Degree Final Project by Daniel Flores Tafur

Implementation of an audio player prototype
Mentored by Juan Climent
FIB, UPC 2014
0. Abstract

This document describes implementation of an audio player prototype based on STM32F4 platform (ARM architecture). The development of audio prototype is put in the context of developing a commercial product based on Open Hardware model, which emerged in recent years. The viability of finished product is analyzed, as well as potential market for Open Hardware products in general. The software implementation for Mikromedia+ (manufactured by Mikroelektronika) development board is discussed in detail and also provided entirely, licensed under Open Source license (BSD 2-clause).
Table of contents

1. Introduction .......................................................................................... 7
   1.1 General purpose of this project ....................................................... 7
   1.2 Acknowledgments ......................................................................... 8

2. Defining specifications ....................................................................... 9
   2.1 The context .................................................................................... 9
   2.2 Different types of consumers ......................................................... 10
   2.3 General specifications ................................................................... 12
   2.4 BIG VS little architectures ......................................................... 13
   2.5 The business model for the future product .................................... 18
   2.6 Choice of license ........................................................................ 21
   2.7 Basic goals of the project and required functionality of prototype ... 22
   2.8 Methodology of work .................................................................. 23

3. The hardware .................................................................................. 25
   3.1 Unification VS master and slaves ................................................. 25
   3.2 The choice of specific microcontroller architecture and vendor ...... 28
      AVR and ARM-based microcontrollers from Atmel ....................... 28
      Microchip’s microcontrollers ......................................................... 28
      ARM-based microcontrollers from ST Electronics ....................... 29
      Other microcontrollers worth mentioning ..................................... 29
      The comparison table between selected candidates ....................... 30
      The choice of platform ................................................................ 30
   3.3 The choice of development board for the prototype .................... 32
   3.4 The general traits of hardware architecture in the prototype ......... 33
   3.5 Detailed description of all hardware ............................................. 35
      STM32F407ZG ................................................................................. 36
      SSD1963 ....................................................................................... 36
      STMPE610 ..................................................................................... 38
      VS1053 ......................................................................................... 39
      SunBond LB04302 LCD Module (AT043B35-15I-10 in some documentation) .............................................................................................................. 44

4. The software ................................................................................... 46
   4.1 Single software approach VS OS-based approach ....................... 46
   4.2 General traits of software architecture ....................................... 49
   4.3 Development tools ...................................................................... 53
   4.4 Coding style .............................................................................. 54
   4.5 Detailed code explanation ............................................................ 55
      main.c ......................................................................................... 56
      ascii.h .......................................................................................... 58
      images.h ...................................................................................... 60
      delay.c, delay.h .......................................................................... 60
      lcd.c, lcd.h, command_listLCD.h ............................................... 62
      touch.c, touch.h, registers_listTouch.h ....................................... 67
      StdPeriphLib .............................................................................. 72
4.6 System in action ............................................................................................... 84

5. Development process ......................................................................................... 91
  5.1 Problems encountered during development ............................................... 91
  5.2 Project timeline ............................................................................................. 92
  5.3 Accomplished goals ...................................................................................... 93

6. Analysis of prototype and future improvements ........................................... 95
  6.1 The purpose of this section .......................................................................... 95
  6.2 Hardware codecs and their advantages and disadvantages ....................... 96
  6.3 File formats .................................................................................................. 98
  6.4 File system layer limitations ....................................................................... 99
  6.5 Possible internal storage ............................................................................. 100
  6.6 Additional connectivity .............................................................................. 102
  6.7 The absence of database driver ................................................................... 103
  6.8 User interface shouldn’t be limited to touch screen only ......................... 104
  6.9 Sorting algorithms must be added to file manager ....................................... 105
  6.10 Lack of Unicode support ........................................................................... 106
  6.11 Long file names must be supported .......................................................... 107
  6.12 We would eventually have to make a transition to an OS....................... 108
  6.13 We should definitely make use of modules available on current board ........................................................................................................ 109
  6.14 Overall look at functionality ...................................................................... 110

7. Considering the eventual commercialization of product. .......................... 111
  7.1 Introduction ................................................................................................... 111
  7.2 Business model: Open Source and Open Hardware .................................... 112
  7.3 The general description of this particular product ...................................... 114
  7.4 A non-exhaustive review of Open Source/Open Hardware products and companies on market .................................................................................. 116
    Efika MX computers from Genesi ................................................................. 116
    Mikroelektronika ............................................................................................ 117
    AdaFruit .......................................................................................................... 118
    Arduino ............................................................................................................ 118
    BeagleBoard ................................................................................................... 119
    Raspberry Pi .................................................................................................... 120
    Conclusion ...................................................................................................... 121
  7.5 Cost of the device ....................................................................................... 122
  7.6 Social, environmental and economical impact of product ......................... 124
  7.7 Possible derivatives of the product and alternative versions ..................... 126
    Alternative versions based on different platforms ....................................... 126
    Open Hardware scientific/engineering calculator ......................................... 126
    Open Hardware gaming console ................................................................... 127
    Open Hardware e-book .................................................................................. 127
8. Conclusion .......................................................................................... 128
9. References .......................................................................................... 129
10. List of acronyms .................................................................................. 133
Table of figures

**Figure 1.** The abstract hardware design of the prototype ............................................. 33
**Figure 2.** The actual design of the prototype with specific hardware and data transmission protocols .................................................. 35
**Figure 3.** The internal architecture of SSD1963 and its connections .......................... 37
**Figure 4.** STMPE610 internal architecture and connections ................................. 38
**Figure 5.** The internal architecture of VS1053 and its connections ......................... 39
**Figure 6.** Internal architecture of VSDSP4 ................................................................. 40
**Figure 7.** Sound from external sources vs. inside-the-head sound ......................... 42
**Figure 8.** The photo of the screen showing graphical content ................................ 44
**Figure 9.** The software architecture of prototype ....................................................... 49
**Figure 10.** The software architecture of prototype after implementation ............. 52
**Figure 11.** SSD1963 painting modes .......................................................................... 66
**Figure 12.** Table of conversion times for an ADC in STMPE610 ......................... 70
**Figure 13.** The position of FatFS layer in the system ............................................... 74
**Figure 14.** Detailed view of FatFS internal architecture ........................................... 75
**Figure 15.** State machine of text viewer function (without error treatment) .......... 77
**Figure 16.** State machine of file manager function (without errors) ................... 79
**Figure 17.** Player state machine (without errors and fast forward) ..................... 83
**Figure 18.** SD-card absent message ......................................................................... 84
**Figure 19.** File manager view and interface .............................................................. 85
**Figure 20.** System message ....................................................................................... 85
**Figure 21.** Browsing a directory ............................................................................... 86
**Figure 22.** Text application view ............................................................................... 86
**Figure 23.** Viewing a text file .................................................................................... 87
**Figure 24.** System message regarding too big text file ............................................ 87
**Figure 25.** A music folder shown by file manager .................................................. 88
**Figure 26.** Music player interface ............................................................................. 88
**Figure 27.** Muted sound icon in music player ......................................................... 89
**Figure 28.** System’s state machine from user’s perspective .................................... 90
**Figure 29.** Efika MX Smartbook ............................................................................... 116
**Figure 30.** Efika MX Smarttop .................................................................................. 116
**Figure 31.** Arduino Uno, as shown on official Arduino website ............................ 118
**Figure 32.** BeagleBone Black as seen on its official page ...................................... 119
**Figure 33.** Raspberry Pi turned into a PDA ............................................................... 120
**Figure 34.** VS1005 internal structure, which offers very high level of integration 126
**Figure 35.** SHARP Memory LCD display, as seen on AdaFruit website ............. 127
1. Introduction
1.1 General purpose of this project

The main purpose of this project is to develop an early prototype of a commercial product, in this case, an MP3-player. The novelty consists in that this project aims to develop a firmware that is fully open source and is also targeted to become a commercial product for a company that aims to use open source, open hardware business model. This business model and the particular details of prototype will be discussed in the following sections of this document.

The early prototype, as I define it, is a functioning device that should have all the basic functionality expected of a commercial product, so it can serve both as a proof of concept and also as a base for future development of complete product. Its features and interface, however, are basic and are not meant to make it into a final product in a state in which they are presented in the early prototype.

It should be noted that this project will not end at the moment in which this Degree Final Project will be defended. Instead, it is expected that the development will continue beyond that with a global objective of helping future developers of similar devices and accelerate the entrance of such devices into the market.
1.2 Acknowledgments

Many people contributed to this work in some form or another. Some contributed directly, while others shaped my coding style and approach to projects like this.

First of all, I’d like to thank my project’s mentor, Juan Climent Vilaró, for offering guidance, support and precious advices during the whole development process.

I’d also like to thank my previous mentors that helped me to grow up as a developer. Those are Ricard Gavalda of software department, who helped me immensely during my initial years at Barcelona School of Informatics, and Matthew Dillon of DragonFly BSD, who introduced me to large real-life projects.

Certain pieces of developed firmware were adapted from publicly available code produced by other developers. Those are Elia from http://eliaselectronics.com/, Peter S. from http://www.mikrocontroller.net/ and, most notably, “mediocre” embedded systems engineer who goes by name Chan from http://elm-chan.org/, which developed great file systems driver called FatFS.

ST Electronics (http://st.com) must be acknowledged since their StdPeriphLib library is used very extensively in the project.

VLSI Solution (http://vlsi.fi/) must be acknowledged since they provided a great base implementation for their hardware audio codec.

Dave L. Jones from http://www.eevblog.com/ must be acknowledged as well, because his ideas and approach were the initial inspiration for this project.

Finally, thanks to Pere Marès Martí, professor from Design of Microcomputer-Based Systems course, for sparkling initial interest in electronics in me.
2. Defining specifications
2.1 The context

At the present time (2013-2014) we are inside so-called “post-PC” era where the consumer market is dominated by new type of personal consumer devices – smartphones and tablets. Those devices replace a wide range of products that previously held huge quotas of market, such as classic mobile phones, MP3-players, PDAs, portable gaming consoles, netbooks and, to smaller extent, even classic notebooks and PCs. It is expected that those types of devices will continue to dominate during the following years, however there is also a smaller, but viable market for niche devices that co-exist together with smartphones and tablets. One part of that niche market consists of devices that are aimed at hobbyists, makers, open source and open hardware enthusiasts. That is itself a new and generally successful market where several successful companies exist with innovative products (those will be discussed in the following sections).

At the same time it is expected that a new and big market will appear and flourish soon – that’s the market of so-called “Internet of Things”, which will include many very different products ranging from small wearables and consumer products, such as smartwatches and smartglasses, to huge global systems, such as smart homes and smart cities. Regardless of whether this market will really appear and whether it will be as successful as it is expected to be or not, most of important players on the market already made their movements to be able to enter it. Those players include Intel with their Galileo project, Google with their Google glass project, Samsung with their smartwatch devices and many other products or concepts that massively appeared during last year. The global industry of consumer products also seems to be interested to move in that direction, even though there are important challenges when it comes to standardization.

When we look at the economy context we mostly find a gloomy landscape. The global economic crisis is still there and it affects negatively the markets all around the globe, even though in recent year it looks like there are signs that the end of it is approaching. One way of combating it would be empowering the local economy and small companies, so that they would be able to successfully sell their products and employ local people. However, small companies always have a hard time competing with bigger players because of their lack of resources and enormous effort that is required in order to develop a modern competitive product. The technological evolution of hardware and new approaches to software development paired with new business models open new opportunities to small companies, which may allow them not only to develop and sell competitive products, but even have some advantages to bigger players.

As a conclusion, from the point of view of an entrepreneur we are living at a difficult moment for small companies, but there may be huge opportunities around the corner, powered by technological evolution and the upcoming aperture of new markets. From the point of view of an engineer we have a lot of inexpensive technology that could be used in many different ways to achieve some exciting goals. And from the point of view of a consumer we have a broad, yet uniform landscape of consumer-oriented devices where the price becomes one of the most significant factors alongside features.
2.2 Different types of consumers

The people are not uniform beings and even if there is a particular type of device that is being used by almost everyone, normally people differ a lot in ways that they are using their devices. There are people who are passionate about technology and they care about the hardware they have, and they want to get all newest software updates and features. Other people care only about satisfying their needs and it doesn’t matter to them, for an instance, what operating system they are using or what is the underlying technology of their devices. There are loyal fans of a particular technology or even a philosophical concept that surrounds technology – for, example, Apple fans that only use Apple devices or Open Source fans that care a lot about the license of products they are using. There are people that aren’t much into technology at all and they care only about the simplicity of their product of choice.

As such, it’s probably a good idea to try to, on one hand, suit the needs of maximum number of people, so that anyone would find something appealing about a product we are offering. On the other hand it is also very important to satisfy completely the needs of a certain group of people, thus specializing on this particular type of product, and producing the best product in that particular category.

Since the focus of our project is developing an MP3-player prototype, we shall focus on all those people that want to listen to their music everywhere. Here we find that generally MP3-players are on decline, because they were almost entirely replaced by smartphones and an average consumer has all her needs satisfied. However there are also people who still may want to use a separate MP3-player:

1. People who don’t use smartphones for some reason, such as that they don’t want to constantly recharge their device. They may be interested in a product that only needs to be recharged, on average, once in a week or 2 weeks.
2. People who think that using a standalone player is generally a better option than using a smartphone. People like this may pay special attention to interface (all interface entirely dedicated to listening the music without interference from incoming calls or other activity inside the device) or other properties, for example, sound quality.
3. People who need some specific characteristics that surpasses characteristics of typical smartphones. For example, people whose music collection exceeds 64 gigabytes, which is the maximum typical amount of microSD card that can be inserted inside a smartphone.

There is also another category of people that may not be interested in an MP3-player specifically, but in the openness of product itself:

4. People who are makers, hackers or hobbyists who are enthusiastic about Open Source and Open Hardware, and want to be able to change any software or hardware aspect of the product and maybe even to contribute to its continuous development and support.

This latest category of people is the one that creates a niche market for many devices based on microcontrollers and they are going to be our primary group of consumers. Other categories, however, should also have their needs satisfied, so the functionality of our product will be complete; it’s just that we’ll allow additional modifications to any part of it for anyone who’ll wish to do so.
Lastly, I shall mention that this document will focus mostly on technical aspects of the product, so there also may be groups of consumers that I can’t talk about because they can only be approached from marketing point of view. For example, there is a group of people which define themselves as “audiophiles” and claim that their main requirement is the quality of sound. However, what they believe to be affecting the quality of sound is often not supported by any factual data and what they assert is often subjective and based on their preconceptions. As such, this category of consumers may be interested in our product, but the task of making it attractive to them is not the work of engineering, but marketing department.
2.3 General specifications

Having defined the groups of consumers that our product will seek to satisfy, we can define some very general requirements about its design and nature, which will later affect our global decisions. Those are not features of product, but rather a set of general characteristics that will define the overall architecture and philosophy.

Our product should:

- Be based on Open Source and, preferably, Open Hardware
- Allow anyone to modify the software, reuse it, share it with any goal
- Allow anyone to use it from the box, just like any other consumer device
- Allow anyone to reprogram the device and maintain the philosophy that user owns the hardware
- Have a large autonomy time
- Have state-of-art functionally of an MP3 player
- Have a good build quality and also good quality of sound and screen when compared to other products of same category
- Have amount of memory that is expected from a modern device

Also the product shouldn’t be expensive, because it will lack the functionality of expensive devices such as smartphones, but it also shouldn’t be cheap, because cheap products often are made cheap by compromising their quality. Also it is very difficult to compete on market of very cheap devices and generally it’s not good if the only advantage of the product is its price. The bottom cheap devices are also unlikely to bring sensible benefits to consumers or improve their life somehow.
2.4 BIG VS little architectures

When we look at the current landscape of consumer products we can notice that roughly they can be divided in two categories when it comes to their overall architecture. On one hand we have complex devices that are very similar to PCs, because they have a very powerful CPU, a very big and complex operating systems, such as Android, and a developed ecosystem of frameworks, drives and applications. Those devices dominate the market and are mostly present as smartphones and tablets, but there are also many other types of consumer devices like this, including MP3-players. The most famous example of that is iPod Touch, which can be roughly defined as MP3-player, but actually is a portable computer that can do almost everything that iPhone can do.

Typical characteristics of such “big” architecture are:

- Being based on a system-on-chip that has both CPU and GPU.
- CPU is very fast, typically it has at least 1 GHz frequency or more, lately it also has several cores
- It has a lot of RAM, usually at least 512 MB
- Operating system is big and needs to be stored inside a massive storage device such as SSD

On the other hand we have simpler and typically less expensive devices that are based on microcontroller that don’t have an operating system or have a very simple operating system, and that typically don’t allow installation of software and are restricted in features. Most of those are typically inexpensive MP3-players that are abundant on market, “featurephones” (classic mobile phones that aren’t smartphones), and all sorts of simple electronic devices.

This “little” architecture has the following characteristics:

- It is based on a microcontroller which features a CPU
- CPU is relatively slow and its frequency is usually lesser than 400 MHz
- It often doesn’t have a GPU or has a very simple GPU
- It has small amount of RAM, often at most just several megabytes
- Operating system is small and simple, sometimes it’s just a firmware without any OS, and it is stored inside a small flash memory integrated on a microcontroller

There are also devices that are somewhere in between, such as iPod Classic, that has a lot of RAM and a relatively advanced operating system. But most of consumer devices fall directly into one or another category and there are striking differences between the two when it comes to their characteristics at all levels.

Most of big players on the market routinely choose the “big” architecture. The reason is that this type of architecture gives their products the maximum amount of possible features. Having an advanced operating system like Android gives to product endless possibilities since it gives it access to huge Android ecosystem. The advanced operating system can also be used in a wide range of possible form factors ranging from classic MP3-players to wristwatches.
However, the “big” architecture carries an important disadvantage – it’s only a big company that can make a truly unique and well-developed product based on it. In spite the apparent fact that an advanced operating system with a huge ecosystem simplifies things, on practice it makes the development much more difficult (often, impossible) and raises the price of final product. The market is inundated with similarly-looking smartphones and tablets with same interface and features. Those types of devices aren’t exciting and their only strong point is price. The best-selling products with different interfaces and meaningful, unique features, all come from big companies that have resources to invest in development, such as Apple, Samsung, Sony and others.

Smaller companies can’t do that, because development of a new interface or a driver for some unique hardware is a very difficult task that can only be accomplished by a group of highly skilled developers. The amount of source code in any advanced OS is just abysmal and then debugging and endless prototyping just make the things worse. Even if a small company can achieve this, it often will result in a product with bugs and some features not quite finished at the date of release. Even the big players can’t often avoid having those types of problems, but they, generally, can allow themselves to take the time to fix them. Smaller company won’t be able to do that.

Another problem that comes with the “big” architecture is that the one who enables the technology will be the one who’ll control the product and make most profits out of it. For example, if we decide to use Android for our product, then we’ll be tied to the decisions of Google regarding its evolution and we won’t have control over the significant part of our product, which is operating system. Of course, Android is Open Source, so in theory we can rewrite it for our needs, but in practice developing an OS is way too difficult task even if we heavily re-use someone’s code. Then Android is not entirely Open Source either, because it’s tailored to use Google’s services which aren’t Open Source and it’s an almost impossible task to replace them [1]. The only company that accomplished this is Amazon with their Kindle Fire tablets and it’s a big company that invested a lot in development.

So for a small company, if they choose Android the most likely result would be a generic Android-based product that would be entirely tied to and dependant on Google. It would be too difficult to make this product unique in any way, even though it could be a nice product in the end, but there wouldn’t be anything special about it.

Even if instead of Android another operating system would be used like Linux or Firefox OS, it would require the same amount of development time since they are also very advanced systems. They wouldn’t tie the company to any specific developer, but they would also contain more bugs than Android since they are less extensively used in devices like MP3-players and they would lack the huge Android’s ecosystem. So their advantages over Android would be equally outweighed by their disadvantages.

Another disadvantage of a big OS is that it carries a lot of functionality and drivers, frameworks, and all sorts of things that will require very powerful hardware to make it run. Android, for an instance, makes use of a virtual machine to just run programs. It has device drivers and kernel that are as complex as drivers and kernels for PCs. Therefore a product based on that sort of operating system will have to have very powerful hardware and huge amounts of memory will be spent to just hold the system on device.
This will increase the cost of device for functionality that doesn’t necessarily require all of that powerful hardware. And it is true that there are also very cheap Android-based phones or tablets, the cheapest of which normally cost around 50 euros on market. But the quality of those devices is, generally, very low. Tablets like those normally offer a slow processor and GPU, low quality screen and low quality sound tract. While in theory they will offer all functionality that any Android tablet can offer, in practice they often struggle to even load their operating system and they are almost unusable for most tasks. For other tasks, such as music listening, they will offer lower quality of sound than a similarly priced device could offer which specializes in that task and doesn’t have all that powerful hardware or advanced OS.

So, while in theory a cheap Android-based player or tablet offers remarkable functionality, in practice it simply doesn’t deliver on any of it. Thus we can conclude that Android-based devices and, generally, devices based on any advanced operating system, should be in middle- and high-tier categories, with prices around, at least, 200 euros to be really usable.

Naturally, the need for powerful hardware also goes against the autonomy time. Modern CPUs used in smartphones and tablets are almost as powerful as some CPUs used in PCs some years ago and, naturally, they can’t run for extended periods of time, otherwise they would empty battery very fast. That powerful hardware is one of the reasons why smartphones and tablets need to be recharged at least once every 2 days for an average user (other reasons being screen and wireless modules).

Another problem is development of applications for systems like Android. Even though it is claimed that it’s not difficult to develop on Java or other high level language, the reality is that programming for those kinds of operating systems requires a lot of experience. Particularly, in case of Android the framework is huge and because there are so many different versions of the system and different devices, the development of serious applications is often a very difficult task even for most experienced programmers. This would go against our goal to focus on hobbyists and non-professional users that would like to extend the functionality of our device.

As we can see, the “big” architecture has a lot of advantages when used by big companies, but in case of small players it may carry more disadvantages, especially for the kind of product that we are developing. On the other hand, the “little” architecture provides a couple of interesting advantages.

First of all, it’s much easier to develop for a “little” architecture. While most architectures use a language like C or other similar languages for developing firmware, the programming is generally much easier than systems programming for more advanced architecture. It is often even simpler than developing regular applications for PCs, because the developer only needs to know basics of programming and target hardware. No deep knowledge of algorithms or frameworks is required. Experience shows that a lot of people who are not involved professionally with programming or electronics are capable of developing applications for microcontrollers and many of them enjoy it. This means that development will also consume fewer resources than development for a “big” architecture.
Second reason is that systems based on microcontrollers tend to be significantly less expensive than systems based on powerful SoCs. For example, STM32F407ZG costs, at maximum, 8.10 dollars [2] and that microcontroller alone is capable of running a firmware we need. On contrast, one of the less expensive “big” SoCs – Freescale’s iMX6SL, costs, at least, 8.36 dollars [3], which may seem very similar, but in order for it to be able to run something it will need a separate RAM module and a persistent storage, like hard drive. So the total cost of the system will be higher with “big” architecture.

A significant advantage of “little” architecture is its ability to achieve certain functionality with less resources. For example, you can play an MP3 using a simple microcontroller from Microchip that runs at just 8 MHz [4]. On the contrast, to do the same with the aforementioned Freescale iMX6SL you would have to have an operating system with several libraries and drivers and a separate media player program. With Microchip’s device this can be achieved with a very small firmware that won’t exceed 64 KB, the size of a block in some file systems.

This also translates into another advantage which is energy consumption. While it’s difficult to directly compare the consumption of “big” SoCs and microcontrollers, because the energy consumption of “big” SoCs is normally presented for low power modes where they are very similar to microcontrollers [5]. But thinking about the consumption of the whole system, it seems obvious that a “big” SoC running an advanced OS which resides in RAM would consume more energy than a microcontroller, at least during the CPU burst cycles when it loads OS, an application or when some function is called which goes from a high level library down to the low level of some driver.

The major disadvantage of “little” architectures comes from the very fact that they are little. A system based on a microcontroller will never have the versatility of a “big” system. It will not be able to play videos encoded in modern formats like h264. It won’t be able to let user play 3D games with cutting-edge graphics or offer internet browsing with Adobe Flash. But it doesn’t have to do all that and it can be fully functioning and good device which will perfectly suit the needs of a particular user.

So, in conclusion, a system based on “big” architecture is perfect for a versatile device designed and sold by a big company. A big company has enough resources to make a unique and well-designed device. This device, to be really versatile and have a good quality, by definition has to be a bit expensive to provide both enough computing power and good quality screen and other components. It’s not trivial for user to modify the programs and operating system that run inside that device.

On the contrast, a system based on “little” architecture is perfect for a device that is specifically designed to serve just a couple of specific functions. Even a small company can design a unique product based on it. This type of device isn’t rock bottom cheap, but it can be very inexpensive and still provide good quality of materials and components. Users who know the basics of programming and electronics can, in theory, modify the firmware and suit it to their particular needs.

Our product, being an MP3-player and thus specific-purpose device with a focus on hobbyists, makers and Open Source enthusiasts, would clearly benefit more from a
“little” architecture. The additional aperture of new markets which we’ll analyze in the following sections would add another, non-technical, incentive to choose this architecture instead of “big” architecture which has a dominant presence on market.
2.5 The business model for the future product

In order for a product to become successful it should aim to “stand out of the crowd” not only technologically, but also conceptually. In our case, our product will stand in between two different types of products – a conventional consumer product and a product aimed at hobbyists and makers community. As such, it will also embrace Open Source and Open Hardware philosophy.

This will be discussed in more detail in Section 7, but for now we will briefly describe the basics of business model for this product. There are “classic” products that are widely available on the market, such as MP3-players and other types of consumer products. Those types of products try to offer the maximum set of functionality and comfort of use. Users can’t change anything when it comes to functionality of product and, in fact, an attempt to modify the software or hardware of the product will normally void the warranty. In spite of that, there are many people who willingly modify the functionality of those products, sometimes using so-called “hacks” or alternative firmware created by hackers or enthusiasts. One example of such alternative firmware is RockBox, which specifically targets popular consumer MP3-players, such Apple iPod Classic series, Cowon’s players, etc. [6] This type of firmware often adds new functions to the product, some of which are useful (for example, older Apple iPods are able to play .flac) and some are just “for fun” (for example, the addition of Doom II game, the classic 1st person shooter, to iPod Classic, that works, but is hardly playable anyway [7]).

In recent years another type of products appeared, targeted at enthusiasts and hobbyists, that normally have some knowledge about electronics and programming, and would love to change something in their device or, in fact, make one from scratch. For those types of users the ability to change something in their devices is the main incentive to have one, more than any set of functionality that comes with it by default. In fact, some devices don’t have any functionality at all and it is expected that it is the user who will make it work. It is common for such users to exchange their source code, often licensed under an Open Source license, as well as other knowledge, such as schematics, documentation and FAQs. Those users often actively participate in what is called hackers or makers scene, and they often meet and develop together on hackathons and other types of specialized conventions. Most of those users have electronics or computer science background, but some of them have nothing to do with any of that in their ordinary life. Many of them are artists or other people that have nothing to do with science or engineering, who want to develop something in their free time because of very different reasons.

In recent years there was an array of relatively successful products of that category. While none of them achieved a commercial success similar to, say, iPhone, many of those products were indeed very successful and since most of them were produced by small companies, those were able to generate significant benefits and maintain themselves strongly in spite of competition from big players and generally hostile economic context. The most successful product of that type is Raspberry Pi, a small inexpensive computer that became widely used in many areas from education to home entertainment [8]. Another great example is Arduino, which is an inexpensive microcontroller platform that is not only used by many enthusiasts, but also has found its place in the industry [9].
Most of those products, however, have a disadvantage – they are often impossible to use out of the box, at least, until a significant development was done by community of enthusiasts. Thus a person who doesn’t know much about programming or electronics wouldn’t benefit much from buying an Arduino, unless she would also know in advance that she would have to learn a lot.

Our product aims to break the border between classic and those new types of products and be equally suited to both usual consumers and makers. To do so it has to work out of the box, thus it has to come with fully functioning and complete firmware that doesn’t require any additional action to be performed by user. In that regard it will be just like any common MP3-player, it will have a nice stylish casing and it will be ready to work once unpacked.

However, it will also have an USB-cable and PC-software which will act like a bootloader. Once connected to a PC, this software will allow erasing the firmware and uploading another one in .hex format. Once a new firmware is uploaded the device will automatically reboot and execute it. In other words, it will act like a developer board, except that it will have a case and function by default as an MP3-player. Obviously, the warranty is not voided in case that user erases the original firmware; in fact she would be encouraged to experiment and write her own software. Also, the user can always download the firmware from the Internet and re-upload it in case she would like to return to default firmware.

Moreover, the code of firmware would be licensed under an Open Source license, so she could simply modify it the way she wants and share it with everybody else. She could even fork it and start her own development branch. Users would be encouraged to write their own modifications or completely different versions based on the firmware or starting from scratch if they prefer to, and share their code via company’s website or any other sharing means, such as github. Of course, they also can form development groups if they wish to. The company wouldn’t stop the development either, but would continuously improve it and incorporate the best improvements from users into the main code branch and, thus, in updated versions of firmware. By doing this our product would appeal to both common users and makers, and everyone would benefit from that.

Furthermore, we can motivate the development of new functionalities for our product by offering monetary incentives to the best developers, organizing contests, participating in hackathons, etc. It would be essential to build a loyal community of developers around our product and to encourage development and ideas sharing.

But it wouldn’t stop there. The product’s casing would be made to be easily opened by user, so that she could extract and use the board inside it. The unused pins from microcontroller would be exposed, so that she would be able to use them for an entirely new purpose, for example, connecting them to another board and starting a completely new project based on our product’s board. Used that way our product would be a development board that could be used for an infinite array of projects. This also implies that our product would also apply the Open Hardware philosophy. While most of microcontrollers sold by vendors like ST Electronics are closed, at the very least schematics and original design files for the board would be provided, as well as the description of everything it contains.
As we can see, that way our product could be used for means that we, as its initial designers, would have never imagined. And that’s what is great about Open Source and Open Hardware. We are just selling our hardware and service to our users, but they can do whatever they wish with their device. By building a community of developers around it, it can evolve into something more than an MP3-player and be perfectly usable for other tasks. The common users would benefit from having an array of options to choose from as a firmware, and still have the company’s firmware in case that something goes wrong. The makers would enjoy developing new functions or entirely new firmware, or using our board as a base for their projects. That way we can offer a truly versatile and unique product and not just another MP3-player.
2.6 Choice of license

Since we already made the choice to use Open Source firmware in our product, we should also decide which specific license we should use. There are many different open source licenses right now [10], but generally the choice would be between a GPL-style license, such as LGPL and GPL itself and other types of Open Source license, like the MIT license.

GPL is the most popular Open Source license featured in some of the most successful open source software, such as Linux kernel, KDE, Mozilla Firefox, etc. This license carries with it certain political baggage, because not only it defines that the original source code is open and protects its developers from potential liability, but also forces that any software derived from that source must also be licensed under GPL terms. This insures that, for an instance, companies that decide to use Free Software (as Free Software Foundation defines it) have to contribute back to Open Source community if they decide to add new functionality or create derived software.

While this had a positive effect in many cases and contributed to a success of many products, it also potentially carries an important disadvantage with it. That disadvantage consists in that once software is made Open Source under GPL terms it will always be software under GPL terms. This potentially may become very problematic for a company that originally develops the software and then decides for whatever reason that they want to close it, for example, to protect their intellectual property. While this is never expected to happen in our case, we also can imagine a situation when our original plan for being an Open Source company doesn’t go well and one option, undesirable, but possible, would be changing the business model. We would run into huge trouble in case our software was originally licensed under GPL, because then we wouldn’t be able to close our derivative software unless we had full copyright over that software. That may not be the case if by that time the software incorporated many additions from makers and developers outside the company, also under GPL terms. Even though there were cases when GPL license was revoked, generally there is no need to go into that trouble anyway. While GPL serves great for many products, it’s possible that it is not the best choice for our product.

So instead of using GPL our code will be licensed under BSD 2-clause license that protects developers from potential liability, establishes copyright and at the same time gives to everyone a complete freedom when it comes to what to do with the code. BSD license is used in many successful products, most notably in BSD family of operating systems such as NetBSD, OpenBSD, DragonflyBSD… Operating systems from BSD family are used in many commercially successful companies. Most notably, they are used by Sony in their latest Playstation consoles, Netflix servers, WhatsApp servers…

It should be noted that GPL is more appropriate license if the company intends to make money on software. However our company intends to sell hardware, so that using BSD license is perfectly logical.
2.7 Basic goals of the project and required functionality of prototype

Based on everything said before, we can define basic goals of our current project which is to develop an early prototype of our audio player. Required functionality of prototype is also described here.

When it comes to general goals, we primarily have the following ones:

- Create an early prototype with basic functionality, which will serve as the basis for possible future development
- Define the general aspects of hardware and software architecture in the early prototype, which will be preserved in future development
- Investigate possible problems in order to avoid them in future development
- Create a high quality, well documented code which can be used by others for any purpose, such as development of other projects, prototypes, education, etc…
- Create examples and functions that should be ready for use in a similar prototype and thus potentially help other developers
- Use only Open Source or, at least, freeware tools during development process and not rely on any closed source component

When it comes to functionality of prototype, it should have:

- A simple graphical interface, since a screen is a must-have in most mid- and high-tier products
- Have a touch screen interface, because this is the current trend in interfaces of consumer products
- A function to browse folders and files on some sort of memory device
- Be able to play, skip and change volume of audio files
- Any other functionality which may showcase possible appliances of future product or some characteristics

As we can see, our project has fairly simple functionality and it’s essential to achieve it during a development of an audio player with touch screen interface. However they are made a bit more difficult because we always need to use Open Source during the development and we have to not only develop the firmware, but also analyze its advantages and disadvantages when it comes to architecture.
2.8 Methodology of work

I’ll follow the methodology that I had successfully applied in other relatively big projects, such as adding compression feature into HAMMER2 file system [11]. This methodology consists in that the code is created by an iterative process in which the code is enhanced each iteration.

First of all, we investigate the theoretical aspects of what we need to do. Sometimes this is unnecessary, but in case of a protocol that I didn’t work on before or some other technology which is new to me, I spend some time investigating it and trying to understand how it actually works. An in-depth knowledge is not always necessary and thus can be avoided, but generally it is never a bad idea to understand how things work.

Second, an informal algorithm would be created to achieve the desired functionality. At this stage it is important to define an algorithm that wouldn’t be too slow in any case and would be correct. It doesn’t have to be very detailed, but it shouldn’t have any dark spots that are not obvious at all to implement.

Third, the actual code would be written. At this stage I don’t worry about code that isn’t very efficient or repeats itself, or uses too many variables. The main goal is to just get it working according to the algorithm. The documentation is written, but it’s short and private.

Forth, the code is tested, debugged and continuously improved. We get rid of magic numbers, parts that repeat themselves, try to reduce the number of variables and apply optimizations where possible. The main goal is to create a clean-looking and efficient code. The cleanliness of code is more important than optimization if the latter introduces obscurity.

Fifth, the code is documented and we try to write down the theoretical aspects, the algorithms and everything that may be unclear to a person who reads the code afterwards. We don’t fear extensive commentary and try not to use one-liners.

After that we go to implement something else, but we can always return to the code if it’s necessary and improve it. In Open Source, the code is never formally finished and the process of development can be infinite.

This iterative process of code development can take more time than more aggressive code-developing techniques, but I find that the produced code is typically more robust and has more quality. It’s generally said that quality is more important than quantity and if the code is easier to maintain and understand, it will save more time on long run than in case if it was created faster, but then it is more difficult to understand or it takes more time to debug.

When it comes to coding style, I try to use the same approach as in the PC-programs. I try to make code that is object-oriented and high-level, and I try achieving performance by algorithm design rather than using obscure tricks and hacks, even though those can be useful too in some situations, like when we lack resources. I also try to not use assembler at all in order to make code more portable and because with modern compilers there isn’t any significant difference between high-level code and code
manually written in assembler. Also high-level code is much easier to understand than assembler, so usually there isn’t any reason to use assembler at all. So generally my approach to writing firmware is the same as my approach to writing conventional programs for PCs. I must note that many people from electronics background have a different approach regarding this.
3. The hardware
3.1 Unification VS master and slaves

Once we selected the “little” architecture we can start defining its more specific characteristics. One of most important and general decisions is the choice between having a single chip which does everything or almost everything in the system, or having a central chip that acts in a way similar to CPU in PC and several slave chips that accelerate some specific tasks, and thus help the master chip.

The current trend in electronics is to try to integrate as many things as possible inside a single chip and reduce the number of chips in the overall design to minimum. The reason for this is that by reducing the number of chips we are reducing the cost of the system. Each added chip adds the cost of the chip into the overall cost, and one old technique to reduce the costs is to try to put into a single chip as much functionality as possible in order to avoid having a separate chip for some specific functions.

This trend is especially notable in device with “big” architecture, where it is common to integrate into the same chip a CPU, GPU and, recently, even a LTE-modem, DSP-processing and more. One example of that is Qualcomm Snapdragon 801 [12]. The vast majority of “big” architecture SoCs integrates more than one function on a single chip, even the low-tier ones, like Freescale iMX6SL [3].

In “little” architecture devices this trend is also present. Many vendors of microcontrollers try to integrate a lot of functionality in their high-tier products, for example ST Electronics has ChromeArt Accelerator and display controller in their high-end microcontrollers. Another design trend is that when you have enough computational power to perform some function by software in a main chip, there is no need to have a specific chip to do the same function. This brings to market a lot of low-cost devices that are based on a single chip that either integrates a lot of functionality inside or has enough CPU capabilities to perform all functions by software.

This approach is perfect for low-cost devices, however if we are making a device that isn’t aiming to be specifically low cost and we also want to make it suitable for hobbyists and makers, so it may be a better choice to have several specific chips performing various functions and helping the master chip. There are several reasons for this.

First reason is the complexity of programming. If our product plays MP3, then we can use a software decoder and then send decoded stream into a DAC and, later, into an amplifier. However, MP3 decoding isn’t a trivial process and it’s an abstract algorithm that may be difficult to implement for novice programmers or amateurs. While the functionality of playing MP3 may be very interesting for a certain project, implementing the details of MP3-decoding algorithm may not be very interesting and, in fact, viewed as an annoying obstacle. So, it may be a good idea to either supply a library for MP3-playing function, or to provide a specific chip which will handle that task itself and free developers from implementing it in software.

Second reason is the performance. In our earlier example, if we’d decided to supply a library, then the process of decoding an MP3-file would be using the main chip and, thus, reducing the amount of available resources. That may put additional constraint on
a project that is being developed and prevent it from having some additional functions. It may also force the developer to use obscure hacks or tricks that will make the code less clear.

The performance is especially important aspect when it comes to driving displays. Displays are very fast devices and supporting a graphical interface is one of the most resource-consuming tasks in both microcontrollers and even PCs. In microcontrollers the things are especially aggravated because of significantly reduced amount of RAM which is clearly insufficient for having a framebuffer that is essential for any basic interface functionality. Because of that it is very common to have a separate display controller even in the devices that aim to be low cost and having just one chip, if those devices have a graphical display. It is often impossible to get rid of display controller, especially if the main chip is 8-bit or 16-bit controller. Even if the main chip is a powerful 32-bit controller with a big amount of RAM, it’s still preferable to have a separate display controller for any kind of advanced interface.

In most cases, the display controller is simply providing a framebuffer memory with some additional very basic capabilities, such as image rotating. In that case the display controller simply receives the information to be displayed on screen and stores it in its memory. It iterates endlessly and flushes the content of memory on screen as the result of a single iteration. Many of inexpensive and commonly used controllers are of that type, such as popular SSD1960/SSD1961/SSD1962/SSD1963 display controllers [13].

There are more advanced display controllers that offer the possibility to store fonts or perform more advanced image manipulation, for example, hardware support for having a cursor (instead of controlling it purely by software inside the main ship). One example of such controllers is SSD1906 [14]. Those controllers typically are more expensive, but they significantly simplify development of interfaces, especially when it comes to fonts manipulation.

Finally, recently a new trend had appeared that consists in that some display controllers basically became simple GPUs. This means that they try to handle the whole interface and support high-level commands for rendering 2D and, sometimes, even 3D objects. With that kind of controller even an 8-bit microcontroller can have an advanced interface. It is also common that those types of controllers support some additional capabilities, such as audio decoding or microSD-card interface. One example of such display controller is FT800 [15]. This controller is used in an Arduino-based game console called “Gameduino 2” where it provides outstanding graphical capabilities for an 8-bit microcontroller [16].

Generally, all those display controllers implement their functionality by hardware, but there are also some controllers that are, in fact, a general-purpose microcontrollers with a specific firmware that makes them drive the display. One example of display controller of that type is SmartGPU 2, which is a generic ARM Cortex-M3 microcontroller that has a firmware which processes graphics by CPU rather than by some specific hardware [17].

Aside from display controller, another common type of slave controller is a touch-controller. It simplifies significantly the process of retrieving touch data and liberates
the main chip from polling or processing data from different ADCs. It is often integrated inside display controller in order to reduce costs.

So when choosing between those two types of architecture for our device, we must take note of the following general characteristics about them:

One-chip architecture:

- Less computational resources overall, which may result in more restricted functionality
- Developers may be forced to use non-obvious hacks in order to achieve certain performance, the code is more prone to become obscure
- Lower overall cost of system
- Less energy consumption

Master and slave chips:

- Simplified programming
- More resources available, since additional chips reduce the load on main chip
- Higher overall cost of system
- More energy consumption

Given the nature of our device and also given that our company is, probably, relatively small, the second type of architecture seems to be more appropriate. That way we will reduce the load on our main chip, simplify the code and make the development easier for less experienced programmers. This will increase the costs, but our device isn’t supposed to be cheap and we will have fewer functional restrictions from start, because more available computing power may result in more advanced functionality.
3.2 The choice of specific microcontroller architecture and vendor

After taking a very important decision in previous section, we must take another important decision on our main chip, namely, which family of microcontroller architectures it will be from. This decision is very important, because it will affect greatly the performance, the coding style, the available tools and the cost of the device. As such, we shall not base our decision only on traits of architecture, but also on other parameters such as the availability of chips, their price, and the simplicity of programming them.

There are very many families of microcontrollers from a wide array of companies like Freescale or Cypress, and it’s not possible to compare all of them within scope of this document. Moreover, most of them are targeted only on specific markets and have very expensive development tools. So instead I’ll just focus on some of the families that have good availability and acceptance in hobbyists and makers community.

**AVR and ARM-based microcontrollers from Atmel**

AVR is very popular in community of makers thanks to the Arduino project [9]. However, the microcontroller used in Arduino boards is very simple and relatively slow. Besides, we’re making an original product and not an Arduino board, even though making another board for that project would be potentially interesting product. However what we are aiming at is creating a completely new product with more possibilities.

Another option would be AVR’s 32-bit microcontrollers and also Atmel’s ARM microcontrollers. AVR’s 32-bit microcontrollers, however, have only up to 512 KB Flash memory and up to 64 KB of RAM memory [18], which is not much compared with solutions from ST Electronics we’ll take a look later.

ARM-based microcontrollers from Atmel [19], on the other hand, have up to 2 MB of Flash memory and up to 160 KB of RAM memory which makes them a very interesting candidate. The maximum frequency for those microcontrollers is 120 MHz and they are based on ARM Cortex-M4 architecture [20]. Since the ARM microcontrollers from Atmel are the most powerful from theirs microcontrollers line, we’ll make them our primary candidate from Atmel.

**Microchip’s microcontrollers**

Microchip is a very well-known company among everyone involved with electronics. Their PIC architecture is robust and time-proved, and their microcontrollers are very popular among hobbyists and makers, because they are powerful, inexpensive and come with free development tools (even thought those have some limitations).

Even though they are mostly known for their 16-bits microcontrollers, they also have very powerful 32-bit chips. The most powerful current chip, PIC32MX795F512L, is featuring 512 KB of Flash memory and 128 KB of RAM. Its frequency can be up to 80 MHz [21]. Those are good characteristics, but numerically those numbers are lower than characteristics of ARM microcontrollers.
It should be noted that Microchip is currently preparing to fully compete with ARM-based microcontrollers. They will soon release a new family of 32-bit microcontrollers called PIC32MZ EC family [22]. This family will feature, most likely, very powerful microcontrollers, the most powerful of which will feature such outstanding characteristics as 2 MB of Flash, 512 KB of RAM and will have maximum frequency of 200 MHz [23]. Neither this or any other microcontroller of that family is on the market at this moment, so we can’t consider them as possible candidates at this moment, but they will be very interesting once they’ll arrive.

**ARM-based microcontrollers from ST Electronics**

Those microcontrollers are also quite famous in community of makers and they are relatively new entries on the market, however they already have many fans, development boards and free development tools. There are two main families – inexpensive 8-bits STM microcontrollers and more advanced 32-bits microcontrollers [24].

Among those 32-bits microcontrollers, the most advanced ones have up to 2 MB of Flash memory, 256 KB of RAM and their maximum frequency is 180 MHz [25]. They are based on ARM Cortex-M4 architecture. Overall, this makes them the most powerful solution offered among those we saw up until this moment.

**Other microcontrollers worth mentioning**

There are other vendors and families of microcontrollers that can’t be candidates for our product for a number of reasons, but I believe that they are worth mentioning since they can become serious alternatives to solutions I’ve mentioned above in the future. There are 3 interesting alternatives for the future…

The first one is VS1005 platform from VLSI. VLSI is Finnish company that specializes in hardware audio codecs, which are quite popular in community of makers and are often used in many hobbyists’ projects. Their new VS1005 platform, however, is different, because it is actually not a slave device that it is meant to support a main chip, but it is a stand-alone solution that doesn’t need a master microcontroller. Instead it is itself a complete microcontroller that also integrates audio decoding capabilities [26].

This platform is very interesting, because it integrates Touch Screen controller, interfaces to SD-card and internal memory, FM-receiver, etc., so basically it is effectively one-chip solution that perfectly fits our needs, because there is no need to implement in software audio-decoding algorithms. The only thing that needs to be added to it is a LCD-controller chip, but other than that it would be very highly-integrated one-chip system.

There are, however, a couple of problems that prevent this solution from being appropriate for our device at the time. First of all, our goal is to create an Open Source device and we don’t know whether VLSI would be willing to open their VSOS operating system or accept an alternative open system. Second reason would be that this solution is new and the community of makers didn’t really work much with it for now. Third reason is that its hardware characteristics are inferior to other microcontrollers which I presented above, which means that a product based on it would, possibly, be
somehow restricted in its functionality. However, it is an interesting alternative that
definitely can be considered in the future.

Second possible alternative for the future is the TIVA platform from Texas Instruments
that appears to be getting some traction in community of makers. However, it is a new
player in that community and their technical characteristics aren’t outstanding. The most
powerful microcontrollers from that series have 1 MB of Flash memory, 256 KB of
RAM and their maximum frequency is 120 MHz [27]. So it could be an interesting
alternative to look into in the future, especially if it will become more popular in the
community, but for now we’ll rule it out.

Finally, third alternative is the new Galileo platform from Intel. This new platform
represents Intel’s move to the market of microcontrollers and their plans to dominate the
market of Internet of Things, and it also targets the community of makers [28]. Its main
advantage seems to be the raw power – its Quark SoC offers impressive characteristics
like 400 MHz of frequency [29]. However, this platform is really new and Intel is a new
player on this market, so it may be less reliable than products from companies that have
been on that market for years. So we won’t consider it for now, but it definitely may be
an option in the future.

**The comparison table between selected candidates**

Discounting the possible candidates for the future (VS1005, TIVA and Intel’s Galileo),
here is the table that summarizes the characteristics of different microcontroller families
that may be chosen for our future product.

<table>
<thead>
<tr>
<th>Family</th>
<th>AVR 32-bit</th>
<th>SAM4</th>
<th>PIC32MX</th>
<th>PIC32MZ EC*</th>
<th>MF40/41</th>
<th>MF42/43</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company</td>
<td>Atmel</td>
<td>Atmel</td>
<td>Microchip</td>
<td>Microchip</td>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>Architecture</td>
<td>AVR</td>
<td>ARM Cortex-M4</td>
<td>PIC</td>
<td>PIC</td>
<td>ARM Cortex-M4</td>
<td>ARM Cortex-M4</td>
</tr>
<tr>
<td>Frequency</td>
<td>84 MHz</td>
<td>120 MHz</td>
<td>80 MHz</td>
<td>200 MHz</td>
<td>168 MHz</td>
<td>180 MHz</td>
</tr>
<tr>
<td>Flash memory</td>
<td>512 KB</td>
<td>2 MB</td>
<td>512 KB</td>
<td>2 MB</td>
<td>1 MB</td>
<td>2 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>64 KB</td>
<td>160 KB</td>
<td>128 KB</td>
<td>512 KB</td>
<td>192 KB</td>
<td>256 KB</td>
</tr>
</tbody>
</table>

* Not on market

**The choice of platform**

Based on the summary of characteristics from the table above, I chose
STMF40/STMF41 family from ST Electronics. This family seems to offer most power
among the families from competing vendors currently presented on market, and thus it
is perfect for a product that aims to offer the maximum functionality and freedom to
user. At the same time it is less expensive than STMF42/STMF43 and since it offers
better characteristics than competitors, for now we’ll go with the less expensive option.
But, since those families are closely related, they should be very compatible and we can
possibly offer an upgraded version of the product in the future, which will use STMF42/STMF43 instead of STMF40/STMF41.

I should note that even though we choose ST Electronics for this moment, the offers from other vendors are also attractive and any of them could potentially be the base platform for our product. Since we don’t have very strict specifications and our product doesn’t have a specific characteristic-based purpose, but rather aims to offer a big variety of options, any of possible candidates could be used. In fact, for many potential users just the fact of being based on a specific platform would be an advantage or disadvantage, since all of those platforms have their fans and, in some cases, anti-fans. STM32 seems to be the platform that gains more fans every day, so that’s also an important reason to choose them, but in the future, if there is enough demand, it could be interesting to offer different versions of the product based on different platforms.
3.3 The choice of development board for the prototype

When a prototype is developed for a commercial product, normally the company that develops it makes or subcontracts someone to make its own prototype board to test different components, board designs, software and cases. Before that it is also common to create prototypes from components, which are sold separately, using techniques such as wire-wrap or soldering.

In case of smaller companies, student projects or hobbyists’ projects, it is also common to use development boards designed by other companies to test software, components, and concepts, or just use them as a base for a project. In our case, we have 2 alternatives – try to make a prototype from components sold separately or to use a complete development board that has everything or almost everything we need. To save the time and reduce to minimum the probability of hardware bugs, I decided to go with the second option. It was also influenced by the fact that my background is mostly from computer science and not electronics.

My board of choice is Mikrolektronika’s board called Mikromedia⁺ for STM32 [30]. This board had the exact architectural traits that were defined for our product. It uses the main chip as its base, but it also has numerous slave modules, such as screen controller and audio codec. Its screen is pretty big, so the size of a prototype device roughly corresponds to size of actual MP3-players and smartphones. The resolution of screen is also bigger than resolution of screen typically found in most boards. Overall, the design seems to be very appropriate for an MP3-player and, specifically, for a product like ours.

I should note that, however, this board was chosen for an early prototype which aims to simply be the proof-of-concept and provide a base for further development. For next prototypes it would be absolutely necessary to make specific, tailored to our product, boards with different components in order to choose the most appropriate ones for us cost- and feature-wise. While it is enough for hobbyists projects to have a development board like this, for a commercial product it would require additional effort and freedom of choice to develop a truly innovative and unique product. We wouldn’t rely just on design from development board vendors for any serious product.

However, for the purpose of the current project, this board provides more than enough functionality and it is perfect for the purpose of our development. By having a complete, professionally designed board we can spend time directly on firmware development instead of soldering components and potentially dealing with hardware bugs. For small companies that would also save time and money on early stages of development, and could also serve as a reference design for their own product.
3.4 The general traits of hardware architecture in the prototype

As we mentioned earlier in section 3.1, our architecture consists of a main chip (base microcontroller) and a number of slave devices (which we also call modules) that help the main chip by accelerating specific processes. On general level, this hardware design is shown on scheme below.

![Diagram of the abstract hardware design of the prototype](image_url)

**Figure 1. The abstract hardware design of the prototype**

As we can see, only 4 slave devices are used in the current design. The screen controller serves primarily as framebuffer and holds the information that is currently shown on screen. It receives the information from the main chip and shows it on screen immediately. It can also transfer the information to the main chip if it’s necessary, but currently we don’t use this functionality. The arrow is bi-directional though to reflect the bi-directional nature of the protocol.

Screen controller simplifies greatly the code related to the interface, since we don’t have to worry to maintain the image on screen – the screen controller will do this for us. If it has an advanced functionality, such as 3D-capabilities, in theory we can also use it to enhance the interface or to add additional functions related to graphics.

The touch controller retrieves touch data from display and greatly simplifies the development, because, once configured, it manages automatically the ADCs used to get the touch point data and it stores it inside its internal memory. Those ADC values then can be easily retrieved when necessary. In our case, the main chip checks periodically if there are new data available in touch controller FIFO-buffer.
The audio codec is the module that decodes and plays audio files. To play the audio file, the main chip simply has to send it, part by part, to the audio codec. The audio codec will then decode it. It also gets the decoded audio stream through internal DAC and audio amplifier, so this module directly plays the audio if it’s connected to an audio speaker or headphones. It simplifies greatly the audio playing functionality and, depending on particular model, can have some DSP-capabilities. The arrow is bi-directional, because the main chip can retrieve data from the audio codec, and some audio codecs also have recording capabilities, which we don’t use currently.

microSD card module doesn’t really have a slave chip, but any microSD card has a microcontroller inside, which retrieves data on demand from the main chip. When connected, both chips will have to negotiate the protocol they will use to communicate. Once the communication is established, we can view the microcontroller inside microSD card as a slave device. The arrow is bi-directional, because the data goes both ways. We don’t use the write capabilities at this moment, but in any case we send orders to microSD card.

Display is simply an LCD screen which displays our interface. It also has to have touch input capabilities in our case, so it is an input/output device. For user, the screen provides the only way to interact with the system in this prototype. It doesn’t have any type of microcontroller inside (it’s a “dumb” display) and it is connected directly to the screen controller. The touch input capabilities are physically provided by a thin layer glued on the screen which is directly connected to the touch controller.

Speaker simply denotes the headphones or actual speaker/speakers connected to the audio codec via standard 3.5 mm audio connector. This module is external to the system and it’s not a part of this project or even the final product. This component greatly affects the audio quality, but we assume that it’s responsibility of user to choose best headphones or speakers for her needs.

Other modules denote possible other chips or devices that we are not using currently, but may use in the future. As an example, such devices could be LEDs to inform users about some activity, physical buttons, temperature sensor, etc.

We should note that the current architecture allows us to experiment with different components for each module in different prototypes before arriving to the best combination for our final product. For example, we can test different displays while maintaining same other modules, as long as screen controller and touch controller can handle them. We can also test different audio codecs from different vendors or different models within the same vendor. In many cases the connections would be very similar or even the same, so there wouldn’t be major modifications to the electronic design of prototypes.
3.5 Detailed description of all hardware

In this chapter we’ll see in detail what actual components are going to be used in our early prototype. Those components are located on board and they were chosen by Mikroelektronika’s designers, however they fit extremely well within the defined architecture.

![Diagram of the prototype with specific hardware and data transmission protocols](image)

**Figure 2. The actual design of the prototype with specific hardware and data transmission protocols**

As we can see, the system is mainly composed of the following modules:

- STM32F407ZG, master device
- SSD1963, slave device, screen controller
- STMPE610, slave device, touch controller
- VS1053, slave device, audio codec
- microSD card, slave device, external storage

The speed of transmission protocol indicated on figure is always the working speed after the slave device was fully configured. Normally, this speed is the maximum achievable speed with those devices, however sometimes it can be increased a little bit, which wouldn’t affect much, in any case, the functionality and perceivable speed of the system.
STM32F407ZG

This is the main chip of the system and, thus, it affects enormously everything else. This is where our firmware, developed in this project, resides and executes. Its speed, arguably, determines the perceivable speed of the whole system.

STM32F407ZG is not the most advanced microcontroller from STM32F4 family, but it is a high-end device and its CPU is the same as in more advanced ones, so it should have the best available performance within that family. It supports several data transfer protocols and has many integrated devices such as DACs and ADCs. The characteristics that are the most important for this prototype are the following:

- ARM 32-bit Cortex-M4 CPU with hardware FPU
- Frequency up to 168 Mhz (this is the frequency used in firmware)
- 1 megabyte of integrated Flash memory for firmware and data
- 192 kilobytes of RAM (+4 kilobytes of backup RAM)
- Several 16-bit and 32-bit timers

The complete list of characteristics can be found at product’s webpage on ST Electronics website [31].

SSD1963

SSD1963 is a screen controller designed by Solomon Systech that achieved certain popularity and is used in many products. From a technical point of view it’s a generic screen controller that has few features, but it has a lot of memory for framebuffer and it is very fast. It also uses a wide bus protocol that allows fast data transfer and so it can be used to show advanced graphical content on screen. Its main characteristics [32] are the following:

- Built-in 1215 kilobytes framebuffer
- Supports up to 864 x 480 display resolution with 24-bit color
- Supports various data transfer interfaces/protocols (24-bit, 16-bit, 8-bit, etc.)
- Programmable brightness, contrast, saturation and backlight control
- Simple image modification functions, such as screen rotation, color inversion, etc.

As we can see, SSD1963 is a fairly basic screen controller and we can’t rely on it in any, but the most basic graphical processing. If we want to create complex graphical effects, such as scrolling, resizing of objects, etc., those should be handled by STM32F407ZG. We can say that graphic processing is mostly done by software, rather than by hardware. Still, SSD1963 simplifies a lot the management of the screen. To show something we simply should send it once to SSD1963 and then it will hold it in its memory automatically and display it on screen as soon as possible. When we want to change the contents of screen, it’s as simple as rewriting SSD1963’s memory.
As we can see on Figure 3, the main chip can send up to 24 bits of data in a single transfer, but in this prototype we only use 16 bits. The screen, as we’ll see later, does not support 24 bit color anyway. The main chip can also configure the device by writing into its registers. It’s absolutely necessary to configure the device before using it. Some parameters of configuration depend on characteristics of screen.

Pins CS#, D/C#, RD#, WR# are used in transfer protocol to select the device, indicate the type of data to be transferred (command or data), etc. The exact details of protocol will be discussed in software section 4.5. CONF pin is not used in this design and TE pin is connected to the main chip, but currently not used (we ignore its value). RESET# can be used to perform hardware reset on device.

The content of each pixel is directly mapped into SSD1963’s memory, so writing to screen is as simple as sending to SSD1963 the coordinates of first pixel to be painted and of the last one, thus defining a rectangular working area, and then sending the color of each pixel consecutively. SSD1963 will automatically increase its internal pixel counter, so we can send the content of next pixel immediately. It makes it extremely simple to paint lines and rectangles on screen.

In our prototype the device runs at its maximum frequency – 120 MHz. That should be enough to display without any lag the complex content on screen, as long as the main chip and the screen are sufficiently fast.

It should be noted that Solomon Systech offers very few information on their products. On the official page of SSD1963 there is no publicly available datasheet or application note [34], and when I requested the datasheet my request was ignored. This may suggest that their products are not the most appropriate for a device based on Open Source philosophy, unless our company will be able to publicly distribute the datasheet on its own.
STMPE610

STMPE610 is an advanced touch screen controller by ST Electronics. Its task consists in measuring current in several wires with the help of several ADCs and storing the values inside its memory buffer. When a value or several values are requested by the main chip, it will transfer them one by one. It can also automatically detect the touch, if necessary, so the main chip may only request a value when a touch is actually detected.

It should be noted that this controller does not automatically convert the values into the actual placement of touch measured in pixels. The main chip will have to perform that conversion. The chip, however, can limit the touch area of the screen to the certain parameters, so in case that the touch-sensitive layer is bigger than the screen, we can limit the touch area exactly to the screen’s size. We don’t use that functionality though. The chip can be configured in a number of ways that will greatly affect touch sensibility and accuracy. We’ll talk about that in more detail in section 4.5.

![STMPE610 internal architecture and connections](image)

Figure 4. STMPE610 internal architecture and connections [35]

As we can see, very few pins of the main chip need to be used to connect to this device. In our case, since we are using I2C, we only need 3 pins. We have found, however, that I2C imposes significant speed limitations on obtaining values from the device.
The most important characteristics of this device are the following [36]:

- Possibility of using SPI or I2C protocols, both supported by hardware in STM32F407ZG
- 12-bit ADCs to convert voltage into numbers
- 128*32 bits buffer
- Interrupt output pin

The actual conversion time per one complete value (X and Y) is 10.9 ms in our case (we calculate this value on page 69 in software section). It is a considerable amount of time, but this whole process does not affect the main chip, so this time is not lost. If we had decided not to use the touch screen controller, then the main chip would handle this processing itself and while it wouldn’t have to wait all this time, the whole process would be more complex and cumbersome.

VS1053

VS1053 is an audio codec device by VLSI. It is one of their most advanced controllers, even though there is a more advanced controller now – VS1963. Its main task is to automatically decode and play an audio file, where audio can be in various formats. It can also record audio and compress it into various formats, even though for now we don’t use this functionality.

![Figure 5. The internal architecture of VS1053 and its connections][37]

As we can see, this device contains inside a complete audio tract that should be connected directly into the headphones or speakers. Thus, there is no need to have a separate DAC and audio amplifier in our system, which reduces the cost. There is an option, though, to use external DAC if necessary and only use this device for decoding. For now we are not considering this option.
The decoding capabilities of this device are based on its proprietary VSDSP4 processor. This processor acts like a DSP and few things are known about its internal architecture. However, judging by its internal architecture shown on Figure 6, it looks like it is actually, a general-purpose CPU, which was designed to be capable enough and most appropriate for audio decoding tasks.

![Figure 6. Internal architecture of VSDSP4][38]

This architecture has several interesting consequences. First of all, since the decoding is not entirely done by hardware and the designers of this processor specifically designed it to be that way; it allows us to partially reprogram it to fix some possible bugs in current decoders.

Second, we can add to it some additional functionality via so-called plugins. This can be used to add all sorts of effects to audio and even add the possibility of decoding some additional formats. VLSI shares many of those plugins for free and encourages users of this processor to write and share new plugins. Those plugins don’t carry a specific Open Source license with them, but the whole approach is very Open Source-like.

Third, in theory this controller can even be used as a standalone device instead of being slave device. We can load a firmware instead of plugin, which will be able to drive peripherals connected to chip via GPIO pins, and thus we can create a very simple system based entirely on VS1053. It doesn’t seem to be an intended use though, but this possibility is explicitly mentioned on product’s webpage [39].
VSDSP4 works with the help of an external clock. Its frequency should be 12-13 MHz or 24-26 MHz. This speed, however, is too low to decode certain audio streams, so this frequency must be multiplied. In our case, the processor runs at 43.008 MHz, which is enough to decode all intended formats.

To play the file the main chip needs to transfer it to VS1053 by chunks of 512 bytes in size. There is no need to strip the file of its header or other non-audio data; in fact VS1053 needs it to identify the format. The actual sound will appear as soon as VS1053 will have enough data to start playing the audio. The main chip needs to supply the data fast enough for continuous and uninterrupted play. If the main chip is not fast enough, then the sound will either be interrupted if there is significant delay or it will sound slower than it should and very badly. STM32F407ZG has more than enough speed to guarantee uninterrupted playback and also to maintain a fairly complex user interface on touch screen. As we can see, in general the algorithm is very simple. The details of it and the actual implementation will be discussed in software section.

To communicate with the main chip VS1053 uses SPI protocol. With the current frequency of VSDSP4 the transfer frequency is limited to 10.752 Mhz. Given the hardware limitations of STM32F407ZG the actual frequency of transfer is 10.5 MHz. This is more than enough for transferring audio files, but it imposes certain delay which should be taken into account.

The most important characteristics of VS1053, including the formats available for decoding, are the following [39]:

- Ogg Vorbis, a popular Open Source format
- MP3, the most popular audio encoding format
- MP1 and MP2, the legacy audio formats
- AAC, an advanced and modern audio format
- WMA, a proprietary audio format from Microsoft, very popular in some Asian countries
- FLAC, a popular and open lossless audio format, requires free plugin
- WAV, lossless audio format, which is mostly used on audio CDs
- MIDI, a synthesized audio format
- Proprietary EarSpeaker Spatial Processing
- Zero-cross detection for smooth volume change
- High-quality DAC with 18 bits resolution [40]
- Total Harmonic Distortion <= 0.07% [40]

Those characteristics deserve to be explored in more detail. This controller supports the vast majority of formats that are actually used nowadays. The only major format, that is absent, is APE, a lossless highly efficient format, which is free to use, but it is closed. It’s possible that the support for this format can be added with another plugin, but the difficulty and viability of that are unknown.

When it comes to supported formats, Ogg Vorbis is a very popular format in the community of Open Source enthusiasts, so it is a must-have in our device. MP3 is the most popular audio format up until this day, and WAV is a standard format used in audio CDs, so they both are must-have as well. MP3 requires additional license to be acquired, and it must be included in the final cost of the device. AAC is modern and
highly efficient lossy format. Some people claim that at bitrates of 160 kbps it is undistinguishable from lossless audio [41]. This claim may be disputed, but in any case supporting this format is a very nice feature for any modern MP3-player. AAC also requires a license.

MP1 and MP2 are old legacy audio formats. While they don’t add substantial value, it’s always nice to have an additional feature. WMA is a proprietary audio format from Microsoft, which is rarely used in Europe, but it is very popular in some Asian countries, such as South Korea. This format also requires a license, but since we aim for a global market (by this I mean that, at least, we can ship worldwide and we are open to users and developers from all countries), it’s a good idea to have it as well. FLAC is the most popular lossless format nowadays, so it’s a must-have too.

When it comes to supported flavors of those formats, this codec doesn’t support all of them, but it supports all sane values. For example, FLAC supports up to 9216 kbps bitrates (192 KHz/24 bit mode), which is absolutely insane, requires huge computational amount of effort to decode and doesn’t bear any improvement in quality. In fact, I would argue that 44.1 KHz with 16 bit resolution is more than enough and it is probably unnecessary to go beyond that for any sort of listening purpose.

After discussing audio formats, we should talk about sound quality and characteristics of VS1053 that highly influence it. For users that are not interested in Open Source aspects of our device this is, perhaps, the most important question of all. Nowadays almost all of MP3-players available on market come with some sort of proprietary technology to help them to enhance sound or to reproduce it more accurately. Often those technologies are most highlighted in commercials, and they influence greatly decisions of potential clients.

VS1053 has a couple of very nice technologies too. First of them is EarSpeaker Spatial Processing. This technology aims to create an illusion of listening to music not through headphones, but through speakers located at a certain distance from user. It gets rid of the sensation that the music is localized inside listener’s head and creates a more pleasant listening experience that also doesn’t affect the actual tonal character of the music (thus, preserving the original sound) [42].

![Figure 7. Sound from external sources vs. inside-the-head sound](image_url)
When user listens to music through speakers, it’s better to turn off EarSpeaker spatial processing, because the source of music is physically external in that case. Since we can’t automatically distinguish between those situations, it’s desirable to have an option to turn it off/on by demand from user.

Zero-cross detection for smooth volume change is a technology that allows to change the volume without interruptions and very smoothly. It gives an impression that the sound changes continuously until a desired value without going through rough and substantial changes in volume. In our prototype the value is changed by very small steps – 0.5 dB. That, combined with this technology, results in a very pleasant volume change, smoother than in the commercial devices which I have (an iPod Classic and Sony Xperia SP smartphone). So this feature is definitely very interesting and can be unique when compared to other devices.

High-quality DAC with 18 bits resolution is more than enough for very high quality audio reproduction. 16 bits is the lossless standard for bit depth which allows to record the entire dynamic range where the music is located (approximately 98 dB), thus going beyond that bit resolution per sample will not give any benefit for listening (contrary to claims of some audiophiles). The detailed discussion of this would be too lengthy and it would go outside the scope of this document, but I will refer an article which explains this in more detail [43].

Finally, Total Harmonic Distortion for the overall device is in the worst case 0.07%. This figure is the amount of distortion when it starts to become unnoticeable, so while it’s not the best figure I saw, it’s enough to aim to be a quality standard.

Subjectively, the sound produced by VS1053 is on par or even better than the sound produced by an iPad Classic and Sony Xperia SP. They seem to sound differently, but this is normal and doesn’t necessary mean inferiority in quality. Also, the opinions on sound quality of VLSI’s products are very favorable. Overall, this means, that sound produced by our system will be on par with the sound produced by some of current high-end MP3-players.

When talking about VLSI’s products such as VS1053, I must note that VLSI as a company made a very good impression on me personally. While this document is mostly centered on investigating technical specifications, which is a job of an engineer, I believe that treatment offered by vendor’s staff can also be a very important factor, especially if we are talking about smaller companies that don’t have huge amounts of resources to back them up in any aspect. Also since our product is aimed to be open to the users and third-party developers, it’s especially good when a vendor of silicon chip offers such a good treatment to developers, even when they didn’t buy directly their product.

VLSI has a large collection of plugins and code examples that can help a lot in development. The code examples are very well written and documented; they offer a great example of well-made code. The company’s documentation is also one of the best I’ve seen while working on this project. In addition to all of that the company has a forum where they offer free technical support to people who work on projects that use their products. Company’s staff is very helpful and responsive. It’s rare to be able to talk to actual developers of chips in such a simple way.
This makes me inclined, at the very least, to highly recommend both the company and their products to any developer or maker.

**SunBond LB04302 LCD Module (AT043B35-15I-10 in some documentation)**

This is the LCD screen installed on Mikroelektronika’s Mikromedia+ for STM32. Its most important characteristics for our project are the following [44]:

- 4.3 inches screen
- 480 x 272 pixels resolution
- 262K colors [45]
- Minimal refresh time around 11.2 ms, typical 16.8 ms, maximum 40.5 ms

This screen is very appropriate for our project, because its size is similar to size of a typical smartphone – it’s not too big, neither it’s too small, even though it’s possible that for many people the smaller design would be preferable. The aspect of the best size could be an important aspect of research in next prototypes. But I think that generally working with bigger screen is more pleasant and it gives more potential uses to the device and as long as it’s thin enough, there shouldn’t be any problem with carrying it just like a typical smartphone.

The resolution is among the biggest that can typically be found in most development boards, but it’s clearly inferior to resolution that most people are used to in their smartphones of mid- and high-range. Working with it, I didn’t find any problem with that resolution on the screen of that size, but the difference between that screen and the screen in a typical mid-range smartphone is very easy to notice.

*Figure 8. The photo of the screen showing graphical content, by AdaFruit Industries [47]*

The general feeling about quality (such as brightness and colors) is better than what I expected. It doesn’t produce any major negative effect on eyes, the graphics and photos look generally well and it is as fast as most of typical PC screens. However, it’s
probably a good idea to search for alternatives, because while it’s an acceptable screen
which would satisfy most of users, generally the people are used nowadays to more
advanced screens, with higher resolution, more colors and better overall quality.

It should be noted that this screen is used in Gameduino 2 project – an Arduino-based
gaming console, and it looks very good there, so generally this screen is indeed
appropriate for an entertainment device [48].
4. The software
4.1 Single software approach VS OS-based approach

Before starting the developing, we should make certain decisions regarding software architecture. The biggest decision here is whether we’ll be developing a typical firmware, which will contain a single multifunctional program, or we’ll make use of an OS, which we may develop ourselves or take some OS licensed under some Open Source license.

Single-software approach is used in most firmware for microcontrollers. This approach is based on that the microcontroller-based device will have only one or few uses. Thus it is more than enough to have a single program, without any OS, running there and accomplishing that function or those few functions. In that sense the system lacks any sort of multiprogramming or process management. It’s just a single program that works directly on hardware.

Normally this approach is more than enough for vast majority of uses for microcontrollers. Even if there are several functions or peripherals, which require constant attention, the program can manage them efficiently using some sort of simple state-machine. Also it can make use of interrupts, so if something requires immediate attention, it can be done too. So many firmware developers avoid using an OS and instead just go with interrupt-management and state machines inside a single program.

An alternative to this is OS-based approach. It consists in that we are using a very simple OS, usually real-time OS, which does, in certain way, the same things as normal OS for PCs. It manages several processes, which are used to implement the functionality of device, and the available hardware resources. Normally it contains some sort of task scheduler, which is usually extremely simple, and some sort of API which simplifies the development of “applications”. It also may contain drivers, if necessary, and advanced functionality, such as graphical interface.

This approach is normally used when the system is very complex and has diverse functionality. When there are different possible functions with competing priorities, it becomes very difficult to manage them though an obvious state machine. More complex algorithms are required to dynamically manage tasks and resources, and, thus, an OS is required.

It should be noted, though, that often in microcontrollers the line between a single-software piece and an OS with several process is very thin. Complex software may use a complex state machine and treat its own functions like processes. It may even contain entire libraries for its own purpose. On the other hand operating systems made for microcontrollers tend to be very simple and they usually are some sort of state-machine accompanied by drivers and libraries. When compiled, it’s often difficult to distinguish one from another, even though OS normally allows more flexibility to user at the cost of additional complexity and code/firmware size.

Sometimes an OS for microcontrollers can become more complex and even execute processes from external memory. However generally they are somewhere between a complex one-piece program and almost PC-like OS. Also they generally lack advanced
features such as virtual memory, because there is no hardware support for that kind of features.

Both approaches have their drawbacks and advantages. One-piece software has the following characteristics:

- It’s easier to develop when the system is of low- or middle- level of complexity
- It’s more tailored to a specific project, which normally results in reduced code size and better efficiency
- It may be difficult to make changes in specific functions depending on how it was developed; if there is no clear separation between different parts of software, one change may provoke many other changes
- It’s more difficult to frequently add different functions, because there are no tasks or threads
- It’s very difficult to implement concurrent functionality in most cases
- If the system’s complexity grows, it can result in a very cumbersome and obscure code

OS-based approach has the following characteristics:

- It’s redundant for systems of low- or middle- level of complexity
- It can be easier to port
- One kernel may be used in very different products, even with different architectures
- It’s easier to add new functions or introduce changes, because it’s simply about making a new task or modifying existing one
- It may allow features that are similar to features from more advanced systems, for example a process execution from external memory, which is great for developing additional applications
- It may use a stack-based approach, similar to used in Unix-like systems (a separate layer for interface, which may allow to easily switch interfaces, add drivers, etc., all based on the same kernel)
- Developing is difficult, it requires a lot of experience and wide vision of operating system’s architecture

Our case here is difficult, because our product certainly doesn’t require an OS to be functional, but it would benefit a lot from having it, especially from the point of view of having extensible and easily modifiable design and functionality. Also the possibility of executing applications from external memory would be really interesting for both users and developers.

After thinking about it, I decided to take a step back and settle on a single-piece software. The reason is that this is an early prototype and thus the main goal of this project is to achieve a minimal set of functionality and provide a base for future development. Developing an entire new operating system would add an unnecessary amount of complexity to this project. Developing of an operating system can be another project altogether by the amount of work it requires.

I also took a look at the already existing implementations of operating systems for microcontrollers and there are several good candidates to be used in our product. One of
them is NuttX, a free and open OS licensed under BSD-license [49]. However, I
decided not to use an off the shelf implementation because they all seem to be complex
and there is always a risk of running into bugs or problems with specific
implementation, which would, again, add an unnecessary level of complexity into this
project.

So, this prototype will use not an OS, but multi-purpose single software. I must note
that doesn’t mean that the code developed here will be useless for future development if
it will be based on an OS. As I said before, sometimes the line between multi-purpose
software and an OS is very thin in microcontrollers, and thus I tried to separate
functionality in several layers and libraries, that can be easily re-used in other software
or an OS, if that’s necessary. I also tried to treat some functions as if they were separate
tasks. I’ll describe this and other decision in detail in next section.
4.2 General traits of software architecture

After deciding to use just one-piece software instead of an OS in the previous section, now we should design architecture in concordance with characteristics defined previously. That is, even though we are not developing an OS, it’s a good idea to try to separate our software in several layers, which are independent from each other and can be re-used in other software. The general structure of our software should look like this:

![Software Architecture Diagram]

**Figure 9. The software architecture of prototype**

Booting stage is the first code that gets to be executed when the system is turned on. It configures internal hardware of the main chip, STM32F407ZG, and then jumps to main(). The things that are configured during booting stage are very basic and they directly influence the performance of the system. For example, the frequency is set at that stage and we set it to the maximum speed (168 MHz). This configuration can be changed later while executing, but currently we don’t do that.

In main() the configuration continues and it is finished. This time the peripherals are configured (such as I2C) and basic I/O pins configuration (such as whether they are configured as input or output). This configuration may change while the system is running. Then the software goes into a main state machine state, which is represented by an infinite loop that determines what the system should do next depending on an input from user. At this point main() calls a function which will represent a certain “application” or state of the system.
By default, main() calls a function called “file_manager()” which allows user to browse files on the microSD card. If the card is not inserted or malfunctioning, main() will display a message asking to insert the card and will try to mount it repeatedly. Aside from “file_manager()” there are functions that represent different applications, and the user may launch them by trying to open a specific file in the system (or, potentially, by performing some other action, such as clicking on an icon).

Those “big functions” are quite complex at this point, because they try to replace an entire application which is often composed of various functions. Internally, all of them consist in setting and executing another state machine implemented by another infinite loop. The difference with the infinite loop in main() is that the user may perform many different actions, which will affect the state of the system and she can also leave the current application, thus leaving from the infinite loop. User never interacts directly with the state machine in the main(), instead it just launches an application depending on previous input, or just waits for some event to occur (currently, the only event is insertion and successful mounting of microSD-card).

Both main() and “applications” use different drivers and libraries. By driver I mean a file which contains a set of functions to interact with certain hardware, such as display controller or touch controller. Those functions are accessible and can be used from everywhere in the system. Sometimes the term driver can also refer to an abstract device or entity, such as file system. Libraries, on the other hand, represent collections of useful functions that may be used in different parts of the system by different “applications”.

Also both main() and “applications” have access to shared resources. Those are not the code, but shared data, currently embedded into the firmware. Generally those are icons that can be used in interface in different parts of the system and fonts, that can be used anywhere where text output is required (currently the system contains only one Arial-based font).

I should note that currently there is no such thing as a unified interface layer in this system. Instead each application should build its own interface. There are certain parts that are reused, but those are just simple structures that can’t aim to present any sort of unified interface API. I decided to do it that way, because designing a unified interface API is a very complex process which isn’t directly related to the main purpose of this project. Moreover, it is not difficult to build an interface on a microcontroller since your software is targeting a specific, previously known, device. So there is no need to have a unified API for it, even though having it would simplify further the development. As such, it is also responsibility of the programmer to maintain coherence in the interface between different parts of the system.

As you can see, the overall architecture of the system is very reminiscent of the architecture of an OS-based system. Instead of main() we would have an OS-kernel, the “applications” would be the actual real applications, and the rest would be the same or very similar. Obviously, an OS would be much more complex, but it also would have a couple of important advantages, such as the possibility of executing several applications simultaneously. For example, at the current state we can’t play an audio file and go to browse file system to choose another file in a different folder, because in order to do that we would have to exit a player “application” and launch a file manager
“application”. The only way to achieve that would be adding audio playing capability to file manager, which would increase its complexity.

On practice, the system software’s architecture turned to be not quite like it was planned initially. Even though generally I succeed in separating functionalities and conceptually different parts into different files that are clearly independent from each other, in one particular case it was inevitable to merge an “application” and a driver. This is the case of player “application”.

The reason for this is that VS1053 is, essentially, a hardware-implemented player and the “player” application is simply a wrap-up around the hardware player. But it doesn’t have a separate API or framework dedicated to VS1053, instead the only thing that it has to do to play a file is simply send it to VS1053. Thus the only work of the player “application” consists in sending the file, setting up the interface, and also changing the state of the hardware player if user wants to (for example, to change the volume level).

Since sending the file in order to play it is a real-time process, we have to do it continuously and we can’t do that job outside the application, because the same execution thread also should check for the input from user. Therefore there is no way to separate the driver of VS1053 from the player. However, it would be preferable to have them separated for simplified development of alternative players. The problem is that this would require the implementation of the driver as a separate thread or process and currently we can’t do that because we don’t use an OS.

Alternatively we can try to make the system periodically check if it needs to send a file to VS1053 using interrupts and a timer. This approach would require having a global structure that would maintain the player’s state and a process that would send the file and then be able to switch to another file. This approach would possibly work, but it also could potentially create some exception cases and make the system very cumbersome from the architectural point of view. Probably a better idea would be just switching to OS-based approach, which would provide more flexibility overall, and not just with the player.

So, the actual architecture after finishing the software for the prototype is shown on scheme on the following page:
Even though the initial plan for the architecture wasn’t fully accomplished, the player “application” is the only instance where the “application” and the driver are merged into one. So, generally, I would say that the core of the initial intent was accomplished and it’s normal when there are some changes in the initial architecture layout related to particularities of hardware, even though we must try to avoid them when possible.

In the following sections we will talk about the contents of each file in the system and the details of implementation. Before that, we’ll also talk about the developing tools and coding style.
4.3 Development tools

When it comes to development tools, since we aim to create an Open Source product, it is logical that our development tools should be, at the very least, free, and preferably licensed under an Open Source license. For this reason compilers and IDE from Mikroelektronika were discarded from the beginning, because they are closed and relatively expensive products [50]. They can be used by other developers, if they prefer them, of course.

Instead I chose to use GNU compiler collection which has an easy to install compiled version specifically for ARM Cortex-M4. It worked great for this microcontroller and this board, and no problem was whatsoever encountered with that compiler and any library I used in the project.

As my IDE I chose free and open CooCox CoIDE, an IDE based on famous Eclipse IDE, which specifically targets development for ARM microcontrollers [51]. It has many built-in tools to simplify the development. For example, it has a very complete collection of libraries and code examples, which allows easily re-use someone else’s code, such as drivers for certain hardware. The integration with GNU compiler is also very easy. It just requires setting a path to the compiler in the options. With detailed instructions regarding how to install the toolchain even the beginners will easily do it, and they’ll have complete and powerful development platform without any restrictions whatsoever.

As for the bootloader, it comes for free with Mikroelektronika’s board. It’s extremely simple to use and it is very robust. It doesn’t have any restrictions, so even though it’s not licensed under Open Source, it complies with our requirements and can be used without any problem.

Sadly, this toolchain is essentially for the Windows platform. There is no version of CooCox CoIDE for Linux or MacOS, and there is also no such versions of bootloader from Mikroelektronika. It looks like the development tools for microcontrollers generally seem to target only the Windows platform and this is a long-lasting “tradition” in the world embedded development. This is not a very huge problem, because we can replace CoIDE with Eclipse or other IDE in Linux, and for our own product we would have to write an alternative bootloader anyway, but it’s indeed a bit disappointing.

With this toolchain I had a complete development environment for Mikroelektronika’s Mikromedia+ for STM32. The only software that needs to be purchased in it is Microsoft’s Windows operating system, if it can be counted as a part of the toolchain.
4.4 Coding style

Since this project aims to produce high-quality code to be released under BSD-license, I decided to define certain characteristics for the code before the development even started.

First of all, I decided to avoid using assembler. Even though there are still some developers in the community that are pushing the development using assembler, with the modern compilers there is really no reason to use it at all or almost at all, except specific circumstances.

Second, I decided to use C, because it’s the language used the most in the industry and with which most people in the community of hackers are familiar with.

Third, I decided to use the same object-oriented style which is used in programming for PC or smartphone applications. Many people who develop firmware for embedded systems are trying to write directly into microcontrollers’s registers to configure it or use other low-level tricks and hacks to perform certain small tasks. While this normally increases the firmware’s efficiency, really the gain is very small and in most cases completely irrelevant, especially with such high-performance devices as STM32F407ZG.

Moreover, it often makes the code incredibly difficult to understand, especially by beginners, and it often results in more bugs and increased development time. As such I decided not to use those tricks at all and always use a high-level approach.

Forth, I decided to document extensively the code within the code itself, so that any new developer could understand it without reading any additional document. If the comments result to be lengthy or contain some theory, that’s ok, because an experienced developer may simply skip them or delete them, but a beginner would find a lot of useful information there and wouldn’t have to search the internet or read other document to understand how system works.

Finally, I decided that for an early prototype I would choose the simplest approach that achieves the minimal level of functionality. Therefore I don’t claim that my code is the most efficient solution among all possible, neither it should have been. It simply aims to work and be usable by an average user, so it also has to be fast and efficient enough, but as long as it complies with essential requirements for efficiency, the next condition for it is just to be simple and easy to understand. Therefore some solutions presented in this firmware are trivial, but they just serve as the foundation on which more efficient and robust structure can be built.

Other than this, I also decided not to use very long lines and try to use self-explanatory names for variables, which may lead to them being longer than names typically used in other software.
4.5 Detailed code explanation

In this section we will take a look at all of the code files and describe their functions and internal details. The complete list of files and components of software is the following:

- main.c
- Filesystem layer
  - diskio.c
  - diskio.h
  - ff.c
  - ff.h
  - ffconf.h
  - integer.h
- SD card driver
  - stm32_eval_legacy.h
  - stm324xg_eval_sdio_sd.c
  - stm324xg_eval_sdio_sd.h
  - stm324xg_eval.c
  - stm324xg_eval.h
  - stm32f4xx_it.c
  - stm32f4xx_it.h
- StdPeriphLib
  - STM32F4xx_StdPeriph_Driver
    - Contains many files...
  - stm32f4xx_StdPeriphLib.h
  - stm32f4xx_StdFramework.h
  - stm32f4xx_StdFrameworkConfig.h
  - stm32f4xx.h
- apps.c
- apps.h
- ascii.h
- built_in.h
- command_listLCD.h
- delay.c
- delay.h
- imafix.plg
- images.h
- lcd.c
- lcd.h
- player.h
- player1053.c
- registers_listTouch.h
- rgb_led.c
- rgb_led.h
- touch.c
- touch.h
- utils.c
- utils.h
- venc44k2q05.plg
- vs1053b-patches-flac.plg
Not all of those file are actually used in the software. For example, rgb_led.c and rgb_led.h are files that provide examples of usage of the external LED, but currently they are not used in the system. Having them in the project doesn’t increase the size of the produced compiled software, because if the function is not used, then it is simply omitted. Thus, we can have some examples or additional functions directly embedded into the code and they wouldn’t affect at all the rest of the system as long as nothing uses them.

Similarly, we don’t use the whole StdPeriphLib, but it doesn’t hurt to have it all provided in the project, because that way any developer would directly know what functionality this component can provide and can start using any part of it right away.

In the rest of this section we’ll review each relevant part or file starting with main.c, then examining the shared data embedded in firmware, and then taking a look at each driver and library. Finally, we’ll finish with “applications” and the player. The description of code is not exhaustive, but it touches on each of most relevant lines in the code and most of the omissions are self-explanatory.

**main.c**

This is main body of our software. It is relatively simple and executes very small amount of time overall compared with other parts of the software. It starts by configuring I/O pins that are connected to LED and the screen.

```c
GPIOLED_Init();
GPIOLCD_Init();
```

LED is used as a means for debugging and the screen, as was determined in previous sections, is a basic input/output device. After that timers and the display controller are initialized.

```c
Timers_Init();
LCD_Init();
```

Timers are needed for implementing delay function, which is essential for the correct initialization of many devices. Without it, we would have to use loops with empirically chosen values to wait until a device is ready after performing some action on it. After executing the screen controller initialization, we can immediately start using the screen. We paint it white in order to show that it works.

```c
paint_areaLCD(0, 0, 479, 271, 0xFFFF);
```

I’ll omit the description of this and other calls that show messages, because we’ll describe how they work when we’ll get to the screen controller driver. After this, the program initializes and configures the touch controller.

```c
GPIOTouch_Init();
Touch_Init();
configure_touch_module();
```
The first function configures pins, the second function checks whether STMPE610 is actually present and responds to main chip, and third one actually configures it. After performing those functions, the system is ready to receive touch input.

Finally, the software initializes and configures VS1053:

```c
GPIOVS1053_Init();
VSTestInitHardware();
while (GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_9) != 1);
int i = VSTestInitSoftware();
```

The first function, as usually, configures pins. The second one performs hardware reset of VS1053. This reset will bring all its settings to default values. Then main chip will wait until the hardware reset is fully finished and VS1053 becomes stable. We can know that, because VS1053 will raise its MP3-DREQ pin, which is connected to PD9 pin of main chip. Then the software configures VS1053 and changes its settings to values we want initially using VSTestInitSoftware().

This function, in fact, performs various tasks: it checks that VS1053 operates properly, configures it and it also loads in it several plugins and patches. It takes some time to load that information into VS1053 and, in fact, this is the function that takes most time in this whole initialization process. However, it is very fast and the overall initialization time is not slower than that of competing products, such as iPad Classic.

After this, the initialization process is finished and the system goes into its main state machine. It is implemented as an infinite loop. First of all, it checks repeatedly whether the microSD card is inserted.

```c
if (SDCard_present()) {

If SD card is present, then a file system object is created and an attempt is made to mount the filesystem.

```c
FATFS file_system;
.
.
result = f_mount(&file_system, "0:", 1);
```

Currently only FAT12/FAT16/FAT32 systems are supported. We’ll discuss the implications in the section 6.4.

If the file system is successfully mounted, then the software launches file manager, which, by default, will display the contents of root directory.

```c
uint8_t command = file_manager();
```

There are only two events that will cause file_manager() to return some value – first, if the user will choose a file to open and, second, if some error will occur, normally provoked by SD card extraction.
Currently the software just checks whether the returned value means a request to open a file. If there is an error, the system will simply leave this state and then go to check if microSD card is inserted.

If the user, indeed, picked a file to open, the software will check if it has a function to open that file. Currently the system can only open audio files that VS1053 can decode (WAV, MP3, FLAC, WMA and M4A/AAC) and plain text files. There is a specific function to check file extension:

```c
check_extension(target_file, ".TXT", 4)
```

Currently there are certain limitations when it comes to file names, which we’ll discuss in the section dedicated to file system driver. Particularly, file extensions can’t exceed 3 symbols, so instead of “.flac”, “.fla” extension is used in this code.

If the user wants to open a text file, text_viewer() function is called.

```c
command = txt_viewer();
```

If it’s an audio file that can be decoded, player function is called.

```c
int success = VSTestHandleFile(target_file, 0);
```

Currently VSTestHandleFile(…) requires an int to be returned, thus we are not reusing the command variable in this case, but that can easily changed in the future.

In case of error, a system_message(…) function is called. This function has all the system messages pre-programmed already and the caller must simply indicate the number of message to be displayed.

If the file system can’t be mounted, the software simply checks for SD card and attempts to mount the file system indefinitely. If there is no SD card, the system will be polling the SD card slot indefinitely.

**ascii.h**

This is the shared resources file which contains our font. It is always used when the system needs to write some text on the screen. The font format aims to store a high-quality font efficiently and to make it easy to visualize it. It also should be trivial to extend it with new characters. Currently the font is based on Arial 24p and it contains all ASCII visible characters.

First of all, ascii_table structure is defined. It is an array which contains the address of each ascii-symbol, in the next array which contains the font.

Then a font-array called bitmap, which contains the actual description of each symbol, is defined.

Together those arrays allow printing any symbol on screen using a very-straightforward algorithm. First of all, we need to get the address of symbol in the bitmap. To get it we can simply access the position “ascii_symbol - 32” in ascii_table array and it will
contain the address of the ascii_symbol in bitmap array. For example, the address of the
description of “a” is located in ascii_table[‘a’ - 32].

Then we can access that position in bitmap array to get the description of “a” and print
it on screen. We have to implement it that way, because in our format symbols don’t
have fixed length of description, so we have to store somewhere an index which will
allow translating the symbol into its address in the bitmap array.

The format consists in that each symbol can be represented by several uint32 numbers
in binary format. In fact, 0 means an empty pixel (with background color) and 1 means
a pixel occupied by the symbol (with symbol color). Using this, we can
straightforwardly represent any symbol, but in order to store them efficiently, we apply
several optimizations.

We know that if the font’s size is 24p, then the maximum height of any symbol is 24
pixels. The maximum length of any symbol is not defined and doesn’t have to be
constrained to 24 pixels. Therefore, we can define certain characteristics of a symbol
without wasting many numbers to describe, for example, all the empty lines that has the
description of point symbol ‘.’. Instead we can just explicitly indicate the amount of
empty lines and lines that contain some information.

Specifically, any symbol is described by 3 numbers and a bitmap. The first number
indicates the number of empty lines starting from the top. The second number indicates
the length of a symbol, since they may have very different lengths. Third number
indicates the amount of non-empty lines, and then follows the bitmap which consists in
the several uint32 numbers, each of which corresponds to a non-empty line, starting
from top.

As an example, we can look at the description of symbol ‘*’:

```
1,
9,
7,
0b00001100000000000000000000000000,
0b00001100000000000000000000000000,
0b01111110000000000000000000000000,
0b00001111000000000000000000000000,
0b00111110000000000000000000000000,
0b00001111000000000000000000000000,
0b00110010000000000000000000000000,
```

The first number indicates that the top line is empty (only one line), but then the non-
empty lines start. The second number denotes the length of the symbol, which in this
case means that there are 9 columns. The third number – 7, shows that next there are 7
uint32 numbers which make a bitmap.

Thus, to print the symbol, the algorithm paints 9 pixels of the first line with the
background color. Then it prints next 7 lines with the content of bitmap. At each
number it prints background color where there is 0 and symbol color where there is 1. It
stops when it prints 9 pixels that constitute the line of that symbol and goes to next line.
Finally, when it printed all 7 lines from bitmap, knowing that each symbol is a part of
24p font, it paints the remaining lines with background color. In this particular case after painting the star it would paint the remaining 16 lines with background color.

This format allows us to store fonts without wasting space that we would have to waste if we had used fixed size to store each symbol. Also we can still use a very straightforward algorithm to print the symbols.

images.h

This is another file that contains shared resources. In this case it contains several images in our internal format, which can be used, mostly, as icons in the system’s interface.

The format doesn’t use any sort of image compression techniques, so it’s very inefficient and only small images, such as icons for the interface, should be stored that way. It’s preferable to store them outside the firmware actually, on some form of external storage, no matter the format.

Each image is stored as const uin16_t array. The “const” keyword denotes that the array should be stored in Flash memory and not placed into RAM. First two values denote image’s resolution and the rest of values denote pixel’s color in RGB565 format, which is the color palette we currently use with SSD1963. To display an image a very straightforward algorithm is used that simply specifies an area which equals the image in size and then directly sends color value for each pixel to SSD1963.

delay.c, delay.h

Those files contain delay functions, that is, functions used to stop the flow of execution for a fixed period of time. Those types of functions are essential in many instances, for example, when the system is configuring a slave chip and must wait for certain amount of time for the slave chip to perform some task.

For many families of microcontrollers their vendors provide an official way to implement delay functions. ST Electronics, however, doesn’t have one, so each developer should implement her own version of it or take someone else’s version. Because of this, there are several competing ways to implement delay functions. The simplest of them all is simply to have an empty loop and measure how much time it takes to finish it.

In my case, however, I decided to implement delays by using timers. This way seems to me to be the most elegant and appropriate for functions like this, but I don’t claim that is the best or most efficient way to implement them.

Those files contain two functions: Delay_ms() and Delay_us() for delaying some amount of time in milliseconds and microseconds, respectively. They use different timers to do so. In order to initialize timers, void Timers_Init() is used.

First of all, we enable clock in those timers. We use standard functions provided in StdPeriphLib developed by ST Electronics.

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
After that, we configure both timers by defining values in TIM_TimeBaseInitTypeDef structure and setting it to a particular timer. This is the recommended way to do it, instead of writing directly in timer’s registers.

```c
TIM_TimeBaseInitTypeDef TimerSettings;
TimerSettings.TIM_Prescaler = 59;
TimerSettings.TIM_Period = UINT16_MAX;
TimerSettings.TIM_ClockDivision = TIM_CKD_DIV1;
TimerSettings.TIM.CounterMode = TIM.CounterMode_Up;
TimerSettings.TIM.RepetitionCounter = 0;

TIM_TimeBaseInit(TIM4, &TimerSettings);

TimerSettings.TIM_Prescaler = 59250;

TIM_TimeBaseInit(TIM3, &TimerSettings);
```

Configuration of timers is identical except TIM_Prescaler value. Since our main chip runs at 168 MHz frequency, those two timers should run at 84 MHz frequency, because of STM32F407ZG clocking scheme (it is described in detail in [52] and also in comments in the code). So theoretically the values of TIM_Prescaler should be 84 and 84000 respectively. However, I found out that the actions of checking the timer’s register and comparing the value there with other value carry a significant overhead, so if we use those values, the actual delaying time will be overly delayed. For that reason, I had to lower those values and empirically find the ones that would be satisfying for our purpose.

After configuring the timers, we just have to enable them.

```c
TIM_Cmd(TIM3, ENABLE);
TIM_Cmd(TIM4, ENABLE);
```

After this, timers start ticking and we can immediately use them. Alternatively, we can enable them just before we need them and then disable them, but this would add another delay.

Delay functions themselves are very simple and straightforward:

```c
void Delay_ms(uint16_t value)
{
    TIM3->CNT = 0;
    while((uint16_t)(TIM3->CNT) <= value);
}

void Delay_us(uint16_t value)
{
    TIM4->CNT = 0;
    while((uint16_t)(TIM4->CNT) <= value);
}
```

Those are one of those exceptional moments where we write directly into registers instead of using wrap-up functions. We do this here because we want to be as fast as possible and we can hardly make any error while setting those registers.
The first line sets the counter register CNT to 0, because it was ticking before entering into this function the whole time. Then we simply enter into a loop when we compare the register’s value with the amount of time we need to wait. When we exit the loop we can guarantee, that, at least, the desired amount of time has passed, but we can’t guarantee that precisely that amount of time had passed.

In addition to those two functions there is also delay 1 instruction function which is the only instance where we use assembler.

```c
#define Delay_1inst() asm("nop")
```

This is the lowest amount of time that can be delayed in this system and it should correspond to the execution time “nop” instruction.

**lcd.c, lcd.h, command_listLCD.h**

Those files represent a driver for SSD1963 and the screen installed in the early prototype (SunBond LB04302). The driver gives access to basic functions such as painting a pixel on the screen or painting a line or rectangle. Other parts of software should use those functions to create their interface, which can be as advanced as the developers want.

SSD1963 is connected to the main chip using 5 control pins (WR, RD, D/C, CS, RST) and 16 pins for data transmission. In this project we are only using transfer in one direction – from main chip to SSD1963, so we’ll be talking about this type of transfer only.

8 pins of 16 are used to transfer commands and all 16 pins are used to transfer data. During one transfer either a command or data can be transferred. Since we can transfer a complete color of a pixel during only one transfer, this allows us to transfer a lot of data very fast.

The protocol of communication with SSD1963 varies depending on the connection mode and the format of data. Here we’ll describe the particular mode we are using in the early prototype, which is 8800 mode with 16 bits data length (RGB565). This protocol is also described in more detail in code itself.

RST pin is used to perform hardware reset on SSD1963. When it is set to 0, it means that the hardware reset is being performed, so during normal operation it should be set to 1.

CS pin is used to select SSD1963 if it shares pins with some other devices. When it is set to 1, SSD1963 will ignore any other input, so during normal operation it should be set to 0.

WR and RD pins serve to indicate whether the main chip wants to write something into SSD1963 or receive something from SSD1963. The active state is 0, so setting WR to 0 means that we want to write something into SSD1963. Same happens with RD pin. This means that setting both WR and RD to 0 is a wrong state and this should never happen during normal operation (in fact, it can only happen when RST is set to 0). When WR
or RD is raised from 0 to 1 this means that the input/output pins with data are stabilized and SSD1963 can either fetch the data or it can assume that the main chip already fetched the data from it.

D/C pin indicates whether the data presented in 16 pins for data transfer is a command or a data like parameter or pixel color. When D/C is set to 0 it means “command” and when it is set to 1 it means “data”.

*command_listLCD.h* contains the complete list of commands for SSD1963. Some commands require a fixed amount of parameters to be transferred right after them. Other commands have an unlimited number of parameters to be transferred, particularly WRITE_MEMORY_START and WRITE_MEMORY_CONTINUE that allow an unlimited amount of pixel data to be transfer after them until SSD1963 receives another command.

Now after getting the basics of communication protocol with SSD1963 we can start analyze the functions provided in lcd.c.

*void GPIOLCD_Init()* is a function that configures I/O pins of the main chip. There is no need to describe it in detail, because the basics were described previously for delay.c. I’ll omit those pin initialization functions in the future.

Then there are a couple of functions to send commands and data to SSD1963.

*void issue_commandLCD(uint16_t command)* {
  uint16_t temp = 0xFF00 & ((uint16_t)GPIOG->ODR);
  temp = temp | command;
  GPIOG->ODR = temp;
  Set_InfoTypeToSend(COMMAND);
  Set_WRbit_low();
  Delay_1inst();
  Set_WRbit_high();
}

*void write_dataLCD(uint16_t data)* {
  uint16_t temp = 0xFF00 & ((uint16_t)GPIOG->ODR);
  temp = temp | data;
  GPIOG->ODR = temp;
  Set_InfoTypeToSend(DATA);
  Set_WRbit_low();
  Delay_1inst();
  Set_WRbit_high();
}

Those functions are an essential foundation for other functions. Other parts of our software aren’t supposed to use those functions directly, but that can be done if necessary. Here we also access our main chip’s registers to write as soon as possible into output pins.

*Delay_1inst()* is a minimal necessary delay for our main chip’s pins to stabilize the output data both in output data pins, D/C and the WR pin. SSD1963 is a very fast device, so there is no need in practice to wait after raising WR pin to 1, because in practice the
delay between one command or data transfer is sufficiently big for SSD1963 to successfully fetch and start processing the data.

void LCD_Init() is the function that initializes and configures the SSD1963 controller. The initialization sequence consists in a set of instructions that must be sent in a particular order (some of them can be sent out of presented order though). From that point of view the sequence is equal for any display. However the parameters sent with instructions can vary a lot depending on the screen. The developer should check the datasheet for the screen and change the parameters accordingly if she wants to reuse our initialization sequence.

I should note that making this initialization sequence was actually the most difficult part of development and it took more time than any other similar proceeding. The reason for this is mostly the low quality of documentation which forces the developer to guess what exactly needs to be done. Also there are few ways to verify whether or not SSD1963 receives and reacts to commands before the display is actually turned on.

There is no need to describe in detail the command sequence, because the code is generally self-explanatory and it contains the complete description of all steps within it. The only thing that needs to be mentioned is that initially SSD1963’s internal clock is slow and thus before setting the PLL we need to always give it enough time to process commands. That’s why there is always a delay after any command or data transfer. After performing the software reset there is no need to insert any delay between transfers.

Next we have a function that sets an area on which the graphical content will be displayed. We can set it to be the whole screen if we want to paint a full-screen image on it, or we can define a smaller area if we don’t need to change the content on the whole screen.

void define_paint_areaLCD(uint16_t x, uint16_t y, uint16_t x_end, uint16_t y_end) {
    issue_commandLCD(SET_COLUMN_ADDRESS);
    write_dataLCD((x >> 8) & 0x00FF);
    write_dataLCD(x & 0x00FF);
    write_dataLCD((x_end >> 8) & 0x00FF);
    write_dataLCD(x_end & 0x00FF);

    issue_commandLCD(SET_PAGE_ADDRESS);
    write_dataLCD((y >> 8) & 0x00FF);
    write_dataLCD(y & 0x00FF);
    write_dataLCD((y_end >> 8) & 0x00FF);
    write_dataLCD(y_end & 0x00FF);
}

SET_COLUMN_ADDRESS command defines the first and the last column of the area. Columns are X coordinates of the screen, meaning that there are 480 columns in total. The first one is column 0 and the last is 479.

SET_PAGE_ADDRESS command defines the first and the last row of the area, so “page” actually should mean “row” in this context. Rows are Y coordinates, so there are 272 rows, the first one is row 0 and the last one is 271.
The system allows defining values that go outside of the screen without producing any error. If the content is only partially on screen, it will be displayed without any problem. However doing so would likely be confusing and also inefficient, because main chip would still have to write pixel data for pixels that aren’t shown on screen.

Next we have the basic function to transfer one pixel color to the screen.

```c
void write_pixelLCD(uint16_t pixel) {
    uint16_t temp = 0xFF00 & ((uint16_t)GPIOG->ODR);
    temp = temp | (pixel & 0x00FF);
    GPIOG->ODR = temp;
    temp = 0x00FF & ((uint16_t)GPIOE->ODR);
    temp = temp | (pixel & 0xFF00);
    GPIOE->ODR = temp;
    Set_InfoTypeToSend(DATA);
    Set_WRbit_low();
    Delay_1inst();
    Set_WRbit_high();
}
```

Since the pin groups that control output pins to send the complete color data are different (GPIOG and GPIOE) we have to take additional care to not accidentally rewrite unrelated data in the ODR register.

The addresses of pixels should have been previously defined by

```c
void define_paint_areaLCD(uint16_t x, uint16_t y, uint16_t x_end, uint16_t y_end) function. After defining them, void write_pixelLCD(uint16_t pixel) will paint the first of them (x, y) with a specified color. Then SSD1963 will automatically increase its internal memory/pixel counter, so the next pixel to be painted will be (x+1, y). When the last column x_end of the row y is painted, the next pixel to be painted will be (x, y+1). SSD1963 will continue to increase its internal memory/pixel counter like this until the end of painting area will be reached (row y_end + 1 was reached). As I mentioned previously, this makes it very easy to paint rectangles.
```

```c
void paint_areaLCD(uint16_t x, uint16_t y, uint16_t x_end, uint16_t y_end, uint16_t color) {
    int i;

    define_paint_areaLCD(x, y, x_end, y_end);
    issue_commandLCD(WRITE_MEMORY_START);

    for(i = 0; i < ((x_end - x + 1)*(y_end - y + 1)); ++i) {
        write_pixelLCD(color);
    }
}
```

The function above is used to paint a rectangular area with a single color and it perfectly describes algorithmically the process described in previous paragraph. First, we define the paint area, which has to be rectangular, by sending coordinates of first pixel (x, y) and coordinates of last pixel (x_end, y_end). Then we send a command to indicate that now we will start to write into the framebuffer. After that we directly send the color information of each pixel consecutively until we have transferred all content.

Graphically, it would look like option 1 of horizontal mode in the following picture:
In fact, this is also how the whole screen is painted during each “refreshing screen” period. This mode of operation and the option were configured in `void LCD_Init()`.

Next we have more advanced functions to display some content on the screen. First we have auxiliary function `uint16_t get_letter_length(char letter)`.

It simply returns the length of any given symbol from the font bitmap. It may be interesting in some cases to know it beforehand to determine, for example, whether it will fit inside some area before attempting to display it on screen.

Next we have a function `uint16_t write_letterLCD(char letter, uint16_t x, uint16_t y, uint16_t letter_color, uint16_t background_color)` which is used to display a symbol from the font bitmap. The algorithm to display it was described informally in section dedicated to `ascii.h`. It’s almost a direct translation of that algorithm in C code. It has to perform bitwise operations on bitmap in order to determine whether a particular pixel pertains to letter or to background.

Then we have `uint16_t write_phraseLCD(char *phrase, uint16_t phrase_length, uint16_t x, uint16_t y, uint16_t letter_color, uint16_t background_color)` which uses the previous function to write a complete phrase.

It should be noted that that both function return the last column of the symbol or phrase. This is necessary because it allows us to store it somewhere and later easily clear the written symbol or phrase from the screen by painting the area with background color. That way we can clear from the screen exactly that symbol or phrase without rewriting nearby areas that didn’t contain the phrase or the symbol and thus not wasting the precious time.

Then we have a function `uint16_t write_numberLCD(char* number, uint16_t number_length, uint16_t x, uint16_t y, uint16_t number_color, uint16_t background_color)` which displays numbers on screen. The difference from previous function is that the strings such as “000902” would be written as “902”. The first 0’s would be cleared from it. We have to use this specific function to write numbers because our itoa function inserts 0’s into the string on positions that precede the first
digit. If the number is 0, its string would look like “000000” and it would be displayed as “0”.

Finally, we have `void paint_imageLCD(uint16_t *image, uint16_t x, uint16_t y)` which displays an image in internal format on screen. It’s used to display icons and other elements of interface. The internal format of images is described in section dedicated to `images.h`. The function is very straightforward and similar to the function that paints rectangles of a single color. The only difference is that now the color of any particular pixel is taken from the image.

touch.c, touch.h, registers_listTouch.h

Those files constitute a driver for STMPE610 touch controller. This controller is connected to the main chip via I2C protocol, which is a de-facto standard protocol used to communicate in both directions with slow peripheral devices. Its advantage is that it only needs two communication lines/pins, one for a clock which will determine the speed of transmission, and another for actual transmission (so it uses one line which can transmit in both directions).

The disadvantages are that it is very slow (the maximum speed is 400 KHz and in our prototype the speed is 300 KHz) and it also seems to be not very reliable in some cases. However, it’s enough to transmit the touch data without any significant delay.

In our implementation we actively use the hardware I2C driver inside STM32F407ZG which simplifies greatly the whole process of working with that protocol. STMPE610 is connected to I2C1 peripheral for I2C.

Here I will describe our particular implementation of protocol for our particular system. I’ll omit the general details of I2C protocol because they go out of the scope of this document and there are plenty of complete specifications regarding it.

To start the transaction, we have `uint8_t start_touch_module_transaction(uint8_t transaction_type)` This function precedes any transmission of data in any direction. Since the transmission can be in both direction, we need to set whether it’s the transmission from the main chip to STMPE610 (Transmitter) or from STMPE610 to the main chip (Receiver).

First, we must wait until the hardware isn’t busy anymore. If it’s busy for too long, we leave the function with an error and thus we avoid hanging up the whole system.

```c
int timeout = 1000;
while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY)) {
    --timeout;
    if (timeout <= 0) return 1;
}
```

After that the main chip generates the start of transaction and sets itself as the master.

```c
I2C_GenerateSTART(I2C1, ENABLE);

while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT)) {
    --timeout;
}```
if (timeout <= 0) return 2;
}

After that it sends the information about the upcoming transaction to STMPE610.

I2C_Send7bitAddress(I2C1, STMPE610_ADDRESS + transaction_type, transaction_type);

The transaction type affects the address of the device in this particular implementation which may be confusing.

After sending it, the main chip should wait for the appropriate response from STMPE610.

if (transaction_type == I2C_Direction_Transmitter) {
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)) {
        --timeout;
        if (timeout <= 0) return 3;
    }
} else {
    if (transaction_type == I2C_Direction_Receiver) {
        while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED)) {
            --timeout;
            if (timeout <= 0) return 4;
        }
    }
}
return 0;

After the expected response is received, the function returns 0 indicating that the transaction’s start was successful.

Next we have function to actually transfer data between the main chip and STMPE610.

uint8_t send_data_to_touch_module(uint8_t data) {
    int timeout = 1000;
    I2C_SendData(I2C1, data);
    while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED)) {
        --timeout;
        if (timeout <= 0) return 1;
    }
    return 0;
}

This function is used to send data to touch module. It’s used mostly to configure it and request some data from it. We rely on StdPeriphLib to actually send the data via I2C protocol and then we simply wait until a confirmation of receiving it has arrived. If transmission failed, 1 is returned and 0 is returned otherwise.

uint8_t receive_data_from_touch_module(uint8_t* data) {
    int timeout = 1000;
    I2C_AcknowledgeConfig(I2C1, ENABLE);
while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)) {
    --timeout;
    if (timeout <= 0) return 1;
}
*data = I2C_ReceiveData(I2C1);
return 0;
}

This function is used to receive more than one byte of data from STMPE610. First, we indicate that the main chip must acknowledge the transmission and then we wait for it to occur. After leaving the loop the data is already fetched by internal I2C buffer inside the main chip and we simply need to copy its content into some variable in the memory.

Because I2C protocol requires explicit STOP condition to stop data transmission we can’t use this function to transfer just 1 byte. Instead we should use another function which also must be used to transfer the last byte of data, if there was more than 1.

uint8_t receive_1byte_from_touch_module(uint8_t* data) {
    int timeout = 1000;
    I2C_AcknowledgeConfig(I2C1, DISABLE);
    I2C_GenerateSTOP(I2C1, ENABLE);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED)) {
        --timeout;
        if (timeout <= 0) return 1;
    }
*data = I2C_ReceiveData(I2C1);
return 0;
}

The only difference with the previous function is that now after receiving the byte of data the I2C peripheral inside the main chip will also generate stop and, thus, will terminate the transmission.

We can also generate stop explicitly by using the following define:

#define stop_touch_module_transaction() I2C_GenerateSTOP(I2C1, ENABLE)

All those data transmission functions are essential for implementing other more sophisticated ones that actually get touch data from the module or configure it. The first of those advanced function is void Touch_Init().

This function simply checks that we can communicate with STMPE610. It requests the ID_VER datum from it and receives one byte of data, which contains it. In some occasions the device doesn’t respond after powering the board and it that case the device should be unplugged and then powered again. The cause if this is currently unknown. Fortunately, it doesn’t happen often.

After verifying that STMPE610 responds, we can configure it using void configure_touch_module() function. The detailed description of all the parameters that it configures is located in code itself. Here I’ll comment on just a couple of parameters that affect accuracy.
Basically, we want to get the maximum accuracy from the touch controller itself, so that the main chip wouldn’t have to perform additional requests or calculations to guarantee it. The touch accuracy is extremely important for good user experience and even though our touch display is of resistive type, generally we should try our best to do what we can with it.

First of all, we use 10 bits resolution for ADC. This is because after trying both 10 bits and 12 bits resolution, I have found that there is no difference between them for this particular screen, probably because it’s small enough. As such, there is no need to have 12 bits resolution.

We also set the maximum sample time to 124 in ADC clock, so it takes the longest time to stabilize and produce the result. This clock is set to 1.625 MHz, so the conversion time should be about 56.4 us. Looking at the table of conversion times we can see that for a human there is absolutely no difference between conversion times, because all of them are very small.

![Figure 12. Table of conversion times for an ADC in STMPE610](image)

As such we can have a perceivable gain in accuracy without affecting at all the user experience.

Then we also set the number of samples to 8 in order to get a single value for X or Y, thus improving the accuracy more. Finally we also set touch detect delay and settling time both to 5 ms. Increasing those values will negatively affect the user experience, because the system will be unresponsive to slight touches. Setting the touch detect delay to significantly smaller value will produce many false touches.

The complete time required to get a touch value can be calculated using the following formula: Touch Detect Delay*2 + Settling Time * 2 + (number of samples * sampling time) * 2 [54]. In our case it equals to 5 ms * 2 + 5 ms * 2 + (8 * 0.0564 ms) * 2 ) = 10.9 ms. This is a very small amount of time to get the data, even though some time must be added to it in order to transfer and process it, but generally the screen is very responsive to touch. The touch data is also accurate, even though some additional effort must be done in main chip in order to achieve that.

The function that is necessary for that is `void reset_touch_fifo()`. This function deletes all the values inside the FIFO-buffer of STMPE610 that stores touch values until they are transferred into the main chip. The reason for that consists in that in our system...
we always react as fast as possible to retrieving touch data and we only need one value at a time. Sometimes an additional touch value gets into FIFO buffer and resides there until it is retrieved when other touch data have been processed and placed right after it. So instead of new value the system would retrieve old and useless value.

By deleting all values in FIFO periodically we manage to completely get rid of this phenomenon. The function to actually detect touch data is `uint8_t detect_touch()`. The system periodically uses it to check whether new touch data is present. This happens quite often, so even though we don’t use interrupts to process touch data, there is no perceptible delay.

To retrieve the data `void get_touch_data(uint16_t* x, uint16_t* y)` is used. It’s called right after the presence of new touch data is detected. In this function we are forced to use bitwise operations, because STMPE610 transfers 12 bits of information for each value. In total it transfers 3 bytes, first contains 8 most significant bits of X, second contains 4 least significant bits of X and 4 most significant bits of Y, and the third byte contains the rest of bits of Y.

However the data we are getting with this function doesn’t correspond with the actual coordinates measured in pixels. It is raw data which needs to be converted into the real coordinates on screen. To do this we use `uint8_t convert_touch_data(uint16_t* x, uint16_t* y)`. This is a very straightforward function.

```c
uint8_t convert_touch_data(uint16_t* x, uint16_t* y) {
    float final_x;
    float final_y;
    if ((*x >= LEFT_BORDER_X) && (*x < RIGHT_BORDER_X)
        && (*y >= UPPER_BORDER_Y) && (*y < BOTTOM_BORDER_Y)) {
        *x /= 10;
        *y /= 10;
        *x -= LEFT_BORDER_X/10;
        *y -= UPPER_BORDER_Y/10;
        final_x = (float)(*x)*FLOAT_PER_PIXEL_X;
        final_y = (float)(*y)*FLOAT_PER_PIXEL_Y;
        *x = (uint16_t)final_x;
        *y = (uint16_t)final_y;
        return 1;
    }
    else return 0;
}
```

`LEFT_BORDER_X, RIGTHBORDER_X, UPPER_BORDER_Y` and `BOTTOM_BORDER_Y` are raw values that correspond to borders of LCD screen. Raw touch values that are outside those borders can actually be obtained, but they are not valid, because they don’t correspond to any coordinate on screen.

We use `float` because STM32F407ZG has hardware FPU (float point unit) which enables fast processing of `float` operations. The support of hardware FPU is also enabled in compiler settings; otherwise it would emulate FPU by using regular ALU (arithmetic-logic unit), which would be significantly slower.
This function returns 1 if conversion was finished correctly and 0 if raw values don’t correspond to any point on screen.

The rest of functions are used to test different configurations of STMPE610 or to provide an example of touch detection and touch screen interface. They are not used in the main firmware, so I’ll omit their description.

**StdPeriphLib**

StdPeriphLib is a library by ST Electronics that provides a simplified API to access STM32F4xx’s peripherals in an easy way. It’s provided by free and the license also allows to modify it and to use it in commercial products as long as ST Electronics copyright is in place, so essentially it’s an Open Source license.

This library is a must-use in order to avoid writing directly in registers to configure main chip’s hardware and peripherals. Using it we can avoid caring about many details of internal hardware implementation of the chip, and this library is especially valuable for beginners. It’s also a great way to avoid many bugs and to make code look more like it’s written for a PC rather than a microcontroller.

The only cost that comes with it is that it creates a certain overhead, but since we use a very fast microcontroller that runs at 168 MHz its effect is mostly negligible. It should be noted though that in spite of the fact it that comes from ST Electronics itself, it’s not entirely bug-free and there are some descriptions of obscure bugs that exist in this library. However most of them only occur in very specific circumstances and they can be fixed. In any case they are very unlikely to affect most of software and they certainly don’t affect our project.

Not all of the files that are present in the library are used by our project, but I think that’s it better to keep them in project tree, so that any developer can explore directly the content of the library and start using it right away. It doesn’t increase the size of firmware, because unused functions aren’t compiled.

**SD card driver**

Communication with SD card is not a trivial process. In fact, the simplified version of specification of SD-card physical layer has 202 pages [55]. There are different versions of SD-card standard (such as cards complying with different speed standards and different capacity standards) which have different initialization sequences and which may have subtle differences in communication protocol. There is MMC-card standard which is pin-compatible with SD-cards, but it’s a slightly different standard as well and it requires different initialization sequence and protocol.

As such, it’s quite difficult to implement a SD-card driver and, especially, to determine that it’s correct and works with all kinds of SD-cards that are present on market. So in order to simplify the development, I decided to use an off-the-shelf implementation, which is, in this case, an implementation from ST Electronics. This implementation was made specifically for their evaluation boards, but it required very few changes in order to be usable in our prototype. This code is also licensed under the same license as StdPeriphLib, so it can be also used in our final product.
I should note that this driver only allows us to access blocks that we specify, but it
doesn’t provide us with any scheme to actually store data in any sorted way. We could
make certain rules to store some parts of data in some fixed blocks, but this system
wouldn’t be compatible with other systems and it would be cumbersome to use it to
store any relatively big data (that requires more than a few megabytes, to say so).

So in order to actually access the data usefully, for example, to be able to play music
transferred to SD-card from a computer, we have to make use of an actual file system.

**File system layer**

We can call the file system layer a driver of special kind. Instead of working with bare
hardware like other drivers, file system driver works on and provides access to an
abstract structure that is file system. File system is used to organize information into a
hierarchical system which allows accessing, modifying, creating, deleting and searching
for data.

File system layer provides to the OS and other processes an API to operate with data
and then translates these abstract accesses to data into accesses to physical blocks (that
are numbered and physically allocated on storage device). After we are left with
accesses to actual physical blocks, the hardware-level driver enters into the picture and
finishes the accesses to physical blocks, depending on what kind of storage devices
those blocks are located on.

File system is mostly independent from storage device, which allows using the same file
system on different types of storage devices, however there also must be certain
considerations regarding the underlying hardware. Sometimes those considerations are
more or less important, but generally file system must be aware at least until a certain
extent of which type of hardware it has under itself, for example, whether it allows a
random block access at all.

All of that makes file system a very complex component of any system. Even a simple
file system driver can be another complete project just by the level of complexity. So,
the only practical solution to provide a file system layer to our prototype is to use
someone’s implementation. In this case, I decided to use tremendously popular Chan’s
FatFS driver [56].

This driver provides access to FAT12/FAT16/FAT32 file systems. It’s a generic driver
written in C and it can be used on almost any system ranging from tiny 8-bit
microcontrollers to complete PCs. It is very lightweight and it allows a wide range of
possible operations to perform on a file system volume, including formatting one. It
allows various possible configurations which provide different levels of complexity and
features. It’s also distributed under its own open license, so this software can be
considered Open Source.

Chan’s FS driver is actively supported and new versions are made available periodically.
Our prototype uses the version R0.10a which is the newest version at the moment of
writing this page. During the development of prototype no bugs were noticed with this
version of it.
From the architectural point of view the component has a much defined place in our and other systems.

![FatFS layer in the system](image)

**Figure 13. The position of FatFS layer in the system** [56]

FatFS layer should be located between the software or OS and a hardware low level driver. FatFS provides high level API to application or OS which allows opening files, browsing folders, etc. Low level driver provides to FatFS module functions to read and write physical blocks. FatFS will “build” an abstract file system which will tie together physical blocks and structures like folders and files.

FatFS should be extremely easy to integrate into any system, unless there are some strong differences between that system and most of other systems. In order to integrate FatFS into a system, the code of that system mush include ff.h in order to be aware of FatFS’s API.

FatFS also contains a couple of functions to access the physical blocks that must be completed with calls to the low level driver in order for the module to properly function. Usually it’s very easy to complete those functions if the low level driver is already implemented in the system.

So, there are only three files that must be modified in order to use the module. The first one is diskio.c which is used to glue FatFS with low level driver. The second one is ffconf.h, which is used to configure FatFS, like enabling or disabling some features which in turn makes it more or less heavy in terms of compiled size and RAM consumption.

The more detailed view of the system, with complete list of files that constitute FatFS, is shown on the scheme on next page.
In the case of our prototype, the low level disk I/O layer is SD-card driver from ST Electronics. We also don’t ever write to SD-card, we only read from it, so FatFS is compiled in read-only mode that doesn’t include any write function and we also don’t have to complete those functions in diskio.h.

Other crucial part of documentation is what kind of filenames we can support. FatFS allows to support complete Unicode names, however since this is an early prototype and we lack any non-ASCII symbols, we only support the simplest mode – ASCII-only symbols and 8.3 names (8 characters for name, a point and an extension that contains 3 characters).

I should note that we specifically use FatFS because it’s the implementation that is robust, popular and easily obtainable, but old FAT12/FAT16/FAT32 systems aren’t the ones currently recommended for SD-cards, especially the newer and high-capacity ones. But for now we can’t use another file system and FAT32 is more than enough for our practical purpose. We’ll talk more about this in section 6.4.

**apps.c, apps.h**

Those files contain huge functions that act as applications in our current system. The code contains a detailed explanation of details, so there is no need to describe in detail each line here, but I’ll talk about the general algorithm that underlies those functions.

```c
void system_message(uint8_t number) is a simple function that will display a window with a system message, which is good to use to display some messages about errors. The calling function must pass a number which identifies messages to be shown. The message itself is hardcoded into the function itself. It’s not the best way to store those messages, but they don’t occupy a lot of space at the current time. If there are many messages we can put them into firmware by using the const keyword.
```

Next we have `uint8_t txtViewer()` which is a very big and complex function that is used to open and display text files. The user can browse the text file by using arrow buttons, like she would be able to do on a PC. The difficulty here consists in that it is
difficult to know how much information we’ll be able to show on screen and we don’t want to even try to store the text file in our RAM. So instead of that we should read the data from SD-card when we need it.

The solution is, perhaps, not very elegant, but it works. First of all, we define a content that is shown on a single screen a single page (in the same sense as page in a book). Then what we do before even showing anything is that we read the whole file calculating the number of pages it has and also storing the initial position of each page in an array. The main difficulty consists in getting right the content as it is shown on screen without actually showing it.

After this process is done, the function goes to the first page and it shows its content. The content is being read, once again, from SD-card. Next, when the user goes to the next and then to previous pages, the function accesses the address of page inside the pages array, l_seeks the file to that position, reads the content and shows it on page.

In spite of being a bit clumsy, this implementation works fairly well and it’s pretty fast. The initial delay is virtually unnoticeable for the size of file that is currently allowed to be opened (100 pages as maximum). In theory the amount of pages and, thus, the file size is only limited to RAM size. Holding addresses for 100 pages requires 3200 bytes of RAM.

Another interesting thing to comment on is the state machine. The state machine is based on a couple of parameters such as the current page and whether there is a new order. First of all, we show the first page, set new order variable to false and then we simply wait for an input from user. Then, when there is an input from user, we modify the variable that determines the current state, which is the current page and set new order variable to true. So next iteration the state machine will automatically process the new order according to newly set state-defining variables, and then it will set the new order variable to false again. The whole process then will be repeated.

The function is also capable of withstanding unexpected events such as SD-card extractions, in which case it will return a value different from 0.
A possible improvement to this function would be the possibility to jump to a page specified by user instead of going only forward and back. This should be trivial to do, since we can set the current page to any arbitrary page between the first and last pages. The function will get the correct address to get data from.

Very similar algorithm could be used to display more advanced files, so the device could potentially act as book reader, even though it’s not initially intended to have this functionality and there are certain limitations regarding the complexity of text file formats that it can open.

Next complex function, this time it’s the most complex of all, is `uint8_t file_manager()`. As the name suggests, it’s our file manager. Currently it’s capable of displaying the content of a directory, browsing inside another directory and selecting a file to open. So it would be more correct to actually call it file system viewer, but it can be expanded into a complete file manager capable of all functions of a typical file
manager from PC-based operating systems (directory creation, file creation, file replication, directory and file deletion, etc.). The function has detailed comments in its own code, so I will not go into details, just like in previous case, however I’ll describe in general how it works.

Like in the previous function, the solution is, perhaps, not the most elegant, but it works for our purpose. Just like before, we don’t want to use much RAM, so we don’t maintain there a complete list of files inside the current directory. Instead every time the content of directory is shown on screen, even if partially, the function reads the whole content of the directory. The screen can show 10 files or directories, so if there is a directory that contains more than 10 files or directories and user just wants to see the complete list, every time the content of screen is refreshed the whole list of files is fetched from the SD-card.

This doesn’t create problems even with directories that contain relatively big amount of files (around 100), but it can potentially create problems with directories that contain thousands of files. The delay is unnoticeable for normal-sized directories though.

The function maintains a list of files and directories that are currently shown on screen in order to identify them if user chooses to open one of them. It also maintains the number of first file or directory shown on screen in order to enable scrolling. The number is stored inside an array called “cursors” which also allows the system to remember the exact position where the list was in upper directories. A side consequence of that it the limitation on how deep we can go inside the directory tree. Currently the depth is limited to 50 directories, but this number can be safely expanded. The function also stores names of traversed directories to easily recover them if user traverses the directory tree back later.

The state machine without error treatment is shown on figure 16 on next page.
This way of displaying content has a very important disadvantage aside of the obvious inefficiency. It consists in that contents of directory appear unsorted alphabetically. Instead they appear in the same order the file system’s inodes are allocated. This makes it impossible to easily find files or even listen to them in order.

As such it would be absolutely necessary to change this mechanism later. One possibility would be allocating a list of files in external RAM and sorting it out alphabetically. This way there also wouldn’t be any need to read the directory every time that the content of screen is refreshed.

Those are all the functions currently presented in apps.c and apps.h. More functions can be added which may provide additional functionality, for example, clock and calendar, temperature measurement, a calculator, etc.
Those files contain the central and most important application of our prototype – the actual music player. Those files also represent both the application and the driver for the reasons explained above. They are also based on the open-source implementation [58] of music player from VLSI, but they were modified to fit our purpose.

Since we communicate with VS1053 using SPI protocol, first we are going to need a couple of base functions to ensure correct data transmission. SPI is a de-facto standard protocol which allows transmission of data between master and slave devices. It can be very fast and there are two lines to transfer bits – one from master to slave and another from slave to master. This makes it more reliable and easier to understand than aforementioned I2C protocol. It also has a consequence which consists in that any transmission is bidirectional. In other words, every time a byte is sent, a byte is also received, even though it doesn’t necessarily have any meaning, but the peripheral of master chip captures it anyway.

There are many documents which describe in details SPI-protocol, so I’m not going to describe it formally and instead I’ll focus on our implementation for our main chip and VS1053. We have 4 functions in total which are related specifically to SPI.

First we have a very basic function uint8_t SPI2_Send(uint8_t data). This function simultaneously sends and receives data, so its name may confuse a bit. We always use it when we send and when we receive data. It’s a very generic function that can be used for any device connected via SPI.

void WriteSci(u_int8 addr, u_int16 data) is used to transfer 2 bytes of data into a specified memory address of VS1053. Similarly, u_int16 ReadSci(u_int8 addr) is used to receive 2 bytes of data from a specified memory address. int WriteSdi(const u_int8 *data, u_int8 bytes) is used to transfer big amounts of data into VS1053. It’s used to transfer audio files.

After those functions we have several functions provided by VLSI: uint32_t ReadVS10xxMem32Counter(uint16_t addr), uint32_t ReadVS10xxMem32(uint16_t addr), uint16_t ReadVS10xxMem(uint16_t addr), void WriteVS10xxMem(uint16_t addr, uint16_t data), void WriteVS10xxMem32(uint16_t addr, uint32_t data), static uint16_t LinToDB(unsigned short n) and void LoadPlugin(const uint16_t *d, uint16_t len). Those functions can be considered an API provided by the vendor, so they can be used in any commercial or non-commercial product based on VLSI’s products.

Finally, we arrive to main functions of the player. First of them is int VSTestInitHardware(void) which simply performs hardware reset on VS1053 and it executes once when the whole system is started.

Right after it we have to execute int VSTestInitSoftware(void). This function performs essential configuration on VS1053 without which it won’t work properly. Initially it will perform a basic sanity check to determine that VS1053 is connected and works properly. Then it sets VS1053 frequency to 43.008 MHz and consequentially it also sets higher frequency to our SPI peripheral. The transmission frequency must be at
least 4 times lower than VS1053 frequency, so in theory it can be 10.752 MHz, however on practice the highest frequency below it that our main chip supports is 10.5 MHz. At this frequency no problems were found with the transmission of data.

`int VSTestHandleFile(char *fileName, int record)` is the first function that manages playback of files. First of all, it opens the initial file that was passed as a parameter and starts its playback using a function that specifically manages playback. However it also acts like a mini-file manager in a sense that when playback is finished, this function will automatically go to the next playable file in the directory and will open and play it.

In fact, user also can interrupt the playback of current file and immediately go to the next one, previous one, first one and last one. The function works very similar to our file manager, because it doesn’t maintain any list of directory’s content and instead it just reads the content from scratch until it finds what it looks for. It doesn’t read the whole content in most cases though.

Finally we arrive to the heart of player - `uint8_t VS1053PlayFile(FIL* audio_file)`. This function actually plays the audio file and checks for the input from user. It’s a little bit more complicated that any of previously described functions, but it’s still not very complicated. First of all, we have two state machines – one state machine for VS1053 and another for our software player that controls VS1053. Those state machines are different, but we reuse some states from VS1053 state machine. We also have to follow it in order to control VS1053, but we can use it as it fits most our purpose within our own state machine. So I’ll focus on our software state machine.

After setting up initial values to variables, we enter a loop that will be exited only if playback of current file is finished or the user wants to interrupt it prematurely. Then, if it could correctly read the file and the player is not paused, we immediately send 512 bytes of file to VS1053. The playback of file immediately starts after that and now we enter real-time processing.

If instead of wanting playback, user decides to cancel it (in order to skip to next file, for example), we have to follow some steps to cancel the playback properly. We must send an explicit request to VS1053 to cancel the playback and then check if VS1053 cancelled it. If it did, only then we formally end it on our side too and leave the main loop. We won’t send any data though regardless of whether VS1053 already cancelled playback or not (it will stop playback in case of not receiving data, but it will wait for new to arrive unless the playback is explicitly cancelled).

If the playback goes as normal, right after sending the piece of audio file we will also collect so called endFillByte which we need to properly cancel the playback (outside of main loop) or if there is any problem with the file.

After that, we check for an input from user. There are many buttons in the interface and we are in real-time now, however this form of checking for touch data and processing it is fast enough not to cause any interruption in playback of files. If there is a valid user input, we perform whatever changes that were requested, including sending new data to VS1053 if volume level was changed.
After that, we need to refresh the playback time. To do that we get the amount of seconds that have passed from VS1053 which actually counts this value for us. However we can’t use it directly, because the amount of time necessary to process it takes too long for real-time. Instead we maintain our own counters which we increase if there is a difference between the current value from VS1053 and the previous value. To display the time we also a special itoa function that doesn’t contain any loop inside it.

After that the process repeats itself until we arrived to the end of audio file or user requested to cancel playback. In both cases we send endFillByte to VS1053 until its decoder can finish decoding properly. Then we cancel the playback formally by sending the cancel request to VS1053 if the file ended and if user didn’t cancel the playback (in that case, we would cancel it inside the loop as I described earlier).

After all that the function finally ends and returns a value to upper function int VSTestHandleFile(char *fileName, int record) which will react in some way depending on received value.

Generally, the player application seems to be very robust and efficient enough for our purpose. It reacts smoothly to changing the audio file and starts the new playback immediately. The volume level changes are smooth and don’t cause any interruption.

It has, however, a very significant problem – the lack of random access support. The reason for that is the fact that we rely only on VS1053 to perform audio file decode and we can’t get correct time to memory address conversion from it, because it simply decodes and immediately plays what it gets, but it has no idea about the overall file content, neither it ever was its purpose. Moreover, we can’t easily perform that conversion using our main chip, because most of audio file formats don’t have that information explicitly either and many of them also have an optional variable bitrate. So in order to actually implement random access we would have to at least partially decode them by software, which is unpractical for the purpose of this project, but would be essential for future development. We’ll discuss this problem in detail in section 6.2.

Instead of random access functionality we have fast forward function, but its speed is very limited by the SPI transmission speed. Moreover, it doesn’t replace random access in any way and it also works well only in case of strongly compressed file formats such as MP3. In case of formats such as FLAC and WAV the speed is very slow, because the compression ratio is smaller or there is no compression, so the files are big and take a very long time to transfer.

This problem was unexpected, because I didn’t foresee it on earlier stages of development and became aware of it when I actually started developing the player application. I wouldn’t say that it significantly affected the project, but it definitely exposed an unexpected difficulty which must be solved in future stages of development.

The state machine for the player application is shown graphically on next page.
As we can see on scheme, the user input is checked only once each loop. But since the loop is executed for a very small part of file that is sent to VS1053, this creates the perception of instant reaction to touch on screen. In fact, the overall processing is so fast, that systems delivers virtually immediate feedback. The process that seems to consume most of time during a single loop is actually getting the touch data from STMPE610 via slow I2C protocol.

Other than the absence of random access within a file the player application has all the functionality typically expected from an application like this. In some ways it can be said that it’s arguably superior, because of smooth volume change and overall very nice response time from prototype.
4.6 System in action

In previous section we took a look at how system works internally. In this section we’ll see how it actually looks from user’s point of view and we’ll also see the prototype user interface.

When user turns the system on, the first screen she’ll see is this:

![SD-card absent message](image)

*Figure 18. SD-card absent message*

At this point the system is completely initialized; it’s just that the SD-card is absent. Now it’s time to insert it. After that, the system will mount it shortly and user will see the following screen (on next page).
The system directly shows root directory and list of files. By clicking on file, the system will attempt to open it. If it can’t open it, the following message will be displayed.

By clicking on directory, the system will open that directory and show its content. Naturally, the user can also scroll the list up and down. It can be seen on figure 21 on next page.
Figure 21. Browsing a directory

The system is capable of opening plain text files. An opened file looks like this:

Figure 22. Text application view

The user can close the text file and go back to file manager by clicking on the arrow icon in the top left corner of the screen. Naturally, she can also scroll the text file up and down the same way as she can do so with file list. This can be seen on figure 23 on next page.
Sometimes a text file is too big and it exceeds our established maximum of 100 pages. In that case the following message is displayed:

![Figure 24. System message regarding too big text file](Image)

To listen to music the user is supposed to navigate to the folder or folder where music is located. She can place and group music files in folders as she finds convenient. For now, this is the only way to sort music. A music folder is shown on figure 25 on next page.
As we can see, the music files aren’t really sorted in any order for reasons we explained in previous section. This folder also contains files like “01.mp3” and “02.mp3” which are in the end of directory, even though they are first songs on album.

By clicking on an audio file, the user will launch a music player:

As we can see, the interface is rather typical. The buttons are big and they are very easy to click. There is enough room to add more information, such as track title, track length, etc., things that we don’t have yet in this early version of software.
The sound can be muted without interrupting the reproduction and the volume can also be changed while still muted.

![Muted sound icon in music player](image)

**Figure 27. Muted sound icon in music player**

Clicking on fast-forward button will automatically mute the sound and the player will attempt to fast-forward the audio file with the maximum possible speed. The time will be accurately shown, so the user may stop this mode when necessary.

Sadly, this is not in any way a replacement for random access functionality, especially in big file formats such as FLAC and WAV where the difference in speed is insignificant comparing with normal speed of reproduction.

Overall, the system’s design is very simple, efficient and straightforward. Any user that has minimal experience with file managers already knows how to use the system’s interface to browse folders and open files. We also don’t waste time on intro screens and directly jump to apps to improve system’s response time and provide direct access to functionality.

The system’s state machine from user’s point of view is shown on figure 28 on next page.
With this brief tour on how our system actually looks and works from user’s point of view we conclude the software section. In next section we’ll discuss the development process.
5. Development process

5.1 Problems encountered during development

The development didn’t go as it was initially planned, however the deadlines were met. The reason for that is the fact that I underestimated the difficulty of developing components such as display controller initialization sequence. On the other hand, I’ve overestimated the difficulty of developing the actual functionality such as audio reproduction and I’ve correctly estimated time needed to fix bugs.

The biggest feat that I had to overcome during the development was either poor documentation or complete lack of thereof. The most difficult part was implementing the SSD1963 controller initialization sequence in spite of application note and datasheet. Both of those documents aren’t clear enough on what has to be done and how it has to be done. Moreover, there are no code examples at all. As the result, I spent around 2 weeks just on that little function that would be done in an hour if there was a working code example.

Similar problems were encountered with STMPE610 and trying to connect SD-card driver with our system, but those weren’t as difficult. In contrast, there wasn’t any problem at all with VS1053 from VLSI, because they provide very good documentation and great code example that worked instantly on my board. I must express, once again, my admiration towards VLSI and highly recommend their products.

VLSI also provides a great example of how things should be done to other venders of chips. This problem with getting things to work isn’t just some issue that affects only beginner developers. I’ve seen stories of experienced developers that spent the same amount of time doing this. I personally think that this situation is clearly unacceptable and there is a great need of creating a database of code examples and complete drivers for slave chips and other devices.

There are already some efforts in this directions by, for example, CoCox CoIDE developers that provide a library of drivers and code examples for various components. Of course, I personally will try to contribute to it with code developed during this project.

Aside from this, there were surprisingly few obstacles during the development. There were a couple of bugs that were extremely hard to spot and they always were caused by a line or two of code, but those type of bugs are to be expected in any kind of systems development.

One interesting thing is that initially I planned to develop music player application and only after that any sort of additional functionality, such as plain text viewer. However, it turned out that it was better to develop text viewer first, because that way I could check the functionality and correctness of file system driver instead of potentially running into huge problems with music player later.
5.2 Project timeline

The firmware development of project comprised 17 weeks or 4 months and one week.

**Weeks 1-2** were spent on studying basics of STM32 platform and reading documentation. I also developed a basic application to control a multi-color LED on my board in order to learn how to perform I/O operations on pins.

**Weeks 3-4** were spent on trying to initialize SSD1963 display controller and by the end of week 4 it finally was done. This was a hardest part of development, as I mentioned before, and I didn’t expect that at all.

**Weeks 5-6** were spent on development of SSD1963 driver and implementation of a font based on Arial. Exhaustive testing and bug hunting were done during this time and by the end of week 6 driver and font were completed.

**Weeks 7-8** were spent on development of a touch input using STMPE610 touch controller. I had to study I2C protocol which I never used before, initialize STMPE610 and get correct entries from it.

**Weeks 9-10** were spent on debugging and getting it to reliably get the touch data. Several test applications were written in the processes, such as a simple app for drawing.

**Weeks 11-12** were spent on connecting SD-card driver and file system driver, and developing the file manager application. Initially I tried to write my own SD-card driver, but I abandoned the idea after encountering an off-the-shelf implementation from ST Electronics that also had Open Source license.

**Week 13** was spent on developing and debugging text viewer application.

**Weeks 14-15** were spent on developing music player application. It took surprisingly little time to initialize VS1053 and get it to work. By the end of week 15 the player application was working.

**Week 16-17** were spent on polishing the overall system, improving the code and enhancing the functionality of player application. By the end of week 17 the prototype firmware had the current state.

Generally the development could have been done faster if I had any previous experience with ARM and STM32, and also with some other complex embedded system. A lot of time was spent on studying protocols such as I2C and SPI that I didn’t have prior experience with. An experienced developer could probably do all of that in 8 weeks or 2 months. If libraries would be already provided and there wouldn’t be a need to write from scratch initialization sequences, the development time could be reduced to just one month.

It should be noted that development of “high level” applications such as text viewer took significantly less time than drivers development. This means that developing high-level applications is suitable for beginners, and the development of code that runs on bare metal is the real complex part of it.
5.3 Accomplished goals

Initially the project had the following goals:

Main goals

- Play .mp3 and .wav formats
- Touch screen interface which allows to browse folders on SD-card
- Usual functionality of an MP3-player, such as to change volume, pause song, etc., including randomly accessing a point in time within a song

Additional goals

- Be able to browse audio files by tags
- Be able to search audio files by typing a word and launching a search
- Be able to create playlists
- Be able to play additional audio formats

More goals (everything that goes beyond audio reproduction)

- Anything else

By the end of the project we have (almost?) all of main goals accomplished, some additional goals accomplished and we have an additional functionality unrelated to audio. Specifically, the list looks like this:

Main goals

- Play .mp3 and .wav formats
- Touch screen interface which allows to browse folders on SD-card
- Usual functionality of an MP3-player, such as to change volume, pause song, etc., including randomly accessing a point in time within a song

Additional goals

- Be able to browse audio files by tags
- Be able to search audio files by typing a word and launching a search
- Be able to create playlists
- Be able to play additional audio formats

More goals

- Text viewer application

As such, we accomplished more than half of goals overall. The reason why one of main goals wasn’t accomplished completely is because I underestimated the difficulty of randomly accessing any point in a music file. The reason is that essentially I didn’t plan to implement any sort of software decoding for audio files, but in order to implement a random access it absolutely has to be done, and it’s impossible to implement decoding (even partially) of several audio formats in a short amount of time.
that was left until the end of the project. However it can be debated how basic is the functionality of accessing a random point inside a file. Also it’s just a part of a bigger goal and not a single goal itself, and that bigger goal was mostly accomplished in everything except that one thing.

Most of additional goals required creation of some sort of database in order to store tags and perform a search on them. Since a lot of time was spent on initializing slave chips, I didn’t have enough time to touch that subject. However its difficulty may be enough for another complete project itself. Fortunately, it was simple to implement reproduction of additional file formats, because we completely rely on VS1053 to do that and it does its job well. Particularly important is the fact that the prototype can reproduce FLAC, which is the most popular lossless audio file format.

Finally, a text viewer application is potentially a very strong addition to a prototype, because it showcases the potential of having a big screen and enough memory to open relatively large files.

Overall, the prototype has all the functionality that it was aimed to have and even a bit more. The functionality which allows browsing songs by tags and performing a complex search was extremely difficult to begin with and it would imply the creation of a database driver, which could be considered a complete and separate project.

We’ll discuss the future development and what should be added to the current prototype in next section which critically analyzes it and proposes changes for the future prototypes.
6. Analysis of prototype and future improvements

6.1 The purpose of this section

As I’ve said previously, this early prototype is just a first step in a possible bigger project which may bring to market a complete product. The main goal of it is to provide a solid base for future development and to discover problems and disadvantages in the proposed architecture and design.

So, in this section we’ll discuss various aspects of prototype, such as the features it lacks and which should be implemented to convert it into a commercial product. We’ll criticize some aspects of the current architecture and we’ll also discuss what alternatives we have. Overall we should arrive to several alternative directions for future development and prototypes.
6.2 Hardware codecs and their advantages and disadvantages

To decode audio file formats our prototype uses VS1053, which is a hardware codec that is extremely easy to use and which has several advanced features. This simplicity of use was the main reason to include a hardware codec into the current design and choose this particular development board.

However, as it became clear during development, usage of hardware codecs also carries possible disadvantages and limitations. The whole concept of sending small pieces of file which are decoded in real-time makes it difficult to implement random point access within a file. This happens because unless the main chip is aware of details of implementation of audio format, it can’t convert a definite point in time into an offset from file’s start. This is especially complicated when the file has variable bitrate, which is common in many types of audio formats, such as MP3.

So, essentially implementing random access functionality would require some sort of software decode. It doesn’t have to be a complete decode, but the extent to which it has to be implemented is currently unknown. In case of some formats, such as .wma, the file format explicitly retains timing information, so in its case it would be easy to implement that functionality. In case of other formats it would be more difficult, such as .mp3, because they tend to have different versions with subtle differences between them and their specification seems to be very flexible, so each file can be constructed differently depending on encoder.

This detail is disappointing, because it means that the best applications for hardware decoding would be streaming or very simple MP3-players that don’t have random access functionality. In case of a system with a strong CPU, such as our prototype, it may be simpler from architectural point of view to use software decode.

However, in our case I would still strongly argue in favor of continuing using VS1053 or similar hardware codec. The reason is that hardware codecs still simplify a lot the development process, so a novice developer, such as a person that develops in her free time, would be able to easily program her own player application within a week or two without ever entering into complex algorithms or format specifications. For some types of usage, such as streaming applications, hardware codecs are extremely appropriate, since no file can be loaded and analyzed before starting actual decoding.

At the same time the price isn’t an issue for us, because we didn’t plan for our device to be cheap from start. Also the hardware codec doesn’t limit us in anything. If we want, we can implement complete software decode library that simply decodes a chosen file format into WAV and then sends it into VS1053. Additional advantage of hardware decoding would consist in saved CPU time that doesn’t get wasted on decoding audio stream.

Now, after deciding to continue using a hardware codec, we should consider whether or not we should keep VS1053 or try something else. I would definitely argue that if using a hardware codec, it would be a good idea to stick to VLSI’s products, because VS1053 had already proven a certain ease of use which their products have and good documentation that comes with them. Among VLSI’s line of products they a have a direct successor to VS1053 (even though it’s still commercialized) – VS1063.
VS1063 is hardware drop-in replacement of VS1053 that has all the features of VS1053, but adds a couple of new ones. For example, it can decode FLAC by default, so it doesn't require any plugin to load, which currently occupies some part of our prototype’s memory. It also has additional functionality which would correspond to an advanced MP3-player, such as an equalizer [59].

The software would need to be changed, according to VLSI’s page on the product [59], but given the simplicity that I experienced with VS1053, there shouldn’t be any problem with that. From the hardware point of view, there is nothing that needs to be changed, so we could use the same board as the current board, but with VS1063 instead of VS1053.

The disadvantage of VS1063 is its elevated price which is considerably higher than VS1053’s price for any of bulk quantities. For example, the price of 500 units of VS1053 is 3.20€ per unit [60] while for VS1063 it’s 6.00€ per unit for the same quantity [61]. However, the price of complete system would be higher only by 2.80€ which is insignificant quantity for a product that should cost, in any case, several dozens of euros. Also most of development boards seem to use VS1053 instead of VS1063, so if we have more advanced version we can also advertise this as an advantage of our product.

In conclusion, I would say that for the future version of prototype it would be preferable to use VS1063 instead of VS1053. If its quality is not inferior to VS1053 (and it shouldn’t be), I would suggest it for use in a commercial version of a product as well.
6.3 File formats

Being able to play many different music formats is essential for any good MP3-player. Our current prototype is able to play the following formats: WAV, FLAC, WMA, MP3, MP1, MP2 and OGG. VS1053 also supports AAC, but we currently can’t properly reproduce it, because most of files which contain that type of audio have metadata at the end of the file instead of having them at initial point. Since currently our firmware simply sends the file to VS1053 regardless of its format, if metadata are at the end, VS1053 can’t identify format and, thus, it’s can’t decode the data.

This can be solved by finding metadata of AAC stream and sending them before starting to send the actual encoded stream. This can be solved relatively easy by studying specifications of AAC and M4V container file format, and then implementing this functionality in firmware.

VS1063 can decode same formats and the same functionality for AAC must be implemented for it. So just by relaying on hardware decoding our product would already decode most of popular file formats. However it would be wrong to stop there.

One file format that VS1053/VS1063 can’t decode is Monkey’s Audio (APE) file format. It’s a lossless file format that has achieved certain popularity, even though it’s not licensed under Open Source license. Still, it would be very interesting to try to add the possibility of playing audio files in that format.

It’s unlikely to be possible to achieve by developing a plugin for VS1053/VS1063, because APE requires a lot computing power to decode its stream, but it may be possible to achieve this using our main chip’s ARM Cortex-M4 processor running at 168 MHz. In that case a player application would decode APE stream into an uncompressed WAV stream that would be transferred to VS1053/VS1063 for reproduction (it also would be necessary to add metadata in order for VS1053/VS1063 to identify the stream correctly).

Sadly, there is strong evidence that probably APE can’t really be decoded in real time with our CPU with all, but less strict encoder settings, as we can see by performance results from RockBox implementation [62]. However, a proof of viability or impossibility of reproducing existing flavors of APE would be a different and complete project itself. Perhaps, here we could rely on the additions from the community of hackers, but it also would be a good idea to perform, at least, some basic research in that direction in future versions of software.

In any case, the decoding of audio formats supported by VS1053/VS1063 is already enough and it surpasses the number of formats supported by most players. Additional support for more obscure formats and/or APE would be a welcome addition, of course, and this is where the open nature of our project and interest from community of hackers would be very relevant.
6.4 File system layer limitations

Currently our prototype supports only the following file systems: FAT12, FAT16 and FAT32. This is enough to support SDSC cards and SDHC cards; however the specification for newest SDXC standard mandates the use Microsoft’s exFAT file system, incompatible with our file system layer. So in order to make our device compatible with this type of cards (which offer highest speed and capacity currently) we would have to purchase the license from Microsoft. The exFAT module also would have to be closed, because Microsoft doesn’t allow Open Source implementations of it.

This wouldn’t go against our Open Source approach, because BSD license is compatible with inclusion of proprietary and closed components in software. Generally this only would be done to ensure the best experience for customers who don’t care much about using proprietary standards and to provide maximum functionality to the device. A compiled and closed library could be provided in the source tree and with some external API in order to be usable by other applications.

The cost of licensing from Microsoft is unknown, so while this improvement is desirable, it’s very difficult to estimate how viable it would be and what implications this would have regarding the price of the device. Apparently the flat fee for licensing it is 300,000$ [63] in which case that would pretty much rule out exFAT support for our device, but it’s unclear whether this fee applies for our type of device.

Aside of exFAT there are plenty of Open Source file systems which could be used for a possible internal storage drive. Some examples include UFS, Ext2, Ext3, Ext4, btrfs and more... Those file systems can also be used for SD-cards, but it’s not recommended generally, since using any other file system than FAT12/FAT16/FAT32/exFAT can potentially reduce card’s lifespan and performance (even though it doesn’t necessarily has to happen). It’s unclear how difficult it would be to implement a driver for those systems. In fact, for some of them the overhead would most likely be too big to handle with our CPU and low amount of RAM. But some, probably more basic, file systems could be implemented and used for internal storage device.

Generally, our current file system layer is actually enough for both SD-card storage and internal storage. However, since FAT32 is an old file system which has limited usage, it would be desirable to have support for more advanced file systems such as Ext2. This, probably, would be a very complex project, but, again, we can hope that a help from community would simplify a bit the task.

Finally, in case of SDXC cards, the support of exFAT is essential; however it’s difficult to estimate its cost and complexity. To approach the status of high-end device for some consumers it would be required to have support of SDXC cards, however for our main audience – hackers and makers community, it’s not necessary and support of FAT32 is more than enough. A support for an advanced Open Source file system such as Ext2 would be very desirable for them.
6.5 Possible internal storage

Most of high-end MP3-players on market have their own internal store or hard-drive, typically in form of SSD, eMMC (very similar to a SD-card, but internal and non-extractable) or even a small HDD. Since our player is aimed at both general audience and makers’ community, the situation regarding internal storage is a bit complicated.

On one hand, most people in makers’ community don’t really need internal non-extractable storage, because they prefer to place their own storage device instead, which is most suitable for their project. On the other hand, most of people outside that community don’t view it very well when a device lacks internal storage and all the data they want to have access to must be placed on some sort of extractable device. Most people prefer to connect MP3-players to their PCs and transfer music to internal storage. Also, microSD-cards are often significantly less reliable than internal storage, so by having no other means of storing data on device, we can run into the risk of subjecting users to higher probability of losing data.

The disadvantage of having internal storage is that it will significantly increase the price of device while not providing a clear advantage to people from makers’ community. So here we stay at crossroads where both ways have advantages and disadvantages.

In my opinion, it would be a good idea to try to appeal to a wider audience, when possible, so I would argue for inclusion of internal storage in future version of prototype. The most appropriate form of storage would be eMMC which would have, roughly, the same cost as adding to the system an actual MMC card. Another alternative would be actually using a real MMC or SD-card inside an internal slot, accessible only by opening the device.

In addition we would still have an external microSD-card, so the user would have the possibility to add more memory if she needs to.

The inclusion of internal memory adds many additional possibilities. First of all, we could store many interface elements, such as icons, pictures and animations inside that internal storage. Today the prices for storage are very low and even small amount of it such as 1 gigabyte or even less would be more than enough to store an advanced interface inside.

Next advantage would consist in that we can store there a database for user’s data, such as the complete list of songs, allowing browsing by tags. We can also store there structures such as directory tree for a file system improving the performance and adding new capabilities to file manager. We can even store several databases for different storage devices, such as different SD-cards, thus improving user experience.

Finally, we can use that storage to store applications and in this case it would be even easier to add new applications or to update existing ones. As we can see, the inclusion of internal storage seems to still provide more advantages than disadvantages even though it would increase the costs.

If we decide on having internal storage, then we should also think about the amount of it. High-end players normally have an amount of 16GB or more, going up to 160GB in
some cases [64]. So, 16GB is the amount of memory we should have at least, if we intend the storage to be used for music. In that case, the cost of inclusion of internal storage drive would be approximately 20-60€ depending on size (60€ for 64GB). If we intend to use internal storage only for database, interface and some user data, then it can be as low as 1GB or less, and the cost would be very small (5€ or less) [65]. In case of having an HDD, the cost would probably be significantly higher, but the size of internal memory would be bigger too.

It would be nice to offer different options regarding internal storage. In order to support better flexibility and reduce costs, we could try to use interchangeable modules which contain different internal storage devices, so those slots could be mounted before sending the purchased product, instead of making different boards for different options. Producing a separate board for module still requires some additional costs though. This would also allow an option to go without internal storage altogether. In that case the absence of internal storage module would give access to additional free pins.

At the present state, we can only use FAT32 for internal storage, but having other file system such as Ext2 would be preferable. It should be noted that our current file system layer supports having multiple volumes at the same time (up to 10 mounted volumes currently).
6.6 Additional connectivity

One disadvantage of our current board (Mikromedia+ for STM32) consists in that it doesn’t have some interesting connectivity modules, which can be very interesting for a commercial product like MP3-player. On the other hand we have an Ethernet connectivity module which is not very interesting for our purpose.

There are a couple of connectivity options which would be very appropriate for a consumer product which we’ll discuss here and now. First of those modules is a radio module. FM and AM radio are common radio standards used everywhere in the world. They also are not very expensive and adding them into a future prototype would be a very nice and obvious addition. As an example, the cost of SI4703-C19-GM FM radio tuner is approximately 3$ (unit price for a bulk of 1000 units) [66].

Another obvious addition would be Bluetooth module which would allow connectivity with other devices such as smartphones, other MP3-players, notebooks, etc. It would allow completely different scenarios for usage of device, so it would be very interesting both from consumer point of view and from point of view of makers’ community. Sadly, those modules tend to be pretty expensive. As an example, WT41-E-AI5, which is a high-quality Bluetooth module, costs around 26$ (unit price for bulk of 1000 units) [67].

Finally, another addition would be a WiFi module, which would allow an easy access to services like internet radio and other internet-based services. As in case of Bluetooth module, those modules tend to be expensive. As an example, MRF24WB0MA/RM WiFi module by Microchip costs 20$ per (unit price for bulk of 100 units) [68]. Adding a companion chip for TCP/IP stack implementation, such as MCW1001A-I/SS, would add additional 3.37$ (unit price, no bulk specified) [69].

Adding those connectivity options would bring some very advanced and, perhaps, unique functionality to our device, but would also add additional costs. The partial solution would consist, as in case of internal storage, in offering options to users and solving the assembling problem by usage of modules, which would increase manufacturing costs, however.

This internal module structure would also allow after-purchase upgrade options to the device and this could potentially become a strong feature of our device, even though it would significantly increase the difficulty of development.
6.7 The absence of database driver

A significant problem of our current prototype is that there is no easy way to add database support needed for advanced functionality such as sorting files by tags (in “Artists”, “Albums”, “Songs” fashion) and searching files by typing a keyword. In order not to use inefficient methods such as brute-force search, we need to use something like a database or even a complete database. Obviously, a generic driver would allow not only the usage for sorting songs, but could potentially find all sorts of various uses in different applications.

Database drivers for embedded applications do exist [70], but the obstacles to creating one are expected to be high. In fact a project like this could be an entire thesis project on its own.

We need a database driver, because today’s consumers are extremely used to this type of functionality. Most of them would naturally expect this sort of functionality from any high-end MP3-player, because it’s so common. So, in this regard we would definitely have to follow the trend, unless we would position our product strictly as low-cost.

It should be noted, that we don’t need a complete database driver, because we would only operate with small amounts of data and we wouldn’t have to use advanced functionality such as retrieving or writing large quantities of data, using triggers and so on. A basic functionality of storing small strings of text data in several tables and be able to perform queries of medium-level complexity would be more than enough for this and many other uses.
6.8 User interface shouldn’t be limited to touch screen only

While our current display is pretty big and very comfortable to use (there are no touch errors and it’s very responsive), it should be noted that in many scenarios of use it’s not the best way to interact with the device. Battery time also becomes an issue when only touch screen is used and it’s constantly turned on in order to display buttons.

So it seems clear that we should aim at having physical buttons for interaction with display. It’s not complex to add buttons to our current prototype, which could serve different purposes. Most notably they could serve to control the functionality of music player app, most notable to pause, stop and skip songs, and also to change the volume. They could also serve to perform scrolling in other apps or adjusting FM radio frequency (when this functionality is implemented).

Another form of interaction can be accelerometer, which is currently not used. It can be used to modify the orientation of screen depending on how the player is oriented in physical space. It also can be used as a controller in some games. Other advanced forms of interaction can be used such as by shaking the device a save file action can be performed (this was implemented in Mintpad MP3-player; sadly I can’t provide a link to it because the company which manufactured it went out of business some years ago).

Finally, we can use a multicolor LED to inform user about current state of device, such as low battery or as notification for something else. Finally, we can even use it to visualize music patterns (this is used in Sony Xperia SP smartphone). This LED can also be used as a lantern, by the way, but we should watch out, because it may be extremely dangerous to eyes.

Overall, the conclusion is that physical buttons must be added into design. Their exact placement on an external casing would be a job for an industrial designer and goes out of scope of this document. Additional form of interaction with the device would be welcomed as well.
6.9 Sorting algorithms must be added to file manager

Our existing file manager doesn’t sort files currently; instead it just displays them in order in which it finds their inodes. This works OK in a prototype, but this wouldn’t be acceptable in a commercial product. I would even say that adding the file sorting capabilities to file manager would be our next number one task.

Fortunately the methods for sorting files are known and were already implemented on several microcontrollers, even significantly simpler ones than ours. Most likely this would involve creating our own implementation of a sorting algorithm, such as quicksort and using it to sort files within a directory, using several parameters to do so. We also would have to use RAM in order to store the current list of files at least for the current directory. It also could be a wise decision to look into nearby directories and sort the files there beforehand.

Our microcontroller has few amount of RAM to spend it to maintain a list of files, in my opinion, but we can use the external RAM which, fortunately, doesn’t seem to cost much. The price for MB85RS256A, which is 256KB FRAM module accessed via SPI protocol, ranges from 0.1 to 10$ dollars depending on bulk size [71]. This module could be used to store exclusively the lists of files used in file manager (and of course, any additional data that would be big enough in size).
6.10 Lack of Unicode support

One of most worrying problem in our current prototype is the fact that it can only display ASCII-characters. This is clearly not enough by current standards and users expect their device to be able to recognize all kind of characters from different languages such as Russian, Chinese, Japanese, Greek, etc. As such it would be necessary to implement a complete font with Unicode characters.

This task is a very difficult one just by the amount of effort it would require, however it can be accomplished by a dedicated team of people. The process can also be accelerated very significantly by using some existing bitmaps and converting them into our internal format or by generating bitmaps from PC fonts using some already existing tools that make the whole process almost effortless comparing to manual font implementation.

In any case, Unicode support seems to be just a question of time, but it’s not in any way a technically complicated problem. Also a complete Unicode font or fonts would require a lot of Flash memory to store it. Fortunately we can add a parallel or serial flash module which could provide more than enough memory to store fonts for a relatively small amount of money. For example, SST38VF6404 parallel Flash memory chip from Microchip costs 3.77€ (unit price for bulk of 1000 or more units) [72] and provides 64Mbit of memory, which should be more than enough for a complete Unicode fonts. If we use an internal storage, the storage size is not a problem at all.
6.11 Long file names must be supported

For reasons of simplicity we limited file names in our current file system layer to old 8.3 convention. However, for future versions of prototype and, obviously, a complete product, it would be absolutely necessary to provide support for modern long file names and Unicode support.

It should be noted that FatFS already supports that and it simply must be enabled in its settings. Unicode font must also be provided in order to be able to use this feature.
6.12 We would eventually have to make a transition to an OS

While we chose to develop one-piece software instead of trying to go with an OS for this early prototype, during the development it became clear that this firmware architecture tends to get really complex with more features and eventually it would become unsustainable in long run.

This is especially reflected in the following limitation – we can’t go to file manager without interrupting playback of an audio file. This could be solved, of course, with some amount of tweaking and implementing some workaround, but if we want a general way to process tasks on background (and we want it in long run), it would be absolutely necessary to switch to an OS-based approach.

Here we could either use someone’s OS as long as it is provided for free and licensed under a compatible Open Source license, or we could try to develop our own OS which would take into account our specific architecture and purpose. It wouldn’t be as difficult as developing a kernel for general purpose PC architecture, because microcontrollers are significantly simpler, but it still would be a difficult task. However, that task would be achievable by a small team of dedicated developers and it would definitely be within reach of a small- or medium- size company.

The component we developed for current prototype would also be reused whenever possible and in most cases we could directly insert the code into the new system as tasks or parts of kernel.
6.13 We should definitely make use of modules available on current board

Many of modules on our current board are still unused. Nevertheless many of them are extremely interesting and could provide additional functionalities to our product.

Those modules are accelerometer, analog temperature sensor, RF transceiver, buzzer and real-time clock.

Accelerometer can be used for some gaming applications and as an additional way to interact with the device, as I mentioned in section 6.8. Temperature sensor can be used to do just that – measure temperature. Buzzer can be used to provide additional feedback to user. RF transceiver can be used to establish connection with all sorts of devices. Finally, real-time clock can provide time, calendar and organizer functionality.

As we can see, those modules should be relatively easy to implement and at the same time together they could provide an interesting addition to overall functionality of the device.
6.14 Overall look at functionality

In previous section we established features that our current prototype lacks and the direction we should move to in order to advance in the development. By adding those features we would essentially convert our little prototype into a fully functional commercial product from the technical point of view.

However, there are also some additional points we didn’t touch that lie outside the technical domain. Those are user interface and overall design.

The current user interface was just developed to be simple and intuitive. It was impossible to dedicate much time to its development; as such it was intentionally very basic (it was only composed of two colors – black and white, among other things). However, interface should be a strong point of any commercially successful product. Developing an interface, however, is not the job of an engineer, but of a designer, so she would have to be involved in the next stage of development. At an earliest stage of prototype design may not matter much, but it would be essential to, at least, start thinking about it in the future version.

Another question is the external design of casing. A good design of casing is also essential for successful products. Once again, it’s not a job of an engineer outside of specifying (or, on the contrary, trying to fit the device into) some characteristics such as size and weight of the device.

Finally, going into next stages of development it would be necessary to develop our own board instead of using current board from Mikroelektronika. This new board should contain any new components that we want to add into finished product and also it should contain slots for modules, if we finally decide to go with that approach. Several developers or even small teams of developers would work on different aspects of software and testing them more or less concurrently. Visual interface and design of casing would also be performed by designer or designers’ team. Marketers would help to spread knowledge about the upcoming product and broaden the audience of potential clients.

Our current project would serve as the solid foundation for that further development while Mikromedia+ board would serve as reference design. Regardless of whether the future development will be ever done, the current firmware will be released and may help to other products and it may introduce STM32 platform to other developers.
7. Considering the eventual commercialization of product
7.1 Introduction

In this chapter we will discuss the final stage where any prototype may arrive as the result of development. That is, the commercial product which is sold to clients in shops or via internet.

For a company that wants to stay in business long enough it doesn’t end with launching a product. After that it will have, at least, some sort of support and possible future iterations depending on how market is responding to the concept. In the world of electronic consumer products, especially high-end ones, the development often doesn’t end with the launch of product, but instead continues steadily during the whole lifecycle and is often offered in form of updates. This is typical in world of smartphones, for example, where companies try to offer support and regular updates to the OS and applications bundled with the phone.

This is especially important in case of our product, which is geared toward community of makers and hackers, who value a lot the involvement of company with development and continuous support. We’ll look at some real cases of products and companies that used Open Source concept to illustrate our point.

Finally, aside of the purely commercial, developmental or technical aspects, there are also aspects of social, environmental and economical impact. Smaller companies which can make use of Open Source concept and our prototype can often do things right or at least better than some bigger companies, and can make it their strong point. So we’ll talk about those concepts as well.

Generally, this section will not be about technical aspects of our product, but will be more about economical, social and environmental impact of it and also about our general approach and philosophy.
7.2 Business model: Open Source and Open Hardware

As I mentioned in chapter 1, our product is essentially a mix between a typical consumer product and a product aimed at Open Source/Open Hardware enthusiasts, who want to tinker with hardware, write their own firmware and share it without any restrictions. There are several companies and products emerged recently that are aimed at enthusiasts and are relatively successful, up to the point where we can say that this approach is viable. At the same time our product is conceptually different from them all, because it’s aimed at the general audience as well.

For general audience it should have the following characteristics:

- it should work out of box
- it should be attractive visually and be portable out of box
- doesn’t have to require any assembly work or programming from user to be fully functional

For community of enthusiasts it should have the following characteristics:

- it should be easy to disassemble and modify
- disassembling and some level of modification should be allowed by design and shouldn’t void the warranty
- it should have a free bootloader to re-upload firmware in any moment
- code should be licensed under some Open Source license (BSD 2-clause currently)
- all the development tools have to be available for free, at least (not necessary licensed under Open Source license, even though that would be preferable)
- board schematics, all PCB design files in original state and extensive documentation should be provided for free and licensed under Creative Commons (except documentation from chip vendors which should be just accessible for free)

In other words, what company sells when it is selling this product is essentially hardware (and the effort that was put into designing it and making it function with firmware). User has full control over hardware by having access to firmware and having ability in any way she wants. Some additions to hardware are also permitted by exposing pins on boards or having special slots for upgrade modules, but further modification of the board itself would void the warranty. There may be some parts for which the modification is not allowed at all for safety reasons (such as battery and the firmware of its controller).

When the product is finished and it has made its entry on market, the company also helps the community to develop it. Particularly, it would maintain a library of firmwares developed by third-parties on its website and engage in active contact with the community via forums, mailing lists and chats. It would also maintain a github-like service to distribute code of both its own firmware and firmware from third-parties. Of course, third-parties can also have their own web pages and independent service, and they would be promoted too as long as they have good quality.
This can result in several different firmwares for the same product, but that’s OK and in fact it would be marketed as a strong feature. This would benefit typical consumers as well, because they would be able to download any alternative firmware (pre-complied, not necessarily in form of source code), try it and switch back if they don’t like it. It could also create new possible usages for the product that we couldn’t foresee and potentially open a new market.

Company’s own firmware would be continuously developed as well and it would accept patches and propositions from the community. In fact, anyone can make a fork of it and introduce any changes without company’s approval, but the company can also include any positive changes in its own main branch of development.

The usual consumers wouldn’t be excluded from that process. Even though they can’t or don’t want to develop themselves, they would be welcomed to make suggestions, propose new features and provide feedback. They also would receive updates of main firmware automatically via company’s update application for PCs, even though they would be able to not install it or refuse updates, of course. They would also be encouraged to try different firmwares and it would be emphasized that the ability to easily switch firmware is one of product’s main features.

Finally, the company would also try to encourage developers from the community by participating in hackathons, offering free hardware and even monetary rewards. For example, if the company doesn’t have enough resources, other than monetary, to develop a database driver, it can simply find a developer or team of developers in community that can work on it and earn money once it’s accomplished. It can even employ them temporarily to work full-time on a certain project, communicating via internet with the company and receiving monthly salary. Some Open Source projects, such as HaikuOS, and some big companies successfully used this model of employment.

I believe that the approach exposed in this section can benefit both the company and its consumers. It gives real freedom to third-party developers and just your regular consumers, who can find the best way to use the product for them. It also makes it easier for company to incorporate changes and introduce improvements, because the whole community works on them. This way the bugs are fixed and new features are introduced more quickly than otherwise. It also can make product more attractive to audiences that it wasn’t even targeted to.

Generally, this approach was inspired by David L. Jones and all credit for it should go to him [73].
7.3 The general description of this particular product

In this section we’ll summarize what was established in previous chapters regarding the characteristics of the final product. Even though we established a description previously in chapter 1, during the developing of the early prototype the description became clearer and now we can aim at a final product with more specific characteristics.

The product is marketed as an advanced MP3-player which is an Open Hardware product with firmware licensed under Open Source license (BSD 2-clause). The product has look and feel of a commercial MP3-player that works out of box. It can play all the most popular audio formats (.MP3, .WAV, .FLAC, .OGG, .AAC, .WMA) and it has high-quality audio tract, which is provided together with hardware decoding by VS1063 or VS1053 (VS1053 is used in the current prototype).

Its main interface provided by big (3.5 inches screen or more; 4.3 inches in current prototype) touch screen, but it also has physical buttons. It has microSD-card slot which is capable of accepting cards up to 64GB formatted with FAT32 (and hopefully, with other file systems such as exFAT). Hopefully, it also has internal memory formatted with Ext2 or similar file system, which is used to store interface and databases (in which case its amount is small, <4GB) or also to store audio files (in which case it should be big, >=16GB). Maybe, the user can choose the amount of memory she wants to have and upgrade it later using modules.

Also, the player has several connectivity options, such as WiFi and Bluetooth and, once again, hopefully the user may choose whether she wants to have them or not. Finally, it has plenty of smaller modules and sensors, such as accelerometer, temperature sensor, FM-radio tuner…

When it comes to technical characteristics, we have:

- >=168 MHz CPU with FPU, ARM Cortex-M4
- Main chip produced by ST Electronics, STM32F4 family
- 192 KB of RAM
- 1 MB of Flash memory
- 256 KB of external RAM
- 1 MB of external Flash memory
- Screen controller for display
- Touch controller for touch input
- Hardware audio codec, such as VS1063

This product doesn’t aim to be cheap, because in the community of makers it’s OK to pay a bit more given the Open Hardware nature of the device. Also since we aim to provide a high-quality sound, it should be OK to most of potential consumers as well. However, it shouldn’t be too expensive either. I would establish that the maximum price point should be 100$ (without counting the price of internal storage of huge size) or, rather, 99.99$. It should be lower if possible, but it wouldn’t make sense to go beyond 50$ price either, because that would be very stretching and unrealistic.

Regarding different versions of the device, there can be a version based on different microcontroller, for example, a version based on PIC. However that would be done only
if there is a demand in the community for it, because we can’t just produce a module with different main chip for the board (it’s probably possible, but it’s very difficult technically and probably it’s cheaper to just make another version of board).

The company would provide a nice documentation that contains schematics, a user manual and also a disc with software such as firmware updater and a bootloader which would allow replacing the firmware. The disc would also contain files with PCB’s design and source code of firmware, even though it would be advised to check for new versions of code before trying to add or modify anything.

In the following section we’ll discuss some real products similar to the one presented here from different companies and their fate.
7.4 A non-exhaustive review of Open Source/Open Hardware products and companies on market

Here I’ll present some real cases of products, successful or not, which try or tried to follow Open Source or Open Hardware model. I’m not pretending to review all of them, but hopefully this review will be representative enough and will provide a nice precedent to our own concept.

**Efika MX computers from Genesi**

![Efika MX Smartbook](image_url1)

*Figure 29. Efika MX Smartbook [74]*

![Efika MX Smarttop](image_url2)

*Figure 30. Efika MX Smarttop [75]*
Efika MX computers, presented on pictures on previous page, were consumer-oriented products introduced on market in 2009 by Genesi, a company that doesn’t focus much on consumer market, but rather provides large scale services. As such, those products didn’t receive big promotion – they were mostly showed on consumer shows and expos, but they weren’t easily obtainable or heavily promoted anywhere.

They used i.MX51 platform from Freescale, featuring an ARM processor running at 800 MHz [76]. The system came with Ubuntu linux preinstalled and it provided a nice level of functionality, however with time the hardware proved to be clearly insufficient to execute applications and display web pages with enough speed (by 2011 standards and beyond). The support for Ubuntu version of operating system was eventually dropped and users had to switch to Debian in order to continue to receive updates (note: at the time of writing this document, a new version of Ubuntu distribution suddenly came out, promising updates up to April 2017).

Genesi created forums for both users and developers, and even though various alternative distributions were provided there, they didn’t achieve the desired level of functionality expected from a typical consumer product (except the Ubuntu distribution and, later, the Debian one). Even when the basic functionality was provided, users generally were limited to software that came with the distribution, because most of programs from repositories didn’t work well enough or didn’t work at all.

However, as one of the users of Efika MX Smartbook I must say that it was a good device in spite of all that and the concept of smartbook is very interesting one. I believe that the future versions of the product could have been improved significantly and provide the level of functionality comparable to tablets. The small community of users and developers around it was generally content with the device and wanted to eventually see the next version.

But Genesi decided not to pursue that route at least at the moment and new versions weren’t released or announced. This is probably because the user base wasn’t big enough and the current market of consumer products, inundated with tablets and hybrid devices, is extremely competitive. However those products weren’t a failure either and both users and the company had a nice experience with them. It should be noted that Genesi gladly buys back any device, working and broken ones, from their users to re-use them.

**Mikroelektronika**

Mikroelektronika is a company that sells development boards and add-on shields for several microcontroller platforms. It is not an Open Source of Open Hardware company, but it has several interesting traits that make it closer to those concepts than many other companies.

First of all, they are not exclusively oriented towards industry, but they are also geared towards hobbyists and students. This is reflected in that they try to provide good documentation and sometimes directly help beginner developers on forums. The schematics they provide with their product are very nice example of good documentation.
Second, they are also popular in makers’ community and they try to actively promote themselves in that community.

The company seems to be doing well, reporting around 1 millions of euros as net income in 2012 [77].

**AdaFruit**

AdaFruit is a company which specializes on selling electronic components and gadgets for enthusiasts of DIY community. In many ways it is similar to Mikroelektronika, because it sells boards and shields, but it’s geared specifically towards hobbyists and makers. It also sells some third-party products, designed by other companies or individuals not working for AdaFruit.

AdaFruit is a great example of successful [78] Open hardware company. Their products, however, are not oriented at typical consumers, just like in case of Mikroelektronika.

**Arduino**

Arduino is the name of the company, the name of several development boards, and also the name of a platform that can contains many highly-compatible products based on various microcontrollers. Arduino is, probably, the most known example of Open Hardware product and Open Hardware based company.

![Arduino Uno, as shown on official Arduino website](image)

*Figure 31. Arduino Uno, as shown on official Arduino website [79]*

It was made specifically for students and people just interested in electronics, not necessarily engineers. As the result of producing high-quality products, offering them at
reasonable prices and providing continuous support and development, the company achieved a huge success and even what some people call a small revolution in the market. Now Arduino is number 1 platform to initialize all kinds of people in electronics and it’s even used in the industry as prototyping platform.

It also benefited other people, because it allowed them to create their own products based on Arduino or being an add-on for it, and sell them. One example of this is Gameduino/Gameduino 2 console, which was funded by Kickstarter campaign and produced in small volume [80].

However, Arduino-based products and Arduino itself are not meant for users that don’t want to develop anything and just want to get a typical consumer-oriented product.

**BeagleBoard**

![BeagleBone Black](image)

*Figure 32. BeagleBone Black as seen on its official page [81]*

BeagleBoard is another famous OpenHardware project. It’s essentially a development board based on an ARM-processor capable of executing linux, thus it’s a platform very
similar to PC and appropriate to be used for consumer-oriented tasks. However, it comes without casing and it’s marketed only to developers and hobbyists.

It’s pretty popular among Open Source enthusiasts, seemingly more among computer scientists and IT-engineers than among electronic enthusiasts.

An interesting aspect of this product is that it’s produced by Texas Instruments, a big company that makes high-end SoCs, microcontrollers and all kinds of other electronic devices.

**Raspberry Pi**

Raspberry Pi is another tremendously popular product, which is currently as famous as Arduino. Raspberry Pi is a complete PC on a development board like BeagleBoard, based on Broadcom’s ARM SoC, having a Linux distribution as its operating system. Its initial intention was to provide an inexpensive, small PC to teach computer science in schools.

However, since its appearance on market in 2012 it was extensively used by Open Source enthusiasts in all sorts of ways. Laptops, embedded car PCs and PDAs were created based on Raspberry Pi. A very big and dedicated community also appeared around the device.

![Figure 33. Raspberry Pi turned into a PDA](image)

It main attractive point is its low price (just 45$ currently), being Open Source and the fact that it can be used by anyone, even people who don’t plan to develop something or tinker much with it. For example, many people use it as a substitute for SmartTV and as a movie player box. As such, it can be effectively considered a product aimed at both Open Source community and at regular consumers. A version of the device inside a case
is available and it can be just plugged into a PC screen or TV and be used as a regular PC.

Raspberry Pi is produced by a charity, so it’s can’t be considered a typical business success, but it is indeed one of the most successful and popular Open Hardware projects in recent years.

**Conclusion**

We just glanced upon some of the most popular Open Hardware or Open Source products, projects and companies. It looks like those products indeed discovered a new, growing market which is built upon principles of common sharing, building a community and cooperation between the vendor, developers and users. Of course, their success is nowhere near the success of smartphones and tablets, but if we take into account that most of those products are produced by either small companies or companies that weren’t oriented at consumer market, their success is bigger than what most people would expect from them.

And this is just the start of it – the Open Hardware market is still in its infancy and it’s possible that it will experience significant growth in the following years. So our initial idea regarding the possibility of creating a product heavily based on Open Source and Open Hardware philosophy, but also targeting regular consumers, seems to be a viable idea.
7.5 Cost of the device

In this section we’ll roughly present the cost of the device. By no means is this exact price estimation, instead I’ll just present the cost of components and try to estimate the costs range for the rest.

Assuming that we’ll want to produce a relatively large bulk of units (1000), the costs would be following:

- Let’s assume that we’d end up using the most expensive microcontroller from STM32F4 line – STM32F417IGH6. Its cost per unit is 8.25€ [83].
- SST38VF6404 parallel Flash, 64Mbit, 3.77€ per unit [72].
- MB85RS256A, 256KB FRAM, exact price unavailable, let’s assume 5€ [71].
- VS1063, 4.80€ per unit [61].
- SSD1963 screen controller (same as in prototype), 3.32$ per unit [84].
- STMPE610 touch screen controller (same as in prototype), 1.03$ [90].
- AT043B35-15I-10 screen (same as in prototype), exact price unavailable, let’s assume 5€ [85].
- SI4703-C19-GM FM radio tuner, 3$ [66].
- ADXL345BCCZ accelerometer (same as in prototype), 3.04$ [86].
- MCP73832 battery charge controller (same as in prototype), 0.43$ [87].
- MCP9700A temperature sensor (same as in prototype), 0.23$ [89].
- Board assembly, case and connectors/smaller parts, let’s assume 10-20€.
- Battery, which cost we can assume at 8€ (for 1420 mAh capacity) [93].

Optional:
- nRF24L01+ RF transceiver (same as in prototype), 1.78$ [88].
- WT41-E-A15, Bluetooth module, 26$ [67].
- MRF24WB0MA/RM WiFi module + MCW1001A-I/SS, 20$ + 3.37$ [68] [69].

As we can see, the cost of essential components sums up 62.91€, which is the cost that we expected. The real cost would likely be smaller, because we probably wouldn’t use as much Flash memory or that particular, most expensive of the entire line, microcontroller. The cost of board assembly per unit, case and smaller parts is very difficult to compute without actually designing it entirely and then contacting board manufacturers, but I would assume that 20€ is the most expensive price that we would get per unit.

Additional connectivity components add significant cost, as we determined previously. Assuming we would add them all, it would increase the costs up to 100.26€.

Internal memory would, most likely, add the following costs to the total cost (based on microSD cards prices):

8GB – 9€
16GB – 13€
32GB – 20€
64GB – 40€
It should be noted that those costs doesn’t include legal costs, costs of development and costs of producing documentation and performing some marketing (all those costs together tend to outweigh the costs of everything else). However, comparing with the prices of the real mid- and high-end MP3 players available on market (for example, iPod nano 16GB costs 165€ [91]) it looks that our final product would be in same price range as those players or slightly more expensive, but it would offer a significant advantage over them – it would be Open Hardware and Open Source product.

It would require a lot of money to bring it to the market (at least 62,910€ for first 1000 units just on components and, overall, it probably would ascend to 150,000-200,000€ for costs of development, taxes and legal stuff). However, this project seems to be viable and costs don’t seem too exaggerated.
7.6 Social, environmental and economical impact of product

When we are talking about Open Hardware products, it’s impossible not to talk about the impact that this whole philosophy may have on our daily lives and the way we do things. Since the start of the whole Open Source movement, it significantly changed the way people saw software and how it could be developed. By its whole nature Open Source helps to spread knowledge about programming and computer science in general, because everyone can see the source code and learn how it works.

Moreover, anybody can adapt the source code to their particular needs, thus making users independent from software developers. Educational avenues, such as schools and universities benefited the most from this. Industry and business benefited from it as well, because any institution could use Open Source programs for them thus reducing costs of development and bringing to their clients cheaper products. Licenses such as GPL make sure that any change done to the code would be released back to the whole Open Source community, so everyone could benefit from it. BSD-type licenses don’t require even that, offering independency from Open Source movement to companies if they wish it.

Open Hardware in many ways can be viewed as the extension of Open Source movement. It brings those concepts to the whole package, making it possible to study in detail any device licensed under Open Hardware and reproduce it if necessary. It naturally benefits schools and universities, but it also brings more freedom to end users – typical everyday consumers, because it makes it possible for them to modify their devices as they wish with help of other people, such as third-party developers. It makes people to really own their devices without any restrictions from patents or obscure terms and conditions.

It also naturally helps people to start their own small businesses and make money, thus helping local economy. If someone has an idea, for example, some sort of significant improvement for an already existing product, such as making a synthesizer out of our MP3-player, she can simply take our design, make a compatible module which performs a new task or a complete derivative product, and manufacture it. It would probably benefit both the company that produces the original device and the new maker.

The knowledge shared as the result of Open Hardware approach would also help standardize best practices and best solutions, making the whole industry more cost-effective and bringing the best quality to end users for less money. This process is, perhaps, similar to practices established in science, where sharing of knowledge is common and it is the only way to validate the current knowledge and further advance it.

When it comes to social aspect, companies based on Open Hardware approach tend to be more responsible in many ways than other companies. They often tend to rely less on outsourcing and promote local assembly even if it costs more. For example, Arduino boards are assembled in Italy. Other aspects include salary to developers and workers on assembly lines. It’s clear that any company should never underpay their employees. Open hardware companies may be less inclined to do this than others, because usually they are more transparent and their popularity often depends a lot on their reputation in community of makers. On return the community is prepared to pay higher price for their products even if competition offers similar products for smaller price.
Finally, there is the question of environmental impact. Making of any electronic component is harmful for ecology and there is, probably, no way of eliminating it, but we can try our best to reduce its impact. One way of doing so is controlling the fate of devices once they have finished their life cycle. The goal is to try to prevent broken or simply old devices ending in a trash bin and getting into junk yards where they’ll be burned, releasing many toxic substances into atmosphere.

The way I think this can be achieved is a buy-back program. This program consists in that any user can sell back to company her broken or old (or simply not anymore useful) device for small monetary reward or discount for newer devices. The company will receive the returned device and then re-use it for testing and development, or simply recycle it.

An addition improvement to that would consist in trying to prolong the life of device. First of all, the Open Source community is, generally, a notorious one in that they tend to support old hardware for a long, long time. So any features presented in newer devices would be brought to old ones as long as it’s possible technically. Second, if the device is broken, there would a possibility to actually repair it by user herself. The company would provide a component (for example, a replacement screen) that needs to be replaced for small price and detailed instructions on repairing process. While this would still void the warranty, potentially user can end up with a repaired device that will serve for long time. Obviously, if someone wants to offer repairing services, the company would only encourage this (but at this point I don’t think that it would be possible for the company itself to try to offer those services because of complex logistics of that process).

I believe that an Open Hardware company, by definition, would be most likely to have a positive social and economical impact. Hopefully, the environmental impact of it would also be minimized, because more and more people in the community are becoming conscious about this, so they would naturally try to reduce the impact if they are both inside and outside the company.
7.7 Possible derivatives of the product and alternative versions

If the initial product is successful, several products that are very alike in the spirit can emerge as additional products to be offered by the same or different company.

**Alternative versions based on different platforms**

In section 3.2 several alternatives to current platform where present. We chose STM32 as the most promising and flexible platform, but there are many alternatives, many of them with a significant number of fans. One of such alternatives is PIC architecture from Microchip and Atmel’s AVR. So if there is a demand from community, an alternative version of our product can be done based on those platforms, while maintaining all other components the same.

A particularly interesting platform for such an alternative version is VS1005 from VLSI because it would, probably, also offer a price reduction of the product while maintaining the same characteristics regarding audio reproduction.

![VS1005](image)

**Figure 34. VS1005 internal structure, which offers very high level of integration**

**Open Hardware scientific/engineering calculator**

Programmable calculators are a still widely used electronic device by many engineers and scientists. Being open hardware and open source could provide very high level of customization for different fields. At the same time the computing power of modern
microcontrollers such as STM34F407ZG is more than enough to perform complex computations.

The calculator would feature a complete keyboard and energy-efficient, non-touch display, such as SHARP Memory LCD display, for graphical content.

Figure 35. SHARP Memory LCD display, as seen on AdaFruit website [92]

Open Hardware gaming console

This would be a significantly more difficult and costly project, but it would be similar in many ways to an MP3-player from the architectural point of view. Gameduino gaming console proved that even computationally weaker microcontroller can execute good-looking, old-school games as long as the display controller is also a GPU and frees the main chip from most of graphical processing. Combining this with STM32 can produce even more interesting results, very similar or the essentially the same in experience to old gaming consoles, such as SNES or Sega Mega Drive. A gamepad, naturally, has to be added to the design and, preferably, a screen with higher resolution. A screen can be detachable and also a plugin to connect the console to TV would be welcomed.

Open Hardware e-book

This would be even more complicated project, because we have to necessarily use more advanced SoC’s than microcontrollers for it and use an advanced OS such as linux. Also it would be difficult to find an appropriate e-ink screen. However, this product can have a lot of potential for a very wide array of customers, including libraries and schools.
8. Conclusion

During this project it was shown that an Open Hardware product, such as an MP3-player, is viable and may enter the consumer market at competitive price, similar to prices of products currently available on market. An initial version of firmware was developed, which is capable of performing base functions of a commercial MP3-player. It was shown as well that usage of hardware accelerators such as screen controller and hardware audio codec simplifies significantly the development process without increasing dramatically the cost of end device.

Several examples of successful Open Hardware or Open Source products were provided and a specific case for Open hardware MP3-player was made to show how it can fit on current market and what kind of people it can attract. Also a couple of ideas for other Open Hardware products were provided to be explored in the future.

It should be noted, however, that this project produced just an early prototype and further development is necessary before it can result in a real, finished product. So the current state of prototype was analyzed as well, including its limitations, and the list of features that should be implemented in the future was created.

Open Hardware movement is environmentally conscious, so a list of recommendations was developed on how to reduce the impact on environment. Also a case was made that Open Hardware products can have positive influence on economy and society in general.

I personally hope that the interest in Open Hardware will grow in the following years and that this project will, at least, help spark an interest in it inside people who read this document.
9. References

[5] - Freescale Semiconductor, i.MX 6SoloLite Power Consumption Measurement, pages 5-8
[27] - Products for C series for Connected MCUs, Texas Instruments’ web page, screenshot available here:
[35] - ST Electronics, STMPE610, S-Touch: advanced touchscreen controller with 6-bit port expander, page 4
[36] - ST Electronics, STMPE610, S-Touch: advanced touchscreen controller with 6-bit port expander, page 1
[38] - VLSI Solution Oy, VSDSP4 user manual, page 8
[40] - VLSI Solution, VS1053b Datasheet, page 8
[42] - VLSI Solution, VS1053b Datasheet, page 35
[44] - SunBond, Specification for approval 4.3 LCD Module, page 4
[45] - The number of colors appears to be different in different documents, I used the value from A-tops Electronics Company, product specifications, Module AT043B35-15I-10, page 3, because it seems to be correct
[46] - Calculated based on data presented in SunBond, Specification for approval 4.3 LCD Module, page 7
[53] - ST Electronics, STMPE610 datasheet, page 27
[54] - ST Electronics, STMPE610 datasheet, page 30
[58] - The source code doesn’t have an explicit license, but it is provided for free and an employee from VLSI confirmed that the code can be re-released under BSD 2-clause
null


[90] - http://www.findchips.com/search/stmpe610 (DigiKey) Retrieved 05/04/2014


10. List of acronyms

AAC – Advanced Audio Codec
ADC – Analog-to-digital converter
ALU – Arithmetic logic unit
AM – Amplitude Modulation
API – Application programming interface
ARM – Acorn RISC Machine or Advanced RISC Machine (it’s unclear)
ASCII – American Standard Code for Information Interchange
AVR – Alf (Egil Bogen) and Vegard (Wollan)’s RISC processor
BSD – Berkeley Software Distribution
CD – Compact Disc
CPU – Central Processing Unit or Central Processor Unit
DAC – Digital-to-analog converter
dB – Decibel
DSP – Digital signal processor
EXT – Extended file system
FAT – File allocation table
FatFS – this is the name of Chan’s file system driver
FIFO – First In, First Out
FLAC – Free Lossless Audio Codec
FM – Frequency modulation
FPU – Floating point unit
FRAM – Ferroelectric RAM
GB – Gigabyte
GHz – Gigahertz
GPL – (GNU) General Public License
GNU – GNU is Not Unix
GPIO – General Purpose Input/Output
GPU – Graphics processing unit
HDD – Hard Disk Drive
I/O – Input/Output
I2C – Inter-Integrated Circuit
IDE – Integrated development environment
IT – Information technology
KDE – K Desktop Environment
LCD – Liquid-crystal display
LED – Light-emitting diode
LGPL – (GNU) Lesser general Public License
LTE – Long Term Evolution standard
TCP/IP – Transmission Control Protocol/Internet Protocol
TIVA – A name for Texas Instruments’ microcontroller
M4A – MPEG 4 Audio
Mbit – Megabit
MHz – Megahertz
MIDI – Musical Instrument Digital Interface
MIT – Massachusetts Institute of Technology
MMC – MultiMediaCard
MP1 – MPEG-1 Audio Layer I
MP2 – MPEG-1 Audio Layer II
MP3 – MPEG-1 or MPEG-2 Audio Layer III
Ogg – simply a name, not an acronym
OS – Operating system
PC – Personal computer
PCB – Printed circuit board
PDA – Personal digital assistant
PIC – initially meant “Peripheral Interface Controller”, now means nothing
RAM – Random Access Memory
RF – Radio frequency
RGB565 – Red 5 bit, Green 6 bit, Blue 5 bit; known as High color graphics
SD – Secure Digital
SDSC – Secure Digital Standard Capacity
SDHC – Secure digital High Capacity
SDXC – Secure Digital eXtended Capacity
SNES – Super Nintendo Entertainment System
SoC – System on Chip
SPI – Serial peripheral Interface
SSD – Solid-state drive or Solid-state disk
VS1005/VS1053/VS1063 – names of VLSI’s hardware audio codecs and SoC’s
UFS – Unix File System
USB – Universal Serial Bus
WAV – Waveform Audio File Format
WiFi – a trademark name for WLAN based on IEEE 802.11 standards