An Accessible Project 25 Receiver Using Low-Cost Software Defined Radio

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Mick V. Koch
August 2016

© 2016 Mick V. Koch. All Rights Reserved.
This thesis titled
An Accessible Project 25 Receiver Using Low-Cost Software Defined Radio

by
MICK V. KOCH

has been approved for
the Department of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

Shawn Ostermann
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
ABSTRACT

KOCH, MICK V., M.S., August 2016, Computer Science


Director of Thesis: Shawn Ostermann

Project 25 (P25) radio, now used by at least 33% of public safety agencies in the US, is accessible to only specialized, digital receivers. These receivers, though, are expensive consumer products – starting at $400. As public safety communications remain legal to receive in unencrypted digital form, the current migration to digital radio has simply made these communications less accessible to the public. What’s missing from the current ecosystem is a sub-$100 P25 receiver with usability similar to a traditional device – automatic, hands-free operation in a portable package – that makes these communications accessible again with a more affordable price.

The result of this research is a device meeting these requirements, made from a $20 RTL-SDR software defined radio, a Raspberry Pi, and a software P25 receiver pipeline. This implementation was evaluated as follows: baseband symbol decoding and frame synchronization accuracies were measured over 4 million random symbols in the presence of varying levels of noise and distortion, and overall performance was compared to a commercial P25 receiver by measuring voice frame muting errors.

This evaluation found the baseband symbol decoder had over 89% accuracy down to a 3:1 SNR, and the frame synchronizer had fewer than 0.0001% false positive and false negative errors at 0.001:1 SNR. Compared to the commercial receiver, the designed receiver recovered over 95% of voice frames without muting errors. These findings show that recent advances in low-cost software defined radio allow the device to satisfy the above requirements with suitable real-world performance.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>List of Tables</td>
<td>6</td>
</tr>
<tr>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Approach</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Document Structure</td>
<td>13</td>
</tr>
<tr>
<td>2 Background</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Project 25 (P25) and Public Safety Communications</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 Evolution of Public Safety Communications</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 P25 Technical Overview</td>
<td>24</td>
</tr>
<tr>
<td>2.1.3 Legality of Receiving P25</td>
<td>30</td>
</tr>
<tr>
<td>2.2 Software Defined Radio (SDR)</td>
<td>33</td>
</tr>
<tr>
<td>2.2.1 SDR for Accessible Radio Applications</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2 The Affordable RTL-SDR</td>
<td>44</td>
</tr>
<tr>
<td>2.3 P25 Software Receivers</td>
<td>56</td>
</tr>
<tr>
<td>3 Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>3.1 Baseband Symbol Decoding</td>
<td>64</td>
</tr>
<tr>
<td>3.1.1 Procedure</td>
<td>65</td>
</tr>
<tr>
<td>3.1.2 Results</td>
<td>69</td>
</tr>
<tr>
<td>3.1.3 Discussion</td>
<td>80</td>
</tr>
<tr>
<td>3.2 Frame Synchronization</td>
<td>81</td>
</tr>
<tr>
<td>3.2.1 Procedure</td>
<td>86</td>
</tr>
<tr>
<td>3.2.2 Results</td>
<td>88</td>
</tr>
<tr>
<td>3.2.3 Discussion</td>
<td>96</td>
</tr>
<tr>
<td>3.3 Comparison to Commercial Receiver</td>
<td>101</td>
</tr>
<tr>
<td>3.3.1 Procedure</td>
<td>102</td>
</tr>
<tr>
<td>3.3.2 Results</td>
<td>105</td>
</tr>
<tr>
<td>3.3.3 Discussion</td>
<td>105</td>
</tr>
<tr>
<td>4 Conclusions</td>
<td>110</td>
</tr>
<tr>
<td>4.1 Review</td>
<td>110</td>
</tr>
<tr>
<td>4.2 Future Work</td>
<td>111</td>
</tr>
</tbody>
</table>


**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Hardware components of P25 receiver.</td>
</tr>
<tr>
<td>2.1</td>
<td>Standard algorithms used to decode CAI error correcting codes.</td>
</tr>
<tr>
<td>2.2</td>
<td>Survey of consumer SDRs, comparing main parameters. [53], [54], [57]–[60]</td>
</tr>
<tr>
<td>3.1</td>
<td>Magnitude of impulse associated with each C4FM symbol.</td>
</tr>
<tr>
<td>3.2</td>
<td>Symbol impulse magnitudes for each waveform.</td>
</tr>
<tr>
<td>3.3</td>
<td>Distribution of symbol values as $\mu \pm \sigma$.</td>
</tr>
<tr>
<td>3.4</td>
<td>Gaussian intersection points.</td>
</tr>
<tr>
<td>3.5</td>
<td>Thresholds calculated using Equation 3.13.</td>
</tr>
<tr>
<td>3.6</td>
<td>Probability of misclassification for each symbol under varying SNR.</td>
</tr>
<tr>
<td>3.7</td>
<td>Percentage of misclassification for each symbol on test data.</td>
</tr>
<tr>
<td>3.8</td>
<td>Confusion matrix for evaluation on Both waveform at 0.001:1 SNR.</td>
</tr>
<tr>
<td>3.9</td>
<td>Percent of incorrect frame synchronization timing.</td>
</tr>
<tr>
<td>3.10</td>
<td>Average probability of false positive errors.</td>
</tr>
<tr>
<td>3.11</td>
<td>Average probability of false negative errors.</td>
</tr>
<tr>
<td>3.12</td>
<td>Percent error from means in Table 3.3 for various SNR, as $\mu % \pm \sigma %$.</td>
</tr>
<tr>
<td>3.13</td>
<td>Percentage of false positive errors.</td>
</tr>
<tr>
<td>3.14</td>
<td>Percentage of false negative errors.</td>
</tr>
<tr>
<td>3.15</td>
<td>Percentage of incorrect symbol clock occurrences.</td>
</tr>
<tr>
<td>3.16</td>
<td>Percentage of misclassification for each symbol on evaluation data.</td>
</tr>
<tr>
<td>3.17</td>
<td>Results of comparison experiment.</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The prototype P25 receiver designed in this research.</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>Components of P25’s CAI standard. [22]</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Lifetime of a P25 conversation. [22]</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Narrowbanding helped increase P25 market share to 33% in 2012. [2]</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Theoretical performance of analog and digital radio when subjected to attenuation. [22]</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Steps to receive various packets in P25 CAI.</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Packets carried over the CAI and their functions.</td>
<td>27</td>
</tr>
<tr>
<td>2.7</td>
<td>Hierarchy of a P25 voice message.</td>
<td>29</td>
</tr>
<tr>
<td>2.8</td>
<td>Many RF protocols can be implemented with the same common hardware by simply swapping out the software.</td>
<td>35</td>
</tr>
<tr>
<td>2.9</td>
<td>Encoding 16 bits with QPSK using IQ modulation.</td>
<td>38</td>
</tr>
<tr>
<td>2.10</td>
<td>Some consumer SDR boards, from lower to higher price. [57]–[59]</td>
<td>39</td>
</tr>
<tr>
<td>2.11</td>
<td>ScAn Testbed flight setup. [65]</td>
<td>42</td>
</tr>
<tr>
<td>2.12</td>
<td>Circuitry of older RTL-SDR, showing common 28.8MHz “can” crystal.</td>
<td>46</td>
</tr>
<tr>
<td>2.13</td>
<td>Circuitry of modern TCXO RTL-SDR, with main components identified: RTL2832, TCXO, and R820T2.</td>
<td>47</td>
</tr>
<tr>
<td>2.14</td>
<td>Components used by the RTL-SDR to capture and sample RF signals.</td>
<td>48</td>
</tr>
<tr>
<td>2.15</td>
<td>Example QPSK receiver using the RTL-SDR. [15]</td>
<td>50</td>
</tr>
<tr>
<td>2.16</td>
<td>Similar approaches to FM demodulation used by two popular, open source SDR applications.</td>
<td>53</td>
</tr>
<tr>
<td>2.17</td>
<td>Signal processing approach for the receiver studied here.</td>
<td>56</td>
</tr>
<tr>
<td>2.18</td>
<td>Architecture of OP25 receiver. [100]</td>
<td>58</td>
</tr>
<tr>
<td>2.19</td>
<td>Dependency graph of Rust modules.</td>
<td>60</td>
</tr>
<tr>
<td>2.20</td>
<td>Parallel distribution of work across threads.</td>
<td>62</td>
</tr>
<tr>
<td>3.1</td>
<td>Nominal waveform in unprocessed form and after application of AWGN at 12:1 through 0.001:1 SNR.</td>
<td>67</td>
</tr>
<tr>
<td>3.2</td>
<td>Histograms and PDFs for Nominal waveform, for various SNR.</td>
<td>70</td>
</tr>
<tr>
<td>3.3</td>
<td>Histograms and PDFs for Scaled waveform, for various SNR.</td>
<td>71</td>
</tr>
<tr>
<td>3.4</td>
<td>Histograms and PDFs for Offset waveform, for various SNR.</td>
<td>72</td>
</tr>
<tr>
<td>3.5</td>
<td>Histograms and PDFs for Both waveform, for various SNR.</td>
<td>73</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison of different waveforms at 3:1 SNR.</td>
<td>75</td>
</tr>
<tr>
<td>3.7</td>
<td>P25 frame synchronization waveform, with impulse instants as dots.</td>
<td>82</td>
</tr>
<tr>
<td>3.8</td>
<td>All symbol clock errors were due to peak power falling on the sample exactly before or after the correct one, shown for Both waveform at 0.001:1 SNR.</td>
<td>89</td>
</tr>
</tbody>
</table>
3.9 Comparison of correlation powers during and after synchronization sequence, with various distortion at 0.001:1 SNR. 90
3.10 Comparison of correlation powers during and after synchronization sequence at SNR extremes. 91
3.11 EWMA accuracy and settling time for various $\alpha$ on Nominal waveform. 92
3.12 Outer-sync correlation power stays constant while inner-sync peak power spreads in the presence of noise, shown for Nominal waveform. 93
3.13 Distribution values for Nominal unprocessed waveform. 94
3.14 Resulting threshold for each waveform at 0.001:1 SNR. 95
3.15 Designed receiver and Motorola recording radio traffic in OUPD HQ. 103
3.16 The Motorola (top) and designed receiver (bottom) waveforms, showing the obvious time separation between voice messages. 104
3.17 Example voice message received on the Motorola (top), and the same message with muted frames (approximately 137.) 104
3.18 Frame muting errors occurred throughout the 2 hour experiment. 106
3.19 Across all voice messages, 81% had zero muting errors, and 97% had less than 40% muted frames. 107
3.20 Muting errors were generally spread across message lengths, but 100% mutings only occurred with messages less than 2 seconds long. 108
3.21 The radio traffic included a mix of message lengths. 109

A.1 Ideal and approximated frequency response of raised cosine filter. 125
A.2 The impulse response satisfies the Nyquist criterion at 48kHz. 126
A.3 Ideal and approximated frequency response of preemphasis filter. 128
A.4 Ideal and approximated frequency response of deemphasis filter. 130
A.5 Frequency response of anti-alias filter. 132
A.6 Frequency response of channel select filter. 134

C.1 Schematic of receiver designed in this research. 136
1 Introduction

Staying informed has become more difficult and expensive with the widespread adoption of Project 25 (P25) radio by US public safety agencies. These agencies – police, fire/rescue, EMS, and others – have been pushed by the poor interoperability of aging equipment and the recent enforcement of a narrowbanding deadline [1] to migrate to digital radio, with P25 radio being the most popular choice in the US [2]. As a result, traditional analog scanner/receivers, used for decades to monitor local happenings in real-time [3]–[5], are becoming obsolete in more and more areas. The digital communications remain legal to monitor in these areas [6], but doing so requires a specialized P25 receiver. These receivers are expensive consumer devices – starting at $400 – and are currently only available from 2 manufacturers [7], [8]. As traditional equipment with equivalent analog functionality can be had for less than $30 [9], the current market of P25 receivers makes the pastime relatively inaccessible. The goal of this research was to make these communications accessible again by creating a sub-$100, trunking-aware P25 receiver that is portable, automatic, and open source.

1.1 Motivation

The accessibility of public safety communications determines how easily the public can receive the information that is broadcast on these channels. This stream of information from law enforcement, fire/rescue squads, emergency medical, and other agencies provides a real-time picture of the surrounding area, and for decades people have monitored these channels to hear firsthand updates on weather emergencies, traffic accidents, road closures, and other local happenings [3]–[5]. Others treat it as just another hobby – to keep up with what goes on “behind the scenes” of daily life – since it takes some amount of research and equipment to tune in. The public interest in this information can be seen, for example, in the popularity of Twitter
feeds and livestreams that rebroadcast it in more familiar forms [10]–[12] (although these rebroadcasts may be illegal [6].) With the current migration to digital P25 radio, though, this information is less accessible than it once was.

Current hardware and software solutions to receive P25 are not ideal for various reasons. As mentioned, the hardware P25 receiver market is a niche one, with little diversity in features and price relative to the analog receiver market. This lack of choice means consumers are locked into the prices whether or not they want all the features they’re paying for. This research shows, however, that if the desired features are simple P25 voice reception in a device with usability similar to a traditional receiver – mainly, hands-free operation in a portable package – the price can be less than $100. In addition, the receiver designed here is open source, so unlike consumer devices, the software can be freely modified and upgraded by the user: More advanced features missing from this first-generation device can be implemented by the community and patched in by users.

Although several software packages exist to receive P25, none meet the additional constraints required by this receiver. These constraints – automatic operation and portability – make the receive more suited to the use cases previously fulfilled by traditional handheld receivers on analog systems. These devices could be left on in the background or carried out in the “field” and would reliably receive what they could off the user-programmed frequencies without manual tuning. In contrast, available P25 software is either written as proof-of-concept or for users in front of a computer screen, which fulfills a different set of use cases.

1.2 Approach

With the above motivations, a suitable receiver for this research was designed as follows. Beginning with the hardware, the components in Table 1.1 were combined
to form the device shown in Figure 1.1. This receiver targets the Raspberry Pi 2 Linux computer [13], which can be powered by battery with moderate current draw (around 1 Amp under load.) In addition, the computer measures 3.5“ × 2.2“ × 1.0“ and provides 40 programmable General Purpose Input/Output (GPIO) pins [14], so it’s suitable to be the core of a portable, embedded package. Connected to the Pi is the essential component that makes this receiver possible: the $20 RTL-SDR (Realtek-based software defined radio), a very recent development in consumer SDR [15]. This USB dongle allows the receiver to tune within the 700MHz band commonly used by P25 sites and provides a stream of RF samples at a programmable sample rate to be processed in software. Using this, the software can demodulate the P25 “air interface” RF signal, which is the lowest-level requirement of any P25 receiver [16]. The price, form factor, and programmability of the RTL-SDR allows the receiver to satisfy the requirements of sub-$100, portable, and automatic.

Table 1.1: Hardware components of P25 receiver.

<table>
<thead>
<tr>
<th>Component</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry Pi 2B</td>
<td>35</td>
</tr>
<tr>
<td>RTL-SDR (TCXO)</td>
<td>20</td>
</tr>
<tr>
<td>4GB SD Card</td>
<td>5</td>
</tr>
<tr>
<td>USB Battery Pack</td>
<td>5</td>
</tr>
<tr>
<td>ABS Box</td>
<td>8</td>
</tr>
<tr>
<td>LCD Screen</td>
<td>20</td>
</tr>
<tr>
<td>Rotary Encoder</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>99</strong></td>
</tr>
</tbody>
</table>

The software interface to this dongle is one of several general, decoupled modules created for this receiver. This software is released free and open source and written
Figure 1.1: The prototype P25 receiver designed in this research.
in the Rust programming language [17]. Rust provides an advanced type system, compile-time and run-time memory safety, and “zero-cost abstractions” that compile down to optimized machine code using LLVM. Using the supporting modules, the main application receives and decodes voice messages broadcast on P25 radio channels as well as hosts a simple user interface based on a rotary encoder and LCD screen. This application is currently targeted for the Raspberry Pi 2, but it could be readily adapted to other platforms due to the supporting modules being generally cross-platform. With battery-powered portability provided by the Raspberry Pi, automatic operation is provided by trunking awareness – by monitoring the trunking packets broadcast on P25 control channels. Using this, the receiver can intelligently follow voices messages across multiple traffic channels without user interaction. These components are described in more detail throughout the remaining chapters.

1.3 Document Structure

The rest of the document is organized as follows: Chapter 2 expands on topics presented in this introduction, including P25, SDR, and the RTL-SDR. It also reviews the current state of software P25 reception in comparison with the solution studied in this research. Next, Chapter 3 evaluates the implementation, with each section containing experimental procedures, results, and discussion. Finally, Chapter 4 concludes the document with an overall review of the main findings and suggestions for future work. The end of the document also includes Appendices for design fragments related to the research.
2 Background

2.1 Project 25 (P25) and Public Safety Communications

2.1.1 Evolution of Public Safety Communications

The current switch to digital radio has been over two decades in the making, with the inception of Project 25 (P25) happening in 1989 [18]. In the years leading up to that date, digital radio for voice and data was becoming more and more practical for manufacturers to produce. The manufacturers, however, were each creating their own proprietary protocols, incompatible with the others [19]. The Association of Public Safety Communications Officials (APCO) foresaw this causing major interoperability problems down the road. For example, if a local police agency chose one system while EMS chose another, they would have to invest in intermediate patch hardware or resort to telephone links for communication between agencies. Further, if a federal, state, or local government chose to mandate one particular system for their agencies, they would then be in a monopolistic situation with the manufacturer of that system. These problems were compounded with the growing congestion of public safety radio bands at that time, especially in urban areas: The bandwidth required by the radio systems was clogging the available public safety spectrum, and the bands were increasingly unable to supply the number of channels needed for dispatch, coordination, and other talk paths in large cities [19].

With these motivations, APCO P25 was initiated in 1989 as a collaboration between government, public safety users, and manufacturers to create a digital radio system for public safety communications needs. The system was planned as an open standards process, so disparate vendors could compete in designing hardware, and that hardware would be interoperable with any other hardware that also followed the
standards [20]. The goal of the standards themselves was to create a full-featured, spectrum-efficient radio system tailored to public safety needs.

The work on these standards progressed for 6 years and culminated in P25 Phase I, released in 1995 [18]. Phase I is a set of over 30 standard documents [21] split across 8 interfaces: the Common Air Interface for the P25 radio protocol; the Fixed Station Interface for command and trunking messages; the Console Subsystem Interface for consoles used by dispatchers; the Network Management Interface for computer networking within P25 systems; the Inter RF Subsystem Interface for connecting individual P25 systems into wide-area networks; the Data Network and Subscriber Data Peripheral Interfaces for connecting P25 radios to laptops and the port protocol to do that; and the Telephone Interconnect Interface for connecting a P25 system to the public telephone network [2], [22]. This research deals almost entirely with the Common Air Interface (CAI) except for pieces from the trunking part of the Fixed Station Interface (FSI).

The CAI and FSI solve the spectrum problem in two ways. First, the CAI provides digital voice and data in a 12.5kHz channel – half the bandwidth needed by 25kHz analog systems – that doubles the number of channels that can be fit into the available bands. Second, the FSI uses trunking to dynamically assign each radio call to a channel within a pool of channels. This allows channels to be shared between multiple agencies and reduces spectrum waste caused by underutilized channels [22]. Within a 12.5kHz channel, P25 delivers 9600 bits per second (4800 baud) using C4FM, a 4-level baseband modulation that is then modulated up to RF using a typical FM transmitter. Both voice and data traffic are supported, with digital voice provided by the Improved Multi-Band Excitation (IMBE) vocoder that packs 20ms of speech into 88 information bits. Both types of traffic are engineered to reduce errors with the
use of error-correcting codes (5 different types), interleaving, and cyclic redundancy checks [23]. Simon [24] gives a very detailed analysis of the CAI, and Figure 2.1 shows a high-level overview of its different parts.

![Diagram of P25's CAI standard]

Figure 2.1: Components of P25’s CAI standard. [22]

Although production equipment must implement both the transmission and reception of P25 traffic, scanner/receivers only need to implement reception. In fact, creating a simple receiver that can tune to a P25 channel and output the recovered voice would require only a subset of the CAI standard alone. For a better user experience, though, the receiver can be smarter. In the simplest case, a trunked P25 system is split into a control channel and one or more traffic channels, as shown in Figure 2.2. The control channel continuously (100% duty cycle) broadcasts outbound “trunking signaling” packets supplied by the trunking controller and receives inbound signaling packets sent from individual units. For example, if a police officer with a P25 handheld keys the push-to-talk, the radio sends a request for a traffic channel as
an inbound packet on the control channel. The trunking controller then either replies with the traffic channel currently assigned to the officer’s talk group (if a conversation is ongoing), allocates a new channel from the pool of available ones and sends that, or sends a notification that no channels are available. The officer’s radio, and other radios in the talk group, monitor the control channel for the reply and either switch to the assigned traffic channel and start transmitting/receiving or output a “busy” tone [22]. This all happens very quickly and, if everything goes well, is invisible to the users.

![Diagram showing the lifetime of a P25 conversation.](image)

Figure 2.2: Lifetime of a P25 conversation. [22]

Traditional analog receivers have little use when applied to this digital radio because they’re capable only of FM demodulating a P25 channel into a C4FM
baseband waveform (which has a characteristic growling sound.) These devices have
been used since at least the 1970s to monitor public safety channels [3]–[5], but the
obsolescence of analog public safety communications in favor of P25 has replaced the
chatter with silence, with more and more areas affected as P25 adoption has grown
since Phase I was finalized in 1995. Through the late 2000s and early 2010s, new P25
deployments were increasing every year, culminating in a spike of of new deployments
in 2012. The rate has tapered off since then, but P25 is still being rolled out across
the country [2].

This growth has been fueled by the same factors that gave P25 its start:
interoperability and spectrum efficiency. Broken interoperability has been a key failing
found in the “post-mortems” of many disaster responses over the years. For example,
during 9/11, firefighters inside the north tower were unable to hear instructions to
leave the building from a police helicopter (on a separate radio system) that could see
it was about to fall [25]. In Ohio, the Multi-Agency Radio Communications System
(MARCS) – which runs the statewide P25 network – was initiated in 1994 in response
to the 1990 Shadyside flood and the 1993 Lucasville prison riot. The failings in the
riot response include the warden being unable to directly communicate and coordinate
with his prison officers while he was returning from across the state as well as a hand
message relay being necessary for communication between the warden and the state
patrol officer negotiating with the inmates [26], [27].

Ideally, these problems are fixed by the standardization of communication and
networking protocols in P25 Phase I. Additionally, P25 requires manufacturers to
undergo compliance testing on their equipment to verify it follows these standards
[28]. For example, compliance with the CAI requires that manufacturers implement
backwards compatibility with analog FM radios, as well as the digital protocol [22].
After many years of deploying such equipment, Ohio MARCS now claims 97.5% P25 mobile radio coverage in the state, higher than it’s ever been. Additionally, the sites that make up the system are linked together into a larger network, meaning that any individual in the system can talk to any other group of individuals statewide by switching the talkgroup on their radio [29]–[33]. Ideally, the standards allow such seamless communications between agencies at the federal, state, and local levels [22].

Along with interoperability, the push for spectrum efficiency in the late 1980s and onward has fueled the growth of P25. The FCC dealt with the spectrum problem on two fronts: using narrowbanding to increase the number of channels available in allocated spectrum and reallocating spectrum to create new channels exclusively for public safety use. Traditionally, public safety communications have happened in the 150 to 174MHz VHF and 421 to 470MHz UHF bands of frequencies [1], [34]. A dozen large cities – Cleveland, New York, and San Francisco among them – also have communications in the 470 to 512MHz “T-Band” [35]. The first two bands are shared with industrial/business users, and the T-Band is shared with TV broadcasters, which reduces the number of channels available for public safety [1]. Each channel was allocated 25kHz of bandwidth (12.5kHz between center frequencies, with overlap, on VHF and 25kHz between center frequencies on UHF [34]), typically for analog FM voice. This convention became unsustainable, though, with congestion limiting the spectrum available for expansion of existing systems and creation of new systems [36]. In response, the FCC began a “refarming” initiative in 1992, and one of the key results was narrowbanding reform: halving the bandwidth of all channels to 12.5kHz. In all the bands this would allow allocating new channels halfway between the center frequencies of existing ones, doubling the number available for use [34]. The FCC worked aggressively to implement this and mandated the end of 2012 as the final
deadline before which agencies must switch to narrowband technology: After the beginning of 2013, the FCC would no longer issue new 25kHz licenses and would prohibit the use and manufacture of 25kHz equipment on the affected bands [34], [36]. It’s no surprise, then, that new deployments of P25 – a 12.5kHz technology – peaked in 2012. As Figure 2.3 shows, those new deployments pushed P25 to a 33% market share in North America.

![NA User Radios Market Share](image.png)

**Figure 2.3:** Narrowbanding helped increase P25 market share to 33% in 2012. [2]

It’s likely that this number will continue to grow and that P25 Phase I will be the standard for many years. As a result, increasingly more people will require a Phase I compatible receiver to tune in to public safety communications, and this will be the case for a while. Although Figure 2.3 shows that almost half of the market has chosen to remain on (narrowbanded) analog equipment – in part due to cost [32], [37], [38] – that number will likely continue to fall as the performance gain of digital radio
on narrow channels becomes more attractive. The FCC doesn’t require that digital radio be used in these channels, but they do recommend it. They estimate a 3dB loss (halving) of coverage area for the same power output when moving an analog system from 25kHz to 12.5kHz, and they credit technology like error correcting codes for making the loss less significant in digital systems [34], [36]. Similarly, a study on P25 coverage found that 12.5kHz analog FM radio requires a 6dB (quadruple) higher Carrier-to-Noise Ratio (CNR) than 25kHz for the same performance, while P25 radio requires 3.3dB less than 25kHz analog and 8.3dB less than 12.5kHz analog [2]. This differing performance in noise is illustrated in Figure 2.4, where the analog system becomes noisy sooner than the digital system, but the digital system has a sharper cutoff after which audio quality degrades rapidly [22].

Public safety agencies are weighing these tradeoffs, and they’re migrating to P25 when they see opportunities for increased coverage or functionality that would be more expensive to implement in analog systems [25], [33]. Among these agencies, the federal Departments of Defense, Homeland Security, Justice, and Treasury have standardized on P25 for their land mobile radio [28]. Each migration creates more momentum behind P25 as these agencies invest large amounts of money in new equipment and infrastructure (for example, the typical price for a P25 handheld is $5000 and up [20], [37].) This equipment has a “useful life” before which it’s very hard to justify replacing [39]. In fact, that same useful life is what in part caused the decades of delay for many agencies to migrate to P25 in the first place [20], so the same cycle will likely occur with P25 Phase I’s successor.

An example of this cycle and momentum can be seen with the recent major update to the APCO standards for P25 Phase II. The standards for Phase II were release in 2012 [18], and uptake so far has been slow [39]. The viability of Phase II
Figure 2.4: Theoretical performance of analog and digital radio when subjected to attenuation. [22]

was hedged on the FCC’s 2017 “ultra-narrowbanding” deadline [2], which had been in planning since 2005, but that deadline was withdrawn in 2014 due to pressure from manufacturers and public safety agencies [34], [39]. Similar to the narrowbanding deadline, the ultra-narrowbanding deadline was meant to halve all channels again – this time to 6.25kHz – by the end of 2016 [34]. P25 Phase II was designed to satisfy this requirement with the use of Time-Division Multiple Access (TDMA), operating across two 6.25kHz channels with harmonic continuous phase modulation [22]. The
deadline never came, though, as manufacturers complained about the extreme signal processing requirements of the narrower bandwidth [40], and public safety agencies complained about having to rework or replace existing systems well before they’d reached their useful lives – many Phase I radio systems in the US would have been less than 8 years old by the end of 2016 [39]. The FCC currently has no plans to enforce ultra-narrowbanding [34], [36], so agencies that migrate to Phase II are doing so due to legitimate capacity needs, such as in large cities [39].

Because it can cost millions of dollars more to implement a Phase II system [39], it seems likely that, without more external pressure, most agencies will continue adopting Phase I in the near future. In the farther future, there are several digital radio protocols with higher data rate that could succeed P25, with the main contenders being Terrestrial Trunked Radio (TETRA) and Long-Term Evolution (LTE). Although P25 isn’t the most advanced or fastest protocol when compared with these, it seems to work “well enough” for the current needs of public safety agencies [32], [33], [38]. Its relatively low data rate may actually be well suited to the coverage requirements of less densely populated areas of the US, in terms of infrastructure requirements. For example, a study in Oregon found that statewide radio coverage would require a total of 200 towers using TETRA but only 140 using P25 [20]. In addition, P25’s relatively simple modulation remains compatible with the nonlinear amplifiers used by analog FM systems, while TETRA and LTE (and P25 Phase II) all require linear amplifiers. These linear amplifiers both cost more and produce less output power for the same input power compared to nonlinear amplifiers [22]. So although 10MHz of public safety spectrum has been set aside for broadband LTE experimentation [1], [23], [41], it seems unlikely that such a system will be
implemented and adopted soon. Rather, a P25 Phase I receiver like the one designed in this research will likely be the requirement and the norm for years to come.

2.1.2 P25 Technical Overview

The P25 Common Air Interface (CAI) protocol is complex enough for an interesting and challenging software implementation but not so complex to have prohibitive signal processing and computing requirements. The flow diagram in Figure 2.5 shows the general steps necessary to receive voice and other messages in a P25 system. This illustrates the various communications engineering and error correction techniques used to support the 7 packet types carried on the CAI. Although the software written for this research doesn’t implement the full CAI, it does implement all of the pieces in this diagram and uses unit and integration testing to measure compliance with the standard.

Consider the steps to receive a voice message. First, the receiver must tune to a control or traffic channel frequency. As described previously, a trunked P25 system is composed of one or more control channels and one or more traffic channels. The primary control channel is used for communication between subscriber unit radios (handheld and mobile radios used by individual officers, firefighters, etc.) and the trunking controller, and there may be additional control channels per site for redundancy in case of equipment failure. These control channels primarily carry trunking control packets and “multi-block trunking” packets, a form of unconfirmed data packet. These packets broadcast various information – such as network status, adjacent P25 sites, and traffic channel allocations [42] – to subscriber units. For traffic channel allocations, the central trunking controller computer assigns inbound voice and data traffic requests to a specific traffic channel and broadcasts the channel number to listening subscribers. These traffic channels primarily carry packets related
Figure 2.5: Steps to receive various packets in P25 CAI.
to voice messages – for traditional spoken communications – as well as confirmed and unconfirmed data packets – for digital communications like text messages and TCP/IP networking [22], [43].

In any case, the RF signal centered on these channel frequencies is FM modulated with a bandwidth of 12.5kHz, so a standard FM demodulator can be used to recover the modulating C4FM signal. This research uses SDR for tuning and FM demodulation, as detailed in Section 2.2.2. The C4FM signal is a filtered-impulse, Audio Frequency (AF) modulation that encodes 2 bits at a time using 4 impulse magnitudes (this leads to a form of constant-envelope phase shift keying in the FM signal.) This baseband signal is first deemphasis filtered, to complement the transmitter’s preemphasis filtering and reduce the effects of high-frequency noise. Then, the receiver must lock on to the frame synchronization sequence, which is a fixed waveform that appears before each packet. From there, the receiver must recover the impulse magnitudes from the C4FM waveform and translate them back to 2-bit symbols called dibits at 9.6kbps (4800 baud) [16]. This research uses a correlator method for frame synchronization and a threshold method for symbol decoding, both of which are characterized and evaluated in Sections 3.1 and 3.2.

These dibits are then buffered to form the packet body. First, the status symbols must be removed from the stream: After every 35 information dibits, a status dibit is interleaved, which is used by subscriber units for Slotted ALOHA access to the control channel [22]. Once 64 information dibits are buffered, the Network Identifier (NID) must be decoded. This is a field at the beginning of every packet body, composed of 16 information bits that are BCH coded with 48 parity bits. Part of this field is a 4-bit flag that determines which of the 7 packets in Figure 2.6 the receiver should
begin reconstructing. These packets range in size from 0 bits for a Simple Terminator, to 1,568 bits for a Frame Group, up to 2,298 bits for a full Confirmed Data packet.

**Voice Header** Appears at the start of every voice message and carries related cryptographic and talkgroup information

**LC Voice Frame Group** Carries digitized voice frames as well as interleaved link control (LC) information and the first half of a low-speed data word

**CC Voice Frame Group** Same as above, but instead carries cryptographic control (CC) information and the second half of the low-speed data word

**Simple Voice Terminator** Signals the end of a voice message

**LC Voice Terminator** Same as above, but also carries link control information

**Confirmed/Unconfirmed Data** Carries arbitrary data messages with/without acknowledgments and retransmissions

**Trunking Control** Carries short trunking system messages

Figure 2.6: Packets carried over the CAI and their functions.

Out of all these packets, 5 are involved in voice messages. The hierarchy and sequencing of these voice packets in the protocol is shown in Figure 2.7. With supporting packets before and after, the core of a voice message is composed of alternating LC and CC Voice Frame Group packets. These are paired into virtual “Superframes”, which encode 360ms of voice and carry link control information (similar to that of trunking packets), cryptographic control information (the cipher, key ID, and synchronization state for the current voice message), and a 32-bit “low-speed data” word [16]. The decoding logic for this research is generally implemented
with layered state machines, operating on one dibit symbol or one baseband sample at each step.

Each Frame Group packet contains 9 voice frames that have been digitized and compressed with the Improved Multi-Band Excitation (IMBE) vocoder. This vocoder packs 20ms of speech into 88 data bits by splitting the voice signal into voiced and unvoiced spectra and compressing each differently. This process is reversed by the receiver, which regenerates and combines the voiced and unvoiced signals in order to reconstruct an approximation of the original voice signal [44]. This voice information may optionally be encrypted: P25 supports several symmetric ciphers, including BATON, 3DES, and AES and provides key management infrastructure such as over-the-air rekeying [22]. The software implemented for this research contains no support for decryption of encrypted frames.

As Figure 2.5 shows, each packet generally has one or more error correcting codes applied to protect its payload. These codes are summarized in Table 2.1 along with the decoding algorithms used in this research, which are standard and well-known in each case [45], [46]. These algorithms correct errors up to the capacity supported by the code and, in some cases, can detect when too many errors have occurred and the data is unrecoverable. This information is used by the vocoder during preprocessing to improve decoding performance [44].

The software implemented for this research supports all of the protocol required to receive voice and trunking messages, but it currently has only partial support for receiving data messages. This is simply due to data messages not being necessary for this research and their relative complexity in the protocol. Although Trunking Control packets and Data packets share the same initial decoding steps, data messages must implement an additional layer of protocol on top of this. This layer includes
Figure 2.7: Hierarchy of a P25 voice message.
Table 2.1: Standard algorithms used to decode CAI error correcting codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Decoding Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Standard (15, 11, 3)</td>
<td>Syndrome Lookup</td>
</tr>
<tr>
<td>Hamming Short (10, 6, 3)</td>
<td>Syndrome Lookup</td>
</tr>
<tr>
<td>Cyclic Short (16, 8, 5)</td>
<td>Cyclic Shift/Syndrome Lookup</td>
</tr>
<tr>
<td>Golay Standard (23, 12, 7)</td>
<td>2-Level Syndrome</td>
</tr>
<tr>
<td>Golay Extended (24, 12, 8)</td>
<td>2-Level Syndrome</td>
</tr>
<tr>
<td>Golay Short (18, 6, 8)</td>
<td>2-Level Syndrome</td>
</tr>
<tr>
<td>BCH (63, 16, 23)</td>
<td>Berlekamp-Massey/Chien Search/Forney</td>
</tr>
<tr>
<td>RS Short (24, 12, 13)</td>
<td>Berlekamp-Massey/Chien Search/Forney</td>
</tr>
<tr>
<td>RS Medium (24, 16, 9)</td>
<td>Berlekamp-Massey/Chien Search/Forney</td>
</tr>
<tr>
<td>RS Long (36, 20, 17)</td>
<td>Berlekamp-Massey/Chien Search/Forney</td>
</tr>
<tr>
<td>Convolutional (2, 1)</td>
<td>Viterbi</td>
</tr>
<tr>
<td>Convolutional (4, 3)</td>
<td>Viterbi</td>
</tr>
</tbody>
</table>

fragmentation/defragmentation of data across packets, packet sequencing, and, in the case of confirmed data messages, acknowledgment and retransmission logic [16]. Additionally, once the data is received, it has to be interpreted: Since these messages are application-specific, individual P25 systems may use closed and proprietary encodings. One standardized use of these messages, though, is for multi-block trunking control information. This is used by the Fixed Station Interface (FSI) to carry larger trunking messages than is possible with the typical Trunking Control packet [42]. In any case, data message support could make the software more useful for future receivers and experiments.

2.1.3 Legality of Receiving P25

Although public safety communications have become more difficult to receive (intercept) due to digitization and other complexities, it remains legal to do so, at
least with some restrictions. It should be noted that, although the law presented here is compiled and interpreted to the best of the author’s ability, a trained law professional may have knowledge of additional law not found in this research and may have a different interpretation of the given language.

Statutes relating to the interception of radio communications appear in Title 18, Chapter 119 of the U.S. Code. Specifically, 18 USC § 2511(1) states “Except as otherwise specifically provided in this chapter any person who … intentionally intercepts … any electronic communication … shall be punished … or shall be subject to suit.” Fortunately, an exception is provided in this chapter for public safety communications: 18 USC § 2511(2)(g) states “It shall not be unlawful under this chapter or chapter 121 of this title for any person … to intercept or access an electronic communication made through an electronic communication system that is configured so that such electronic communication is readily accessible to the general public.” The phrase “readily accessible to the general public” is defined in 18 USC § 2510(16) as

... Communication [that] is not

(A) scrambled or encrypted;

(B) transmitted using modulation techniques whose essential parameters have been withheld from the public ...;

(C) carried on a subcarrier or other signal subsidiary to a radio transmission;

(D) transmitted over a communication system provided by a common carrier ...;

(E) transmitted on frequencies allocated under part 25 [satellite], subpart D [remote pickup broadcast station], E [aural broadcast auxiliary
station], or F [television broadcast auxiliary station] of part 74, or part 94 [fixed microwave] of the Rules of the Federal Communications Commission ...;

Given that (B)-(E) aren’t applicable to P25, the only part that may apply is (A), due to P25’s support of encrypted talkgroups. This shows that it’s not unlawful to receive unencrypted P25 communications.

However, there appear to be restrictions on what can be done with the information after it has been intercepted. This is codified in 47 USC § 605(a), which states

... No person not being authorized by the sender shall intercept any radio communication and divulge or publish the existence, contents, substance, purport, effect, or meaning of such intercepted communication to any person.

No person not being entitled thereto shall receive or assist in receiving any interstate or foreign communication by radio and use such communication (or any information therein contained) for his own benefit or for the benefit of another not entitled thereto.

No person having received any intercepted radio communication or having become acquainted with the contents, substance, purport, effect, or meaning of such communication (or any part thereof) knowing that such communication was intercepted, shall divulge or publish the existence, contents, substance, purport, effect, or meaning of such communication (or any part thereof) or use such communication (or any information
therein contained) for his own benefit or for the benefit of another not entitled thereto. ...

This language seems to clearly prohibit divulging public safety communications or using what’s heard for (malicious) benefit. This notion is challenged, however, by the Radio Television Digital News Association, an electronics journalism group. They claim that a rebroadcast or divulgence of scanner traffic has never been prosecuted, and they point to a 2001 Supreme Court decision (Bartnicki v. Vopper) as evidence that the law may be unconstitutional on First Amendment grounds and could be challenged as such in court [47]. Whether this opinion is accepted or not, it’s clear that receiving unencrypted P25 with a radio device is undeniably legal as long as the received information isn’t divulged or used maliciously.

2.2 Software Defined Radio (SDR)

The RTL-SDR is the key technology that makes this receiver possible, and it is simply a low-cost Software Defined Radio (SDR). SDR makes radio applications more accessible, and recent developments in consumer SDR have brought the entry-level price down to $20. Along with this has come research into the capabilities of these devices and an active software ecosystem.

2.2.1 SDR for Accessible Radio Applications

Although SDR has existed for some time, consumer affordability is a recent development in the technology. The SDR concept was first formally described by Mitola in 1991: He proposed a “software radio” with a common hardware core that could be reconfigured on demand for different radio applications [48], [49]. It took time for technology to mature enough to make this feasible, but between 1997 and 2000, several international conferences brought researchers together to
discuss implementations and applications of SDR [48]. The Ettus Research Universal Software Radio Peripheral (USRP) was perhaps the first “affordable” SDR, at least for academia, although it initially cost several thousand dollars [50]. From there, these devices have continued to become more accessible, with sub-$1000 options appearing recently, for example the HackRF, which was first prototyped in 2012 [51].

As Mitola described, SDR in general reduces the amount of hardware necessary to implement Radio Frequency (RF) applications – SDR devices have just enough core circuitry to capture and sample RF signals, but further signal processing is left up to software. This can be software implemented on a general purpose processor or microcontroller or on a reprogrammable device like a Field-Programmable Gate Array (FPGA). In any case, the benefits include quick design and implementation turnaround as well as radio systems that are patchable and upgradable. By using this architecture, SDR allows a common hardware component to be applied to many RF applications by simply swapping out the software (illustrated in Figure 2.8.) Conversely, SDR makes RF application development more accessible by removing the need for design and fabrication of complex RF circuitry [52].

To support this, SDR circuitry is generalized for abstraction across applications, with tunable and programmable components. Typically, software is able to control center frequency tuning, sample rate, and amplifier gains [53], and higher-end SDRs may also support selecting and tuning hardware filters [54]. As such, the main parameters of SDR devices are tuning range, maximum bandwidth (determined by the maximum sample rate), transmit/receive capability, and dynamic range (determined by sample bit depth.) The last parameter affects the performance of the main input/output of these devices: In-phase/Quadrature (IQ) sampling.
Figure 2.8: Many RF protocols can be implemented with the same common hardware by simply swapping out the software.

The interface between SDR hardware and software is typically through IQ sampling. Either the SDR supplies the samples to the software (reception) or the software supplies the samples to the SDR (transmission). This type of sampling allows the application to process a modulating signal without concern for the carrier wave [55]. This separation is shown in the derivation: Given a real signal $s(t)$ with angular carrier frequency $\omega_c = 2\pi f_c$, amplitude $a(t)$, and phase $\phi(t)$,

$$s(t) = a(t)\cos(\omega_c t + \phi(t))$$

$$= \text{Re} \left[ a(t)e^{j(\omega_c t + \phi(t))} \right]$$

$$= \text{Re} \left[ a(t)e^{j\omega_c t}e^{j\phi(t)} \right]$$

$$= \text{Re} \left[ p(t)e^{j\omega_c t} \right]$$

(2.1)
where \( p(t) = a(t)e^{j\phi(t)} \) is the “complex envelope phasor”. Then let

\[
p(t) = i(t) + jq(t)
\]

(2.2)

where

\[
i(t) = a(t)\cos\phi(t)
\]

(2.3)

\[
q(t) = a(t)\sin\phi(t)
\]

Substituting back,

\[
s(t) = \text{Re} \left[ \{i(t) + jq(t)\} e^{j\omega_c t} \right]
\]

(2.4)

\[
= i(t)\cos(\omega_c t) - q(t)\sin(\omega_c t)
\]

Therefore, any real signal can be defined by the angular carrier frequency \( \omega_c \), the in-phase component \( i(t) \), and the quadrature component \( q(t) \) [55].

On the receiving side, \( i(t) \) and \( q(t) \) can be recovered from a real signal \( s(t) \) using, for example, mixing circuits operating in quadrature [55]:

\[
i'(t) = s(t) \cdot 2\cos(\omega_c t)
\]

\[
= [i(t)\cos(\omega_c t) - q(t)\sin(\omega_c t)] \cdot 2\cos(\omega_c t)
\]

(2.5)

\[
= 2i(t)\cos^2(\omega_c t) - 2q(t)\sin(\omega_c t)\cos(\omega_c t)
\]

Applying the trigonometric identities \( \cos^2 x = \frac{1}{2} [1 + \cos 2x] \) and \( \sin x \cos x = \frac{1}{2} \sin 2x \),

\[
i'(t) = i(t) + i(t)\cos(2\omega_c t) - q(t)\sin(2\omega_c t)
\]

(2.6)

As the last two terms are oscillating at the same frequency, \( 2\omega_c \), they can be attenuated using a low pass filter with that frequency in the stopband:

\[
i(t) \approx \text{lpf}\{i'(t)\}
\]

(2.7)
Similarly for the quadrature component,

\[ q'(t) = s(t) \cdot -2\sin(\omega_c t) \]
\[ = 2q(t)\sin^2(\omega_c t) - 2i(t)\cos(\omega_c t)\sin(\omega_c t) \tag{2.8} \]
\[ = q(t) - q(t)\cos(2\omega_c t) - i(t)\sin(2\omega_c t) \]

Applying the same low pass filter,

\[ q(t) \approx \text{lpf}[q'(t)] \tag{2.9} \]

As an example of IQ modulation applied to digital modulation schemes, consider Quadrature Phase Shift Keying (QPSK), shown in Figure 2.9. In this case 16 bits, 0100100001001001, are being encoded according to the modulation

\[ i(t) = \frac{A}{\sqrt{2}}[2b_0(t) - 1] \]
\[ q(t) = \frac{A}{\sqrt{2}}[2b_1(t) - 1] \tag{2.10} \]

where \( A \) is the desired amplitude and \( b_0(t)/b_1(t) \) is the most/least significant bit for the given time. These components are applied to a carrier wave as in Equation 2.4 to get \( s(t) \). QPSK being a 4-level modulation encodes 2 bits at a time, so \( s(t) \) contains 7 phase shifts between the 8 pairs of bits, as shown in the figure. It can be seen that, on the receiving side, it would be relatively easy to decode the bits again from \( i(t) \) and \( q(t) \), for example by using a matched filter and threshold detector.

Given this theoretical background of SDR, practical hardware implementations are available in devices at many price levels, roughly divided into commercial (4 to 5 figure prices) and consumer (2 to 3 figure prices.) A survey of consumer SDRs is given in Table 2.2. This table shows that in general, higher price leads to a larger tuning range, wider bandwidth, and increased bit depth, as well as transmit capability. The Airspy, HackRF, and BladeRF boards are shown in Figure 2.10, which illustrates the
additional circuitry required for transmission in the higher-end devices. A common characteristic of the HackRF, BladeRF, and USRP is an embedded FPGA, which may be required if USB or CPU bandwidth become a bottleneck [56].

Table 2.2: Survey of consumer SDRs, comparing main parameters. [53], [54], [57]–[60]

<table>
<thead>
<tr>
<th>Name</th>
<th>Price ($)</th>
<th>Tuning (MHz)</th>
<th>Max BW (MHz)</th>
<th>TX/RX</th>
<th>ADC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL-SDR</td>
<td>20</td>
<td>25 to 1750</td>
<td>2.8</td>
<td>RX only</td>
<td>8</td>
</tr>
<tr>
<td>SDRPlay</td>
<td>150</td>
<td>0.1 to 2000</td>
<td>10</td>
<td>RX only</td>
<td>12</td>
</tr>
<tr>
<td>Airspy</td>
<td>200</td>
<td>24 to 1800</td>
<td>20</td>
<td>RX only</td>
<td>12</td>
</tr>
<tr>
<td>HackRF</td>
<td>315</td>
<td>1 to 6000</td>
<td>20</td>
<td>Both</td>
<td>8</td>
</tr>
<tr>
<td>BladeRF</td>
<td>420</td>
<td>300 to 3800</td>
<td>40</td>
<td>Both</td>
<td>12</td>
</tr>
<tr>
<td>USRP</td>
<td>675</td>
<td>70 to 6000</td>
<td>56</td>
<td>Both</td>
<td>12</td>
</tr>
</tbody>
</table>
These devices and more proprietary SDRs have been used in both academic research and commercial/governmental applications. In academia, the USRP has been a popular choice. Zitouni et al used the USRP to implement the Zigbee protocol [61], a low power and low speed radio protocol commonly used in embedded devices. The SDR implementation operates on the 868 or 915MHz bands with a data rate
of 20kb/s or 40kb/s, respectively, using Differential Binary Phase Shift Keying (DBPSK) modulation and Direct-Sequence Spread Spectrum (DSSS) coding. The USRP is controlled by software running on the GNU Radio framework, which provides common signal processing “blocks” that are connected together into a “flowgraph”. This implementation was evaluated by measuring bit and packet error rates between transmitter and receiver, and it was found that both quantities remained close to their ideal lower bounds [56]. This is an example of how these devices are capable of implementing complex protocols with performance matching what would be expected of a traditional implementation.

Another use for the USRP is as an Intermediate Frequency (IF) transceiver for a low-cost, open source 60GHz experimentation platform. Zetterberg and Fardi designed open source transmit and receive boards based around a pair of 60GHz upconverter/downconverter chips, and they use the USRP to transmit and receive on the chips’ 70MHz IF. Along with this, they’ve developed an open source software infrastructure to allow relatively low-cost experimentation with this recently standardized band of spectrum. In particular, they use this infrastructure to implement a 60GHz Orthogonal Frequency-Division Multiplexing (OFDM) transceiver in software. They use this implementation to measure performance metrics like link budget, phase noise, and Multiple-Input, Multiple-Output (MIMO) gain for future researchers, and they note that results from this system will likely need to be extrapolated due to its relatively low transmit power, antenna gain, and bandwidth compared to more expensive testbeds [62].

The lower-cost BladeRF has also been used in research. Akhtyamov et al used this SDR with stratospheric High Altitude Balloons (HABs) to simulate a federated satellite network, where a satellite may have to relay communications through other
satellites in order to reach a ground station. Their simplified simulation used 2 HABs, each with a BladeRF and Raspberry Pi 2 on board, and a ground station also powered by a BladeRF. Using this equipment, the balloons were networked with inter-balloon radio links, and the ground station used a directional antenna to maintain a radio link with the balloons. The authors used the BladeRFs to implement a software Gaussian Minimum Shift Keying (GMSK) radio protocol suited to their needs (to transmit telemetry like position and altitude), with a data rate of 3.3kb/s. Through a 2.5-hour flight test of this system, they found that the link between the balloon network and ground station was never lost, and the inter-balloon link remained established for over 55 miles with 0.5W power. They conclude that SDR is a promising technology for communications interoperability in space applications [63].

NASA apparently thinks so too. Their current Space Network Ground Segment Sustainment (SGSS) project is updating the ground segment of the Tracking and Data Relay Satellite System (TDRSS) to a distributed SDR architecture and is expected to be operational in 2017. As part of this effort, the analog frontends and distribution channels are being replaced with early digitization, sample distribution over 10 gigabit Ethernet, and FPGA reprogrammable modulation and demodulation. The system uses mostly Commercial Off-The-Shelf (COTS) equipment, which is pooled and dynamically allocated. They say this pooling reduces the amount of necessary equipment, increases the flexibility of resource allocation, and eases system group, and they attribute SDR as the technology that makes this architecture possible [64].

Additionally, NASA has created the Space Telecommunications Radio System (STRS), which includes an open, standardized interface between radio components as well as a software library to abstract radio applications across STRS-compliant SDRs. This system is currently being flight tested with the Space Communications
and Networking (SCaN) Testbed, shown in Figure 2.11, which consists of 3 STRS-compliant SDRs from 3 different vendors operating from a truss of the International Space Station. The testbed is open to S-band, L-band, and Ka-band experimentation, and STRS software building blocks are being accumulated in an open repository for application development [65]. This and STRS in general demonstrate how decoupling radio applications from the underlying hardware can create a more open/non-proprietary and reusable development environment.

SDR isn’t always utilized in an “open” way, though: these concepts are also used in more proprietary applications by the military and private industry. The military
has an ongoing Joint Tactical Radio System (JTRS) program that’s set to begin hardware production in 2017. This program aims to reduce the number of radios that must be deployed and supported by replacing the separate radios used for each radio protocol with a single software-reconfigurable one. This radio has the goal of being interoperable across handheld, vehicular, airborne, and other military radio systems as well as adaptable to the frequency bands available in different areas of the world. An SDR architecture meets these requirements by allowing an RF core to be reconfigured on demand with a software upgrade or switchover [52], [66], [67].

Outside of the military, SDR concepts are in use by “multimode” system-on-chips for protocols including GSM, CDMA, WiFi, and Bluetooth [50], [52], and fully-integrated SDR products exist for both commercial and amateur users. An example commercial product is the Harris XG-100, which is a handheld radio for public safety applications that uses SDR to implement analog FM, P25 Phase I, and P25 Phase II communications [68]. Some example amateur products are the Icom IC-7300, Elecraft KX3, and Flexradio 6000 transceivers. These use SDR with upgradable firmware to operate on the 160 through 6m amateur bands and to modulate and demodulate AM, FM, SSB, and CW modes. Additionally, the Flexradio and Elecraft transceivers have IQ output ports for interfacing with external peripherals and computer applications [69]–[71].

Although SDR is suitable for many applications, it also has drawbacks compared to completely hardware implementations. For one, software is simply not as fast as circuitry. Certain applications, for example those requiring gigahertz of bandwidth, are limited by the rate at which samples can be processed, which includes the speed of the signal processing implementation as well as the speed of the bus pushing the samples into software. Additionally, SDR implementations can have higher power and
footprint requirements, especially if a general purpose processor is used: In contrast to the general, abstract sampling circuitry used by SDR devices, purpose-built circuitry can be electrically simplified and compacted into a smaller footprint [52].

With these tradedoffs taken into consideration, the benefits outweigh the drawbacks in this research. Most importantly, SDR makes this receiver more accessible, not only in the sense of affordability, but also in the sense of device assembly. Unlike a traditional custom RF receiver that would require the user to fabricate a circuit board and solder in components, this receiver requires the user to simply plug in the SDR and run the main application. In this case, the application uses the SDR to tune to P25 control and traffic channel frequencies (typically in the 700MHz band) and, with the stream of IQ samples, FM demodulate the signal there. Because of the relatively modest requirements, it’s able to use the very low-cost RTL-SDR, which is discussed in the next section.

2.2.2 The Affordable RTL-SDR

The RF signal processing section of the receiver studied here is implemented with software defined radio, specifically the RTL-SDR. This is a $20 USB SDR – slightly larger than a flash drive – that recently became more suitable for noninteractive radio applications, as required by this research. The ecosystem around it is still growing and includes a relatively small amount of academic research compared to the number of hobby and amateur applications available. Compared to some of these applications, the requirements of this P25 receiver are relatively modest, so the RTL-SDR performs well while making the receiver more powerful and accessible.

The RTL-SDR’s ability to function as an SDR is a result of tinkering and reverse engineering. Prior to 2012, these devices were known only as cheap USB dongles for Digital Video Broadcasting (DVB) TV reception. In fact, the main function of the
Realtek RTL2832 chip on these dongles is as a DVB demodulator that pushes MPEG video frames over USB for display on a computer. In 2012, however, it was found that this chip has additional functionality. By sniffing the USB traffic generated by a program for receiving broadcast FM radio, it was found that the RTL chip could be put into a “test” mode where it would instead output IQ samples over USB for processing by the host computer. Through further reverse engineering, the commands were found for tuning the center frequency, changing the sample rate, and otherwise controlling the device [15]. From there, a community formed, software infrastructure grew, and manufacturers began producing dongles more suited to SDR applications, with the label RTL-SDR.

The dongles selling around the $20 price level all had a common flaw, though: they used a relatively unstable “can” crystal as input to the clocks and oscillators. This component can be seen in Figure 2.12, with a $20 dongle that was popular in 2014 [72]. These crystals could require an initial frequency correction of 80ppm and, after warming up, small changes in ambient temperature would cause signals to drift several ppm around their center frequencies. This made noninteractive radio applications tedious to implement, usually requiring a carrier tracking layer to compensate [15] (in place of manual user interaction to continuously retune the frequency.) In August 2015, however, a new dongle was introduced that replaced the can crystal with a Temperature Compensated Crystal Oscillator (TCXO) and kept the $20 price. These dongles now have a maximum initial frequency correction of 3ppm and no significant drift while in use [53]. This recent development makes the technology more suitable for noninteractive applications by removing the need for additional software complexity to compensate for hardware quirks.
Figure 2.12: Circuitry of older RTL-SDR, showing common 28.8MHz “can” crystal.

The TCXO RTL-SDR is the specific device utilized for this receiver, and its circuitry is shown in Figure 2.13 (notice the layout remains similar to the dongle in Figure 2.12.) The figure points out the three main components of these devices, from left to right: the Realtek RTL2832 chip, the TCXO, and the Raphael Micro R820T2 tuner chip. The flow diagram in Figure 2.14 shows how this circuitry captures and samples RF signals. At the frontend, the real signal from the antenna is fed into the tuner, where it’s first amplified with a Low-Noise Amplifier (LNA) whose gain is controllable from software. Then, it enters a mixing circuit, where it’s mixed with a Local Oscillator (LO) and shifted down to a low IF. The LO frequency is determined by a software-controlled center frequency, which can range from 25 to 1750MHz. Next, the signal enters an Automatic Gain Control (AGC) type amplifier, which can be enabled or disabled from software, and is finally fed into the RTL chip.

From there, the real, analog signal is digitized with a high-speed, 8-bit Analog to Digital Converter (ADC), resulting in a discrete signal. The in-phase and quadrature components are then recovered from this signal using Numerically Controlled Oscillators (NCOs) and mixing circuits similar to those in Equations 2.5 and 2.8.
These mixers also shift the spectrum down to a zero IF, around DC. After that, the signals are decimated to the sample rate requested by software, which can range from 196kS/s (kilosamples per second) to 2.8MS/s. Finally, the IQ samples are streamed over USB as interleaved byte pairs – an 8-bit in-phase sample followed by an 8-bit quadrature sample, and so on. Because of the use of complex IQ sampling, the resulting spectrum has bandwidth equal to the sample rate and can be asymmetric around DC [15]. The design of this RF receiver is very similar to the linear receiver architecture described by Kenington [48].

This overview reveals some shortcomings of the RTL-SDR. First, it’s obviously a receive-only device, which may rule out its use in some applications but may be beneficial in others: lack of transmitting circuitry means a smaller footprint and less power usage for applications that don’t need this functionality. Second, samples are only 8 bits each. This results in a relatively small dynamic range, which makes these devices more susceptible to overload by strong noise and other high-power, out-of-band signals. Finally, the maximum sample rate is relatively low and unsuitable for some protocols, for example, higher-capacity LTE cellular channels. For the price,
Figure 2.14: Components used by the RTL-SDR to capture and sample RF signals.

though, the RTL-SDR provides good value and performs well in applications with compatible requirements.

So far, there has been a small amount of academic work published that uses the device. Grunroos et al used 4 RTL-SDRs to create a distributed spectrum sensing network. Spectrum sensing is used when dynamically allocating RF spectrum (for example in cognitive radio applications) to determine spectrum usage in a geographical area. The authors designed a distributed system to do this that combines data from many low-cost RTL-SDR nodes. Each RTL-SDR is interfaced with a Raspberry Pi 2 computer and is used with a 1.25MS/s sample rate to capture slices of spectrum from 110 to 1200MHz (which takes about a minute.) The nodes then
perform an FFT on each slice and store the magnitudes to a central database. Finally, this data is combined together by a central computer to determine spectrum usage. Through comparison with a commercial RF sensing device, the authors found their solution had 93% agreement while being significantly cheaper, although significantly slower too [73].

The RTL-SDR is also being investigated for use in education. Stewart et al have developed a 630-page workbook of hands-on communications engineering exercises for students, based around the RTL-SDR and Matlab/Simulink. These exercises range from basic spectrum exploration to implementation of a QPSK receiver (a sample receiver diagram is shown in Figure 2.15.) In general, they note how greatly learning opportunities are expanded when students can carry around an SDR in their pocket [15]. Similarly, Prust et al have designed a mobile project studio for students based around the RTL-SDR. Each studio is equipped with a laptop running Matlab and GNU Radio and interfaced with an RTL-SDR dongle, resulting in a package affordable enough for distribution to a classroom of students. They suggest projects like broadcast FM radio demodulation and NOAA weather satellite image reception. Again, they note how the RTL-SDR allows hands-on application of common communications engineering topics like multirate signal processing and digital filter design [74].

Outside of academia, there is a sizable software and hardware ecosystem available for hobbyists and radio amateurs. Applications like Dump1090 [75] and AISDeco2 [76] receive and decode Automatic Dependent Surveillance Broadcast (ADSB) and Automatic Identification System (AIS) location information transmitted by planes and boats, respectively, to generate real-time, virtual radars. This information is being crowdsourced by FlightAware, who have created a custom RTL-SDR ADSB decoding
package that uploads location data to their online flight tracking application [77]. Similar to the receiver studied here for P25 voice messages, there are applications to receive Terrestrial Trunked Radio (TETRA) – which is more popular in Europe – voice and data messages [78], [79]. These applications also take advantage of the SDR for trunking awareness. There are also applications for receiving satellite communications, including Inmarsat data messages [80], NOAA Automatic Picture Transmission (APT) weather satellite images [81], and Russian Meteor Low-Resolution Picture Transmission (LRPT) weather satellite images [82]. A final example of interesting
software is WebSDR, which streams data from an SDR to users across the Internet. Users can change the center frequency, filter parameters, and demodulation modes through a browser application to explore the radio spectrum in foreign areas. This can be used, for example, to test worldwide signal propagation or experience shortwave phenomena like the Russian “Buzzer.” Some WebSDR sites have combined multiple RTL-SDRs to (incoherently) increase the bandwidth available for use [83].

The hardware ecosystem includes devices and modifications to widen the tuning range and increase the performance of the RTL-SDR. For receiving the High Frequency (HF) bands, several sources [84]–[87] exist for upconverter accessories, which typically use mixing and filtering circuitry to shift lower frequency signals up into the tuning range of the device (with supporting software to make tuning easier.) Alternatively, the “direct sampling modification” [88] can be used to connect an end-fed antenna directly to one of the pins of the RTL chip and, bypassing the tuner chip, allow the device to receive lower frequencies. For receiving frequencies above the upper tuning range, one solution is to use a Low-Noise Block downconverter (LNB) [89] – typically attached to a satellite TV dish – which shifts the signals down to an IF within the tuning range of the device. A common modification to improve performance is to add a form of shielding. New dongles typically come in plastic cases, which makes them – like any RF circuitry – more susceptible to parasitics and environmental noise. These effects can be reduced by enclosing the dongles in a Faraday cage of metal shielding. Some manufacturers sell custom aluminum cases [90], [91], but aluminum or copper foil can also be used [92]. The dongles in this research were all shielded with copper tape, as this can be easily wrapped around an insulating layer and readily soldered to the USB and RF grounds. This creates a
low-impedance return path for unwanted signals and is less likely to be affected by oxidation. It typically results in a noticeable 6dB drop in the noise floor.

Compared to some of the software available, the requirements for this P25 receiver are relatively modest. The receiver needs to tune between control and traffic channel frequencies (typically in the 700MHz band) and FM demodulate the signal there. These frequencies easily fall within the tuning range of the RTL-SDR, and changing the center frequency is accomplished with a single function call. Then, to design the FM demodulator, two popular open source spectrum exploration applications were used for reference. The first, GQRX [93], is implemented with the GNU Radio framework and uses the signal processing approach in Figure 2.16a for FM demodulation. Similarly, CubicSDR [94] is implemented with the LiquidSDR library and uses the approach in Figure 2.16b.

The approaches are similar, with the first step being to convert the interleaved byte pairs received from the RTL-SDR to floating point complex number objects (with real and imaginary parts in the range $[-1, 1]$.) Next, the samples are fed into a software mixer to shift the desired signal within the available bandwidth to DC. After that, since both allow setting an arbitrary SDR sample rate, the signal is decimated to the baseband sample rate using a rational resampler, with anti-alias filters that are designed at runtime (for example with Parks-McClellan.) With the signal decimated to a lower sample rate, GQRX applies a channel select filter using a Finite Impulse Response (FIR) bandpass filter designed at runtime with tunable parameters. This filter likely helps reduce out-of-channel interference compared to CubicSDR. After analog squelch steps, the signal is finally FM demodulated. Both do this with a common digital demodulation method: calculating the change in phase
Figure 2.16: Similar approaches to FM demodulation used by two popular, open source SDR applications.

at each complex sample [95],

\[ x[t] = \omega^{-1}_\Delta f_s \arg(p[t]p[t-1]^*) \]  \hspace{1cm} (2.11)

where \( \omega_\Delta \) is the angular FM deviation, \( f_s \) is the baseband frequency, and \( p[t] \) is the complex sample for time \( t \). This creates a stream of real samples \( x[t] \) at the baseband sample rate. To see how this functions as FM demodulation, first consider the classical
equation for an FM signal [96]:

\[ s(t) = a(t) \cos(\omega_c t + \phi(t)) \]  

(2.12)

where

\[ \phi(t) = \omega_\Delta \int_0^t x(\tau)d\tau \]  

(2.13)

and \( x(t) \) is the modulating signal to be recovered. From this it’s clear that

\[ \frac{d\phi(t)}{dt} = \omega_\Delta x(t) \]  

(2.14)

so

\[ x(t) = \omega_\Delta^{-1} \frac{d\phi(t)}{dt} \]  

(2.15)

Because differentiation in continuous time is approximated by finite backwards difference in discrete time [97],

\[ x(t) \approx \omega_\Delta^{-1} \Delta_T [\phi](t) = \omega_\Delta^{-1} \frac{1}{T} (\phi[t] - \phi[t - 1]) = \omega_\Delta^{-1} f_s (\phi[t] - \phi[t - 1]) \]  

(2.16)

Then applying Equations 2.1 and 2.2 to Equation 2.12,

\[ i(t) = a(t) \cos \phi(t) \]

\[ q(t) = a(t) \sin \phi(t) \]  

(2.17)

\[ p(t) = a(t) \cos \phi(t) + ja(t) \sin \phi(t) \]

Evaluating the complex argument of this,

\[ \arg(p(t)) = \phi(t) \]  

(2.18)

so

\[ \arg(p(t)) - \arg(p(t - 1)) = \phi(t) - \phi(t - 1) \]  

(2.19)

Applying the complex number identities \( \arg(uv) \equiv \arg(u) + \arg(v) \) (mod \( (-\pi, \pi) \)) and \( \arg(u^*) = -\arg(u) \) [98], [99],

\[ \arg(p(t)p(t - 1)^*) = \arg(p(t)) - \arg(p(t - 1)) = \phi(t) - \phi(t - 1) \]  

(2.20)
Finally, combining Equations 2.16 and 2.20 leads to Equation 2.11.

With these two applications as reference, the signal processing approach used for this receiver is shown in Figure 2.17. Importantly, this design takes advantage of the more specific requirements in order to optimize and remove certain steps. This became important due to the relatively low clock speed and less advanced floating point and Single Instruction, Multiple Data (SIMD) instruction sets of the Raspberry Pi 2. In a similar design choice, the signal processing was implemented as a small, decoupled module in favor of GNU Radio or LiquidSDR. These packages give relatively little control over heap memory allocations and threading, which are important to the performance of real-time, multithreaded signal processing on the Raspberry Pi. In contrast, this custom module is completely allocation free and thread safe, and it takes advantage of static inlining and vectorization compiler optimizations.

The first optimization in the signal processing of this application is to use a fixed SDR sample rate of 240kS/s. Although a P25 channel would theoretically require a sample rate around just 12.5kS/s, the minimum sample rate of the RTL-SDR is 196kS/s. A slightly higher sample rate of 240kS/s is a clean multiple of the baseband 48kS/s sample rate, and this circumstance simplifies the following decimation step. The next optimization stems from tuning each channel frequency as the RTL-SDR center frequency, which removes the need for a software mixer. Next, due to the fixed input and output sample rates and constant downsampling factor, the decimation step is optimized to use a custom (see Appendix A.4) anti-aliasing filter. These optimizations also allow enough headroom for a channel select filter, as used by GQRX, which is also custom designed (see Appendix A.5) for the fixed P25 channel width. Finally, the output of this filter is input to the FM demodulator, which uses
Equation 2.11 to create a stream of real samples at the fixed baseband sample rate. This stream is then suitable for input to the P25 decoder.

2.3 P25 Software Receivers

P25, being a digital, packetized protocol, is suited to software implementation, and this has been realized in a small number of applications available for receiving P25 voice messages. None of these applications, however, satisfy the requirements of being both automatic – noninteractively switching between multiple traffic channels – and portable – able to run on an embedded platform like the Raspberry Pi. Further, because of inherent design choices in their implementations, it would be difficult to
adapt this software to these requirements. Still, this software provides a reference for the receiver studied here.

There exists at least one academically-motivated P25 receiver: OP25 [100]. For this, Glass et al created several GNU Radio blocks to receive P25 voice and data messages in order to study the security of P25’s encryption implementation. These blocks can be connected together (along with other GNU Radio blocks) to form the receiver in Figure 2.18. This receiver takes input samples from the USRP SDR and eventually outputs voice messages to an IMBE decoder and data messages to Wireshark, a protocol analyzer. For baseband symbol decoding, they use an external 4-Frequency Shift Keying (4FSK) block, and for frame synchronization, they use a correlation approach. Although the software is SDR aware, it doesn’t support automatic operation using multiple traffic channels: The authors note there is no support for trunking awareness, and there is no built-in support for the alternative, channel scanning (although there may be external GNU Radio blocks to implement this.) The software has been released as open source, but it hasn’t been updated for API changes in GNU Radio, so it no longer builds. As a workaround, older versions of GNU Radio and supporting software must be built and maintained.

Other software available is DSD [101], SDRTrunk [102], DSD+ [103], and Unitrunker [104]. DSD is an open source, cross-platform, command-line application that can receive several digital voice protocols, including Digital Mobile Radio (DMR), Digital Smart Technologies for Amateur Radio (D-STAR), and P25. It takes as input an FM demodulated signal, either from an audio cable attached to a tapped discriminator (on an external radio) or from external software (such as rtl_fm.) As such, it has no trunking or SDR awareness and no support for channel scanning. Additionally, the default symbol decoder performance seems to degrade over time.
Figure 2.18: Architecture of OP25 receiver. [100]

until, after several minutes, almost no voice messages are successfully decoded. Manually restarting the application seems to fix this behavior. This may be due to the “adaptive” decoding heuristic it uses, which gets reset when the application is restarted.

The remaining open source application, SDRTrunk, is the most mature of the choices in this category. It’s both SDR and trunking aware, so it uses control channel trunking packets for switching between traffic channels. Unfortunately, the application has no headless interface and is clearly written for GUI interaction, with GUI logic and P25 logic combined within modules. Additionally, because the application is written in Java, the memory and CPU requirements may not be suitable for real-time SDR signal processing on the Raspberry Pi (for example, the application uses garbage-collected heap allocations in some of its processing routines.)

Along with these open source, cross-platform options, there are also two closed source, platform-restricted options: Unitrunker and DSD+. Both of these are Windows-only GUI applications. Unitrunker supports trunking awareness (including parallel channel decoding with 2 RTL-SDRs), and DSD+ claims to have improved
Decoding performance compared to DSD. Both are incompatible with this research, though, due to their platform restriction and, being closed source, are not modifiable. Although the Raspberry Pi 2 supports a version of Windows called “Windows 10 IOT”, neither of these applications provide ARM builds necessary to run on this architecture. Additionally, applications for this version of Windows must use the “Universal Windows Application” framework, so the authors would have to (nontrivially) redesign their software [105].

One component that all these applications have in common is IMBE voice decoding – recovering an analog voice signal from digital IMBE frames. This is made slightly more complicated due to IMBE using algorithms that are under patent by Digital Voice Systems, Inc (DVSI). The IMBE encoding and decoding processes are an open standard, as part of the P25 Phase I suite of documents [44], but using that standard to distribute hardware products or software binaries that implement IMBE encoding/decoding requires that a license fee be paid to DVSI [106], [107]. So, as the OP25 authors point out, IMBE decoding is most commonly implemented with an external component, either a hardware board sold by DVSI or an open source software library (distributed as source code) [100]. Because the cheapest boards sold by DVSI range from $499 to $925 [108], the software library is the most attractive choice. Both OP25 and DSD use the mbelib library [106], which is written by the author of DSD. This library isn’t terribly optimized (for example, it uses arrays of Booleans to represent bits) and diverges from the standard in unvoiced signal generation (which leads to relatively poor quality recovered voice.) SDRTrunk uses the Java JMBE library [107], which is written by the SDRTrunk author but, again, external to the main application.
With this prior work for comparison, the receiver created for this research was implemented as several general, decoupled modules, with a main application specifically targeted to the Raspberry Pi 2. The dependency graph of the main application in Figure 2.19 illustrates this architecture (with modules written for this research shown in blue.) The two largest modules are p25 and imbe. The p25 module implements all the components necessary for a P25 receiver, starting with baseband deemphasis filtering, to frame synchronization and symbol decoding, to packet reconstruction and error correction, to (at the highest level) convenience state machines for handling all the expected and unexpected events that can occur when receiving P25 air interface messages. This module exports a public API suitable for any receiving application – including both GUI and headless – and can be easily used as an external library with Rust’s Cargo build system. Additionally, it contains zero heap allocations, and the frame synchronization and symbol decoding components were academically designed and studied to characterize their expected performance (see Sections 3.1 and 3.2.)

Figure 2.19: Dependency graph of Rust modules.
Similar to the other projects, IMBE decoding is implemented as a module external to the main application and will not be distributed in binary form. This module decodes complex IMBE voice in real-time on the Raspberry Pi 2 – with voiced and unvoiced signals synthesized in accordance with the standard, unlike mbelib. This is made possible with optimizations like replacing expensive sine and cosine function calls in tight loops with a few multiplications and additions (based on trigonometric identities and implemented in the quad_osc module), multithreading the final synthesis step for a 1.6× speedup, and again using zero heap allocations.

These two modules, and the others in Figure 2.19, are imported and composed by the main application to implement a trunking-aware receiver for P25 voice messages that runs in real-time on the Raspberry Pi 2. To do this, the main application parallelizes the work across threads, as in Figure 2.20, and creates the communication channels for these threads to share data and events. This architecture benefits from Rust, whose type system statically guarantees that parallel code is free of data races and memory errors (but doesn’t protect against asynchronous faults like deadlocks) and whose standard library includes building blocks for parallel applications like lockless, multi-producer/single-consumer “channels”.

At the lowest layer of the receiver are the SDR Controller and Reader threads. The SDR Controller receives events to change frequency and tuner gain and communicates those changes to the hardware. The SDR Reader continuously reads chunks of samples from the RTL-SDR and buffers them to a lockless memory pool. These chunks are sent to the Demod thread, which uses signal processing blocks from the dsp module to FM demodulate them using the approach in Figure 2.17. This creates chunks of baseband samples that are sent to the P25 Decoder thread, which uses a high-level state machine from the p25 module to handle all the different
P25 packets and events. This thread also implements simple trunking awareness by monitoring control channel trunking packets and traffic channel terminator packets. The “Group Voice Update” trunking packet signals when a voice message is starting and what traffic channel it’s assigned, and the terminator packet signals the end of a voice message. The thread uses this information to switch between control and traffic channel frequencies and to update the UI with the talkgroup and channel frequency of the current conversation. Finally, whenever this thread encounters a voice message IMBE frame, it sends the frame to the IMBE Decoder thread. This thread uses the `imbe` module to decode each frame into a chunk of analog voice samples and writes these to a named FIFO pipe. On the other end of the pipe is the `aplay` system program, which interfaces with the Advanced Linux Sound Architecture (ALSA) subsystem to output the audio to the soundcard (and also resample and reformat the samples if necessary.) The remaining threads monitor user input, maintain the application state, and redraw the LCD when necessary.
This application requires no user interaction, so it can be ran on a “headless” system as part of a startup service. It is customizable, however, through a configuration file and talkgroup database. The configuration file consists of multiple P25 sites, each of which has a name, control channel frequency, and a set of traffic channel frequencies (which can be looked up on the FCC Universal Licensing System (ULS) database [109].) The talkgroup database is a Comma Separated Values (CSV) file that maps talkgroup hexadecimal IDs to descriptive name (these mappings can be found in databases like RadioReference [110].) All of this software is released free and open source, and most of the supporting modules – including p25 and imbe – are cross-platform compatible. Because of this, the receiver can be freely modified and enhanced (unlike consumer devices), and the software is suitable as a foundation for future P25 receivers and experiments.
3 Evaluation

3.1 Baseband Symbol Decoding

P25’s C4FM modulation uses 4 symbols to encode 2 bits each: 00, 01, 10, and 11. Each symbol is encoded as an impulse of varying magnitude, as shown in Table 3.1, such that there are 4,800 impulses per second. Therefore, with a sample rate of 48,000 samples per second, an impulse appears every 10 samples. These magnitudes can be scaled by any factor so long as their relative scaling of $1, \frac{1}{3}, -\frac{1}{3}, -1$ remains constant [16].

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>+1800</td>
</tr>
<tr>
<td>00</td>
<td>+600</td>
</tr>
<tr>
<td>10</td>
<td>-600</td>
</tr>
<tr>
<td>11</td>
<td>-1800</td>
</tr>
</tbody>
</table>

The resulting impulse train is then filtered with a raised cosine filter followed by a preemphasis filter, whose ideal frequency responses are given in the CAI standard [16]. The FIR filters used here are derived from the ones given by Ramsey [43] (see Appendix sections A.1 and A.2 for design parameters.) This filtered waveform is then transmitted as the baseband signal and must be decoded by a receiver to recover the symbols. A simple version of such a receiver could read the sample at each sampling instant (assuming perfect symbol clock timing), compare it to a set of thresholds derived from Table 3.1, and pick the associated symbol with the closest match. This is the type of receiver studied here.
3.1.1 Procedure

The first task was to characterize the distribution of samples found at sampling instants for each C4FM symbol. To start, a large sample set was created by generating 4,000,000 random symbols, chosen from a uniform distribution using a random number generator seeded with \([0x2A; 4]\). After every 4,000 random symbols, a 24-symbol frame synchronization sequence was inserted for the experiment in Section 3.2. All of these symbol values were written into a file for later association with sample values. At the very beginning of the generated symbols was added a 40-symbol 00 10 00 10 \ldots sequence to create a 2.4kHz sinusoid [16] for manual symbol clock determination. This sequence was otherwise skipped over in later sampling and not included in the symbol value file.

These symbols were then modulated into an impulse train at 48,000 samples per second using the magnitudes in Table 3.2 to create 4 waveforms. The Nominal waveform is the ideal C4FM signal with no distortion. The Scaled waveform represents a common real-world “distortion”, where the received impulses have some arbitrary (but still consistent relative to each other) amplitude due to scaling by an unknown factor between transmitter and receiver; in this case a scaling factor of 0.4 was applied to each impulse magnitude. The Offset waveform represents a signal affected by another common distortion of negative (or positive) DC bias, where all the impulse magnitudes are simply shifted by the same amount (−0.3 in this case.) The final Both waveform has both scaling and offset distortion by the above amounts.

The resulting impulse train for each waveform was then filtered with the FIR filters described previously. Each filtered waveform consisted of over 40,000,000 32-bit floating point samples, for a little over 15 minutes of P25 baseband “audio”. These unprocessed waveforms were then subjected to various amounts of average
white Gaussian noise using Matlab’s `awgn` function. The function was applied in the 'measured' mode, so noise power was relative to the Matlab-measured power of the signal [111], and 5 SNR (dB) values were used: 12:1, 6:1, 3:1, 1:1, and 0.001:1. The result of all this was 20 baseband files, 5 per waveform, for use in later experiments. An example of the effects of this noise is given in Figure 3.1.

To determine the first sampling instant, the Nominal unprocessed waveform was first filtered with the receiver’s deemphasis filter (see Appendix A.3 for design parameters), as required by the standard [16], to induce the FIR time shift, and the resulting waveform was manually inspected. The sampling instant for the first
Figure 3.1: Nominal waveform in unprocessed form and after application of AWGN at 12:1 through 0.001:1 SNR.

random symbol was derived from the preceding sinusoidal sequence by finding the symbol clock for that sequence (an impulse occurs at each peak in the sinusoid) and adding 10 samples to the last sampling instant in it. From this, the first sampling instant for the sequence of studied symbols was found to occur at 498 samples into the file, and this was verified to be the same with the other waveforms.

Using this information, the sample value at the first sampling instant and every 10 samples after was recorded and associated with the sequence of symbols in the
symbol file created earlier, and this was done for all 20 baseband files. Then, a 20-bin histogram was created for the sample values found for each symbol in each file, and the mean $\mu$ and standard deviation $\sigma$ were calculated. As would be expected with data like this [112], the resulting histograms fit perfectly to Gaussian Probability Density Functions (PDFs) with the same $\mu$ and $\sigma$. Using these PDFs we can easily calculate

$$P(a \leq \text{sample} \leq b \mid tx = X) = P(a \leq S \leq b \mid X)$$  \hspace{1cm} (3.1)$$

for some sample value bounds $a$ and $b$ and transmitted symbol $X \in \{00, 01, 10, 11\}$ by integrating the area under the associated curve [113]. Applying Bayes’ rule, we can then calculate the probability that the transmitted symbol for the current sample was $X$ given that the sample value falls between $a$ and $b$:

$$P(X \mid a \leq S \leq b) = \frac{P(a \leq S \leq b \mid X)P(X)}{P(a \leq S \leq b)}$$  \hspace{1cm} (3.2)$$

where

$$P(a \leq S \leq b) = P(a \leq S \leq b \mid 00)P(00) + \cdots + P(a \leq S \leq b \mid 11)P(11)$$  \hspace{1cm} (3.3)$$

Assuming an equal chance of any symbol being transmitted (which was verified to be true on this dataset but may be weighted in some way with real-world transmissions),

$$P(00) = P(01) = P(10) = P(11) = \frac{1}{4}$$  \hspace{1cm} (3.4)$$

Substituting all this back into Equation 3.2 and simplifying,

$$P(X \mid a \leq S \leq b) = \frac{P(a \leq S \leq b \mid X)}{\sum_{Y \in \{00, 01, 10, 11\}} P(a \leq S \leq b \mid Y)}$$  \hspace{1cm} (3.5)$$

We can also calculate the probability that the transmitted symbol wasn’t $X$ for the current sample [114]:

$$P(tx \neq X \mid a \leq S \leq b) = P(X^c \mid a \leq S \leq b) = 1 - P(X \mid a \leq S \leq b)$$  \hspace{1cm} (3.6)$$
With the distribution of samples characterized, the PDFs were then used to calculate the ideal sampling threshold for each symbol. These thresholds were then generalized into a set of equations that work across all the waveforms. Finally, a threshold decoder based on these equations was implemented in software – using hardcoded means for each waveform – and evaluated on a new set of random symbols. Again, 4,000,000 random symbols were chosen from a uniform distribution, this time using the seed [0xBE; 4], and the same frame synchronization interleaving and sinusoid prefixing were applied. The symbol file was then created in the same way as above and verified to be different from the previous one. The symbols were then modulated and filtered for all 3 waveforms using the magnitudes from Table 3.2, and the resulting files were processed with Matlab’s `awgn` using the same settings to create another set of 20 baseband files. These files were then ran through the implemented decoder, with the decoder outputting its symbol decision, rather than the sample value, at each sampling instant. Finally, these symbol decisions were compared with those in the true symbol file to find the number of errors made by the decoder.

### 3.1.2 Results

The Gaussian distribution of sample values is shown for each SNR in Figures 3.2 through 3.4, with the associated histogram filling each curve. These PDFs in all 4 waveforms are affected in the same general way by increased noise: spreading and overlapping with adjacent PDFs as \( \sigma \)s increase. This increase in \( \sigma \) with noise can be seen quantitatively in Table 3.3. The table also shows that \( \mu \) for each symbol remains constant per waveform regardless of noise, and all \( \mu \) have less than 1% error compared to the values in Table 3.2.
Figure 3.2: Histograms and PDFs for Nominal waveform, for various SNR.
Figure 3.3: Histograms and PDFs for Scaled waveform, for various SNR.
Figure 3.4: Histograms and PDFs for Offset waveform, for various SNR.
Figure 3.5: Histograms and PDFs for Both waveform, for various SNR.
Table 3.3: Distribution of symbol values as $\mu \pm \sigma$.

(a) Nominal

<table>
<thead>
<tr>
<th>SNR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:1</td>
<td>0.060 ± 0.012</td>
<td>0.181 ± 0.012</td>
<td>-0.060 ± 0.012</td>
<td>-0.181 ± 0.012</td>
</tr>
<tr>
<td>6:1</td>
<td>0.060 ± 0.023</td>
<td>0.181 ± 0.023</td>
<td>-0.060 ± 0.023</td>
<td>-0.181 ± 0.023</td>
</tr>
<tr>
<td>3:1</td>
<td>0.060 ± 0.033</td>
<td>0.181 ± 0.033</td>
<td>-0.060 ± 0.033</td>
<td>-0.181 ± 0.033</td>
</tr>
<tr>
<td>1:1</td>
<td>0.060 ± 0.041</td>
<td>0.181 ± 0.041</td>
<td>-0.060 ± 0.041</td>
<td>-0.181 ± 0.041</td>
</tr>
<tr>
<td>0.001:1</td>
<td>0.060 ± 0.046</td>
<td>0.181 ± 0.046</td>
<td>-0.060 ± 0.046</td>
<td>-0.181 ± 0.046</td>
</tr>
</tbody>
</table>

(b) Scaled

<table>
<thead>
<tr>
<th>SNR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:1</td>
<td>0.024 ± 0.005</td>
<td>0.073 ± 0.005</td>
<td>-0.024 ± 0.005</td>
<td>-0.073 ± 0.005</td>
</tr>
<tr>
<td>6:1</td>
<td>0.024 ± 0.009</td>
<td>0.073 ± 0.009</td>
<td>-0.024 ± 0.009</td>
<td>-0.073 ± 0.009</td>
</tr>
<tr>
<td>3:1</td>
<td>0.024 ± 0.013</td>
<td>0.073 ± 0.013</td>
<td>-0.024 ± 0.013</td>
<td>-0.073 ± 0.013</td>
</tr>
<tr>
<td>1:1</td>
<td>0.024 ± 0.017</td>
<td>0.073 ± 0.017</td>
<td>-0.024 ± 0.017</td>
<td>-0.073 ± 0.016</td>
</tr>
<tr>
<td>0.001:1</td>
<td>0.024 ± 0.018</td>
<td>0.073 ± 0.019</td>
<td>-0.024 ± 0.019</td>
<td>-0.073 ± 0.019</td>
</tr>
</tbody>
</table>

(c) Offset

<table>
<thead>
<tr>
<th>SNR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:1</td>
<td>0.030 ± 0.012</td>
<td>0.151 ± 0.012</td>
<td>-0.090 ± 0.012</td>
<td>-0.211 ± 0.012</td>
</tr>
<tr>
<td>6:1</td>
<td>0.030 ± 0.024</td>
<td>0.151 ± 0.024</td>
<td>-0.090 ± 0.024</td>
<td>-0.211 ± 0.024</td>
</tr>
<tr>
<td>3:1</td>
<td>0.030 ± 0.033</td>
<td>0.151 ± 0.033</td>
<td>-0.090 ± 0.033</td>
<td>-0.211 ± 0.033</td>
</tr>
<tr>
<td>1:1</td>
<td>0.030 ± 0.042</td>
<td>0.151 ± 0.042</td>
<td>-0.090 ± 0.042</td>
<td>-0.211 ± 0.042</td>
</tr>
<tr>
<td>0.001:1</td>
<td>0.030 ± 0.047</td>
<td>0.151 ± 0.047</td>
<td>-0.090 ± 0.047</td>
<td>-0.211 ± 0.047</td>
</tr>
</tbody>
</table>

(d) Both

<table>
<thead>
<tr>
<th>SNR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:1</td>
<td>-0.006 ± 0.005</td>
<td>0.043 ± 0.005</td>
<td>-0.054 ± 0.005</td>
<td>-0.103 ± 0.005</td>
</tr>
<tr>
<td>6:1</td>
<td>-0.006 ± 0.010</td>
<td>0.043 ± 0.010</td>
<td>-0.054 ± 0.010</td>
<td>-0.103 ± 0.010</td>
</tr>
<tr>
<td>3:1</td>
<td>-0.006 ± 0.015</td>
<td>0.043 ± 0.015</td>
<td>-0.054 ± 0.015</td>
<td>-0.103 ± 0.015</td>
</tr>
<tr>
<td>1:1</td>
<td>-0.006 ± 0.018</td>
<td>0.043 ± 0.018</td>
<td>-0.054 ± 0.018</td>
<td>-0.103 ± 0.018</td>
</tr>
<tr>
<td>0.001:1</td>
<td>-0.006 ± 0.021</td>
<td>0.043 ± 0.021</td>
<td>-0.054 ± 0.021</td>
<td>-0.103 ± 0.021</td>
</tr>
</tbody>
</table>
The differences in distribution due to distortion can be seen in Figure 3.6, which compares the 4 waveforms at 3:1 SNR. In comparison to Nominal, Scaled is simply compressed closer to zero due to the fractional scaling factor, Offset has identically-shaped PDFs that are simply shifted left by the offset amount, and Both has a combination of the previous two effects, as expected.

Using all this data we can start designing the thresholds for the simple receiver studied here. As mentioned earlier, the receiver should decide at each sampling instant the most likely transmitted symbol for the current sample value. Additionally, it
should make a hard decision at each instant (a soft/ambiguous decider isn’t considered here.) With 4 symbols, therefore, the receiver needs 3 thresholds: a lower threshold \( L \) to separate 11 and 10, a middle threshold \( M \) to separate 10 and 00, and an upper threshold \( U \) to separate 00 and 01. The probability of decoding each symbol in that case is

\[
P(\text{rx} = 11) = P(-\infty < S \leq L) \\
P(\text{rx} = 10) = P(L < S \leq M) \\
P(\text{rx} = 00) = P(M < S \leq U) \\
P(\text{rx} = 01) = P(U < S < \infty)
\]

From these we can calculate the probability of a correct decoding with, for example, \( P(tx = 11 \mid \text{rx} = 11) = P(11 \mid -\infty < S \leq L) \), which can be computed using Equation 3.5. Similarly, we can calculate the probability of a misclassification using Equation 3.6 with the threshold bounds for the desired symbol. Since we’re assuming that all symbols have equal probability of being transmitted, neither symbol in each pair of symbols separated by a threshold should be considered more important. In that case the optimal thresholds that minimize the probability of misclassification are at the intersections of the Gaussian PDFs [115] – simply the value halfway between the \( \mu \) of each symbol. These intersection points, which were constant per waveform regardless of noise level, are shown in Table 3.4.

Now we can starting designing the equations to compute these thresholds. In the ideal case of a distortionless channel, we could simply hardcode the thresholds:

\[
L = -\frac{2}{3} \cdot 0.18 = -0.12, \quad M = 0, \quad U = \frac{2}{3} \cdot 0.18 = 0.12
\]

This works well for the Nominal waveform, but it immediately breaks down in the presence of scaling distortion. To fix this, we can replace the hardcoded ideal means
Table 3.4: Gaussian intersection points.

<table>
<thead>
<tr>
<th>SNR</th>
<th>Lower</th>
<th>Middle</th>
<th>Upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>-0.121</td>
<td>0.000</td>
<td>0.121</td>
</tr>
<tr>
<td>Scaled</td>
<td>-0.049</td>
<td>0.000</td>
<td>0.049</td>
</tr>
<tr>
<td>Offset</td>
<td>-0.151</td>
<td>-0.030</td>
<td>0.091</td>
</tr>
<tr>
<td>Both</td>
<td>-0.079</td>
<td>-0.030</td>
<td>0.019</td>
</tr>
</tbody>
</table>

with those from Table 3.3 (how to determine the means for an arbitrary C4FM signal is studied in Section 3.2):

\[ L = \frac{2}{3} \cdot \mu_{11}, \quad M = 0, \quad U = \frac{2}{3} \cdot \mu_{01} \] (3.12)

Then for the Nominal waveform, \( L = -0.121, M = 0, U = 0.121 \), and for the Scaled waveform, \( L = -0.049, M = 0, U = 0.049 \), which works well for both. Again, though, the thresholds calculated this way are far from ideal in the presence of another form of distortion: DC offset. This can be fixed by replacing implicit uses of true zero with the offset “zero”:

\[ L = M + \frac{2}{3}(\mu_{11} - M), \quad M = \frac{1}{2}(\mu_{11} + \mu_{01}), \quad U = M + \frac{2}{3}(\mu_{01} - M) \] (3.13)

Applying this equation to all 4 waveforms gives the thresholds in Table 3.5, which match perfectly with the optimal thresholds in Table 3.4.

Table 3.5: Thresholds calculated using Equation 3.13.

<table>
<thead>
<tr>
<th>SNR</th>
<th>( L )</th>
<th>( M )</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>-0.121</td>
<td>0.000</td>
<td>0.121</td>
</tr>
<tr>
<td>Scaled</td>
<td>-0.049</td>
<td>0.000</td>
<td>0.049</td>
</tr>
<tr>
<td>Offset</td>
<td>-0.151</td>
<td>-0.030</td>
<td>0.091</td>
</tr>
<tr>
<td>Both</td>
<td>-0.079</td>
<td>-0.030</td>
<td>0.019</td>
</tr>
</tbody>
</table>
To calculate the theoretical probability of misclassification using these thresholds, we use Equations 3.7 through 3.10 along with Equation 3.6 to compute $P(tx \neq X \mid rx = X)$ for each symbol $X$. These probabilities are given in Table 3.6.

Table 3.6: Probability of misclassification for each symbol under varying SNR.

(a) Symbol 00

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>$2.1 \cdot 10^{-7}$</td>
<td>$9.2 \cdot 10^{-3}$</td>
<td>$6.5 \cdot 10^{-2}$</td>
<td>0.14</td>
<td>0.19</td>
</tr>
<tr>
<td>Scaled</td>
<td>$2.1 \cdot 10^{-7}$</td>
<td>$9.2 \cdot 10^{-3}$</td>
<td>$6.5 \cdot 10^{-2}$</td>
<td>0.14</td>
<td>0.19</td>
</tr>
<tr>
<td>Offset</td>
<td>$3.5 \cdot 10^{-7}$</td>
<td>$1.1 \cdot 10^{-2}$</td>
<td>$7.0 \cdot 10^{-2}$</td>
<td>0.15</td>
<td>0.20</td>
</tr>
<tr>
<td>Both</td>
<td>$3.2 \cdot 10^{-6}$</td>
<td>$1.9 \cdot 10^{-2}$</td>
<td>$9.7 \cdot 10^{-2}$</td>
<td>0.19</td>
<td>0.24</td>
</tr>
</tbody>
</table>

(b) Symbol 01

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>$1.1 \cdot 10^{-7}$</td>
<td>$4.7 \cdot 10^{-3}$</td>
<td>$3.3 \cdot 10^{-2}$</td>
<td>$7.2 \cdot 10^{-2}$</td>
<td>$9.6 \cdot 10^{-2}$</td>
</tr>
<tr>
<td>Scaled</td>
<td>$1.2 \cdot 10^{-7}$</td>
<td>$4.7 \cdot 10^{-3}$</td>
<td>$3.3 \cdot 10^{-2}$</td>
<td>$7.4 \cdot 10^{-2}$</td>
<td>$9.7 \cdot 10^{-2}$</td>
</tr>
<tr>
<td>Offset</td>
<td>$2.0 \cdot 10^{-7}$</td>
<td>$5.5 \cdot 10^{-3}$</td>
<td>$3.5 \cdot 10^{-2}$</td>
<td>$7.5 \cdot 10^{-2}$</td>
<td>0.10</td>
</tr>
<tr>
<td>Both</td>
<td>$1.8 \cdot 10^{-6}$</td>
<td>$1.0 \cdot 10^{-2}$</td>
<td>$4.9 \cdot 10^{-2}$</td>
<td>$9.2 \cdot 10^{-2}$</td>
<td>0.11</td>
</tr>
</tbody>
</table>

(c) Symbol 10

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>$2.2 \cdot 10^{-7}$</td>
<td>$9.2 \cdot 10^{-3}$</td>
<td>$6.6 \cdot 10^{-2}$</td>
<td>0.14</td>
<td>0.19</td>
</tr>
<tr>
<td>Scaled</td>
<td>$2.2 \cdot 10^{-7}$</td>
<td>$9.1 \cdot 10^{-3}$</td>
<td>$6.5 \cdot 10^{-2}$</td>
<td>0.14</td>
<td>0.19</td>
</tr>
<tr>
<td>Offset</td>
<td>$3.5 \cdot 10^{-7}$</td>
<td>$1.1 \cdot 10^{-2}$</td>
<td>$7.0 \cdot 10^{-2}$</td>
<td>0.15</td>
<td>0.20</td>
</tr>
<tr>
<td>Both</td>
<td>$3.1 \cdot 10^{-6}$</td>
<td>$1.9 \cdot 10^{-2}$</td>
<td>$9.8 \cdot 10^{-2}$</td>
<td>0.19</td>
<td>0.24</td>
</tr>
</tbody>
</table>

(d) Symbol 11

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>$1.4 \cdot 10^{-7}$</td>
<td>$4.9 \cdot 10^{-3}$</td>
<td>$3.3 \cdot 10^{-2}$</td>
<td>$7.2 \cdot 10^{-2}$</td>
<td>$9.7 \cdot 10^{-2}$</td>
</tr>
<tr>
<td>Scaled</td>
<td>$1.5 \cdot 10^{-7}$</td>
<td>$4.8 \cdot 10^{-3}$</td>
<td>$3.3 \cdot 10^{-2}$</td>
<td>$7.1 \cdot 10^{-2}$</td>
<td>$9.6 \cdot 10^{-2}$</td>
</tr>
<tr>
<td>Offset</td>
<td>$2.2 \cdot 10^{-7}$</td>
<td>$5.6 \cdot 10^{-3}$</td>
<td>$3.7 \cdot 10^{-2}$</td>
<td>$7.7 \cdot 10^{-2}$</td>
<td>0.10</td>
</tr>
<tr>
<td>Both</td>
<td>$1.9 \cdot 10^{-6}$</td>
<td>$1.0 \cdot 10^{-2}$</td>
<td>$5.1 \cdot 10^{-2}$</td>
<td>0.10</td>
<td>0.13</td>
</tr>
</tbody>
</table>
As described in Section 3.1.1, the thresholds in Table 3.5 were then implemented in software and ran on a new set of test symbols. For each symbol the percentage of misclassification on this test data is shown in Table 3.7.

Table 3.7: Percentage of misclassification for each symbol on test data.

(a) Symbol 00

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.99</td>
<td>6.70</td>
<td>15.0</td>
<td>19.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.84</td>
<td>6.20</td>
<td>14.0</td>
<td>19.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>1.10</td>
<td>7.00</td>
<td>15.0</td>
<td>20.0</td>
</tr>
<tr>
<td>Both</td>
<td>$3.0 \cdot 10^{-4}$</td>
<td>1.80</td>
<td>9.30</td>
<td>18.0</td>
<td>23.0</td>
</tr>
</tbody>
</table>

(b) Symbol 01

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.41</td>
<td>3.10</td>
<td>6.80</td>
<td>9.30</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.57</td>
<td>3.60</td>
<td>7.70</td>
<td>10.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.55</td>
<td>3.50</td>
<td>7.60</td>
<td>10.0</td>
</tr>
<tr>
<td>Both</td>
<td>$2.0 \cdot 10^{-4}$</td>
<td>1.20</td>
<td>5.40</td>
<td>10.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>

(c) Symbol 10

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.96</td>
<td>6.70</td>
<td>15.0</td>
<td>20.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>$9.9 \cdot 10^{-5}$</td>
<td>0.83</td>
<td>6.20</td>
<td>14.0</td>
<td>19.0</td>
</tr>
<tr>
<td>Offset</td>
<td>$9.9 \cdot 10^{-5}$</td>
<td>1.10</td>
<td>7.00</td>
<td>15.0</td>
<td>20.0</td>
</tr>
<tr>
<td>Both</td>
<td>$2.0 \cdot 10^{-4}$</td>
<td>1.80</td>
<td>9.40</td>
<td>18.0</td>
<td>23.0</td>
</tr>
</tbody>
</table>

(d) Symbol 11

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.42</td>
<td>3.10</td>
<td>6.90</td>
<td>9.30</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.56</td>
<td>3.60</td>
<td>7.80</td>
<td>10.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.55</td>
<td>3.60</td>
<td>7.60</td>
<td>10.0</td>
</tr>
<tr>
<td>Both</td>
<td>$7.9 \cdot 10^{-4}$</td>
<td>1.20</td>
<td>5.40</td>
<td>10.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>
3.1.3 Discussion

Comparing Tables 3.6 and 3.7 shows that the actual performance of the designed decoder on unknown data matches closely with the predicted performance – with less than 1% difference in all cases. In the worst case of a 0.001:1 SNR signal with scaling and DC offset distortion, the decoder made an error 13% of the time on the “outer” 11/01 symbols and 23% of the time on the “inner” 00/10 symbols. How these errors were made is illustrated by Table 3.8. All of the transmitted symbols show the same pattern of being misclassified as one of the closest adjacent symbols, as would be expected, but the inner symbols were also misclassified as the “opposite” outer symbol (albeit less than 0.02% of the time.) The worse performance by the decoder on the inner symbols is likely due to these symbols being surrounded on both sides by other symbols. The two thresholds required on either side of these inner symbols create two sources of error, as neighboring PDFs overlap the thresholds on both sides. The outer symbols, however, have only one source of error with their one threshold and otherwise have freedom out to infinity.

Table 3.8: Confusion matrix for evaluation on Both waveform at 0.001:1 SNR.

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>768121</td>
<td>128710</td>
<td>120580</td>
<td>218</td>
</tr>
<tr>
<td>01</td>
<td>114833</td>
<td>881085</td>
<td>190</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>120753</td>
<td>199</td>
<td>769758</td>
<td>128248</td>
</tr>
<tr>
<td>11</td>
<td>187</td>
<td>0</td>
<td>115140</td>
<td>883978</td>
</tr>
</tbody>
</table>

Although the performance of this simple decoder is suitable for this research, it could be improved and evaluated further. One of the two assumptions made when designing this decoder was that each symbol has an equal chance of being transmitted,
so the \textit{a priori} probabilities are equal. This may not be true on real-world P25 transmissions, however, and better \textit{a priori} knowledge could lead to more optimal thresholds, weighted for one symbol or another. The other assumption, of perfect symbol clock timing, is more like a strict requirement for this decoder, as being even a few samples off from the true sampling instant can lead to widely varying sample values, likely resulting in very poor performance when decoded by threshold. It’s unclear that this can be solved by the decoder, though, so the external module that determines the symbol clock should have suitable performance at satisfying this requirement (this problem is studied in Section 3.2.) To further characterize the performance of the decoder on real-world signals, it could be evaluated on additional waveforms with more noise and distortion applied, including Rayleigh fading and shot and colored noise. As this decoder is open source, though, anyone with the motivation and knowledge could implement these improvements and feed them back to the community.

3.2 Frame Synchronization

This set of experiments investigates recognition of the P25 frame synchronization sequence and use of the samples within the sequence to estimate $\mu_{11}$ and $\mu_{01}$, as needed by the symbol decoder. Each packet in P25 is preceded by a frame synchronization sequence, which is generated by modulating the sequence of 24 symbols

$$01 01 01 01 01 01 01 01 01 11 11 11 01 11 11 11 11 01 11 11 11 11 11 11$$

into a 231-sample waveform as described previously [16]. Doing so creates the waveform in Figure 3.7. As this sequence signals the start of a new packet, the performance in recognizing it contributes to overall decoding performance: The synchronizer that recognizes it should minimize false positives and false negatives,
as both can cause lost packets, and it should minimize symbol clock error, which occurs when the classification instant doesn’t align exactly with the impulse instant of the last symbol in the sequence (sample 230 in Figure 3.7), as such errors can degrade later symbol decoding performance.

Figure 3.7: P25 frame synchronization waveform, with impulse instants as dots.

A classical method to implement such a synchronizer is with cross-correlation, where a “fingerprint” $F[n]$ of the synchronization waveform is continually correlated with the input signal:

$$S[n] = \frac{1}{231} \sum_{i=0}^{230} F[n - i] \cdot x[n - i] = \frac{1}{231} \sum_{i=0}^{230} F[n - i] \cdot h[i]$$  \hspace{1cm} (3.14)

This is done until the returned correlation power rises above a certain threshold $T$:

$$S[n] \geq T$$  \hspace{1cm} (3.15)
When the power peaks above this threshold, the instant of the peak should be considered the classification instant, when ideally the fingerprint is exactly aligned with itself in the input signal [96]. The problem is then to find the threshold that minimizes false positives and false negatives.

The thresholding method investigated here computes the threshold as a function of the input signal power. Signal power is defined as

$$P = \frac{1}{N} \sum_{n=1}^{N} x[n]^2$$ \hspace{1cm} (3.16)

over $N$ samples $x[n]$ [96], and it must be calculated at each sample for this method. An approximation to this value can be calculated more efficiently for large $N$ using an Exponentially-Weighted Moving Average (EWMA),

$$\tilde{P}[n] = \alpha \cdot x[n]^2 + (1 - \alpha) \cdot \tilde{P}[n-1] \quad \text{where} \quad \tilde{P}[1] = x[1]^2$$ \hspace{1cm} (3.17)

for some smoothing parameter $\alpha$ [96]. Then for the Nominal waveform with samples $x_N[n]$,

$$\tilde{P}_N[n] = \alpha \cdot x_N[n]^2 + (1 - \alpha) \cdot \tilde{P}_N[n-1]$$ \hspace{1cm} (3.18)

$$S_N[n] = \frac{1}{231} \sum_{i=0}^{230} F[n-i] \cdot x_N[n-i]$$ \hspace{1cm} (3.19)

Since scaling and offset distortion are assumed to be simple multiplication or addition with $x_N[n]$,

$$x_S[n] = m \cdot x_N[n]$$ \hspace{1cm} (3.20)

$$x_O[n] = k + x_N[n]$$ \hspace{1cm} (3.21)

for some $m$ and $k$. Using Equation 3.20 with Equations 3.18 and 3.19 gives

$$\tilde{P}_S[n] = m^2 \tilde{P}_N[n]$$ \hspace{1cm} (3.22)

$$S_S[n] = m S_N[n]$$ \hspace{1cm} (3.23)
while using Equation 3.21 with the same equations gives

\[ \tilde{P}_O[n] = \tilde{P}_N[n] + 2\alpha k x[n] + \alpha k^2 \]

\[ S_O[n] = S_N[n] + k \sum_{i=0}^{230} F[n - i] \]

Assuming \( k \) is small relative to the signal amplitude, the \( \tilde{P}_N[n] \) and \( S_N[n] \) terms will dominate their respective sums, so

\[ \tilde{P}_O[n] \approx \tilde{P}_N[n] \quad (3.24) \]

\[ S_O[n] \approx S_N[n] \quad (3.25) \]

These relations suggest that if we can empirically determine a suitable threshold \( T \) to compare \( S_N[n] \) against, then we can find a suitable threshold for an arbitrary signal using \( T \) scaled by that signal’s power relative to \( \tilde{P}_N \). We first need to satisfy

\[ S_N[n_{inner}] \geq T \approx t(\tilde{P}_N[n_{inner}]) \quad (3.26) \]

where \( S_N[n_{inner}] \) is a correlation power within a frame synchronization sequence, \( T \) is a constant found empirically and \( t(p) \) is the threshold function. Then using the above relations,

\[ S_S[n_{inner}] \geq mT \approx t(\tilde{P}_S[n_{inner}]) \quad (3.27) \]

\[ S_O[n_{inner}] \geq T \approx t(\tilde{P}_O[n_{inner}]) \quad (3.28) \]

Therefore, \( t(p) \) should satisfy

\[ t(\tilde{P}_N[n]) \approx t(\tilde{P}_O[n]) \approx T \quad (3.29) \]

\[ t(\tilde{P}_S[n]) \approx mT \]

A function that does so is

\[ t(p) = \sqrt{p} \left( \frac{T}{\sqrt{\tilde{P}_N}} \right) = \sqrt{p} \cdot q \quad (3.30) \]
where \( \bar{P}_N \approx \bar{P}_N[n] \) is the mean EWMA power over all \( x_N[n] \), so \( \bar{P}_S[n] \approx m^2 \bar{P}_N \) and \( \bar{P}_O[n] \approx \bar{P}_N \). Then \( S[n] \geq t(\bar{P}[n]) \) is evaluated at each sample.

Once the synchronization sequence is recognized, the correlation history \( h[0], h[1], \ldots, h[230] \) provides a set of samples for the sequence of 11/01 symbols in the fingerprint. These samples can then be analyzed to estimate \( \mu_{11} \) and \( \mu_{01} \) for the decoder to use in the upcoming packet body. Let

\[
i[n] = h[n_0 + 10n]
\]

be the value at the \( n \)th impulse instant within the correlation history, where \( n_0 \) is the index of the first impulse instant to use. To find \( n_0 \) for this method, notice that when \( S[n] \) peaks on a synchronization sequence, the history values will be approximately those in Figure 3.7 with the same indices. When that peak is observed at \( S[n + 1] \), however, the history array has been shifted to \( h'[0] = h[1], h'[1] = h[2], \ldots, h'[230] = x[n + 1] \). Therefore, the impulse instant of the first symbol has been shifted out, and the impulse instant of the second symbol now occurs at \( h'[9] \). Therefore, \( n_0 = 9 \) and 23 impulse instants can be analyzed, 10 for symbol 01 and 13 for symbol 11. Using Equation 3.31, let

\[
I_{01} = \{i[n] \mid n \in \{0, 1, 2, 3, 5, 6, 9, 10, 15, 17\}\}
\]

\[
I_{11} = \{i[n] \mid n \in \{4, 7, 8, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22\}\}
\]

\[
I_A = \{|i[n]| \mid 0 \leq n \leq 22\}
\]

We can then find the average of these sets with

\[
I_{01} = \frac{1}{|I_{01}|} \sum_{i[n] \in I_{01}} i[n], \quad I_{11} = \frac{1}{|I_{11}|} \sum_{i[n] \in I_{11}} i[n], \quad I_A = \frac{1}{|I_A|} \sum_{i[n] \in I_A} i[n]
\]

Using these, we can estimate that \( \mu_{01} = I_{01} \) and \( \mu_{11} = I_{11} \) for the upcoming symbols and calculate \( \{L, M, U\} \) as detailed in Section 3.1. Alternatively, we could estimate
that $\mu_{01} = \bar{I}_A$ and $\mu_{11} = -\bar{I}_A$, but this is shown to perform relatively poorly in the presence of DC offset distortion.

### 3.2.1 Procedure

The baseband files generated in the first part of Section 3.1.1, with 4 waveforms at 5 SNRs, again form the “training” set of this experiment. As mentioned, 1000 frame synchronization sequences were embedded in each file, with 4000 random symbols between each. The number of these sequences is relatively low compared to the total number of symbols, but this matches real P25 signals, where relatively large packets are separated by the short synchronization sequences. Additionally, a comparison was done between the Nominal waveform with 1000 and 100,000 embedded sequences, and no significant difference to the following results was noticed, just an increase in processing time.

The first experiment on this training data evaluated the symbol clock error performance. To do this, the sequence of samples in each file was cross-correlated with the 231-sample fingerprint in Figure 3.7, and the power returned by the correlator was recorded for “inner-sync” samples – those within the 1000 embedded synchronization sequences – as well as 10 additional samples past the end of each sequence. The result was 1000 sets of 241 power values for each file. From this the $\mu$ and $\sigma$ of the power at each sample index was plotted to characterize the shape of the power curve across noise and distortion levels. The sample index of peak power was also extracted for each sequence and compared to the expected value (sample 230) to calculate the extent of symbol clock errors made by the correlator. Additionally, a 10-bin histogram was created for the peak power values across all the sequences, and the distribution was found to fit a Gaussian PDF with the same $\mu$ and $\sigma$. Finally, the relations in Equations 3.23 and 3.25 were verified across the waveforms.
The next experiment characterized the distribution of cross-correlation powers over “outer-sync” samples – those outside any of the embedded synchronization sequences. To do this, all such samples were sequentially cross-correlated with the fingerprint, and the returned power was recorded for each. A 20-bin histogram of these power values was created, and the distribution was found to fit a Gaussian PDF with the same $\mu$ and $\sigma$.

After that, the overall power of the Nominal unprocessed waveform was calculated, and an EWMA filter was designed to approximate this value. To start, $x[n]^2$ was calculated for each sample $x[n]$ and recorded, and the overall power was computed using Equation 3.16. Next, the EWMA filter in Equation 3.17 was sequentially computed over the $x[n]^2$ values and the returned power recorded at each sample; this was done for each $\alpha \in \{0.0001, 0.0002, 0.0005, 0.0007, 0.0010\}$. Using this, the $\mu \pm \sigma$ error relative to $P$ and the “settling time” – the initial number of samples needed to reach one $\sigma$ – was compared across $\alpha$. Additionally, the relations in Equations 3.22 and 3.24 were verified.

With the distributions of EWMA power, inner-sync correlation power, and outer-sync correlation power characterized, the next experiment used the Nominal unprocessed waveform to determine a value for $T$ in Equation 3.30 that would minimize the probability of false positives with the outer-sync PDF and minimize the probability of false negatives with the inner-sync PDF. The resulting $t(p)$ was then applied to the other waveforms to calculate the probability of false positives and negatives in each case.

Next, a method was designed to estimate $\mu_{11}$ and $\mu_{01}$ using only the samples within each synchronization sequence. To start, the 231 sample values in each of the embedded sequences was extracted and used to calculate $I_{01}, I_{11},$ and $\pm I_A$. The
percent error of each compared to the respective $\mu$ in Table 3.3 was then computed, and the $\mu$ and $\sigma$ of these errors was used to evaluate each estimation’s performance.

The final experiment evaluated a synchronizer implemented according to the results found here, using the evaluation set of baseband files generated in the second part of Section 3.1.1. The synchronizer uses a cross-correlator with the 231-sample fingerprint shown in Figure 3.7 and an EWMA filter with $\alpha = 0.0005$. Both are evaluated at each input sample $x[n]$ to compute $S[n]$ and $\tilde{P}[n]$ according to Equations 3.14 and 3.17. If at any point $S[n] \geq t(\tilde{P}[n])$, the synchronizer enters the ramp state. This check is disabled for the first 6000 samples to allow the correlator and EWMA filter to settle. Once the synchronizer enters the ramp state, it monitors the cross-correlation power until $S[n] < S[n-1]$. When this happens, the synchronizer considers $x[n-1]$ as the final impulse instant in the synchronization sequence and $x[n]$ as part of the first outer-sync symbol. Using the 231-sample history of the correlator, the synchronizer uses the impulse instants at indices $n = 9, 19, \ldots, 229$ to calculate $I_{01}$ and $I_{11}$. These values are then passed to the previously-implemented symbol decoder to calculate $\{L, M, U\}$ and decode the symbols that follow. This implementation was first ran on each file to evaluate the number of false positives, false negatives, and symbol clock errors. Then, the symbol decoder was activated at each of the 1000 correct symbol clocks with the estimated $\mu_{01}$ and $\mu_{11}$ and used to decode symbols up to the next synchronization sequence. These symbol decisions were then compared with the true values to evaluate decoding accuracy.

### 3.2.2 Results

The percentage of incorrect symbol clock timings over all 1000 sequences is shown in Table 3.9. In all cases, these errors are due to peak power falling on either side of the correct sample (230), as shown for the worst case in Figure 3.8.
Table 3.9: Percent of incorrect frame synchronization timing.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.30</td>
<td>2.00</td>
<td>6.10</td>
<td>8.40</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.10</td>
<td>1.60</td>
<td>5.70</td>
<td>9.70</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.10</td>
<td>2.10</td>
<td>8.10</td>
<td>10.40</td>
</tr>
<tr>
<td>Both</td>
<td>0.00</td>
<td>0.20</td>
<td>2.90</td>
<td>9.30</td>
<td>14.40</td>
</tr>
</tbody>
</table>

Figure 3.8: All symbol clock errors were due to peak power falling on the sample exactly before or after the correct one, shown for Both waveform at 0.001:1 SNR.

The power curve returned by the cross-correlator as it’s sliding across each synchronization sequence generally has the same shape across distortion levels, as shown in Figure 3.9, and is just expanded or compressed due to the relative amplitude of the input signal. There’s little difference, however, due to DC offset distortion. The
variation in the curve generally shrinks from left to right, reaching a minimum at the final ramp up and peak. This section’s variation becomes overall more significant with increased noise, as Figure 3.10 illustrates.

Figure 3.9: Comparison of correlation powers during and after synchronization sequence, with various distortion at 0.001:1 SNR.

The effects of varying the EWMA smoothing parameter $\alpha$ can be seen in Figure 3.11. All curves have a $\mu$ very close to the true power calculated with Equation 3.16, and in general decreasing $\alpha$ leads to a smoother curve. Doing so also leads to longer
settling and response times, though, and this can actually hurt the accuracy, as seen by the slightly wider $\sigma$ of the $\alpha = 0.0001$ case. The $\alpha = 0.0005$ EWMA gives a good tradeoff between accuracy and settling time, with 0.55% higher error compared to $\alpha = 0.0002$ but less than half the settling time (below 1/8th of a second.)
Figure 3.11: EWMA accuracy and settling time for various $\alpha$ on Nominal waveform.
The distribution of correlation powers for outer-sync and peak inner-sync samples, along with the average power returned by the EWMA, is shown in Figure 3.12 for the Nominal waveform. As noise increases, the inner-sync distribution spreads while the outer-sync distribution changes very little. Additionally, the increased noise power slightly increases the average power returned by the EWMA.

Figure 3.12: Outer-sync correlation power stays constant while inner-sync peak power spreads in the presence of noise, shown for Nominal waveform.

For the Nominal unprocessed waveform with parameters shown in Figure 3.13,

\[
\begin{align*}
\mu_O &= -5.217920935230727 \cdot 10^{-7} \\
\sigma_O &= 0.004889642330127736 \\
\tilde{P}_N &= 0.01732663212798269 \\
\mu_I &= 0.03176918776883118
\end{align*}
\]  

(3.34)

Then the threshold \( T \) is taken as midway between the mean of the outer-sync distribution and the mean of the peak inner-sync distribution:
\[ T = \frac{\mu_O + \mu_I}{2} = 0.015884332988368825 \]  

(3.35)

Scaling this by the average EWMA power,

\[ q = \frac{T}{\sqrt{\bar{P}_N}} = 0.1206734989540087 \]  

(3.36)

So the threshold function becomes

\[ t(p) = 0.1206734989540087\sqrt{p} \]  

(3.37)

Figure 3.13: Distribution values for Nominal unprocessed waveform.

Applying this threshold to the other waveforms with \( p \) taken as the average EWMA power for each resulted in thresholds sampled in Figure 3.14. Evaluating
this threshold function gave the theoretical probability of false positives and false
negatives shown in Tables 3.10 and 3.11.

Comparing the $\mu_{01}$ and $\mu_{11}$ calculated at each synchronization sequence with the
respective $\mu$ in Table 3.3 on average gives the percent errors shown in Table 3.12.

The synchronizer implemented using these results had the percentage false
positive and false negative errors shown in Tables 3.13 and 3.14 when ran on the
Table 3.10: Average probability of false positive errors.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.0 \cdot 10^{-5}</td>
</tr>
<tr>
<td>Scaled</td>
<td>1.2 \cdot 10^{-5}</td>
<td>1.2 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.1 \cdot 10^{-5}</td>
</tr>
<tr>
<td>Offset</td>
<td>1.1 \cdot 10^{-5}</td>
<td>1.0 \cdot 10^{-5}</td>
<td>1.0 \cdot 10^{-5}</td>
<td>1.0 \cdot 10^{-5}</td>
<td>9.6 \cdot 10^{-6}</td>
</tr>
<tr>
<td>Both</td>
<td>1.9 \cdot 10^{-6}</td>
<td>2.0 \cdot 10^{-6}</td>
<td>2.1 \cdot 10^{-6}</td>
<td>2.1 \cdot 10^{-6}</td>
<td>2.3 \cdot 10^{-6}</td>
</tr>
</tbody>
</table>

Table 3.11: Average probability of false negative errors.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>3.3 \cdot 10^{-146}</td>
<td>8.7 \cdot 10^{-36}</td>
<td>2.2 \cdot 10^{-20}</td>
<td>3.0 \cdot 10^{-12}</td>
<td>5.8 \cdot 10^{-10}</td>
</tr>
<tr>
<td>Scaled</td>
<td>1.3 \cdot 10^{-147}</td>
<td>1.5 \cdot 10^{-37}</td>
<td>5.5 \cdot 10^{-19}</td>
<td>5.0 \cdot 10^{-12}</td>
<td>2.5 \cdot 10^{-9}</td>
</tr>
<tr>
<td>Offset</td>
<td>1.2 \cdot 10^{-139}</td>
<td>2.3 \cdot 10^{-39}</td>
<td>2.0 \cdot 10^{-17}</td>
<td>2.2 \cdot 10^{-12}</td>
<td>2.2 \cdot 10^{-9}</td>
</tr>
<tr>
<td>Both</td>
<td>1.1 \cdot 10^{-78}</td>
<td>2.6 \cdot 10^{-19}</td>
<td>2.7 \cdot 10^{-11}</td>
<td>5.0 \cdot 10^{-7}</td>
<td>2.6 \cdot 10^{-6}</td>
</tr>
</tbody>
</table>

evaluation data. Additionally, it encountered the percentage symbol clock errors given in Table 3.15 over the 1000 synchronization sequences in each file. Again, these errors were all due to peak power falling on the sample exactly before or after the correct one. Finally, the symbol decoder made the percentage misclassifications in Table 3.16.

3.2.3 Discussion

Overall, the mathematical motivation for the choice of $t(p)$ seems to hold true in practice, and symbol decoding performance using locally estimated $\mu_{01}$ and $\mu_{11}$ remains relatively close to that using a globally measured mean. Comparing Tables 3.16 and 3.7 shows that decoding performance using these estimates is slightly worse than the “ideal” decoding performance, likely due to the relatively small sample size in each sequence for determining the averages, which makes the computations more
Table 3.12: Percent error from means in Table 3.3 for various SNR, as $\mu\% \pm \sigma\%$.

(a) 12:1

<table>
<thead>
<tr>
<th>Waveform</th>
<th>$\bar{I}_{01}$</th>
<th>$\bar{I}_{11}$</th>
<th>$\bar{I}_A$</th>
<th>$-\bar{I}_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>1.80 ± 1.30</td>
<td>1.50 ± 1.10</td>
<td>1.20 ± 0.86</td>
<td>1.20 ± 0.86</td>
</tr>
<tr>
<td>Scaled</td>
<td>1.90 ± 1.40</td>
<td>1.50 ± 1.20</td>
<td>1.20 ± 0.90</td>
<td>1.20 ± 0.90</td>
</tr>
<tr>
<td>Offset</td>
<td>2.10 ± 1.60</td>
<td>1.30 ± 1.00</td>
<td>23.0 ± 1.60</td>
<td>13.0 ± 1.20</td>
</tr>
<tr>
<td>Both</td>
<td>3.30 ± 2.50</td>
<td>1.20 ± 0.99</td>
<td>82.0 ± 2.80</td>
<td>25.0 ± 1.10</td>
</tr>
</tbody>
</table>

(b) 3:1

<table>
<thead>
<tr>
<th>Waveform</th>
<th>$\bar{I}_{01}$</th>
<th>$\bar{I}_{11}$</th>
<th>$\bar{I}_A$</th>
<th>$-\bar{I}_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>4.80 ± 3.50</td>
<td>4.00 ± 3.00</td>
<td>3.00 ± 2.30</td>
<td>3.00 ± 2.30</td>
</tr>
<tr>
<td>Scaled</td>
<td>4.80 ± 3.50</td>
<td>4.20 ± 3.30</td>
<td>3.10 ± 2.40</td>
<td>3.10 ± 2.40</td>
</tr>
<tr>
<td>Offset</td>
<td>6.20 ± 4.70</td>
<td>3.50 ± 2.70</td>
<td>23.0 ± 4.80</td>
<td>12.0 ± 3.40</td>
</tr>
<tr>
<td>Both</td>
<td>9.10 ± 7.00</td>
<td>3.40 ± 2.60</td>
<td>82.0 ± 7.30</td>
<td>25.0 ± 3.00</td>
</tr>
</tbody>
</table>

(c) 0.001:1

<table>
<thead>
<tr>
<th>Waveform</th>
<th>$\bar{I}_{01}$</th>
<th>$\bar{I}_{11}$</th>
<th>$\bar{I}_A$</th>
<th>$-\bar{I}_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>6.80 ± 5.20</td>
<td>5.90 ± 4.50</td>
<td>4.30 ± 3.30</td>
<td>4.30 ± 3.30</td>
</tr>
<tr>
<td>Scaled</td>
<td>6.60 ± 5.00</td>
<td>6.00 ± 4.40</td>
<td>4.40 ± 3.40</td>
<td>4.40 ± 3.40</td>
</tr>
<tr>
<td>Offset</td>
<td>8.20 ± 6.10</td>
<td>5.10 ± 4.00</td>
<td>23.0 ± 6.40</td>
<td>12.0 ± 4.50</td>
</tr>
<tr>
<td>Both</td>
<td>13.0 ± 10.0</td>
<td>5.10 ± 4.00</td>
<td>82.0 ± 11.0</td>
<td>25.0 ± 4.40</td>
</tr>
</tbody>
</table>

susceptible to noise. One solution could be to use EWMAs for both $I_{01}$ and $I_{11}$ to reduce this susceptibility, but settling time might be a problem depending on the amount of smoothing required.

Comparing the percent errors in Table 3.12, though, it’s clear that in the presence of arbitrary distortion, separate $I_{01}$ and $I_{11}$ averages perform better overall than a single $I_A$ average. Although the single average performs better on the Nominal and
Table 3.13: Percentage of false positive errors.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>$6.5 \cdot 10^{-5}$</td>
<td>$6.5 \cdot 10^{-5}$</td>
<td>$8.4 \cdot 10^{-5}$</td>
<td>$6.9 \cdot 10^{-5}$</td>
<td>$6.7 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>Scaled</td>
<td>$5.7 \cdot 10^{-5}$</td>
<td>$7.9 \cdot 10^{-5}$</td>
<td>$6.0 \cdot 10^{-5}$</td>
<td>$5.7 \cdot 10^{-5}$</td>
<td>$6.2 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>Offset</td>
<td>$6.0 \cdot 10^{-5}$</td>
<td>$6.5 \cdot 10^{-5}$</td>
<td>$5.0 \cdot 10^{-5}$</td>
<td>$7.9 \cdot 10^{-5}$</td>
<td>$5.5 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>Both</td>
<td>$2.5 \cdot 10^{-6}$</td>
<td>$5.0 \cdot 10^{-6}$</td>
<td>$1.2 \cdot 10^{-5}$</td>
<td>$1.5 \cdot 10^{-5}$</td>
<td>$7.4 \cdot 10^{-6}$</td>
</tr>
</tbody>
</table>

Table 3.14: Percentage of false negative errors.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Both</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Scaled waveforms (likely due to the larger sample size), that advantage is heavily outweighed by the massive errors seen with the introduction of DC offset. This is in contrast to $I_{01}$ and $I_{11}$, which have relatively stable performance with and without DC offset.

Table 3.15: Percentage of incorrect symbol clock occurrences.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.00</td>
<td>1.70</td>
<td>6.00</td>
<td>9.50</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.20</td>
<td>2.40</td>
<td>7.50</td>
<td>10.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.30</td>
<td>1.80</td>
<td>7.50</td>
<td>10.0</td>
</tr>
<tr>
<td>Both</td>
<td>0.00</td>
<td>0.40</td>
<td>3.30</td>
<td>10.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>
Table 3.16: Percentage of misclassification for each symbol on evaluation data.

(a) Symbol 00

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>1.20</td>
<td>7.50</td>
<td>16.0</td>
<td>21.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>$4.0 \cdot 10^{-4}$</td>
<td>1.20</td>
<td>7.30</td>
<td>16.0</td>
<td>21.0</td>
</tr>
<tr>
<td>Offset</td>
<td>$2.0 \cdot 10^{-4}$</td>
<td>1.40</td>
<td>8.00</td>
<td>16.0</td>
<td>21.0</td>
</tr>
<tr>
<td>Both</td>
<td>$3.0 \cdot 10^{-4}$</td>
<td>2.40</td>
<td>11.0</td>
<td>20.0</td>
<td>26.0</td>
</tr>
</tbody>
</table>

(b) Symbol 01

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.58</td>
<td>3.80</td>
<td>7.80</td>
<td>10.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.63</td>
<td>3.80</td>
<td>7.80</td>
<td>10.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.67</td>
<td>4.20</td>
<td>8.30</td>
<td>11.0</td>
</tr>
<tr>
<td>Both</td>
<td>0.00</td>
<td>1.20</td>
<td>5.70</td>
<td>10.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>

(c) Symbol 10

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>1.10</td>
<td>7.40</td>
<td>16.0</td>
<td>20.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>$1.0 \cdot 10^{-4}$</td>
<td>1.10</td>
<td>7.30</td>
<td>15.0</td>
<td>21.0</td>
</tr>
<tr>
<td>Offset</td>
<td>$1.0 \cdot 10^{-4}$</td>
<td>1.30</td>
<td>7.90</td>
<td>16.0</td>
<td>21.0</td>
</tr>
<tr>
<td>Both</td>
<td>$7.0 \cdot 10^{-4}$</td>
<td>2.30</td>
<td>11.0</td>
<td>20.0</td>
<td>25.0</td>
</tr>
</tbody>
</table>

(d) Symbol 11

<table>
<thead>
<tr>
<th>Waveform</th>
<th>12:1</th>
<th>6:1</th>
<th>3:1</th>
<th>1:1</th>
<th>0.001:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>0.00</td>
<td>0.57</td>
<td>3.60</td>
<td>7.80</td>
<td>10.0</td>
</tr>
<tr>
<td>Scaled</td>
<td>0.00</td>
<td>0.52</td>
<td>3.60</td>
<td>7.70</td>
<td>10.0</td>
</tr>
<tr>
<td>Offset</td>
<td>0.00</td>
<td>0.64</td>
<td>3.80</td>
<td>7.90</td>
<td>11.0</td>
</tr>
<tr>
<td>Both</td>
<td>$6.9 \cdot 10^{-4}$</td>
<td>1.10</td>
<td>5.20</td>
<td>9.80</td>
<td>13.0</td>
</tr>
</tbody>
</table>
The symbol clock error measured on the evaluation data in Table 3.15 generally matches that measured on the training data in Table 3.9. It’s unclear how to reduce this error, since it occurs as part of the cross-correlation process. One method was investigated where, in calculating $I_{01}$ and $I_{11}$ after synchronization, $n_0$ was varied between 7 and 11, and the values with the lowest variance were chosen for each average. This property, that the average using the correct symbol clock has the lowest variance, holds in the higher SNRs, but is again unreliable in the presence of noise, and the added complexity was determined to outweigh any benefit in performance. In all cases of error, however, the clock is off by only one sample, so errors in later symbol decoding will be minimal relative to those cause by a larger error offset.

Moving on to the threshold function, the performance of the synchronizer on the evaluation data in Tables 3.13 and 3.14 generally exceeds the predicted performance in Tables 3.10 and 3.11. This is likely due to the fact that the probabilities are calculated using a global average of the EWMA power while the evaluations use the EWMA power calculated at each sample, which varies around the average as illustrated in Figure 3.11. In this case that variation apparently led to a $t(p)$ better scaled to the local sample power. The performance is good enough for 0% false negative errors across the 1000 synchronization sequences in all levels of noise and distortion, and the theoretical probabilities in Table 3.11 show that, in the worst case, at least an order of magnitude more sequences would be need to be trialled to see a nonzero percentage. Still, many of the cases would likely remain at 0% error without several orders of magnitude more trialled sequences.

Although false positive and false negative errors are both shown to be rare events, false positives have a slightly higher chance of occurring. This is desired, as real-world synchronization sequences generally follow immediately after the end of the previous
packet [16], so the synchronizer is designed to err on the side of greediness in that situation to reduce the chances of mid-transmission packet loss. The tradeoff is then that one or more packets at the start of a transmission may be lost due to the greediness (as a false positive may cause a real synchronization sequence to be lost if it occurs during the NID decoding step that takes place after synchronization.)

### 3.3 Comparison to Commercial Receiver

With the previous two sections evaluating the expected performance of individual components of the receiver, this section evaluates the real-world performance of all the individual components combined, through comparison to a heavily-engineered, commercial P25 receiver. Although a more meaningful comparison might be to a consumer devices with capabilities closer to the designed receiver, there was little justification in spending several hundred dollars on one of these devices for a one-off experiment, and no sources were known to borrow one. Given this situation, the next available option was to borrow time with a commercial receiver owned by the university police department. Because this device gives no detailed error statistics, the only available metric for comparison is the end result of the receiving and decoding pipelines: analog voice frames. In particular, the IMBE vocoder standard specifies that individual 20ms frames should be muted when error rates rise above a given threshold [44]. Therefore, the number of these muted frames gives insight into the overall performance of frame synchronization, symbol decoding, and error correction routines. Although muted frames are specified to be output as 20ms of “comfort noise“, the software was modified for this experiment to instead produce no output. With this change, mutings are relatively easy to detect (as phase discontinuities), and voice messages containing muted frames appear shortened. This shortening is what will be used to measure the number of frame muting errors.
3.3.1 Procedure

This experiment was facilitated by the Ohio University Police Department (OUPD), who provided access to one of the P25 handhelds used by their officers. The experiment was conducted on Saturday, April 16, 2016, the second day of “Numberfest” (which takes place annually in Athens, Ohio.) This event typically generates a large amount of law enforcement and public safety traffic [116], so this day was chosen in order to create a more suitable sample size. The location of the experiment was a conference room in OUPD headquarters, in the Scott Quad building on Ohio University’s Athens campus. The commercial device supplied by OUPD was a Motorola XTS 2500 [117] – a $3800 handheld transceiver [118] – set to receive-only mode. A handset accessory was attached, which provided a jack for audio output. Alongside this was the designed receiver, powered by a USB battery pack. Both radios were tuned to the local WOUB P25 site (approximately 2.78 miles west), and the Motorola was fixed on the Athens County Sheriff talkgroup. As Figure 3.15 shows, the audio output of each receiver was captured with an audio cable, and these cables were input to soundcard microphone ports on a laptop. With this setup, radio traffic was simultaneously recorded for 2 hours, from 4:00PM to 6:00PM. During this time, the Motorola typically had “full” signal strength, and the designed receiver typically had an S3/S4 reading on the S-meter. The audio was recorded in mono, as 16-bit signed PCM, with a 44.1kHz sample rate.

This experiment generated a control audio file from the Motorola’s output and a test audio file from the designed receiver’s output. These two waveforms were manually analyzed using Audacity, an audio editing application. First, after roughly synchronizing the two, each individual voice message in the control file was identified by looking for the obvious time separation between messages (as shown in Figure
3.15: Designed receiver and Motorola recording radio traffic in OUPD HQ.

3.16) and listening to verify the difference in voices. For each message, the rough starting and stopping samples were identified to get a measurement of the message length in samples and 20ms frames. Then, the corresponding voice message in the test file was analyzed in the same way to get a measurement of the number of muted frames. An example of a shortened message due to frame mutings is given in Figure 3.17. Because only the Motorola was fixed on the Sheriff talkgroup, occasionally the designed receiver would receive a message from a different talkgroup that overlapped
in time with a message from the Sheriff talkgroup. In those cases, the messages simply weren’t counted towards the results.

Figure 3.16: The Motorola (top) and designed receiver (bottom) waveforms, showing the obvious time separation between voice messages.

Figure 3.17: Example voice message received on the Motorola (top), and the same message with muted frames (approximately 137.)
3.3.2 Results

As given in Table 3.17, there were 418 voice messages transmitted during the experiment session. Over these, the designed receiver muted approximately 4.63% of the voice frames due to errors. These mutings occurred throughout the experiment session, as shown in Figure 3.18. In contrast, no mutings were observed in the messages received by the Motorola, making it an apparently perfect control for reference.

Table 3.17: Results of comparison experiment.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total voice messages</td>
<td>418</td>
</tr>
<tr>
<td>Total voice frames</td>
<td>77,271</td>
</tr>
<tr>
<td>Muted voice frames</td>
<td>3,581</td>
</tr>
</tbody>
</table>

3.3.3 Discussion

The designed receiver performed well compared to the commercial receiver (especially considering the $\times38$ price difference), successfully recovering over 95% of voice frames. As Figure 3.19 illustrates, 81% of voice messages had zero muted frames, and 97% had less than 40% of frames muted. The 100% frame mutings were all due to the receiver “missing” short voice messages (less than 2 seconds long), as shown in Figure 3.20.

Because no detailed error profiling was recorded during the experiment, the cause of the frame mutings can’t be pinpointed exactly. One contributing factor could be the relatively low (less than half) signal strength reported by the receiver during the experiment, which would lead to a noisier demodulated signal. The cause of this may be indoor interference overloading the RTL-SDR – due to the lack of frontend filtering and the small dynamic range – which would result in a higher noise floor and less
sensitivity to the desired signal. Additionally, the frame synchronization and symbol decoding methods used for this receiver likely have much worse performance in noise compared to the proprietary methods implemented by the Motorola engineers. Finally, there may be issues in the implementation of the error correcting code algorithms, although no anomalies have been found by the unit tests (with exhaustive testing used for the Hamming and Golay implementations.)

The cause of the receiver completely missing some of the short messages can’t be exactly pinpointed either. One cause may be the receiver occasionally failing to decode any of the relatively few trunking packets that precede these messages on the control channel, due to bursts of noise or other issues in the decoding pipeline. If this was the case, then longer messages would also be affected, although only frames at

Figure 3.18: Frame muting errors occurred throughout the 2 hour experiment.
Figure 3.19: Across all voice messages, 81% had zero muting errors, and 97% had less than 40% muted frames.

The beginning of these messages would be lost. The cause may also be due to issues in the implementation of the trunking protocol (such as missing support for new logic added in later versions of the standard [42] or as part of a separate standard), but it’s unclear how or why a special case would be added for short messages. Fortunately, 94% of messages less than 2 seconds long were at least partially received, and 88% were received with zero muted frames.

The sample size for this experiment was large enough for interesting results without being prohibitive to manual analysis. In particular, the level of radio traffic was relatively high due to the ongoing Numberfest, with around 3.5 voice messages per minute on average (from experience, this number would typically be less than
Figure 3.20: Muting errors were generally spread across message lengths, but 100% mutings only occurred with messages less than 2 seconds long.

1). Additionally, Figure 3.21 shows the traffic included a mix of message lengths, which exercised the receiver’s ability to both react quickly and decode voice frames for extended periods in real-time.
Figure 3.21: The radio traffic included a mix of message lengths.
4 CONCLUSIONS

4.1 Review

Public safety communications provide a key source of real-time information on local happenings and have been monitored for decades by the public. As these communications are increasingly carried over digital P25 radio, though, a specialized P25 receiver is becoming the requirement in more and more areas, and this will likely be the case for years to come. With little choice in the current P25 receiver market, the receiver designed in this research provides an alternative with a more accessible price. This is made possible with SDR, which is the foundation for implementing radio applications in software. In particular, the RTL-SDR supplies a practical implementation of these concepts at a very affordable price. Although several software packages exist as candidates to be the core of this SDR receiver, none are compatible with its requirements. Accordingly, new software libraries were built up using good software engineering practices, classical communications engineering algorithms, and modern digital radio signal processing approaches. These libraries were composed into a real-time, trunking-aware P25 receiver targeting the Raspberry Pi 2, and this receiver was found to perform well compared to a commercial P25 receiver in an exercise of its protocol implementation and decoding pipeline.

Overall, this receiver satisfies the hypothesis of the thesis: The goal was to create a sub-$100 device that can receive P25 voice messages with portable usability and automatic operation, and the details for how this was done and verified have been presented throughout the preceding sections. The low-cost hardware core is provided by the Raspberry Pi 2 and RTL-SDR. Importantly, the RTL-SDR allows much complexity to be shifted to (free) software. Then, the ability of this receiver to suitably receive P25 voice messages was verified using a commercial P25 receiver for
reference. Finally, the combined footprint and power usage of the Raspberry Pi and RTL-SDR give the receiver battery-operated, portable capability, and the software trunking awareness allows the receiver to follow conversations across traffic channels without user interaction.

The end result of this research is an accessible P25 receiver for hobbyists, radio amateurs, and other potential users. Supporting this receiver is a standalone, cross-platform, and framework-agnostic library for receiving P25 packets – the first of its kind – that can provide the foundation for future receivers and experiments. This is joined by several other implemented utility modules, such as a Rust interface to the RTL-SDR system library. Beyond the software, this thesis provides a new addition to the young academic ecosystem surrounding the RTL-SDR.

4.2 Future Work

The receiver in its current state is suitable for typical use cases, but there are many improvements that could be made or at least investigated. One high priority task is to implement a utility program that maps trunking channel numbers to traffic channel frequencies for each P25 site. This mapping currently requires an initial manual investigation that involves monitoring when traffic channels become active and correlating that information to channel numbers simultaneously being transmitted in Group Voice Update packets on the control channel. This initial configuration step could easily be automated by a program that scans through the configured traffic channels and notes which are active during each transmitted channel number. This program could collect data as long as desired and, at the end, give a mapping with confidence probabilities. The user could then use this information to fill in the channel numbers in the configuration file.
Another usability improvement would come from more talkgroup intelligence in the trunking awareness implementation. Currently, the receiver switches to the first talkgroup traffic channel it receives when monitoring the control channel, but this can cause voice messages to be lost mid-conversation and gives no user priority to certain talkgroups over others. This could be enhanced by allowing runtime talkgroup filtering and priority assignment. Additionally, encrypted talkgroups could be filtered out, as there are no plans to support decryption (due to legal issues), and decoding these conversations results in unpleasant garbled audio. Finally, some form of timeout-based “hysteresis” would be useful to prevent rapid jumping between talkgroups mid-conversation, although, without parallel control and traffic channel decoding, this could also result in missed beginnings and whole parts of conversations.

Such parallel channel decoding would fully take advantage of the RTL-SDR by better utilizing its bandwidth capabilities (up to 2.4MHz at least) to capture and decode multiple P25 channels simultaneously. Using this, the receiver could decode and buffer conversations happening on multiple traffic channels and use a configurable algorithm to filter, prioritize, and queue the audio streams. Ideally, this would result in complete flexibility with zero lost conversations. Unfortunately, due to the computing constraints of the Raspberry Pi, it’s likely that only a fraction of the control channels could be decoded in parallel, so the full benefits wouldn’t be realized.

In fact, the initial receiver and decoding pipeline was designed around this idea of parallel decoding, but the initial signal processing implementation simply couldn’t process 2.4MS/s fast enough on the Raspberry Pi (although it could on a dual-core laptop.) Through profiling, it was found that an enormous amount of processing time was being spent on the FIR decimation anti-alias filters. Although a multi-stage decimation approach [52] was used to reduce the order of the FIR filters and overall
time complexity, the ARM processor couldn’t perform the floating point FIR dot-
product operations fast enough for 2.4MS/s, even with less than 50 FIR coefficients.
It’s likely that these FIR anti-alias filters could be replaced with Infinite Impulse
Response (IIR) filters or a Cascaded Integrator-Comb (CIC) filter approach [52]
– the latter of which uses zero multiply operations – and retain suitable filtering
performance. An IIR approach was investigated with a simple 2-biquad filter created
in Matlab, but for unknown reasons the actual frequency response had massive gain
compared to the designed response, even when the filter was applied using Matlab’s
filtering routine.

Other optimizations could be applied throughout the pipeline as well, such
as in calculating the complex argument for FM demodulation, which boils down
to an \texttt{atan2} function call to compute the arctangent. This relatively expensive
trigonometric computation could be replaced with an approximation based on the
Taylor series expansion of the function [119]. As this operation is applied at the
relatively low 48kHz sample rate, though, more profiling would be needed to weigh the
expected speedup against the corresponding approximation error. Finally, there are
two relatively obvious optimizations that could be investigated in the IMBE decoder.
Currently, the 256-bin Discrete Fourier Transform (DFT) and Inverse DFT (IDFT)
operations required by the unvoiced synthesis routine use a relatively naïve (but
optimized) implementation, and this may be sped up by calling out to one of the
heavily-optimized Fast Fourier Transform (FFT) libraries available [120] (although
the overhead of doing so would have to be weighed against the relatively small 256
FFT size.) Additionally, a thread pool could be utilized in the final synthesis routine
and persisted across frames to reduce the kernel and context-switching overhead
needed to spin up the threads for each frame decode operation.
Additional future work could investigate improvements in receiving sensitivity and decoding performance. On the hardware side, the practical gains of a tuned antenna for the 700MHz band could be measured. A non-resonant “rubber ducky” antenna tuned for the 70cm band was used with the receiver in this research, but a resonant and more engineered (especially collinear) antenna could increase the amount of captured signal as well as selectivity to reject interference (although local P25 sites already broadcast at very high power levels, so the higher gain antenna may instead increase noise pickup or cause frontend overload with the RTL-SDR.) In a related issue, the Raspberry Pi itself may be a source of local Electromagnetic Interference (EMI) due to the high-speed switching nature of its circuitry and its proximity to the antenna. A partial investigation found the radiated EMI to be low on the 700MHz band, but overloading noise on lower frequencies could also contribute to reduced receiving performance. One solution may be to shield the Raspberry Pi (with copper tape or otherwise) and measure the EMI and noise floor before and after. A current workaround is to simply move the RTL-SDR and antenna farther from the computing package, either with a USB extension jumper or coaxial antenna cable (this can also serve to get the antenna higher above ground.)

On the software side, decoding performance may be improved by utilizing an EWMA or other method to smooth the estimates for $\mu_{01}$ and $\mu_{11}$ to, as described in Section 3.2.3, reduce their susceptibility to noise. The amount of required smoothing to see this benefit, though, would have to be weighed against the method’s responsiveness to changes in overall signal amplitude. The decoding performance may further be improved by enhancing the spectral content of the baseband signal itself. Although the deemphasis filter already significantly reduces a portion of high-frequency noise, its response is only specified up to 2880Hz, which leaves 21120Hz of
unspecified response at a 48kHz sample rate. As such, applying a low pass filter with a cutoff around 3000Hz before applying the deemphasis filter may further attenuate unwanted high-frequency components.

Finally, miscellaneous improvements include completing the CAI protocol implementation by adding full support for data packets and packet generation. Access to data packets could be useful for experiments into their performance in noise and challenging propagation environments like hilly terrain. Further, data packets combined with packet generation and transmission could be utilized as an experimental alternative to the Automatic Packet Reporting System (APRS) and for amateur radio repeaters [121]. An IMBE encoder implementation would then create the opportunity for low-cost P25 “walkie-talkies” and other transceivers.
REFERENCES


<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>


OUPD Officer, Private Communication, Apr. 13, 2016.


A.1 Raised Cosine Filter

The raised cosine filter is used to transform an impulse train at 4800 baud to a C4FM baseband waveform used by P25 Phase I. At 48,000 samples per second, this means an impulse every 10 samples. Because this filter must satisfy the Nyquist criterion to minimize intersymbol interference, at 48kHz the impulse response should pass through zero every 10 samples before and after the center coefficient. The ideal frequency response of this filter is given by the standard [16] as

\[
H(f) = \begin{cases} 
1 & f < 1920 \\
0.5 + 0.5 \cos \frac{2\pi f}{1920} & 1920 \leq f < 2880 \\
0 & f \geq 2880 
\end{cases}
\] (A.1)

This response is approximated by a 121-coefficient FIR filter, compared in Figure A.1. Additionally, Figure A.2 shows this filter satisfies the Nyquist criterion with a 48kHz sample rate. The filter is identical to Ramsey’s design [43] except the coefficients have been scaled to give unity gain in the passband. It was designed in Matlab, with code identical to Ramsey’s except updated to use `rcosdesign` instead of the deprecated `firrcos`, as seen in Listing A.1.

```
1 % All frequency values are in Hz.
2 % Sampling Frequency
3 Fs = 48000;
4 % Order
5 N = 1002;
6 % Cutoff Frequency
7 Fc = 2400;
8 % Transition Mode
9 TM = 'Rolloff';
10 % Rolloff
11 R = 0.2;
```
Figure A.1: Ideal and approximated frequency response of raised cosine filter.
Figure A.2: The impulse response satisfies the Nyquist criterion at 48kHz.
% Design Type
DT = 'Normal';
% Create the window vector for the design algorithm.
win = barthannwin(N+1);

rcf = rcosdesign(0.2, 12, 10, 'normal');
rcf = rcf .* win((502-60):(502+60))';
rcf = rcf .* 0.1 / max(rcf);
Hd = dfilt.dffir(rcf);

A.2 Preemphasis Filter

The preemphasis (or “shaping”) filter functions as a typical FM preemphasis filter to reduce the effects of high frequency noise. The ideal frequency response is given by the standard [16] as

$$H(f) = \frac{\pi f/4800}{\sin(\pi f/4800)}$$  \hspace{1cm} (A.2)

and is only specified up to 2880Hz due to the raised cosine filter cutting off above that frequency. This response is approximated by a 39-coefficient FIR filter, compared in Figure A.3, which is identical to the filter designed by Ramsey [43]. The Matlab code is given in Listing A.2.

Listing A.2: Matlab code to generate preemphasis filter. [43]

```matlab
% FIR constrained equiripple Inverse Sinc Lowpass filter
% designed using the FIRCEQRIP function.
% All frequency values are in Hz.
% Sampling Frequency
Fs = 48000;
% Order
N = 38;
% Cutoff Frequency
Fc = 6800;
% Stopband Slope
slope = 0;
% Inverse Sinc Frequency Factor
isincffactor = 4;
% Inverse Sinc Power
isincpower = 1.7;
```
Figure A.3: Ideal and approximated frequency response of preemphasis filter.
\textbf{A.3 Deemphasis Filter}

The deemphasis (or “integrate and dump”) filter functions as a typical FM deemphasis filter to attenuate high frequency noise collected on the channel and reverse the effect of preemphasis on the baseband signal. The ideal frequency response is given by the standard \cite{16} as

$$H(f) = \frac{\sin(\pi f/4800)}{\pi f/4800} \quad (A.3)$$

and is again only specified up to 2880Hz. This ideal response is approximated by a 39-coefficient FIR filter, as compared in Figure A.4. The filter was designed in Matlab’s \texttt{fdatool}, with the generated code in Listing A.3.

Listing A.3: Matlab code to generate deemphasis filter.

```matlab
% Generated by MATLAB(R) 8.5 and the DSP System
% Toolbox 9.0.
% FIR constrained equiripple Inverse Sinc Lowpass filter
% designed using the FIRCEQRIP function.
% All frequency values are in Hz.
Fs = 48000;  \% Sampling Frequency
N = 38;      \% Order
Fc = 2900;   \% Cutoff Frequency
slope = 300; \% Stopband Slope
isincffactor = 1; \% Inverse Sinc Frequency Factor
isincpower = 3.5; \% Inverse Sinc Power
Dstop = 0.01; \% Stopband Attenuation
Dpass = 5.7501127785e-05; \% Passband Ripple

% Calculate the coefficients using the FIRCEQRIP function.
b = firceqrip(N, Fc/(Fs/2), [Dpass, Dstop], ...
    'slope', slope, 'invsinc', [isincffactor isincpower]);
Hd = dfilt.dffir(b);
```
Figure A.4: Ideal and approximated frequency response of deemphasis filter.
A.4 Anti-Alias Filter

The anti-alias filter attenuates signal components that would cause aliasing when downsampling from 240kHz to 48kHz. This is realized in a 41-coefficient FIR filter whose frequency response is given in Figure A.5. As the output of the decimator is immediately cascaded into a 5kHz channel select filter, the passband of this filter is also set at 5kHz. This allows a shallow rolloff compared to a higher cutoff frequency, which leads to good stopband attenuation with a reasonable filter order. The filter was designed in Matlab’s `fdatool`, with the generated code shown in Listing A.4.

Listing A.4: Matlab code to generate anti-alias filter.

```matlab
% Generated by MATLAB(R) 8.5 and the Signal Processing Toolbox 7.0.
% Equiripple Lowpass filter designed using the FIRPM function. % All frequency values are in Hz.
Fs = 240000; % Sampling Frequency
N = 40; % Order
Fpass = 4000; % Passband Frequency
Fstop = 24000; % Stopband Frequency
Wpass = 1; % Passband Weight
Wstop = 1; % Stopband Weight
dens = 400; % Density Factor

% Calculate the coefficients using the FIRPM function.
b = firpm(N, [0 Fpass Fstop Fs/2]/(Fs/2), [1 1 0 0], ...
    [Wpass Wstop], {dens});
Hd = dfilt.dffir(b);
```
Figure A.5: Frequency response of anti-alias filter.
A.5 Channel Select Filter

The channel select filter is applied just before FM demodulation to reduce out-of-channel interference. It was designed to accept most of the signal inside the 6.25kHz sidebands of a P25 channel but sharply reject components outside. This is realized with a 66-coefficient FIR filter, whose frequency response is shown in Figure A.6. The sharper rolloff increases the order, but this is a reasonable tradeoff due to the relatively low 48kHz sample rate. This filter was designed in Matlab’s `fdatool`, which generated the code in Listing A.5.

Listing A.5: Matlab code to generate channel select filter.

```matlab
1  % Generated by MATLAB(R) 8.5 and the Signal Processing Toolbox 7.0.
2  % FIR Window Lowpass filter designed using the FIR1 function.
3  % All frequency values are in Hz.
4  Fs = 48000;  % Sampling Frequency
5  N   = 64;    % Order
6  Fc  = 5500;  % Cutoff Frequency
7  flag = 'scale';  % Sampling Flag
8
9  % Create the window vector for the design algorithm.
10  win = hamming(N+1);
11
12  % Calculate the coefficients using the FIR1 function.
13  b   = fir1(N, Fc/(Fs/2), 'low', win, flag);
14  Hd  = dfilt.dffir(b);
```
Figure A.6: Frequency response of channel select filter.
Appendix B: Open Source Release

The software written for this research is all under Git version control and will be released open source, mostly using the MIT license, on the author’s Github account [122]. The software requires Rust, at least version 1.7.0, and can be built and tested using the Cargo build system distributed with Rust, e.g., cargo build and cargo test. Additionally, the supporting modules will be uploaded for an initial release to Rust’s “crate” database [123]. They can then be used as dependencies in other Rust projects by adding them to the project’s Cargo.toml. These releases will occur shortly after the thesis defense.
Appendix C: Hardware Schematic

The interconnection of hardware components for this receiver is shown in Figure C.1. A Raspberry Pi 2B is used as the core computer, and a TCXO RTL-SDR is simply connected to one of the USB ports. To enable the user interface, an Adafruit 16 × 2 LCD [124] and Adafruit rotary encoder [125] are connected through GPIO pins, with the numbered pins given in BCM pin numbering format. Then the device can be powered through the micro USB port, and audio is output through the 3.5mm jack.

Figure C.1: Schematic of receiver designed in this research.