UNIVERSITY OF CALIFORNIA, SAN DIEGO

Design and Implementation of an Encryption Framework for APCO P25 using an open source SDR platform in an OSSIE Environment

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Electrical Engineering (Signal Image Processing)

by

Anthony Anelechi Nwokafor Jr.

Committee in charge:

Professor William Hodgkiss, Chair
Professor Tara Javidi
Professor George C. Papen

2012

This Thesis of Anthony Anelechi Nwokafor Jr. is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California, San Diego

2012

DEDICATION

To my best friend, Ariel J. Willingham.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AES Advanced Encryption Standard

ALGID Algorithm ID

AMBE Advanced Multiband Excitation

APCO Association of Public-Safety Communications Officials-International

CF Core Framework

COM Computer On Module

DDC Digital Down-Converter

DES Data Encryption Standard

DUC Digital Up-Converter

DVSI Digital Voice Systems, Inc.

ES Encryption Sync

FDMA Frequency Division Multiple Access

GPP General Purpose Processor

HDU Header Data Unit

IMBE Improved Multi-Band Excitation

JPEO Joint Program Executive Office

JTRS Joint Tactical Radio System

KID Key ID

LC Link Control

LDU Logical Link Data Unit

LSD Low Speed Data

MI Message Indicator

OE OpenEmbedded

OFB Output Feed Back

P25 Project 25

PTT Push-To-Talk

SCA Software Communication Architecture

SDR Software Defined Radio

TDEA Triple Data Encryption Algorithm

TDES Triple DES

TDMA Time Division Multiple Access

TDU Terminator Data Unit

TIA Telecommunications Industry Association

UHD Universal Hardware Driver

USRP Universal Software Radio Peripheral

VC Voice Codeword

ACKNOWLEDGEMENTS

First, I would like to thank my Lord and Savior Jesus Christ for all the wonderful blessings he's surrounded me with.

I would like to acknowledge my adviser, Dr. William S. Hodgkiss for the time and effort he put into advising me on this work. Thank you. It is very much appreciated. I would also like to acknowledge the input from the WISE Group at California Institute for Telecommunications and Information Technology (CALIT2), specifically Per Johansson, Zhongren Cao, Wenhua Zhao, Jeff Cuenco and Justyn Bell. Their input and help was instrumental in figuring out various aspects of integration and debugging the encryption framework.

A big thanks to Miss. Ariel Willingham (Pookie) for her understanding, support and companionship throughout my years in school. I could not have asked for a better "best friend". I would love to acknowledge and thank my family; Dad, Mom, Chioma, Kelechi and Ugochi, for their love, concern, prayers, help and support, everyday. I would like to acknowledge all my friends and well wishers who through their concern, advice, prayers and moral support have made my journey through Graduate school memorable. Thank you all so much!

Last, but not least, I would like to acknowledge that this work was supported by the Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS) through SPAWAR Systems Center Pacific under contract no. N66001-08-D-0155/T.O.0001

ABSTRACT OF THE THESIS

Design and Implementation of an Encryption Framework for APCO P25 using an open source SDR platform in an OSSIE Environment

by

Anthony Anelechi Nwokafor Jr.

Master of Science in Electrical Engineering (Signal Image Processing)

University of California, San Diego, 2012

Professor William Hodgkiss, Chair

Secure and reliable communication is one of the most important issues in the public safety domain. For public safety and emergency response organizations such as the Police and Fire departments, reliability and security of their communications is fundamental and requires both authentication of users as well as encryption of voice and data communication. Project 25 (P25) public safety waveform is the waveform of choice for most public safety and emergency response organizations in Northern America and includes features to enhance reliability and security of communications. This thesis describes the design and implementation of an encryption framework for a P25 waveform in a Software Communication Architecture (SCA) environment on an open-source Software Defined Radio (SDR)

platform. The design and implementation of the framework which starts with a high level modeling of its state machine using pseudocode, goes through a bit-true intermediate implementation and ends with the final cycle-true and bit-true platform-specific implementation is discussed. This thesis proposes an encryption framework that is feasible for implementing the P25 encryption specifications and can be rapidly prototyped in an SCA environment on a cheap off-the-shelf SDR platform involving multiple processors.

Chapter 1

Introduction

Public safety and emergency response operations often require coordination across multiple government and civilian agencies. These operations are often geographically distributed, as different teams or agencies could be spread out over the region of operation carrying out specialized tasks while the command center may be situated in a remote location. Since different participating agencies may use different communication systems, a communication bottle neck may develop that can negatively impact activities essential to the emergency response or security operation. Interoperability has, therefore, become one of the most critical requirements for public safety radio systems.

A key component of the solution to interoperability is the capabilities provided by Software Defined Radio (SDR). SDR provides increased flexibility in interoperation and the ability to adapt to evolving technologies. These SDR capabilities allow key radio operating parameters to be controlled through software, leading to tremendous flexibility in the radio (e.g., changing frequency bands on-the-go or upgrading capabilities by downloading software over-the-air) [1].

By adopting SDR technology, different radio form factors from different manufactures can support multiple waveforms. Achieving this objective requires the use of a standardized open architecture – the Software Communication Architecture (SCA), which defines the common interfaces of waveform components [2]. The SCA is an architectural framework that was designed to maximize portability, configurability of the software (including changing waveforms) [3] and enable

software reuse, hence, reducing the development time and associated costs.

The Association of Public-Safety Communications Officials-International (APCO) Project 25 (P25) came into existence to address the issue of interoperability amongst government and civilian agencies.

1.1 What is APCO P25?

APCO P25 is the standard for the design and manufacture of interoperable digital two-way wireless communications products. It was developed as a collaborative effort between state, local and federal representatives and Telecommunications Industry Association (TIA) governance [2]. P25 has gained worldwide acceptance for public safety, security, public service, and commercial applications due to the benefits it offers. The project specifies a narrowband waveform with two phases of implementation which take different approaches to the Vocoder and Channel access schemes. Phase 1 waveform uses a 12.5 kHz bandwidth channel, with Frequency Division Multiple Access (FDMA) access methods and the Improved Multi-Band Excitation (IMBE) voice codec while the Phase 2 waveform uses a 6.25 kHz bandwidth channel with a 2-slot Time Division Multiple Access (TDMA) access scheme and the Advanced Multiband Excitation (AMBE)+2 voice codec for a reduced bitrate.

Among the many benefits of P25 is secure communications. Public safety radio systems are vulnerable to eavesdropping and can easily be exploited by criminals [4]. Readily available scanners can be used to receive voice communications on public safety radio systems, potentially exposing sensitive information to unauthorized listeners. To ensure that sensitive information is shared only among authorized individuals or organizations, the confidentiality of sensitive radio traffic needs to be ensured. This is typically accomplished through voice encryption [4].

APCO P25 supports secure communication through the use of encryption, key management and equipment authentication. This thesis focuses on the implementation of an encryption framework that meets the requirements of the APCO P25 encryption specifications.

1.2 Overview

An overview of the thesis document is presented next with a brief description of the topics discussed in each chapter.

1.2.1 Background

Chapter 2 presents and discusses the concepts of Encryption and SDR. It describes the SDR environment used in this thesis and also describes the P25 implementation on which the encryption framework is built.

1.2.2 Encryption Framework

Chapter 3 discusses the software and hardware architecture of the Encryption framework and describes the software implementation. The discussion on the architecture and design of the framework covers the specifications and constraints imposed by the requirements, the P25 Encryption state machine and multiprocessor design. The discussion on implementation covers the usage of the OpenSSL Library, integration of the Encryption framework into the existing P25 implementation and discusses various issues encountered during this stage as well as the functional and performance tests that were performed.

1.2.3 Summary

Chapter 4 presents a summary of the work done in this thesis. It presents its conclusions and provides recommendations for possible future work.

Chapter 2

Background

2.1 Encryption

Encryption is defined as the process of changing information from one form to another in an attempt to hide its meaning. In the context of data communications, encryption is the process of transforming raw data "plain text" to cipher text in order to make the data unintelligible to unauthorized persons [5]. Encryption is achieved by applying a specified algorithms to a block of data. The reverse process of converting cipher text back into its original "plain text" format is called decryption. Encryption and decryption are facilitated by the use of a piece of information referred to as a key. A key is unique information, known only to the message originator and the intended receiver, which is used to control the encryption process, thus yielding unique cipher text that can only be decrypted using the key. Encryption keys selected at random and of sufficient length are considered almost impregnable [5]. To give an idea of how impregnable an encryption key could be using a brute force attack, a key length of 128 bits which is equivalent to 16 characters selected from the 256 available ASCII characters could take far longer than 15 billion years to decode, assuming that the attacker was attempting 100 million different key combinations per second.

In many contexts, encryption implicitly refers to the process of decrypting the cipher text. Two classes of encryption exist today. They are:

- **Symmetric encryption**, which requires the same key for both encryption and decryption and
- **Asymmetric encryption** also referred to as public-key cryptography, which requires a pair of keys; one for encryption and the other for decryption.

Encryption is used to protect the confidentiality of data in transit and data at rest. It is incorporated in many applications where confidentiality of transmitted messages is of importance such as the data networks and mobile telephones. Encrypting data in transit aids with security as it can be difficult to physically secure all access to the channels on which the data is communicated. For encryption to be effective as a security measure, it must be applied at the time of message composition on the originating device to avoid the possibility of tampering. Otherwise a message could be intercepted and tampered with at any point between the sender and the encryption agent.

It is easy to see why encryption is important and beneficial as part of the security suite for applications involving homeland security and public safety. Unencrypted communication on radio systems for public safety missions could prove disastrous if critical information is intercepted by an unauthorized eavesdropping intruder. Project 25 specifies and standardizes three encryption algorithms that could be implemented to enhance security. They are the Data Encryption Standard (DES), the Triple Data Encryption Algorithm (TDEA) or Triple DES (TDES) and the Advanced Encryption Standard (AES). The encryption framework implemented in this thesis meets the P25 specifications for DES and AES block encryption schemes only but could easily be extended to include the TDES.

2.2 Software Defined Radio

Components of radio communication systems such as filters, mixers, modulators and demodulators have, traditionally, been implemented in special-purpose hardware with predetermined functionality due to the need to meet certain timing constraints. These hardware based radio devices limit cross-functionality and reuse since modifications to any aspect of the radio would require physical intervention.

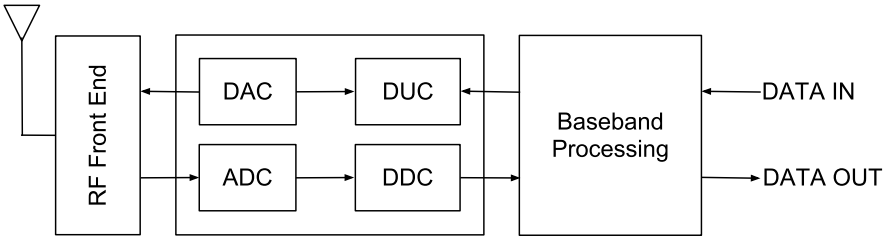


Figure 2.1: Generic SDR Architecture

With the emergence of computationally powerful General Purpose Processor (GPP), it has become increasingly feasible to implement, in software, many of the radio components that have typically been developed in special-purpose hardware. This is the concept of a SDR system. Simply stated, SDR can be defined as radio in which some or all of the physical layer functions are software defined [6].

SDR refers to radio system technologies where one hardware unit can receive, process and decode multiple signals in software. These signals can vary largely in frequency and protocol. The bulk of the signal processing is usually performed on a GPP. Figure 2.1 shows a generic SDR architecture with three main parts: an analog RF frontend component, an IF component, responsible for Analog to Digital and Digital to Analog conversions as well as Digital up and down conversions, and finally, a baseband processing component responsible for all other processing such as framing/defaming data. Listed below are some of the benefits associated with SDR:

- It enables the implementation of multiple radio systems using a common platform architecture.
- A single SDR system can be programmed with multiple functionality, that can be switched on-the-go.
- Software components can be reused across radio systems, dramatically reducing development costs.
- SDR systems are highly configurable with over-the-air reprogramming, allowing new features and capabilities to be added while the radio is in service, and reducing the time and costs associated with operation and maintenance.

The focus of this thesis is on the implementation of an Encryption archi-

tecture that meets the requirements for the encryption of voice data as laid out in the P25 specifications. A detailed discussion of the P25 implementation upon which the encryption framework is built can be found in section 2.5.

The P25 implementation used was developed on an open source multi-processor SDR platform built with off-the-shelf components. The SDR platform can be divided into Software and Hardware environments. The software environment consisted of multiple Linux operating systems, the OSSIE open-source SCA implementation, the Universal Hardware Driver (UHD) library, theDigital Voice Systems, Inc. (DVSI) AMBE+2^{TM} Vocoder library, OpenEmbedded and TI DSP/BIOS Link Library. The hardware environment consisted of an x86 GPP, a Gumstix Overo Tide Computer on Module with TI OMAP processor, a Gumstix Tobi Expansion Board and a Universal Software Radio Peripheral (USRP) N210 with WBX RF daughter-board. Figure 2.2 shows the SDR environment and how the components of the software and hardware environments are related. These components are described in greater detail in the following section.

2.3 SDR Hardware Environment

The hardware environment is made up of specific hardware that define the physical architecture of the SDR and in turn the physical architectures of the P25 radio system and the Encryption framework.

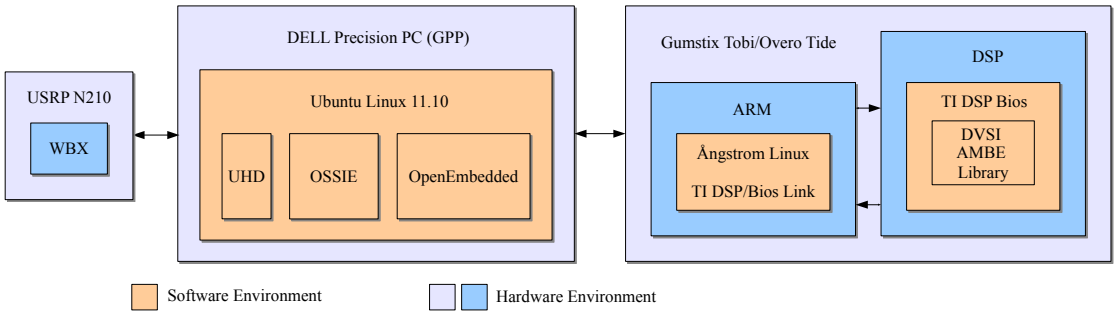


Figure 2.2: SDR Hardware and Software Environment.

(a) Dell Precision 690 PC

(b) USB Audio device

Figure 2.3: GPP Components

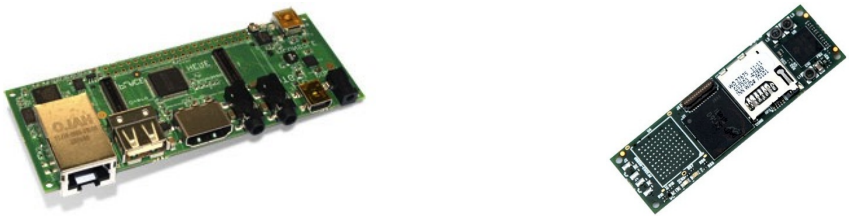
2.3.1 GPP

A Dell Precision 690 Desktop PC shown in Figure 2.3a was used as the GPP in the SDR platform. Configured with a single Dual core Intel Xeon 5100 Processor clocked at 2.33 GHz, 6 GB of RAM and a Gigabit Ethernet port, it proved to be a more than capable GPP for the SDR platform. A generic USB audio device was substituted for the PC's inbuilt audio device, as it was discovered that the USB audio device performed more reliably in terms of its interaction with the Linux ALSA audio driver and its sampling consistency.

2.3.2 Overo Tide COM and Tobi Expansion board

The Overo Tide is a Computer On Module (COM) board from Gumstix configured with a TI OMAP 3530 Applications Processor, 512 MB of RAM and a microSD card slot for storage expansion. The TI OMAP 3530 processor is a multicore system on chip (SoC) for portable and mobile media applications. This processor combines a general-purpose 720 MHz ARM Cortex-A8 processor core and a 520 MHz TMS320C64x+ DSP core into a single package.

The Overo Tide COM requires an expansion board that provides power and other peripherals. In this SDR platform an Overo Tobi Expansion board was used for that purpose. The Tobi board provides a 140 pin dock for the Overo Tide COM from which the COM gains access to the Tobi's 10/100BaseT Ethernet port, USB ports, DVI display port and audio card. Figure 2.4 shows images of the Overo Tide COM and the Tobi Expansion board. The Overo Tide could also be considered a GPP component in this environment and is used only for the purposes of voice



(a) Tobi Expansion board (b) Overo Tide COM

Figure 2.4: Gumstix Boards

data encoding and decoding.

2.3.3 USRP N210 and WBX RF Daughterboard

The USRP is an open-source host-based radio platform designed and sold by Ettus Research that is intended to be an inexpensive hardware platform for software defined radio. It is commonly used by hobbyists and in research environments. The USRP N210 is one of the products from the USRP product family and is fitted with a Xilinx Spartan-3A DSP 3400 FPGA, a Gigabit Ethernet interface, one dual 100 MS/s, 14-bit, analog-to-digital converter and one dual 400 MS/s, 16-bit, digital-to-analog converter. It also offers flexible clocking and synchronization using external clock sources. The N210 connects to a host computer using the Gigabit Ethernet interface. This interface is used by software on the host computer to send/receive data from the USRP and also load/reload the firmware on the FPGA. Ettus Research provides an API library (see Section 2.4.4) and a set of programs for these purposes.

(a) USRP N210 (b) WBX RF Daughterboard

Figure 2.5: USRP Components

The USRP N210 employs a modular design that enables it to operate between DC and 6 GHz. A motherboard provides the host interface, FPGA, ADC, DAC as well as clock generation and synchronization functionality, while a daughterboard, that attaches to the motherboard, provides the up/down conversion to and from a specified carrier frequency, filtering and other signal conditioning. Figure 2.5 shows the USRP N210 and the WBX RF daughterboard used in this SDR environment. The WBX provides 40 MHz of bandwidth capability and has an operating frequency range of 50 MHz to 2.2 GHz which makes it suitable for Land-Mobile Radio (LMR) communication applications, such as the narrowband P25 waveform, which operate in the UHF and VHF bands. The WBX also has two antenna ports: a dedicated receiver port and a transceiver port. In this SDR platform, the USRP N210 and WBX are used for up and down conversion of the P25 complex baseband signals to and from the carrier frequency.

2.4 SDR Software Environment

The software environment is made up of mostly open-source software that define the logical architecture of the SDR and in turn the logical architectures of the P25 radio system and the Encryption framework.

2.4.1 OSSIE SCA Implementation

SCA is an open architecture framework developed by the U.S. Department of Defense Joint Tactical Radio System (Joint Tactical Radio System (JTRS)) as a way to ensure portability and interoperability of protocols on different radios. It provides specifications on how hardware and software components of an SDR are to operate together. The SCA defines a Core Framework (CF) which is the essential set of open application-layer interfaces and services that provide an abstraction of the underlying system software and hardware [3]. These interfaces and services cover the deployment, management and interconnection of software components in the SDR.

OSSIE SCA is an open source implementation effort of the SCA CF from

Virginia Tech. It includes a set of tools for rapid development of SCA components and waveform applications and also includes a library of pre-built components and waveform applications. Although it is not necessary that a waveform such as P25 be developed in an SCA environment, it is beneficial to the rapid porting and deployment of the waveform on other SDR systems supporting an SCA environment. OSSIE SCA has also often been referred to as being "SCA-like" since it is not a complete implementation of the JTRS SCA specification.

The OSSIE SCA implementation runs within the Ubuntu Linux operating system on the GPP. It directly interacts with the USB audio card and the Overo Tide COM which houses the vocoder.

2.4.2 DVSI AMBE+2^{TM} Vocoder Library

The DVSI AMBE+2^{TM} Vocoder library is the latest iteration of enhanced vocoders for P25 and is fully interoperable with the current 7200 *bps* IMBE vocoder specified in the P25 standard. It is a proprietary vocoder and as such is closed-source. In addition to voice encoding and decoding, the software library includes other features such as DTMF and single tone detection and voice activity detection (VAD). The Vocoder expects as input PCM speech sampled at 8 kHz and outputs synthesized speech at the same rate after decoding. Forward Error Correction (FEC) encoding is applied to the encoded speech frames and FEC decoding is applied to received FEC encoded speech frames before decoding and synthesizing the speech.

The AMBE+2^{TM} vocoder library is packaged as a dual-rate vocoder with full-rate (7200 *bps*) and half-rate (3600 *bps*) modes. Both bit rates use a 20 ms frame consisting of two $10ms$ subframes. Each 20 ms frame equates to 160 samples of an 8 kHz sampled PCM speech. The samples are represented as 16 bit signed integers resulting in a total of 320 bytes for each 20 ms frame. The vocoder encodes the speech frames and produces a quantizer-frame of compressed data whose bit length depends on the rate mode of the vocoder. The full-rate mode produces 144 bits per frame (88 voice data bits and 56 FEC bits) while the half-rate mode produces 72 bits per frame (49 voice data bits and 23 FEC bits) [7]. The P25

standard Phases 1 and 2 both use the full-rate mode while the AMBE+2^{TM} can also operate at half-rate for Phase 2.

DVSI offers multiple hardware and software products that implement the AMBE/IMBE vocoders. The particular library used in this SDR platform was compiled for the TI C6x DSP series and is the only reason for the addition of the Gumstix Overo Tide and Tobi boards to this SDR platform. An x86 version of the library would have been preferred but the cost of acquiring that version of the library was quite steep. Open source vocoder projects exist, such as codec2.org, that implement a vocoder that can easily be substituted in place of the proprietary AMBE/IMBE vocoder but these were not viable options for the P25 implementation used on this SDR platform since one of its aims was to show ability to communicate with off-the-shelf P25 Handsets.

The AMBE+2^{TM} Vocoder Library provides separate Voice coding and FEC coding functions which is necessary for the implementation of P25 encryption. According to the P25 standard, encryption of voice data must occur after voice encoding and before FEC is applied. Conversely, FEC decoding must occur on encrypted voice data before decryption of the data. Since these operations must be carried out one after the other, it would make sense to have the Vocoder and encryption components reside physically and logically close together. But for radio systems such as those used by the military and the agencies involved in Homeland Security, the National Security Agency has specified minimum physical distances between components and wires processing or carrying red (unencrypted signals) and black (encrypted signals). In this case, encryption may happen in another module on the same processor or on an entirely different device or processor within the same radio and this would require the transfer of encoded voice data to the encryption device, returning the encrypted voice data for FEC encoding before it is transmitted. The encryption framework proposed in this thesis does precisely this due to the architectures of the open-source SDR platform and the P25 implementation used.

2.4.3 OpenEmbedded, BitBake and TI DSP/BIOS Link

OpenEmbedded (OE) is an open-source cross-compilation environment for building embedded Linux distributions. It offers support for many hardware architectures including ARM. Its advantages include easy customization and rapid build out of an embedded Linux environment. In order to perform its build tasks, OE requires a tool called BitBake which is a simple tool for task execution and is most commonly used to build packages. It is the basis for OE. A set of rules and instructions called a "recipe" have to be generated to build an application for a specific environment or architecture. These specify the location of source code for the application, options used during the configuration and build phase, etc.

OE was installed on the GPP (Dell desktop machine) and used to cross-compile the Angstrom Linux distribution for the ARM processor on the Overo Tide COM. DSP/BIOS Link from Texas Instruments, a foundation software for inter-processor communication across GPP-DSP boundary, was also configured and packaged for the ARM processor using OE. DSP/BIOS Link is an open-source library providing a generic API that abstracts the underlying characteristics of the physical link between the ARM and DSP. The provision of such a library eliminates the need to develop a communication API from scratch and allows for development of higher level abstraction interfaces such as Remote Procedure Calls. Using DSP/BIOS Link an asynchronous data transport protocol was built to move audio and control data between the ARM and the DSP on the OMAP processor.

2.4.4 Universal Hardware Driver (UHD) library

The UHD library is an open-source multi-platform hardware driver for the open-source USRP hardware. It provides the host driver and API that allows applications on a host PC send application data and control messages to and from the USRP. The UHD library resides on the GPP (Host PC) and allows it to interface directly with the USRP.

2.5 P25 Implementation

The Encryption Framework discussed in Chapter 3 was developed as part of a project sponsored by the Joint Program Executive Office (JPEO) JTRS through SPAWAR Systems Center Pacific. The project aimed at the rapid development, implementation and porting of a P25 Phase 1 waveform across multiple SDR platforms ranging from Spectrum Signals Processing's SDR 4000 Integrated Development System for SDR Application Development to the open-source SDR platform described in earlier in this Chapter. Part of goals of the project was to demonstrate the ability to rapidly prototype a waveform using SCA and SDR methodologies on a cheap open-source platform. Due to the complexity of the P25 standards and time constraints a subset of the entire standard was chosen for implementation. The subset of P25 functionality chosen was guided by SPAWAR such that the final implementation would be able to perform basic group and unit-to-unit voice calls with any off-the-shelf P25 hand-held radio.

The P25 implementation began with initial waveform development using a methodology that first required the waveform to be modeled in Matlab as a platform agnostic executable waveform. An intermediate implementation was then produced using open tools on a generic Linux platform before the final platform specific implementation on the SDR 4000. This initial work is discussed in greater detail in [8].

2.5.1 Development of components and modifications

The result of the initial P25 development effort was a basic P25 implementation allowing group and unit-to-unit voice calls. This implementation was then ported to the open-source SDR platform described here using the same development methodologies mentioned earlier. Although most of the code in the previous implementation was written in C (with the exception of the FPGA code) and could fairly easily be ported to the open-source platform, quite a number of modifications had to be made for the port to work due to major differences in hardware and software architecture of the platforms. The following is a list of high level

changes that had to be made:

1. Communication across multiple hardware devices and processors had to be implemented since these components were not as integrated as was the case with the SDR 4000 which came with quicComm, Spectrum's hardware abstraction layer and API library.
2. Functions that previously existed in the FPGA on the SDR 4000 had to be relocated and re-implemented on the GPP since the open-source SDR platform did not readily have a user accessible FPGA.
3. The software vocoder had to be integrated into the open-source SDR platform and tested whereas this was not the case with the SDR 4000 environment where a hardware vocoder solution from DVSI was used.
4. Differences in the implementation of the SCA CF in OSSIE and the implementation in the comprehensive proprietary version that came with the SDR 4000 plus a less intuitive and not as feature rich component design tool for OSSIE made it difficult to simply move SCA components and devices from the SDR 4000 environment into the open-source SDR environment.

Figure 2.6 shows a block diagram of the P25 implementation (without encryption) on the open-source SDR platform. The three main hardware components of the platform are shown as separate blocks connected by red colored arrows which represent IP socket connections. The PC is connected to the USRP using UDP sockets and to the OMAP (Overo Tide) using TCP sockets. The modules within the OSSIE CF communicate using CORBA. The communication between the OSSIE CF and the C4FM Modem is established using Named Pipes (FIFO) while the communication between OSSIE CF and the Java GUI component is established using TCP sockets. In SCA contexts, such as in OSSIE CF, modules that act as interfaces to hardware devices or external modules are referred to as Devices (red blocks) while autonomous modules within the CF that provide their own functions for data processing are referred to as Components (blue blocks).

The Java GUI represents the radio's User Interface and allows a user manipulate the configuration of radio parameters such as the Unit ID (UID) of the radio, Destination ID in the case of a unit-to-unit voice call, the Network Access

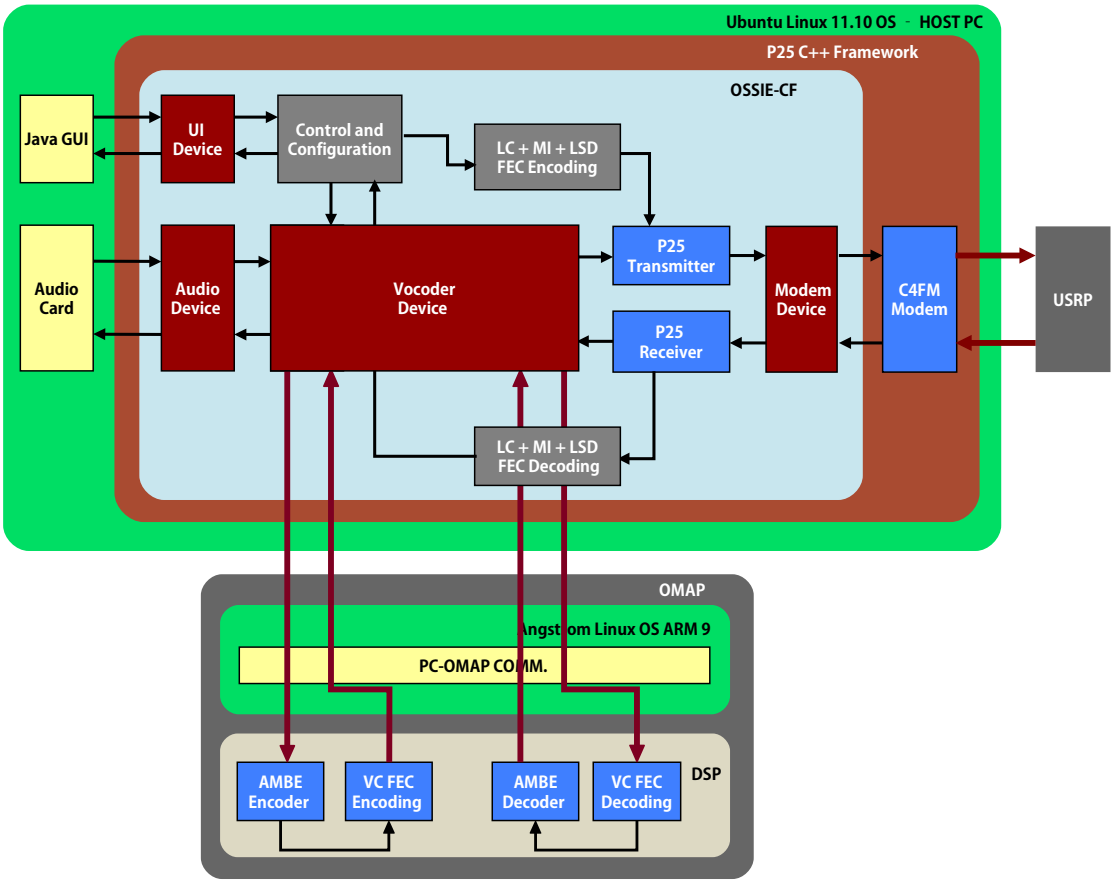


Figure 2.6: Block Diagram of P25 Implementation without Encryption

Code (NAC), etc. It is also the driver for the radio system using a Push-To-Talk (PTT) button to switch the radio between transmit and receive modes. Pressing the PTT button causes the emission of a control packet to the OSSIE CF that results in the activation of the Vocoder Device, to accept data from the Audio Device, and the generation of FEC encoded Link Control (LC) information using data from the control packet. The Vocoder Device routes the audio data to the OMAP processor where it undergoes AMBE encoding and FEC encoding before being sent back to the Vocoder Device. The encoded voice data is then assembled into P25 frames, in the P25 Transmitter component, along with the LC and other control information. These frames are sent to the C4FM modem for baseband processing and modulation before they are sent to the USRP for transmission.

The C4FM modem is a compound component consisting of a bits-to-symbol conversion module and a modulator/demodulator module. A P25 frame bitstream enters the bits-to-symbol conversion module at a rate of 9600 baud. The bits are converted to symbols with 2 bits to a symbol, 10 times upsampled and filtered using a raised cosine filter. The bits-to-symbol module produces an output bitstream at 48 Kilo-Symbols per second (KSps) or 96 Kbps. This bitstream enters the modulator/demodulator module where it is frequency modulated and resampled to 200 KSps before exiting to the USRP to be upconverted and sent over the air. The C4FM modem outputs baseband complex IQ signal to the USRP.

On the Receiver path, the USRP downconverts the received signal from the carrier frequency to 200 KSps and sends it to the modulator/demodulator module where it is resampled to 48 KSps and demodulated. The demodulated signal is forwarded to the bits-to-symbol module where a synchronization search takes place. Once the synchronization has been determined, the bits-to-symbol module decodes the symbols to bits and forwards the resulting P25 frame bitstream to the P25 Receiver component which then extracts the LC information, other control information and the voice data. The voice data is sent to the Vocoder Device from where it is eventually routed to the OMAP for FEC decoding and AMBE decoding. The OMAP sends back the resulting synthesized PCM audio data which is forwarded to the Audio Device for playback. The LC and other control information extracted from the P25 frame are sent to the UI Device to update the Java GUI.

This P25 implementation is the basis of the Encryption Framework discussed in the next Chapter. Most of the Encryption Framework is concerned with P25 components in the OSSIE CF and the OMAP processor.

Chapter 3

Encryption Framework

In this chapter the architecture, design and implementation of an encryption framework for an implementation of Project 25 on the open-source SDR platform discussed previously is presented. A discussion of the problems encountered and the solutions implemented is also presented.

3.1 P25 Encryption Specifications

The P25 Encryption specification is designed to be compatible with voice messages and data packets on both trunked and conventional radio systems. Three different encryption processes are currently standardized in the P25 specifications. They are DES, TDEA and AES. All three encryption processes are used in the Output Feed Back (OFB) mode and are denoted as DES-OFB, TDEA-OFB and AES-OFB respectively. The DES algorithm uses a block length $n = 64$ bits with a key variable of length $k = 64$ bits. The TDEA algorithm uses 3 chained DES encryption/decryption cycles with a key bundle consisting of 3 separate 64-bit DES key variables (K1, K2, and K3) for each encryption/decryption operation. The AES algorithm uses a block length $n = 128$ bits and a key length $k = 256$ bits.

In OFB mode keystream blocks are generated, which are then XORed with the plaintext blocks to get the ciphertext. Due to the symmetry of the XOR operation, the encryption and decryption processes are exactly the same hence the P25

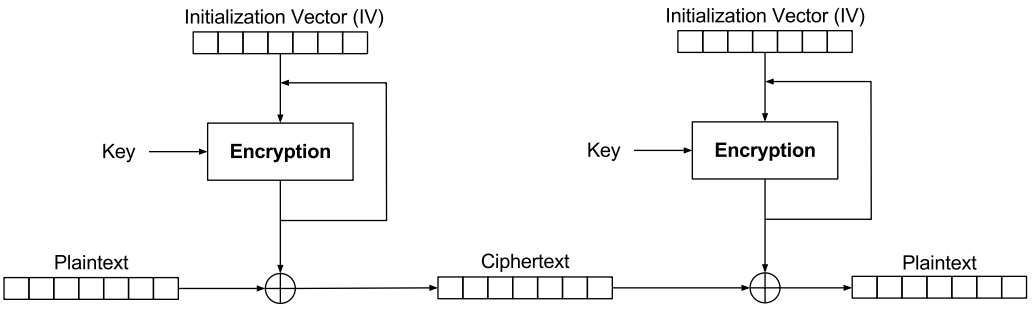


Figure 3.1: Output Feedback (OFB) Mode Encryption

standard specifies that OFB operation shall always use the encrypt mode for both the transmitter and receiver. That is, both encryption of plaintext and decryption of ciphertext should be performed using the encrypt mode for the algorithms. Figure 3.1 shows a block diagram of the OFB mode of encryption where the plaintext and ciphertext blocks have a length equal to the blocksize n of the particular algorithm. To recover the plaintext, the roles of the ciphertext and plaintext blocks only need to be reversed as shown. For streams of bits greater than the blocksize n of the chosen algorithm, the stream is broken into segments of length n and the encryption algorithm is iterated until the last segment is processed. Zero-padding may be applied to the last segment if its length does not equal the required blocksize n for that algorithm.

Furthermore, the P25 standard specifies that encryption and decryption functions should generally take place near the end points of a message path in a system which means that encryption and decryption functions should be provided at the points were voice information is AMBE encoded or decoded. Hence encryption must be performed before FEC is applied to the encoded voice data and decryption performed after FEC decoding of encoded voice data is done. This does not necessarily pose a problem when dealing with an SDR system where all components reside on a single processor but could become a challenge when the SDR system has components distributed over multiple processors as is the case with the open-source SDR platform used here. In this case, the encryption module resides in a separate processor on a separate hardware device from the Vocoder requiring

the coordination of data transfer in order to achieve optimum performance while meeting the timing constraints of the P25 standard.

A clock schedule is specified for synchronization of the encryption state machine on the transmitter and receiver. The clock schedule specifies the generation of a new Initialization Vector (IV) using a Linear Feedback Shift Register (LFSR) that is clocked 64 times to generate its next state from its present state. The IV is sent out regularly to help the receiver synchronize its encryption state machine. An encryption schedule for the sequence of input bits for encryption is also specified in the standard and is discussed in the next section. More details on the P25 Block Encryption specifications can be found in [9].

3.1.1 P25 Encryption Schedule

The P25 Encryption schedule specifies which bits from the P25 superframe structure are encrypted and the order they are encrypted in. Figure 3.2 shows the frames that typically make up a P25 message transmission. Each superframe contains 360 ms of voice data.

A superframe is made up of two Logical Link Data Unit (LDU) frames referred to as LDU1 and LDU2. The superframe repeats until a Terminator Data Unit (TDU) frame which signifies the end of a message. The Header Data Unit (HDU) frame is an optional frame that signifies the start of a message. It contains control information about the message it precedes such as the Talk Group ID which the receiver looks at to determine if the frames of the message should be processed. The HDU also contains an information field that signifies to that a message is either encrypted or unencrypted. If the message is encrypted, this field signifies the Algorithm in use. The ID of the Key being used for encryption at the

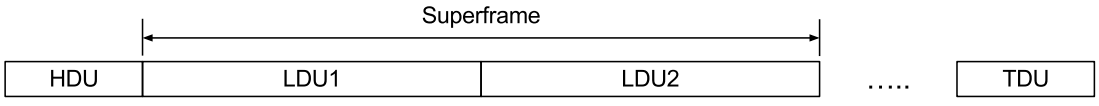


Figure 3.2: P25 Superframe

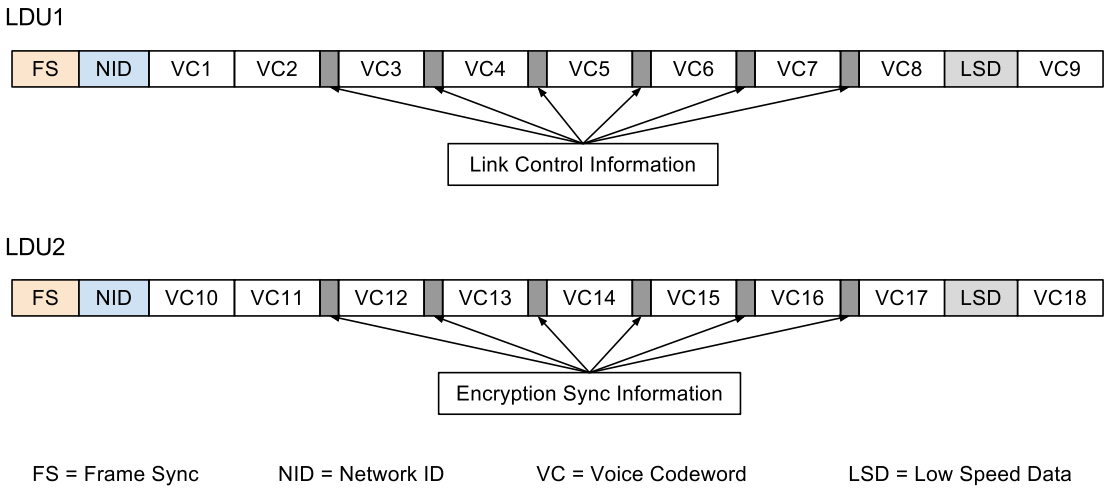


Figure 3.3: P25 LDU1 and LDU2 frame details

transmitter and the current IV are also part of the HDU. The HDU frame is an optional frame because the information it carries is repeated and updated in the subsequent superframes.

Each LDU frame, as well as the other frame types, begins with a Frame Synchronization (FS) sequence followed by a Network ID (NID). This allows receivers to enter "late" into a conversation in the event that the receiving radio was turned on after a message had begun. Both LDU1 and LDU2 frames contain 9 Voice Codewords (VCs) as depicted in Figure 3.3. A VC is generated using an FEC encoding schedule that takes in a 20 ms block of AMBE encoded audio, 88 bits in length, and transforms it into a 144 bit error protected voice codeword. LDU1 and LDU2 frames contain different control information. Some of the control information contained in these frames is also contained in the HDU. The LDU1 frame contains the LC, a 240 bit block of FEC encoded information about the source and destination of the superframe. The LC contains other information bits that are not necessarily relevant in the P25 implementation used here. The LDU2 frame contains the Encryption Sync (ES) control information. Also a 240 bit block, the ES is FEC encoded but contains only information pertinent to encryption which are the Key ID (KID), Algorithm ID (ALGID) and the Message Indicator (MI). The encryption sync information could be used to support a multi-key encryption

system but is also used for single key and clear messages. The LDU frames also carry a Low Speed Data (LSD) field. The LSD information consists of 4 bytes. The first two bytes are sent in LDU1 and the last two bytes are sent in the LDU2. The LSD is designed to be used by radio manufacturers for proprietary signaling or applications that require low data rates such as GPS location reporting.

The P25 Encryption Schedule is such that it repeats every 213 octets, 112 octets for LDU1 and 101 octets for LDU2. The discrepancy in the number of octets required for the LDUs is due to the ES control information not being encrypted in LDU2. This allows "late" entry into encrypted conversations. A radio entering "late" into a conversation can receive the KID, ALGID and MI and synchronize its encryption state machine with that of the transmitter to receive and decrypt the rest of the message. Table 3.1 shows the break down of encrypted information for one P25 superframe. A byte level detailed encryption schedule for both LDU1 and LDU2 frames showing the input and output block boundaries for DES ($n = 64$) and AES ($n = 128$) encryption can be found in [9]. It is necessary to note that the input and output block boundaries in the P25 encryption schedule sometimes straddle as many as three VCs but are always aligned at the end of an LDU. That is, at the end of every 9 VCs in the encryption schedule, the bits are aligned such that no padding is necessary to meet up the required blocksize for a particular algorithm. The straddling of multiple VCs to form an input block increases the complexity of the Encryption Framework state machine as the correct number of

Table 3.1: P25 Superframe Encrypted Information

Name	**Size (octets)**
Reserved	3
Link Control Information	8
IMBE frames 1 - 8	88
Low Speed Data	2
IMBE frames 9 - 17	99
Low Speed Data	2
IMBE frame 18	11
Total	213

VCs must be available to produce one output encrypted or decrypted VC.

3.2 Architecture and Design

The Encryption framework architecture is constrained by the architecture of the SDR platform and the P25 implementation in which it must exist. Hence, the framework is divided into three components; one component exists as an SCA Encryption Device in the OSSIE CF on the GPP (Host PC), another component resides on the ARM processor and the final component on DSP processor. The latter two components are embedded within the GPP-ARM communication infrastructure on the ARM and the AMBE Encoder/Decoder modules respectively. These components of the Encryption framework provide the awareness of Encryption capabilities to the modules they reside in and are primarily responsible for maintaining the state of the Encryption framework in those modules. Figure 3.4 shows a block diagram of the P25 implementation with the encryption framework. The Vocoder Device is shown as two separate devices to simplify the block diagram and aid understanding of the flow of data.

When encryption is requested from the GUI, control information must be sent to the framework components on the ARM and the AMBE Encoder/Decoder modules to notify them of the encryption request. This forces AMBE encoded voice data to be sent back to the Host PC for encryption before being FEC encoded in the case of transmission and forces encrypted AMBE encoded voice data to be sent back for decryption before AMBE decoding in the case of reception. When encryption is not requested, the entire encryption framework is bypassed. The exchange of control and data packets between the Framework components is facilitated by the inter-processor communication fabric developed for the respective processors on which they reside. These facilities are augmented to understand and interpret special control messages that turn encryption mode on and off.

The Encryption Framework is designed to be as modular and reconfigurable as possible given the constraints of the SDR platform and the P25 implementation. It is designed such that platform specific communications are separated from the

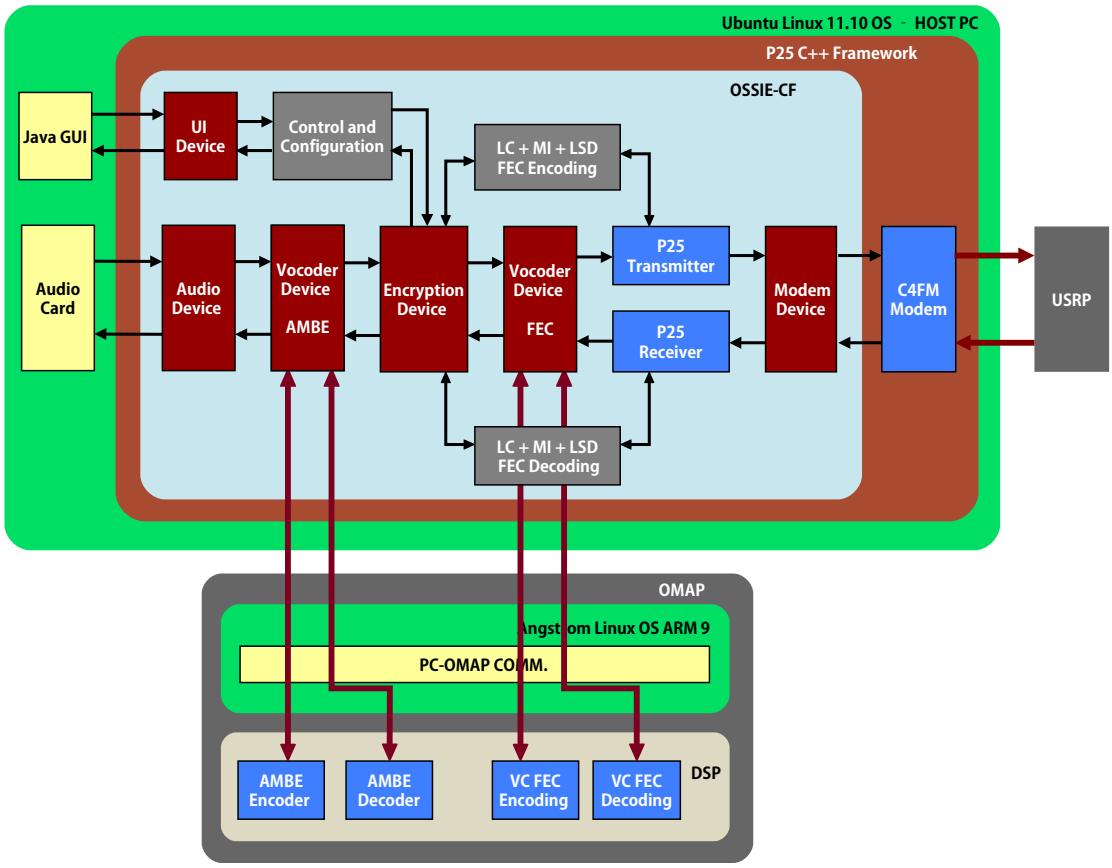


Figure 3.4: Block Diagram of P25 Implementation with Encryption

core encryption and data processing functions. The design of the framework also allows for easy drop-in replacement of the encryption algorithm as long as the algorithm does not apply non-linear transformations to the plaintext to generate ciphertext.

3.2.1 GPP (Host PC) Encryption Component

The Encryption device, which is the main component of the Encryption Framework, directly interacts with only four components in OSSIE. They are the UI device, the Vocoder device, the P25 Transmitter and P25 Receiver components. The latter two components have the functions of constructing or deconstructing valid P25 frames. The two components together are also interchangeably referred to as the Packetizer. The Encryption device is implemented as an SCA device

instead of a component since there are cases, as described earlier, when a radio system might have separate encryption hardware that performs the encryption and decryption tasks.

Data of a few different types are processed by the Encryption device; VCs from the Vocoder device and LC information, LSD and MI from the Packetizer. The Encryption device reads and writes the Packetizer data (LSDs, LCs, MIs) in both Transmit and Receive modes using separate read and write FIFO buffers. A set of buffers is shared between the Transmitter and Receiver components since the P25 system is implemented as a half duplex system and can only be in either Transmit or Receive mode at any point in time. Separate buffers are used for reading and writing VCs from the Vocoder device. The actual tasks of encryption and decryption, which follow the P25 encryption schedule in [9], are performed in the Encryption device. The device is dependent on the state of the P25 system and sits in Standby mode until it receives a control message from either the Transmitter or Receiver components, in which case it switches to either Transmit mode or Receive mode, respectively. At the end of an encrypted transmission, the device returns to Standby mode once all of its buffers have been emptied. An internal state machine, specific to the P25 encryption schedule, is also maintained within the Encryption device.

Figure 3.5 shows the important blocks of the Encryption device. A total of six processing threads, shown in green, are instantiated on the Encryption device. Two threads handle data input from the Transmitter and Receiver components and put received data into a single Packetizer input FIFO. Another thread handles the data input from Vocoder Device and puts the received data into a Vocoder input FIFO. One thread handles the main task; pulling data from the Input FIFOs for encryption or decryption depending on radio mode and putting data into two output FIFOs. The last two threads handle sending encrypted or decrypted data from the output FIFOs back to the Vocoder device, the Transmitter and Receiver components. Modifications to the original Vocoder device were necessary since it must not just receive, in the case of Transmit mode, the final AMBE encoded and FEC encoded VCs from the OMAP but must also be prepared to receive,

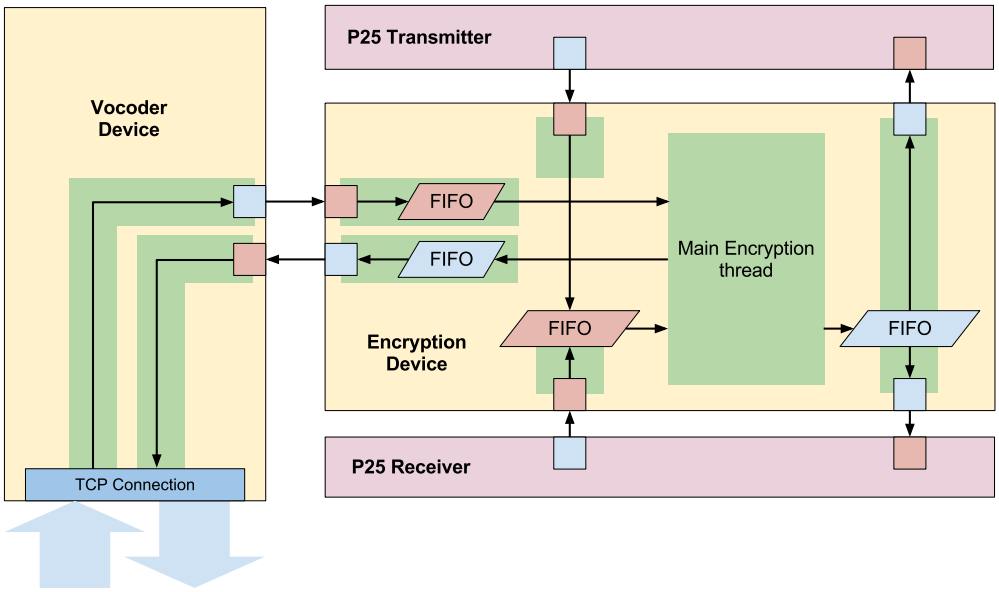


Figure 3.5: Encryption Device Block Diagram

intermediately, AMBE encoded audio data to be encrypted and sent back to the OMAP. To this effect, two threads are added to the Vocoder device; one to receive data from the OMAP and forward it to the Encryption device and the other to receive data from the Encryption device and forward it to the OMAP. These threads interact with the OMAP processor using a shared TCP socket connection and are bypassed completely, along with the Encryption device, when encryption is not required. These threads comprise the Host PC side of a logical channel that connects the DSP Encryption component to the Encryption device.

3.2.2 OMAP Encryption Component

The OMAP Encryption component is composed of ARM and DSP components which work together to provide the Encryption device the required number of VCs at each point in the Encryption Framework state machine based on the P25 Encryption scheduled. However, these components do not communicate directly with the Encryption device but are interfaced through the Vocoder device. The ARM and DSP encryption components exchange data using DSP/BIOS Link. Figure 3.6 shows a block diagram of the OMAP Encryption component. As with

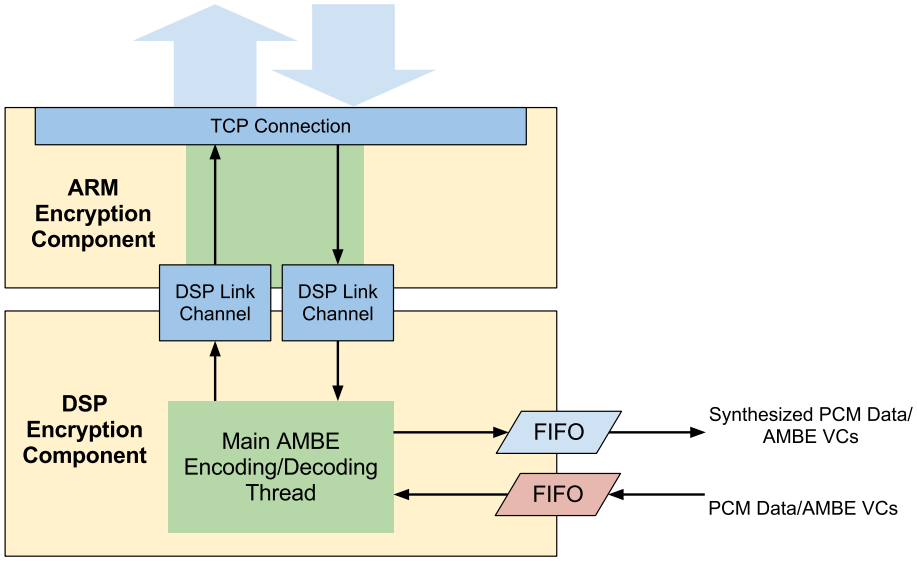


Figure 3.6: Omap Encryption Component Block Diagram

the Encryption device, this component is completely bypassed when the radio transmission does not require encryption.

OMAP ARM Encryption Component

The ARM Encryption Framework component is the ARM side of the logical channel that links the DSP and the Host PC Encryption components. It simply monitors the output from the DSP encryption component using a DSP/BIOS Link Channel and forwards any data it receives to the Vocoder device. It also listens to the Vocoder device using a TCP socket connection and forwards received data to the DSP encryption component. Both tasks are performed using a single thread and two connection resources that are multiplexed for either task. Though the ARM Encryption component is data driven it maintains an internal state machine that determines how much data it should expect to send or receive to and from the DSP component or the Vocoder device.

OMAP DSP Encryption Component

The DSP Encryption component is the originator of packets on the logical channel that connects it to the Encryption device on the Host PC. It maintains a similar state machine as is in the ARM Encryption component which allows it to manage, precisely, the data driven encryption and decryption processes. The DSP Encryption component does not buffer the voice data that must be encrypted or decrypted but instead, in the case that the radio is in Transmit mode, sends the AMBE encoded voice packet to the Encryption device as soon as it is encoded, receives back the encrypted voice data and proceeds with forward error correction. The same applies in the case that the radio is operating in receive mode.

The next section discusses the details of the Encryption Framework implementation. Ultimately, the encryption and decryption processes are implemented in series with the AMBE encoding/decoding functions and the forward error correction encoding and decoding processes such that for each packet of voice data that is sent out for encryption or decryption, one data packet of encrypted or decrypted voice data must be received before processing can continue. The exceptions are at the beginning and end of an LDU frame where two voice data packets are sent and received.

3.3 Implementation

The Encryption Framework was implemented first for the DES algorithm only and then extended to include the AES algorithm. The framework was implemented in C and C++ and the process broken into three phases. In phase one, the encryption framework was prototyped in pseudo code as a first step in defining the state machines for the encryption and decryption process chains based on the P25 Encryption Schedule. In phase two, a standalone version of the framework containing only the essential encryption/decryption functions was developed to solidify the state machines and to verify a compliant encrypted output bitstream given a test input sequence. In the third and final phase, the encryption framework was implemented on the SDR platform incorporating hooks for data injection and

extraction to and from the relevant modules to construct valid input bitstreams and send valid output streams. These three phases are discussed in Sections 3.3.2, 3.3.3 and 3.3.4, but first a discussion on the selection of an open-source implementation of the AES and DES algorithms is presented.

3.3.1 OpenSSL Cryptography Library

As mentioned in the previous chapter, the P25 Encryption specifications provide support for the DES, TDES and AES algorithms. The Encryption Framework proposed in this thesis is focuses on the DES and AES algorithms only. Performing encryption or decryption of data using these algorithms requires access to a verified implementation of the algorithms. For this reason, the decision to use an open-source implementation of the encryption algorithms was made instead of developing a new implementation from specifications.

There are several open-source cryptography libraries, written in C and C++, that exist today but only a small number of seemingly popular libraries were considered. They include Crypto++, libgcrypt, libmcrypt and OpenSSL. A few considerations are taken into account in choosing a cryptography library. These considerations include the speed of the implementation, resistance to well-known cryptographic attacks, the adoption level of the library and community support for the library (bug fixes, security fixes, etc).

The OpenSSL cryptography library was chosen over the others as it appeared to perform better than the other libraries considered. The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source implementation of the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. As a security minded toolkit it also provides a full-strength general purpose cryptography library as well. The project is maintained by an active worldwide community of volunteers and is one of the most widely used cryptography libraries. The OpenSSL library does not implement as many algorithms as some of the other libraries considered but the algorithms it does implement, which include DES and AES, have been optimized heavily for speed. The library also has lower level functions that give the user complete con-

trol over the cryptography process and allow access to some of the state variables of the AES and DES encryption algorithm implementation.

The OpenSSL cryptography library plugs into the core of the Encryption device. Two functions from the library are used for the cryptography tasks. The `EVP_Encrypt_Update()` function is used for AES encryption and decryption, in OFB mode, while the `DES_ofb64_encrypt()` function is used for DES encryption and decryption, also in OFB mode. These functions allow the appropriate block size of data to be passed to them along with an output buffer in which the encrypted or decrypted data is returned. The setup and initialization of the library is discussed within the next few sections along with the implementation stages of the Encryption Framework.

3.3.2 Phase One: Pseudo Code

The first step toward the realization of the Encryption Framework was to develop a state machine for the DES and AES algorithms according to the P25 encryption schedule. The following is C pseudocode written as a model of the state machine for the DES algorithm in the Encryption Framework. The DES algorithm uses a blocksize of $n = 64$ bits or 8 bytes.

```
load_encryption_key(keyID);
state = STATE_INITIAL;

while (in_encryption_mode) {
    switch (state) {
        case STATE_INITIAL:
            if (in_receive_mode) {
                recv_public_mi(public_mi,8);
            }
            init_lfsr(public_mi);
            output_word_count = 1;
```

```
        // run DES_OFB once.
        // increment output_word_count
        xcrypt(public_mi,private_mi);
        state = STATE_PROCESSING;
        break;
    case STATE_PROCESSING:
        switch (output_word_count) {
            case 2:
                memset(buf,0,3);
                recvLC(buf+3,5);
                xcrypt(buf,output);
                sendLC(output+3,5);
                break;
            case 3:
                recvLC(buf,3);
                recvVCW(buf+3,5);
                xcrypt(buf,output);
                sendLC(output,3);
                sendVCW(output+3,5);
                break;
            case 14:
                recvVCW(buf,3);
                recvLSD(buf+3,2);
                recvVCW(buf+5,3);
                xcrypt(buf,output);
                sendVCW(output,3);
                sendLSD(output,2);
                sendVCW(output,3);

                if (in_transmit_mode) {
                    new_public_mi = generate_MI();
```

```
                    sendMI(new_public_mi);
                }
                break;
            case 27:
                recvLSD(buf,2);
                recvVCW(buf,6);
                xcrypt(buf,output);
                sendLSD(output,2);
                sendVCW(output,6);
                break;
            case 28:
                recvVCW(buf,5);
                memset(buf+5,0,3);
                xcrypt(buf,output);
                sendVCW(output,5);
                state = STATE_IV;
                break;
            default: // all other cases
                recvVCW(buf,8);
                xcrypt(buf,output);
                sendVCW(output,8);
                break;
        }
        break;
    case STATE_DONE:
    default:
        state = STATE_INITIAL;
        break;
  }
}
```

The state machines for both encryption and decryption are identical except for subtle differences in `STATE_INITIAL` and `STATE_PROCESSING case 14`. In encryption mode, a public MI is generated and sent out at the beginning of the encryption cycle and at the end of each LDU1 frame. In decryption mode, a public MI is received at the beginning of every superframe and used to decode the entire superframe. The public MI generated at the end of each LDU1 frame is not used for encryption until the beginning of the next superframe where it is used as the Initialization Vector (See Figure 3.1).

The `STATE_PROCESSING` state is made up of several substates which, mostly, represent special cases in the P25 encryption schedule where the cryptography input blocks are formed using bits from multiple input fields. The AES state machine, although similar in structure to the DES state machine, differs in the number of substates it requires in the `STATE_PROCESSING` state because its blocksize of $n = 128$ bits is twice as large as the DES blocksize of $n = 64$ bits and its input and output block boundaries are different from those for the DES algorithm. In each substate of the `STATE_PROCESSING` state, the correct input block is constructed by pulling the appropriate bits of VCs, LC or LSD from the respective components according to the P25 encryption schedule.

3.3.3 Phase Two: As Standalone executable

The pseudocode shown in the previous section defines, generally, the state machine of the Encryption Framework. In phase two, the same pseudocode is transformed into a standalone executable version of the Encryption Framework implemented with all the functions necessary for the cryptography process. The standalone executable allowed for easy validation of the state machine and verification of a compliant output bitstream given the right input bitstream. Again the standalone implementation was developed first for the DES algorithm and later extended to include the AES algorithm. For access to the entire source code for the combined DES and AES standalone implementation please see Appendix A.1.

The implementation features four communications based functions named `send_crypto_pktizer()`, `recv_pktizer_crypto()`, `send_crypto_vocoder()` and

`recv_vocoder_crypto()` which are used as interfaces to read and write data from the input and output FIFO buffers. A structure, `P25_ENC_SCHED_STATE`, was introduced in this implementation to track the internal state machines of the OpenSSL DES and AES encryption functions. It is initialized using in the function `init_eState()` which is also where the OpenSSL DES and AES libraries are initialized.

Two functions, `init_lfsr()` and `lfsr_new_mi()` manage the LFSR functionality, initializing the LFSR and producing new MIs respectively. The Encryption Framework state machine is contained within the `cryptography_process()` function and integrates both DES and AES substate machines. This implementation also includes the `simulate_data()` function which populates the input FIFOs with test data for encryption, the `reverse_data()` function which transfers data from the output FIFOs into the input FIFOs for decryption and the `dump()` function which prints the contents of a given FIFO buffer. These functions allow enable testing and debugging of the Encryption Framework.

The standalone implementation facilitated modularizing parts of the Framework such as the LFSR MI generator and the MI Expander used to expand a DES 64 bit MI to a 128 bit MI for the AES algorithm as specified in [9]. The modularized parts were easily tested individually for required functionality and together as a Framework and helped simplify the task of porting this implementation to the SDR platform, which was the end goal. The implementation was tested using a test encryption key, MI and input bitstream provided in the P25 encryption standard. The following properties of the framework were verified:

1. The LFSR MI generator produced the correct sequence of MI outputs for each iteration given a particular IV.
2. The LFSR MI Expander produced the correct sequence of expanded MI outputs for each iteration give a particular IV.
3. The correct output bitstream is produced for both DES and AES algorithms given the correct input bitstream sequence. Note that this standalone implementation does not take timing into account and assumes the input data is readily available since the it is hard-coded into the respective buffers.

3.3.4 Phase Three: On SDR Platform

The third phase of the Encryption framework implementation involved developing the ARM and DSP Encryption components as well as importing the standalone framework from the previous implementation phase into the SCA environment on the SDR. This implementation phase also involved connecting together multiple components in the SCA environment and connecting some of those components to the ARM and DSP Encryption components through physical and logical channels. First the development and modifications in the OMAP environment are discussed and then those in the SCA Environment are discussed. The discussions make reference to the final implementation of the Encryption Framework which is available online. Please see Appendix A.2 for access to the relevant source code.

OMAP development and Modifications

The ARM and DSP Encryption components were developed according to the architecture laid out in Figure 3.6. The ARM Encryption component is responsible for connecting the DSP Encryption component to the Encryption device on the Host PC through a logical data channel. This component is contained within the arm_ambe module that is responsible for controlling all data flow from the Host PC (SCA components) to the DSP and vice versa. The ARM encryption component manages two dedicated data channels `CHNL_ID_CINPUT` and `CHNL_ID_COUTPUT` for input and output respectively. These channels carry voice data and control packets to and from the DSP. When the P25 radio system is in Transmit mode and encryption is requested `CHNL_ID_CINPUT` is used to receive AMBE encoded bits that will be sent to the Host PC for encryption and `CHNL_ID_COUTPUT` is used to send encrypted AMBE encoded bits back to the DSP. When the radio is in Receive mode and decryption is requested, `CHNL_ID_CINPUT` is used to receive encrypted AMBE encoded bits that will be sent to the Host PC for decryption and `CHNL_ID_COUTPUT` is used to send decrypted AMBE encoded bits back to the DSP. The ARM encryption component connects to the Host PC using a TCP/IP socket connection that is completely data driven.

The DSP Encryption component is responsible for routing, to the Encryp-

tion device, all voice data packets requiring encryption or decryption depending on the current mode of the P25 radio system. This component is contained within the dsp_ambe module, which interfaces directly with the DVSI AMBE+2^{TM} library, and is responsible for processing PCM audio data or AMBE encoded data from the Host PC (SCA components). Two DSP Link Channels, one for input and the other for output, are added to the dsp_ambe module to provide a dedicated communications link for cryptography related packets to and from the ARM Encryption component. The channels are implemented using the SIO interface of the TI DSP BIOS operating system as are the other two channels on the module that are used to transport PCM audio data to and from the other ARM components. These channels, labeled `inCryptStream` and `outCryptStream`, carry both voice data and control packets, that specify the configuration parameters for the current encryption or decryption mode and sometimes act as acknowledgment packets. The channels are linked to the `CHNL_ID_CINPUT` and `CHNL_ID_COUTPUT` channels in the ARM encryption component.

The DSP and ARM encryption components operate together to ensure the cryptography services for the vocoder. After AMBE encoding is performed, the encoded audio data is sent up to the Encryption Device using the dedicated cryptography logical channel. The encryption had to be implemented in series with the AMBE Voice coding and FEC coding functions due to the fact that on the Receiver path, the FEC decoder computes some information that must be passed to the AMBE Voice decoder for error mitigation. Without implementing the encryption in this fashion, the error mitigation information would have to either be stored while the voice packet is being decrypted and passed to the AMBE voice decode function when the voice packet returns for FEC decoding or it would have to be forwarded alongside its corresponding voice packet as it travels through the Encryption Framework.

Implementing the first of the latter two solutions would have required the maintenance of a buffer that would need to be synchronized across multiple threads. The second solution would have required the propagation of excess data, that has nothing to do with the encryption process, through the Encryption Framework.

The solution implemented was to perform the encryption process in series with the AMBE Voice coding and FEC coding functions. This means that each voice packet of 20 ms audio data is FEC decoded and sent to the Encryption device, putting the DSP in a semi-blocking mode. The vocoder does not process any further voice data but waits until the decrypted voice packet is returned from the Encryption device before continuing. This ensures that the FEC information from each FEC decode operation is used with the corresponding block of audio data. This solution appeared to be the simplest as there was no need for synchronization of buffers across multiple threads or unnecessary passing around of extra data. This solution is only necessary for the Receiver path, where FEC decoding is performed, since it is the path affected by the initial problem but for the sake of uniformity and consistency in the code, this solution was applied to the Transmitter path.

Implementing the encryption process in series with the AMBE Voice coding and the FEC coding functions, combined with the cross-VC input and output word boundaries in the P25 encryption schedule, led to a dependency in the Encryption Framework that required a certain number of voice packets to be sent to the Encryption device at certain points in the encryption or decryption process for the data to keep moving through the state machine. In some cases an output from the Encryption device can only be generated if part or all of the next VC is provided (See table 5-5 and table 5-6 in [9]). To remedy this situation, in the case of a transmission, the DSP Encryption component sends two initial VCs to the Encryption device in order to get the output stream started and then it successively receives one encrypted VC, while sending a single VC for every received VC until the end of an LDU, or 9 VCs have been processed (See Figure 3.7). At this point the encryption output block boundaries and LDU boundaries line up and the DSP receives two VCs back from the Host PC. This process is repeated until the end of the transmission.

The ARM and DSP encryption components both implement the described state machine which allows them operate in sync. The state machine is also helpful in the cleanup and shutdown process on the OMAP at the end of either an encrypted radio transmission or reception by ensuring that each component has

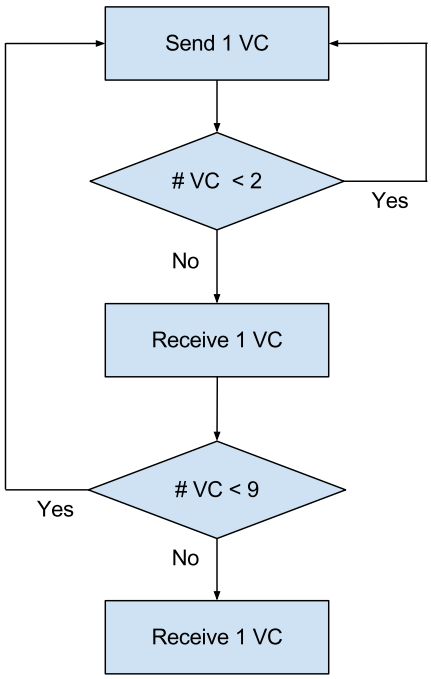


Figure 3.7: State machine in ARM and DSP Encryption components.

received all the data it expects and is not sitting in an intermediate state.

In the DSP implementation of the state machine, certain context variables of the DVSI AMBE+2^{TM} Vocoder library have to be buffered since the corresponding AMBE Voice coding and FEC coding are offset by at least 1 and at most 2 VCs. This is because both the AMBE Voice encoder and AMBE FEC decoder functions maintain some state variables which are used by the respective functions in successive calls. For each call to the AMBE Voice encoder or FEC decoder, the current context of the call is saved to be used on the successive corresponding call to FEC encoder or the AMBE Voice decoder respectively.

SCA development and Modifications

The Encryption components in the Host PC (SCA Environment) were implemented according to the framework architecture depicted in Figure 3.5. The standalone implementation from phase two was directly ported to the Encryption device in the OSSIE CF. A skeleton for the Encryption device was first created using OSSIE tools and most of the Encryption Framework functionality from the

previous phase were dropped into the device skeleton. Some new functionality was added to allow the device interoperate with other components in real time. They included adding multiple threads to manage the separate input and output FIFO buffers along with the appropriate semaphores and mutexes. New functionality also included setting up communication ports to and from the Vocoder device, Transmitter and Receiver components.

The Vocoder device was modified to incorporate functionality for the logical channel that connects the Encryption device to the DSP Encryption component. This included establishing a TCP/IP socket connection to the ARM processor and two pushPacket ports (using CORBA) to the Encryption device. A separate thread was instantiated to coordinate data transfers on the logical channel. Additional modifications were made to allow the Vocoder device compress and uncompress data from the DSP and Encryption device respectively. This was needed because the output data from the AMBE+2$^{\text{TM}}$ Library functions are represented using `short` (16 bits wide) representation but only the least significant 8 bits contain valid data and so the output data needs to be compressed from 16 bits to 8 bits in order to create a continuous stream for the Encryption device. The reverse process is performed to expand data received from the Encryption device back to 16 bits before sending to the ARM processor.

The P25 Transmitter and Receiver components were also modified to send the MI, LC and LSD to the Encryption device. Prior to adding encryption functionality, the Transmitter component assembled and FEC encoded the LC at the start of a transmission. This encoded LC was then saved for reuse with each superframe since the information contained in the LC is static for the duration of a transmission. With the addition of encryption functionality, the P25 Transmitter (and Receiver) had to be modified to encode (or decode) the LC separately for each superframe after sending the LC to the Encryption device for encryption (or decryption). Other modifications were made to the Transmitter and Receiver components to send and request the MI and LSD to and from the Encryption device respectively.

Having verified the correctness of the output bitstream of the Encryption

Framework in implementation phase two, the focus of tests in this implementation phase was verifying the correctness of timing. The intent was to verify that the correct bits of information were available to the Encryption device when they were needed. It was discovered, while testing and debugging this real time implementation of the Encryption Framework that there existed a deadlock situation between the Packetizer and the Encryption device. The deadlock situation occurred only in the states where the LSD was required to produce an output word from the cryptography process and was caused by a lack of synchronization between the Packetizer state machines and the Encryption Framework state machine. For instance when transmitting an encrypted LDU1 frame, the Packetizer requests 9 VCs from the Vocoder device and blocks waiting for the request to get filled. The vocoder device starts sending PCM audio data to the DSP for AMBE encoding. The encoded voice data is then sent back to the Encryption device for encryption. When the process gets to the 14th input block to the encryption algorithm, the P25 encryption schedule requires that the input block be constructed using bits from the 8th VC, 9th VC and LSD. The problem is that for the Packetizer to send the LSD bits to the Encryption device, it needs to receive the 9 VCs it initially requested so it can continue on to send the LSD but this cannot happen since the Encryption device needs the LSD in order to produce the 8th and 9th VC. This results in a deadlock situation.

One solution that could have been implemented in an attempt to resolve this issue would have been sending the LSD, to the Encryption device, along with the LC at the beginning of the LDU frame, since it is independent of the VCs, and at the same time, modifying the state machines in the ARM and DSP to send the 8th and 9th VCs together at the end of an LDU and receive the 7th, 8th and 9th VCs together from the Encryption device. The second, and implemented, solution was to send the LSD as suggested in the first proposed solution, and then take advantage of the fact that the DES-OFB and AES-OFB algorithms use an XOR, a linear operation, to transform the plaintext into ciphertext. This fact allows encryption, decryption to be performed on blocks of plaintext that are partially complete providing flexibility in the state machine of the Encryption Framework.

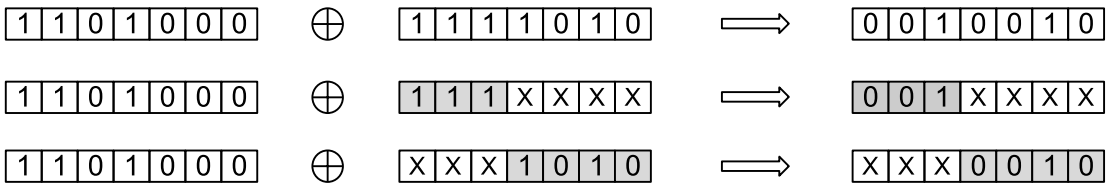


Figure 3.8: OFB Mode XOR operation workaround

With this solution, when the state machine gets to the 14th input block, it has both the 8th VC and the LSD but even though it doesn't have the 9th VC, it can encrypt the parts it has by padding the input plaintext block with zeros and ignoring that portion of the output ciphertext block (See Figure 3.8). The state of the encryption algorithm is saved before the encryption is carried out and the encrypted bits are sent off to their respective components. When the 9th VC becomes available, the previous encryption state is loaded and the bits from the 9th VC that are required for the 14th output block are encrypted in the same way, using zero padding for the other bits.

This solution meant that extending the Encryption Framework DES-OFB state machine to incorporate the AES-OFB algorithm was only a matter of performing the proper bit manipulations at the right locations in the P25 encryption schedule. Encryption input blocks that spanned multiple VCs could be constructed using this method without having to change the DSP or AES state machines. This also means that incorporating the TDES-OFB algorithm, or any algorithm that performs only linear operations on the input plaintext to generate the output ciphertext, could be done quite easily without modifying the state machines of the ARM and DSP. However, it is necessary to note that the penalty for using this solution could be high depending on how many fragments the input plaintext needs to be broken into to, as this influences how many times the cryptography function will be called. In our case, the penalty for performing the cryptography function twice instead of once is not great as timing constraints for the P25 waveform are still met. This penalty could be drastically reduced by storing the output of the OFB mode algorithm (before the XOR operation) and then performing the XOR operations as many times as required.

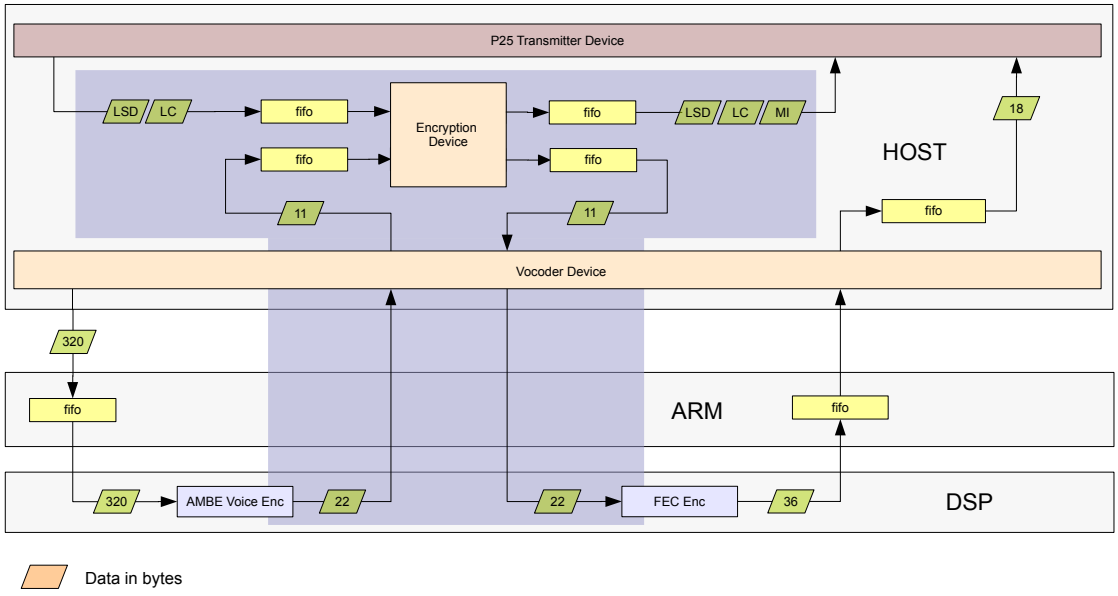


Figure 3.9: Data path of Encrypted Voice Transmission

Figure 3.9 shows the path of voice data packets through the Encryption Framework during the transmission of an encrypted message. This data path is identical to the data path during the reception of an encrypted message. In the case of transmission, the Encryption device is activated by the P25 Transmitter component. On activation, it initializes the LFSR with a previously saved IV (Public MI). Note that in the current implementation the IV is hard-coded but could easily be randomly generated for added security. The LFSR goes through one iteration to produce the Private MI that will be used in the actual encryption algorithm. The Encryption device also requires activation from the Vocoder device and waits to be activated before it begins processing data from the respective sources. Once activated by the Vocoder device, it requests LC/LSD data from the Transmitter component and VCs from the Vocoder device.

The Vocoder device, after being activated by the P25 Transmitter, first sends a control packet down to the ARM informing it of the encryption mode. The ARM forwards the same instruction to the DSP. An acknowledgment is received from the DSP, and propagated up to the Vocoder device, audio packets are streamed down to the DSP through the ARM. A FIFO in the ARM allows for buffering of a number of PCM data packets. The PCM data packets contain 160

16-bit audio samples (320 bytes). The buffered PCM data is fed to the DSP where it is AMBE encoded. The output of the encoding is 88 bits (11 bytes) with each byte represented as a `short` which gives a total of 22 bytes. The 22 bytes are channeled through the ARM back to the Vocoder device where they are compressed from `short` representations to `char` (single byte) representations and forwarded to the Encryption device. At the same time, the P25 Transmitter sends the LC and LSD information bits to the Encryption device where they are assembled into plaintext input blocks according to the P25 encryption schedule. The Encryption device loads the appropriate encryption key for this transmission as instructed by the user through the provision of a Key ID in the GUI. In receive mode, the Key ID is extracted from the HDU or LDU2 and is used to load the appropriate key for decryption.

After encryption, the output from the encryption algorithm is disassembled and the constituent bits are sent to their respective source components. Public MIs are generated and sent to the Transmitter component at the end of a processing round for one LDU worth of VCs. These Public MIs are encoded in the unencrypted ES information. A Public MI is also sent to the Transmitter component at the beginning of a transmission and is encoded in the HDU. The encrypted VCs are expanded from `char` representation to `short` representation and sent through the ARM to the DSP where FEC encoding occurs before the VCs are sent, again, to the Transmitter component through the Vocoder device. The highlighted area of Figure 3.9 shows the paths that are specific to Encryption Framework. These path are completely bypassed in unencrypted transmissions as the AMBE Voice Encoder sends its output directly to the FEC encoder whose output is immediately forwarded to the Transmitter component.

3.4 Integration

The integration of the Encryption Framework components was not complicated since the components were designed to have well defined interfaces with the goal of integrating them together. It was in fact quite straightforward and was

performed in parallel with phase three of the implementation. In the OSSIE CF, the use of standard pushPacket ports provided the advantage of having already defined, tested and proven communication architecture for the Encryption components. Integrating the Host PC encryption components with the OMAP encryption components was also straightforward using a standard TCP/IP socket connection. A message structure was developed to facilitate easier exchange of data and control messages across the TCP connection. Communication between the ARM and DSP encryption components was facilitated through the use of DSP Link Channels and the SIO interface of the DSP BIOS operating system. A message structure already being used for unencrypted voice transmissions was extended and used for sending and receiving control and voice data messages between the DSP and ARM processors.

3.5 Testing

Testing of the Encryption Framework was divided into two categories; functional tests and performance tests. The functional tests were designed to check that the functional requirements of the P25 Encryption Specification were met while the performance tests were designed to check that the timing constraints for the P25 waveform were met.

3.5.1 Functional Tests

Functional tests were performed at two points during the implementation and integration of the Encryption Framework components. First a functional test was performed during implementation phase two, on the standalone version, to verify that the sequence of MIs produced by the LFSR MI Generator was correct. This was performed using the P25 specified test sequences and verifying that the sequence produced by the Encryption Framework matched exactly. The LFSR MI Expander was also tested in the same way as was the final encrypted bitstream. The resulting encrypted bitstream was fed back into the Encryption Framework to simulate reception of an encrypted transmission and the output matched the orig-

inal input bits verifying that both encrypting plaintext and decrypting ciphertext using the Framework worked. The same functional tests were performed after the implementation and integration of the Encryption Framework on the open-source SDR platform. The test input bitstream were hard-coded into the different Encryption components and the results were exactly the same as in the standalone version.

A Final functional test performed was to transmit an encrypted voice message to an off-the-shelf P25 Handset and receive an encrypted voice message from the same Handset. These two tests were successful as the Handset was able to decrypt and playback the encrypted voice message from the open-source SDR platform while the SDR platform was able to decrypt and playback the encrypted voice transmission from the P25 Handset. These tests verified that the Encryption Framework does what is expected of it.

3.5.2 Performance Tests

The performance of the implementation in terms of execution time was measured to verify the conformance of the implementation to the timing constraints of the P25 waveform specifications. According to [10], a P25 system could have a maximum Receiver Unsquelch Delay of up to 460 ms (when both talk groups and encryption are used), a maximum Receiver Unsquelch Delay of up to 370 ms (when either talk groups or encryption is used, not both) and a maximum Receiver Unsquelch Delay of up to 125 ms (when neither talk groups or encryption is used). A maximum Transmitter Power and Encoder Attack Time of 100 ms is also specified [10]. Since the open-source SDR platform is not a well integrated SDR solution it is somewhat difficult to verify that it does meet these timing constraints. Instead an attempt is made to measure the average time it takes to perform AMBE Voice coding followed by cryptography and then FEC coding, in both transmit and receive modes, including traversal of the relevant communication paths. The average times provide an idea of the percentage of the timing constraint spent on this portion of the system. The average times measured for encrypted mode are also compared with the average times measured in unencrypted mode.

Table 3.2: P25 Encryption Framework Timing Performance Measurements

Radio mode	Encryption	Avg. Vocoding Time	Time Remaining
Transmit mode	AES	27.3 *ms*	72.7 *ms*
Transmit mode	DES	27.3 *ms*	72.7 *ms*
Transmit mode	None	4.5 *ms*	95.5 *ms*
Receive mode	AES	28.5 *ms*	341.5 *ms*
Receive mode	DES	26.0 *ms*	344.0 *ms*
Receive mode	None	20.9 *ms*	349.1 *ms*

The time taken to traverse the encryption path (the highlighted paths in Figure 3.9) was measured using the `gettimeofday()` function in C++. For Transmit mode, the timer was started in the Vocoder device, just before the 20 *ms* PCM audio packet from the Audio device in OSSIE is sent to through a TCP socket to the OMAP for AMBE Voice encoding and stopped when the same audio packet, as an FEC encoded VC, is received again in Vocoder device. The measurement results are shown in Table 3.2. Average Vocoding time was computed on 60 secs of audio data (3000 VCs). Time Remaining is the difference between the P25 specification time constraint for that mode of operation and the Average Vocoding time.

These results are quite crude, but give an estimate of the performance of the encryption framework in light of the P25 timing constraints. These measurements are dependent upon the speed of the processors involved in voice coding and encryption and is also dependent on the speed of the communication interfaces over which data must travel.

3.6 Issues and Fixes

A few issues were experienced during the implementation and integration stages of the Encryption Framework and solutions to those issues were immediately considered and implemented. In this section, a summary of the issues experienced and the fixes implemented is recapped.

The first issue encountered was the propagation of required FEC decode in-

formation to the AMBE Voice decoder function in encrypted transmission mode. A few solutions were considered including the option of appending the FEC decode information to the corresponding voice packets, which would increase traffic through the communication channels, and buffering the FEC decode information in a shared buffer, which might have resulted in thread synchronization issues. The implemented solution was to perform the decryption in series with the AMBE Voice coding and FEC coding functions. This implementation forces the voice packets to be encrypted and returned right away allowing the Vocoder to proceed without sending extra information through the Framework or dealing with synchronization issues. More details are found in Section 3.3.4.

The next issue encountered was the misalignment in timing between the order of the bits in the P25 encryption schedule and the arrival of LSD and VC bits from the respective components. One solution considered was to modify the Transmitter implementation to encode bits in a certain order and modify the state machines in the ARM and DSP to cater to this specific misalignment. This solution would be problematic as it removes the generalization from the ARM and DSP state machines that would make it easy to incorporate more encryption algorithms into the Framework. The implemented solution took advantage of the fact that an XOR operation, a linear operation, was the transforming function from plaintext to ciphertext allowing the P25 encryption bit schedule to be broken down into smaller pieces based on available bits in order not to stall the encryption process.

Chapter 4

Summary

This thesis described and discussed the design, architecture and implementation of an Encryption framework for the P25 public safety waveform on an open-source SDR platform in an OSSIE environment. The SDR platform comprises of a Host PC (GPP), a Gumstix Overo Tide with an OMAP processor (ARM/DSP) on a Tobi expansion board and a USRP front-end. The framework is designed to support the encryption algorithms standardized by the P25 specifications which are DES, TDES and AES but only implements state machines for the DES and AES algorithms.

The Encryption Framework was implemented in three incremental phases beginning with a pseudocode implementation phase, going to a stand alone implementation and finally implementation on the SDR platform. Each phase built upon the implementation from the previous phase. The Framework has three main components:

1. The Encryption device implemented in the OSSIE CF which is the core of the Encryption framework. It receives audio data for encryption or decryption from the Vocoder device , also in the OSSIE CF, and sends the results back to the Vocoder device. The Encryption device also receives other data to be encrypted or decrypted from the Packetizer (P25 Transmitter and Receiver also in the OSSIE CF) and performs encryption based on a schedule of bits that interleaves data from the Packetizer with those from the Vocoder device.

2. The ARM encryption component implemented on the ARM processor in the OMAP. It is the ARM side of the logical channel that connects the AMBE Voice encoder and FEC decoder to the Encryption device in the OSSIE CF. Its primary duty is to monitor the input channels from the Encryption device and the DSP encryption component and forward data from one component to the other.
3. The DSP encryption component implemented on the DSP processor in the OMAP. it runs along side the AMBE Voice coders and FEC coders and routes data to be encrypted or decrypted through the ARM encryption component to the Encryption device. This component maintains a state machine that tracks its current location on in the P25 encryption schedule allowing it to send and receive the right number of voice packets at each state.

The Encryption Framework was tested for functionality and performance and was found to function correctly and perform within the constraints of the P25 standard. Various conclusions were drawn up after the completion of implementation, integration and testing, namely:

- A functional P25 Encryption Framework supporting current P25 encryption standards was successfully implemented with the capability of supporting other similar encryption algorithms.

- A simple and inexpensive open-source SDR platform could be used for rapidly prototyping System components such as the P25 Encryption Framework without loss of functionality.

4.1 Future Work

Based on the work done in this thesis and the conclusions from the work, several recommendations for future work are presented.

4.1.1 Encryption Framework Enhancements

The P25 encryption standard specifies that the transmitter and receiver LFSR are to be synced using the Public MI. It does not specify what should be done in the case that the internally generated Public MI differs from the received Public MI. There is an opportunity here to enhance the robustness of the encryption framework by implementing various schemes to determine and mitigate error situations that stem from bit errors in the received bit stream. For example, one of such schemes could be storing the last three received Public MIs, computing the next two Public MIs based on the first stored Public MI and comparing the results with the stored Public MIs. This detects which Public sequence is wrong since the likelihood of bit errors in the received Public MI producing correct successive Public MI sequences is very low. With such a scheme, the Framework can decide to discard the received Public MIs for a number of frames based on the Public MI sequence that was found to be wrong.

4.1.2 Alternate SDR Architectures

The Encryption Framework could also be ported to alternate SDR architectures as a test of its portability. Some alternate architectures implementations are proposed below.

Implementing vocoder on E100

The E100 is an Ettus USRP much like the N210 used in this thesis except it incorporates the Gumstix Overo Tide and Tobi expansion boards into an integrated SDR solution. This eliminates the need for a Host PC as the ARM on the OMAP fills this role. With the vocoder and OSSIE CF sitting back to back, porting the Encryption Framework to this platform should not present too many challenges.

Implementing Encryption module on a separate device for increased security

There are alternate SDR architectures that could benefit from an implementation of the Encryption Framework. One such architecture is an SDR architecture which incorporates a separate physical device for encryption and physically separates encrypted data from unencrypted data. Since this is the preferred method for most government organizations requiring secure communications, porting the Encryption framework to such an architecture would be beneficial.

4.1.3 Key Management

Key management is a very important issue in the security of encryption algorithms requiring keys. The security of the data is directly dependent on how protected the encryption/decryption key is. In this thesis, the encryption key used was hard-coded into the Encryption framework since communications were only tested with one off-the-shelf P25 Handset which had the same key installed in it. One extension of this thesis would be to implement a simple Key management system allowing multiple keys to be stored in P25 radio system. The system should have unique ID's for keys, be re-programmable and, most importantly, should allow keys be assigned to different Talk Group IDs or Unit IDs.

Another extension of this work could be to implement the Over The Air Rekeying (OTAR) function as standardized in P25. OTAR allows the transfer of encryption keys via radio. This remote rekey ability means that radios don't have to be physically handled to install a new or replacement key and keys can be installed on-the-go. OTAR signaling is sent as Packet Data Units over the Common Air Interface and hence the P25 system would need to have a working implementation of the P25 Packet Data specification.

4.1.4 TripleDES and other encryption algorithms

Another outlook of the thesis is the support for other encryption options including TDES which is part of the supported algorithms by the P25 encryption

standard. The Encryption framework is designed to be easily adaptable to other encryption algorithms as long as the transformation of plaintext to ciphertext is performed using only linear operations. Algorithms that perform more complex transformations on the plaintext can still be incorporated into the Encryption framework but would require more effort.

Appendix A

Source code

A.1 Standalone Implementation of Encryption Framework

The source code listings for the standalone implementation of the Encryption Framework can be obtained at `http://mesh.calit2.net/junnytony/p25_ef/`.

A.2 P25 Implementation with Encryption on an open-source SDR platform in OSSIE environment

The source code listings for the final released implementation can be obtained from the files section of JTRS Open Information Repository at `http://gforge.calit2.net/gf/project/jtrs_open_ir/frs/`. Select Package named P25_Calit2-JTRS_v1.3_USRP-OSSIE.

Bibliography

[1] P. Kiley and T. Benedett, "Public safety interoperability with an sca military radio using the p25 waveform," in *Military Communications Conference, 2007. MILCOM 2007. IEEE*, pp. 1 – 8, oct. 2007.

[2] P. T. I. Group, "Technology benefits to p25," April 2012.

[3] J. T. R. S. Standards, *Software Communications Architecture Specification*. Joint Program Executive Office (JPEO), 33000 Nixie Way, San Diego, CA 92147-5110, February 2012.

[4] N. I. of Justice, "Voice encryption for radios," *In Short*, vol. NCJ 217103, March 2007.

[5] "Data encryption," *Encyclopaedia Britannica. Encyclopaedia Britannica Online*, vol. Encyclopaedia Britannica Inc., May 2012.

[6] S. Forum, "Cognitive radio definitions." SDRF-06-R-0011-V1.00, November 2007.

[7] D. V. S. Inc., *Software Interface Specification*. DVSI APCO Dual-Rate AMBE+2 Vocoder, 234 Littleton Road, Westford, MA 01886 USA, 00 ed., September 2009.

[8] Z. Cao, P. Johansson, W. Hodgkiss, W. Zhao, A. Nwokafor, J. Cuenco, and B. Hobson, "Design and rapid prototyping of sca-compliant public safety p25 waveform and p25—fm3tr—voip bridge," *Analog Integr. Circuits Signal Process.*, vol. 69, pp. 245–257, Dec. 2011.

[9] T. I. A. E. I. A. Standard, *Project 25 Block Encryption Protocol, TIA/EIA-102.AAAD*. Telecommunications Industry Association, 2500 Wilson Boulevard, Arlington, VA 22201 U.S.A., July 2002.

[10] T. I. A. E. I. A. Standard, *Land Mobile Radio Transceiver Performance Recommendations - Project 25 Digital Radio Technology, C4FM/CQPSK Modulation, TIA-102.CAAB-A*. Telecommunications Industry Association, 2500 Wilson Boulevard, Arlington, VA 22201 U.S.A., September 2002.