Dimitrije Erdeljan

Artificial EM side channel

Computer Science Tripos – Part II

Trinity College

May 18, 2018

Proforma

Name:	**Dimitrije Erdeljan**
College:	**Trinity College**
Project Title:	**Artificial EM Side Channel**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	**10789**[1]
Project Originator:	Prof. Ross Anderson
Supervisor:	Dr Markus Kuhn

Original Aims of the Project

The main goal of the project was to demonstrate that a covert one-way communication channel can be created on a microcontroller by only making software alterations. Such a channel was meant to be created by using a pin as a radio antenna, and transmitting data using an appropriate modulation technique.

The project aimed to implement several modulation techniques and the matching demodulators, as well as to evaluate their performance over a range of distances and compare their performance.

Work Completed

I successfully demonstrated that the covert channel can be created, and implemented a complete frequency-shift keying and phase-shift keying modulator and demodulator. I also implemented a direct-sequence spread spectrum modulator, as well as a complete prototype and a partially complete "real" demodulator.

To evaluate the performance of the two modulation techniques which I completely implemented, I made measurements of the bit error rate over several distances. I also tested the prototype direct-sequence spread spectrum demodulator on computer-generated data, and also demonstrated that the transmitted signal contains the expected pseudonoise sequence.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Special Difficulties

The main difficulty I did not predict when planning the project was the implementation of modulators with timing accurate to a single cycle. I planned to implement them in assembly, using the architecture documentation as a source of information about the individual instructions' execution times. While working on the project, I discovered that this documentation does not provide a complete description of the real timings. To overcome this issue, I had to adjust my programs to minimise the possible impact of unexpected timing delays, and make measurements to confirm that they are functioning correctly.

Declaration

I, Dimitrije Erdeljan of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: May 18, 2018

Contents

1 Introduction **11**
1.1 Motivation . . . 11
1.2 Related work . . . 12

2 Preparation **13**
2.1 Theoretical background . . . 13
2.1.1 Radio transmission . . . 13
2.1.2 Frequency-shift keying . . . 14
2.1.3 Phase-shift keying . . . 14
2.1.4 Direct-sequence spread spectrum modulation . . . 15
2.2 Hardware . . . 16
2.2.1 Microcontroller . . . 17
2.2.2 Software-defined radio . . . 18
2.3 Software and libraries . . . 18
2.4 Preliminary work . . . 19
2.5 Starting point . . . 20

3 Implementation **21**
3.1 Overall structure . . . 21
3.2 Transmitter overview . . . 21
3.3 Data encoding . . . 22
3.3.1 Line code . . . 22
3.3.2 Framing . . . 22
3.4 Modulation . . . 23
3.4.1 Frequency-shift keying . . . 23
3.4.2 Phase-shift keying . . . 24
3.4.3 Spread-spectrum modulation . . . 25
3.5 Receiver overview . . . 26
3.6 Demodulation . . . 26
3.6.1 Frequency-shift keying . . . 27
3.6.2 Phase-shift keying . . . 28
3.6.3 Spread-spectrum modulation . . . 29

4 Evaluation **33**
4.1 Bit error rate measurement . . . 33
4.1.1 Setup . . . 33
4.1.2 Frequency-shift keying . . . 34
4.1.3 Phase-shift keying . . . 35
4.2 Direct-sequence spread spectrum . . . 38
4.2.1 Offline prototype . . . 38
4.2.2 Online processing . . . 39

5 Conclusion **43**
5.1 Results . . . 43

Bibliography **43**

A Source samples **47**
A.1 Modulator . . . 47
A.1.1 Header . . . 47
A.1.2 Frequency-shift keying . . . 47
A.1.3 Phase-shift keying . . . 48
A.1.4 Direct-sequence spread spectrum modulation . . . 50
A.2 GNURadio blocks . . . 51
A.2.1 Main body . . . 51
A.2.2 Acquisition . . . 52
A.2.3 Cross-correlation . . . 52

B Project Proposal **55**

List of Figures

2.1 Plot of the autocorrelation of the linear-feedback shift register sequence with feedback polynomial $x^7 + x^6 + 1$ 16
2.2 Plot of offset error estimate Δ_ϕ as a function of the true error 17

3.1 Diagram showing the data frame format 22
3.2 Block diagram of the frequency-shift keying demodulator 27
3.3 Block diagram of the phase-shift keying demodulator 28

4.1 Plot of bit error rate of the frequency-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 50 cm from the antenna 35
4.2 Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 50 cm from the antenna 36
4.3 Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 1 m from the antenna 37
4.4 Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 3 m from the antenna 37
4.5 Plots of direct-sequence spread spectrum demodulator output before thresholding (in blue) and after thresholding (in orange), for several values of signal-to-noise ratio S and relative frequency error δ_f 41
4.6 Plots of cross-correlation of the recorded direct-sequence spread spectrum signal and the chipping sequence 41

Acknowledgements

I would like to thank my supervisor, Dr Markus Kuhn, for the advice and guidance he provided during my work on this project, as well as his assistance with measurement equipment. I would also like to thank Prof. Ross Anderson for the project idea, and everybody who read the drafts of this document for the feedback regarding its content and presentation.

Chapter 1

Introduction

This dissertation describes my project, which demonstrates the possibility of creating an artificial electromagnetic side-channel in a microcontroller by modifying the software it is running. I have successfully implemented two modulation techniques: frequency-shift keying and phase-shift keying, as well as the matching demodulators, and evaluated their performance by measuring the bit error rate of the covert channel over several distances. I have also implemented a direct-sequence spread spectrum modulator, with a prototype demodulator and a partially completed "real" implementation.

1.1 Motivation

In order to motivate the topic of this project, consider the following scenario: an attacker wants to backdoor an electronic device which does not have any communication hardware, but needs a way to transmit data. An example would be a keyboard that logs the user's keystrokes, where the attacker would normally have to physically recover the keyboard to read the recorded data.

Simply soldering a radio transmitter to the microcontroller in the device would provide a communication channel. This is, however, easily detected even without specialised equipment. More sophisticated hardware modification is possible, up to inserting a transmitter in the microcontroller design at some stage in the production chain, but requires significant resources.

This project demonstrates that the creation of a covert channel is also achievable for an attacker with limited resources. In particular, the attacker is not allowed to alter the hardware in any way. They are limited to modifying the device's firmware, and can overwrite the code executed by the microcontroller to insert the backdoor.

Since consumer electronic devices are generally produced using off-the-shelf microcontrollers, which come in a limited range of packages, it is likely that at least one of the pins on the microcontroller is not used. The side channel is created by using one of the pins as an antenna for an improvised radio transmitter – by rapidly toggling the pin's voltage, we can transmit a radio signal the attacker can listen to.

The project examines three different modulation techniques: frequency-shift keying, phase-shift keying and direct-sequence spread spectrum modulation. They provide dif-

ferent tradeoffs between simplicity, robustness, performance and stealth. For example, a frequency-shift keying transmitter is quite simple to implement, but easily detected, while one using spread-spectrum modulation is much more difficult to spot, but requires very precisely timed code.

1.2 Related work

Emission security is an active research area, with targets including devices such as PC monitors, credit card chips, network cables, etc [1]. While these attacks are often passive, active attacks have been demonstrated as well, such as the use of images generated to control electromagnetic radiation of a monitor [2].

There has been recent work in the area of hardware Trojans, including both the development of attack methods and detection techniques [5]. Some of the techniques used to broadcast data are similar to the one used in for this project, but it should be noted that these backdoors are implemented on a lower level than firmware modification.

Chapter 2

Preparation

In this chapter, I will describe how the project was planned, as well as the hardware, software tools and libraries that I used. I will also give a brief overview of the relevant theory, including a description of the modulation techniques I implemented. Finally, I will describe the preparatory work I did before submitting the project proposal, and summarise the starting point from which I developed the project.

2.1 Theoretical background

In this section, I will give an overview of the theory relevant to this project. I will give a brief description of the behaviour of a short microcontroller pin when used as a radio-antenna. I will also describe the modulation techniques used in the project, as well as the methods used for their demodulation.

2.1.1 Radio transmission

We can view a microcontroller pin, whose voltage is periodically toggled at frequency f, as a monopole antenna transmitting a square wave at frequency f.

The wavelength at which a monopole antenna of length l has the greatest gain is $\lambda = 4l$. Even with a generous estimate of the pin length $l = 1\,\text{cm}$, this frequency is $f = c(4l)^{-1} \approx 7.5\,\text{GHz}$, which is quite far from the frequencies a microcontroller (with a clock rate on the order of tens of megahertz) can produce.

Since the microcontroller is transmitting a square wave, we can improve the gain by using a higher harmonic of the wave's frequency for transmission. A square wave with frequency f can be expressed as a sum of sine waves (harmonics) at frequencies $f, 3f, 5f, \ldots$, so we can receive the signal at kf for some larger k, for which the transmitter antenna gain is higher. In particular, the Fourier series of a square wave is:

$$\frac{4}{\pi}\sum_{n=1}^{\infty}\frac{1}{2n-1}\sin((2n-1)\cdot 2\pi f t)$$

The amplitude of the k-th harmonic is inversely proportional to k. Therefore, we get a trade-off where increasing k improves the performance of our antenna, but decreases the available power that can be transmitted.

If the pin is connected to a length of wire that can serve as an antenna, receiving at a higher harmonic is sufficient to get a signal that is strong enough to transmit data. This can, for example, happen with keyboards, where the wires of the matrix circuit can be used.

Even if this is not the case, objects near the pin can be exploited to improve the signal strength. A voltage on the pin can induce an voltage in nearby objects, making them capacitively coupled. The object then effectively becomes an antenna, increasing the transmitter's gain.

If the pin we are using to transmit is grounded instead of floating, we can view the resulting transmitter as a magnetic-loop antenna formed by the pin and power supply lines. While this risks overheating and damaging the microcontroller if the resulting current is too high, it allows us to exploit the (long) power supply cables as a part of the antenna.

2.1.2 Frequency-shift keying

A frequency-shift keyed signal represents data by modulating the frequency of the carrier wave. In particular, we are interested in binary frequency-shift keying, where a frequency f_0 encodes a 0-bit, and a different frequency f_1 encodes a 1-bit.

To demodulate such a signal, we can construct a pair of filters matched to f_0 and f_1. Assuming that there are no other transmissions at these frequencies, we can compare the power of their outputs to detect whether the input encodes a "0" or a "1".

2.1.3 Phase-shift keying

A phase-shift keyed signal represents data by modulating the phase of a carrier wave. For a sinusoidal carrier with angular frequency ω, it can be described as

$$s(t) = Ae^{j(\omega t + d(t) + \phi_0)}$$

where $d(t)$ is the phase shift that is used to encode data, and ϕ_0 is an unknown phase offset. The data can be recovered by multiplying the signal by a sinusoidal wave with angular frequency $-\omega$:

$$s'(t) = s(t) \cdot e^{-j\omega t} = Ae^{j(d(t) + \phi_0)}$$

Taking the phase of $s'(t)$ gives us $d(t) + \phi_0$.

For the purposes of this project, we are interested in binary phase-shift keying, where the two values are represented as $d(t) = 0$ and $d(t) = \pi$. Implementing it on a transmitter that modulates a square wave is straightforward, since a phase shift of π is equivalent to inverting the output.

Since the transmitter and receiver do not share a clock, it is not possible to recover the phase offset ϕ_0. To resolve this issue, I used differential phase-shift keying, which encodes data in the phase change instead of directly modulating the phase. A phase jump of π between two consecutive bits encodes a "1", and a constant phase encodes a "0".

If we take the derivative of the phase of $s'(t)$, we get $d'(t)$, which is independent of the offset. 1-bits are then visible as spikes in $d'(t)$.

2.1.4 Direct-sequence spread spectrum modulation

Direct-sequence spread spectrum modulation is similar to phase-shift keying, as they both encode data in the phase of a carrier wave. They differ in the choice of carrier: instead of a sine or square wave, direct-sequence spread spectrum modulation uses a wide-band signal.

The carrier wave (called a chipping sequence) is a periodic pseudo-noise sequence. For this project, I used a maximum-length linear-feedback shift register to generate it. A single bit is modulated on the carrier for each period of the sequence, by transmitting the pseudo-noise sequence directly if it is "0" and inverting it if the bit is "1". Therefore, the signal can be described as

$$s(t) = Ad(t)c(t)$$

where A is the amplitude, $d(t) \in \{-1, 1\}$ the data, and $c(t) \in \{-1, 1\}$ the periodic pseudo-noise sequence.

To demodulate the signal, it is multiplied by a replica of the chipping sequence:

$$s(t)c(t) = Ad(t)c(t)^2 = Ad(t)$$

Similarly to phase-shift keying, the value of A is unknown, and we cannot determine which of the values A and $-A$ corresponds to which bit. The solution is, again, differential encoding, where data is encoded in the phase change.

Direct-sequence spread spectrum modulation offers several advantages over frequency- and phase-shift keying, including the following:

1. It is more difficult to detect by an observer that does not know what the chipping sequence is.
2. It is more resistant to adversarial jamming, since the signal energy is spread over a wider frequency range.
3. It has a better signal-to-noise ratio, where the improvement is equal to the ratio of the chip rate and data rate.

Carrier acquisition

To demodulate a direct-sequence spread spectrum signal, the receiver needs to find the offset at which the transmitted chipping sequence begins. To do this, we can choose a sequence such that its autocorrelation is small for all offsets apart from the zero offset. As an example, the autocorrelation for a 128-bit linear-feedback shift register sequence is shown in Figure 2.1.

This is satisfied by maximal-length sequences, such as the one I used. Therefore, the correct offset can be found by computing the cross-correlation of the signal and chipping sequence and finding the offset at which it has a peak.

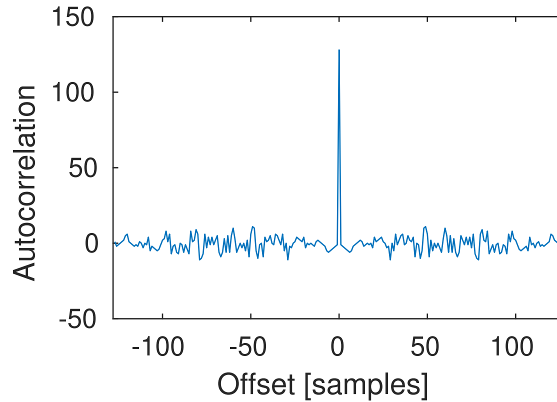


Figure 2.1: Plot of the autocorrelation of the linear-feedback shift register sequence with feedback polynomial $x^7 + x^6 + 1$

Delay-locked loop

The acquisition procedure described above is not sufficient for practical demodulation. If the frequencies of the transmitted chipping sequence and local replica do not match exactly, the offset will slowly drift over time, causing the receiver to lose lock on the carrier. Therefore, some mechanism which can maintain the offset is necessary, such as a delay-locked loop.

The autocorrelation of the chipping sequence at an offset $0 < \epsilon < 1$ is equal to a linear combination of the autocorrelations at offsets 0 and 1. Due to the peak at zero, this value will be linearly decreasing with ϵ. Similarly, at offsets $-1 < \epsilon < 0$, the value will be increasing linearly with increasing ϵ.

Consider two replicas of the chipping sequence, at offsets $+\frac{1}{2}$ and $-\frac{1}{2}$. Let the correlation of the input signal with the two replicas (call them "early" and "late") be E and L respectively. If there is no error in the offset, the two values will be equal (due to the symmetry of the autocorrelation peak at zero offset). Otherwise, $|E|$ will be greater than $|L|$ if the early replica is "closer" to the correct offset, and vice versa.

Assuming that the error in the offset is less than one half of a sample, it can be computed from E and L using the following equation [3]:

$$\Delta_\phi = \frac{1}{2} \frac{|E|^2 - |L|^2}{|E|^2 + |L|^2}$$

This estimate can then be used to update the offset estimate and maintain the chipping sequence replica. An example plot of the value of Δ_ϕ near zero is shown in Figure 2.2.

2.2 Hardware

In this section, I will give a description of the hardware I used for the project and highlight their relevant features. I will also describe some factors that can introduce an error in the transmitted or received signal, which I had to take into account when working on the project.

The hardware I used (other than a computer) is:

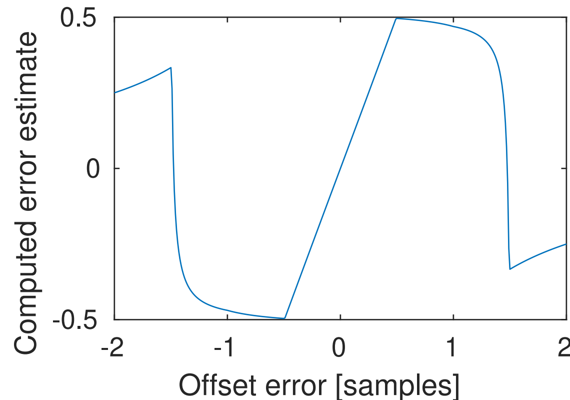


Figure 2.2: Plot of offset error estimate Δ_ϕ as a function of the true error

- An **mbed NXP LPC11U24** microcontroller and **Arm mbed OM13032** development board, used as the transmitter, and
- an **Ettus USRP2** software-defined radio peripheral, with the **WBX-40** daughterboard, to receive the signal and forward it to the computer for processing.

2.2.1 Microcontroller

The development board I used contains the NXP LPC11U24 microcontroller and USB interface over which the controller is programmed.

The microcontroller contains an Arm Cortex-M0 processor with a 48 MHz system clock, 8 KB of random-access memory and 32 KB of FLASH storage. It can be programmed over the USB interface, which presents the controller as a storage drive. Compiled programs are copied to the storage as `.bin` files. When the microcontroller is powered on or reset, it will load this binary and execute it.

Apart from transferring executable code, the USB interface can be used by the running program. It provides serial communication with the attached computer, which I used to send commands to the microcontroller. These commands were used to start transmissions automatically, which allowed me to partially automate the testing of the project.

The microcontroller has multiple general purpose input/output pins. These are accessed by writing to memory-mapped registers, which trigger actions such as setting the pin state or toggling them. These operations take the same time as writing to any other memory location (two cycles).

Error sources

It is important to mention that the clock rate of the microcontroller is not necessarily exactly 48 MHz. The clock signal is synthesised using an internal crystal oscillator and a frequency multiplier. Its frequency can drift over time, primarily due to changes in temperature. The manufacturer datasheet specifies that the clock rate can deviate by up to 1%, unless the microcontroller is configured to use an external clock source.

This is not important for the code running on the microcontroller, since all instructions are slowed down or sped up by the same factor. It does, however, affect the frequency of the transmitted waveform, and I needed to account for this when implementing the demodulators. I also configured the microcontroller to use the external crystal oscillator provided on the development board, in order to reduce the range of the frequency error.

While testing the microcontroller, I noticed that the execution time of instructions does not always match the cycle counts given in the manufacturer's documentation. One common source of delays I identified are branches: if a branch target falls in the same 16-byte block as the source (that is, they differ only in the last four bits), it takes three cycles, as specified in the documentation. However, if this is not the case, the branch time includes an additional four-cycle delay.

In order to make sure that the timing of my implementation is accurate, I had to account for these undocumented delays. I partially mitigated them by designing the code to minimise the number of locations where this could be an issue. Since this does not eliminate all timing errors, I had to plan to test the output using an oscilloscope, and manually adjust the timing until it is correct.

2.2.2 Software-defined radio

The Ettus Universal Software Radio Peripheral is a device that functions as a radio receiver which samples the received signal and forwards it over Ethernet to an attached computer for processing.

Since sampling a signal containing high-frequency components at the Nyquist rate would produce prohibitively many samples, the signal is downconverted first. If the device is configured to receive at a centre frequency f_c, the signal is multiplied by $e^{2\pi j f_c t}$ to shift the centre frequency to zero. Low-pass filtering the shifted signal allows the software-defined radio to sample at a lower rate, as long as we are only interested in a narrow band of the spectrum.

Since the signal is multiplied by a complex exponential, it is no longer real-valued. Therefore, the samples will be complex values.

Error sources

Leakage from the receiver's local oscillator introduces a noise component in the received signal at the centre frequency. This noise is stronger than signals that need to be demodulated, and therefore I had to ensure that the software-defined radio is configured to avoid overlapping it with the signal. This can either be done by configuring an offset in the software radio peripheral driver, or "manually", by multiplying the data with a complex phasor to shift the frequency range.

2.3 Software and libraries

Since the toolchain developed by Arm for use with the development board is based on a C++ compiler, I chose to implement the majority of the code running on the micro-

controller in C++. This allowed me to make use of the *mbed* libraries provided with the toolchain, which include support for serial communication over the USB interface. Using the library meant that I did not need to spend a lot of time working on the computer-controller interface, and allowed me to focus on implementing the modulation algorithms.

While I implemented most of the transmitter in C++, some routines had to be written in assembly to provide timing accurate on a cycle level. I organised the code in such a way that these routines are as small as possible, so that I could keep the rest of the implementation in a high-level language. This made the development process easier, and simplified understanding and maintaining the codebase.

After implementing the transmitter, I recorded some sample data and wrote prototype demodulators in MATLAB. The choice of MATLAB provided me with an extensive library of mathematical operations, allowing me to test the prototypes without having to debug low-level implementation details. Analysing data “offline” also meant that I did not need to write optimised code, since it did not need to be able to process incoming samples in real time.

Once I had working prototypes, I re-implemented them to process data and output the demodulated information in real time. For this, I used GNURadio, which is a free and open-source toolkit that is used to process software radio data by composing signal processing blocks. A GNURadio project is a directed acyclic graph of these blocks, where each block transforms the samples it receives on its input and provides its output to the next step in the processing chain.

GNURadio includes a library of signal processing blocks for common operations, such as basic arithmetic, filtering, plotting and output. Where possible, I used them to minimise the amount of code I had to write. In cases where I needed an operation that is not included in the library, I had to implement custom blocks myself.

These custom blocks can either be written in C++ or Python. Since some of them had to be able to support large amounts of data (on the order of millions of samples per second), I chose to implement them in C++ to avoid the overhead of Python interpretation.

2.4 Preliminary work

Before I submitted the project proposal, I created a small proof of concept implementation to verify that the project goal is achievable. In particular, the purpose of this test was to confirm that a microcontroller can transmit a signal that is strong enough to detect.

The proof of concept was a small program, implemented in a mixture of C++ and assembly, that toggles a pin at a constant frequency. The program switches between two different frequencies, which should be visible as two peaks on the received frequency spectrum.

With the help of Dr Kuhn, I recorded this signal using the R&S FSV7 spectrum analyser and a 30–300 MHz biconical antenna. With the microcontroller plugged into a breadboard, at a distance of around two metres, we confirmed that there is a visible peak at the frequency of the 51st harmonic of the transmission frequency.

The peak was around 10 dB greater than ambient noise, confirming that it should be possible to transmit data using a pin as an antenna.

2.5 Starting point

When I started working on the project, I had an environment for programming the microcontroller set up, along with the program written for the test described in the previous section. I was familiar with assembly programming, but did not have any knowledge of the tools used for digital signal processing, which I had to learn while working on the project.

Excluding the test program described in the previous section, all components of the project were implemented after the proposal was accepted.

Chapter 3

Implementation

In this chapter, I will describe the modules that this project consists of: the transmitter and the receiver. For both of them, I will first give an overview of their structure, followed by a more in-depth description of the algorithms used and relevant implementation details.

3.1 Overall structure

The project consists of two main components:

- The **transmitter**, which is a microcontroller running a program that outputs a modulated waveform to one of its pins. This pin serves as an antenna for the transmitter.
- The **receiver**, which uses a software-defined radio peripheral to receive the transmitted signal, and processes the signal on a computer to demodulate it.

They both support three different modulation techniques:

- frequency-shift keying,
- phase-shift keying, and
- direct-sequence spread spectrum modulation.

All three techniques are implemented as components of a single program for the transmitter, which selects the appropriate modulation based on external commands. The receiver consists of three separate GNURadio projects, each used for demodulating one of the methods listed above.

3.2 Transmitter overview

I implemented a simple external interface for the transmitter, which receives commands over a USB connection. The commands are strings of the form `<modulation> <sequence generator>`. Here, `modulation` is one of `FSK`, `PSK` or `DSSS`, each selecting one of the three

implemented modulation techniques. The sequence generator is a number describing the generating polynomial for the binary sequence that should be transmitted.

After a command is received, the first step is to generate the data. I did this using a linear-feedback shift register of fixed length, with the positions of the taps given by the binary representation of the sequence generator. For example, a generator equal to 13 (1101 in binary) corresponds to taps on bits 1, 3 and 4 (zero-indexed).

This data is then passed on to the preprocessing step shared by all three modulation techniques, which encodes it and wraps it in a frame. The output of this stage is then passed on to the modulator selected in the first half of the command and transmitted.

3.3 Data encoding

All modulation techniques share a common preprocessing step, which takes an array of data bits as the input and returns an array of bits that represent a frame containing the encoded data. This output is then passed on to one of the modulation functions and transmitted.

3.3.1 Line code

Due to the transmitter and receiver not sharing a clock, timing information must be extracted from the received signal. To ensure that the signal contains frequent transitions, I used the Manchester code as the line code.

Zero-bits are encoded as “01”, and one-bits are encoded as “10”. This halves the communication channel’s data rate, since transmitting each bit requires two symbols, but ensures that there will be at most two symbols between transitions.

3.3.2 Framing

To allow the receiver to recognise the beginning and end of a transmission, I implemented a simple framing mechanism. Each data frame starts with a header, consisting of a sequence of 0-bits and a preamble marking the start of the frame. The data bits follow, with an “end of frame” marker after them. A diagram of the frame format is shown in Figure 3.1.

Figure 3.1: Diagram showing the data frame format

The sequence of 0-bits is used to ensure that no data is lost if the receiver drops some bits from the beginning of a frame. This can occur because the receiver computes an average of the signal power in order to detect transmissions. This average takes some time to cross the detection threshold, and all bits that are received before the threshold

is crossed are discarded. The 0-bits are padding that is safe to drop and long enough to guarantee that the signal will be detected before the beginning of the preamble.

A sequence of 1-bits is used as a marker signifying the beginning and end of a frame. This sequence cannot occur in the data segment, since the Manchester code can contain at most two consecutive 1-bits. Therefore, a block of data can only be mistaken for an end-marker if a half of the bits are corrupted, which is very unlikely.

The preamble ends with a single 0-bit. This allows the receiver to detect the end of preamble unambiguously – without the 0-bit, if the data segment starts with a "1", it could be mistaken for the last bit of the preamble.

3.4 Modulation

3.4.1 Frequency-shift keying

The frequency-shift modulator iterates through data bits until it reaches an "end of data" marker. For each bit, it calls a routine that transmits a square wave at the frequency that corresponds to the bit's value. Most of the modulator does not require precise timing, and is therefore implemented in C++. The routines that generate the square wave, however, need delays accurate to a single cycle and are implemented in assembly.

Since the complexity of the modulator is entirely in the code transmitting the square waves, the rest of this subsection will describe the function of these routines. As an example, consider the loop given in Algorithm 1:

Algorithm 1 Loop transmitting a square wave

```
LDR R1, =0x50002300  // NOTP0
MOVS R2, #0x4  // pin to toggle

    .balign 16
loop:
STR R2, [R1,#0]
SUB R0, R0, #6
BPL loop
```

This routine is called with a single parameter passed in the register R0, representing the transmission time in cycles. It makes use of the memory-mapped NOTP0 register, which is used to invert the state of a pin. Each bit of the register corresponds to a single pin, and writing "1" to a bit toggles the associated pin. The loop repeatedly writes to this register to generate a square wave, while keeping track of the remaining time.

As described in the Preparation, the "BPL loop" jump may take additional time if it crosses the boundary of a 16-byte block. In order to avoid this, the loop is preceded by an assembler directive that aligns it to the start of a block.

This example loop takes six cycles to toggle the pin (2 for STR, 1 for SUB and 3 for BPL). The square wave it transmits therefore has a 12 cycle-long period, and its frequency is:

$$f = \frac{48\,\text{MHz}}{2 \cdot 6} = 4\,\text{MHz}$$

The routines used by the frequency-shift keying modulation code are similar to the example loop given above, with additional NOPs to increase the iteration time and therefore transmit at a particular frequency.

I chose the two frequencies f_0 and f_1, representing "0" and "1" bits, to minimise the difference of the received harmonics. Since the performance of the antenna I used is optimal at 90 MHz, I constrained the harmonics to the 80 MHz to 100 MHz range.

The chosen frequencies are $f_0 = 3\,\text{MHz}$ (generated by an 8-cycle loop) and $f_1 = 2.67\,\text{MHz}$ (a 9-cycle loop). The receiver uses their 31st and 35th harmonics, which are 93 MHz and 93.33 MHz respectively.

3.4.2 Phase-shift keying

The phase-shift modulator takes two inputs: an array of bits that should be transmitted, terminated by an end-marker, and the length of each symbol in cycles. It transmits a square wave at a constant frequency, with the phase of the wave encoding the current bit. Since it uses differential phase-shift keying, this means that the phase of the wave is inverted at the start of every 1-bit, and kept unchanged at the start of a 0-bit. The implementation of the modulator can be described by the pseudocode given in Algorithm 2.

Algorithm 2 Phase-shift keying modulator

```
transmit-psk(data, symbol_len):
    for each b in data:
        if b is end-marker: break
        if b == '0':
            toggle pin
        else:
            wait
        transmit square wave for symbol_len cycles
```

Most of the output wave is generated in a loop that transmits a square wave at a fixed frequency, which functions identically to the ones used for frequency-shift keying. The loop takes ten cycles and therefore transmits at $48\,\text{MHz}/20 = 2.4\,\text{MHz}$.

The remaining logic needs to be timed precisely, so that the transmission loop starts at the same phase offset for each bit. To make sure that this is the case, the entire modulator is implemented in assembly. This allows the length of the code path corresponding to loading a 0-bit (when the phase should not be changed) to be exactly divisible by the length of the transmission loop, ensuring correct alignment.

The logic that fetches the next bit and initialises the transmission loop also includes several instructions that toggle the output pin at correct times, in order to keep generating the carrier wave. If a 1-bit is loaded, one of these instructions is skipped, which inverts the phase of the wave.

In practice, getting the timings of all instructions to align correctly was not trivial. This is partly due to the code being longer than 16 bytes, which means that some jumps must cross a 16-byte boundary and take additional cycles to execute. However, even after I adjusted the timings to account for this effect, some code paths took longer than the documentation predicts.

To get a modulator that performs well, I manually adjusted the delays and measured the phase shifts after outputting a 0-bit and a 1-bit. The best set of delays I found does not produce phase shifts of exactly 0 and π. The delays are, however, close enough to these values to allow successful demodulation.

3.4.3 Spread-spectrum modulation

The direct-sequence spread spectrum modulator takes three inputs:

- an array of bits representing the data to be transmitted, terminated by an end-marker,
- an array of bits representing the chipping sequence, and
- the length of the chipping sequence.

The chipping sequence is generated by a linear-feedback shift register. This is implemented in C++ and done as a preprocessing step, before the modulator is started. It could be computed during transmission, but the overhead for calculating the next bit is greater than a memory lookup, so decoupling these steps both increases the carrier frequency and makes the implementation more modular.

The implementation of the spread-spectrum modulator can be summarised by the pseudocode in Algorithm 3.

Algorithm 3 Direct-sequence spread spectrum modulator

```
spread-spectrum(data, chip, chip_len):
    d_i = c_i = 0
    while data[d_i] is not end-marker:
        pin = data[d_i] xor chip[c_i]
        c_i += 1
        if c_i == chip_len:
            c_i = 0
            d_i += 1
```

The spread-spectrum demodulator, unlike the one used for phase-shift keying, cannot tolerate timing errors. Consequently, all possible code paths between consecutive writes

to the output pin must take exactly the same time. This is, as described in the previous section and the Preparation, not trivial to accomplish. In order to reduce possible sources of timing errors and make the implementation more robust, I chose to implement the modulator in such a way that the total number of code paths is as small as possible.

This choice sacrifices some performance (and therefore reduces the carrier frequency). For example, the current data bit is fetched from memory every iteration, instead of caching it in a register, which increases the length of the loop by two cycles.

The benefit of this trade-off is a simple implementation, with only two possible paths: one when moving to the next data bit, and another in other iterations. Matching their lengths was straightforward, and the final length of a single iteration is 20 cycles (giving a 1.2 MHz carrier wave).

3.5 Receiver overview

The receiver module is comprised of three separate GNURadio projects, each demodulating the signal received from one of the three transmitter modules. They share some common features, which I will describe in this section.

The source node of the signal processing graph is a "USRP Source" block, which must be configured with a centre frequency and bandwidth for which it will output samples. If the centre frequency is set to the value the rest of the demodulator expects (label it f_c), the output will contain significant noise around this frequency, due to local oscillator leakage.

To avoid this, I configured the source block to a frequency $f_c + \Delta_f$, which is outside the frequency range we are interested in, with the bandwidth wide enough to cover the desired frequency range. The output of the source is then multiplied by $e^{-2\pi j \Delta_f t}$, shifting the spectrum to center it on f_c. This is then low-pass filtered and downsampled to discard the unneeded parts of the spectrum.

The intermediate processing that consumes this signal varies between the three demodulators, and will be further described in the next section. They all output a sequence of bits obtained from the signal, used as the input to the common final stage.

The final stage is a custom signal processing block that recognises the frame start and end marks, and discards all data that lies outside. It also decodes the frame contents and outputs the decoded data to a sink node, which writes it to a file.

3.6 Demodulation

In the following subsections, I will describe the algorithms used to demodulate the incoming signal, starting from the output of the first filtering and downsampling stage. I will also highlight all signal processing blocks I implemented, and give a brief description of each.

3.6.1 Frequency-shift keying

The frequency-shift keying demodulator splits the data stream into two paths, one centered on the frequency used to encode "0" and the other on the frequency used to encode "1". They are separately filtered and downsampled, as described in the previous section.

A block diagram of the frequency-shift keying demodulator is shown in 3.2.

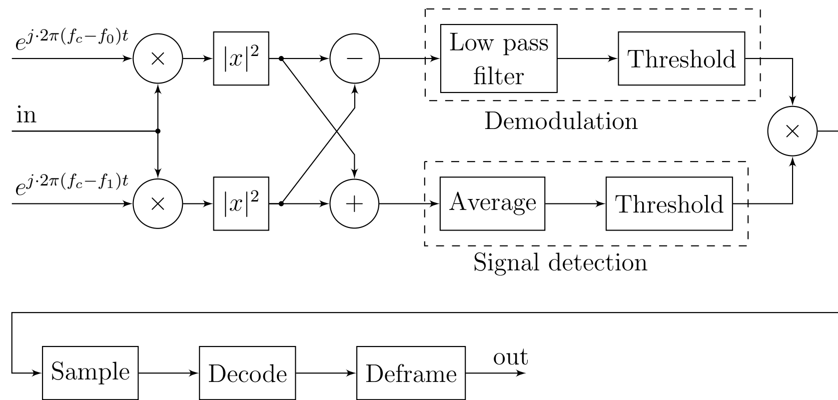


Figure 3.2: Block diagram of the frequency-shift keying demodulator

Signal detection

To detect that the microcontroller is transmitting, I use the sum of magnitudes of the two low-pass filtered signals. The sum is averaged over a short time period to reduce noise. The signal detector then converts this average into a binary signal by comparing it with a predefined threshold, and outputting "1" if the signal is strong enough.

Demodulation

The frequency-shift keying demodulator works by computing the difference between the magnitudes of the two signals. This difference is low-pass filtered, and the filtered signal is converted into binary data.

If there was no noise, it would be sufficient to digitise the signal by setting it to "0" when the difference is negative (stronger signal at f_0), and to "1" when it is positive. However, the noise is large enough that the difference may contain zero-crossings. To avoid brief changes in the output caused by the zero-crossings, the output value is not changed unless the difference is greater than a threshold d. In particular, if the current output is "0", it will only be changed after the difference crosses d, and if it is "1", it will be changed after crossing $-d$.

Sampling

In order to eliminate data recovered from noise while the transmitter is inactive, the outputs of the signal detector and demodulator are multiplied. The resulting binary signal is passed on as an input to a custom GNURadio block, which samples the signal and outputs the resulting bit sequence.

The sampling block is configured with a single parameter: the length of a single symbol in samples. It synchronises an internal clock with the transmitter on edges of the input signal, and outputs a bit every time the clock indicates the input is the midpoint of a symbol. In order to avoid losing synchronisation due to an edge caused by noise, the clock recovery algorithm filters out transitions that do not occur very near the expected positions (an integer number of symbol periods from the last transition).

3.6.2 Phase-shift keying

After the signal is filtered and downsampled, the phase-shift keying demodulation splits into two separate paths: detecting whether a transmission is in progress and recovering phase information. A block diagram showing the structure of the demodulator is given in Figure 3.3.

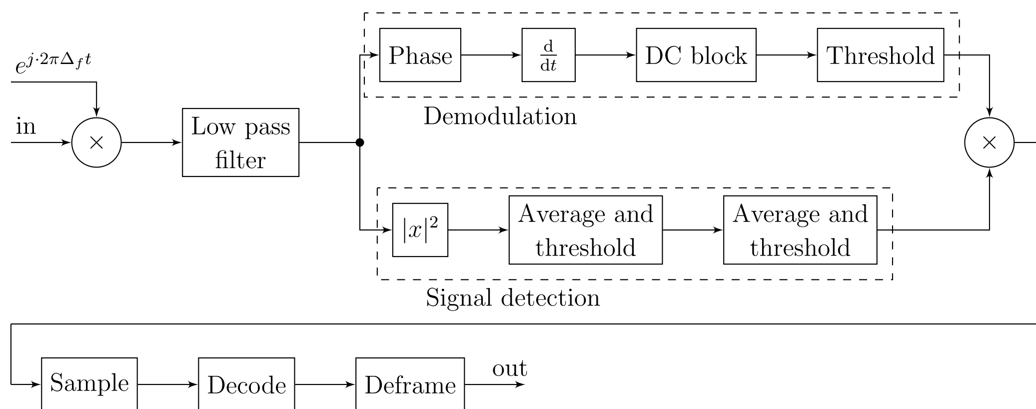


Figure 3.3: Block diagram of the phase-shift keying demodulator

Signal detection

To detect transmissions, I compute an average of the signal magnitude over a short time period. This average is then converted into a binary signal by comparing it to a predefined threshold. To avoid noisy output at the beginning of a transmission, when the average is near this value, this is passed through another averaging filter and compared to a second threshold. The final output of the signal detector is therefore “1” only when the average power exceeds the detection limit for several samples in a row.

Phase extraction

Phase information can be directly recovered from the I/Q samples by taking the argument of the complex-valued samples. To detect phase transitions, I implemented a custom GNURadio block that takes this phase information and computes the derivative. Its input is sampled above the Nyquist limit, so the phase cannot change by more than π between two samples. The derivative is therefore limited to the $[-\pi, \pi]$ interval, and the block wraps all output values to this range.

If the received signal was shifted so that the carrier is exactly at the zero frequency (ignoring noise), we could describe it (before differentiation) as

$$s(t) = Ae^{j(\phi_0 + \delta(t))}$$

where A is the amplitude, ϕ_0 an unknown phase offset between the receiver and transmitter, and $\delta(t)$ the phase shift at time t. In this case, the output of the differentiating block would be $\frac{\mathrm{d}}{\mathrm{d}t}\delta(t)$, which is π at the start of a 1-bit and zero otherwise.

In practice, this is not the case, because there is uncertainty in the microcontroller's clock rate. This means that the received signal would still contain a sinusoidal component with frequency Δ_f, equal to the difference between the frequencies at which the transmitter and the receiver are operating:

$$s(t) = Ae^{j(\phi_0 + \Delta_f t + \delta(t))}$$

The derivative of the phase of this signal is $\Delta_f + \frac{\mathrm{d}\delta(t)}{\mathrm{d}t}$, which has a DC component proportional to the frequency error. To eliminate this, the output of the differentiator is passed through a DC blocking filter. It is then thresholded to convert it into a binary signal, giving an output that is equal to 1 at the start of 1-bits and zero otherwise.

Sampling

The outputs of the two components (signal detector and phase demodulator) are combined by multiplying them. This is done in order to eliminate data that is obtained by demodulating noise when there is no ongoing transmission.

This signal is then passed to a custom block that samples it to recover the "raw" received bit sequence (prior to deframing and decoding). It takes the length of a single bit in samples as a parameter, and constructs a replica of the transmitter clock with the given frequency. Every time it receives a phase transition on the input, the sampling block re-synchronises this clock replica with the signal, since it corresponds to the beginning of a transmitted bit.

The sampling block outputs a single bit for every edge of the replica clock. The output is "1" if there was a phase transition at the last clock edge, and "0" otherwise.

The output of the sampling block is passed on to the deframing and decoding block, and finally written to a file for later examination.

3.6.3 Spread-spectrum modulation

After filtering and downsampling, the direct-sequence spread spectrum signal can be described as

$$s(t) = Ad(t)c(t) + n(t)$$

Here, A is the signal amplitude, $d(t) \in \{-1, 1\}$ the data, $c(t) \in \{-1, 1\}$ the periodic chipping (pseudonoise) sequence, and $n(t)$ a term describing the noise. To recover the data, this signal should be multiplied by a replica of $c(t)$, giving the following:

$$s(t)c(t) = Ad(t)c(t)^2 + n(t)c(t) = Ad(t) + n(t)$$

Here, the $c(t)^2$ term disappears because it is always equal to 1. Assuming that the noise is Gaussian, multiplying it by the chipping sequence does not change its characteristics.

Therefore, the main problem I had to solve when implementing the spread-spectrum demodulator was generating a replica of the chipping sequence synchronised with the transmitter. This process is split into two stages:

- acquisition, where the receiver has no information about the chipping sequence and is searching for a rough estimate, and
- tracking, which keeps a replica of the chipping sequence "locked" to the signal and improves the accuracy.

Acquisition – prototype

I implemented two different acquisition modules: a simple module as a part of the MATLAB demodulator prototype, and a more complex one as the first stage of a custom GNURadio demodulation block. They are both based on the same principle of searching for an offset at which the cross-correlation between the signal and a generated chipping sequence is high.

The prototype demodulator assumes that the frequency error in the transmitter is not significant, and does not attempt to compensate for this.

It first computes the cross-correlation of the signal and a single period of the chipping sequence. This is then used to find the current transmitter offset in the chipping sequence for each input bit by taking the difference between that bit's position and the last cross-correlation peak. The peak is defined as the maximal value in the past $L + \epsilon$ samples, where L is the length of the chipping sequence, and ϵ some small integer.

After computing the cross-correlation, we can use a naive algorithm to find the maxima by iterating through the previous $L + \epsilon$ values for each sample. This algorithm would have a time complexity of $\mathcal{O}(NL)$, where N is the number of samples in the signal. Since N is on the order of tens of millions, and L on the order of thousands, this would take unacceptably long. To improve the running time, I implemented the following dynamic programming algorithm to find the maxima, with a time complexity of $\mathcal{O}(N)$:

Let a be the (0-indexed) array of values for which we need to compute the maxima, and l be the length of the interval over which they should be taken. Divide a into segments of length d. Define an array pre_i as the maximum of elements from the beginning of the interval containing a_i up to a_i, and $post_i$ as the maximum of elements from a_i to the end of the interval. They can be computed as:

$$pre_i = \begin{cases} a_i, & i \bmod d = 0 \\ \max(pre_{i-1}, a_i), & \text{otherwise} \end{cases}$$

$$post_i = \begin{cases} a_i, & i \bmod d = d - 1 \\ \max(a_i, post_{i+1}), & \text{otherwise} \end{cases}$$

The maximum of $a_{i-d+1}, a_{i-d+2}, \ldots, a_i$ can be split into elements from a_{i-d+1} to the next interval boundary, and the elements from that boundary to a_i. Therefore, it can be computed as $\max(pre_i, post_{i-d+1})$.

Acquisition – online

The GNURadio block that I implemented for spread-spectrum demodulation works in a similar way to the prototype described above, but does not assume that the transmitter frequency is exactly known. Since testing all frequencies for each position in the input stream would be too time-consuming for online processing, it divides the stream into segments and tests one candidate frequency per segment.

For each candidate frequency, it consumes samples until the total length is equal to several repetitions of the chipping sequence. Since all transmissions start with a long sequence of 0-bits, there is no data modulated on the sequence in this period. Therefore, computing a correlation over multiple periods will improve the signal-to-noise ratio.

After collecting the desired number of samples, it computes a circular cross-correlation of the signal and chipping sequence. This is done in $\mathcal{O}(N \log N)$, using the identity

$$a \star b = \mathcal{F}^{-1}\{\mathcal{F}\{a\}^* \cdot \mathcal{F}\{b\}\}$$

To compute the Fourier transform, I used the implementation provided as a part of the GNURadio libraries. In particular, I used the `gr::fft::fft_complex` class, which is instantiated to provide an object that computes the Fourier transform of a fixed-size input. To eliminate the overhead due to initialising these objects, I create an instance for each FFT size the block uses as a part of the initialisation, and reuse them instead of constructing fresh ones.

After computing the cross-correlation, the maximal value is examined as a candidate for the offset. It is possible that there is no ongoing transmission, or that the frequency candidate is incorrect, in which case the candidate should be rejected. As an acceptance criterion, I compare it with the average cross-correlation, and accept it if the average is exceeded by a pre-defined threshold.

After a pair of an offset and frequency is accepted, the demodulation block moves to the tracking stage.

Tracking

The tracking stage is an implementation of a delay-locked loop, which keeps track of the transmitted chipping sequence and improves the accuracy of the offset. It maintains three copies of the chipping sequence:

- a **prompt** replica, using the current estimate of the offset,
- an **early** replica, with the offset shifted by one half of a chip, and
- a **late** replica, with the offset shifted by one half of a chip in the other direction.

The prompt replica is used for demodulation: the block's output is calculated by multiplying it with the input signal.

The early and late chipping sequences are used to calculate the offset error. Let E be the integral of samples multiplied by the early replica, and L be the same for the late replica. The offset error is then computed as

$$\Delta_\phi = \frac{1}{2}\frac{|E|^2 - |L|^2}{|E|^2 + |L|^2}$$

This error is then used to update the offset used for the next chipping sequence period.

For a complete demodulator, the tracking loop would need a method of detecting when the offset error is too large, in which case it would switch back to acquisition. As my implementation of the tracking loop loses lock too quickly to be of practical use, I did not implement this. The tracking loop runs for a pre-defined number of iterations instead.

Chapter 4

Evaluation

4.1 Bit error rate measurement

In order to evaluate the performance of the transmitter, I measured the bit error rate of the frequency-shift and phase-shift keying modulators at different distances and bit rates. In this section, I will describe the setup used for measuring. I will then give the results of the measurements, as well as the calculated error rates.

4.1.1 Setup

I measured the bit error rate by transmitting a frame containing random data and recording the demodulated output. The data was generated as the first 400 bytes of a linear-feedback shift register sequence. I chose this frame size since it is the largest value for which the entire frame (after encoding) can fit into the memory available on the microcontroller.

For measurements at shorter ranges, the development board containing the microcontroller was placed on a wooden table, without anything attached to the pins. It is important to note that this influences the signal strength significantly, due to the capacitively coupled table surface acting as an antenna – if the microcontroller is suspended in air, the signal strength is too low to transmit at usable data rates.

The microcontroller was powered through the USB interface on the development board. The power was supplied from a laptop over an approximately 1 m long USB cable.

For the phase-shift keying transmitter, I also recorded data at a longer distance (three metres), with the microcontroller plugged into a breadboard. This is meant to emulate a scenario where there is some wiring in the backdoored device that we can safely use as an antenna and improve the transmitter gain.

Antenna

To receive the signal, I used a 165 cm folded-dipole antenna. Its resonant frequency is approximately 90.9 MHz, which is near the 93 MHz and 93.33 MHz pair used to encode the frequency-shift keyed data.

The 37th harmonic of the 2.4 MHz phase-shift keying carrier is closest to the resonant frequency, at 88.8 MHz. This frequency is, however, in the FM radio broadcast range, and a radio station is transmitting a signal that is significantly stronger than the one produced by the microcontroller. The 39th harmonic is similarly occupied by a second radio station. To avoid this issue, I chose to tune the receiver to the unoccupied 41st harmonic, at 98.4 MHz.

It should be noted that using a directional antenna instead of the omnidirectional folded dipole would give a higher gain and therefore increase the range at which a usable signal can be received. For example, for a three-element Yagi-Uda antenna, we can expect a gain of approximately 7 dB. Assuming that the transmitter is in the far field, the signal power in the far field is inversely proportional to the square of the distance, this would correspond to a slightly more than doubled range. This assumption might not necessarily hold – approximating the boundary between the near and far field as 1/6 of the signal wavelength gives around 50 cm, which is quite close to some of the distances used for measurement.

Corrupted frames

At higher baud rates, the error rate is high enough that some frames contain an error in the frame header. If the last header bit is corrupted, the raw frame contents will be offset by a single bit. This causes the bits to be wrongly grouped when decoding the Manchester-coded data and will effectively scramble the entire frame.

The corrupted frames have an expected error rate of 50%, since the frame contents are random. For baud rates where the bit error rate is otherwise low, these frames are clear outliers. Including these data points when calculating the average error rate would give a skewed result, since the single bit error corrupts the rest of the frame. Therefore, I have not included the corrupted frames in the calculation where they can be cleanly separated from the rest of the data.

4.1.2 Frequency-shift keying

For the frequency-shift keying modulator, I measured the bit error rate at a distance of 50 cm from the receiver antenna. The bit error rates of individual measurements are given in Figure 4.1.

At higher baud rates, the additional edges introduced into the demodulated signal result in synchronisation errors in the receiver clock. This manifests as frames that are mostly without error up to a point, after which they become scrambled due to the receiver skipping or duplicating a bit.

All frames were transmitted without errors for baud rates of up to 50 bits per second. At 100 bits per second, most frames do not contain any errors, but several have a corrupted segment and the average error rate is 0.05 ± 0.1.

At 200 bits per second, the frames become corrupted almost immediately. The average error rate is 0.42 ± 0.03, making the transmitter effectively unusable.

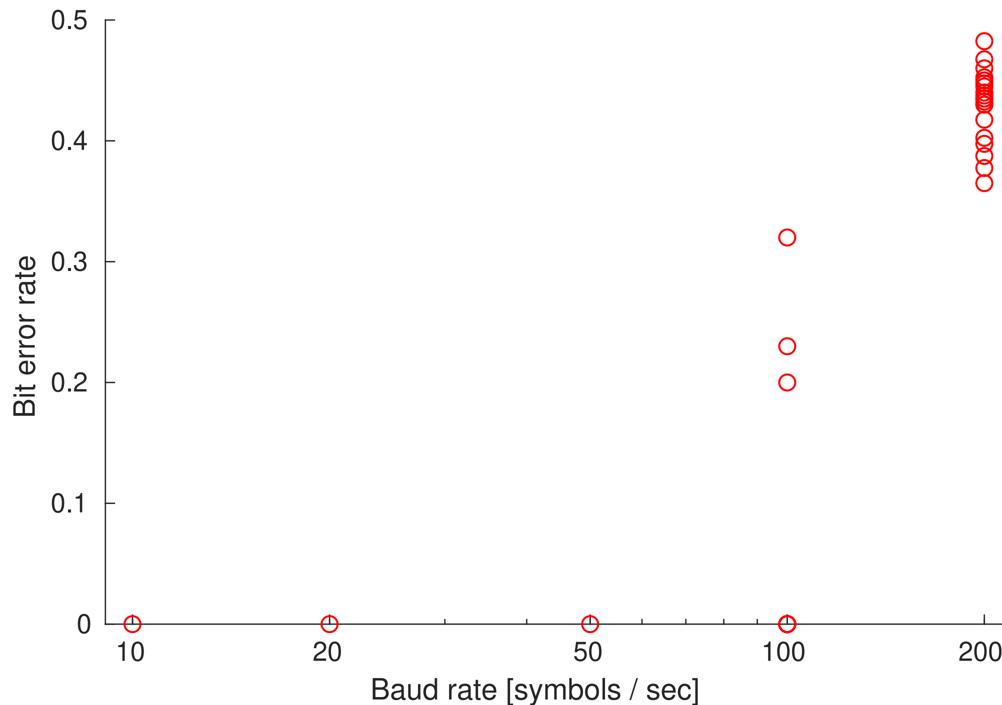


Figure 4.1: Plot of bit error rate of the frequency-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 50 cm from the antenna

4.1.3 Phase-shift keying

I measured the bit error rate of the phase-shift keying demodulator at three distances: 50 cm, 1 m and 3 m. The first two measurements were made with the microcontroller placed on a wooden table, and the third one while plugged into a breadboard.

For each distance, I made measurements at baud rates starting at 10 symbols per second. I increased the baud rate until the error rate was large enough to make the transmitter unusable for practical communication.

I recorded 20 frames for each baud rate, except 10 and 20 symbols per second, where I only made ten recordings each (since transmitting a single 400 byte long frame at these rates takes over a minute).

Results – 50 cm

At a distance of 50 cm, the highest baud rate at which I made measurements is 1000 symbols per second. The error rate of individual measurements can be seen in Figure 4.2.

At up to 200 bits per second, all frames were transmitted without error. The measured bit error rate at higher baud rates is given in Table 4.1.

At the highest baud rate, 10% of the received frames were corrupted and therefore excluded from this estimate.

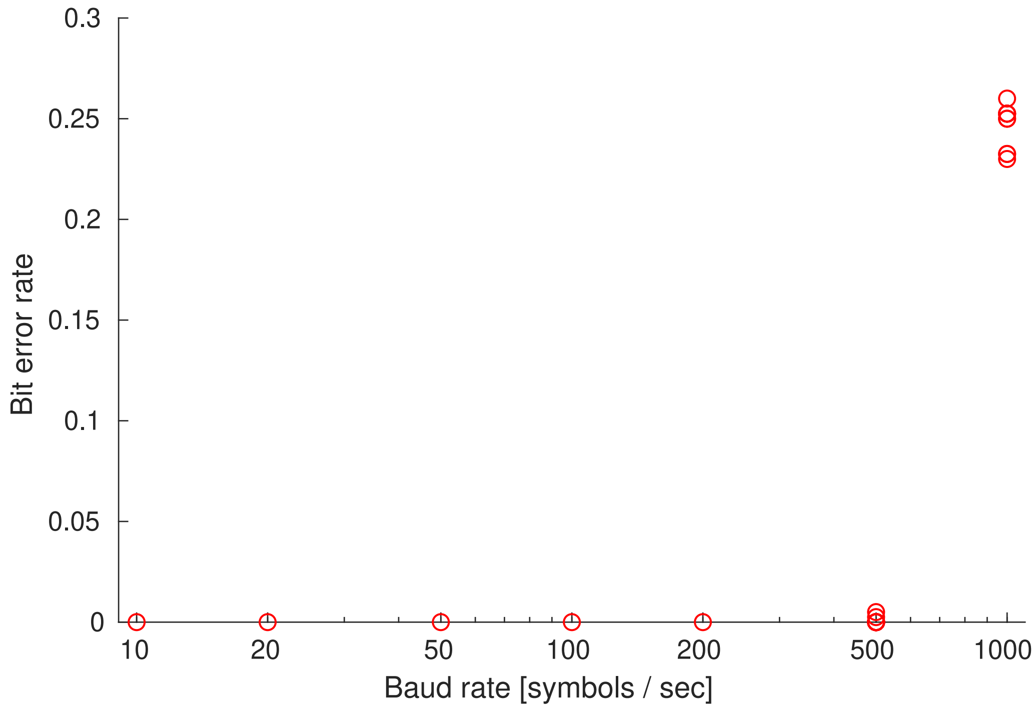


Figure 4.2: Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 50 cm from the antenna

Baud rate [symbols/sec]	Bit error rate
500	$(3.75 \pm 12) \cdot 10^{-4}$
1000	0.26 ± 0.1

Table 4.1: Bit error rate measurements for the phase-shift keying modulator at a distance of 50 cm

Results – 1 m

At 1 m, the highest baud rate at which I made measurements is 500 bits per second. The error rates of individual measurements can be seen in Figure 4.3.

At the two lowest baud rates, the recordings do not contain any errors. The average bit error rates at higher baud rates are given in Table 4.2.

At the highest baud rate, 20% of the frames were corrupted and therefore excluded.

Results – 3 m

At 3 m, with the microcontroller plugged into a breadboard, the highest baud rate at which I made measurements is 2000 bits per second. The error rates of individual measurements can be seen in Figure 4.4.

At baud rates up to 100 bits per second, all frames were transmitted without error. The bit error rates at higher baud rates are given in Table 4.3.

At 2000 bits per second, 15% of the frames were dropped completely. This is caused by the frame preamble being shorter than the time over which the signal detector computes

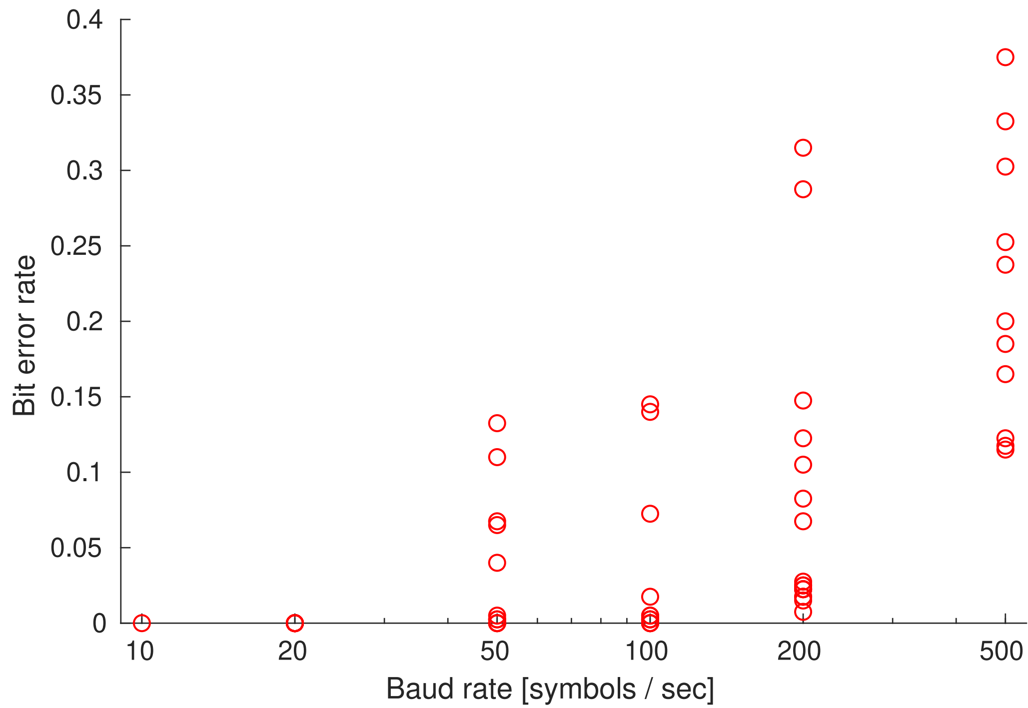


Figure 4.3: Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 1 m from the antenna

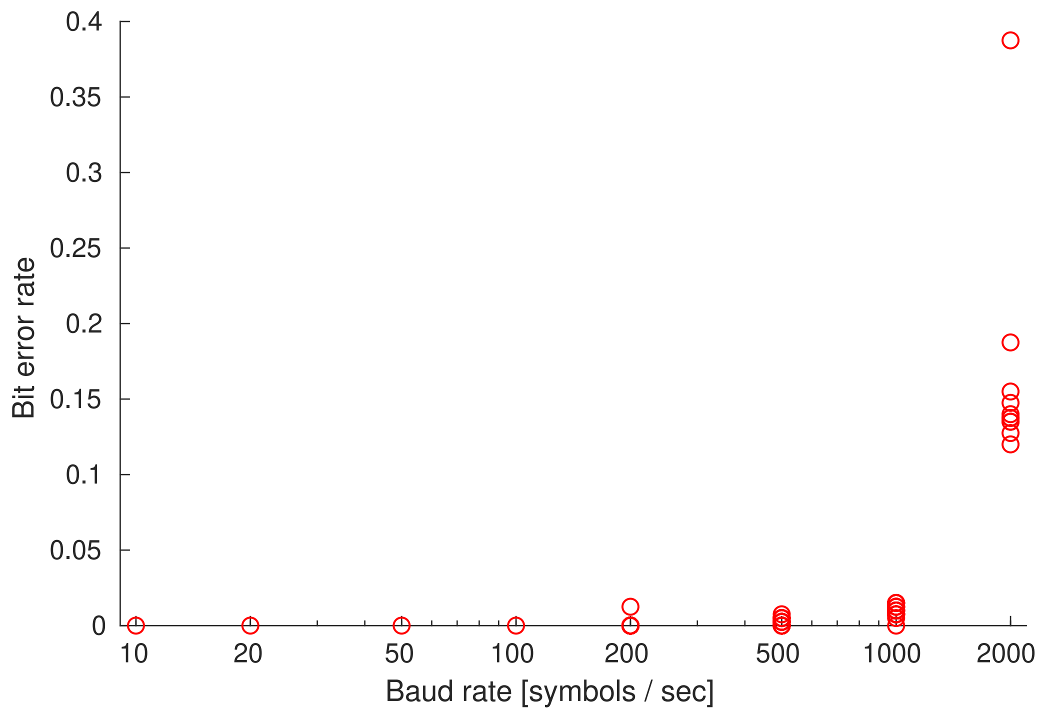


Figure 4.4: Plot of bit error rate of the phase-shift keying demodulator for individual frames against the baud rate, with the transmitter placed at a distance of 3 m from the antenna

Baud rate [symbols/sec]	Bit error rate
50	0.03 ± 0.05
100	0.03 ± 0.05
200	0.07 ± 0.09
500	0.35 ± 0.20

Table 4.2: Bit error rate measurements for the phase-shift keying modulator at a distance of 1 m

Baud rate [symbols/sec]	Bit error rate
200	$(0.6 \pm 2.8) \cdot 10^{-3}$
500	$(2.5 \pm 2.3) \cdot 10^{-3}$
1000	0.01 ± 0.04
2000	0.17 ± 0.08

Table 4.3: Bit error rate measurements for the phase-shift keying modulator at a distance of 3 m

averages. The average only crosses the detection threshold after the frame header is transmitted, and the receiver never "sees" the preamble.

This issue could be solved by increasing the length of the frame header. However, the length of the header is limited by the memory available on the microcontoller, since the entire frame must be preprocessed before starting transmission. If we wanted to have a long preamble at high bit rates, a more complex implementation is necessary, where the header is generated during transmission instead of being loaded from memory.

4.2 Direct-sequence spread spectrum

In this section, I will first give plots of the output of the prototype direct-sequence spread spectrum demodulator on artificial data for several signal-to-noise ratios.

Since my implementation of the tracking loop does not maintain lock for long enough to demodulate a significant amount of data, I could not make error rate measurements such as the ones given in the previous section. To demonstrate that the transmitted signal contains the expected pseudonoise sequence, I will give plots showing the cross-correlation of the received signal and this sequence. I will also give output logs of the acquisition block, to demonstrate that it successfully detects these peaks.

4.2.1 Offline prototype

Artificial data

In order to test the correctness of the direct-sequence spread spectrum demodulator, I wrote a test MATLAB program that generates an artificial signal and uses it as an input to the demodulator. The program takes four parameters:

1. length of the signal in samples,
2. length of a single chip in samples,
3. multiplicative frequency correction factor, to emulate an error in the transmitter frequency, and
4. the desired signal-to-noise ratio.

The signal is formed by generating a chipping sequence, with the symbol length scaled by the correction factor. Gaussian noise with amplitude $\sqrt{S^{-1}}$, where S is the signal-to-noise ratio, is then added to the signal.

This is passed on to the prototype, which is configured with the original chipping frequency (without the multiplicative factor).

Example plots of the output signal, before and after thresholding to convert it into binary data, are displayed in Figure 4.5. All tests use the same chipping sequence: a 1024-bit long linear-feedback shift register sequence.

Recorded signal

The prototype demodulator is not capable of demodulating a recording of "real" data transmitted by the microcontroller. It is, however, sufficient to demonstrate that the expected chipping sequence is present in the signal.

For this test, I recorded a sample spread-spectrum transmission with the microcontroller 50 cm away from the antenna. I then computed the cross-correlation of the signal with the chipping sequence to detect it.

The signal was sampled at 10 MHz. Since the chipping sequence frequency is 1.2 MHz, the length of a single chip is $10\,\mathrm{MHz}/(2 \cdot 1.2\,\mathrm{MHz}) \approx 4.17$ samples. The chipping sequence used for this test is 128 bits long, and we expect peaks every $128 \cdot 4.17 \approx 534$ samples.

In Figure 4.6a, the magnitude of the cross-correlation over the entire signal can be seen. There are two segments that have a visibly higher cross-correlation than the rest of the signal. The first one is caused by a transmitter reset – while recording the data, the transmitter was configured to begin broadcasting immediately after a reset. Pressing the "reset" button on the development board triggers two successive resets, interrupting the first broadcast.

The second, longer interval is due to the entire transmitted frame. In Figure 4.6b, a zoomed-in view shows the cross-correlation peaks occurring with the expected period.

4.2.2 Online processing

To demonstrate that the online demodulation block correctly implements signal acquisition, I implemented some logging functionality to print the acquired offset and frequency.

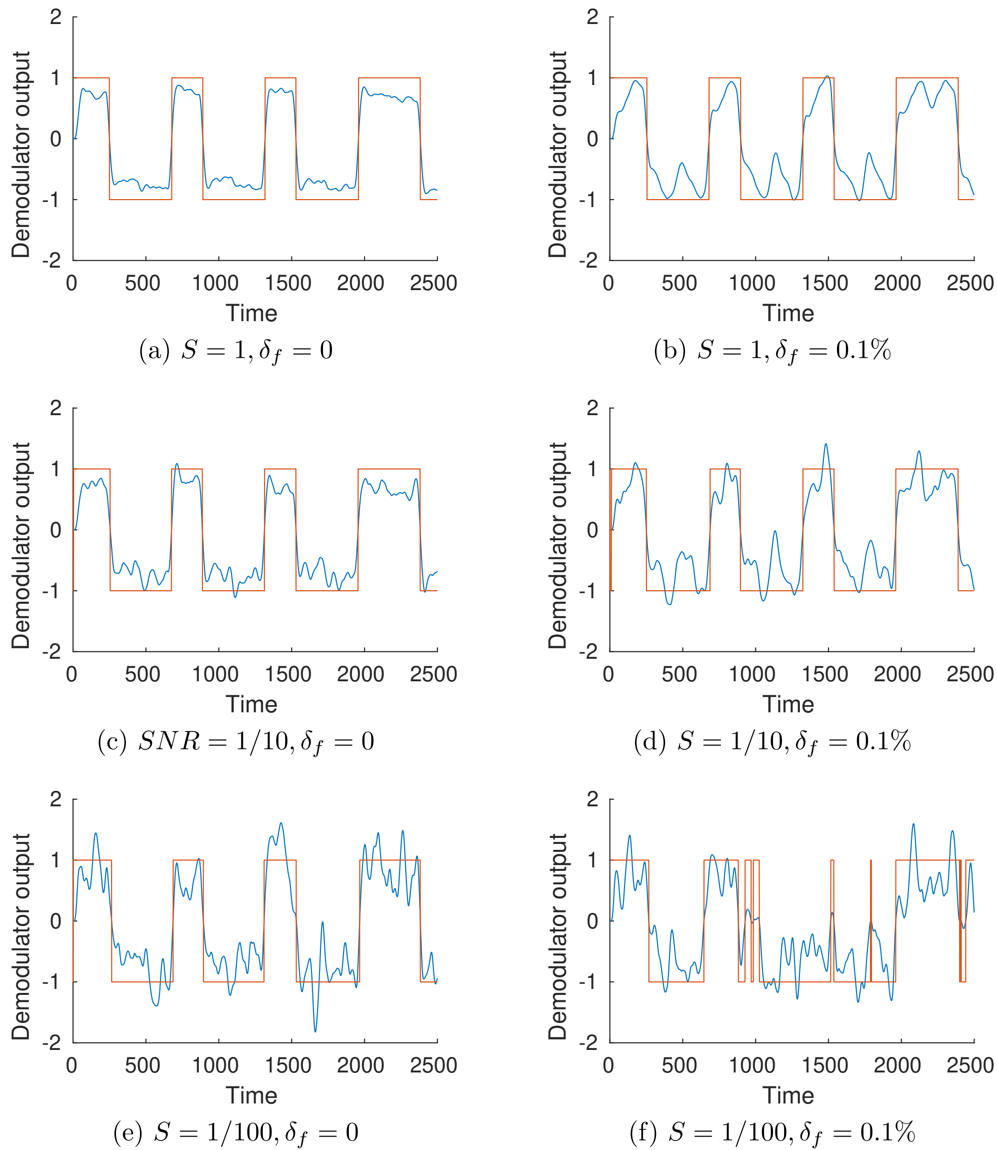
Since the tracking loop does not perform well enough to provide a decision to switch back to acquisition, the block was configured to make this transition after ten chipping sequence periods have passed. An excerpt from the log printed with the input from the previous subsection is shown below, showing the output at the beginning of the transmitted signal. Some line breaks have been added manually for clarity:

```
current sample: 15000000

current sample: 16000000

Switching to tracking, pos = 103.49,
  mult_idx = 1 (frequency error 1.000)
delta = 0.2835, pow = 0.0098
pos = 115.49, size = 128, ttl = 10
delta = 0.0073, pow = 0.2856
pos = 127.43, size = 128, ttl = 9
delta = -0.4491, pow = 0.0509
pos = 127.90, size = 128, ttl = 8
delta = -0.0804, pow = 0.0355
pos = 12.01, size = 128, ttl = 7
delta = 0.0869, pow = 0.0368
pos = 24.03, size = 128, ttl = 6
delta = 0.0280, pow = 0.0533
pos = 36.01, size = 128, ttl = 5
delta = 0.0009, pow = 0.0395
pos = 48.00, size = 128, ttl = 4
delta = -0.0456, pow = 0.0387
pos = 60.00, size = 128, ttl = 3
delta = 0.0980, pow = 0.0091
pos = 72.01, size = 128, ttl = 2
delta = -0.0314, pow = 0.0644
pos = 83.99, size = 128, ttl = 1
switched to acquisition
Switching to tracking, pos = 91.50,
  mult_idx = 1 (frequency error 1.000)
delta = -0.0009, pow = 0.0289
pos = 103.50, size = 128, ttl = 10
delta = 0.0922, pow = 0.0132
pos = 115.50, size = 128, ttl = 9
delta = -0.0193, pow = 0.1298
pos = 127.48, size = 128, ttl = 8
delta = 0.2659, pow = 1.0000
pos = 127.96, size = 128, ttl = 7
delta = 0.2023, pow = 0.0671
pos = 11.90, size = 128, ttl = 6
delta = -0.0413, pow = 0.0244
pos = 23.85, size = 128, ttl = 5
delta = -0.0318, pow = 0.0008
pos = 35.86, size = 128, ttl = 4
delta = -0.1129, pow = 0.0264
pos = 47.87, size = 128, ttl = 3
delta = 0.3431, pow = 0.0061
pos = 59.89, size = 128, ttl = 2
delta = -0.2846, pow = 0.0026
pos = 71.81, size = 128, ttl = 1
switched to acquisition

Switching to tracking, pos = 111.49,
  mult_idx = 1 (frequency error 1.000)
delta = 0.0479, pow = 0.1526
pos = 123.49, size = 128, ttl = 10
```

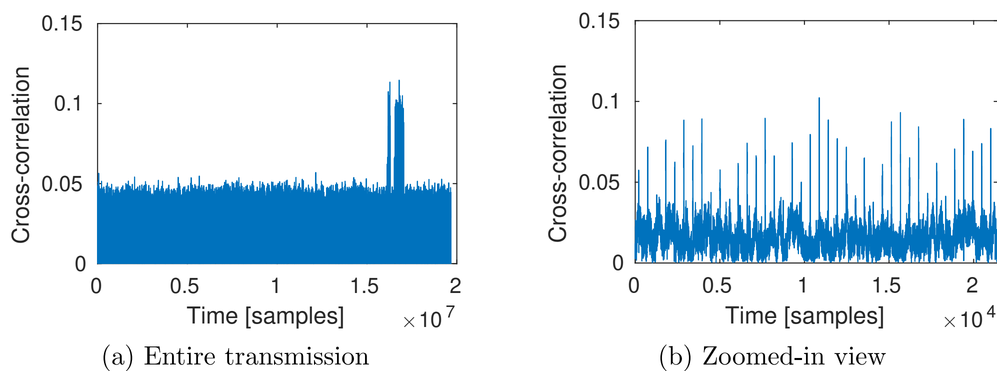


(a) $S = 1, \delta_f = 0$ (b) $S = 1, \delta_f = 0.1\%$

(c) $SNR = 1/10, \delta_f = 0$ (d) $S = 1/10, \delta_f = 0.1\%$

(e) $S = 1/100, \delta_f = 0$ (f) $S = 1/100, \delta_f = 0.1\%$

Figure 4.5: Plots of direct-sequence spread spectrum demodulator output before thresholding (in blue) and after thresholding (in orange), for several values of signal-to-noise ratio S and relative frequency error δ_f



(a) Entire transmission (b) Zoomed-in view

Figure 4.6: Plots of cross-correlation of the recorded direct-sequence spread spectrum signal and the chipping sequence

Chapter 5

Conclusion

In this dissertation, I have described the work that I had undertaken to demonstrate that a microcontroller running a suitable program can be used as a transmitter, without using any special electronics or an antenna. This shows that it should be possible to insert a firmware-level backdoor in an electronic device, and leak data over an electromagnetic covert channel.

I started by presenting the relevant theory and modulation techniques I implemented: frequency-shift keying, phase-shift keying and direct-sequence spread-spectrum modulation. I then described the hardware I used for the project and highlighted possible sources of error that needed to be accounted for.

I then described the implementation of the project, covering the structure of the transmitter and receiver, as well as the major implementation details and algorithms I used.

In order to evaluate the performance of the frequency-shift keying and phase-shift keying transmitters and demodulators, I gave the results of bit error rate measurements over several distances and multiple baud rates. While it was not possible to give such figures for the spread-spectrum demodulator, I gave a demonstration of a prototype, as well as evidence that the recorded signal contains the expected chipping sequence.

5.1 Results

I have implemented working frequency-shift keying and phase-shift keying transmitters and receivers, and demonstrated that they can be used to transmit data.

The frequency-shift keying modulator performance was evaluated at a distance of 50 cm from the receiver. It has a negligible error rate for baud rates up to 50 symbols per second, and a bit error rate of 5% at 100 symbols per second. The modulator was also tested at 200 symbols per second, but the error rate is too high to be usable, mainly due to synchronisation errors.

The phase-shift keying modulator was evaluated at 50 cm and 1 m without any attached hardware, and at 3 m while plugged into a breadboard. The highest baud rates at which the signal was successfully demodulated, and the corresponding bit error rates, are given in Table 5.1.

Distance [cm]	Baud rate [symbols/sec]	Bit error rate
50	500	$(3.75 \pm 12) \cdot 10^{-4}$
	1000	0.26 ± 0.1
100	200	0.07 ± 0.09
	500	0.35 ± 0.20
300	1000	0.01 ± 0.04
	2000	0.17 ± 0.08

Table 5.1: Bit error rate measurements for the phase-shift keying modulator at selected distances and baud rates

Bibliography

[1] Ross Anderson. *Security engineering.* John Wiley & Sons, 2008.

[2] Markus G Kuhn and Ross J Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *International Workshop on Information Hiding*, pages 124–142. Springer, 1998.

[3] Roger L Peterson, Rodger E Ziemer, and David E Borth. *Introduction to spread-spectrum communications*, volume 995. Prentice hall New Jersey, 1995.

[4] Raymond Pickholtz, Donald Schilling, and Laurence Milstein. Theory of spread-spectrum communications–a tutorial. *IEEE transactions on Communications*, 30(5):855–884, 1982.

[5] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1), 2010.

Appendix A

Source samples

A.1 Modulator

In this section, I will provide a sample of the modulation code used to transmit data. All routines implemented in assembly will be provided, as well as the C++ header declaring the function headers.

A.1.1 Header

```
#ifndef MODULATION_H
#define MODULATION_H

// fsk.s
// Transmit at the frequency encoding "0" for the specified time period
extern "C" int write_zero(unsigned);
// Transmit at the frequency encoding "1" for the specified time period
extern "C" int write_one(unsigned);
// Delay (do not transmit)
extern "C" int write_blank(unsigned);

// psk.s
extern "C" int write_psk(const unsigned char*);

// dsss.s
extern "C" void write_dsss(const unsigned char *data,
    const unsigned char *chip, int chip_len);
#endif
```

A.1.2 Frequency-shift keying

```
    .text
    .global write_zero
    .global write_one
```

```
    .global write_blank

write_zero:
    MOV R1, R0
    LDR R0, =0x50002300  // NOTP0
    MOVS R2, #0x4  // pin to toggle

    .balign 16
loop_zero:
    STR R2, [R0,#0]
    NOP
    NOP
    SUB R1, R1, #10
    BPL loop_zero

    BX LR

write_one:
    MOV R1, R0
    LDR R0, =0x50002300  // NOTP0
    MOVS R2, #0x4  // pin to toggle

    .balign 16
loop_one:
    STR R2, [R0,#0]
    NOP
    NOP
    NOP
    SUB R1, R1, #11
    BPL loop_one

    BX LR

write_blank:
    MOV R1, R0
    .balign 16
loop_blank:
    SUB R1, R1, #4
    BPL loop_blank
    BX LR
```

A.1.3 Phase-shift keying

```
    .text
```

```
    .global write_psk

    // Macros for output pin functionality
    // Using R3 as mem-mapped location of NOTP0 and R4 as pin specification

    .macro INIT_TOGGLE
    LDR R3, =0x50002300
    MOVS R4, #0x4
    .endm

    .macro TOGGLE_PIN
    STR R4, [R3, #0]
    .endm

write_psk:
    PUSH {R4}
    MOV R2, #0x84
    INIT_TOGGLE

    .balign 32
    NOP
    NOP
    NOP

main_loop:
    STR R2, [R3]

    // load next character, exit if '\2'
    LDRB R1, [R0]
    CMP R1, #2
    BEQ done

    // advance buffer pointer
    ADD R0, R0, #1

    TOGGLE_PIN

    // skip next toggle (phase shift) if outputting a '1'
    // in reality the phase shift when not skipping is larger
    CMP R1, #0
    BEQ skip_flip

    TOGGLE_PIN
skip_flip:
```

```
    // initialise clock counter for fixed-frequency transmission
    LDR R1, =24000
    // 4th cycle from SUB in loop

transmit_loop:
    SUB R1, R1, #10
    TOGGLE_PIN
    BPL transmit_loop

    B main_loop

done:
    POP {R4}
    BX LR
```

A.1.4 Direct-sequence spread spectrum modulation

```
    .text
    .global write_dsss

    .balign 16
write_dsss:
    PUSH {R4, R5, R6, R7}
    MOV R5, R2
    SUB R5, #1  // R5 = chip_len - 1
    MOV R2, R1
    LDR R6, =0x50001008  // PWORD for P0.2
    MOV R7, #0  // needed because LDRSB does not support imm offset

    MOV R3, R5

    NOP

main_loop:
    LDRSB R1, [R0, R7]  // load data
    LDRSB R4, [R2, R3]  // load chip

    // xor; if top bit is 1, this is the end-marker for data
    // EOR stores top bit in APSR.N, BMI branches on N == 1
    EOR R1, R1, R4
    BMI done

    STR R1, [R6, #0]
```

```
    SUB R3, #1
    BPL skip_advance
    MOV R3, R5
    ADD R0, #1
skip_advance:

    B main_loop

done:
    POP {R4, R5, R6, R7}
    BX LR
```

A.2 GNURadio blocks

In this section, I will provide some sample code from the GNURadio block used for demodulating spread spectrum signals. The sample code covers the main processing loop of the block, as well as the main body of the acquisition module and code used for computing cross-correlations. Some portions of the code, mainly providing logging, were ommited for brevity.

A.2.1 Main body

```
int
despreader_impl::work(int noutput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
{
    const gr_complex *in = (const gr_complex *) input_items[0];
    gr_complex *out = (gr_complex *) output_items[0];
    int n_out = 0;
    for(int i = 0; i < noutput_items; i++)
    {
        debug++;
        if(debug % 1000000 == 0) fprintf(stderr, "pos = %d\n", debug);

        if(state == ACQUISITION)
            process_sample_acquisition(in[i]);
        else if(state == TRACKING)
            process_sample_tracking(in[i]);
         out[i] = in[i] * (float)chip[(int)chip_pos];

        chip_pos += 1.0 / chip_samples;
        if(chip_pos >= chip.size())
```

```
        chip_pos -= chip.size();
    }
    consume_each(noutput_items);
    return noutput_items;
}
```

A.2.2 Acquisition

```
inline void despreader_impl::process_sample_acquisition(gr_complex val)
{
    history[hist_pos++] = val;

    int n = xcorr_length(freq_mult_i);
    if(hist_pos == n)
    {
        chip_xcorr(history, history);

        double avg = 0;
        for(int i = 0; i < n; i++)
            avg += norm(history[i]);
        avg /= n;

        int max_i = 0;
        for(int i = 1; i < n; i++)
            if(norm(history[i]) > norm(history[max_i]))
                max_i = i;

        if(history[max_i] > avg * ACQ_TRESH)
        {
            chip_pos = max_i;
            chip_pos -= trunc(chip_pos / chip.size()) * chip.size();
            fprintf(stderr, "Switching to tracking, [...]");
        }

        hist_pos = 0;
        freq_mult_i++;
        if(freq_mult_i == N_CHIP_LEN_STEPS)
            freq_mult_i = 0;
    }
}
```

A.2.3 Cross-correlation

```
void despreader_impl::chip_xcorr(gr_complex *f, gr_complex *out, bool dbg=false)
```

```
{
    int n = xcorr_length(freq_mult_i);
    gr::fft::fft_complex *Fi = F[freq_mult_i], *Gi = G[freq_mult_i],
        *Oi = O[freq_mult_i];

    gr_complex *F_in = Fi->get_inbuf(), *G_in = Gi->get_inbuf(),
        *O_in = Oi->get_inbuf();
    gr_complex *F_out = Fi->get_outbuf(), *G_out = Gi->get_outbuf(),
        *O_out = Oi->get_outbuf();

    float mult = freq_mult(freq_mult_i);
    float pos = 0;
    for(int i = 0; i < n; i++)
    {
        F_in[i] = f[i];
        G_in[i] = chip[(int)pos % chip.size()];
        pos += 1.0f / (chip_samples * mult);
    }

    Fi->execute();
    Gi->execute();

    for(int i = 0; i < n; i++)
    {
        O_in[i] = conj(F_out[i]) * G_out[i];
    }
    Oi->execute();

    for(int i = 0; i < n; i++)
        out[i] = O_out[i];
}
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Artificial EM Side Channel

D. Erdeljan, Trinity College
Originator: Prof. R. Anderson
13 October 2017

Project Supervisor: Dr M. Kuhn
Director of Studies: Dr S. Holden
Project Overseers: Dr R. Mortier & Dr A. Rice

Introduction

The aim of this project is to create a one-way communication channel on a microcontroller, that is designed to leak sensitive data to an outside observer (for example, transmitting the keystrokes from a modified keyboard controller). The scenario assumes that the attacker inserting the side channel has no control over the hardware the microcontroller is used in, but can freely modify the firmware (either as the device manufacturer or by having an opportunity to rewrite it).

Assuming that the device being modified does not use all available pins on the controller, an otherwise unused pin can serve as an antenna. The modified firmware can toggle it at a known frequency, which will produce a square-wave electromagnetic signal that can be received by an observer with an antenna. The project consists both of implementing such a transmission mechanism and processing the received signal to reconstruct leaked data.

The choice of modulation technique used to transmit the data affects several properties of the side channel, such as maximal achievable bit rate and reliability over longer distances. The visibility to an observer examining the device is also affected – for ex-

ample, a frequency-shift keyed signal would be much easier to spot than one using a spread-spectrum technique.

Starting point

To verify that the signal such a channel would create is strong enough, I have created a proof of concept: a microcontroller (NXP LPC11U24) programmed to rapidly toggle a pin (at a fixed frequency). The pin acts as an antenna, broadcasting the square-wave signal. With the help of Dr Kuhn, the resulting signal was recorded using the RS Signal and Spectrum Analyzer FSV7, using a biconical antenna at a distance of around 2 m as the receiver.

The microcontroller was programmed to create a square wave with a frequency of 2.4 MHz. This component is not visible on the received signal's spectrum (both using the biconical and a near-field antenna), since the frequency is too low for a short pin to act as an efficient antenna. Higher harmonics of the fundamental frequency, however, are clearly visible. At 122.4 MHz (51st harmonic), without significant filtering, the signal differs from ambient noise by around 20 dB.

Resources required

For the microcontroller on which the side channel will be created, I am planning to use an NXP LPC11U24 (same as for the proof of concept). As a backup in case of hardware failure, I have two of these.

I have spoken to Dr Kuhn about the hardware required for signal recording, and he has confirmed that it is available in the CL and that I will be given access. The spectrum analyser used for the proof of concept will be on loan to Cavendish Laboratory during Michaelmas term, but other receivers are available as well.

For processing the recorded data, I shall use my own laptop (quad-core Intel i5 processor, 20 GB RAM, running Windows). The data will be backed up to cloud storage and an external drive. In case of hardware failure, MCS computers will be used as a backup measure.

Work to be done

The project breaks down into the following sub-projects:

1. Choosing appropriate encodings and modulation schemes for data transmitted via the side channel and implementing the transmission on a microcontroller.
2. Recording the signal the microcontroller transmits.
3. Implementing the necessary signal processing to recover leaked data from the signal.
4. Evaluation of the side channel (maximal distance over which communication is possible, bit rate at shorter distances).

5. Writing the Dissertation.

Success citeria

The main success criterion is to demonstrate that data can be leaked via the side channel and recovered by a receiver (the proof of concept suggests that this should minimally be possible at a distance of several metres).

The modulation techniques used can be further evaluated by measuring the maximal bitrate at which they can transmit reliably (with an error rate below a specific margin) at different distances. This would allow comparing them both by their bandwidth and distance over which they are usable.

Possible extensions

If I achieve the main result early, possible extensions to the project are:

1. Demonstrating the side channel on a microcontroller running other code in parallel with the "leaking" routine (instead of having the entire processing power available for generating the signal). This would be done by modifying a "real" device – for example, implementing a custom keyboard firmware that transmits the user's keystrokes while they type.

2. Evaluating the "visibility" of the implementations used by standard methods for side channel detection.

Timetable

Planned starting date is 21/10/2011.

1. **Michaelmas weeks 2–3 (ends Oct 25):** Preliminary reading and deciding on the specifics of the project (in particular, the modulation used for the data).

2. **Michaelmas weeks 4–5 (ends Nov 8):** Implementation of the side channel on a microcontroller. Recording data produced by this, to be used as sample data while implementing signal processing.

3. **Michaelmas weeks 6–8 (ends Nov 29):** Start implementation of signal processing necessary to extract data out of the signal.

4. **Michaelmas vacation:** Finish signal processing implementation.

5. **Lent weeks 0–1 (ends Jan 26):** Write progress report.

6. **Lent week 2 (ends Jan 31):** If the work done with the recorded data indicates that a modified encoding would be more suitable, implement it. In either case, test the signal processing implementation on newly recorded data.

7. **Lent weeks 3–4 (ends Feb 14):** Record detailed data for evaluation (received signal over multiple distances and bit rates).

8. **Lent weeks 5–8 (ends Mar 14):** Evaluation of the project on recorded data and final modifications.

9. **Easter vacation:** Extensions and writing dissertation main chapters.

10. **Easter term 0–2 (ends May 9):** Further evaluation and complete dissertation.

11. **Easter term 3 (ends May 16):** Proof reading and then an early submission so as to concentrate on examination revision.