V_{REF}, I_{KA}=1.0~100mA, f≤1.0KHz	

Fig.1- Test Circuit for V_{KA}=V_{REF}

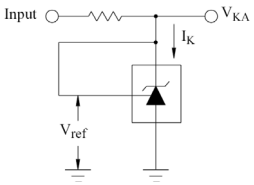


Fig.2- Test Circuit for V_{KA}>V_{REF}