POLITECNICO
MILANO 1863

EMG Signal Processing For Embedded Applications

Embedded Systems - Project Report

Github repo:
https://github.com/sorre97/STM32-EMGsensor

Author Sorrentino Alessandro - *CP. 10746269*

Supervisor Dr. Federico Terraneo

Year: 2020/2021

Contents

Abstract

Wearable devices are becoming more and more essential to regain a normal life. In particular, embedded medical applications are gaining more importance since they are used to help rehabilitation or to fulfill biological deficits such as limbs amputation. Not considering material cost, embedded applications regarding prosthesis are however still a big challenge and are still an open research topic because of limited resources of microcontrollers in contrast to the required computational power to elaborate and process biomedical signals. My work focuses on the data acquisition, data processing and actuation also considering response time and performance since having an almost real time application is crucial.

1 Introduction

In this section the main components of the project are presented.

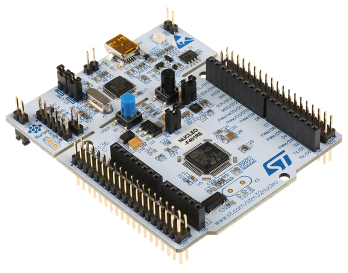
1.1 Miosix

Miosix[1] is an embedded OS kernel developed by Federico Terraneo designed to run on 32bit microcontrollers, especially on STM32, EFM32 and LPC2000 micrcontroller families. It is used for real-time operating applications and provides operative system abstraction layers such as single process, multi-threading and multiprocess (experimental). There is also support for the standard POSIX thread API (threads, mutexes and condition variables) other than full support for C and C++ standard libraries. The Miosix kernel is licensed under the GPL license.

1.2 Why Miosix?

This project heavily relies on multi-threading and on-board devices configurations such as ADC, external timer, analog pins etc. Not only Miosix provides definition of all registers that can be used to configure hardware on the fly, but also supports multi-threading natively, synchronization mechanisms between threads, abstracts USB communication interfaces (e.g. USART via `printf()`) and inter-thread communication mechanisms such as synchronized queues and variants. Another advantage is the small code size, fundamental for limited flash memory as in the case of most microcontrollers.

1.3 STM32 nucleo board



The microcontroller used for the project is the `STM32f401re nucleo` board. It mounts an Arm Cortex-M4 core operating at 84 MHz internal clock frequency. In addition, it has a floating point unit (FPU), a memory protection unit (MPU) and implements a full set of DSP instructions. This model of STM32 board incorporate high-speed embedded memories: 512 Kbytes of Flash memory and 96 Kbytes of SRAM. It also offers an up to 12-bit precision SAR ADC (with maximum clock frequency of 21Khz), six general-purpose 16-bit timers including one PWM timer for motor control and a set of ports with general I/O pins (either configurable in analog, digital or alternate mode). The rest of the specifications can be found in the relative datasheet or in the manufacturer website[2].

[1]See more at https://miosix.org
[2]ST website: https://www.st.com

1.4 EMG sensor

The Electromyography Sensor (EMG) allows the user to measure the electrical activity produced by skeletal muscles. It is composed by three electrodes: two of them responsible to acquire electrical potential difference from muscle electrical activity (red and green) and the third one as ground reference (yellow). The type of sensor used is a *surface EMG* (sEMG) that does not require intra-muscular insertion of electrodes and is, therefore, less invasive but it produces a noisier analog output.

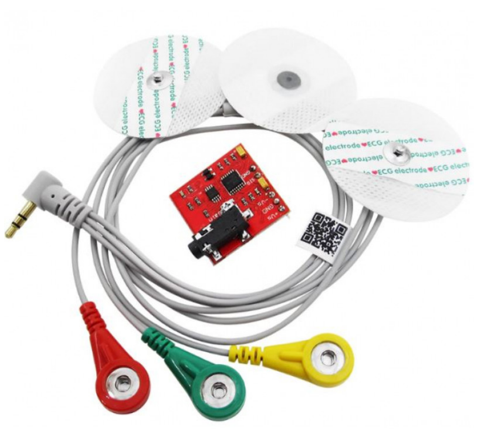


Figure 1: Surface EMG sensor

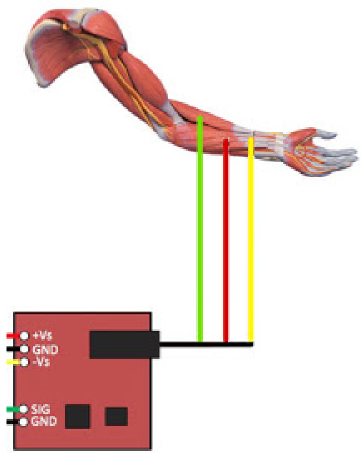


Figure 2: EMG connections

The sensor outputs an analog voltage proportionate to the muscular contraction. Since there is a slight electric potential difference between the two electrodes, a pre-stage amplifier is embedded in the sensor (which also amplifies noise that will be filtered via DSP later on).

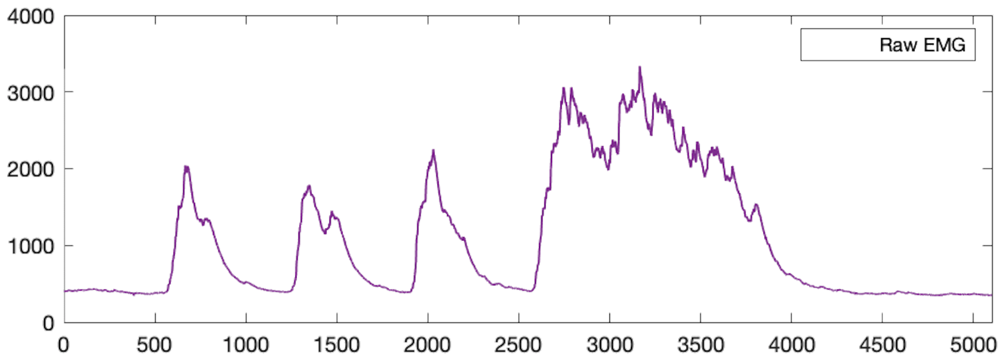


Figure 3: Digitalized output with 12-bit ADC (4095 max value)

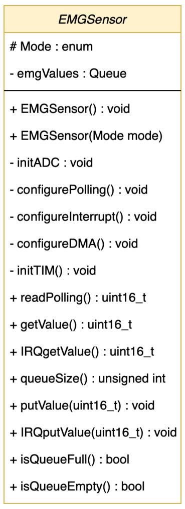
2 Application design and implementation

The designed embedded application aims to sample and process analog signal incoming from the EMG sensor, send processed data via serial interface (e.g. for live-plotting) and use these data to control movement of a prosthetic arm according to sensor response[3]. This application requires high parallelism among operations (sampling, data processing and data communication). The main requirement for the application is to be almost real time and therefore operations need to run in parallel achieving maximum throughput. Due to the aforementioned requisite, a precise balancing among operations is required. An high density of study cases and parameter exploration were taken into consideration to ensure best possible performance and optimization.

2.1 Hardware peripherals configuration

For the purpose of sampling and processing analog signals, hardware peripherals has to be properly configured. The input pin in which the sensor is connected is to be set in analog mode and the analog input signal redirected to the ADC input. The ADC needs must operate in interrupt mode and conversion triggered at a frequency such that the application does not become `interrupt-bound`. To achieve fully custom conversion rate and avoid dependency on ADC prescaling, an external (and prescaled) timer has also to be configured. The bottleneck on serial communication via USART interface has to be avoided and therefore the boudrate needs to be properly set up according to data processing speed and conversion rate. All those cases were studied, tested and the best configuration was chosen.

EMGSensor The EMGSensor class encapsulates all peripherals configuration (ADC, input port and TIM) and is responsible for their automatic initialization in the constructor. The constructor can take a parameter that defines the ADC conversion mode. The supported ADC conversion mode so fare are *polling* and *interrupt*. If no parameter is provided, the ADC is automatically configured in polling mode. The rest of the configuration parameters (TIM frequency etc) is hard-coded. Moreover, this class contains an internal synchronized queue from the Miosix `Queue` class) of dimension `100` and provides wrapper functions to retrieve or store converted samples, while granting multi-thread safeness. This internal queue can also be used in IRQ or when interrupts are disabled since wrapper IRQ methods of the Queue class were defined. Future development will allow configuring ADC in DMA mode and multiple parameter in the constructor such as TIM frequency, ADC resolution etc.



[3]See *Future development* section for more

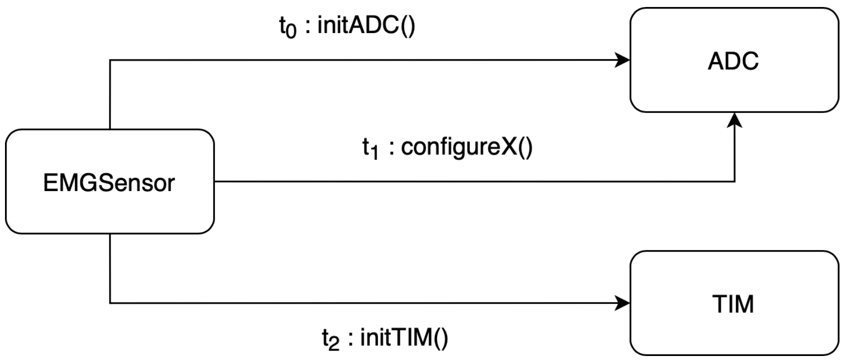


Figure 4: EMGSensor peripheral configuration (temporal order)

All peripheral configurations are detailed in the following sections.

ADC configuration The ADC can be configured in three different conversion modes: polling, interrupt (the one actually used in this project) and DMA (not currently supported). The EMGSensor class is responsible for the ADC initialization and configuration. During the initialization phase, the ADC is turned on, the input clock required to operate is enabled, the correct analog input channel is selected and the resolution bit is chosen (12-bit hard coded). In the configuration phase instead, all specific registers for a certain conversion mode are set. For the operation mode chosen in the application (interrupt), end-of-conversion interrupt is enabled, the conversion trigger depencency on external event is set and the interrupt is enabled in the `NVIC`. Each conversion is indeed triggered by an output event propagated by a timer. By doing that, we have full control on conversion frequency f_{smpl}. This was crucial since an elevate conversion rate, in interrupt mode, would make the system interrupt-bound and therefore every other threads would not have possibility to run since they would be continuously preempted by the interrupt handler. After the ADC configuration has finished, the external timer is configured.

TIM configuration The timer was configured to output an update event at a `1khz` rate and trigger the ADC conversion. The initial timer frequency was chosen to be half of the internal clock ($84Mhz$) with a x2 prescaler.

$$f_{tim} = \frac{f_{ck\_int}}{2} = 42Mhz$$

The frequency is furthermore prescaled with a x42 prescaler and the ARR register was set to 1000. The overall timer frequency becomes

$$f_{tim} = \frac{f_{ck\_int}}{2 \cdot 42 \cdot 1000} = 1000hz$$

USART The USART serial interface is known to be slow. Conversion and processing of samples could be much faster than sending data and may become a bottleneck. To mitigate this problem, the speed of transmission had to be increased. The boudrate was set to `230400` which was empirically proved to be efficient.

2.2 Multi-threading design

As explained in previous sections, parallelism is used to maximize the throughput and satisfy the real time requirement. The main components of the applications are

- DSP thread
- Serial communication thread
- ADC interrupt handler

Since all the main components of the application must synchronize and communicate with one another, synchronization mechanisms are mandatory. A particular consideration has to be made for IRQ since interrupt handler cannot block themselves and wait for a synchronization signal.

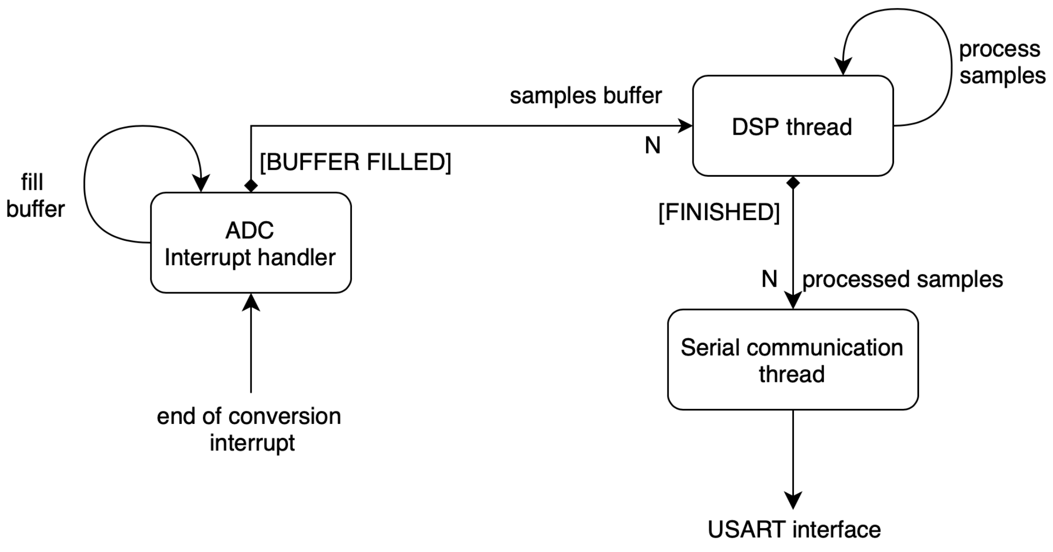


Figure 5: Relations between components

This graph however does not describe the relationship on synchronization and communication mechanisms. The main idea is to have the ADC interrupt to fill a buffer while the DSP thread processes a second buffer and the serial thread communicates data using a third buffer. All the presented mechanism are explained more in detail in the *section 2.3 Inter-thread-IRQ communication and synchronization.*

ADC Interrupt handler The ADC has a sample rate of 1000hz, which means 1000 samples per second. Every time a conversion is completed, the ADC triggers an interrupt via the EOCIE flag which is intercepted by the NVIC controller and an interrupt request is generated. That has as consequence the calling of a specific function in the interrupt vector table corresponding to the `ADC_IRQHandler`. The control is passed, after saving the context, to the real interrupt handler function called `ADC_IRQHandlerImpl`. The ADC interrupt handler stores converted samples into a static buffer and wakes up the DSP thread once the buffer is filled.

Serial communication thread This is the main thread and is also responsible to launch the DSP thread. The purpose of this thread is to continuously take processed values from buffers (if any is available) and send them via serial interface.

DSP thread The DSP thread is responsible for digitally filtering sample data coming from the ADC interrupt handler. After conversion, processed data are passed to the serial thread which will then send data via serial interface.

The incoming signal from the sensor is affected by noise. In particular, by analyzing the power spectrum of the incoming signal (blue) in the figure below, we can note that at DC frequency the 1/f noise is dominating and any signal appearing after 300hz clearly is noise (mostly thermal) since the band of a biomedical signal never goes beyond 60hz. There is also a 50hz noise component induced by the electric network. Therefore, we need a *bandpass* filter with [30hz, 300hz] range and a *stopband* at 50hz. All those noise sources need to be filtered out to clean sampled data and extract useful features from the signal[4].

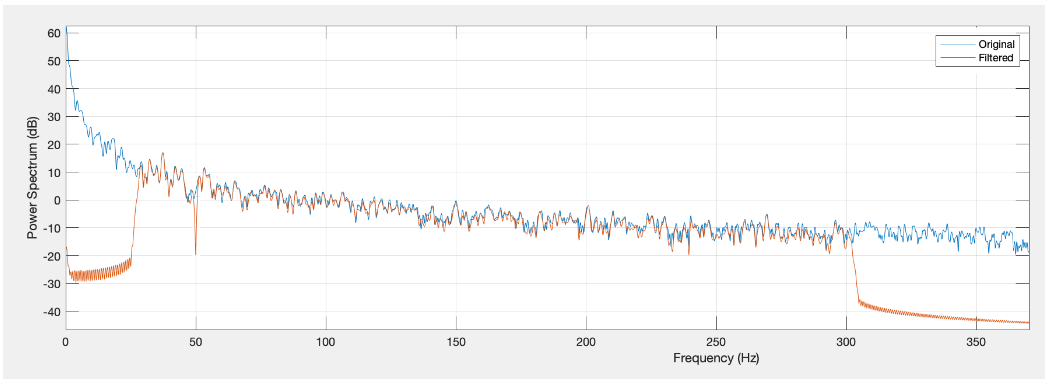


Figure 6: Power spectrum of signal before and after filtering

As we can see, after filtering appliance, our bandwith is aproximately restricted to [30hz, 300hz] with an high attenuation around 50hz.

The applied filtering makes use of two IIR filters (bandpass and stopband) and *zero-phase filtering* tecnhinque to remove filtering delay. Zero-phase filtering is a non-causal procedure, so it cannot be done in real time. That is why blocks of N samples are converted each time making a piece-wise filtering on the whole incoming signal. Because of that, filter's initial conditions was necessary to avoid DC noise on interval borders. Filtering is then performed twice for each filtering, one back and one forth inverting the signal on second pass. In the end, N samples are filtered 4 times. Twice for the two filters (bandpass and stopband).
The general form of an IIR discrete filter is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N} b_i \cdot z^{-1}}{1 - \sum_{i=1}^{N} a_i \cdot z^{-1}}$$

[4]See section 3.1 (Future development) for more details

$$y[t] = b[0] \cdot x[t] + b[1] \cdot x[t-1] + ... - a[1] \cdot y[t-1] - a[2] \cdot y[t-2] - ...$$

where N is the order of the filter, a_i are denumerators coefficient and b_i are numerator coefficients. The bandpass and stopband filters were designed using Matlab$^{\text{TM}}$

The bandpass filter is a 7_{th} order filter with coefficients

$$a_{coeff} = \begin{bmatrix} 1 \\ -2.13183455555828 \\ 1.47978011393210 \\ -0.679740843101842 \\ 0.584825906895303 \\ -0.218461835750097 \\ -0.0211926261278646 \end{bmatrix} \qquad b_{coeff} = \begin{bmatrix} 0.199880906801133 \\ 0 \\ -0.599642720403399 \\ 0 \\ 0.599642720403399 \\ 0 \\ -0.199880906801133 \end{bmatrix}$$

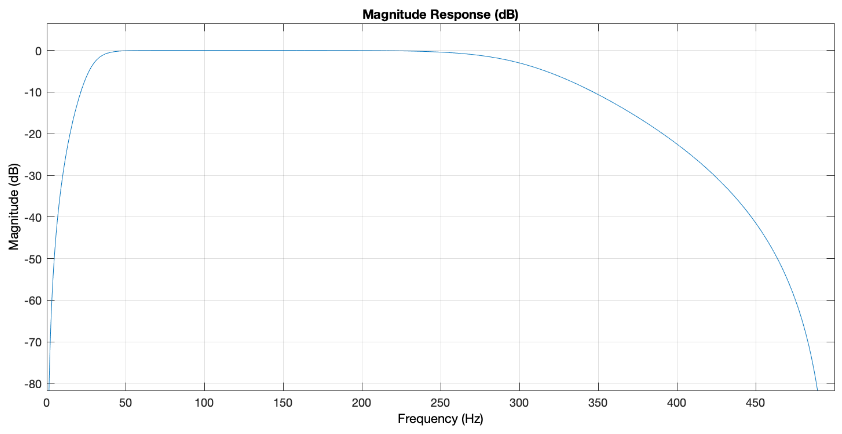


Figure 7: Transfer function of bandpass filter

This transfer function leaves untouched frequencies between 30hz and 300hz while cuts frequencies outside that interval attenuating by over 80db.

The stopband filter is instead a 5_{th} order filter with coefficients

$$a_{coeff} = \begin{bmatrix} 1 \\ 3.78739953308251 \\ 5.56839789935512 \\ -3.75389400776205 \\ 0.982385450614124 \end{bmatrix} \qquad b_{coeff} = \begin{bmatrix} 0.991153595101663 \\ -3.77064677042227 \\ 5.56847615976590 \\ 3.77064677042227 \\ 0.991153595101663 \end{bmatrix}$$

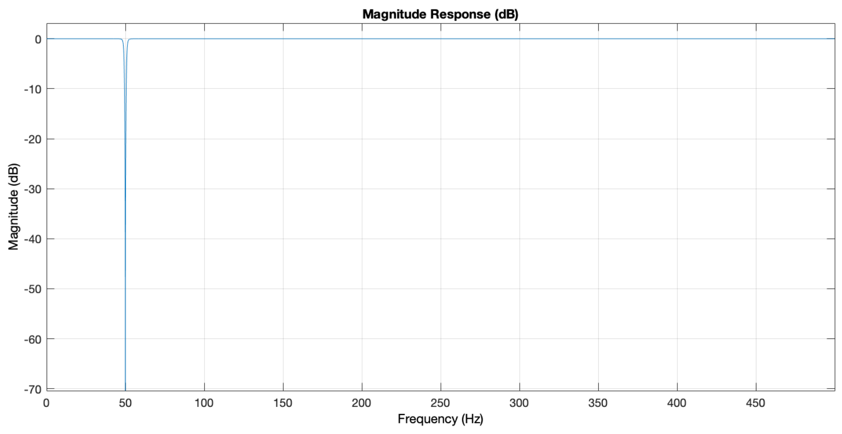


Figure 8: Transfer function of stopband filter

This transfer function cuts frequencies in a small interval around 50hz attenuating it by over 80db while leaves untouched the rest.

The comparison between filtered and unfiltered signal is reported below (mind the different Y scale of the two signals). Note that DSP cannot however solve aliasing which requires analog filtering the signal entering into the ADC.

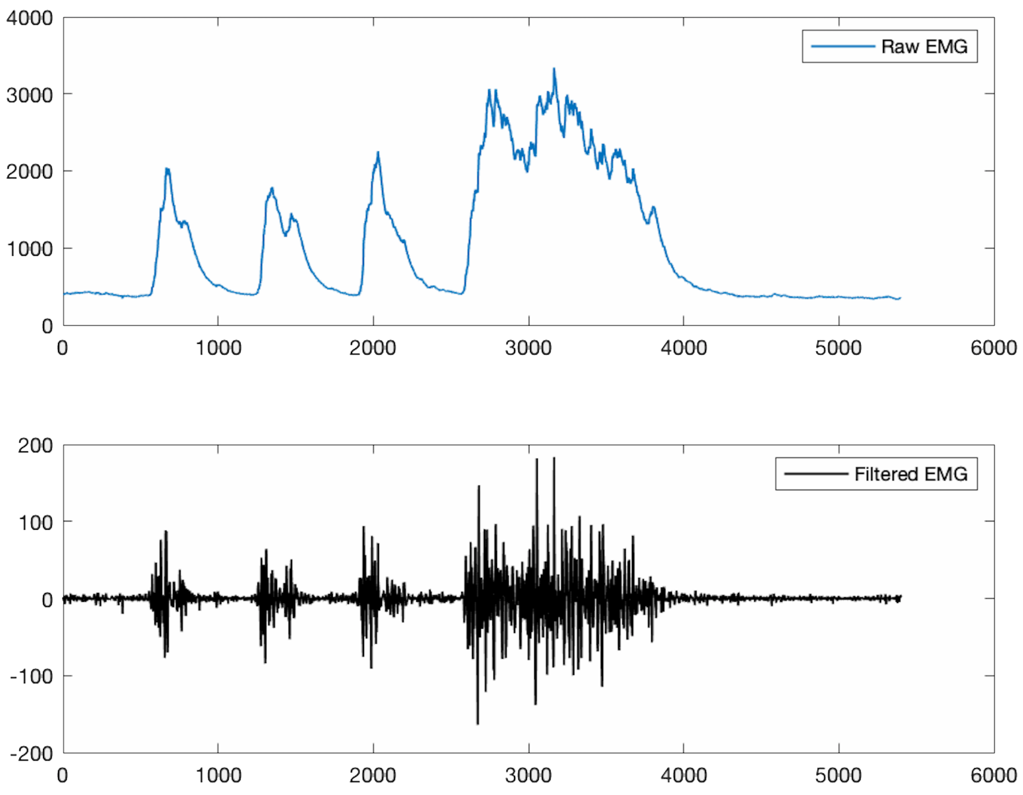


Figure 9: Signal before and after DSP filtering

After processing the data, they are inserted into a buffer and passed to the serial communication thread.

2.3 Inter-thread-IRQ communication and synchronization

All the three main components of the application require a way to exchange values and synchronize. The Miosix `BufferQueue` class was used to instantiate a set of shared buffer to be written or read in order to exchange data. The main goal is to obtain *software pipelining.* While the ADC interrupt handler fills the N_{th} buffer, the DSP is processing data of the $(N+1)_{th}$ buffer while the serial thread is reading data from the $(N+2)_{th}$ buffer. This is translated with some delay Δt from which samples are converted to the time data is transferred to the computer. That is why a precise tuning of various parameter is fundamental. In particular, because of various configurations applied it has been proven empirically that

$$\Delta t \simeq 300ms$$

which is enough to be considered real time for the suited application.

BufferQueue class This class encapsulates methods to handle in reading and writing mode N buffers. Given N initially empty buffer, a writer can request an empty buffer to the queue and fill it. It is then mark as filled. A consumer can require a filled buffer from the queue in order to read it, and mark it as emptied. It is possible then to implement consumer-producer paradigm with N shared buffers. However, it does not provide synchronization mechanisms. If required, those cases must be handled with mutex locks and condition variables (except for IRQs). However, note that in the following implementation none of the three components access the same buffer pool in the same operational mode (read or write).

ADC_IRQ $\Longleftrightarrow$ DSP thread First synchronization mechanisms to consider is the one between the interrupt routine and the thread that performs DSP. Standard mutexes and condition variables can neither be used since IRQ methods nor block themselves (would cause a deadlock). A set of 5 buffers of size 500 are shared among those two components. The ADC_IRQ fills a buffer with converted samples while the DSP is in a IRQ waiting state until at least one buffer is made available for reading. As soon as a buffer has been filled, the DSP thread is woken up using IRQ compliant methods and data is processed. Since IRQs cannot block, the ADC_IRQ simply does not copy sampled values to the shared buffer and exits from the interrupt routine if no buffers are available. DSP thread is a bottleneck in this application so a writable buffer could be made available after a certain time. Because of that, it may happen that the ADC_IRQ passes old samples to the DSP thread by the a time writable buffer is made available. Having at least 5 shared buffers between the two components mitigates this problem.

Please note: the following diagram focuses more on the interaction between ADC interrupt handler and DSP thread. Therefore, it does not keep into consideration synchronization between the DSP and the serial communication thread.

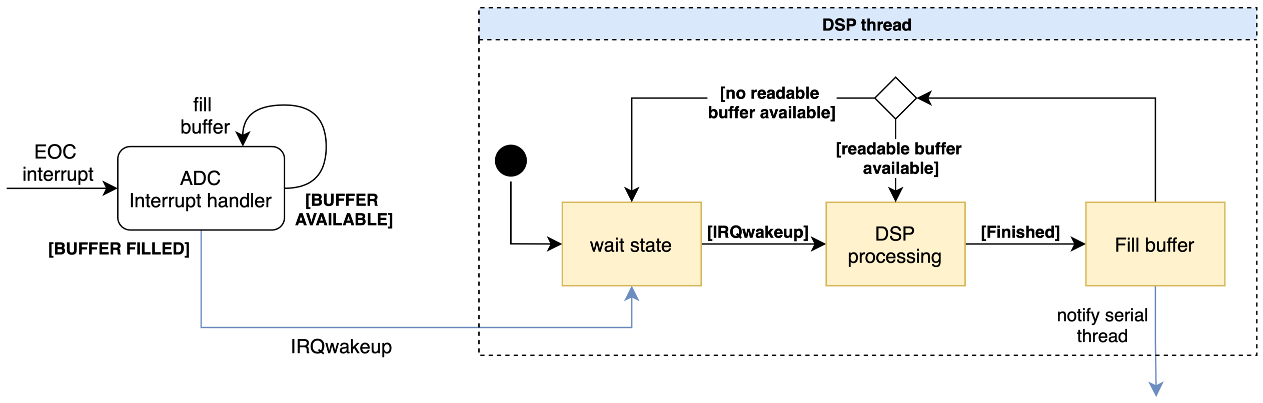


Figure 10: ADC_IRQ and DSP thread communication

DSP thread $\Longleftrightarrow$ Serial thread Once data is processed, it is passed to the serial thread and sent via serial interface. In order to share samples between those two components, a second queue of buffers is used. In particular, there are 3 buffers at the dimension of 300. Those two components are threads, so synchronization can be handled using conditional variables and mutex locks (both threads access shared resouces using a mutex lock which will be implied from now on). At the beginning all buffers are empty and, therefore, available for writing, but not for reading. The serial thread goes in a wait state on the *consumer condition variable* and waits for at least one buffer available for reading. Once the DSP has filled at least one buffer, it wakes up the serial thread that is now able to send data to the computer. If there is no available buffer for writing, the DSP thread goes in a wait state on the *produces condition variable* because the serial thread still has to communicate all the converted samples. Once the serial thread has made available for writing at least one buffer, the DSP thread is woken up by the serial thread and can so fill the new processed values into the new available buffer. The problem of latency, as seen before, does not concern this case because the serial thread is faster than the DSP. It rarely happens that the DSP waits for available writable buffer. It happens more frequently, however, that the serial thread waits for available readable buffer.

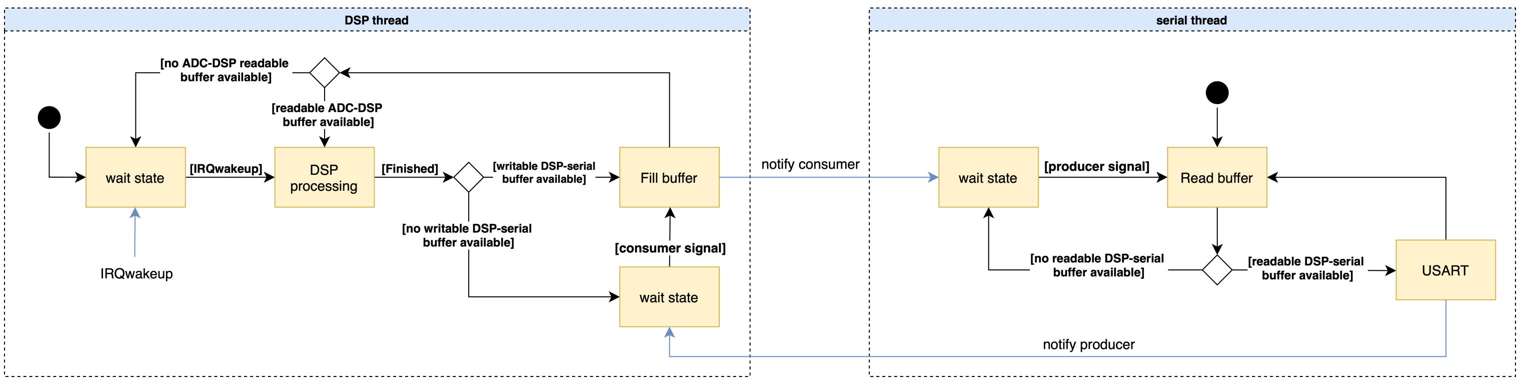

DSP thread
[no ADC-DSP readable buffer available]
[readable ADC-DSP buffer available]
wait state
[IRQwakeup]
DSP processing
[Finished]
[writable DSP-serial buffer available]
Fill buffer
[no writable DSP-serial buffer available]
[consumer signal]
wait state
IRQwakeup
notify consumer
serial thread
wait state
[producer signal]
Read buffer
[no readable DSP-serial buffer available]
[readable DSP-serial buffer available]
USART
notify producer

Figure 11: DSP and serial thread communication

3 Conclusions

3.1 Future development

Due to lack of time, not everything was implemented. However, here follows a few possible future extensions ideas.

Machine learning Once data are cleaned via DSP, they can used to perform pattern recognition which aims to understand the type of motion that the user is intending to accomplish according to the characteristics of the EMG signal (such as pointing the index finger, closing hand, grabbing movements etc.). In order to extract information from the signal and classify the type of movement, machine learning would be a good idea.

Possible paradigms The first approach is to use plain machine learning. There are simple but efficient methods such as KNN or SVM to perform movement classification. Also neural networks if features are hand-crafted (to remain on the plain machine learning paradigm). The possible features to consider when having an EMG signal are:

- Amplitude of the first burst (AFB)
- Difference absolute mean value (DAMV)
- Difference absolute standard deviation value (DASDV)
- Difference log detector (DLD)
- Difference temporal moment (DTM)

  etc...

The second approach is to use deep neural networks. Neural networks have the capability of extracting best representative features from the dataset and obtain very high accuracy. The problem is that deep learning techniques are very data hungry and the size of the network tends to grow up quickly. However, the ST company has published the *X-CUBE-AI* tool to convert pre-trained neural networks into C code while performing compression to obtain small in size code and fast speed. Once processed, the network can be used for an embedded application.

Prosthesis control After the classification of movement using machine learning, a prosthesis can be controlled in order to replicate the movement intended by the patient. A preset of offset angles could be applied to some stepper encoder motors linked to some 3D-printed hand structure. Motors applied rotation angle can either be intended as finger junctions rotations or as actuators to pull tendons-like strings and make the hand assume the wanted pose.