jamk.fi

Open source OTP voice encryption in microcontroller

Securing electrical voice communication

Ari Nevalainen

Master's thesis
June 2019
Technology, communication and transport
Degree Programme in Cyber Security

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author	Type of publication	Date
Nevalainen, Ari	Master's thesis	June 2019
		Language of publication: English
	Number of pages 84	Permission for web publication: x

Title of publication

Open source OTP voice encryption in microcontroller

Securing electrical voice communication

Degree programme

Master's degree Programme in Cyber Security

Supervisors

Kotikoski, Sampo; Hautamäki Jari

Assigned by

Own topic

Abstract

One-time pad is a futureproof and a simple encryption method. It has been used by e.g. armies, diplomats, rulers and banks. Yet, it has not knowingly been available for everyone on a microcontroller.

The purpose of the development project was to show how a cheap microcontroller can be used to encrypt speech in real time using one-time pad. The resulting device was to be portable and open design.

Teensy 3.5 was chosen as the development platform due to its features and add-on soundcard. The hardware setup required minimal changes, after which most work was about programming.

The programming was performed in Arduino software in C language. Especially the search for a proper gateway appeared to be hard, as the original plan of using USB connection did not work as expected. Finally, serial communication was used between the boards, while they were on the same desktop.

One-time pad encryption and one-time authentication were used for speech encryption with a limited success. The limitations dealt with one-time pad key data. These limitations can be mostly overcome by software development.

Speech encryption can be achieved by using a microcontroller and a one-time pad. This protects the speech data when traveling in end-point devices and data networks. It is not simple because of the key data management; however, it should be a considered option along the others. This applies especially to a situation when information needs to be kept secret both now and forever.

Keywords/tags

encryption, microcontroller, speech, one-time pad, Teensy

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä	Julkaisun laji	Päivämäärä
Nevalainen, Ari	Opinnäytetyö, ylempi AMK	Kesäkuu 2019
	Sivumäärä 84	Julkaisun kieli englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

OTP-salauksella ja mikrokontrollerilla toteutettu puheensalaus

Sähköisessä muodossa olevan puheen salaaminen.

Tutkinto-ohjelma

Degree Programme in Cyber Security

Työn ohjaajat

Sampo Kotikoski, Jari Hautamäki

Toimeksiantaja

Oma aihe

Tiivistelmä

One-time pad (kertasalaus) on tulevaisuuden kestävä yksinkertainen salausmenetelmä. Sitä ovat käyttäneet armeijat, diplomaatit, hallitsijat ja pankit. Tiettävästi se ei ole vielä ollut kaikkien saatavilla mikro-ohjaimen muodossa.

Kehitysprojektin tavoitteena oli esittää, miten edullisella mikro-ohjaimella voidaan salata puhetta reaaliajassa käyttäen *one-time padia*. Laitteen tuli olla kannettava ja avoin suunnitelmiltaan.

Kehitysalustaksi valittiin *Teensy 3.5*. Valinta pohjautui ominaisuuksiin ja erityisesti lisäosana myytävään äänikorttiin. Laitteisto vaati hyvin vähän muutoksia, minkä jälkeen käytännön työ oli ohjelmointia.

Ohjelmointi tehtiin Arduino sovelluksella C ohjelmointikielellä. Erityisesti sopivan yhteyskäytävän löytäminen osoittautui hankalaksi, kun alkuperäinen suunnitelma USB-yhteydestä isäntälaitteeseen ei toiminut odotetusti. Lopulta päädyttiin sarjaliikenneportin käyttämiseen, *Teensy* alustojen sijaitessa samalla työpöydällä.

One-time pad toimi *one-time* autentikoinnin kanssa puheen salauksessa rajoitetusti. Rajoitukset liittyivät salausavaindataan. Rajoitukset ovat ratkaistavissa lähinnä sovelluskehityksen kautta.

Puheensalaus on mahdollista tehdä mikrokontrollerin avulla käyttäen *one-time padia*. Tällöin salattu puhe on suojassa kulkiessaan päätelaitteissa ja tietoverkoissa. Se ei ole yksinkertaista, lähinnä salausavaimeen liittyvien asioiden vuoksi, mutta sen tulisi olla vaihtoehto muiden joukossa. Näin etenkin, kun halutaan, että informaatio pysyy salassa nyt ja tulevaisuudessa.

Avainsanat

salaus, mikro-ohjain, puhe, one-time pad, Teensy

Muut tiedot

Contents

Figures

Tables

Acronyms

ADC	Analog-to-digital converter
API	Application programming interface
ASIC	Application-specific integrated circuit
CMC	Computer mediated communication
COTS	Commercial off-the-shelf (product)
DAC	Digital-to-analog converter
DSP	Digital signal processor
FICORA	Finnish Communications Regulatory Authority
FFT	Fast Fourier transform
FPGA	Field programmable gate array
LSB	Least significant byte
MAC	Message authentication code
MSB	Most significant byte
OTP	One-time pad
PKI	Public key infrastructure
QKD	Quantum key distribution
SPI	Serial peripheral interface
SSH	Secure Shell
TRNG	True random number generator
VHF	Very high frequency
VoIP	Voice over IP
XOR	Exclusive OR logical operation

Glossary

Arduino	Open development board
ARM TrustZone	System-wide hardware isolation for software in ARM-platform
Audacity	Audio editor software
Cheri	Capability hardware enhanced RISC instructions is an extension bringing memory protection and software compartmentalization
DREAD	Risk assessment model with categories damage, reproducibility, exploitability, affected users and discoverability
Eagle	PCB design software*FIPS-140-2* Federal information processing standard (and its requirements)
FLAC	Free lossless audio codec, used to compress audio
HiFive1	Open hardware and software development board
I2S	Inter-IC sound bus
IDA	Interactive disassembler for software
Intel SGX	Intel software guard extensions aim to provide trusted space for instructions
Jack Pair	Salsa20 based voice encryption device in Kickstarter
LowRISC	Non-profit hardware organization developing open hardware
Metadata	Information about data
Moteino	Low power Arduino based development board
Nonce	Random number than can be used only once
Olimex	Company producing development boards

One-time MAC	Carter-Wegman's one-time MAC is similar to one-time pad but used in authentication
perfect secrecy	When ciphertext contains no info about plaintext
Poly1305	Cryptographic message authentication code used to verify integrity and authenticity
Pseudorandom data	Predictable almost random data
PulseAudio	Sound server software
Raspberry Pi	Small size PC
RFM69HW	Radio transceiver
rtl-entropy	Software that generates random data using RTL-SDR
RTL-SDR	Software definable TV-dongle
Salsa20	A stream cipher
SGTL5000	Hardware audio module
Signal	Encrypted messaging service
STRIDE	Threat model system noting spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege
Teensy	Development board
Telegram	Encrypted messaging service
true random data	Truly random data, unlike *pseudorandom data* produced often by computers
TweetNaCl	Small size cryptographic library derived from NaCl
WebRTC	Web browser based real-time communication focused project

1 Introduction

1.1 Subject and motivation

This development project presents a development process of a portable microcontroller-based voice encrypting device that can be attached to a PC and possibly to a mobile phone. It uses a one-time pad to encrypt voice before delivering it to the host device.

A device such as this is not yet available for everyone. Yet, the one-time pad (OTP) is a future proof encryption method when applied correctly. Many OTP applications exist today focusing mainly on PC software. The hardware solutions found in a preceding school exercise are mostly based on quantum key encryption, which is not small, portable, wireless and available to everyone. The exercise was also a launch pad for this thesis, as it became clear, that the planned device would be unique on its properties, being open and using OTP for voice communication. A similar product development exists with a name Jack Pair in Kickstarter; however, it uses Salsa20 encryption instead (Chang 2018).

The need for OTP based encrypted communication systems arises from a simple fact: It is the only encryption method that lasts over time and holds up against technology advancements (Chandrakar et al. 2014). It can be used in communication systems today, and it is a significant option, when the encrypted data must remain secret today and in the future.

The general reasons to encrypt communications are not within the scope of this research; however, there is plenty of material available on the subject offered by e.g. Schneier and EFF (Schneier 2016; Privacy, n.d.).

1.2 Tasks

One-time pad is already used in many systems throughout the world, but the focus has been on QKD and backbone infrastructure. Using OTP to encrypt voice allows a hardware using it to be downsized into a portable microcontroller based device.

Microcontroller based device is used because of its portable size, price, simplicity and isolation based security against many vulnerabilities originating from software and operating system vulnerabilities. Speech and one-time pad are used, since speech is considered the primary method of communication and OTP because it is the most secure encryption method (chapter 3.1). Text based communicator would cause a size dilemma.

Interconnection between analog and USB interface was chosen, because USB host device can feed power along with data traffic, so easing the development process and reducing the footprint of the device. USB is also widely available on IT devices. The analog interface represents voice input & output e.g. microphone and speaker, where-as the USB-interface is USB audio or -serial (or other device class).

Hardware based true random data generation is part of the development (chapter 4.3.1), as it is crucial for the system to work safely. The system needs random key data in order to fulfill the requirements of OTP design. It is to be created using a cheap and feasible method.

The demonstration is executed transmitting speech over the implemented OTP encrypted path to confirm the functionality and to expose possible limitations and problems. The audio channel needs to be uncompressed, since truly random data does not compress.

Voice was chosen as the communication content because it is viewed as the primary method of communication. It is speaker specific and communicates not only words, but also emotions and personality (Tiwari & Tiwari 2012). A person can be recognized by voice (ibid.).

Maximum amount of open design and code is limited because of resources and possible commercial aftermath. Whereas fully open hardware microcontrollers have started showing up during recent years and months, they lack in terms of built in features, such as ADC and DAC. Additionally, the software libraries, userbase and supportive forums are not extensive enough for such a leap, when compared to easily adaptable hobbyist microcontroller platforms. Hobbyist platforms suit particularly well for this kind of project, since everything can be published, which is often the requirement at least at some some level in brands such as Arduino and Teensy because of the licenses they may use.

The development tasks are as follows:

- Present a theory overview.
- Collect potential hardware solutions and choose the best option.
- Build the hardware platform for the software with suitable interfaces.
- Test its functionality in phases leading to audio throughput simultaneously into both directions.
- Generate true random data.
- Write the software.
- Test, review and report.

2 Research plan

2.1 Research questions

Four research questions are to be answered:

- Can a microcontroller based device encrypt speech with one-time pad?
- Can the device be portable?
- Can the device operate between audio and USB-interface?
- Can the device be open in hardware and software?

2.2 Research method

Research focuses on software and hardware development. Thus, qualitative research and experimental research are used. Software, hardware, inputs and data are manipulated, and the changes are observed. Continual change and observation sequence go on until the expected result is achieved.

2.3 Research reliability

The theory part (Chapter 3) bases mostly on online literature, articles, papers and researches and known experts in the field. Academic sources are sought especially relating to essential security and encryption topics. Guides, forums and less reliable sources are used, when needed, as they could be only sources available and in some occasions developers communicate that way. Preferably some disagreeing messages are taken into account.

The hardware solution is picked from well-known options, with the main focus on availability, support and openess. There should be an active user base with support from a developer available.

The hardware platform construction should be in its simple form plugging a module into another. This is taken into consideration, since it spares time and resources. In addition, it affects audio quality.

Since the cost of errors delivered in the latter phases of development is multiplied, the errors should be found in time. The features of the hardware are thus tested together with its modules.

True random data is a requirement. It should be tested to ensure it is fit for the purpose and does not compromise the system.

Software is written taking advantage of all easily available code and libraries that are suitable to progress the development. Code is not to be top quality, that would require more resources. Instead the main focus is on functionality. Software development follows a logical path of first ensuring the hardware functionality in software and then building the functions from most important functions e.g. *sd-card reading* to the least critical. Finally and during the software development, tests are run to find possible faults and to correct them.

3 Background

3.1 One-time pad

The one-time pad was invented by a banker named Frank Miller during 1882, and it was made known in a publication called *Telegraphic Code to Insure Privacy and Secrecy in the Transmission of Telegrams.* Nevertheless, Vernam and Mauborgne are usually credited for the invention. (Bellovin 2011).

Several requirements must be fulfilled for an encryption technique to be qualified as OTP:

- The key length must equal to plain text length.
- The key must be truly random.
- The key must be used only once. (Pardo 2013; One Time Pad Encryption, n.d.).

One-time-pad functionality is based on the concept of logical XOR-operation on two data units with the same length. XOR-operation is reversible by nature and thus it forms the basis for the OTP-functionality. XOR principle using binary is displayed in Table 1.

Table 1. XOR-operation

INPUT		**OUTPUT**
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Assuming the data unit is a byte consisting of 8 bits. In its simplicity, it can be put into a following example frame: A plaintext *"Q"* with which a secret key *"W"* is XOR-operated, producing a ciphertext of *"E".* Now using the *ciphertext XOR secret key* the original plaintext "*Q*" is produced. When applied to words, a plaintext of *"duck"* could be interpreted as *"word"*, "*idea*" or whatever four-letter word one can

imagine, when the secret key remains unknown. A more detailed example is presented in a research paper by Devipriya and Sasikala (2015). Should hexadecimal cipher text *16 0a 15 0e 14 09 6d* be XOR-operated with some random key, the result could be equally random as seen in Table 2.

Table 2. OTP principle

Keys		**Operation**	**Result**
Key 1	QETYUPL	XOR	GOAWAY!
Key 2	XFYBYEW		NLLLML:
Key 3	ZCVMNXI		LICCZQ$
Key 4	QLASUYK		GFT]AP&
Key 5	EOV\Q]R		SECRET?

3.1.1 Benefits and limitations

Most used cryptographic algorithms provide just practical security (Borowski & Leśniewicz 2012, 6). That is not the case with one-time pad that is the only unconditionally and perfectly secure cryptographic mechanism (Elbirt 2009, Chapter 4.2; Meter 2013, Chapter 5.1; Pardo 2013, Chapter 3.2). OTP encrypted message contains “no information about the original message” (Kleidermacher & Kleidermacher 2012, Chapter 4.3).

One-time pad is the only system providing perfect confidentiality (Anderson 2001, Chapter 5.2.2; Schneier 1996, 26). It has very simple principle and it is simple to implement with technology, nevertheless it survives over time and technology advancements (Kleidermacher & Kleidermacher 2012, Chapter 4.3; Schneier 1996, 28; Rijmenants 2017).

Limitations to one-time pad’s usage are mostly key based. The key distribution is cumbersome (Bloch & Barros 2011, Chapter 3.1). It requires a secret delivery for a great amount of data. Key synchronization takes effort, as both ends need to be aware of the current position in key data (Swenson 2008, Chapter 4.14; Pardo 2013, Chapter 3.2; Schneier 1996, 28). Key and message lengths need to be equal and same key cannot be used twice (Pardo 2013, Chapter 3.2; Elbirt 2009, Chapter 4.2;

Schneier 1996, 26). In the counterintelligence program called the Venona project, same key material was used more than once (by target) which led to successful decryption of OTP traffic (Benson 2001). Though it could be argued that it was not real OTP if it was used more than once. Moreover, truly random key is hard to create (Pardo 2013, Chapter 3.2). E.g. computer can not create it and the increasing traffic speeds in digital networks demand more capable random data generators than before.

One-time pad does not protect the message's integrity nor provide authenticity (Anderson 2001, Chapter 5.2.2; Rijmenants 2017; Schneier 1996, 28). Schneier also claimed OTP is not for mass-market while Anderson wrote that it is "too expensive for most applications" (ibid.). In conclusion it has limited use in most applications because of the downsides (Pardo 2013, Chapter 3.2).

It should be noted that the long key length is also a benefit, since if, e.g. thousand persons each have a 2 GB of key data, it demands 2 TB of capacity for an actor to store all the key data. It could be said that implementing an impractical OTP makes its surveillance impractical.

Key distribution compared to PKI infrastructure is also beneficial when keys are delivered physically and kept secret until their deletion, since public keys are supposedly broken by quantum computers in the future, if they are ever developed in such a scale (Chen et al. 2016), whereas OTP is and will be secure. Moreover, key distribution has become easier due to technology advancement (Borowski & Leśniewicz 2012, 6).

Anderson wrote in 2001 that the OTP is too expensive to apply for most applications because of the high consumption of the key material (Anderson 2001, Chapter 5.2.2). Jenkin and Dymond (2002) from York university wrote that 16 – 64 MB SD-cards were available back in 2002, when they planned an OTP-based implementation using Palm505 handheld. This, however, is even less of an issue nowadays, since the prices have gone down while the capacity has increased. Amazon's bestseller micro-SDXC 64 GB memory card is sold with a price of 21.99 $ (SanDisk 2017). This would provide enough space for key material to operate 138 890 hours (64 kbps); in addition, with

the same amount for receiving. Limitations can be handled using several means of few of which are mentioned in Table 3.

Table 3. Compensating the OTP limitations.

Key distribution	• Distribute the key physically using secure methods (Piper & Murphy 2002, Chapter 4). • Quantum key distribution (Rijmenants 2017; What is Quantum 2014)
Key synchronization	• Jump to a certain point of key on error (Swenson 2008, Chapter 4.14).
Key length	• Share a short random key and use it to generate a longer one (Kleidermacher & Kleidermacher 2012, Chapter 4.3). This is no longer OTP.
Key randomness	• Use a TRNG (Kleidermacher & Kleidermacher 2012, Chapter 4.13; Rijmenants 2017). o Collect real world randomness, which can be from Geiger counter, free running oscillator, diode thermal noise, readymade chips etc. (Schneier 1996, 354-355). o Confirmed by compressing the data. If it compresses, it is not random (ibid.)
Message integrity	• "Use a hash algorithm on the plaintext and send the hash output value, encrypted along with the message" (Rijmenants 2017)

3.1.2 Usage and key distribution

Because of the trouble involved with the keys, OTP is a good option for some certain scenarios only (Piper & Murphy 2002, Chapter 4). Especially, in a chatroom or similar, where many people want to discuss with each other, the OTP system becomes impractical because of the amount of key material needed (Meter 2013, Chapter 5.1). Nevertheless, one-time-pad has been known to be used in several occasions since 1940s:

- World War II in multiple purposes including diplomatic traffic (Anderson 2001, Chapter 5.3.2).
- Espionage (Anderson 2001, Chapter 5.2.2; Rijmenants 2017).
- Between Washington and Moscow (Pardo 2013, Chapter 3.2; Piper & Murphy 2002, Chapter 4; Schneier 1996, Chapter 1.5).
- Supposedly American diplomatic communications during 1970s in Kenya, South Africa, Lebanon, Syria, Ethiopia and Spain (Situation in 1975; Technical inspection 1974; COMMUNICATIONS: COMSEC 1976; EVACUATION OF 1976).

Several companies deal with devices or software using OTP. The most credible ones deal with quantum key distribution (QKD). Both QuintessenceLabs and Toshiba have their devices that use Quantum channel for key distribution. QuintessenceLabs uses normal Ethernet for OTP encrypted data. It can achieve 1 Gb/s speed in its full entropy random data generation, that equals the maximum possible traffic flow (QuintessenceLabs delivers 2017). There are other operators as well (Silver 2017).

Non-QKD OTP uses a trusted courier or similar safe physical method for key delivery as illustrated in Figure 1 as stated earlier. Key distribution becomes burdensome, as there is a need to transfer a storage device.

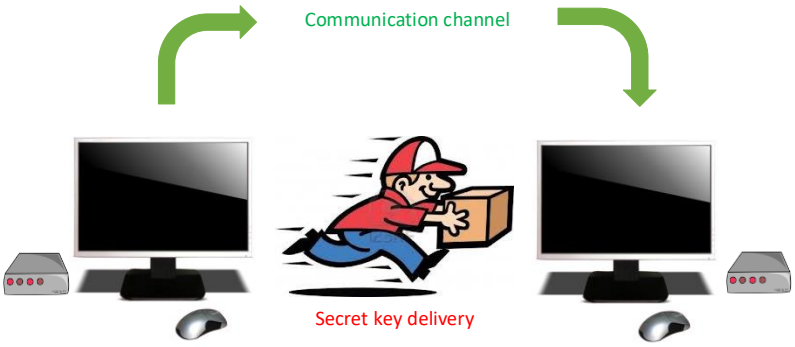

Figure 1. OTP key delivery

Quantum key distribution system shares encryption key in its quantum channel (Figure 2). It is secure because all eavesdropping attempts can be detected. (Toshiba's QCCS 2015). Multiple key related issues have been solved with QKD; however, it is not suitable for mobile usage or for individual people. It is, nevertheless, excellent for companies, organizations, ministries and similar entities, where backbone infrastructure exists. It is and has been used in such and other instances (On Security Issues of QKD):

- Geneva elections 2006-2007.
- SECOQC project 2008.
- SwissQuantum 2009-2011.
- South Africa 2009.
- Singapore 2009.
- China 2009.
- Madrid.
- Tokyo 2010.

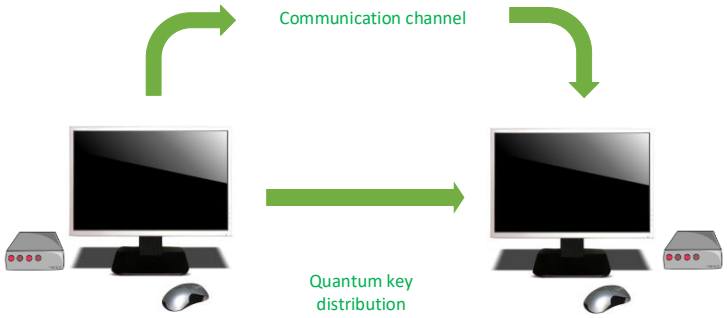

Figure 2. QKD key delivery

One-time pad is also used by US military, government and some bank in Australia (Darren 2015; McLean 2016). These assumptions are based on the idea that using QKD makes no sense without applying OTP alongside.

While one-time pad provides confidentiality, it does not solve the issues of integrity and authenticity. Authentication along with one-time session values can protect against masquerading, content modification, sequence manipulation and submission modification (Tipton & Krause 2007; Halunen et al. 2018, 52). Authentication should take place after the encryption. The method is called *encrypt-then-authenticate*. **(**Krawczyk 2001; Houthen 2017, 130).

3.2 Legislation

In Finland, the Finnish Communications Regulatory Authority (FICORA) monitors the compliance of Information Society Code that deals with protection of communication. An electrical message along with its identification data can be protected by using available technical methods, unless forbidden elsewhere in law. The protection should not interfere network service nor the communication service. (Information Society Code 917/2014, 269 §). Moreover, the constitutional law aims to protect private life, honor and domestic peace. A letter, call or other confidential message should remain secret. If constitutional rights need to be safeguarded or

crimes solved, certain measures can violate the domestic peace. Message confidentiality can be intervened by legislation in case of (Finnish Constitution, 731/1999, 10) the following:

- threat against individual's or society's security,
- crimes endangering domestic peace,
- trial,
- security check,
- deprivation of liberty,
- information gathering of military operation or
- other serious threat to national security.

An expert from FICORA explained in an email reply that using OTP and Salsa20 for securing communication is permitted, when they do not interfere with other systems. In addition, the law rather protects privacy and confidential communication.

A publication *Encryption and protection methods in electronic communications* from Ministry of Transport and Communications from March 2018 confirms the prior information. Government intelligence forced decryption might prove impossible and require weaker encryption systems or backdoors. This would cause further problems with businesses and seems unlikely now. (Halunen et al. 2018, 37-39). The authors conclude that

- Finnish encryption methods should remain free of backdoors,
- the national know-how in encryption methods should be taken care of,
- Finland should be pioneer in information security as well and
  Quantum computing must be considered in the solutions (ibid., 61-62).

In Europe, the proposal for Regulation on Privacy and Electronic Communications explains that electronic communications data are to be kept confidential (COM/2017/003 final, 14). Their content belongs to the "*right to respect for private and family life, home and communications*" and interference can only be allowed under "*very clear defined conditions, for specific purposes and be subject to adequate safeguards against abuse*". The terminal equipment belonging to end-user and involved in electronic communications are members of the private space and require "*enhanced privacy protection*". (ibid., 16).

Since the nature of one-time pad and exclusively its key length, it is illegal in some cases. This is the case with China especially; however, with also the United States, in some measure (Saper 2013, 678-784; 2018 REPORT ON, 83-86).

There is also Wassenaar Arrangement control regime on dual-use goods, conventional arms and technologies that lists controls, however, they do not affect software that is

- "*generally available to the public*",
- "*in the public domain*" or
- *"the minimum necessary*" object code "*for the installation, operation, maintenance (checking) or repair of those items whose export has been authorised*".

As the software result of this project is going into a public domain, there should be no trouble with Wassenaar Arrangement that covers most of Europe and North America. (List of Dual-Use 2017, 3). The dual-use list further explains that the Category 5 part 2 is "*not used since 2015*" and does not "*apply to products when accompanying their user for the user's personal use*" (ibid., 91). Until then, there were limitations on symmetric and asymmetric encryption key lengths (ibid., 93).

3.3 Hardware

Even the most sophisticated digitally applied encryption runs on a hardware operated by a software. Moreover, while one can still mostly distinguish people from each other, it is very hard for a normal person to confirm the correct level of security of a software or electronic hardware. It requires expensive hardware and knowledge to see inside the electronic chips and understand the sight.

Hardware level viruses or bugs can infect all layers of the software system that depends on the hardware (Yang et al. 2016, 1). Most software tools do not even find them (Fournaris et al. 2017, 2). Hardware Trojans can be set into ASICs, COTS -parts, microprocessors and -controllers and DSP- and network processors. FPGA bitstreams can be altered as well. (Tehranipoor & Koushanfar 2010). The malicious hardware can be sized in micrometer range, when it uses only a gate of a transistor. Since there

may be 100 000 gates, it is hard to detect the unwanted one. Authors of a research wrote that even an electron microscope is unlikely to help. (Yang et al. 2016, 16). They think that "*trusted circuits should monitor the execution of untrusted circuits*" and their behavior (ibid., 19).

Hardware threats are not limited to only Trojans, but there are backdoors and counterfeits as well (Jin 2015, 8). However, there has been effort to bring trust into hardware level. ARM TrustZone partitions hardware and software into secure and non-secure, departing them with a security wall by using access control, permissions, secure system calls and -interrupts (ibid., 10). Intel SGX, CHERI and LowRISC are other assistants in field of hardware security (ibid., 11-12).

When utilizing the encryption in hardware before reaching a PC or a mobile phone, no operating system, software or hardware vulnerability in those host devices can reach the audio between the end user and the microcontroller, nor the device's firmware. The encryption could be applied with PC, mobile phone, Raspberry Pi, Arduino or with similar hardware; yet, the size, portability and the software make a difference. There is no need for extra complexity and size brought in by Raspberry Pi, large size by normal PC or not so much support for USB audio in Arduino. Nor is there a need for an operating system, as the task is simple and can be mostly automated.

In long term and considering the goals of the thesis, the microcontroller board should be open source as well; however, during this development process it is not seen as requirement, as it would only lengthen the work. Yet, the resulting functionality should be applicable to other common microcontrollers as well.

Even though embedded systems tend to have less software, interfaces and internet connectivity, they are not necessarily more secure than e.g. general PCs. Embedded systems have insufficient security design, implementation and too hard update process (Papp et al. 2015). It is not easy to protect them.

Not even the best system is perfectly and eternally secure; very least the worse systems. Both software and hardware can be insecure; however, if hardware is insecure, it can undermine all software protections abiding on a higher level in the system. Embedded systems make not much difference, since they are vulnerable as well, except perhaps less since the fewer number of the interfaces they reveal to

normal usage. Wherein a normal computer running e.g. Windows has usually Ethernet and/or WLAN and USB ports, an embedded device does not need to be as general and can expose only e.g. serial port. The serial port can be separated from the programming interface, thus invalidating the channel of reprogramming or infecting the software through the interface connecting it to outside world.

3.4 Open source

The source code of open source software can be seen and read by anyone. This means not just developers, but also experts and novices can investigate it. They can find bugs and make other suggestions. (Clarke et al. 2005, 6-7). Additionally, the observers can become developers and either advance the existing software or make their own version (ibid., 26). A software can thus update faster and live longer in some form (ibid., 27). However, Viega claims that usually it is a small group of people who inspect the source code. The idea of open source can create "*a false sense of security*", as people trust the other people to review the code. Reviews could be too few because (Tipton & Krause 2007):

- Code can be messy, complex and suffer from uneven reviews.
- Reviewer may have to fight the mood of doing a monotonous job and should have comprehensive view of the software.
- Developers may not understand security well and be looking to create e.g. new features instead of focusing to quality.
- Documentation is often too inadequate.

Moreover, if the software is offered by a community instead of a company, some corporations are reluctant to use it (Clarke et al. 2005, 23). Although closing source code from curious eyes could indicate that something is wrong with it (Tipton & Krause 2007). This seems to be the case with Microsoft:

> *In May 2002, Jim Allchin, Group Vice President for Platforms at Microsoft, testified before a federal court regarding the security of Windows itself. Among some rather fascinating commentary, Allchin claimed that exposing the source code and details of the application programming interfaces (APIs) for Microsoft products would represent a threat to national security. Apparently, there are problems so significant in Windows that mere disclosure of the source would threaten us all.*

> *When asked about which areas were of most concern, Allchin mentioned Microsoft's message queuing functionality. This capability supports retrieving user input from the keyboard and mouse and passing that input to applications. Allchin did not want to divulge details, and admitted, "The fact that I even mentioned the message queuing thing bothers me."*

Attacks were released against the message queuing, after the disclosure. One or more were successful and inspired by Allchin's comments. (Tipton & Krause 2007).

Closed source software could be safer, if the code was protected and remained closed for its lifetime. However, it can usually be reverse engineered. (Clarke et al. 2005, 30). Code is not needed to find vulnerabilities, instead, there are tools (Tipton & Krause 2007) that allow

- an insight view of the "executable program's code",
- a walk through the program's function calls "step-by-step to see the flow of the program and determine how to break it",
- attacker to step and inspect the raw machine language code,
- attacker to manipulate data structures and parameters during the program execution and "inject faults into the program to see how it bleeds" and
- "attacker to inject random-looking data into a program to see if it can cause it to crash".

Tool include e.g. APISpy32, Sharefuzz, SPIKE, Heap Debugger, APIHooks, Feszer and IDA Pro. (ibid.). IDA is still active.

Open source software rather favours correct security practices through transparency, wherein it involves multiple people. This can provide people more secure, reusable and working code to use. (Clarke et al. 2005, 30). In closed source software, the security issues stay hidden until some most likely experienced or well-funded attacker finds them (Tipton & Krause 2007).

Bahamdain wrote a survey paper about open source software quality assurance. It listed multiple aspects (see Table 4) from users, developers and system's point of view. (Bahamdain 2015).

Table 4. Pros and cons of open source software

	Pros	Cons
Users	"flexibility, strong value, code availability, possibility to modify the code, knowledge sharing through the community, and increased motivation"	"incomplete or bad documentation, unstructured development methodologies, and irresponsible individuals"
Developers	"the ability to create customizable solutions, the potential to reuse many existing parts and functionalities, reduced production time, and increased motivation"	"the lack of tools, collaborating with new developers, and reviewing large projects"
System	"faster bug detection and fixing, more reliability, the freedom to customize, cost effectiveness, re-usability, rapid evolution, portability, and multitude licensing"	"the lack of formal process centralized management release and documentation, poor design, no single responsibility for problems, version proliferation, and the difficult estimation of manpower requirements, complex licenses, and high short-term cost"

The security of software is not connected to the openness of the source code. Instead, security depends on the software development process and the development team. Correct software design, "*careful implementation and comprehensive testing*" make a difference along with proper configuration and maintenance. (Tipton & Krause 2007).

4 Research implementation

4.1 Threat model

Threat modelling is for threat identification in the system. It can reduce vulnerabilities when applied in an early phase. There are multiple methods available, e.g. STRIDE, security cards and persona non grata (Shull 2016). Yet in this project, this is not extensively composed. Should a product go into production, it would be crucial.

In this project in its simplest form, there are two units communicating with each other. Both have audio in and -out, key data, functions and the gateway / transfer medium in between as seen in Figure 3.

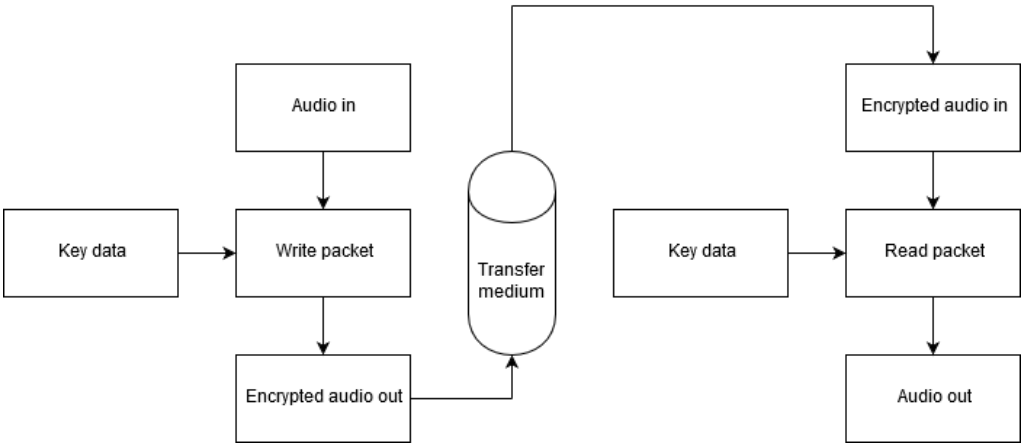

Figure 3. Operation principle

There are many vulnerabilities that could be considered, yet some are more likely than others. Vulnerabilities include but are not limited to:

- Device is compromised before delivery
  - Threat: E.g. hardware Trojan
  - Mitigation: Safe delivery and/or trusted hardware solution
- Device is left unattended
  - Threat: E.g. memory copy, firmware update

    - Mitigation: Small size to encourage carrying in pocket, shielding/protecting in hardware and software
- Device is lost
    - Threat: Key material leak
    - Mitigation: Add authentication method for operator, e.g. touchpad gesture
- Conversation is bugged
    - Threat: Can happen by host device or by external devices e.g. headset
    - Mitigation: Avoid unsupported devices, reliable shielding
- Denial of service
    - Threat: Likely to target gateway communication meaning that it should be changed e.g. from Skype to VoIP/GSM
    - Mitigation: Ease gateway change process
- Information leak by cryptoanalysis
- Software bug
- Traffic analysis and pinpointing endpoint devices
- Key data is invalid, stolen or reused

The short mitigation summary is that the device should

- use secure/trusted hardware parts,
- use well coded software,
- have good instructions for safe operation and
- be of small size.

4.2 Hardware

Hardware decision defines many after comes which means it should be considered precisely. Software used in hardware should be adaptable into an open source commercial product, like LoFive RISC-V, HiFive1, Arduino or similar. Olimex and Teensy are less open options.

Multiple things should be considered including:

- Support in the form of
    - libraries,
    - support and
    - size and activity of community.
- Openness of the design:
    - Hardware schematics
    - Hardware layouts
    - Bill of materials
    - Programming
- Reusability of everything
- Portability

In long term, choosing Olimex could make sense because it is more professional oriented by nature. Especially sharing their Eagle layout and schematics for some products is beneficial. However, it lacks in support and community, wherein Teensy does not. In addition, for Teensy there is SD-card slot, ready-made audio library and -shield in small size. Portable device is desired (Chapter 1.1). It was also the easiest platform for the project, so it was chosen to be used. Teensy boards were connected to audio boards and then to breadboards as illustrated in Figure 4.

Arduino 1.8.4 was installed since it is supported by a software add-on called Teensyduino 1.39 that supports the Teensy audio library. Install process was described on Teensy website (Stoffregen, n.d.f).

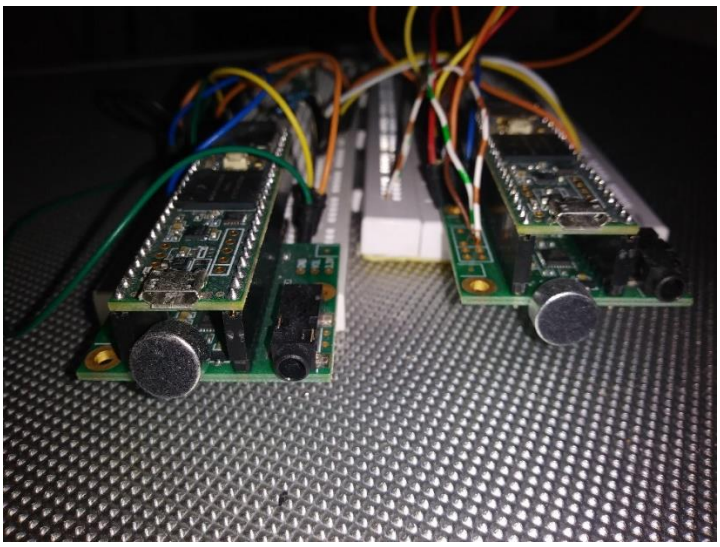

Figure 4. Teensy 3.5 boards together with audio boards below them

After the installation process, a USB-cable was connected to Teensy 3.5 board that had the audio adapter attached. Several tests were needed to test the functionality.

The needed features were:

- SD-card reading
- Audio inputs
- Audio outputs

The tests were carried out running some example programs to confirm the functionality of the hardware. They shipped with the Arduino and Teensyduino platforms:

- BlinkWithoutDelay. Programming and status LED.
- PassThroughUSB. Audio USB-input and headphone connector output.
- WavFilePlayer. SD-card reading and headphone connector output.

Only one audio input and output were available at this point, since no microphone nor audio lineout connector had been attached yet. Thus, they needed to be connected and soldered before one additional test called *FFT* that displayed a frequency analysis of microphone input in serial monitor.

Multiple program codes required some changes to work correctly but eventually, all of them worked confirming that the required features were fully functional, when using USB as the primary audio input and output. A normal headphone jack and microphone were also used on the Teensy board. The board wirings are visible in Figure 5.

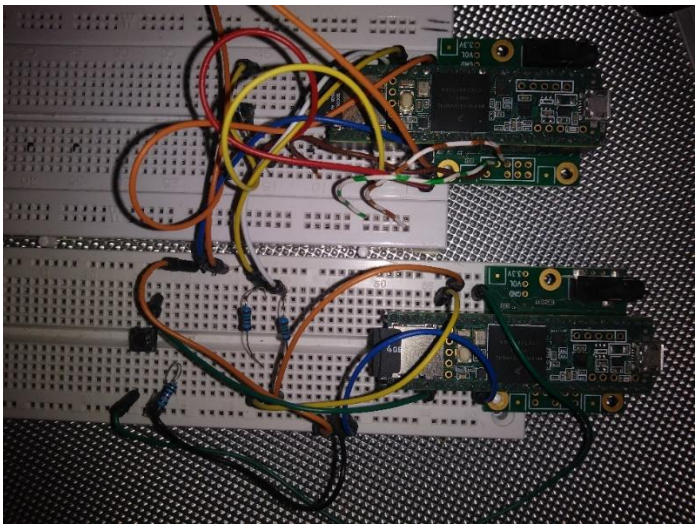

Figure 5. Circuit wiring from above

4.3 Software

4.3.1 Random data generation

Random data is crucial and required for one-time pad encryption wherein random key is used only once. It is not usually available on PC that can produce pseudorandom data. At least a gigabyte of random data must be created for OTP testing. The resulting *null* values in the random data must be replaced with *non-null* random values.

A radio receiver can be used to produce random data. (Halunen et al., 56). Cheap and easily available TV tuner dongle may be used as a software defined radio. It is often known by name *RTL-SDR*, and it is based on RTL2832U chipset (Mishra et al. 2017). These devices can be bought for 20 $ or less from the internet and their

- frequency range can be as much as 52 – 2200 MHz,
- maximum feasible sample rate is 2,4 MS/s and
- ADC resolution is 8 bits.

While there are many other and better options available, the price for features value is excellent on these dongles. They can be used to monitor a wide spectrum of radio signals with a variety of software. Most known probably being SDR#, GNU radio and GQRX.

To generate random data, there is a software on GitHub by Paul Warren called *rtl-entropy* (Warren 2018). It utilizes multiple libraries to create random data from atmospheric noise. After collecting the raw data, it makes Von-Neumann debiasing and runs the result through FIPS-140-2 tests. Theoretically, the maximum speed with RTL-SDR could be 19.2 Mb/s; however, it depends at least on the frequency used and the amount of random noise. On some frequencies it did not seem produce any data while on others it produced more than 1 GB/h. Using amplified antenna with maximum gain of 20 dB (VHF) also increased the randomness. While the software run in Linux, it printed following lines into to the terminal:

- Failed: FIPS 140-2(2001-10-10) Monobit
- Failed: FIPS 140-2(2001-10-10) Long run
- Failed: FIPS 140-2(2001-10-10) Runs

- Failed: FIPS 140-2(2001-10-10) Poker

The messages were rarer, when more amplification was used. Random data generation process was started by running *rtl_entropy -s 2.4M -f 135.5M >rdata.bin* until 1 GB of data was saved. It was tested afterwards (Chapter 5).

4.3.2 SD-card and audio

SD-card storage

The random data (encryption key) needs to be stored somewhere. SD-card is used for that purpose as there is no better alternative. The card and the involving software create limitations that need to be considered. The limitations affect the programming.

Key file sizes must be considered from the perspective of relatively limited microcontroller compared to full scale PC, where they are created in. Arduino's SD-library supports a function called *position,* that gets the current byte position inside a file (Position 2019). It returns *unsigned long* that has range from 0 to 4 294 967 295. Thus, the maximum key file size can be approximately 4.294 GB to keep track of the current reading position of the file.

Nearly a must-have feature is to read two files simultaneously to have full-duplex communication. This could appear to be complicated, as two files are positioned in separate areas on SD-card. Default Arduino SD-library does not allow multiple simultaneous files open at all (SD Library 2019).

The SD-card reading process is likely to bring lag. To minimize that according to the developer of Teensy, best speed should be achievable by using 512 bytes block size, that equals the sector size (Stoffregen 2018). In addition

- file reading must end latest at *file size – 512*
- file size must be more than 512 bytes

Audio features

Audio processing consumes more resources than e.g. a text processing. It demands more speed from the processor and more space to store the audio. In addition, some metadata needs to flow in or with the audio. These must be considered prior to programming.

Using only half-duplex communication initiated by push-to-talk -button simplifies the design process. Existing Teensy audio boards with SGTL5000 chips can be used and they have e.g. microphone amplifiers, I2S communication, headphone amplifier, line-out connection. In addition, people do not make up good conversation by speaking at the same time in both ends and key data is spared by enabling the data flow only when activated. A button activation is likely the best, as voice level activation or similar method is not only more complicated, but also more prone to errors.

Audio data is stored in 128 units long *short integer* -type arrays. *Short integer* is 16 bits with the processor used. This was confirmed by increasing its value until overflow.

There is no metadata option to be included within arrays on Teensy audio library environment. Therefore, there were a few things that could be done. Either audio was to be compressed, or quality/sample amount decreased. Compression would have required FLAC or similar lossless codec, increased delay and the load on CPU, made software more complicated and be in vain anyhow, since it is useless with truly random data. Instead, quality was altered by using a header and a footer wrapping the message in between.

4.3.3 Stream cipher

Albeit one-time pad is used, an authentication is required as well. Usable authentication functions often ship with cryptography libraries. Thus, it was a potential bonus if symmetric encryption could have been used in the project as well.

Such an encryption does not need one-time key material and is easier and cheaper to use in terms of encryption key. Yet it is not as robust.

While seeking for one-time authentication, multiple libraries were investigated. Most promising appeared to be TweetNaCl, that provided all 25 NaCl functions in a compact form (Bernstein et al. 2017). This opened a possibility to add authenticated symmetric encryption and possibly other functions as well, into the software. Default cipher for authenticated symmetric encryption was Salsa20, which is a stream cipher. This means it produces a cipher text with size equal to plaintext (Aumasson 2018). Thus, it is probably feasible to add both symmetric encryption and authentication if there is no speed or load issue because of this. In addition, this will spare memory and allow drop into less secure symmetric encryption mode, if needed. OTP could be used alone, but the risk of not running random enough key material is always present, unless the material is checked before XOR-operation. Yet that is not resource friendly and is not easy to do comprehensively.

Some best practices for Salsa20 are

- to use default number of rounds (20) instead of 12 or 8 rounds. In theory, Salsa20/8 is already broken, because key can be recovered with less than 2^{256} operations and
- not reuse nonce with same encryption key. (ibid.).

Practises for safe operation for this authenticated encryption function (*secretbox*) are

- using sequential nonces, e.g. 1,2,3… but also, because of 24 nonce bytes, random nonces have only negligible risk of collision and can be used (Bernstein et al. 2019a),
- filling *crypto_secretbox_ZEROBYTES* length with nulls (0) on each message's beginning and counting them along in *mlen* (Bernstein et al. 2019b) and
- checking before *crypto_secretbox_open* decrypting operation "that the first *crypto_secretbox_BOXZEROBYTES* bytes of ciphertext *c* are all 0" (ibid.)

In addition to Salsa20 encryption, Poly1305 is used for authentication. This one-time authentication in NaCl uses Poly1305-AES message-authentication code (Bernstein et al. 2019a). It computes 16-byte authenticator for message lengths up to at least 4096 bytes. It is closely related to AES, as breaking AES would break Poly1305-AES, but only for AES part. AES function could be switched to another, should it be broken. (Bernstein 2015, 2.).

Only a few hundred lines were taken from the library (See Appendix 1). Just enough to be able to use Salsa20 and Poly1305 together or separately. Simple test was run to see if encrypting and decrypting function worked as was expected, and they did. What is more, the original plain text length remained unchanged, even though authentication is used as well as seen in Figure 6. The authentication was tested to see if ciphertext changes between *0* and *crypto_secretbox_BOXZEROBYTES* or *crypto_secretbox_BOXZEROBYTES* and *mlen* would affect the outcome, and they did. Decrypting function returned error code and did not return the plaintext, when it noticed a mismatch, e.g. flawed ciphertext.

```
Message:
115     101     99      114     101     116     32      109     101
Ciphertext:
132     26      2       106     133     5       108     231     233
Message:
115     101     99      114     101     116     32      109     101
```

Figure 6. Symmetric encryption test

Calls for encryption and decryption functions *(Bernstein et al., 2019b)* look the following:

- *crypto_secretbox(ciphertext, message, message length, nonce, secret key)*
- *crypto_secretbox_open(message, ciphertext, ciphertext length, nonce, secret key)*

Nonces alter the ciphertext and they need to be same for both encrypt and decrypt operation or otherwise the decrypt operation will fail as can be seen in Figure 7. There are at least two ways to keep them equal in case of full-duplex communication:

- Use separate keys for transmitting and receiving on both ends.
- Use e.g. even-numbered values from 2^0 to 2^{128} for transmit and odd numbers for reception.

```
Message:
115     101     99      114     101     116     32      109     101     115
Ciphertext:
63      21      48      247     73      234     150     101     183     28
Message:
0       0       0       0       0       0       0       0       0       0
```

Figure 7. Failed decryption results in empty plain text

4.3.4 Test platform

A setup was needed wherein two devices would communicate through USB to PC and from PC to another using some other interface. Very least the two devices should communicate with each other. One-time pad encrypted audio would flow into both directions.

Audio over USB and network

Most simple solution for real time raw audio transfer in network seemed to be audio over SSH connection. It would not be the easiest solution for real world, but for this test it would be enough. Two Raspberry Pis from 2011 with Raspbian were used to host the devices. The step by step process was following

1. Load *PassThroughUSB* code to Teensy boards using a USB type that has *audio*, after which they appear as soundcards to computers
2. Download and install Raspbian image to SD-card
3. Boot to Raspbian
4. Change settings in *raspi-config*, e.g.
    a. change boot to CLI with automatic login
    b. root password (for user pi), default is raspberry
    c. keyboard layout
    d. hostname
    e. expand filesystem
5. Set static IP and gateway address
6. Connect to network
7. Upgrade and update
    a. *sudo apt-get update && time sudo apt-get dist-upgrade*
8. Enable ssh in *raspi-config* for remote control
9. Install PulseAudio etc.
    a. *sudo apt-get install pulseaudio pulseaudio-module-zeroconf pavucontrol paprefs alsa-utils avahi-daemon*
10. Backup and configure */etc/pulse/default.pa*, by uncommenting/adding
    a. *Load-module module-native-protocol-tcp*
    b. *Load-module module-zeroconf-publish*
    c. *Load-module module-zeroconf-discover*

11. Run *start x* and *paprefs*, enable Make discoverable PulseAudio network sound devices available locally and all three available options on Network Server tab. And *Simultaneous Output* option to output on all local sound cards.
12. *pactl list sinks short* displays *Tunnel.raspberry…Teensyduino…* that can then be chosen with *pacmd set-default-sink <id>* on both RPis. Tunnel term represents the remote Teensy board.
13. *aplay /usr/share/sounds/alsa/Front_Right.wav* was run to test the remote output on both ends but it appeared to be unfunctional or too quiet. Test audio of over 3 minutes long using 16 bits and 44 100 kHz was used instead and it was confirmed that audio is present but too quiet.
14. *Pactl set-sink-volume <id> 100%* was run successfully. Audio was noticeable. Music could be heard in both ends in real-time.

Each time RPis would restart, they needed to rerun

- *pactl list sinks short*
- *pacmd set-default-sink <id>* and
- *pactl set-sink-volume @DEFAULT_SINK@ 100%*

It was because sink list would change order. The same procedure should be done, if either Teensy would be reconnected. However, as the settings can be static, they could be set to default by modifying *./etc/pulse/default.pa.* In the end of the file, it should state e.g. following:

Make some devices default

Set default-sink output tunnel.raspberrypim.local.alsa_output.usb-Teensyduino_Teensy_MIDI_Audio_2447880-02.analog-stereo.2

Set-default-source input alsa_input.usb-Teensyduino_Teensy_MIDI_Audio_2447850-02.analog-stereo

Yet, the defaults set in *default.pa* did not function. Using microphones instead of audio files also proved to be an obstacle, as most simple solution of using *pavucontrol* in *X Window System* was not working for RPi, as CPU usage rose to 100 % and the OS became unstable. It supposedly should have been possible to use *pavucontrol* kind of functionality in terminal by writing *pactl load-module module-loopback latency_msec=1 source=1 sink=0* . Yet this did not fully work, as it could only loopback in same hardware but not through a tunnel.

Using old Pentium 4 computer in the other end proved to be a viable option. Live audio from a microphone could be properly routed in a tunnel, when set in *pavucontrol*. Two computers were used instead.

A software firewall caused trouble at times and required reboot/*pulseaudio* restart sometimes. In addition, *module-loopback* would be only required if no application was playing sound. Therefore, with loopback, no application was required. Loopback was set with *pactl load-module module-loopback latency_msec=100.* Settings in Volume Control would display (when *Show: All streams*) Loopback from Teensy MIDI/Audio Analog Stereo *on* Teensy MIDI/Audio Analog Stereo on *<hostname>.*

Audio over serial interface

In principle the setup was working, but mostly with an application as input instead of Teensy microphone. Audio through USB interface feature (Chapter 1.1) was thus abandoned. Teensy boards were instead connected using their hardware serial ports to make things simpler and more functional. Another one's audio hosted by Windows PC and another by Linux PC. Software on Linux was used for reception and software in Windows for transmitting until the very last version. Transmitted audio was picked from microphone and received audio was played back to headphone-jack, that was connected to either to headphones or Windows PC audio input, which was monitored by Audacity. USB was for programming. The connections between PC, Teensy- and audio boards are seen in Figure 8.

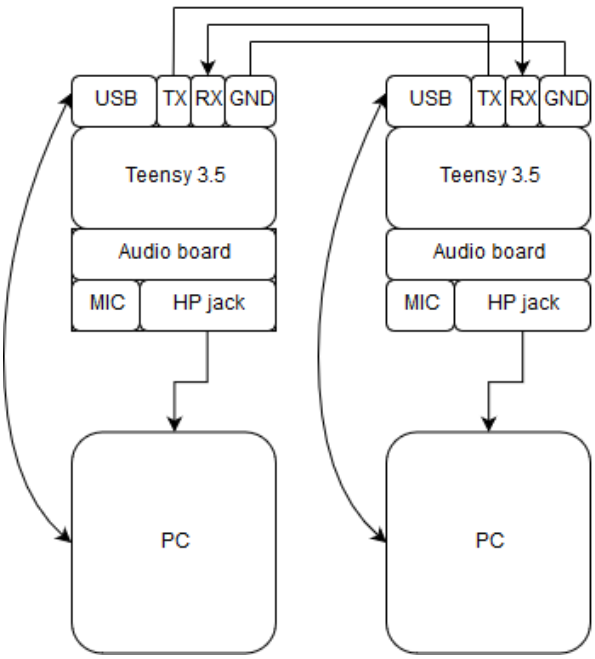

Figure 8. Device connections

4.3.5 Programming

Final programming started after all preceding tasks had provided the proper platform to run the software. The research question and objectives should be completed as the result of the programming.

Right from the start serial port messages, LED indicators etc. needed to be used to see if software functions as supposed. Debugging was not as easy as in *Microsoft Visual Studio* and similar. Three methods of receiving state information from software were one LED, serial messages and audio output.

The testing was performed along with developing. Over complicated structures would only make fault finding more difficult, and thus everything should be kept simple and tested early enough in the process. The next module was added after the previous was considered functional.

Programming was done using C programming language. It started by creating *SetPacket* and *GetPacket*. *Audacity* was used to record and play for transmit and receive. It appeared as if static audio packet values like *ZZ* were not working. To ease the development, the audio was altered greatly enough to confirm in Audacity, that it was indeed manipulated by *SetPacket* as supposed. Audio was played to Teensy's microphone, modified by the code and output was finally picked up by Audacity for analysis. Figure 9 illustrates the method used for modifying the audio packet.

```
j=0;
while(j<256) // i
{
  tmp=audio[j]&0xF000; // FFF0
  audio[j]=tmp&packet[j];
  j++;
}
```

Figure 9. Altered audio packet

First error and debugging

Changing highlighted value (Figure 8) from 0xFFF0 into 0x0000 step by step made no noticeable difference as seen in Figures 10 and 11. Unconsistent music was used to produce the waveforms.

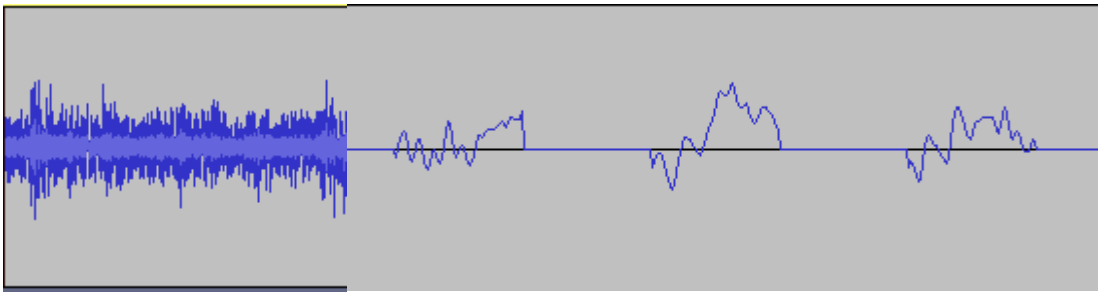

Figure 10. AND 0xFFF0 modification with two views

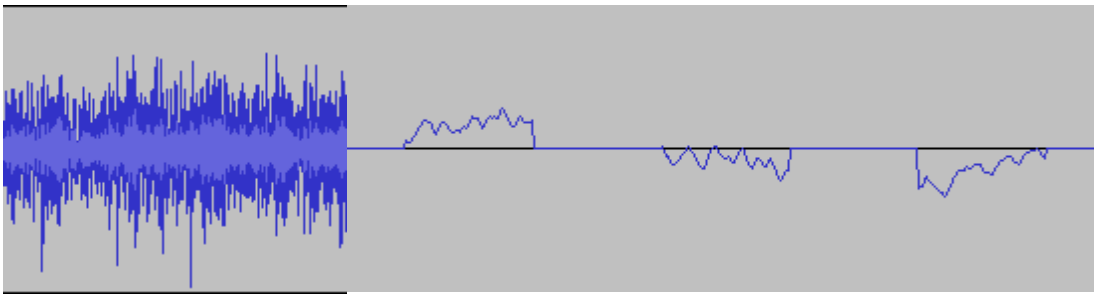

Figure 11. AND 0x0000 modification with two views

Next, a deeper modification was used, changing 64 samples. 1 kHz sine tone was used to create the waveform. The audio was still modified inside the while-loop as can be seen in Figure 12 and the waveform was not complete.

```
j=0;
while(j<64) // i
{
  //tmp=audio[j]&0xFFF0; // FFF0
  //audio[j]=tmp&packet[j];
  audio[j]=0xFF00+j;
  j++;
}
```

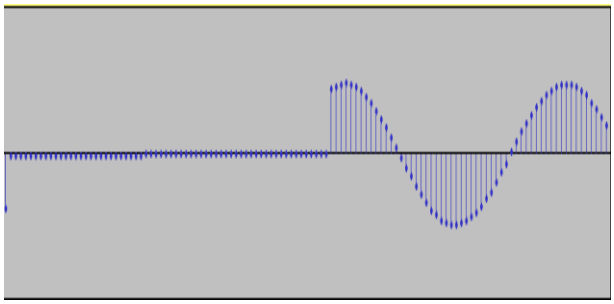

Figure 12. Increasing audio sample's value

Then the *audio[j]=0xFF00+j;* line was commented out after which same audio produced the correct result seen in Figure 13, meaning that audio material was modified in *SetPacket*.

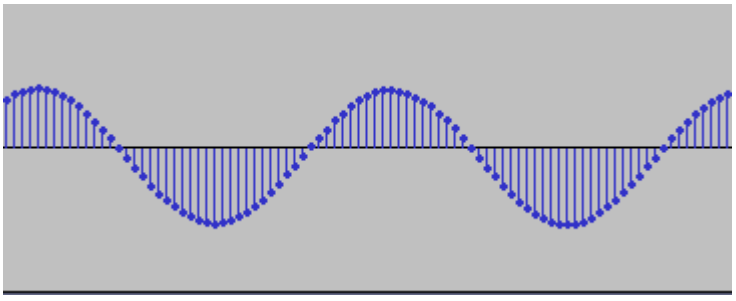

Figure 13. Sine wave

The audio settings were fixed, as 32-bit sample format had been default. Real-time and high-quality conversion settings were also changed to best quality; Figure 14 illustrates this.

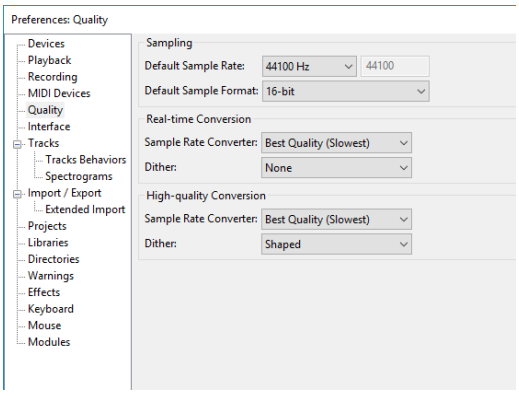

Figure 14. Audacity's quality settings

Yet, it did not matter if audio sample was set to *0x0000* or *0xFFFF* as both would end up in silence. Further test illustrated in Figure 15 revealed that bit depth or -type was incorrect.

```
if(j%2==0) audio[j]=0x00FF;
else audio[j]=0xFF00;
```

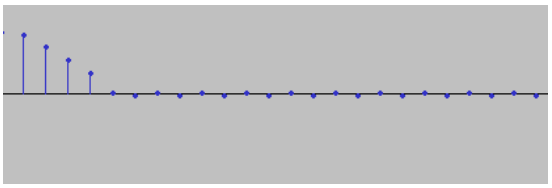

Figure 15. Code and resulting distorted waveform

Audio packets were 16-bit signed integers and used values in binary were *0000 0000 1111 1111* and *1111 1111 0000 0000.* After fixing static values into a dynamic bit shift, the resulting waveform should have been a rising slope. Yet every second sample was zero as illustrated in Figure 16.

```
audio[j]=0x0001<<j;
j++;
```

Figure 16. Test using bit shift

So it seemed obvious there was some kind of type/unit conversion mismatch, as there was approximately 31 high peaks plus the low unmodified values. Using the following code illustrated in Figure 17, there appeared to be wrong type *int* (32-bit) in use instead of *short* (16-bit). The code was updated to output values into serial terminal for easier debugging.

```
j=0;
while(j<32) // i
{
  //audio[j]=tmp&packet[j];
  Serial.print(audio[j],DEC);
  Serial.print(",");
  tmp=1<<j;
  audio[j]=tmp;
  j++;
}
Serial.println(audio[j],DEC);
```

Figure 17. Audio packets to serial terminal

The values received are illustrated in three charts of which each was 32 samples long as shown in Figure 18. 1 kHz sine tone was used.

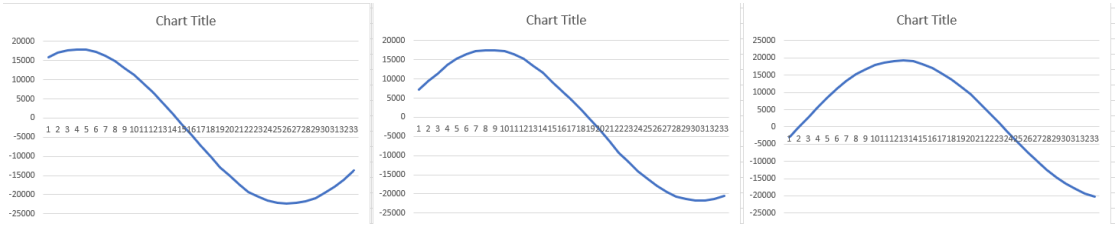

Figure 18. Correct sine wave

The next capture in Audacity appeared just like originally wanted as illustrated in Figure 19. There were no zero samples in between the modified samples as at least two errors in prior steps were corrected:

- Using greater than 16 value in *while(j<16).*
- Change from *short* to *int* for some time during the development.

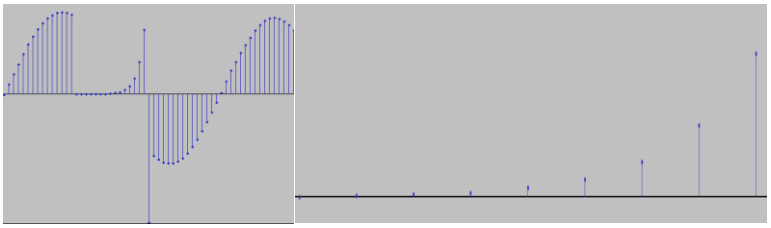

Figure 19. Correct waveform

Packet structure and troubleshooting

The next test involved packet integration into each audio packet's lowest byte: first the message tag bytes, then source byte and lastly a command byte. The lowest byte was erased and overwritten with data as shown in Figure 20.

```
// Passthrough mode message structure, 4 bytes to
packet[i]=tag[i++];
packet[i]=tag[i++];
packet[i++]=src;
packet[i++]=cmd;

j=0;
while(j<i) // i / 16
{
  audio[j]=audio[j]&0xFFF0; // FFF0 or 1111111111110000 to clear lowest byte
  audio[j]=audio[j]|packet[j];
  j++;
}
```

Figure 20. Data inside audio packet

Multiple serial printers existed in *GetPacket* function to find out how long it would execute. During multiple tens seconds of test, it never found tags in the audio. This was assumed to be because of the audio volume level. If this was the case, at least one tag should have been found when volume was modified during a long time scale. Over 29 seconds of sample music was played twice to see if any tags were to be found, but they were not.

It was suspected that for this cause, differential level might work. So, instead of *ZZ* tag use *ZA* instead. This meant changes to the protocol as well, had it worked. Tag could have been lengthened into *ZAZO* or similar. *ZA* is valued *90 65* in decimal, so value with 25 less value in the last character would need to be found as demonstrated in Figure 21. ZA is same as hexadecimal *5A 41* or binary *01011010 01000001*.

```
j=audio[i]&0x000F; // low byte
k=audio[i+1]&0x000F;
if(j-k==25)
{
  Serial.println("Found tag");
  flag++;
}
```

Figure 21. Search for value difference

After printing *SetPacket* bit manipulation in serial terminal it was clear that the root problem was in this function. More correct way to clear bits was found in Arduino library function *bitClear.* It could set bits to *0, as* shown in Figure 22, but debugging messages appeared to display as if sample depth was 32 bits instead of 16. Nevertheless, bits above 16th did not change to *0* during testing.

```
while(j<4) // i / 16
{
  Serial.println(audio[j], BIN);
  while(k<8)
  {
    bitClear(audio[j],k);
    k++;
  }
```

Figure 22. bitClear function

The tags started to be found, as the function was written anew as seen in Figure 23. Yet, it worked only with differential tags. If set into static *ZA* nothing would be found. The found decimal values were *91 & 66, 58 & 33, 88 & 63, 89 & 64, 85 & 60, 191 & 166, 153 & 128* etc. A new test with static values *90 & 65* appeared to work. Yet, Teensy (or its serial communication) was crashing at times and this was supposedly the reason for misinterpretation earlier. The operation was not reliable. When both static and dynamic tags were looked for simultaneously, dynamic were found 97 times during 12 seconds, whereas static were not found at all. Even dynamic values werent consistent, as only 7 of them had correct address information.

```
j=0;
while(j<4) // i / 16
{
  while(k<8)
  {
    bitClear(audio[j],k);
    k++;
  }
  audio[j]=(audio[j] & 0xff00) | packet[j];
  //Serial.println(audio[j], BIN);

  k=0;
  j++;
}
```

Figure 23. Functional bit manipulation

Data was not flowing correctly on the audio waves. *GetPacket* found data at times even without *SetPacket* in use. The next test involved using only header data without audio carrier. This resulted in visible header values in the waveform, wherein they were periodically printed as illustrated in Figure 24.

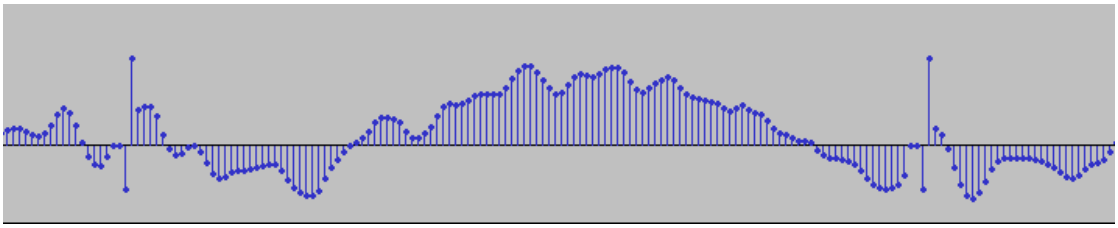

Figure 24. Visible packet header (high peaks)

Faulty unit type

There was also a misunderstanding of *signed short* type. The correct way of setting minimum and maximum value is:

array1[0]=0b0111111111111111; // highest pos val

array1[1]=0b1000000000000000; // lowest neg val

This also affected the data that is embedded, as bits should be flipped if attached into negative values. The highest negative value in binary would be *1111 1111 1111 1111* . So, there is 1-bit for polarity and 15 bits for bit depth.

If the highest positive and negative values could be duplicated from PC to Teensy as well, all data values could be static. A square wave form using maximum and minimum values was played from Audacity into Teensy input and resulting values were read from serial terminal. Values were e.g. 32 276, 32 277, 32 275, -32 275, -32 276 and -32 273 when they should have been two constant values, e.g. 32 767 (high value) and -32 768 (low value). Audacity's *Sample Data Export* printed all sample values into a text file that demonstrated the erroneous values. When value 1 was expected, it was instead e.g. *0.99991, 0.99997, 0.99994* and *0.99997*. The same applied with negative values.

Sample data was modified to hold only -1.00000 and 1.00000 values, read back in with S*ample Data Import* and played to Teensy input. Resulting values in serial terminal were:

> *32269,32269,32272,32274,32275,32275,32278,32278,32282,32282,32285,-32287,-32286,-32290,-32292,-32293,-32294,-32297,-32295,-32299,-32300,-32302,-32305,-32305,-32309,-32308,-32311,-32311,-32313*

The value drifting appeared to be due to export settings, wherein *Channel layout for stereo* must be set into anything else but *L-R on Same Line*. There seems to be some noise on the unused audio channel that affects the total values, when used together. This setting resulted in only two values, that are *0.99997* and *-1.00000*.

Abandoning PC as gateway

But if Teensy reads value 1 from Audacity as something else than a constant value 1, the digital USB communication loses its main purpose of lossless connection and it increases the workload of the original plan. Moreover, streaming lossless audio appeared to be troublesome earlier.

Supposedly digital audio was too noisy or unstable to be used in itself. USB audio was abandoned and other options looked for. There were a few options. Either to use

software audio modem or use another host instead of PC. Software audio modems werent many, but only one supporting Arduino (not Teensy) was found. So the options to replace PC were:

1. Raspberry Pi
2. Moteino with RFM69HW radio
3. Direct serial communication

By connectivity, using Teensy offered not only the longest range, but also greatest variety of additional options. It could connect to *RFM69HW* as well, using SPI. For this reason it was seen as appropriate solution. However, if additional options are not required, USB-serial connection could perhaps be used just as well. Same serial terminal used for debugging could be harnessed for data. Yet this method using *SoftwareSerial* might not be the best option in terms of efficiency as it wastes processor resources (Stoffregen n.d.a) Hardware serial should be used instead (Stoffregen n.d.b). Teensy audio board uses serial communication but *Serial1* should remain free to use. SPI is used by audio library. (Stoffregen 2019).

Serial communication link

Since serial connection between RPi and Teensy did not work using the official RPi documentation, straight serial wiring between two Teensy boards was used instead. Baudrate was set to 1 Mbps. The simplest form of packet was used having only header, footer and data as is illustrated in Figure 25.

```
while(i<ret)
{
  memcpy(&array1[0], rqueue3.readBuffer(), 2 * AUDIO_BLOCK_SAMPLES); // 128 sample array of 16 bit integers
  rqueue3.freeBuffer();

  j=0;

  HWSERIAL.write(0xAA);
  HWSERIAL.write(0xAA);
  HWSERIAL.write(0xAA);
  while(j<AUDIO_BLOCK_SAMPLES)
  {
    HWSERIAL.write(lowByte(array1[j]));
    HWSERIAL.write(highByte(array1[j]));
    j++;
  }
  HWSERIAL.write(0xFF);
  HWSERIAL.write(0xFF);
  HWSERIAL.write(0xFF);


  memcpy(pqueue4.getBuffer(), &array1[0], 2 * AUDIO_BLOCK_SAMPLES);
  pqueue4.playBuffer();
  roundi++;
  i++;
```

Figure 25. Packet assembly process

Yet the receiving end was not working correctly. Audio output line with headphones was beeping continuously and serial communication halted. Changing *AudioMemory* in the Teensy library from 24 to 12 and all to way to 256 made no difference. It is for storing audio while processing other tasks. After analysing the program structure, multiple possible faults were found.

Firstly serial data was sent in pieces and not as a whole. This likely meant that each byte had its own start bit, stop bit etc. This was fixed as shown in Figure 26.

```
txbuffer[j++]=0xAA;
txbuffer[j++]=0xAA;
txbuffer[j++]=0xAA;
while(k<AUDIO_BLOCK_SAMPLES)
{
  txbuffer[j++]=arrayl[k]>>8;
  k++;
}
txbuffer[j++]=0xFF;
txbuffer[j++]=0xFF;
txbuffer[j++]=0xFF;
HWSERIAL.write(txbuffer,j);
```

Figure 26. New packet assembly

Secondly, the baudrate was lowered after decreasing the bit depth into 8. So only the 8 highest bits were read from sample audio data and baud rate was set to 500 000.

Then on no audio was recorded at all but data was zeroes in the main development program, so a baseline version with audio throughput, queues and short array needed to be used as a backup. It supposedly worked as serial monitor was observed. Audio levels affected the traffic and there were multiple values visible. After correcting the receiver to play it through analog out, traffic would only last for a few seconds after restarting the loop. Yet, this was corrected by changing AudioMemory from 24 to 256. Audio quality was barely tolerable and waveform distorted as seen in Figure 27.

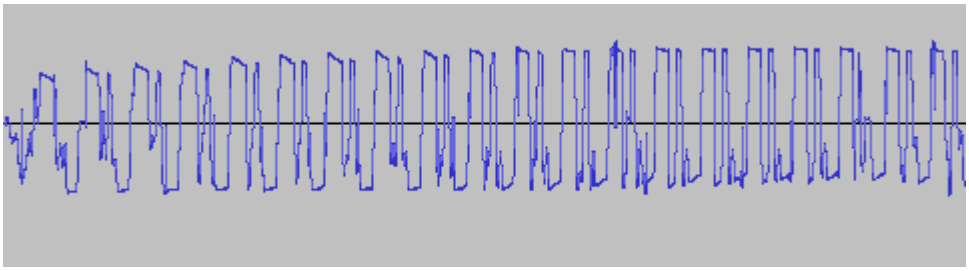

Figure 27. Distorted audio

Debugging became harder, as with 3 bytes of information printed in terminal from each 128 bytes of audio resulted in program halt inside 15 seconds. Other problems in this stage were values ranging from *0* to *FF* in transmitter and receiver, plus noise in receiver. Noise was likely caused by the bit shift that skipped four lowest bits.

Next, known data was sent. It was displayed mostly correctly in the receiving end, but only for a second, after which program and audio output was halt as shown in Figure 28.

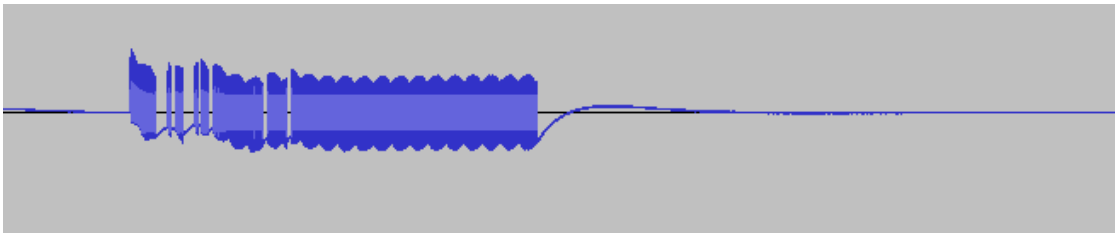

Figure 28. Program halt

Multiple *Serial.print* lines were removed from the loop and program run for over 30 seconds. After removing all *Serial.print* lines from both transmitter and receiver, all the gaps vanished as seen Figure 29 and both the transmitter and receiver run for at least three minutes without failure. Yet, audio would go zero at times.

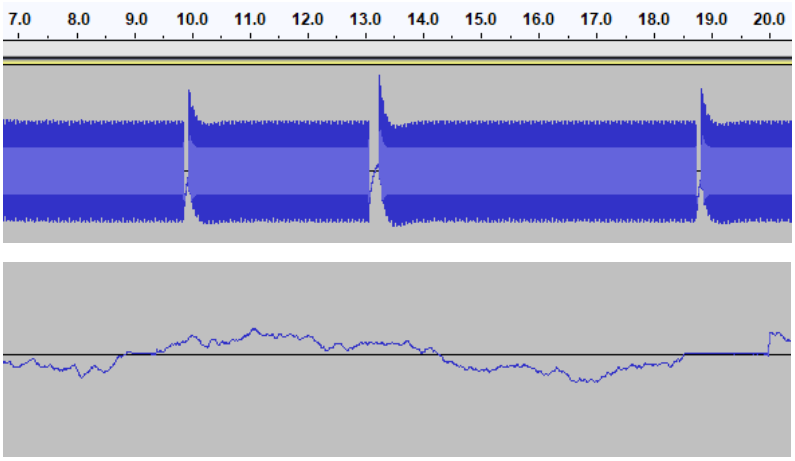

Figure 29. Audio with zero values

Audio samples were still signed integers and had to be changed properly into unsigned. This was done by using constant 255 in division/multiplication and 32768 for offset. In transmitting device negative values were thus lifted above -1 and the resulting 0…65536 value was divided with 255, producing 0…255. In receiver this was done the other way around.

The sound quality remained somewhat unsatisfying when looped back to PC, but for using headphones it was well enough. This was likely due not using most optimal output, as there were two of them.

So, 272 bytes long audio data packets were successfully sent from one Teensy to another using 1 Mbps serial link. It could have been tried in full-duplex, but to keep program more simple, a half-duplex software was continued further.

One-time pad integration

Prequisities for proper OTP functionality include that there must be no *null* or *zero* values in the random data. They caused trouble with SdFat-library functions earlier. In addition, Teensy audio library can not be easily used with SdFat (Stoffregen 2016).

So when using audio, SD-card could not be used like previously planned. Also, this library does not permit multiple open files simultaneously, as stated in function *ReadWrite.ino*. Thus operation would not easily conform into full-duplex.

Teensy's SD-card library was tested in multiple steps while streaming audio into one direction:

- List SD-files in *Setup()* → Pass
- Read 100 bytes from *rdata.001* into serial terminal → Pass
- Read 100 bytes from chosen position into serial terminal → Pass
- Read 256 bytes from chosen position, XOR with audio data and send to receiver…
    - XOR supposedly worked well but there was either offset, quality issue or similar, since audio was not correctly received using constant 256 byte long OTP key

Since 16-bit to 8-bit and the reverse operation can affect the value precision and is hard to debug, 16-bit samples should have been sent instead of 8-bit samples. This demands more bandwidth but baud rate of 1M should still be enough. 16-bit data transmission was tested but did not work correctly, but resulted in distorted audio.

Transmitter's relevant code:

```
// 16-bit transmission
m=32768+array1[k];
txbuffer[j++]=highByte(m); // MSB first
txbuffer[j++]=lowByte(m);
```

Receiver's relevant code:

```
if(i%2==0)
{
        Array3[startp++]=((bytebuff<<8) | incomingByte)-32768;
        Bytebuff=0;
}
else bytebuff=incomingByte;
```

Operation arrangement

The software was reverted to use 8-bit samples again. This time it was obvious that there had been error in the software. Transmitter and receiver should have both handled the byte in the data stream before doing anything else, but instead it was XOR-operated in latter phase instead, in the receiver. After correcting receiver and checking transmitter, audio was correctly XOR-operated with constant value 56. This formed the basis for next steps of using read data from SD-card instead.

Using one byte for XOR from SD-card was successful, but utilising the whole 128-byte data was not functional. It was unclear if this was because of the serial data speed and possible errors. For this reason, authentication needed to be added just to remove some possible sources of error.

Symmetric encryption trial

Not only authentication was introduced but also encryption. *SecretBox* from TweetNaCl -library was used. After setting transmitter and receiver to use the same constant *nonce* and *key*, the audio was unusable a seen in Figure 30. No constant audio was received correctly but it was only pieces of it. Baud rate was fixed from 1 000 000 to 500 000.

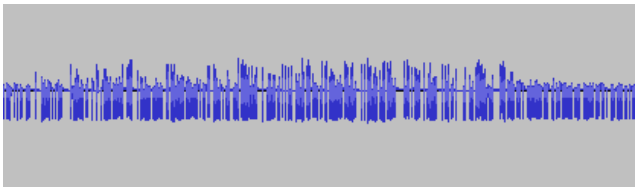

Figure 30. Distorted encrypted audio

Two corrections were made:

- Play audio only if it was successfully dissambled from the packet
    - *BoxZeroBytes* length of data must be equal to 0
    - Message authenticated and unencrypted correctly
- Start playing audio from sample *crypto_secretbox_ZEROBYTES* and not before

The result was better. But the voice was still untolerable and not clear. As if not every sample was played correctly. This could have been because of lines printed in serial terminal or baud rate, as packets had now maximum of 160 bytes of data, compared to 128 before. Removing print lines from serial terminal was one step forward, yet the voice was high pitch and little distorted as shown in Figure 31. It was comprehensible.

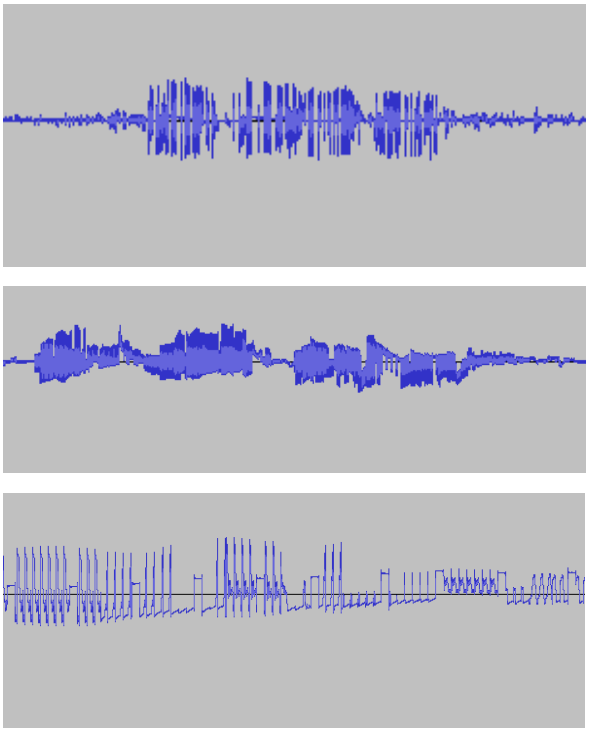

Figure 31. The progress in quality with different magnification levels

The baudrate was the next suspect if it would make a difference. At baudrate 1M the voice became electric and not recognizeable. Further increase to 2M made it even more unclear, as if noises were overlapping with next ones and continuing further than supposed to. Too small baud rate of 250 000 did the same.

Same spoken word was recorded on Audacity from both the transmitted audio (see Figure 32) and Teensy's Digital audio interface (microphone to USB to PC). The differing waveform is visible in frequency analysis as well.

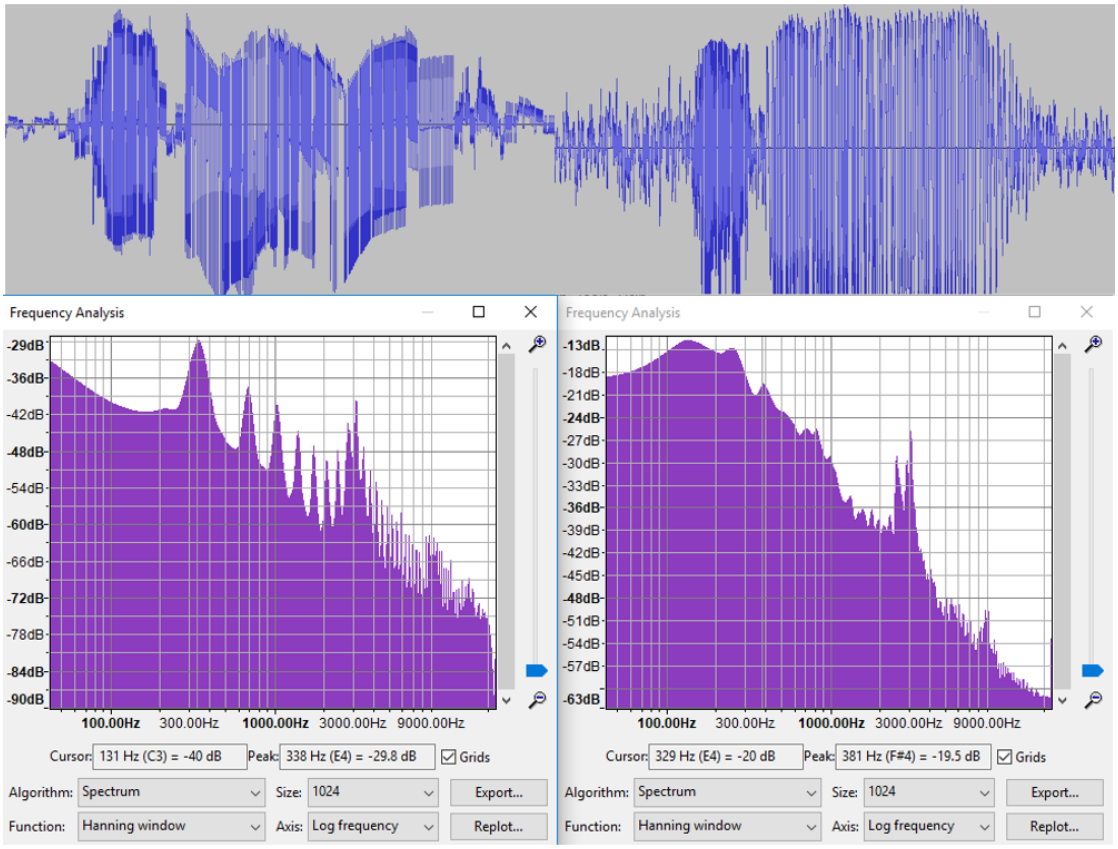

Figure 32. Audio frequency analysis

Something changed so that waveform went back to previous non-continuous shapes. Approximately 90 % of samples were missing from disassembled packets (Figure 33).

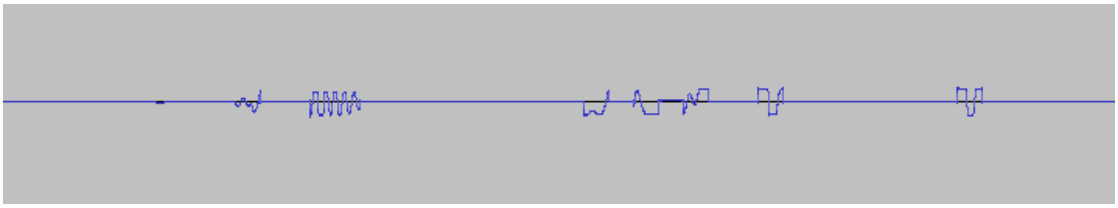

Figure 33. Missed audio

The baud rate affected the waveform again, wherein 500 000 seemed the most promising, as stated earlier. 2M, 1M and 250 000 were not as fit as can been seen in Figure 34.

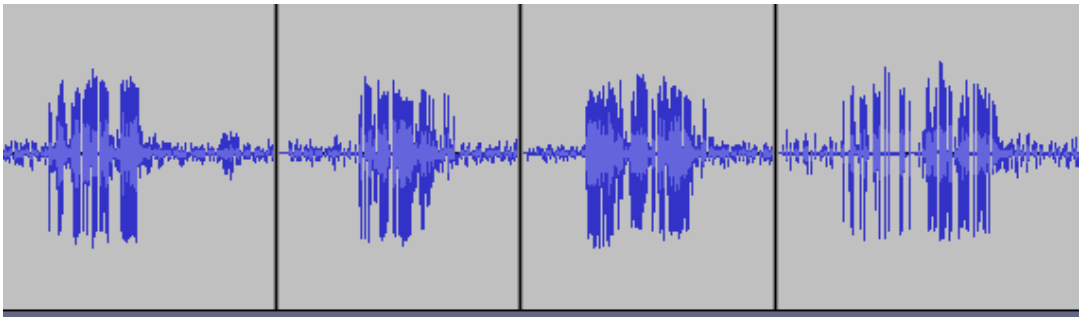

Figure 34. Result of 2M, 1M, 500 000 and 250 000 serial speeds

Since it was supposedly possible that the TweetNaCl library's *SecretBox* was too heavy for Teensy and its 120 MHz clock speed, OneTimeAuth was picked to be used by itself. Symmetric encryption was left out. Software was reverted back to the serial communication version that transfers audio packets with only 16-bit to 8-bit sample modification. The next steps were

- Add and test One-Time authentication
- Use One-Time-Pad

One-time pad with one-time authentication

One-time authentication appeared to be working for 70 % of packets during a 10 minute test; however, using a static packet was 100 % successful. Furthermore, sending the same failed packet 175 652 times would fail each time in the receiver (Figure 35). Whether this was transmitter's fault or receiver's, needed to be found out.

```
byte testbuff[128]= { 112, 111, 112, 112, 112, 112, 112, 112, 112, 112, 111, 112, 111, 111,
111, 111, 112, 112, 113, 113, 114, 114, 114, 115, 116, 116, 116, 117,
117, 116, 116, 116, 115, 116, 116, 116, 116, 116, 116, 117, 117, 117,
117, 117, 118, 118, 117, 118, 118, 117, 117, 117, 117, 118, 118, 118,
117, 117, 117, 116, 115, 115, 115, 115, 115, 115, 115, 115, 115, 115,
115, 116, 116, 117, 117, 117, 117, 117, 117, 117, 117, 117, 116, 116,
116, 115, 115, 115, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116,
117, 117, 117, 117, 117, 117, 118, 118, 117, 118, 118, 118, 119, 119,
118, 119, 118, 118, 119, 119, 119, 119, 120, 120, 121, 122, 122, 122,
123, 123 };
```

Figure 35. One of the failed packets

Somehow the authenticator changed between the processing at transmitter and receiver. While the correct (sent) authenticator for a testbuff was:

107, 3, 246, 61, 77, 148, 97, 177, 255, 97, 180, 182, 212, 32, 136, 243

The receiver handled an authenticator that was:

107, 3, 246, 61, 77, 148, 97, 177, 97, 180, 212, 32, 136, 243, 0

The baud rate was 500 000. Changing it into 9,600 made no difference. Yet, the footer held the same 255 value (differing value in the prior authenticators), giving a clue what could be the cause. There was an incomplete *if*-condition, that did not consider the location of the footer, but only that packet is being disassembled and that value is *FF*. This was fixed to check that footer *FF* values are only considered if found after data length (128) plus authenticator length (16). After this correction, 98.53 % of 420 879 audio packets were correctly authenticated in the receiver during a test run.

Another issue was so called broken samples, as there were not a constant waveform. It could have been due to lost samples, and thus software was written to measure the amount of packets sent and received. Transmitter sent 100 000 packets of which 80 588 were received and 78 967 passed. Nevertheless 79 % were received and valid. Increasing baud rate to 2 000 000 increased received and valid share to 83 %, whereas decreasing it to 250 000 lowered valids to 76 %.

Serial1_tx_buffer_size and *rx_buffer_size* were increased from 64 bytes to 512 bytes in internal library file *serial1.c*. Until now 2M, audiobuffer 256 and serial buffer 512 provided best results. Testing 4 608 000 speed did not work for long before halt. After increasing audio buffer to 512, the quality was even better than 2M. Of all packets received, 99 % were valid. Compiler optimization was set to *Fastest* from the default *Faster*.

There still was an audio quality issue related to microphone input volume and supposedly frequency as well. The quieter and higher frequency the sound, the better the output was.

Introducing 128-byte static pad, used for all audio data into the software pulled passed packet share downwards to 81 % during 550 000 received packets. Yet, the audio was understandable as can be seen in Figure 36.

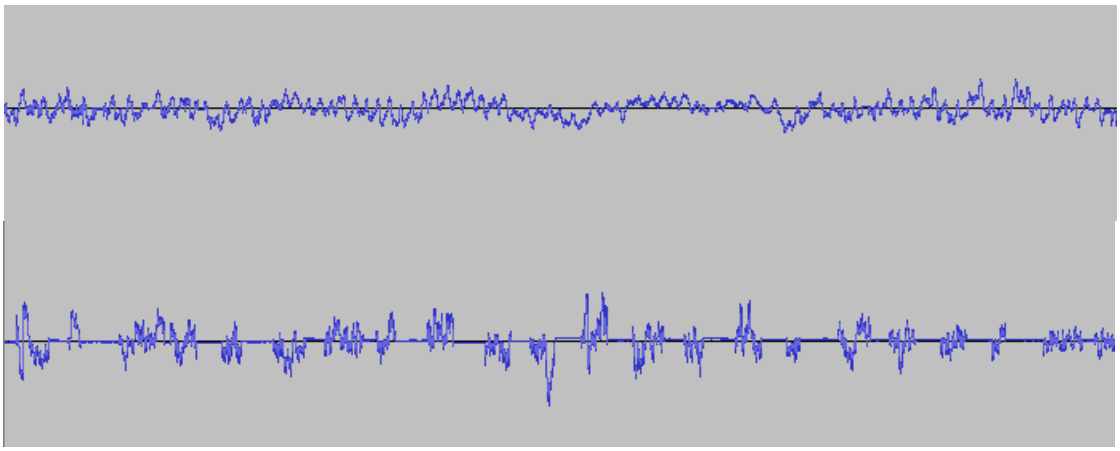

Figure 36. Audio without and with OTP

Next, address information was added to the packet, to point OTP memory location for the receiver. OTP would be read 512 - 1024 bytes at once, when needed using 3 bytes for address and 1 for file. Moreover, receiver packet dissambling should be done only in case of succeeded authentication.

Receiver end would needed to be redone using the old functionality. The debugging appeared to be hard again, as Arduino serial monitor tended to crash if there is too much of data. This is why *random()* function was used to randomize the *Serial.print()* function to run only e.g. once during every 2,000 sent packets.

The receiver was unable to receive data after rewriting the receiver. Some inspections were done. The packet length was now 154 (starting from 0) and it was confirmed that the transmitter sends that amount of data. Yet, receiver did not see that much, if the serial monitor is trusted. Since it is under doubt, simple and fast print sentences were used instead. E.g. instead of printing whole buffer and counting hexadecimal *FF* values when they are in a series. Then print the location of the last one.

Receiver was not handling the footer of the messages in the new version of receiver. Finally a modificated example provided on Arduino forums (Serial Input… 2016)

seemed like best tested serial receiver so far and was used. During a half-duplex test, of 500 000 packets sent, only one was not authenticated correctly. It was supposedly the first one, that was not correct looking data.

Successful OTP encryption

After playing the audio in the receiver during the receiving process, 8 of 115 164 packets failed in authentication. Later test had 2 failures in 822 854 received packets.

Adding a 128-byte static pad proved to be efficient and functional. There was no inconstinency in the audio wafeform as seen in Figure 37, and only 1 packet in 1 949 883 failed authentication.

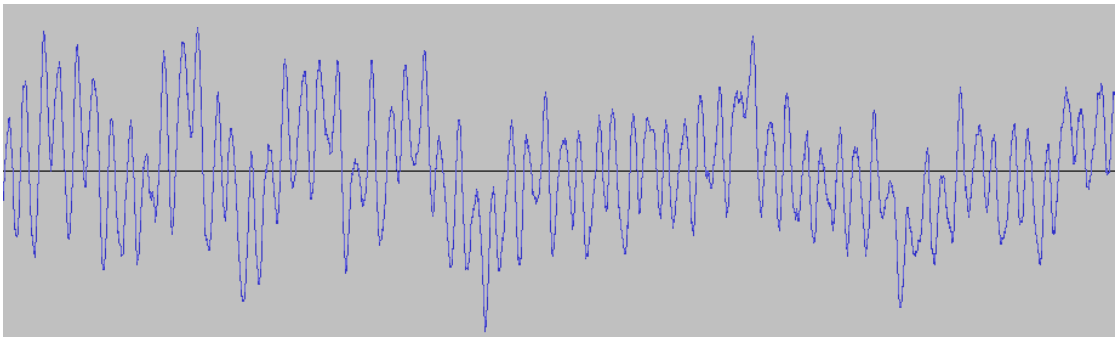

Figure 37. Valid audio that was encrypted and decrypted

Yet, using OTP wherein SD-card is read forward throughout the key data was not working. If same position was read constantly and the *SD.open()* was called only once, it was alright. When position changed, it stopped working. This was due to setting memory address into 4 bytes and its translation back to *unsigned long*. *OR* operation was forgotten from the process, as it should be e.g.

```
Memaddr |= receivedChars[i++]<<24;
Memaddr |= receivedChars[i++]<<16;
```

Instead of

```
Memaddr = receivedChars[i++]<<24;
Memaddr = receivedChars[i++]<<16;
```

Since the fix, one-time pad consisting of 10 MB of data enough for about 30 seconds of audio was run constantly a total of 10 minutes successfully. Button initiation was added by using an example called *change* that uses *Bounce.h.* This way encrypted audio was sent only during a correct button state. Although this simpliest software solution and a cheap button bring the button's *click* noise into the audio. This could be avoided by skipping e.g. next 20 000 samples after state change or perhaps by using *retrigger* example to make up a more sophisticated button interpreter.

5 Result

5.1 Random data

True random data forms basis for OTP. It was successfully created using a software radio. After that *rngtest* was used to run FIPS 140-2 tests (Rodríquez et al., 2017). It produced 417 401 of FIPS-140-2 successes and 333 failures. Failures were about 0.798 % of the whole test amount, wherein 20 kilobits were tested at a time. This meant 832.5 kilobytes of failed data. In worst case scenario, it is all in one spot. Using 16 bits, 44.1 kHz and 1 GB of data, approximately 90.5 seconds of plain audio could appear to be plain open. Yet, 11 247 seconds would remain secure as seen in Table 5.

Table 5. Rngtest results

File	Frequency	Sample rate	FIPS 140-2 successes	FIPS 140-2 failures	Failures of all samples
rdata.bin	135.5 MHz	2.4 MS/s	417,401	333	0.792 %
rdata1.bin	177.5 MHz	2.4 MS/s	400,313	312	0.788 %
rdata2.bin	185.5 MHz	2.4 MS/s	400,843	316	0.788 %
prdata.bin	-	-	0	436,207	100 %

In addition, random number generator testing tool *dieharder* (Alani 2010) was used for a second test by executing *dieharder -a -v rdata.bin* producing only *PASSED* results, except for *sts_serial 16* where result was WEAK. Test reliability was marked as *Good*, so it was supposedly still a trustworthy test. *Dieharder* was very time consuming on Pentium 4 CPU taking 1 hour and 30 minutes. The earlier *rngtest* used only 5 minutes for same amount of data.

For comparison, pseudorandom 1.1 GB data file was created with *openssl rand -out sample.txt -base64 $((2**30 * 3/4))* but that did not pass *rngtest*, whereas *dieharder* left things to consider, as it passed every test except *diehard_bistream* and *rgb_permutations*.

Random data could as well be tested by compressing it. Real random data does not compress. (Schneier 1996, 353).

5.2 One-time pad

Encryption was successfully used. Voice was encrypted and decrypted using one-time pad and the functionality was demonstrated in both directions using encryption like specified (Chapter 2.1). Though the platform is not the most optimal for long term and wide spread usage. With some changes in software, same key data would not be used again, but its erasure is not as easy. The program flow is shown in Figure 38.

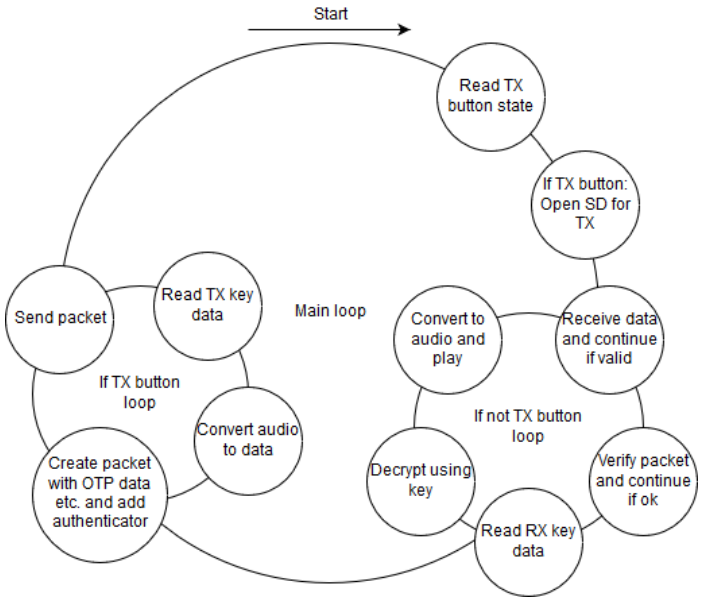

Figure 38. Program flow

6 Discussion

6.1 Research questions

Research questions were following:

- Can a microcontroller based device encrypt speech with one-time pad?
- Can the device be portable?
- Can the device operate between audio and USB-interface?
- Can the device be open in hardware and software?

Such a device can be created and is increasingly possible. Electronic devices are getting smaller and more processing power fits into smaller enclosure. Teensy 3.5 development board was chosen to be used with the compatible audio board. This was the easiest option found for this development project, as all libraries except NaCl were provided by the developer and the community. Yet, it was not the most open development platform. OTP usage was successfully demonstrated, and code is available for test and development (Appendix 1). More was expected, however, as it was assumed there would be less setbacks and more swift progress. Especially leaving USB-audio out hurt the usability and connectivity.

Device can be portable, and it is possible to design it into same form factor that dongles have. It could be carried in a pocket just as easy as mechanical keys and this could make key distribution easier, should there be a way to connect two devices. Wireless transfer would be most convenient but not safe. Light Fidelity or equivalent is preferable for such a purpose. Pocketable size makes it vulnerable to theft and other malicious acts and creates a need for proper physical security, that was not considered during the project.

USB interface was available but not used for audio but programming. USB audio did not function like planned. The Teensy boards used serial communication with 10 cm distance in between to communicate using both one-time pad and one-time authentication. Data transmitted in the serial wiring had OTP encrypted audio inside and it was correctly unencrypted. Audio bit depth was 8 bits and frequency supposedly 44,100 kHz, yet speech was well understandable.

The device can be fully open, but it requires much more resources for software and hardware development. Software libraries would need to be created or converted to new platform. Their licenses could be restrictive. Moreover, open hardware is limited in optional accessories increasing the workload on setup. Lastly, openness does not guarantee security (Chapter 3.4).

6.2 Reliability

The most reliable part besides OTP was likely the random data, that was tested using two methods (Chapter 5). It is also the most crucial part. If all random data bytes were the same, the resulting audio would be plain open. Nevertheless, just 0.8 % of created random data failed the randomness tests, proving it was of good quality. While the creation is easy, managing the key material is not. Its integrity should be observed, and it should not fall into wrong hands.

There were limitations that could be rightfully used to argue that it was not real OTP except during the first run. This is because same key data was used all over again, to make debugging easier. This was also because there was not a valid way to erase old data. Nevertheless, the OTP encrypted data (before next loop cycles) was and remains secure in transit. The weakness is not as much in the gateway as in the end point devices and especially the random data.

Same result can be achieved by using the same setup with the available project documentation, same hardware and software versions. The result is not applicable to all microcontroller development boards without changes. Using e.g. a slower processor might not be enough for a real time audio processing. Also, not all microcontroller boards have audio boards available nor do all of them have extensive software libraries.

6.3 Limits

The audio and serial traffic proved to be troublesome. The project was spread over multiple years breaking a smooth and constant development cycle. Project development setup required two host devices, played mostly by Linux and Windows PC. A software is easier to move and change than the used test setup.

Threat model was very limited. A comprehensive one was started but never finished, as it became too heavy. Just listing the assets used multiple pages. The documented threat model is very simple with the idea of including some threat model with an idea of what is at stake. Even in its simple form, it outlined very important factors of using secure and trusted hardware parts, well-coded software, having good instructions and being small size.

Hardware development was not fluent. For a moment, it seemed as if audio board needs to be left out and Teensy board's own ADC and DAC used instead. This was tried and would have simplified the hardware part, but the quality was too bad. In the end, the audio board was successfully used.

The communication gateway trials took much longer than expected. Unexpected results and errors in USB-audio changed the plan. Also, too complicated solutions were tried, when serial communication option was there from the beginning.

The programming process should have been planned better. Perhaps half of time and errors were involved with the software process and especially audio data handling. Moreover, the software development in Arduino environment brought unexpected problems. It seemed better to accept them and adapt instead of working on them possibly for hours in vain. Programming could have been greatly improved by a better development environment wherein debugging is advanced.

Maximum reasonable amount of openness was desired from the developed system (Chapter 1.1). This was achieved from the software side apart from the Teensy bootloader that is not open source. Source codes are available for the Teensy and Arduino software in Appendix 1. The *32-bit 120 MHz ARM Cortex-M4* processor used in Teensy 3.5 is not open hardware, wherein one could see how the processor is built. Same likely applies to other hardware chips as well, but they are not as essential.

Symmetric authenticated encryption was desired, but not an objective. It was left out in the development when considered supposedly too heavy to process for the microcontroller. It could probably be applied in the latest software version, as many bugs were fixed after trying it the last time. OTP is in principle a very light encryption method, but only for the very core (XOR operation). Many features surrounding and

required by OTP like especially SD-card reading (in this project) are not as light. Symmetric encryption might prove to be faster and more usable, as it does not require SD-card. It is ready to be applied with slight code changes.

6.4 Usability and future development

Resulting project can be a discussion opener and a basis for new projects. There was not similar project available for Teensy environment. It is likely that the project cannot be used for anything serious concerning OTP, because of the limitations and the platform. Yet, using symmetric encryption instead, might prove to be a viable option and is something to test, if the code is used. OTP could still be enabled by button or by firmware update. In this case, the device would rival Jack Pair.

So far, the audio traffic had been half-duplex, but as full-duplex was desired and both boards have physical microphones, they should be further away from each other than 10 cm. This increase of distance between the end points was left out to save time.

It would be sensible to remove the used random data. Yet, flash memory's data cannot be overwritten in-place (Reardon et al., 3-8). A non-flash solution is needed, when proper key erasure is desired. This is the main hardware limitation along with EEPROM, why platform is not ideal for OTP. Moreover, it should also be inspected whether key data has been used already by writing e.g. five nulls into the beginning of the file, since there should be no nulls in the random data (chapter 4.3.1).

Teensy board's EEPROM memory could be used to keep track of current key file location between boots. EEPROM size is 4096 bytes. Agreeing to use only 5 bytes for file name (e.g. *01.rd*) and 5 bytes for file location would make enough room for over 400 files, or 200 individual connections/contacts, as each requires one file for reception and another for transmission. But EEPROM is limited in its write cycles, that are only 100 000 in this case (Stoffregen, n.d.c). In the current software version, the EEPROM write process would need to start each time when transmit ends, to save current file location, as there is no information if the power will go down soon. This creates a need for proper shutdown sequence or another memory.

The serial wiring connection is not fit for demands of today by itself. It can and should be bridged over e.g. Ethernet, WLAN or Bluetooth.

There is perhaps an option of using USB serial connection instead of hardware serial port (Stoffregen n.d.d; Stoffregen n.d.e). This would require more effort from the host-device software than what audio would.

Ultimately, it could be a good idea to transfer the project into another platform, such as Olimex and/or SiFive perhaps. A platform with easier adaptability into a commercial product is desirable. For such product a number of subjects should be considered in addition to the existing ones. They are e.g. integrity of messages and keys, authentication of devices, key generators and operators, secret key integrity and distribution (Borowski & Leśniewicz 2012, 3).

6.5 Influence

There is an opening for a personal portable encryption device. Any device can and should not be trusted. Using a trusted hardware encryption, however, allows people to bypass many concerns with today's computers and mobile phones. Such a device should be open design from an open developer, to raise trust. It should be audited too and is unlikely to become widespread unless embedded or easily pluggable to a host device, such as a mobile phone.

There is a reason to advance the old and secure OTP that is not much used nowadays in the public at least. It still is and will remain as the safest encryption method in the world and age to come when properly applied.

One does not need to be a professional to understand the functionality of OTP. Even a child can comprehend and use it. The idea is simple, but applying it is harder. People tend to use many systems and encryption algorithms that they do not understand, dismissing the simple one they would. OTP is often undervalued for real world usage supposedly mostly because of its downsides. Instead, there should be increased focus on how to make it usable and available. One can hope this development project brings some discussion around OTP concerning today's applications and communication system security.

References

2018 Report on foreign policy-based export controls. U.S. Department of Commerce Bureau of Industry and Security. Accessed 7 November 2018. Retrieved from https://www.bis.doc.gov/index.php/documents/pdfs/2186-bis-foregin-policy-report-2018/file

Alani, M.M. 2010. Testing Randomness in Ciphertext of Block-Ciphers Using DieHard Tests. *IJCSNS International Journal of Computer Science and Network Security, 10*. Accessed 4 May 2019. Retrieved from https://pdfs.semanticscholar.org/05a2/ddf1c73a958d39ea1719ac480927af4739df.pdf

Bahamdain, S.S. 2015. *Open Source Software (OSS) Quality Assurance: A Survey Paper.* Elsevier B.V. Accessed 11 June 2018. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877050915017172

Bellovin, S.M. 2011. *Frank Miller: Inventor of the One-Time pad*. Accessed 19 October 2017. Retrieved from http://www.cs.columbia.edu/~CS4HS/talks/FrankMillerOneTimePad.pdf

Benson, R.L. 2001. *The Venona Story*. Accessed 19 October 2017. Retrieved from https://www.nsa.gov/about/cryptologic-heritage/historical-figures-publications/publications/coldwar/assets/files/venona_story.pdf

Bernstein D.J. 2005. *The Poly1305-AES Message-Authentication Code*. Springer. Accessed 6 May 2019. Retrieved from https://cr.yp.to/mac/poly1305-20050329.pdf

Bernstein, D.J., Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsers, S. 2017. *TweetNaCl: a crypto library in 100 tweets*. Accessed 6 May 2019. Retrieved from https://tweetnacl.cr.yp.to/

Bernstein, D.J., Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsers, S. 2019a. *Secret-key single-message authentication: crypto_onetimeauth*. Accessed 6 May 2019. Retrieved from https://nacl.cr.yp.to/onetimeauth.html

Bernstein, D.J., Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsers, S. 2019b. *Secret-key authenticated encryption: crypto_secretbox.* Accessed 6 May 2019. Retrieved from https://nacl.cr.yp.to/secretbox.html

Borowski, M., Lesniewicz, M. 2012. Modern usage of “old” one-time pad. *Communications and Information Systems Conference (MCC), 2012 Military.* Accessed 1 June 2019. Retrieved from https://www.researchgate.net/publication/261056835_Modern_usage_of_old_one-time_pad

Chandrakar, S., Tiwari, S., Jain, B.S. 2014. An Innovative Approach for Implementation of One-Time Pads*. International Journal of Computer Applications, 89,* 35-37. Accessed 3 May 2017. Retrieved from http://research.ijcaonline.org/volume89/number13/pxc3894579.pdf

Chang, J. 2018. *JackPair: secure your voice phone calls against wiretapping*. Accessed 30 November 2018. Retrieved from https://www.kickstarter.com/projects/620001568/jackpair-safeguard-your-phone-conversation/description

Clarke, R., Dowring, D., & Nash, R. 2005. *Is Open Source Software More Secure?* Accessed 11 June 2018. Retrieved from https://courses.cs.washington.edu/courses/csep590/05au/whitepaper_turnin/oss(10).pdf

COM/2017/003 proposal. *Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL concerning the respect for private life and the protection of personal data in electronic communications and repealing Directive 2002/58/EC (Regulation on Privacy and Electronic Communications).* 2017. European Commission. Accessed 7 November 2018. Retrieved from https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:52017PC0010&from=EN

Devipriya, M., Sasikala, G. 2015. A New Technique for One Time Pad Security Scheme with Complement Method. *International Journal of Advanced Research in Computer Science and Software Engineering 5,* 220-223. Accessed 3 May 2017. Retrieved from https://www.ijarcsse.com/docs/papers/Volume_5/6_June2015/V5I6-0106.pdf

Fournaris, A.P., Fraile, L.P., Koufopavlou, O. 2017. Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: a Survey of Potent Microarchitectural Attacks, 6, 52. *MDPI Electronics journal*. Accessed 10 November 2018. Retrieved from https://www.mdpi.com/2079-9292/6/3/52/pdf

Halunen, K., Suomalainen, J., Lehtonen, S., Karinsalo, A., Vallivaara, V. 2018. *Sähköisen viestinnän salaus- ja suojausmenetelmät* [*Encryption and protection methods in electronic communications*]. Ministry of Transport and Communications, 7 March 2018. Accessed 15 June 2018. Retrieved from http://julkaisut.valtioneuvosto.fi/bitstream/handle/10024/160614/LVM_02_2018_Sahkoisen_viestinnan%20salaus_ja_suojaus.pdf?sequence=1&isAllowed=y

Houthen, L.V. 2017. *Crypto 101*. Accessed 3 November 2018. Retrieved from https://www.crypto101.io/Crypto101.pdf

Jin, Y. 2015. Introduction to Hardware Security, 4, 763-784. *MDPI Electronics journal, 4*, 763-784. Accessed 8 November 2018. Retrieved from http://jin.ece.ufl.edu/papers/Electronics15.pdf

Krawczyk, H. 2001. *The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?).* Accessed 3 November 2018. Retrieved from https://iacr.org/archive/crypto2001/21390309.pdf

List of Dual-Use Goods and Technologies and Munitions List. 2017. Accessed 7 November 2018. Retrieved from https://www.wassenaar.org/app/uploads/2017/12/2017_List_of_DU_Goods_and_Technologies_and_Munitions_List-1.pdf

Mishra, M., Potnis, A., Dwivedy, P., Meena, S.K. 2017. Software defined radio based receivers using RTL — SDR: A review. *2017 International Conference on Recent*

Innovations in Signal processing and Embedded Systems (RISE), 62-65. Accessed 4 May 2019. Retrieved from https://ieeexplore.ieee.org/document/8378125

On Security Issues of QKD. Accessed 19 October 2017. Retrieved from http://www.uqcc.org/images/towards.pdf

One Time Pad Encryption. Accessed 19 October 2017. Retrieved from http://www.cryptomuseum.com/manuf/mils/files/mils_otp_proof.pdf

Papp, D., Ma, Z., Buttyan, L. 2015. *Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy*. 2015 Thirteenth Annual Conference on Privacy, Security and Trust (PST). Accessed 2.2.2019. Retrieved from http://www.cse.psu.edu/~pdm12/cse597g-f15/readings/cse597g-embedded_systems.pdf

Position. 2019. Accessed 6 May 2019. Retrieved from https://www.arduino.cc/en/Reference/FilePosition

Privacy. n.d. Electronic Frontier Foundation. Accessed 2 May 2017. Retrieved from https://www.eff.org/issues/privacy

QuintessenceLabs delivers fast, commercial rate True Random Numbers at 1Gb/sec. 2017. Accessed 19 October 2017. Retrieved from https://www.quintessencelabs.com/quantum-cybersecurity/?tab=random-number

Reardon, J., Basin, D., Capkun, S. n.d. *On Secure Data Deletion*. Institute of Information Security, ETH Zurich. Accessed 27 December 2018. Retrieved from https://www.inf.ethz.ch/personal/basin/pubs/onsecuredeletion.pdf

Rodríquez, R.J., Garcia-Escartin, J.C., Sánchez-Ballabriga, V. 2017. Security Assessment of the Spanish Contactless Identity Card. *IET Information Security*, *11*, 386-393. Accessed 4 May 2019. Retrieved from https://uvadoc.uva.es/bitstream/10324/25915/1/SecurityAssessmentDNIe.pdf

Saper, N. 2013. International Cryptography Regulation and the Global Information Economy, 11. *Northwestern Journal of Technology and Intellectual Property*. Accessed 7 November 2018. Retrieved from https://scholarlycommons.law.northwestern.edu/cgi/viewcontent.cgi?article=1205&context=njtip

Schneier, B. 1996. *Applied Cryptography*. John Wiley & Sons.

Schneier, B. 1999. *Crypto-Gram February 15, 1999*. Accessed 30 November 2018. Retrieved from https://www.schneier.com/crypto-gram/archives/1999/0215.html

Schneier, B. *The Importance of Strong Encryption to Security*. 2016. Accessed 2 May 2017. Retrieved from https://www.schneier.com/blog/archives/2016/02/the_importance_.html

Schneier, B. *The Value of Encryption*. 2016. Accessed 2 May 2017. Retrieved from https://www.schneier.com/essays/archives/2016/04/the_value_of_encrypt.html

SD Library. Accessed 6 May 2019. Retrieved from https://www.arduino.cc/en/Reference/SD

Serial Input Basics – updated. 2016. Accessed 8 May 2019. Retrieved from https://forum.arduino.cc/index.php?topic=396450

Shull, F. 2016. *Cyber Threat Modeling: An Evaluation of Three Methods.* Carnegie Mellon University. Accessed 20 November 2018. Retrieved from https://insights.sei.cmu.edu/sei_blog/2016/11/cyber-threat-modeling-an-evaluation-of-three-methods.html

Silver, A. 2017. *China's 'future-proof' crypto: We talk to firm behind crazy quantum key distribution network.* Accessed 19 October 2017. Retrieved from https://www.theregister.co.uk/2017/07/19/cryptography_quantum_key_distribution_china/

Stoffregen, P. 2016. *Audio library and SdFat Library.* Accessed 8 May 2019. Retrieved from https://forum.pjrc.com/threads/34327-Audio-library-and-SdFat-Library

Stoffregen, P. 2018. *Teensy 3.2 Writing to SD-Card - how to increase speed?* Accessed 6 May 2019. Retrieved from https://forum.pjrc.com/threads/50110-Teensy-3-2-Writing-to-SD-Card-how-to-increase-speed

Stoffregen, P. 2019. *Audio Adaptor Board for Teensy 3.0 - 3.6.* Accessed 8 May 2019. Retrieved from https://www.pjrc.com/store/teensy3_audio.html

Stoffregen, P. N.d.a. *NewSoftSerial Library.* Accessed 8 May 2019. Retrieved from https://forum.pjrc.com/threads/50110-Teensy-3-2-Writing-to-SD-Card-how-to-increase-speed

Stoffregen, P. N.d.b.. *Using the Hardware Serial Ports.* Accessed 8 May 2019. Retrieved from https://www.pjrc.com/teensy/td_uart.html

Stoffregen, P. N.d.c. *EEPROM Library*. Accessed 8 May 2019. Retrieved from https://www.pjrc.com/teensy/td_libs_EEPROM.html

Stoffregen, P. N.d.d. *USB Virtual Serial Receive Speed*. Accessed 8 May 2019. Retrieved from https://www.pjrc.com/teensy/benchmark_usb_serial_receive.html

Stoffregen, P. N.d.e. *USB: Virtual Serial Port*. Accessed 8 May 2019. Retrieved from https://www.pjrc.com/teensy/usb_serial.html

Stoffregen, P. N.d.f. *Download Teensyduino, Version 1.46.* Accessed 9 May 2019. Retrieved from www.pjrc.com/teensy/td_download.html

Suomen perustuslaki [Finnish constitutional law]. 2018. Accessed 4 November 2018. Retrieved from https://www.finlex.fi/fi/laki/ajantasa/1999/19990731#L2P10

Tehranipoor, M., Koushanfar, F. 2010. A Survey of Hardware Trojan Taxonomy and Detection, 27, 10-15. *IEEE Design & Test for Computers, 27,* 10-15. Accessed 8 November 2018. Retrieved from https://ieeexplore-ieee-org.ezproxy.utu.fi/document/5406669

Tietoyhteiskuntakaari 917/2014 [*Information Society Code*]. Accessed 15 June 2018. Retrieved from https://www.finlex.fi/fi/laki/alkup/2014/20140917

Tipton, H.F., Krause, M. 2007. *Information Security Management Handbook, Sixth Edition, Volume 1*. Auerbach Publications. Accessed 20 November 2017. Retrieved from http://library.books24x7.com/

Tiwari, M., Tiwari, M. 2012. Voice – How humans communicate?, 3, 3-11. *Journal of Natural Science, Biology and Medicine.* Accessed 13 November 2018. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3361774/#

Toshiba's QCCS. 2015. Accessed 19 October 2017. Retrieved from http://www.tqccs.com/cl/tech/qccs/en/about/

Warren, P. 2018. *rtl-entropy*. Accessed 16 November 2018. Retrieved from https://github.com/pwarren/rtl-entropy

Yang, K., Hicks, M., Dong, Q., Austin, T., Sylvester, D. 2016. *A2: Analog Malicious Hardware*. Department of Electrical Engineering and Computer Science University of Michigan. Accessed November 8, 2018. Retrieved from https://web.eecs.umich.edu/~taustin/papers/OAKLAND16-a2attack.pdf

Appendices

Appendix 1. The source code

```
/*
Functional OTP demo for Arduino software using two Teensy boards, two audio
shields and serial wiring between them. Microphones are soldered to both audio
boards and a button is used to initiate the transmission, one per board
*/

#include <Audio.h>
#include <Bounce.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>

// SecretBox from https://tweetnacl.cr.yp.to/index.html
#define crypto_secretbox_xsalsa20poly1305_tweet_KEYBYTES 32
#define crypto_secretbox_xsalsa20poly1305_tweet_NONCEBYTES 24
#define crypto_secretbox_xsalsa20poly1305_tweet_ZEROBYTES 32
#define crypto_secretbox_xsalsa20poly1305_tweet_BOXZEROBYTES 16

#define crypto_secretbox_xsalsa20poly1305_KEYBYTES
crypto_secretbox_xsalsa20poly1305_tweet_KEYBYTES
#define crypto_secretbox_xsalsa20poly1305_NONCEBYTES
crypto_secretbox_xsalsa20poly1305_tweet_NONCEBYTES
#define crypto_secretbox_xsalsa20poly1305_ZEROBYTES
crypto_secretbox_xsalsa20poly1305_tweet_ZEROBYTES
#define crypto_secretbox_xsalsa20poly1305_BOXZEROBYTES
crypto_secretbox_xsalsa20poly1305_tweet_BOXZEROBYTES

#define crypto_secretbox_PRIMITIVE "xsalsa20poly1305"
#define crypto_secretbox crypto_secretbox_xsalsa20poly1305
#define crypto_secretbox_open crypto_secretbox_xsalsa20poly1305_open
#define crypto_secretbox_KEYBYTES crypto_secretbox_xsalsa20poly1305_KEYBYTES
#define crypto_secretbox_NONCEBYTES
crypto_secretbox_xsalsa20poly1305_NONCEBYTES
#define crypto_secretbox_ZEROBYTES
crypto_secretbox_xsalsa20poly1305_ZEROBYTES
#define crypto_secretbox_BOXZEROBYTES
crypto_secretbox_xsalsa20poly1305_BOXZEROBYTES
#define crypto_secretbox_IMPLEMENTATION
crypto_secretbox_xsalsa20poly1305_IMPLEMENTATION
#define crypto_secretbox_VERSION crypto_secretbox_xsalsa20poly1305_VERSION

#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void
```

```
AudioControlSGTL5000     sgtl5000_1;                 //xy=302,184

//AudioInputUSB         usb1;                       //xy=200,69  (must set Tools > USB Type
to Audio)
AudioOutputI2S          i2s1;                       //xy=365,94   // headphones
//AudioRecordQueue       rqueue1;
AudioPlayQueue          pqueue2;

//AudioConnection        patchCord1(usb1, 0, rqueue1, 0);
AudioConnection         patchCord2(pqueue2, 0, i2s1, 0);  // SERIAL RX to PQUEUE2
to HEADPHONES

AudioInputI2S           i2s2;                       //xy=105,63
//AudioOutputUSB         usb2;
AudioRecordQueue        rqueue3;
//AudioPlayQueue         pqueue4;

AudioConnection         patchCord3(i2s2, 0, rqueue3, 0);  // MIC I2S to RQUEUE3 to
SERIAL TX
//AudioConnection        patchCord4(pqueue4, 0, usb2, 0);

bool debug = true;
bool debug1 = true;

#define HWSERIAL Serial1

const int buttonPin   = 37;
const int myInput     = AUDIO_INPUT_MIC;
const int ledPin      = 13;
int ledValue          = LOW;

Bounce bouncer = Bounce( buttonPin, 5 );

int buttonState   = 0;

int ret=0;
int i=0;
int j=0;
int k=0;
int l=0;
int au=0;
int counter1=0;
int counter2=0;

unsigned long file_size = 0;
unsigned long file_size1 = 0;

unsigned long long m = 0;

short array1[AUDIO_BLOCK_SAMPLES];
short array2[AUDIO_BLOCK_SAMPLES];
short array3[128]={0};
```

```
byte sdbuffer_rx[512]={0}; // 512
byte sdbuffer_tx[512]={0}; // 512

// sd-card functionality from listfiles-example
File myFile;
File myFile2;

const int chipSelect            = BUILTIN_SDCARD;
byte file                       = 0;
unsigned long file_location     = 0;
unsigned long last_file_location  = 0;

// serial receiver
const byte numChars = 151;
byte receivedChars[numChars];
boolean newData = false;

// serial transmitter
byte txbuffer[256]={0};

// from TweetNaCl (READ documentation before using @ https://nacl.cr.yp.to/ )
typedef unsigned char u8;
typedef unsigned long u32;
typedef unsigned long long u64;
typedef long long i64;
typedef i64 gf[16];
extern void randombytes(u8 *,u64);
static u32 L32(u32 x,int c) { return (x << c) | ((x&0xffffffff) >> (32 - c)); }
// static const u8 sigma[16] = "expand 32-byte k";
static const u8 sigma[16] = {'e','x','p','a','n','d',' ','3','2','-','b','y','t','e',' ','k'};


unsigned char c[512]   =   {0};
unsigned char key1[32]         =   {43, 1, 25, 21, 56, 7, 21, 56, 7, 78, 9, 9, 56, 7, 78, 9,
                            54, 67, 13, 89, 244, 65, 73, 213, 51, 42, 1, 5, 12, 12, 56, 212
                            };
unsigned char n[crypto_secretbox_NONCEBYTES]        =   {5, 4, 6, 7, 1,
                           3, 8, 9, 24, 4,
                            6, 5, 7, 3, 21,
                            6, 5, 7, 3, 21,
                             6, 5, 7, 3};
unsigned char message[128]     =   {'s', 'e', 'c', 'r', 'e', 't', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e'};
unsigned long long mlen1        =   14;
unsigned char authenticator[32] =   {0};
unsigned char key[crypto_secretbox_KEYBYTES] = {43, 1, 25, 21, 56, 7, 21, 56,
                                    7, 78, 9, 9, 56, 7, 78, 9,
                                    54, 67, 13, 89, 244, 65, 73, 213,
                                    51, 42, 1, 5, 12, 12, 56, 212};
unsigned char msg[] = {'s', 'e', 'c', 'r', 'e', 't', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e',
                's', 'e', 'c', 'r', 'e', 't', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e',
                's', 'e', 'c', 'r', 'e', 't', ' ', 'm', 'e', 's', 's', 'a', 'g', 'e'};
unsigned long long mlen         =   42;
unsigned char a[16] = {0};
```

```
#define BUFFER_SIZE (256 + \
                crypto_secretbox_ZEROBYTES + \
                crypto_secretbox_BOXZEROBYTES)

static u32 ld32(const u8 *x)
{
  u32 u = x[3];
  u = (u<<8)|x[2];
  u = (u<<8)|x[1];
  return (u<<8)|x[0];
}

sv st32(u8 *x,u32 u)
{
  int i;
  FOR(i,4) { x[i] = u; u >>= 8; }
}

sv add1305(u32 *h,const u32 *c)
{
  u32 j,u = 0;
  FOR(j,17) {
    u += h[j] + c[j];
    h[j] = u & 255;
    u >>= 8;
  }
}
static const u32 minusp[17] = {
  5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 252
} ;

sv core(u8 *out,const u8 *in,const u8 *k,const u8 *c,int h)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];

  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
```

```
    t[0] ^= L32(t[3]+t[2],18);
    FOR(m,4) w[4*j+(j+m)%4] = t[m];
   }
   FOR(m,16) x[m] = w[m];
  }

  if (h) {
   FOR(i,16) x[i] += y[i];
   FOR(i,4) {
    x[5*i] -= ld32(c+4*i);
    x[6+i] -= ld32(in+4*i);
   }
   FOR(i,4) {
    st32(out+4*i,x[5*i]);
    st32(out+16+4*i,x[6+i]);
   }
  } else
   FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
}

int crypto_core_salsa20(u8 *out,const u8 *in,const u8 *k,const u8 *c)
{
  core(out,in,k,c,0);
  return 0;
}

int crypto_stream_salsa20_xor(u8 *c,const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
   crypto_core_salsa20(x,z,k,sigma);
   FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
   u = 1;
   for (i = 8;i < 16;++i) {
    u += (u32) z[i];
    z[i] = u;
    u >>= 8;
   }
   b -= 64;
   c += 64;
   if (m) m += 64;
  }
  if (b) {
   crypto_core_salsa20(x,z,k,sigma);
   FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

```
int crypto_onetimeauth(u8 *out,const u8 *m,u64 n,const u8 *k)
{
  u32 s,i,j,u,x[17],r[17],h[17],c[17],g[17];

  FOR(j,17) r[j]=h[j]=0;
  FOR(j,16) r[j]=k[j];
  r[3]&=15;
  r[4]&=252;
  r[7]&=15;
  r[8]&=252;
  r[11]&=15;
  r[12]&=252;
  r[15]&=15;

  while (n > 0) {
    FOR(j,17) c[j] = 0;
    for (j = 0;(j < 16) && (j < n);++j) c[j] = m[j];
    c[j] = 1;
    m += j; n -= j;
    add1305(h,c);
    FOR(i,17) {
      x[i] = 0;
      FOR(j,17) x[i] += h[j] * ((j <= i) ? r[i - j] : 320 * r[i + 17 - j]);
    }
    FOR(i,17) h[i] = x[i];
    u = 0;
    FOR(j,16) {
      u += h[j];
      h[j] = u & 255;
      u >>= 8;
    }
    u += h[16]; h[16] = u & 3;
    u = 5 * (u >> 2);
    FOR(j,16) {
      u += h[j];
      h[j] = u & 255;
      u >>= 8;
    }
    u += h[16]; h[16] = u;
  }

  FOR(j,17) g[j] = h[j];
  add1305(h,minusp);
  s = -(h[16] >> 7);
  FOR(j,17) h[j] ^= s & (g[j] ^ h[j]);

  FOR(j,16) c[j] = k[j + 16];
  c[16] = 0;
  add1305(h,c);
  FOR(j,16) out[j] = h[j];
  return 0;
}
```

```
int crypto_core_hsalsa20(u8 *out,const u8 *in,const u8 *k,const u8 *c)
{
  core(out,in,k,c,1);
  return 0;
}

int crypto_stream_xor(u8 *c,const u8 *m,u64 d,const u8 *n,const u8 *k)
{
  u8 s[32];
  crypto_core_hsalsa20(s,n,k,sigma);
  return crypto_stream_salsa20_xor(c,m,d,n+16,s);
}

static int vn(const u8 *x,const u8 *y,int n)
{
  u32 i,d = 0;
  FOR(i,n) d |= x[i]^y[i];
  return (1 & ((d - 1) >> 8)) - 1;
}

int crypto_verify_16(const u8 *x,const u8 *y)
{
  return vn(x,y,16);
}

int crypto_onetimeauth_verify(const u8 *h,const u8 *m,u64 n,const u8 *k)
{
  u8 x[16];
  crypto_onetimeauth(x,m,n,k);
  return crypto_verify_16(h,x);
}

int crypto_stream_salsa20(u8 *c,u64 d,const u8 *n,const u8 *k)
{
  return crypto_stream_salsa20_xor(c,0,d,n,k);
}

int crypto_stream(u8 *c,u64 d,const u8 *n,const u8 *k)
{
  u8 s[32];
  crypto_core_hsalsa20(s,n,k,sigma);
  return crypto_stream_salsa20(c,d,n+16,s);
}

int crypto_secretbox(u8 *c,const u8 *m,u64 d,const u8 *n,const u8 *k)
{
  int i;
  if (d < 32) return -1;
  crypto_stream_xor(c,m,d,n,k);
  crypto_onetimeauth(c + 16,c + 32,d - 32,c);
  FOR(i,16) c[i] = 0;
  return 0;
}
```

```
int crypto_secretbox_open(u8 *m,const u8 *c,u64 d,const u8 *n,const u8 *k)
{
  int i;
  u8 x[32];
  if (d < 32) return -1;
  crypto_stream(x,32,n,k);
  if (crypto_onetimeauth_verify(c + 16,c + 32,d - 32,x) != 0) return -1;
  crypto_stream_xor(m,c,d,n,k);
  FOR(i,32) m[i] = 0;
  return 0;
}

int ReadSD(byte filen, unsigned long flocation, unsigned int bsize) // sd data to
sdbuffer
{
  if(bsize<512) bsize=512;
  unsigned int blocksize=bsize;
  bool retval=false;
  int yi=0;

  // re-open the file for reading:
  myFile2 = SD.open("rdata.002");
  if(myFile2==false)
  {
    return -4;
  }
  retval=myFile2.seek(flocation);
  if(retval==false)
  {
    return -3;
  }

  if(flocation==0)              // reading a file from the beginning (NOT saving location
but restart on each boot)
  {
    file_size1 = myFile2.size();   // get size in bytes
    if(file_size1<512) return -1;  // file size should be >= 512 bytes (yet only enough for
4 packets)
  }

  if(flocation + blocksize >= file_size1)  // read should not go off the file
  {
    return -2;
  }

  if(myFile2)
  {
    while (myFile2.available() && yi<blocksize)
    {
      sdbuffer_rx[i]=myFile2.read();
      //Serial.write(sdbuffer_rx[i]);
      yi++;
```

```
   }
   // close the file:
   myFile2.close();
   return 0;
  }
  else
  {
   // if the file didn't open, print an error:
   if(debug) Serial.println("error opening rdata.002");
  }
}


int HandlePacket()
{
  int leni            = 5; // mem file and addr
  int yx              = 0;
  unsigned long mem_loc = 0;
  byte mem_fil        = 0;

  leni+=128;    // data
  leni+=16;     // authenticator

  for(int y=leni-16; y<leni; y++) // collect authenticator
  {
   a[yx++]=receivedChars[y];
  }

  int j=crypto_onetimeauth_verify(a,receivedChars,leni-16,key);
  if(j==0) // authentication success
  {
   yx=0;
   mem_fil = receivedChars[yx++]; // collect memory file & its share of 512
   mem_loc |= receivedChars[yx++]<<24;
   mem_loc |= receivedChars[yx++]<<16;
   mem_loc |= receivedChars[yx++]<<8;
   mem_loc |= receivedChars[yx++];

   if(mem_fil==0)
   {
    yx = ReadSD(0,mem_loc*512,0); // mem_loc*512,0);      // read data to sdbuffer
    if(yx<0)
    {
     if(debug && random(100)<1)
     {
      Serial.print("SD fail (512*) ");
      Serial.println(mem_loc);
     }
    }
   }

   counter1++;
```

```
   for(int xi=0; xi<leni-16;xi++)
   {
    // convert 8 bit samples to 16 bit samples and XOR operate with sdbuffer

array3[xi]=(receivedChars[xi+5]^sdbuffer_rx[(mem_fil*AUDIO_BLOCK_SAMPLES)+xi]
)*255-32768;
   }
   memcpy(pqueue2.getBuffer(), &array3[0], 2 * AUDIO_BLOCK_SAMPLES); // 2 *
because bytes vs shorts
   pqueue2.playBuffer();
   memset(array3,0,2*AUDIO_BLOCK_SAMPLES);
 }
 else
 {
  counter2++;
 }
 return 0;
}

int recvWithStartEndMarkers() {
   static boolean recvInProgress = false;
   static byte ndx = 0;
   byte startMarker = 0xAA;
   byte lastMarker = 0x00;
   byte endMarker = 0xFF;
   char rc;
   int len=0;

   while (HWSERIAL.available() > 0 && newData == false) {
      rc = HWSERIAL.read();
      //Serial.print(rc);

      if (recvInProgress == true)               // 2 start markers have been found
      {
         if ((rc != endMarker || ndx < numChars-2)) // either one should be true to
expect data + length should be no longer than 160, ever
         {
            receivedChars[ndx] = rc;
            ndx++;
            if (ndx >= numChars)
            {
               ndx = numChars - 1;
            }
         }
         else                              // footer found after numChars-2 (data)
         {
            recvInProgress = false;
            len=ndx;
            ndx = 0;
            newData = true;
         }
      }
```

```
      else if (rc == startMarker) // check for 2 x AA
      {
         if(lastMarker==startMarker)
         {
          recvInProgress = true;
          lastMarker=0;
         }
         else
         {
          lastMarker=startMarker;
         }
      }
      else
      {
       lastMarker=0;
      }
   }

   return len;
}

void showNewData() {
   if (newData == true && debug) {
      Serial.print("RX: ");
      for(int n=0; n<numChars; n++)
      {
       Serial.print(receivedChars[n]);
       Serial.print(',');
      }
      Serial.println();
      newData = false;
   }
}

void setup()
{
 pinMode(buttonPin, INPUT);
 AudioMemory(512);
 sgtl5000_1.enable();
 sgtl5000_1.inputSelect(myInput);
 sgtl5000_1.volume(1);

 Serial.begin(9600);      // 9600
 HWSERIAL.begin(4608000);  //  4608000 2000000 1000000 500000
 HWSERIAL.setTX(26);
 HWSERIAL.setRX(27);

 rqueue3.begin();
 Serial.println("Begin");
 delay(2500);

 Serial.print("Initializing SD card...");
 if (!SD.begin(chipSelect))
```

```
 {
  Serial.println("initialization failed!");
  return;
 }
 Serial.println("initialization done.");
}

void loop() {
 int retval=0;
 ret=0;
 newData = false;
 l=1; // wait for button

 if ( bouncer.update() )
 {
   if ( bouncer.read() == HIGH)
   {
    if ( ledValue == LOW )
    {
     if(debug) Serial.println("B DN");
     ledValue = HIGH;
     l=0;
     rqueue3.clear();
     rqueue3.begin();
    }
    else
    {
     if(debug) Serial.println("B UP");
     myFile.close();
     ledValue = LOW;
     rqueue3.end();
    }
   }
 }

 i=0;
 // open the file for reading only once, since repeated opening does not work
correctly with audio and serial
 if(l==0)
 {
  myFile = SD.open("rdata.001");
  if(myFile==false)
  {
   Serial.println("SD TX open fail");
  }
  file_size = myFile.size();
  if(debug) Serial.println(file_size);
 }

 // RX SEQUENCE
 if(ledValue == LOW)
 {
  ret=recvWithStartEndMarkers();
```

```
  if(ret>0)
  {
   //if(debug) Serial.print("RX");
   if(random(1000)<2) showNewData();
   HandlePacket();
  }
 }

 // TX SEQUENCE
 if(ledValue == HIGH)
 {
  if(((file_location*512) + 512) <= file_size) // only if SD ok, continue to read queue
  {
   //if(debug) Serial.print("TX");
   ret=rqueue3.available();
   if(ret>=4)    // only continue if 4 or more packets, that will make 512 bytes or
more audio
    {
     // SD TX read
     unsigned long sum = 0;

     myFile.seek(file_location*512);
     if(myFile)
     {
      while (myFile.available() && i<512)
      {
       sdbuffer_tx[i]=myFile.read();
       i++;
      }
     }
     else
     {
      if(debug) Serial.println("error opening rdata.001");
     }
     if(debug1 && file_location % 100 == 0) Serial.println(file_location);


     // make packet
     i=0;
     while(i<4)  // 4 packets, since SD
     {
      memcpy(&array1[0], rqueue3.readBuffer(), 2*AUDIO_BLOCK_SAMPLES);
      rqueue3.freeBuffer();

      j=0;
      k=0;

      txbuffer[j++]=0xAA;
      txbuffer[j++]=0xAA;
      txbuffer[j++]=file=i;           // add file: pointing to 0, 128, 256 or 384
      txbuffer[j++]=file_location>>24;  // add file location: MSB
      txbuffer[j++]=file_location>>16;
      txbuffer[j++]=file_location>>8;
```

```
        txbuffer[j++]=file_location;     // LSB

        while(k<AUDIO_BLOCK_SAMPLES)
        {
          m=32768+array1[k];             // -32768...32768 to 0..65536
          //txbuffer[j++]=m/255;         // from 0..65536 to 0..255 // RAW
          txbuffer[j++]=(m/255)^sdbuffer_tx[(i*AUDIO_BLOCK_SAMPLES)+k]; // XOR
audio with sdbuffer from 0..127, 128..256,.. 512
          //txbuffer[j++]=(m/255)^sdbuffer[k];
          k++;
        }

        au = crypto_onetimeauth(a,&txbuffer[2],k+5,key);  // make 16-byte
authenticator for txbuffer data of len k+5 with key
        if(au!=0 && debug) Serial.println("Auth fail");

        k=0;
        while(k<16)
        {
          txbuffer[j++]=a[k];
          k++;
        }

        txbuffer[j++]=0xFF;
        HWSERIAL.write(txbuffer,j); // j == 152, of which msg data is 128, header and
footer are 3, authenticator is 16 (+5 of key stuff)
        //if(debug && i==0) Serial.println("TX");
        i++;
       }
       file_location++;              // locate next 512 bytes
     }
    }
    else
    {
     Serial.print("RESETING: ReadSD TX over limits ");
     Serial.println(file_location);
     file_location=0;
    }
  }

  if(debug && random(1000000)<1)
  {
    Serial.print(counter1);
    Serial.print("/");
    Serial.print(counter1+counter2);
    Serial.println(" pass");
  }
  l++;
}
```