**Project documentation**

**Emergency call button**

Date: **14/02-11**

Advisor: Kim Bjerge

Project participants: Jørgen Vrou Hansen (20042728)

Saiid Shah Alizadeh (19963592)

Anders Hvidgaard Poder (19951439)

Revision: **A**

Document ID: **PRODDOC**

# Introduction

This document details the process, decisions and result of the Emergency button project. A prerequisite for reading this document is to have read the project proposal and the one-page intro.

The document begins by describing the process and then continues to describe the individual parts of the project as it progresses through the process. Naturally the process is iterative in nature, yet in the document any changes will be described under the section they relate to, and the document may therefore not be seen as a chronological detail of the process. Where changes occurs the point of origin will naturally be described along with its affect to earlier section in the change log.

# Process

1. Analyse project proposal and create SRD
   1. Domain model
   2. System level requirements
      1. Use cases.
      2. Sequence diagrams where needed.
      3. Non-functional requirements.
2. Refine SRD to SRS
   1. State and activity diagrams to clarify use cases.
   2. Detailed requirements from system requirements – Should we do this?
   3. Requirement traceability from system to detailed requirements.
   4. Traceability for changed requirements.
3. Overall architectural design
   1. Identify blocks and create overall structure
   2. Mapping of blocks to requirements, both in diagram and RVTM (V is not part of report)
   3. Create internal block diagram for important blocks.
   4. Create activity, state, sequence and other diagrams where needed.
   5. Modify requirements if needed.

3a. SySMl for SoC and design and SystemC transformation

* 1. Relate SySML block to SystemC module
  2. SystemC notation on SysMl diagrams
  3. Partitioning SysML blocks into 1 or more SystemC modules.
  4. Describe the process, important considerations

1. SystemC TLM of overall architectural design
   1. Map functional blocks to SystemC module.
   2. Create communication channels (mostly standard FIFO).
   3. Modify architecture/requirements if needed.
2. Architecture mapping
   1. Identify alternative architectures.
   2. Create architectural design for each alternative.
3. Process mapping for each alternative architecture
   1. Identify processes.
   2. Identify communication.
   3. Map Processes to PE and Communication channels to CE
4. SystemC Timed TLM for each alternative architectures
   1. Update and refine SystemC to the alternative architecture
   2. Identify delayes in the proceses and communication channel based on rough estimation.
   3. Implement delays in SystemC.
   4. Simulate the system and compare the results.
5. Conclusion
   1. Evaluate the pros and cons of the alternative architectures.
   2. Evaluate the process.

# Domain model



Figure 1 - Domain model drawing

In Figure 1 may be seen the overall parts of the Emergency call system including their communication paths. Only a small part of the above is part of this project, yet it is important to realize the domain in which the project operates.

The system functions as follows:

1. An emergency call base is installed in the home of the person receiving care and an emergency call button is issued to that person.
2. The Emergency button communicates with the emergency call base using the ISM network.
3. The emergency call base communicates with the phone company server using its built-in GSM modem and SIM card via a Base Tranceiver Station (BTS).
4. The phone company server forwards the communication via the internet to/from the emergency call server (naturally in a properly protected tunnel).
5. The emergency call server forwards the communication to/from the interested parties; technician, care giver or both.

The above may be formalized in a SysML Domain Basic Block Diagram as shown in Figure 2.



Figure 2 - Emergency call system Domain

In Figure 2 may be seen not only the System Of Interest, but also the environment in which it operates and the blocks with which it interconnects.

The environment in which the emergency call button operates is one where it may be operated by an elderly or disabled person, and may therefore be exposed to some degree of moisture and shocks, as well as worn continuously for a long time.

The ISM network naturally suffers from limited range, noise and interference, which is all part of the ISM network.

The Emergency call base is the communication gateway between the Emergency call server and the Emergency call button. It also has other responsibilities, yet they are irrelevant for this project. The communication goes via the GSM network, which also suffers from limited range, noise and interference as well as a third party service provider (the phone company). In the target area it is believed that with a sufficiently large and well placed GSM antenna a sufficient signal may always be achieved (assumption). Furthermore the GSM network has been thoroughly tested and the service providers have strong incentives to keep a very high availability.

The Emergency call server is placed in a protected environment with sufficient power and network access, and can reach the technician and caregiver on duty using either the head-office LAN , SMS Gateways and/or other third party services.

Focussing on the System-Of-Interest a set of requirements may be created from the project proposal. As the focus on this project is proof-of-concept some of the detail requirements (e.g. colour, frame IP rating, …) will be excluded just like only the critical requirements, especially those containing implemtational risks, will be broken down in the architecture.

# Requirements

This section contains both the overall system requirements (SRD) and the refined requirements (SRS) in one, as the project is an in-house project with close proximity to the “customer”. It therefore does not make sense to create two separate documents, but is far more efficient to simply create the SRS right away.

Again the requirements are only the system requirements and do not include all detailed requirements, yet should contain sufficient data to complete a proof-of-concept.

Before we dive into the creation of the requirements we should look at the Quality attributes of importance to this system, as well as the challenges in meeting these requirements.

## Quality attributes

There are several schools of system architecture, many of which define their own set of quality attributes. A well known and recognized set is the one defined by [BASS] as follow:

* Availability
* Modifiability
* Performance
* Security
* Testability
* Usability

Many add an extra quality attribute:

* Safety

And finally there are some business oriented attributes:

* Time to market
* Cost

Naturally all of these are important, but some are more important than others, and some are also more likely to cause trouble. It is therefore important to include a risk assessment in the quality attribute analysis.

Naturally Safety is vitally important, but at the same time as this product focuses almost entirely on communication, most of the safety aspect is contained in Availability, and the only true Safety aspect is pinching ones finger in the button, scratching oneself on the frame or the battery blowing up. Looking at cell phone manufacturers the last safety aspect is actually a real one, especially as the emergency call button might get damp and warn close to the skin. We do however believe that the current EU regulations cover these aspects, and that sufficient experience exists to avoid this problem, and the risk of violating requirements for the safety attribute is low.

Second come Availability. The button must “always” be able to report an emergency (including battery low situations). Naturally there is no such thing as “always”, so an acceptable probability of failure must be agreed to. In the present system there is no fault detection, but the “buttons” are made so simple that a very high Mean-Time-Between-Failure (MTBF) can be achieved. In this must also be included the probability of a failure later in the communication path, but as these units are on a continuous power supply they can maintain a much higher fault detection frequency. It is believed that the probability of mechanical failure (the button gets stuck or the micro switch fails) can be considered negligible, making the state of the microcontroller a sufficiently deep fault detection.

Usability is also vitally important. Here the experience from existing system may be used to determine the optimal size, colour, location and feel of the button itself. Charging the emergency call button is a much more interesting aspect, as the existing “buttons” very rarely needs to have their battery replaced.

Testability is interesting in the sense that it is important to verify especially the Availability requirements, so it should be considered when writing the test cases and also when implementing the product.

Naturally a certain level of Performance is required, but the response times that is expected lies well below the technical capabilities of the technology available, and the risks here are therefore minor.

Modifiability is more of a nice to have as the cost of the individual emergency call button is such that the entire button may simply be replaced in lieu of updating the firmware, and also gaining access to the buttons themselves are relatively simple, as central lists of their whereabouts are kept at the municipalities.

Finally there is Security, and for this project it is not important. No confidential information should be exchanged on this medium and the desire for someone to want to impersonate an emergency call button is most likely quite small. Naturally the emergency call buttons must be distinguishable from each other, but protection against someone intentionally trying to spoof an emergency call button is not required.

From the above it is possible to create a prioritised list of quality attributes along with their estimated level of complexity (and thereby risk).

|  |  |  |  |
| --- | --- | --- | --- |
| Priority | Quality attribute | Complexity (0 – 10, where 10 is highest complexity, i.e. highest risk) | Flexibility (0 – 10, where 10 is highest flexibility) |
| 1 | Safety | 1 | 0 |
| 2 | Availability | 8 | 3 |
| 3 | Usability | 3 | 2 |
| 4 | Testability | 4 | 8 |
| 5 | Performance | 4 | 5 |

The reason it is important to include the complexity of achieving the required level and the allowed flexibility, is so it is possible to know where there may be some leeway. If the flexibility is 0 it is not even worth looking at, as there are (most likely) strong legislative reasons why this quality attribute must be met to the exact specification. Naturally the complexity and flexibility is a rough estimate based on the developers and architects understanding of the task at hand, and may change as the project progresses, yet if there is a high degree of uncertainty it might be a good idea to reduce this uncertainty before progressing, or at least address this module first in the further development.

When these priorities are in place it is possible to address the business side of the quality attributes. Many newer schools of development argue that you cannot (or at least should not) change the quality, only adjust the number of features included to meet the time and cost schedule. This is often illustrated by the triangle shown in Figure 3, often called the iron triangle because it is possible (ideally) to control two sides of the triangle, but never three. So if you are willing to pay anything, you can have all the features in a very short amount of time, but if you are not willing to pay very much, and you want it yesterday, then you cannot have very many features.

The problem with this triangle is that it does not take non-functional requirements into account, like many of the quality attributes are. The reason this is important is that it may also be possible to save significant money and time if the client is willing to lower the availability requirement, without touching the functional requirements (the features). This is where the complexity comes into play, as it shows how much there might be potentially gained in time and money by adjusting these parameters. This should then be combined with the flexibility, to gain an impression not only on how much is there to be gained, but also how willing (able) is the client to make changes to these requirements.



Figure 3 - Iron triangle

## Rationale

Diving into the actual requirements we can derive a lot of the information directly from the project proposal, current legislation, existing solutions, and some from the specification of the emergency call base. Other requirements, like the maximum allowed time between failure detection, are derived from a combination of risk assessment and acceptable delays as indicated by the caregivers. By estimating a mean time between failures (MTBF) and an acceptable response delay and failure probability it is possible to calculate the required fault detection interval in order to meet the indicated fault probability figure.

This calculation can be done as follow:

* MTBF = 365 days.
* Acceptable extra delay = 30 minutes (a total response time of 60 minutes).
* Acceptable probability that the +30 minutes requirement is not met = 0.1%.

The probability that the error occurs in the time more than 30 minutes from a fault detection is (X – 30) / (365 \* 24\* 60) => (X – 30) / 525600, where X is the interval between fault detection. Combining this with the required probability gives (X – 30) / 525600 = 0.001 => X – 30 = 525.6 => X = 555.6 minutes, or a fault detection every 9 hours, and the2 hours is therefore acceptable.

Naturally the probability that one unit in a collection of units do not meet the above specifications is different (relative to the size of the collection).

## Content

Please refer to [REQSPEC] for the actual requirements.

TODO: Should we insert the requirements here, or refer to the document?

# Architecture

The architecture of the system can be separated into two parts

1. General architecture
2. Mapping specific architecture

The general architecture is characterised by being independent on mapping to HW and SW, where the mapping specific architecture is dependent on selecting the platform on which the general architecture must exist. The Mapping specific architecture is a refinement of the general architecture, and must therefore not be in conflict with the general architecture.

## General architecture

In order to define the general architecture the individual blocks that make up the system of interest must be defined. This is done using a Block Definition Diagram, as may be seen in Figure 4.



Figure 4 - General Architecture Block Definition Diagram

The creation of Figure 4 comes from analyzing the requirements and grouping the required functionality. It may furthermore be seen that no decision has been made as to what is realized in HW or SW, with the exception of the Housing, which must, for obvious reasons, be realized in HW.

When transforming the requirements into an overall architecture many different techniques may be used. An example is [SOFTARC], and though these books focus on Software Architecture, they may also be for HW/SW architecture. Though many books on architecture exist, a certain level of experience is extremely valuable when choosing the correct architecture.

Looking at the Quality Attributes we can see that it is a relatively static system; no Plug-and-play, no on-the-fly core updates, no third party peripherals. This means that the architecture does not need to include pluggable components or extendable drivers.

The high requirements for availability points to a simple system; the number of errors in a system is always proportional to the amount of code, and also a system that can be tested thoroughly, which requires interfaces for testing the modules independently.

There are many different architectures that can fulfil the requirements, and there may even be several equally good architectures. Therefore choosing an architecture becomes a matter of weighing the quality attributes against the architectural patterns that are suited to satisfy them and combine this with experience. Naturally the architectural patterns should be modified as needed to only include the parts of the patterns that are necessary for this application.

Grouping the requirement into independent blocks using the techniques described above, a possible collection of architecture blocks could be the one shown in Figure 4.

Here is may be seen that the following blocks and their responsibilities has been identified:

* Housing
  + The physical frame that holds the electronics. This is per definition HW, and it has no intelligence or electrical components, only housing.
* Antenna
  + The physical antenna used by the ISM block, as well as potentially any required physical components not part of the ISM block.
* ISM
  + The logic required to package and transmit a data stream / data frame to the Antenna and receive a data stream / data frame from the Antenna. The block is also responsible for maintaining the connection, if there is a connection to maintain (depends on the low-level protocol), required channel hopping, etc. The exact division of responsibility between the Antenna and the ISM block with respect to physical components may be adjusted depending on the mapping.
* LED
  + The physical LED as well as any required low-level driver and physical components required by the LED (often a discrete output is dimensioned so it can drive an LED directly).
* Button
  + The physical button as well as any required low-level driver and physical components required by the button (e.g. a simple hysteresis circuit to prevent multiple activations).
* Battery
  + The physical battery, the charging circuitry and monitoring circuitry as well as any required low-level driver.
* Microphone
  + The physical microphone as well as any required low-level driver and physical components required by the microphone (e.g. an external filtration or amplification circuit).
* Speaker
  + The physical speaker as well as any required low-level driver and physical components required by the speaker (e.g. an external filtration or amplification circuit).
* Control
  + The control logic which maintains the overall state of the system, including timing (when to check battery status, when to send heartbeats), commands from the base (cancel emergency, update firmware) and peripherals (button, LED). The Control logic also is responsible for turning on and off the Audio.
* Communication
  + The communication logic which handles the application level data to and from the ISM block. It is responsible for adding the any required frame header and/or footer specific to the application and to distribute received data to Control or Audio depending on the content (received frame header/footer).
* Signal Strength control
  + The logic responsible for adjusting the transmission strength of the ISM module based on e.g. number of retransmissions (BER), received transmission strength or information from the based.
* Audio
  + The logic responsible for; sampling the microphone, filter out the feedback noise and package the data and send it to the Communication block, receiving audio frames from the communication block, unpack the data and send it to the speaker and the Echo cancellation filter. The Audio block may be enabled and disabled depending on whether an emergency is ongoing.

### Dependencies and flow

The above Basic Block Diagram (BBD) only shows the blocks that make up the system and some information about what blocks are part of other blocks, but it does not show how blocks communicate. For that we need the Internal Block Diagram (IBD).

Here we will focus on the important blocks, meaning the blocks that have some data flow, and the blocks that have an interesting interface.

The remaining blocks should naturally also be done, but they are not imperative to do a proof-of-concept or to choose an architecture and platform.

#### Control

The main state machine, as shown in the requirements specification, is implemented in the Control block, and it is therefore interesting to see which interfaces this block exposes and needs. This may be seen in Figure 5.

Here it may also be seen that the all interfaces are standard interfaces, there are no flow data. This is an architectural decision to keep the flow data and the control data separate.



Figure 5 - Control Internal Block Diagram

In Figure 5 the following sub-blocks has been defined with their own responsibilities:

* Command Handler
  + Handle all commands, whether from the communication channel, the button or a timeout of the RTC (heartbeat).
* Communication Handler
  + Handle all communication to and from the Communication block. This block is responsible for parsing the command messages and parsing on the correct command, as well as packaging any responses for transmission.
* Test battery status and perform heartbeat
  + This is the only timer that must be active at all times. It is responsible for waking the system at predefined intervals to send a heartbeat and/or test the battery.

#### Audio

The Audio block is very interesting as it is the block with the most application level real-time processing and filtering.

To specify the inner workings of the Audio block it is necessary to consider the exact division of the responsibility of the Audio block and the speaker and microphone. From the BDD it is indicated that the Microphone and Speaker may contain more than just HW, but it does not say that it has to. Only the Audio block uses the speaker and microphone, so there are no external demands on the separation of responsibility.

As availability (and thereby stability) is very important using tried and proven technology. This goes well with the fact that quality of the audio is not imperative (no need to use state of the art technology to improve sound quality) and with the cost constraint. The battery constraint might warrant considering a non-common design.

In order to make a decision here a small investigative project could be done to determine alternative audio stages and their power requirements, price and maturity. An alternative is to make an educated guess and then make a note that there might be a potential for improving the power consumption, should it later be needed.

Due to the high requirement for availability and the fact that a lot of research has already been done into the power consumption of traditional audio stages, it is believed to be beneficial to postpone the investigation into alternative audio stages.

The traditional implementation of an audio stage is shown in Figure 6.



Figure 6 - Traditional Audio stage

As analog data is subject to noise and loss it is preferable to have the speakers and/or microphone as close to the DAC / ADC as possible. This is the principle used in e.g. active speakers, where the DAC has been moved all the way into the speaker. However for this project the speaker and microphone is already in the immediate vicinity of the Audio block, so there is nothing gained by moving the ADC / DAC out of the Audio block, and this allows a very nice separation into HW and SW, having the speaker and microphone encapsulate everything before the DAC / ADC, and the Audio block contain everything after the DAC / ADC. The DAC / ADC themselves are placed in the Audio block, yet depending on the mapping this might be changed. If the ADC /DAC is an external component it might make more sense to logically see them as HW, where if the ADC / DAC is an integral part of the processing component it might make more sense to model them as part of the Audio block.



Figure 7 - Audio Internal Block Diagram

In Figure 7 the following sub-blocks has been defined with their own responsibilities:

* Audio Control
  + Exposes the interface to turn on and off the Audio component. Turning off the audio component entails disabling the peripherals (shut down the amplifier and microphone), turn off the ADC and DAC and disable the encoding and decoding logic. Anything received from the Communication block is simply discarded. Turning the Audio component on is doing just the opposite.
* DAC
  + Converts a discrete value to an analog voltage level. The DAC may be a simple Zero-order hold, and then the speaker contains the restoration filter (a simple passive low-pass filter), or it might have a more complex output. The analog voltage level is filtered and amplified as needed and used to drive the speaker
* ADC
  + Converts the analog input from the Microphone to a discrete value and copy the value to the Echo cancellation on each Audio tick.
* Splitter
  + Makes a single value available to the DAC and the Echo cancellation at the same time, and notify the interested parties when the value changes.
* Echo cancellation
  + Filter out the echo from the speaker that is picked up by the microphone. This uses the transfer function from the Splitter to the secondary side of the ADC to determine the required echo cancellation.
* Audio decoding
  + Receive frames from the communication block and buffer a predefined number of frames. Decode the frames into the output buffer for the splitter and copy one value to the splitter on each Audio tick.
* Audio encoding
  + Receive discrete values from the echo cancellation and encode them for transmission over the ISM. When a full frame has been encoded transmit the frame using the Communication block.

There are several decisions that go into this diagram. Firstly the formats of the flow data have been specified; however there is still the matter of how data flows through and between the different internal blocks (the formats only specify what data and the frequency).

There are several different approaches, were the first decision is whether data-loss is allowed. When implementing a system like this there is always at least two clocks; the audio clock (8 kHz) and the main clock (significantly faster). The main clock indicates the speed of calculation and must be able to perform the necessary calculations on a sample before the next sample is ready. This is illustrated on the sequence diagram in



Figure 8 - Audio input processing sequence diagram

In order to evaluate the system’s ability to live up to these timing requirements, there are both external and internal considerations.

As the system is relatively simple and is not required to handle other streams at the same time as the audio stream, it is possible to quite accurately determine the required main clock. The other parts of the system which might consume some processing power while Audio is processing is any commands received on the ISM, heartbeats and battery test as well as button pushes. As Audio is only active during an emergency all these processes (except ISM) can be shut down during an emergency, and the only command that should be received during an emergency is the “Cancel Emergency” command, in which case it is irrelevant whether the Audio processing fails to meet its deadline. It is therefore possible to “simply“ count the required clocks from the ADC is sampled until the data is transmitted on the ISM and add the required clocks from the ISM to the DAC. To this the external factors must be added.

* ISM may lose packages (both ways)
* The may be significant delays in the GSM/ISM network

For lost packages there are two solutions; accepting that they may be lost and continuing (fire and forget) or using an ACK and retransmitting lost packages. The delays are “solved” using buffering. The simplest buffer is 1 frame. This means the buffer only holds what is required to handle one data package, and if the next package is delayed a hole in the sound is experienced. An alternative is to buffer a certain number of ms of data before beginning the playback – effectively introducing a delay. When there is a delay on the transmission line it is usually followed by a burst of data, as all the frames that have been waiting is pushed through. With a single frame buffer only the last package is kept. If there is a buffer it able to smooth out the burst so potentially no data is lost. If the delay is greater than the buffer then data is lost no matter what.

The delay should also be considered, as it should be able to maintain a normal conversation. Naturally much inspiration can be taken from the cell phone industry, and a buffer of 500ms - 1s is a good place to start. Naturally if the implementation phase shows otherwise it may be changed to accommodate.

An obvious place for this buffer is in the audio decoding block. There are two options; Keep the received data buffered and decode it so it fits with the Audio clock; decode everything into a buffer and simply pop from this buffer using the audio clock. The second solution gives the most deterministic analysis, as the buffer size directly indicate the time delay. If the buffer contains encoded data it is not possible to say exactly how much PCM data there is (without decoding it) – this is naturally only true if an encoding scheme is used that encode certain sounds/frequencies differently than others (no linear relationship between PCM data size and encoded data size). However the first solution requires the least amount of buffer space, as the encoded data always take up less space than the PCM data. By using the minimum required buffer space to keep the desired buffer of encoded data, the smallest possible platform can be chosen, and if a platform with sufficient memory is chosen then the raw buffer solution can always be implemented.

The buffer, whether raw or encoded, must always be circular, so that if a long delay occurs then when the data is burst through, then the oldest data is overridden. Naturally during this long delay the buffer would have been emptied, and though there are algorithms for smoothing out small gaps by using e.g. interpolation and statistical guessing. These algorithms are believed to be unnecessary for this application, and if the buffer is empty a value of 0 is sent to the DAC.

In Figure 8 a section has been left out. It is the secondary input to the Echo cancellation. Echo cancellation is simply put just a matter of subtracting the “noise” generated by the speakers from the microphone input. To do this it is required to know the exact delay and transfer function for the system from the data is sent to the DAC until it is pickup up by the ADC. The exact calculation of the transfer function and delay will not be done here, and will be left for the implementation phase. The technology is well known and implemented in any semi-modern cell phone and the algorithms are readily available. The Echo cancellation might also need a small buffer, as it may be necessary to keep a certain number of past values, as well as the coefficients for the Echo cancellation algorithm. The simplest form of Echo cancellation is if the distortion is 0 and the delay is 0. In that case it is simply a matter of subtracting what is played on the speaker from what is received by the microphone. The data flow for the speaker and microphone as it relates to Echo cancellation is shown in Figure 9.



Figure 9 - Echo cancellation

As the main clock must always be able to handle one sample in the time before the next sample arrives, no buffers are required to hold the sampled data from the ADC until the frame buffer holding the encoded data. It is however imperative that there is complete control of the timing in the splitter and Echo cancellation.

The simplest is to run both the Audio from the microphone (ADC) and the audio to the speaker (DAC) on the same audio clock and design the splitter and feedback filtration so the Feedback filtration always gets the current value or the last value (deterministically). It is important to consider when the feedback filtration request / uses the value from the splitter. The simplest design for the splitter is a one value buffer that updates it value on the on the audio clock and then indicate that a new value is available. It simply mirrors the audio clock with the delay it takes to copy the value into the hold buffer. The Feedback filtration is then sensitive to two values, but the input from the ADC and the input from the splitter begin ready. As the ADC performs it sample and hold on the Audio clock as well the delay before the Feedback filtration is allowed to run is the delay imposed by MAX(DELAY(ADC), DELAY(Splitter)).

A little detail about the decoding of received data. It may be seen that it is not required that an entire frame between two samples being written to the DAC, but only that the entire frame can be decoded at least as fast as the DAC need the raw PCM data. In order to allow for burst reception without re-buffering the decoding must however be faster than the consumption of the decoded values, and a good rule of thumb is that the decoding of N bytes of data must be completed within ½ the time used to consume the N bytes. In other words as 8000 12 bit values (~ 96000 bytes) is consumed in 1 second then the decoding of the 9600 bytes (GSM 06.10 has a compression rate of 10:1) should be completed in no more than 500ms.

#### Communication

The communication block also has an aspect of data flow, yet as it is merely responsible for forwarding the data and commands to other blocks it has a less complex content.

As with the other blocks it is necessary to decide on the exact division of responsibility. Here the division is between the ISM block and the Communicatioin block. It has been mentioned earlier that the Communication block is responsible for the application layer handling of data. It is defined in the OSI model what this means.

The Communication block must peek at all received data and parse it on to the interested parties. There are three types of data;

1. Control data (commands)
2. Firmware update data
3. Audio data

Audio data is streamed and a data packages has a very limited life-span. If the package is not received in a timely manner it might as well be lost. This points towards a fire and forget protocol, e.g. UDP. There are special Audio protocols that may also be used.

The Control data on the other hand are small packages and must not be lost. This indicates a safe communication protocol, e.g. TCP.

Finally the Firmware update data. This is also streaming data, yet it is not allowed to lose any packages, so here it is generally preferable to use a safe protocol as well.

The data being sent must have any required header and footer added and then be sent using the desired protocol. If the desired protocol is a safe protocol it is the responsibility of the ISM block to ensure end to end transmission.

This may be shown in the Internal Block Diagram in Figure 10.



Figure 10 - Communication Internal Block Diagram

In Figure 10 it may be seen that any buffering of the data to be written (if the ISM block is unable to transmit it immediately) should be done within the ISM block.

Distinguishing between control data, firmware update data and audio data can be done in several ways.

1. Single data stream
2. Multiple ports (data streams)
3. Multiple protocols.

The single data stream operate at a lower level, and down at the lower layers of the ISM block there is naturally only a single data stream. However this single data stream can be used directly by adding a header to each package containing the type of content (control, firmware, audio), length and any other information needed, and then have the Data parser and control block distribute the content accordingly.

Working at a slightly higher level it is possible to have the ISM block do this division. This can e.g. be done using the TCP / UDP protocol, where a port number may be used to distinguish between control, firmware and audio data. Naturally this has a higher overhead, as there are extra information in the headers that are not relevant for this application, on the other hand the protocols are tried and proven and mach available implementation (both in ASIC, SW and VHDL) exists.

Finally there is the protocol level. There are special protocols designed for transmitting audio data (lossy streaming), binary data (lossless streaming) and control data (RPC). Using these high level protocols naturally has an extra overhead, yet they also come with advantages, as the code generally exist and is ready to use, and the protocols are tweaked to the specific need of the communication, and contain parameter like e.g. QoS for Audio.

Which solution is preferable depend on two aspects; the overhead as it relates to power consumption and processing requirements, and the capabilities of the chosen platform.

To illustrate the dynamic behaviour of the Communication block the activity diagram in Figure 11.

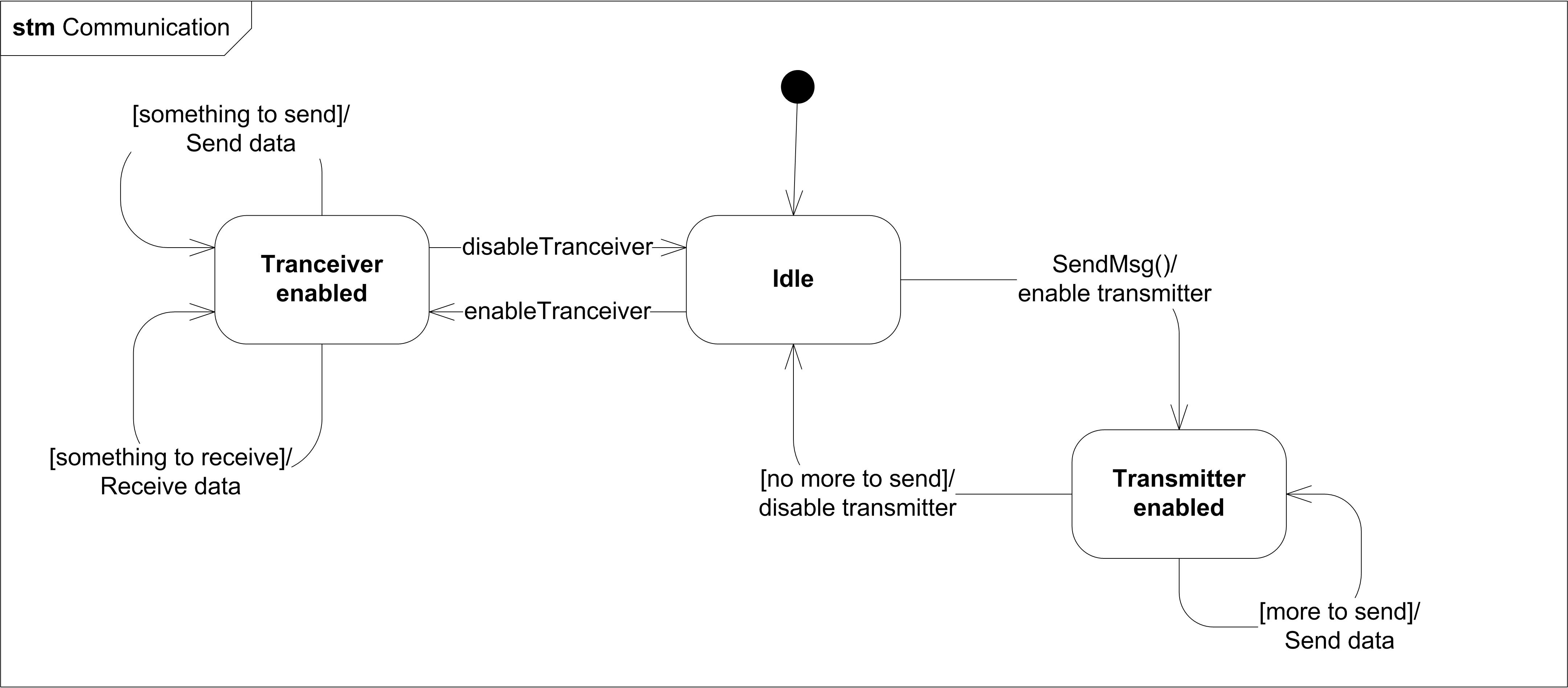


Figure 11 - Communication Activity Diagram

#### ISM

<<pending>>

### Evaluation

Based on the above general architecture it is possible to determine the critical path through the system and to determine the required execution and communication requirements for the HW. Unfortunately it is not as simple as it sounds, as there are several mapping-specific considerations that go into the calculation.

Firstly it is necessary to not only identify the critical path, but also to indicate which functionality can be executed in parallel, as this may greatly lower the required execution requirements. An example could be the Audio encoding. If the execution path is:

1. Sample -> Filter -> Encode -> Send

Then the total time for the encoding is include, yet if multiple samples can be or must be buffered and encoded in parallel with other samples being buffered, then the execution path is:

1. Sample -> Filter -> Buffer
2. Encode -> Send

Where the Encode -> Send has to be performed within the time it takes to fill the buffer, and not the time between two samples.

In Table xx may be seen an overview of the different parallel execution paths, as well as their maximum frequency of execution. Only the blocks with a relatively high flow of information are included, as the other paths will never be part of the critical execution and communication path.

|  |  |
| --- | --- |
| **Execution path** | **Frequency** |
| 1. Microphone -> ADC - > Echo cancellation -> Audio encoding | 8kHz |
| 1. Audio encoding -> Data Handler -> ISM -> Antenna | 8kHz / frame size |
| 1. Antenna -> ISM -> Data Parser -> Audio decoding | 8kHz / frame size |
| 1. Audio decoding -> Splitter -> DAC -> Speaker | 8kHz |
| 1. Audio decoding -> Splitter -> Echo cancellation -> Audio encoding | 8kHz |

Table 1 - Execution paths of interest

It may be seen the Firmware update (the other data intensive instruction) is not included. This is due to the relatively low communication requirements. Based on the type of application a total footprint of <100kbyte (SW implementation) is very realistic, and therefore the throughput of 100kbyte in 30 minutes, gives a required bit rate of 100000\*8 / (30 \* 60) = 444 bit/s. Naturally this will increase with sending acknowledgement. Depending on how the data is stored there could be a delay for writing to Flash, yet as very little processing of the data is required, and as one way GSM audio require an absolute minimum of 6,5kbit/s [*GSM*], it is believed that the critical path is not found in the Firmware update.

Knowing these paths it is possible to look at the individual blocks in the paths and evaluate their requirements with respect to execution and communication, and from there determine the critical path, yet to do this it is necessary to know the execution and communication time for the different parts of the paths. This is heavily dependent on the mapped to the selected HW, as a HW implementation (FPGA) has an entirely different “required clock cycles” to implement a functionality than a processor has. Where an FPGA is well suited for filtration and data flow, it is not very well suited for control logic. This is however very mapping dependent and will be postponed until the mapping phase.

### Transaction Layer Modelling

Once the General architecture is established it may be interesting to determine if it is feasible from a functional point of view. This may be determining by developing an executable Transaction Layer Model (TLM) of the system. It is however not necessarily a good idea. If there is a high level of confidence in the design, then it maybe a waste of time. There is however a second advantage with developing the TLM, and that is that it may be extended to be a Timed TLM, BCAM, CCAM or CAM. Doing so will allow a very fine grained evaluation of the requirements to execution and communication.

There is a high degree of confidence in the design, and the possible critical paths have been identified. It would therefore be quite possible to determine the execution and communication requirements for the individual blocks and simply calculate the requirements, as may also be seen in the mapping section. However, as part of the project is to gain knowledge of and evaluate the process involving SysML -> SystemC, it is acceptable that extra time is set aside for SystemC analysis of the system, SystemC being the language of choice for the TLM.

It is however not necessary to implement all parts of the system, only the parts that are necessary to determine the execution and communication requirements of the critical path. This requires implementing the Microphone, Speaker, Audio, Communication, ISM and Antenna blocks. As the purpose of the implementation is execution and communication requirements evaluation it is not necessary to implement the physical HW units, like the Antenna, Microphone and Speaker, as they may be seen as having a 0 delay and execution requirement.

The remaining blocks are implemented in the code project [*SystemCTLM*].

When executed the ISM simulated receiving data from the antenna by reading bytes from a file. This data is then packages into frames and transmitted to the communication block. The communication block then peek at the header and forward the data as needed. This is equivalent with the Single data stream version of the communication block, as mentioned above. This is due to the fact that this has the simplest data path (no advanced protocols) and is sufficient to determine a critical path, and if a higher level ASIC ISM chip is chosen then the Communication block will only get simpler (worst case).

The data progress through the system according to the general architecture ending at the DAC simulator, which writes the data to the speaker to a file.

The ADC simulator also read its sample data from a file, and also read the Acoustic feedback noise from the DAC simulator and sends it through the system until the ISM simulator, which writes the data to a file.

To generate the input files a special tool has been developed [*SystemCTLMFileGenerator*], which takes a running time as an input and generate two files with a known pattern. The pattern may be seen in the SystemCTLMFileGenerator.

The test procedure is as follow:

1. Generate two files of 60 seconds duration.
2. Run the SystemCTLM
3. Switch the output files so they become input files
4. Run the SystemCTLM
5. Verify that the newly generated output files are identical to the original input files.

TODO: insert diagram

#### Experiences with SystemCTLM

It took 6 hours to create the SystemC model of the system. It resulted in no updates to the architecture. It shows that it is possible to execute the architecture.

If no further work was done on the model, then it contains no additional information than what can be read from the architecture.

## Mapping of the general architecture

When the general architecture has been defined and potentially verified (SystemCTLM), it is necessary to look at how this architecture may be mapped to HW/SW.

There are multiple ways of doing this mapping. In an ideal situation it would be possible to compare the different HW directly, but implementation and instruction set differences makes it impossible, as previously mentioned. It is therefore necessary to look at a finite collection of solutions, which are created from experience with similar systems.

With a system containing both control and flow data; states and filters, the following mappings are feasible:

1. Microcontroller and ISM transceiver ASIC.
   1. ISM transceiver ASICs specially designed for low power consumption may be readily purchased, some even with a built-in microcontroller in a single IC.
2. DSP, Microcontroller and ISM transceiver ASIC.
   1. If the filtration and encoding/decoding parts cannot be handled by a microcontroller (cost effectively) then adding a DSP may solve the problem.
3. FPGA, ADC/DAC and ISM oscillator and LNA receiver filter.
   1. Using an FPGA is a high-cost solution, and doing so would require that almost all logic can be mapped to HW, so the external components become very cheap, yet the high performance make it possible to implement the ISM module without an ASIC.

Further combinations then ones mentioned here naturally exist, as will be shown later during the Pareto point analysis.

All the solutions require external amplifier, speaker, microphone, antenna and housing, as well as some external components for the button, LED and battery.

### Microcontroller and ISM transceiver ASIC

### Multiple solutions for this exist, yet a good example is the CC430 low power microcontroller and integrated ISM transceiver. Basically it is a TI MSP430 and a CC1101 in a single chip. As this is a very possible choice this will be used as an example for the Microcontroller and ISM transceiver ASIC platform.

This platform may be seen in Figure 12, where also the processing elements and communication elements have been indicated. It is important to realise that at this point it is only a mapping proposal, and it has not been verified that the microcontroller can actually perform the required processing in a timely manor.



Figure 12- Microcontroller and ISM transceiver ASIC

In order to determine feasibility first the characteristics of the platform must be determined. These may be seen in Table 2.

|  |  |
| --- | --- |
| Frequency | 20MHz |
| ADC | 12Bit SAR  >1 MSPS |
| Flash | 32KB |
| RAM | 4KB |
| DAC | Not supported |
| GPIO | 44 |
| SPI | 1 (USCI) |
| ISM | Up to 250kbit/s |
| Power | On: 160 µA/MHz  Sleep: 2 µA  Rx LNA: >15mA  (Supply = 1.8 – 3V) |
| Price | 5.35USD |

Table 2 - CC430 characteristics

As it may be seen the CC430 do not have a built in DAC, and an external one must be chosen. A good choice is the TLV5616 from TI, to stay in the same family. It also comes in a dual DAC package, and a second analogue output may be used to control the Antenna power booster and LNA to lower power consumption (Signal Strength Control). Its characteristics are quite common for a DAC, and can therefore easily used even if another DAC is chosen.

|  |  |
| --- | --- |
| Resolution | 12bit |
| Settling time | 9us  0.102 MSPS |
| Interface | SPI |
| Power | On: 3mW  Sleep: 10nA  Supply = 3V |
| Price | 3.5USD |

Table 3 – TLV5616 characteristics

The CC430 also require external transmitter amplification. There are several options, e.g. using an entire transceiver front end, like the SKY65346 FEM. It supports several of the high level protocols. Alternatively a simply amplification stage (Power Amplifier (PA)) for the CC430 transmitter may be implemented. The maximum transmission power allowed for the frequency band is 500mW, so with a little loss in the amplification stage a power consumption of 600mW worst case may be assumed, i.e. very similar to using the SKY 65346 FEM, as may be seen in Table 4.

|  |  |
| --- | --- |
|  | 9us  0.102 MSPS |
| Interface | SPI |
| Power | On: 200mA  Sleep: 5µA  Supply = 3.3V |
| Price | 9.3USD |

Table 4 – SKY65346 FEM characteristics

It may be noticed that the physical dimensions have been excluded. This is due to the fact that all the proposed solutions has a very small form factor (single or dual IC), and the size difference so small that it is not going to affect the possible battery package size.

Knowing the platform it is possible to determine how much processing and communication is required. This can be done in several ways. As it may be seen in the SystemCTLM the algorithms have been split out, so they could be implemented and run on the actual target to determine the required clocks cycles. Alternatively an Instruction Set Simulator (ISS) may be employed to simulate the execution without having the actual HW and determining the required clock cycles from there. A third alternative is to estimate the number of instructions based on the algorithms and communication paths and then use these number. The SystemCTLM may then be augmented with the given execution and communication times resulting in a Timed Transaction Level Model (see later). The numbers may also be directly plotted against the different execution paths to determine the critical path and total required clock cycles, avoiding the SystemC simulation.

Here the third alternative will be employed, and an estimation of the required instructions to implement the algorithms (blocks) on the MSP430 microcontroller instruction set will be performed, and can be seen in Table 5. As the communication between the blocks except the DAC is purely internal it is a matter of writing the internal RAM, and communication of a 16bit integer is therefore a matter of a write and a read (if not synchronization is needed), and adding two clocks between each block should be sufficient. Writing to the SPI interface is about 10 clocks/sample.

|  |  |  |
| --- | --- | --- |
| **Block** | **Description** | **Clock cycles** |
| Audio.AudioEncoding | Encoding of 160 integers of audio data [*libgsm*] | 500 (152 bytes stack RAM) |
| Audio.AudioDecoding | Decoding of 32,5 bytes of encoded audio data [*libgsm*] | 500 (152 bytes stack RAM) |
| Audio.Echo cancellation | Filtration of autistic feedback\*   * Delay in air (20cm): 587µs * Delay in ADC: 1µs * Delay in comm to DAC: <2µs * Delay in DAC: 10µs * Delay from value from splitter is received by the microphone: 10+2+587+1µs * Filter size at 8kHz (number of samples):   600µ\*8k=4.8 tab. [*AEC*] | 15 clocks (5 int tabs = 14 byte RAM) |
| Audio.Splitter | Copy into shared RAM block | 1 |
| Audio.ADC | Sample ADC | 1 |
| Audio.DAC | Write integer to SPI | 10 |
| Communication.DataHandler | Add header and host to network order copy of 33 byte   * Add header: 10 * Loop: 2 * Int to 4 byte network order: 4 * Copy 4 byte: 1 | 10 + 2 +( 4 + 1)\*33 = 177 |
| Communication.DataParser | Parse header, copy data, network to host order of 37 byte:   * Parse header: 10 * Copy data: 37 * Compare: 3 * Loop: 2 * 4 byte to int host order: 4 | 10 + 37 + 3 + 2 + 4\*37 = 200 |
| ISM | Receive/Transmit across ISM | ? (dedicated ASIC) |

Table 5 - CC430 execution times

Knowing the different paths from Table 1 it is possible to determine the required execution power from the CC430 (including communication as it is handled by the same micro controller).

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution path** | **Clocks** | **Frequency** | **Clocks/s** |
| 1. Microphone -> ADC - > Echo cancellation -> Audio encoding | 1(ADC) +  2(Comm) +  15(Echo) +  2(Comm) = 20 | 8kHz | 160k |
| 1. Audio encoding -> Data Handler -> ISM -> Antenna | 500(Enc) +  2\*33(Comm)  652 (hdl) +  2\*37(Comm) = 1292 | 8kHz / frame size =  8000 / 160 = 50 | 64.6k |
| 1. Antenna -> ISM -> Data Parser -> Audio decoding | 2\*37(Comm) +  655 (parse) +  2\*33 (Comm) +  500 (dec) = 1295 | 8kHz / frame size =  8000 / 160 = 50 | 64.75k |
| 1. Audio decoding -> Splitter -> DAC -> Speaker | 2 (Comm) +  1 (Split) +  2 (Comm) +  10 (DAC) = 15 | 8kHz | 120k |
| 1. Splitter -> Echo cancellation | 2 (Comm) | 8kHz | 16k |
| Total |  |  | 425k |

Table 6 - Execution paths of interrest

The result of 1.1 is most likely low, as it based on an ideal compilation of the instructions, yet using a simple factor-10 rule of thumb it is still possible for the CC430 to run the code, as it has a 20 MHz clock, and with a factor-10 increase, a 4.25MHz clock would be sufficient. This will also allow very low power consumption on the microcontroller, as it uses 160uA per MHz, and as 1MHz is sufficient that is very good.

It may be seen that the ISM block does not require any execution. This is due to the fact that the ISM block is a PE (Processing Element) in itself and dedicated to handling the ISM. It is therefore not able to run any user code, yet also do not impose any execution on the microcontroller.

As a single clock takes 1/20000000 = 50ns it is possible to augment the SystemCTLM with the correct wait and thereby achieving a Timed TLM. As it has already been shown that it is possible there is no real reason to create the SystemC Timed TLM version for the Microcontroller solution, other than to gain experience with Timed TLM, and for that reason the SystemC\_TTLM\_Micro project has been created. It performs the same operations as the TLM version, but with correct waits.

#### Timed Transaction Layer Modelling

SystemCTimedTLM

<<TODO>>

### Microcontroller, DSP and ISM transceiver ASIC

As the Microcontroller is able to handle the execution by itself, it would be illogical to add an extra component, yet if a smaller Microcontroller might be chosen and then augmented with a DSP then it might be beneficial.

A suggested platform is shown in Figure 13. An alternative to this platform is to implement the CC1101 functionality in the DSP. A way of doing this is with an oscillator and LNA and then a very fast ADC to recreate the received signal. Unfortunately as recreating a signal requires a minimum sampling frequency twice as high as the carrier wave frequency, or 868 \* 2 = 1,7Ghz. This is within the possible range of a DSP, but a very high end, and this alternative is much more suited for an FPGA implementation, which will be explored in the FPGA platform.



Figure 13 - DSP and Microcontroller platform

Using the same DAC and PA as before the ADC, DSP, Microcontroller, CC1101 may be summed up in the tables below:

|  |  |
| --- | --- |
| Resolution | 12bit |
| Settling time | 8us  100 KSPS |
| Interface | SPI |
| Power | On: 10.8mW  Sleep: 5µA  Supply = 3V |
| Price | 4.27USD |

Table 7 – ADC AD7854 characteristics

|  |  |
| --- | --- |
| Frequency | 5MIPS |
| Flash | 14KB |
| RAM | 512B |
| GPIO | Up to 25 |
| SPI | 1 (USCI) |
| UART | 1 |
| Power | On: 600nA  Sleep: 30nA  (Supply = 1.8V) |
| Price | 1.2USD |

Table 8 – Microcontroller PIC16F1516 characteristics

|  |  |
| --- | --- |
| Frequency | 50MHz |
| Flash | 14KB |
| RAM | 16KB |
| SPI | 2 (McBSP) |
| DMA | 1 |
| Power | On: >22mA  Sleep: 20uA  (Supply = 1.8V) |
| Price | 3.5USD |

Table 9 – DSP TMS320VC5401 characteristics

|  |  |
| --- | --- |
| Resolution | 12bit |
| Settling time | 9us  0.102 MSPS |
| Interface | SPI |
| Power | Sleep: 200nA  Rx LNA: >15mA  Supply = 1.8 – 3.6V |
| Price | 2.0USD |

Table 10 – ISM transceiver CC1101 characteristics

As with the microcontroller it is now possible to determine if the microcontroller and DSP is sufficiently powerful. Naturally as the DSP is more powerful than the microcontroller then it is, but it may be interesting to see how much better it can do the calculations.

|  |  |  |
| --- | --- | --- |
| **Block** | **Description** | **Clock cycles** |
| Audio.AudioEncoding | Encoding of 160 integers of audio data [*libgsm*]   * Shift and multiply-accumulate in single instruction. | 200 (152 bytes stack RAM) |
| Audio.AudioDecoding | Decoding of 32,5 bytes of encoded audio data [*libgsm*]   * Shift and multiply-accumulate in single instruction. | 200 (152 bytes stack RAM) |
| Audio.Echo cancellation | Filtration of autistic feedback\*   * Delay in air (20cm): 587µs * Delay in comm. to ADC: < 2µs * Delay in ADC: 8µs * Delay in comm.. to DAC: <2µs * Delay in DAC: 10µs * Delay from value from splitter is received by the microphone: 10+2+8+2+587µs * Filter size at 8kHz (number of samples):   609µ\*8k=4.87 tab. [*AEC*] | 6 clocks (5 int tabs = 10 byte RAM) |
| Audio.Splitter | Copy into shared RAM block | 1 |
| Audio.ADC | Read integer from SPI | 10 |
| Audio.DAC | Write integer to SPI | 10 |
| Communication.DataHandler | Add header and host to network order copy of 33 byte   * Add header: 10 * Loop: 2 * Int to 4 byte network order: 1 * Copy ram block: 20 | 10 + 2 +33 + 20 = 65 |
| Communication.DataParser | Parse header, copy data, network to host order of 37 byte:   * Parse header: 10 * Copy data: 20 * Compare: 3 * Loop: 2 * 4 byte to int host order: 1 | 10 + 20 + 3 + 2 + 37 = 72 |
| ISM | Receive/Transmit across ISM | ? (dedicated ASIC) |

Table 11 - DSP execution times

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution path** | **Clocks** | **Frequency** | **Clocks/s** |
| 1. Microphone -> ADC - > Echo cancellation -> Audio encoding | 1(ADC) +  10(Comm) +  6(Echo) +  2(Comm) = 19 | 8kHz | 152k |
| 1. Audio encoding -> Data Handler -> ISM -> Antenna | 200(Enc) +  2\*33(Comm)  160 (hdl) +  2\*37(Comm) = 500 | 8kHz / frame size =  8000 / 160 = 50 | 25k |
| 1. Antenna -> ISM -> Data Parser -> Audio decoding | 2\*37(Comm) +  162 (parse) +  2\*33 (Comm) +  200 (dec) = 502 | 8kHz / frame size =  8000 / 160 = 50 | 25.1k |
| 1. Audio decoding -> Splitter -> DAC -> Speaker | 2 (Comm) +  1 (Split) +  2 (Comm) +  10 (DAC) = 15 | 8kHz | 120k |
| 1. Splitter -> Echo cancellation | 2 (Comm) | 8kHz | 16k |
| Total |  |  | 338k |

Table 12 - Execution paths of interest

There is no surprise here; the DSP is more efficient at performing filtration and encoding/decoding than the microcontroller, even with the increased communication. Luckily the DSP <-> microcontroller communication is at an absolute minimum, as the microcontroller is very limited in its capabilities, yet monitoring a button, a few LEDs, sending a heartbeat and turning on and off the DSP is certainly within its capabilities.

An interesting possibility is mapping all the execution to the DSP, completely removing the need for a microcontroller. In order to determine this, it is required to make an estimate of the required clocks for the remaining functionality. This is done in Table 13. As this functionality is pure control there is no real difference in implementing them in DSP or microcontroller, so the same clock cycle figures may be used for the microcontroller case. This is not true for Firmware Update for the microcontroller and DSP platform, as the update must be separated into a microcontroller update and a DSP update, or only a very limited update may be performed.

|  |  |  |
| --- | --- | --- |
| **Block** | **Description** | **Clock cycles** |
| Control | * Simple state machine and RTC (10) * Perform heartbeat (100) * Perform emergency (100) * Perform Battery test (100) | 310 every 2 hours. |
| Button | * Wake up in interrupt (button) (1) | 1 |
| LED | * Set GPIO (1) | 1 |
| Battery | * Request battery status (100) * Handle battery status (100) | 200 every 2 hours. |
| Firmware update DSP | * Microcontroller:   + Parse command (100)   + DSP in update mode ( 10)   + Send response (100) * DSP:   + Enter update mode (10)   + Receive and parse up to 100k in 30 minutes = 55 bytes/ sec: 125k   + Write and send response: 1k | DSP: 126k in 30 min  Mic: 210 in 30min |
| Firmware update Microcontroller only | * Microcontroller:   + Parse command (100)   + Microcontroller in update mode ( 10)   + Receive and parse up to 100k in 30 minutes = 55 bytes/ sec: 500k   + Write and send response: 1k | 502k in 30 min |
| Signal strength control | * Request signal strength (50) * Compare signal strength (100) * Write new signal strength (50) | 200 every 2 hours |

Table 13 – Other blocks execution times

There is however one important aspect that goes against the single DSP solution and that is the power consumption of the DSP. It is far greater than the microcontroller, and as the DSP only have to be turned on during an emergency call, this is a quite important aspect.

### FPGA, ADC/DAC and ISM oscillator and LNA receiver filter

An FPGA is more expensive and more power consuming than a DSP and a microcontroller, yet it ideally suited for parallel operations and fast date flow, so if the microcontroller alone can handle the execution, why even consider an FPGA? Because it might be able to remove a component; the CC1101. As mentioned in the DSP section it might be able to use a simple LAN and oscillator along with a fast ADC to implement the CC1101. This would require an extremely fast DSP, but how does it look for an FPGA.

In order to get a ball-park idea of the required FPGA we can look at the sampling. Operating at 868MHz, and remembering the Nyquist sampling theorem [*Nyquist*], it is required to sample the LNA at 2\*868MHz in order to restore the original signal. Naturally a low-pass filter eliminating higher frequencies is required, yet this can be done as a simple analogue filter. Sampling and analysing at this rate requires a frequency of at least 1.8GHz. The parallel nature of the FPGA would allow the processing of the sample (and continued processing of the previous samples) to be done in parallel, otherwise a much higher rate would be needed; at least a factor 10 to process the sample, or a DSP of at least 18GH, if it was to be implemented in DSP. Restoring the signal from the ADC allows for both Amplitude Shift Keying (ASK) and Phase Shift Keying (PSK) encoding.

There are several ADCs that can handle a sample rate of 1.8GHz. An example could be the National GHz ADC family ADC12D1x00, which is specifically designed to handle RF application in conjunction with an FPGA.

There are also several FPGAs, yet a good, and relatively small, option is the Altera Cyclone V GX. It is an up to 3G FPGA. The family supports up to 5G. As it is also mentioned in the risk analysis there are very few FPGA resources in the team, and this section may therefore be less accurate and thorough than the previous sections.



Figure - FPGA platform

|  |  |
| --- | --- |
| Frequency | <3,75Gbps |
| Power | 88 mW per channel  Supply = 3.3V |
| Price | 300USD |

Table 14 – FPGA Cyclone V GX characteristics

|  |  |
| --- | --- |
| Resolution | 12bit |
| Settling time | 2 GSPS |
| Power | On: 3.38W  Supply = 1.9V |
| Price | 995USD |

Table 15 – ADC12D1000 characteristics

The first thing that springs to mind is price, and the second is power consumption. These alone would be sufficient to exclude this as an option, however from an academic point of view it is interesting to continue with the analysis.

|  |  |  |
| --- | --- | --- |
| **Block** | **Description** | **Clock cycles** |
| Audio.AudioEncoding | Encoding of 160 integers of audio data | 50(100 gates) |
| Audio.AudioDecoding | Decoding of 32.5 bytes of encoded audio data. | 50 (100 gates) |
| Audio.Echo cancellation | Filtration of autistic feedback\*   * Delay in air (20cm): 587µs * Delay in comm. to ADC: < 2µs * Delay in ADC: 8µs * Delay in comm.. to DAC: <2µs * Delay in DAC: 10µs * Delay from value from splitter is received by the microphone: 10+2+8+2+587µs * Filter size at 8kHz (number of samples):   609µ\*8k=4.87 tab. [*AEC*] | 7 (50 gates) |
| Audio.Splitter | 16 bit hold | 1 (16 gates) |
| Audio.ADC | Read integer from SPI | 10 (100 gates) |
| Audio.DAC | Write integer to SPI | 10 (100 gates) |
| Communication.DataHandler | Add header and host to network order copy of 33 byte. | 10 (50 gates) |
| Communication.DataParser | Parse header, copy data, network to host order of 33 byte:   * Parse header: 10 * Copy data: 20 * Compare: 3 * Loop: 2 * 4 byte to int host order: 1 | 10 (50 gates) |
| ISM | Receive/Transmit across ISM | 4G (200k gates) |
| CPU | Required by Control | 20M (100k gates) |

Table 16 - FPGA execution times

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution path** | **Clocks** | **Frequency** | **Clocks/s** |
| 1. Microphone -> ADC - > Echo cancellation -> Audio encoding | 1(ADC) +  10(Comm) +  6(Echo) +  2(Comm) = 19 | 8kHz | 152k |
| 1. Audio encoding -> Data Handler -> ISM -> Antenna | 200(Enc) +  2\*33(Comm)  160 (hdl) +  2\*37(Comm) = 500 | 8kHz / frame size =  8000 / 160 = 50 | 25k |
| 1. Antenna -> ISM -> Data Parser -> Audio decoding | 2\*37(Comm) +  162 (parse) +  2\*33 (Comm) +  200 (dec) = 502 | 8kHz / frame size =  8000 / 160 = 50 | 25.1k |
| 1. Audio decoding -> Splitter -> DAC -> Speaker | 2 (Comm) +  1 (Split) +  2 (Comm) +  10 (DAC) = 15 | 8kHz | 120k |
| 1. Splitter -> Echo cancellation | 2 (Comm) | 8kHz | 16k |
| Total |  |  | 338k |

Table 12 - Execution paths of interest

An extra set of data has been added for later use: ISM in DSP only and ISM in microcontroller only. These numbers can be used for a Pareto analysis later.

|  |  |  |
| --- | --- | --- |
| **Block** | **Description** | **Clock cycles** |
| ISM Microcontroller | * ADC sampling: 10(clocks) \* 1.8GHz * Frequency analysis: 4 \* 1,8GHz * Protocol stack (50\*33 bytes/s): 165k * GPIO oscillator output (50\*33 bytes/s): 165 | 25,2GHz |
| ISM DSP | * ADC sampling: 2(clocks) \* 1.8GHz * Frequency analysis: 1 \* 1,8GHz * Protocol stack: (50\*33 bytes/s): 82.5k * GPIO oscillator output (50\*33 bytes/s): 165 | 5,4GHz |

References:

* Pattern-Oriented Software Architecture