Wireless Communication Demonstration in Hardware Using an Exactly Solvable Chaotic System

by

D. Aaron Whitney

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 4, 2019

Keywords: chaos, chaotic oscillator, communication, wireless, matched filter, mixed-signal processing

Approved by

Robert N. Dean, Chair, McWane Professor of Electrical and Computer Engineering
Victor P. Nelson, Professor of Electrical and Computer Engineering
Thaddeus Roppel, Associate Professor of Electrical and Computer Engineering

Abstract

This work presents a hardware demonstration of a mixed-signal wireless communication system that utilizes a chaotic oscillator based on an exactly solvable piecewise linear set of differential equations, along with a matched filter derived from its exact analytical solution. An analog temperature sensor serves as the input for the system. The analog output from the sensor is converted to an 8-bit value via a microcontroller; this value is then encoded into an analog chaotic waveform via a linear controller, and data is transmitted serially over a wireless transmitter and receiver at 2.3 GHz. A matched filter defined in software extracts the binary data from the received analog signal, and a second microcontroller samples this binary data and sends it to a computer for verification. Tests show that the system is able to accurately transmit and receive the sensor data in the intended manner. Included in this work is relevant background information and theory for the system, a description of the design, function, and implementation of each component in the system, hardware test results and verification of the system's intended function, and the results of multiple bit-error-rate (BER) tests in varying ambient conditions.

Table of Contents

List of Figures

Chapter 1

Introduction

Initially, the area of chaos was primarily studied by mathematicians and physicists in order to describe or model physical, chemical, or naturally occurring phenomena [1, 2, 3]. This motivation has now shifted towards taking advantage of the inherent properties found in chaos for various applications, such as communication systems, radar, random number generation (RNG), and noise signal generation. Some of the advantageous properties include continuous power spectral density for communication, radar, and noise signal generation. In particular, communication systems can utilize the spread spectrum properties in order to minimize the detectability of the signal. This is because the transmitted power is spread out over a large range of frequencies, which gives the illusion of an increase in the noise floor. There is a large amount of theory involved in taking advantage of chaotic dynamics; however, there is room for applying chaos theory to real-world systems. Utilizing chaos theory in electronic circuitry allows for the realization of complex waveforms and functionality with minimal electronic circuitry, potentially reducing size, weight, and cost, while enhancing reliability.

A wireless communication system based on an exactly solvable chaotic equation has been demonstrated. The system consists of a data input from a temperature sensor, a chaos oscillator controller, an exactly solvable chaotic oscillator, an AM wireless communication system, a matched filter, and a data output section. Accurate transmission and reception of the sensor data is verified via serial buses from ST microcontrollers on both ends of the system.

The exactly solvable chaotic oscillator has a fundamental frequency of approximately 18.4 kHz. It produces a baseband chaotic signal using a single-transistor sinusoidal oscillator circuit where the signum function-based nonlinearity is generated using operational amplifiers (op amps), comparators, and digital logic devices. The oscillator is controlled into two distinct orbits, representing 1s and 0s, using proportional feedback control. This type of controller compares the measured waveform with a desired waveform and applies a voltage pulse that is proportional to the magnitude of the difference between these two waveforms. This voltage pulse is then applied at regular intervals to the chaotic waveform in order to steer the trajectory to the desired orbit.

A standard frequency modulated (FM) transmitter up-converts the chaotic modulated signal onto a 2.3 GHz carrier for wireless transmission to a receiver that down converts it back to baseband. For the purposes of simplicity and reliability, as well as to maintain focus of work on the novel portions of the system, an off-the-shelf transmitter and receiver were used in the final iteration.

A matched filter for the exactly solvable system was previously developed. The matched filter was developed utilizing the exact analytical solution of the chaotic waveform, which is written as a linear convolution of a fixed basis function. It was shown that the matched filter could be written as a delay differential equation. The electronic matched filter was realized using a difference amplifier and an analog integrator, and utilizes all-pass filters to generate the necessary delay circuit that recovers the information from the received signal. The matched filter was also realized in software using an ST microcontroller. The matched filter's output waveform was sampled by a second ST microcontroller that communicates over a serial bus to recover the encoded information.

Chaotic oscillators have a wide range of possible applications, including random number generation [4], communication systems [5, 6], ranging for vehicle collision detection [7], and noise signal generation [8]. Some distinct characteristics of chaotic systems include topological mixing, determinism, long-term aperiodic behavior, sensitivity to initial conditions, as well as a spread spectrum response. The theoretical uniform power density of a chaotic system is one of the key characteristics that could be taken advantage of in their designs.

A majority of these chaotic systems are defined by a an ideal set of differential equations. One of the problems with implementing these in electronics is having to account for the non-ideal properties, such as temperature dependencies and limited bandwidth, of the electrical components. In addition, many of these systems are typically based on a set of higher order nonlinear dynamical equations. These systems often lack an exact analytical solution, limiting their applications in communication systems. To improve the performance of these systems in the presence of additive white Gaussian noise (AWGN), a matched filter is often used. This requires an exactly solvable solution to develop. However, there are some lower order linear systems that exhibit chaotic behavior that have been developed. An example of this can be seen in the piecewise linear system developed [10]. This system is of particular interest, due to the fact that an exact analytical solution has already been developed [12, 11].

This system is defined by a linear second-order set of differential equations with discrete states that provide a third dimension of freedom. This chaotic system has been used in the lower audio frequency range (approximately 84 Hz) for vehicle ranging and detection applications [20, 21]. It has been shown that a relatively simple matched filter can be derived and implemented at this low frequency making the system suitable for communication applications [22]. However, the low operating frequency of this design limits its practical use in a communication system that typically operate in the RF range. For this reason, the frequency of the oscillator design needs to be increased.

The low frequency oscillator featured analog and discrete components implemented on a non-permanent prototype board. One of the key components was a negative resistance-inductor-capacitor (RLC) resonance circuit realized using a negative impedance converter (NIC). The limited bandwidth of the NIC and prototype layout board proved to be one of the limiting factors in scaling the frequency. An alternative approach to the NIC has been developed with an emphasis on the hardware implementation.

Presented is a mixed-signal electronic implementation of an exactly solvable chaotic oscillator. The design is based on a single transistor in a common-base amplifier configuration combined with an parallel inductor and capacitor (LC) resonance tank circuit and a mixed-signal feedback network. The oscillator features a simple topology that is implemented using commercial-off-the-shelf (COTS) parts. This approach is intended to increase the operating frequency of the oscillator through careful board design. The intended application for this design is for it to be used in a communication system. Included is a circuit based model of the system and simulation results.

This approach reduces cost by replacing the NIC, which requires a high bandwidth operational amplifier, with a single transistor circuit. This design takes careful consideration of the layout of the oscillator to minimize trace lengths and to reduce the overall footprint of the components.

Chapter 2

Background

2.1 Beginnings of Chaos Theory

The discovery of the conditions in which random behavior occurs is generally attributed to H. Poincaré and his study of the three-body problem using Newtonian assumptions. Poincaré noted that the trajectories of the bodies was dependent on initial conditions, and for certain initial conditions, the behavior of the system was difficult to predict. Numerous studies on the three-body problem were performed [29], [30]; the prevailing consensus for some time was that the unpredictable behavior was due to noise and/or measurement error.

2.1.1 The Lorenz System

An early mathematical representation for chaotic behavior in nature was that of a model of atmospheric convection developed by E. Lorenz. [24] This model showed that a fluid placed within a box that is heated in a uniform manner from the bottom and cooled from the top is a nonlinear and deterministic system, as modeled by Lorenz's simplification of the three-dimensional Navier-Stokes equations,

$$dx/dt = \sigma(y - x) \tag{2.1}$$

$$dy/dt = -x(\rho - z) - y \tag{2.2}$$

$$dz/dt = xy - \beta z \tag{2.3}$$

where σ corresponds to the Prandtl number, ρ corresponds to the Rayleigh number, and β corresponds to a physical dimension of the system. $x(t)$ represents the rotational speed of the

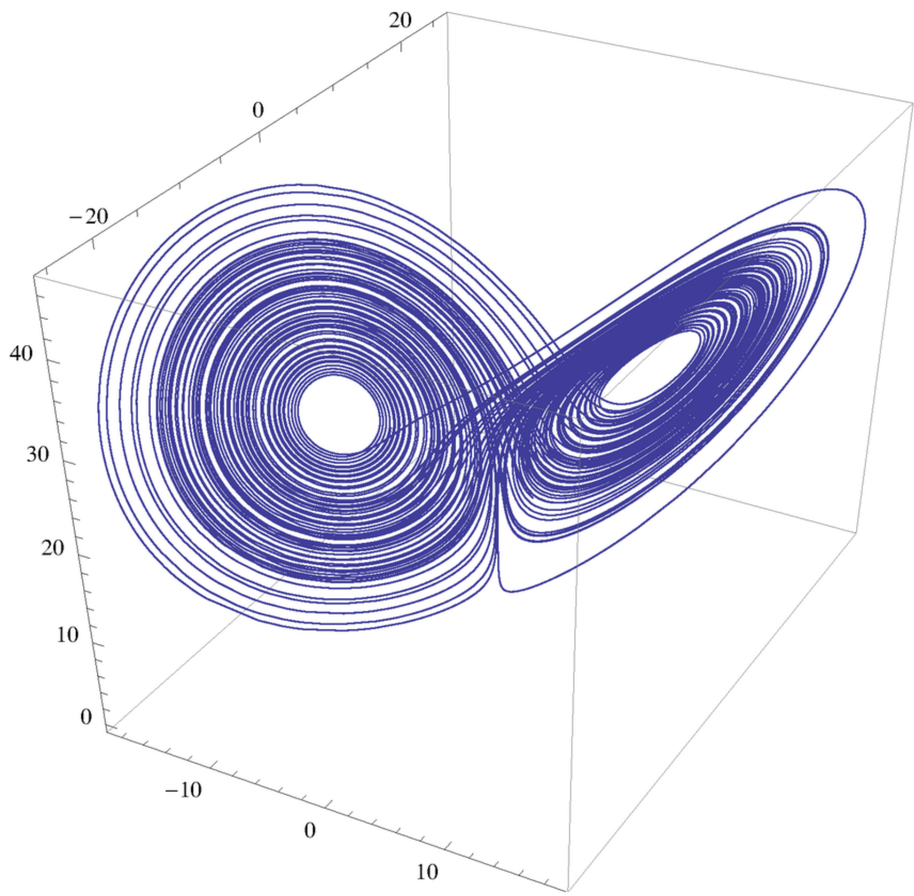

Figure 2.1: Phase-space representation of the Lorenz system exhibiting chaotic behavior. Note the two orbits, giving the system the appearance of a butterfly. Source: Adapted from [27]

system, and $y(t)$ and $z(t)$ represent the distribution of temperature. The Rayleigh number is of particular interest in this model; it represents the amount of turbulence present in the system dynamics. At high values of p, the convection in the system becomes unstable. [28] Lorenz showed that the system exhibited chaotic behavior when $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$. The trajectory of this system is highly sensitive to initial conditions, though nearly all initial conditions will tend to a strange attractor trajectory known as the "Lorenz attractor". A three-dimensional phase-space representation of this attractor is shown in Fig. 2.1.

The sensitivity of this system to initial conditions led to the acceptance of the "Butterfly Effect" in which a small change in initial conditions can greatly vary the behavior of a large system. Lorenz hypothesized that further understanding of this system could lead to improved understanding and prediction of large-scale weather patterns.

2.1.2 Universality and The Logistic Map

Though chaos had been mathematically modeled, the manner in which it emerges had yet to be theorized. An early demonstration of the conditions in which chaotic behavior arises was shown by M. Feigenbaum. [31] Feigenbaum showed that chaotic behavior exists throughout nature and natural phenomena, specifically citing a population of organisms with a static birth rate. To demonstrate this, a model for a dilute population of organisms was modeled as:

$$p_{n+1} = bp_n \tag{2.4}$$

where p_{n+1} is the population value dependent on the previous population value p_n multiplied by a constant birthrate b. Eq. 2.4 accurately describes the growth of the organism population with the solution $p_n = p_0 b^n$ so long as there is no mutual interference or competition and the environment is fixed. When these conditions inevitably break down, the population is determined by a varying growth rate, shown in Eq. 2.5:

$$p_{n+1} = b_{eff} p_n \tag{2.5}$$

where $b_{eff} < b$, and b_{eff} is a function of p. One can infer that given a limited amount of resources, $b_{eff} \cong 0$ when the population is sufficiently large. If p_n is defined as $(b/a)x_n$, where a is some scaling factor, then the equation can be written in the general form of a logistic map:

$$x_{n+1} = bx_n(1 - x_n) \tag{2.6}$$

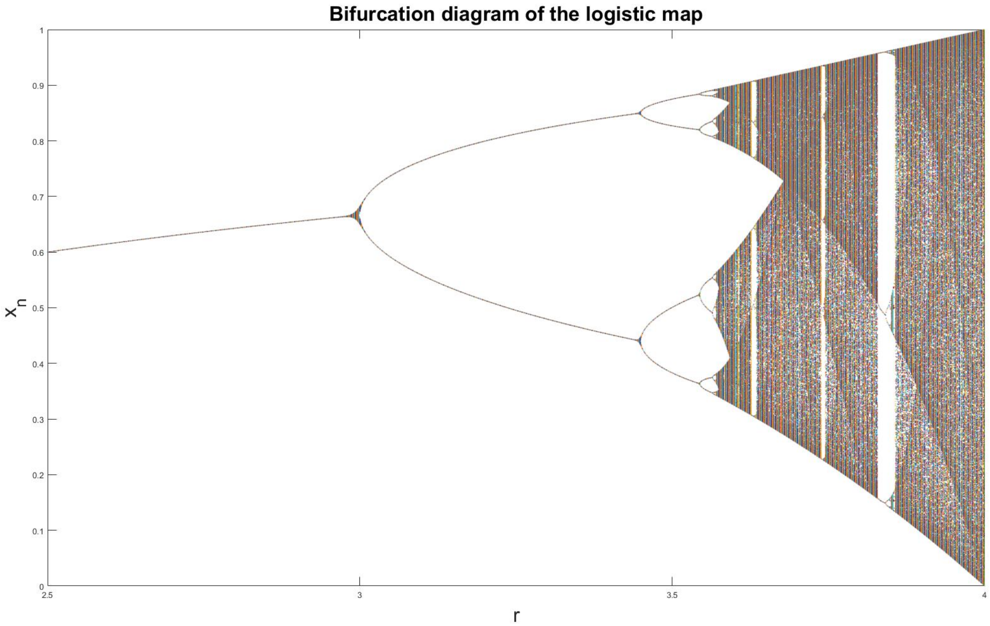

Figure 2.2: Bifurcation diagram showing the value of a population given its growth rate. A single cycle exists until $r \approx 3.0$. The single cycle becomes unstable and is replaced by a period-doubled cycle. The pattern of instability being replaced by period doubling continues until $r \approx 3.6$ when the system tends to an infinite number of values. Source: Adapted from [32]

By varying the growth rate, the population value can become unstable. Typically, new population values emerge as previous cycles become unstable, causing the system to oscillate between these new paths, until the system tends to an infinite number of values. This phenomena is shown as a bifurcation diagram in Fig. 2.2. Fig. 2.3 shows that the seemingly random oscillations can be brought back in to order at certain growth rates, and then quickly bifurcate back to chaotic behavior. Feigenbaum discovered that any system which exhibits this period-doubling path to chaotic behavior also exhibits the following characteristic – the distance between bifurcations asymptotically approaches the number 4.669. [32] Additionally, the strange attractors of these systems are fractals: each new bifurcation is a scaled duplicate of the original bifurcation.

These models, in addition to Lorenz's model, also display a characteristic that is often taken advantage of in its applications – sensitivity to initial conditions. Fig. 2.4 shows a

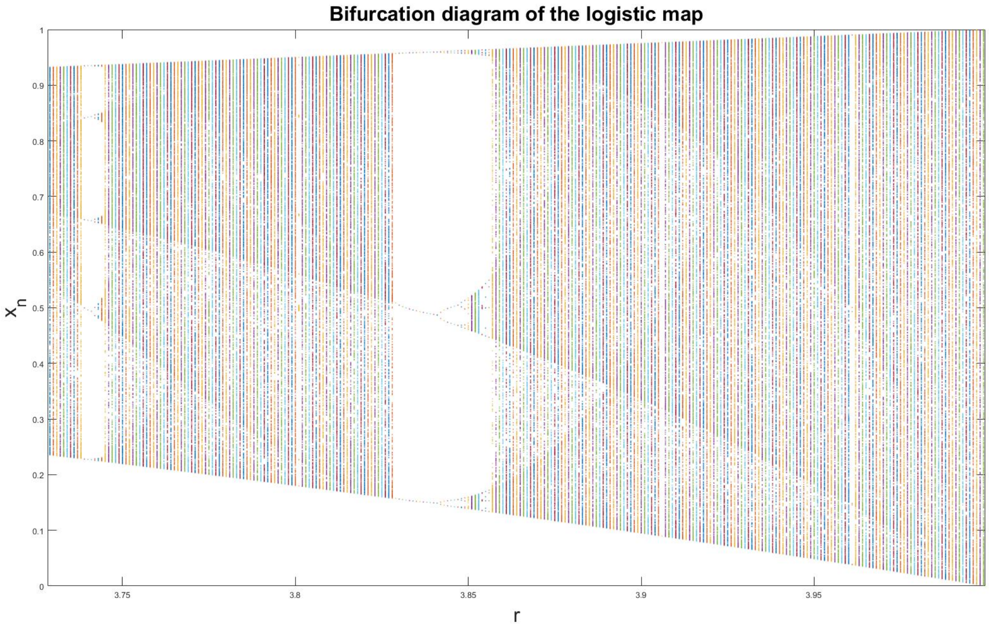

Figure 2.3: Zoomed view of bifurcation diagram. Notice the sudden transition from chaotic behavior to period-5 at $r \approx 3.74$, and the transition to period-3 at $r \approx 3.83$. It has been shown that any system which exhibits period-3 at any particular value is capable of chaotic behavior at other values. [33] Note the bifurcations after period-3, which are scaled duplicates of the initial bifurcation. Source: Adapted from [32]

plot of the values of a population for a given generation, with two initial conditions that vary by a small amount. The system dynamics, including a positive Lyapunov exponent, are identical, but a minute change in initial conditions produces a divergence that, if one did not have access to the underlying system parameters, one could not infer with any certainty that the two systems were identical.

2.2 Saito and Fujita's System

Although chaotic behavior had been identified and studied to a significant degree [13, 14, 15, 16, 17, 18, 19], there still existed difficulty in performing mathematical analysis on the systems when chaotic. Saito and Fujita proposed a novel differential equation that exhibited chaotic behavior, called the manifold piecewise linear system. [10] This system was described in physical form as a resonant circuit with a negative resistance, shown in Fig. 6.

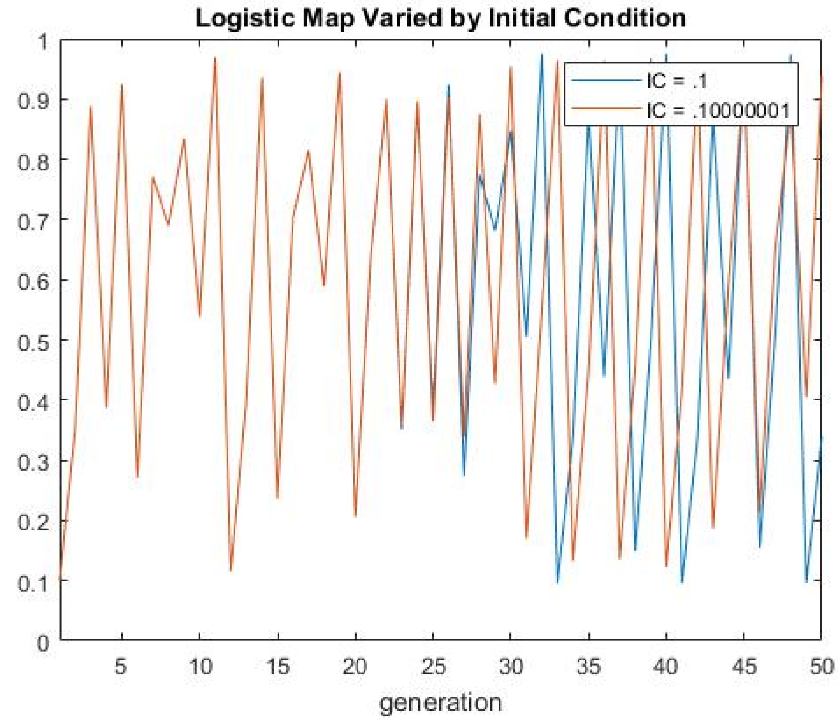

Figure 2.4: Plot of two initial conditions of a dynamic population model with a positive Lyapunov exponent. Slight variance in initial condition causes divergence at roughly the 33rd generation. Source: Adapted from [32]

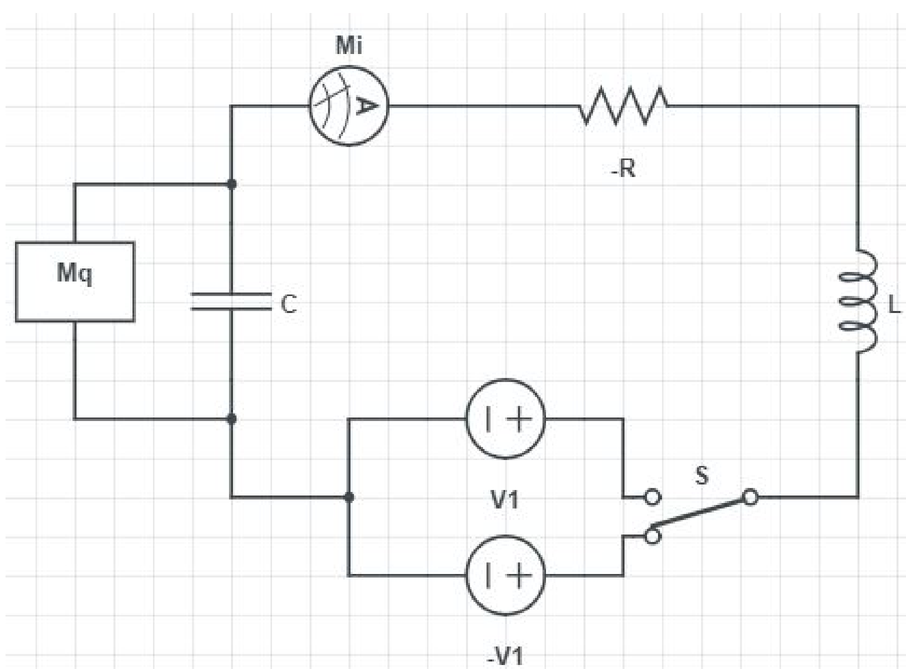

Figure 2.5: Physical representation of the manifold piecewise linear system as a resonant circuit with a negative resistance. Source: Adapted from [10]

In this circuit, $-R$ is some negative resistance/impedance, L is and inductance, and C is a capacitance. V_1 and $-V_1$ represent two identical voltage sources with opposite signs. M_q and M_i are charge and current measuring devices, respectively, and S is a switch dependent on a number of characteristics of the circuit at a given time. The general function of the circuit can be described as follows: When the switch is closed upward, the bias in the circuit is equal to V_1. When the charge measuring device detects that the charge on the capacitor $q \leq q_{th}$, where q_th is an arbitrary charge threshold, and the current measuring device detects $i = 0$, the switch is flipped and the polarity of the circuit is reversed. Once the circuit, now biased at $-V_1$ reaches the condition $i = 0$ and $q > q_{th}$, as detected by the charge and current measuring devices, the switch will return to its original position. The circuit is governed by the following parameters and equation:

$$x = \frac{q}{CV} \tag{2.7}$$

$$\tau = \frac{1}{\sqrt{LC}} t \tag{2.8}$$

$$2\delta = R\sqrt{\frac{C}{L}} \tag{2.9}$$

$$\beta = \frac{q_{th}}{CV} \tag{2.10}$$

$$\ddot{x} - 2\delta\dot{x} + x = \begin{cases} 1 \\ -1 \end{cases} \tag{2.11}$$

Fig. 2.6 shows the solution of Eq. 2.11 in phase-space representation, with $\delta = 0.11$, $\beta = 0$, and initial conditions $x(0) = 0.6$ and $\dot{x}(0) = 0$. The thick bands show the presence of chaotic behavior, and the overall space resembles Lorenz's two-spiral chaos. Saito and

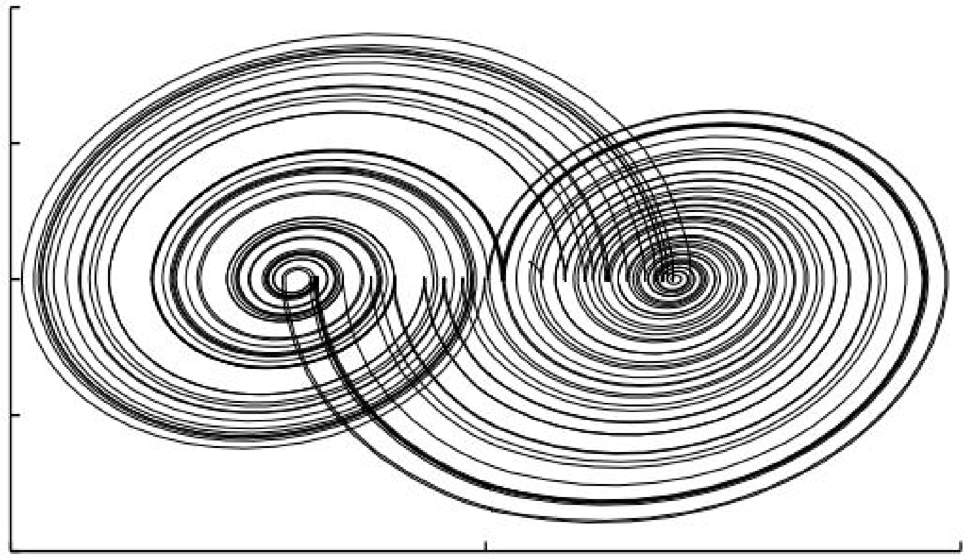

Figure 2.6: Phase-space plot of the solution to Eq. 2.11, with $\delta = 0.11$, $\beta = 0$, and all initial conditions set to 0. Source: Adapted from [10]

Fujita's system showed that multi-dimensional sets of linear differential equations were not the only path to chaotic behavior – chaos could be achieved using a single equation with piecewise components. Additionally, these equations could be represented with relatively simple circuitry, as nonlinear electrical components could handle the discrete-time caveat to these systems. This analytical and simplistic approach is the basis on which the theory for the presented communication system is derived.

2.3 The Exactly Solvable Chaotic System

Despite the long-standing assumption that the inherent nature of chaotic systems produced no exact solution, [34] a novel chaotic oscillator system was conceived that yielded an exact analytical solution. [25]. The system is based on a linear, constant-coefficient ordinary differential equation with a discrete-time forcing function

$$\ddot{x} - 2\beta\dot{x} + (\beta^2 + \omega^2)x = (\beta^2 + \omega^2)s(t) \tag{2.12}$$

where $s(t)$ is a binary waveform. For the initial conditions $x(0) = x_0$ and $\dot{x}(0) = y_0$, and the parameter values $\beta = ln(2)$ $\omega = 2\pi$. The general solution to this system takes the following form:

$$x(t) = x_b(t) + x_u(t) \tag{2.13}$$

where $x_b(t)$ is a bounded particular solution and $x_u(t)$ is a homogeneous solution. To achieve an exact solution, there must be a condition for which $x_u(t) = 0$ for any time t. Upon analysis of the system, it was found that the term β was equivalent to a positive Lyapunov exponent, indicating chaotic behavior. Additionally, this condition was shown to take the form of a shift map, and it has been shown that shift maps have the chaotic characteristics of dense orbits, sensitivity to initial conditions, and topological transitivity. [35]

A significant advantage of this derivation of chaos is that the bounded solution can be realized as a linear superposition of a basis function, and the waveform can be realized in full for any binary sequence $s(t)$. This approach to the system is defined in Eq. 2.14 and Eq. 2.15:

$$x_b(t) = \sum_{i=0}^{\infty} s_i P(t-i) \tag{2.14}$$

$$P(t) = \begin{cases} 2^{t-1}(cos\omega t - \frac{\beta}{\omega} sin\omega t), \quad t < 0 \\ 1 - 2^{t-1}(cos\omega t - \frac{\beta}{\omega} sin\omega t), \quad t = 0 \\ 0, \quad t > 0 \end{cases} \tag{2.15}$$

where $x_b(t)$ is the bounded general solution and $P(t)$ is the basis function. A plot of the basis function for the considered system is shown in Fig. 2.7. So, for any given sequence $s(t)$ an initial condition exists that yields a bounded particular solution. It stands to reason, then, that there must exist an initial condition for which a certain binary sequence yields a bounded particular solution.

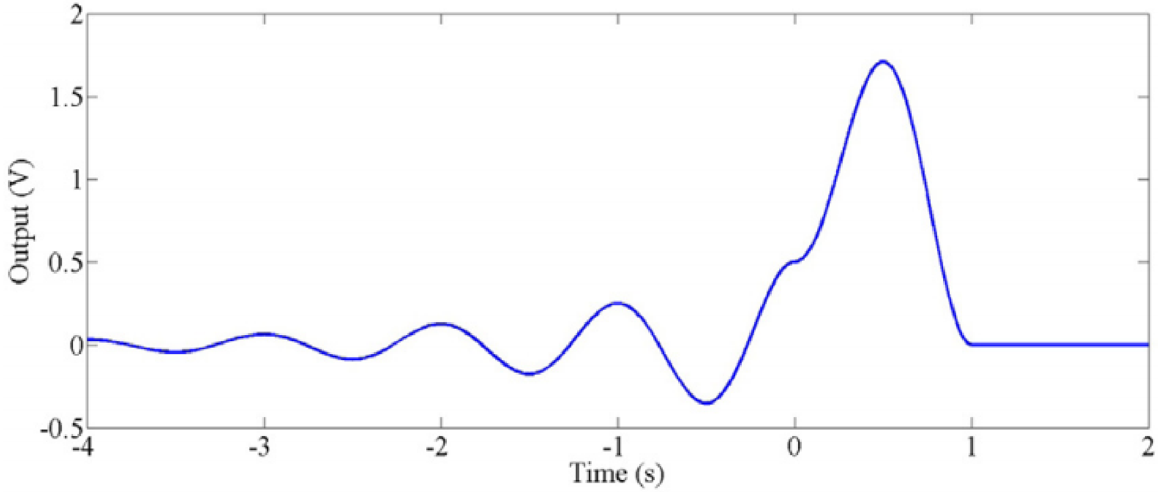

Figure 2.7: Basis function that can be superposed over a binary sequence to form a chaotic wave. Source: Adapted from [25]

2.4 The Exactly Solvable System in Communications

From this defined set of linear differential equations, a nonlinear system was considered based on the fact that the mapping of initial conditions to binary sequences constitutes a present nonlinearity in the general system. A nonlinear ordinary differential equation was considered:

$$\ddot{u} - 2\beta\dot{u} + (\beta^2 + \omega^2)u = (\beta)^2 + \omega^2)[2s(t)] \tag{2.16}$$

where $x(t)$ is the output of the system, ω is the fundamental frequency, and β corresponds to a Lyapunov exponent. In order for the system to exhibit chaotic behavior, the Lyapunov exponent must remain positive; for this system, $0 < \beta \leq ln(2)$ constitutes a positive lyapunov exponent. $s(t)$ is a nonlinear forcing function described as the piecewise function:

$$s(t) = \begin{cases} +1, & x(t) \geq 0 \\ -1, & x(t) < 0 \end{cases} \tag{2.17}$$

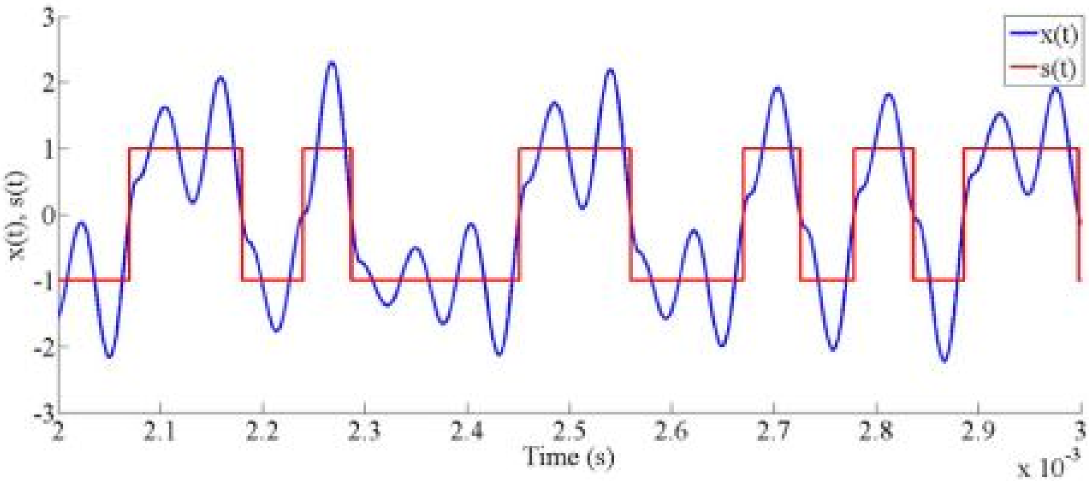

Figure 2.8: Plot of the output of the exactly solvable system. The growing oscillation occurs around one of two equilibrium points until a zero crossing of the output and its derivative, and the oscillation is forced to the equilibrium point with opposite sign. Source: Adapted from [36]

with the initial conditions defined as $s(0) = u_0$ and $\dot{u}(0) = v_0$, with $\beta = ln2$ and $\omega = 2\pi$. The piecewise portion $s(t)$ can be treated as a feedback applied to an oscillator, and can be represented practically as a signum function that constitutes the equilibrium point about which the unstable oscillation occurs. The oscillation will increase about an equilibrium point until the guard condition is reached, at which time the equilibrium point will switch, and the unstable oscillation will continue with lower energy around the new equilibrium point. The guard condition is defined as the point at which the output of the system $u(t)$ and the derivative of the output $\dot{u}(t)$ both cross zero. This means that the oscillation will grow until it reaches the energy required to maintain an amplitude of 1, and once a local maximum/minimum of oscillation is reached, the oscillation will be steered to a new equilibrium point, and the energy of the oscillation will be lost, allowing it to grow again. A graphical representation of the function of the system is shown in Fig. 2.8.

It has been demonstrated that an oscillation of this type can be controlled without knowledge of the system dynamics by using small perturbations to prevent the guard condition from forcing the oscillation to the next equilibrium point. [37] This effectively enables

$s(t)$ to be controlled externally, and this ability is the basis on which data is injected into the communication system. If $s(t)$ can be controlled, then it follows that a sequence of data could be used as the basis of operation of a control system, and the oscillations would occur around the desired equilibrium point.

2.5 A Matched Filter for the Chaotic System

In order for communication to occur, it is necessary that the binary data injected into the oscillator be separated from the oscillatory portion of the signal [40, 41]. This is achieved by inserting a matched filter on the receive side of the system, which is designed to maximize signal-to-noise ratio. [38] Chaotic systems are known to have the property of flat power-spectral density, an attribute shared with AWGN. [39] Therefore, a matched filter designed to separate the desired signal, which resembles noise, from any noise present in the environment should be extremely effective in noise-laden environments. A mathematical representation of a matched filter for this exact purpose was developed by Corron et al. [22] This derivation utilizes the fact that the chaotic system has an exact solution and can be written as a convolution of a basis function with a series of binary symbols, and the fact that the matched filter can be realized as a finite-impulse response (FIR) filter, using the time-reversed basis function as the impulse response. $P(-t)$ is defined as:

$$\ddot{P} + 2\beta\dot{P} + (\omega^2 + \beta^2)P = (\omega^2 + \beta^2)h(t) \tag{2.18}$$

where $h(t)$ is a pulse,

$$h(t) = \begin{cases} 1, & -1 \leq t < 0 \\ 0, & all\ \ other\ \ t \end{cases} \tag{2.19}$$

Differentiation of Eq. 2.19 yields a time-shifted unit impulse function combined with another unit impulse function at $t = 0$. This is a general form of the impulse response of the matched filter for the basis function, and therefore the equation of the matched filter can be described by the following:

$$\dot{\eta} = v(t+1) - v(t) \tag{2.20}$$

$$\ddot{\xi} + 2\beta\dot{xi} + (\omega^2 + \beta^2)\xi = (\omega^2 + \beta^2)\eta(t) \tag{2.21}$$

where $v(t)$ is the input to the matched filter, $\eta(t)$ is an intermediate state, and $\xi(t)$ is the output of the matched filter. The intermediate state $\eta(t)$ is defined as:

$$\eta(t) = \int v(t' + 1 - v(t')dt'. \tag{2.22}$$

The intermediate state is equivalent to the input to the filter subtracted from the input delayed by one period. The output of the intermediate stage is then compared to a threshold to recreate the symbolic data of the original waveform. This generated waveform is then fed to a resonant circuit that matches the resonant circuit present in the original system, to recreate the original chaotic waveform. This waveform is then compared to a threshold to extract the symbolic data.

Chapter 3

Development

3.1 Chaotic Oscillator Realization

3.1.1 Low-Frequency Oscillator in Electronics

Due to the simplicity of the discussed exactly solvable chaotic system, successful design and implementation in electronics can be achieved with relative ease. A low-frequency oscillator was developed by Corron et al. [12], and the design was based on the construction of an exactly solvable system with respect to a folded band map, instead of a shift map. This allowed for the process of encoding data to be executed with less difficulty. This system can also be written as a linear convolution of a basis function and symbolic data, allowing for the same derivation of a matched filter. The continuous-time portion of the oscillator is described by the following:

$$\frac{d^2u}{dt^2} - 2\beta\frac{du}{dt} + (\omega^2 + \beta^2) \cdot (u - s) = 0 \tag{3.1}$$

where $\omega = 2\pi$ and $\beta = 0$. The transitions of the discrete-time portion of the system are described as:

$$\frac{du}{dt}(t) = 0 \Rightarrow s(t) = H(u(t) - 1) \tag{3.2}$$

where H(t) is the left-continuous Heaviside function,

$$H(t) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{3.3}$$

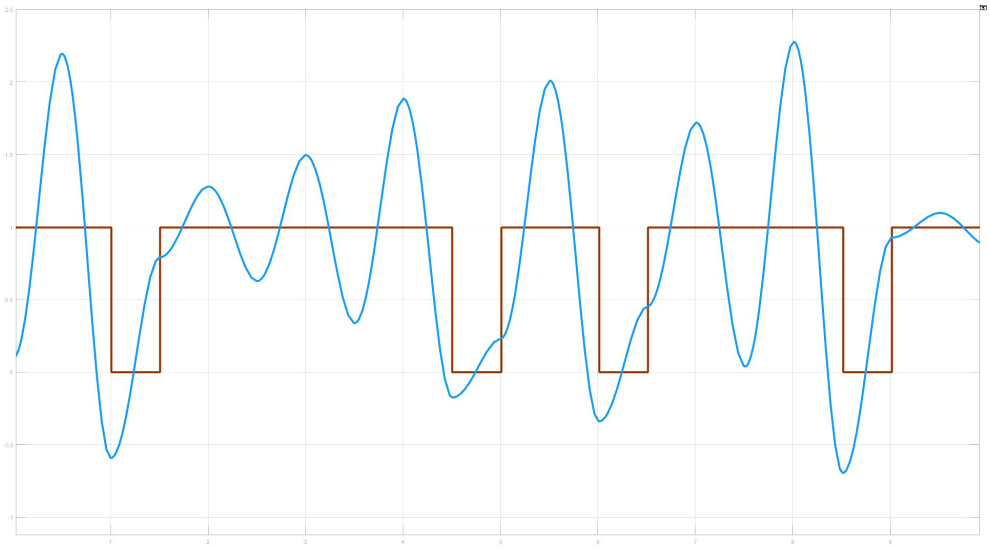

Figure 3.1: Operation of the exactly solvable folded-band oscillator. The oscillation grows until the amplitude surpasses +1, then at the local maximum of the oscillation (roughly $t = 7.5$) the equilibrium point is switched to 0 for one half period. The equilibrium point is then switched back to 1, effectively removing energy from the oscillation and allowing it to continue without becoming unbounded. Source: Adapted from [12]

.

This describes the guard condition for the system, as the binary state $s(t)$ is assigned the value of the shown Heaviside function shifted by unity when the derivative of the output of the system reaches zero. This system operates in a similar manner to the previously discussed system based on a shift map. Energy is injected into the system until the guard condition is triggered. The equilibrium point is switched, and the oscillation continues for half the oscillation period about the new equilibrium point, at which time the point switches back. This effectively removes energy from the system to a value below the guard condition, and allows it to continue functioning in a bounded manner. This operation is shown in Fig. 3.1.

The dynamics of this system can also be viewed from the perspective of a phase space, with a clockwise spiral outward about an attractor at the origin until the x-value passes one, and once the spiral crosses the y-axis (derivative equal to zero), the attractor shifts to 1, and the radius of the spiral tightens around the new attractor for half a rotation, then the attractor shifts back to the origin and the spiral "folds" back on itself and continues its

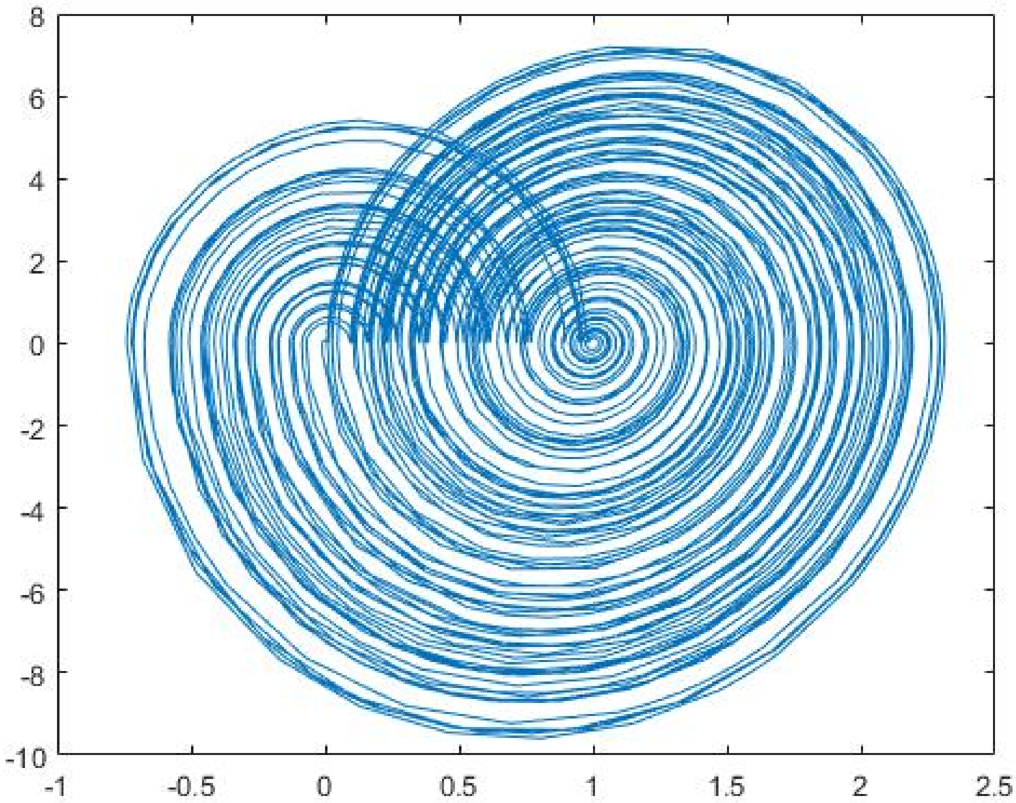

Figure 3.2: Phase-space representation of the function of the chaotic system. Source: Adapted from [12]

original clockwise spiral about the origin. The phase-space representation is shown in Fig. 3.2.

Another way to grasp the function of this system is through a Poincaré return map. The map is generated using the local maxima of the oscillator's waveform (Fig 3.1). For any t_n where the $s(t)$ remains the same, the solution

$$u(t) = s_n + (u_n - s_n)e^{\beta(t-t_n)}cos\omega(t - t_n) - \frac{\beta}{w}sin\omega(t - t_n) \tag{3.4}$$

is valid. The times at which the guard condition allows the state to change are described by

$$u(t_n + 1/2) = s_n - e^{\beta/2}(u_n - s_n). \tag{3.5}$$

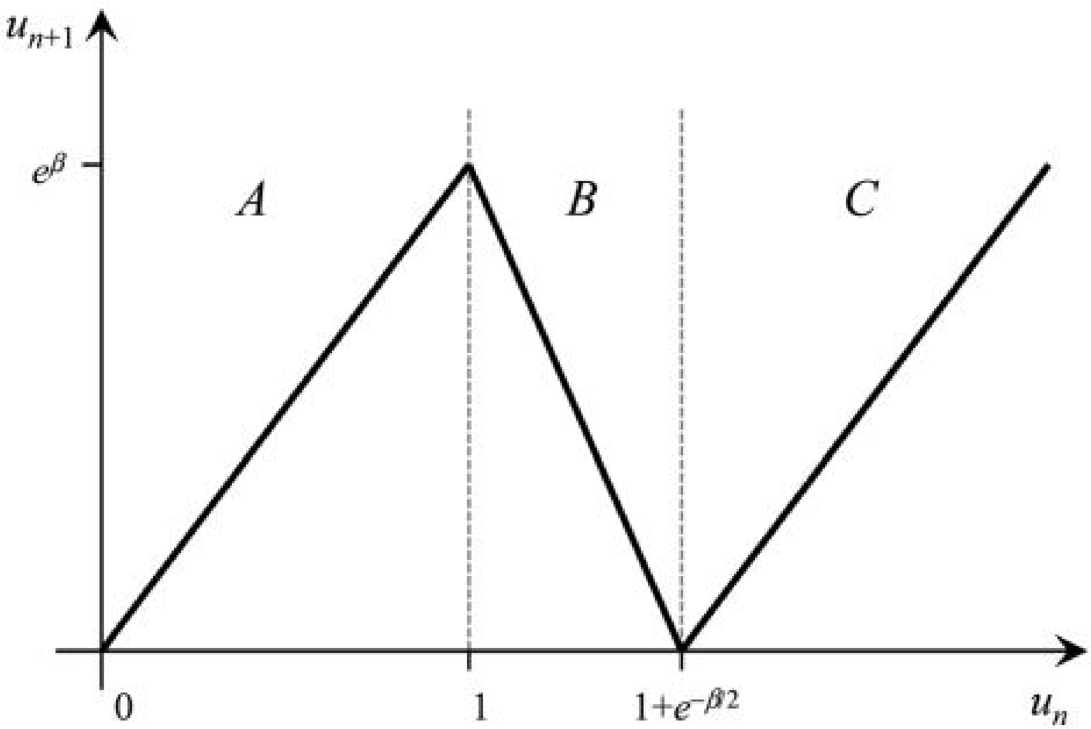

Figure 3.3: Return map for the chaotic system. Source: Adapted from [12]

The return map can then be derived by comparing each local maximum with the next local maximum in time. This relationship is modeled by

$$u(t_n + 1) = e^{\beta} u_n > 0, \tag{3.6}$$

which describes the first segment of the return map. The second and third segments are modeled by

$$u(t_n + 3/2) = (e^{\beta} + e^{3\beta/2}) - e^{3\beta/2} u_n > 0 \tag{3.7}$$

and

$$u(t_n + 1) = e^{\beta} u_n - (e^{\beta/2} + e^{\beta}), \tag{3.8}$$

respectively. The return map generated by these three parameters is shown in Fig. 3.3. The slopes of segments A and C are identical, and the slop of segment B is negative, but greater in magnitude than A and C. Segment B corresponds to the "folding" shown in the phase-space representation. Each segment has a slope magnitude greater than one, indicating chaotic behavior [42].

This chaotic system was also developed in circuitry [42]. This circuit was constructed using commercially available analog and digital electronic components, and the schematic for this circuit is shown in Fig 3.4. Notable off-the-shelf components are as follows: om-amps are all of type TL082, diodes 1N4148. The circuit is comprised of three stages: an RLC resonant circuit, a guard condition evaluation stage, and a switching stage. The box denoted by $-R$ corresponds to the realization of a negative resistor using an op-amp circuit, and the box denoted by L corresponds to the realization of an inductance by a negative impedance converter (NIC). The resonant portion of the oscillator circuit is modeled by

$$C\frac{dv}{dt} - \frac{v}{R} + i = 0 \tag{3.9}$$

and

$$L\frac{di}{dt} = v - v_s \tag{3.10}$$

where v is the voltage of the tank circuit, i is the inductor current, and v_s is a feedback term generated by the switching stage. This stage is responsible for adding oscillatory energy in the system. The value of $-R$ is paramount to generating chaotic behavior, and through tuning it was found that a magnitude of $R \approx 6.5k\Omega$ yielded the desired function. The tank voltage is fed to an op-amp configured as a comparator with the negative terminal tied to ground, which measures the sign of the tank voltage. The comparator op-amp is railed, so a positive tank voltage will correspond to a digital high value, and a negative tank voltage to a digital low. The diode and voltage divider bring the op-amp output voltage down to

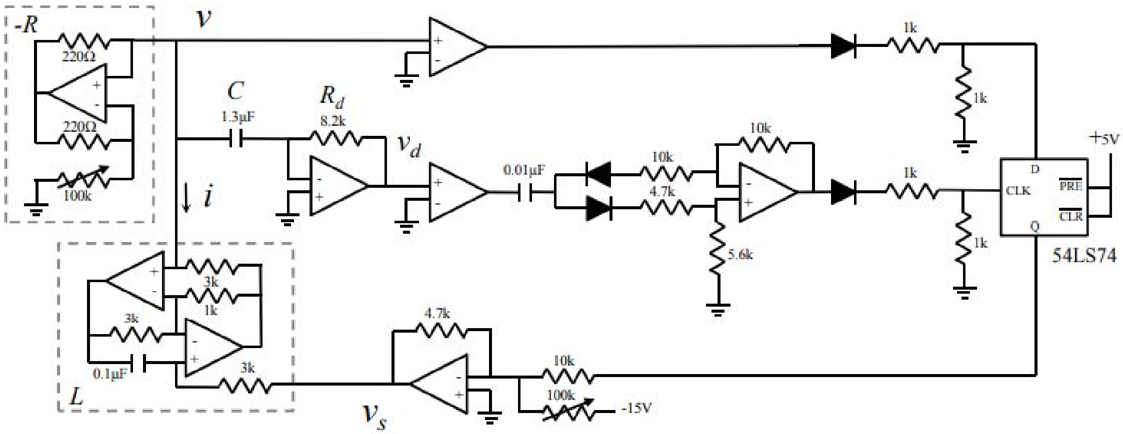

Figure 3.4: Schematic of low-frequency oscillator circuit. Source: Adapted from [42]

standard logic levels. The current through the capacitor is converted to a voltage v_d using an op-amp configured as a voltage-to-current converter; v_d is described by

$$v_d = -R_d C \frac{dv}{dt} \tag{3.11}$$

A comparator, configured in the same manner as the one first discussed, checks the sign of v_d. The DC logical output of the comparator is blocked by the 0.01μF capacitor, and is converted into a short pulse when the output of the comparator changes. This pulse is brought to the appropriate logic levels by the diodes and difference amplifier. Concisely put, the middle trace of the circuit generates a pulse when the sign of the derivative of v changes. This pulse is treated as the clock input to the flip-flop. The sign of the output voltage is treated as the input of the flip-flop, and the output is fed back to the resonant circuit. The flip-flop serves the function of holding the sign of the output voltage at every change in sign of the derivative of the output voltage. The output is fed to an op-amp configured as a summer; this amplifies and level-shifts the output of the flip-flop to symmetrical voltages, in this case $\pm$ 15V. The feedback circuit can effectively be modeled mathematically as a signum function

$$v_s = V sgn(v) + V_0 \tag{3.12}$$

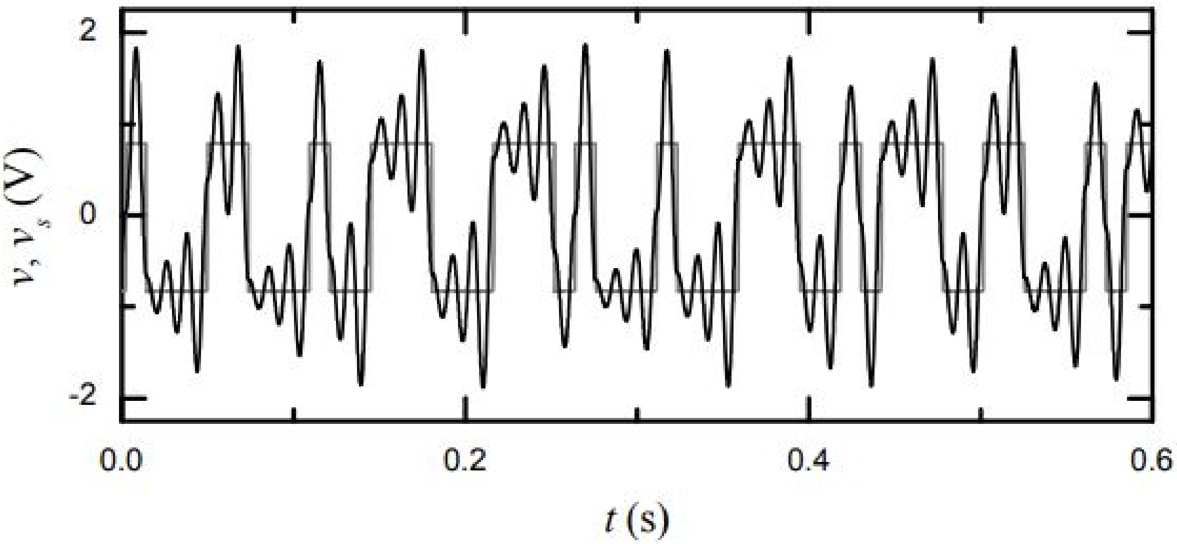

Figure 3.5: Simulated output of low-frequency electronic oscillator. Source: Adapted from [42]

where V_0 is a small offset to account for asymmetries in the physical implementation of the circuit. A simulated output of this circuit is shown in Fig. 3.5.

This circuit operates in a similar manner as those previously discussed: an oscillation is centered around ± 1V, the oscillation grows until the amplitude is greater than 1V and the oscillation reaches a local maximum. The guard condition forces the circuit to the other attractor and the oscillation begins to grow again. The location of the attractors over time defines the binary sequence, and this sequence is overlaid on the oscillator output. The operation of the circuit can also be understood through the phase-space representation, as shown in Fig. 3.6.

Examining the phase-space representation, assuming a starting position around the negative attractor, it can be seen that the trajectory spirals clockwise outward from the attractor until its radius from the attractor is greater than 1, and when the spiral reaches a y-axis value of zero, the feedback network forces the oscillation to take place around the opposite attractor, causing a fold in the trajectory. The trajectory then continues around the new attractor until the guard condition is met again. Upon simple visual examination, the presence of chaos can be deduced from the thick, dense bands around each attractor.

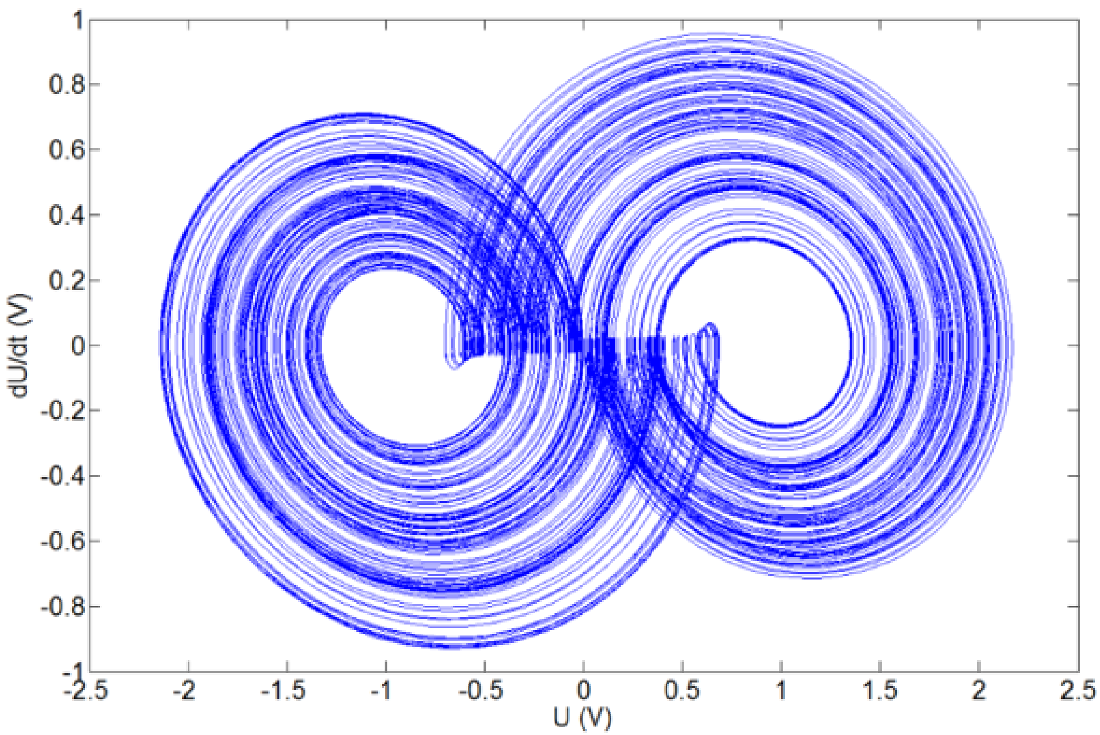

Figure 3.6: Phase-space representation of the electronic chaotic oscillator. Source: Adapted from [42]

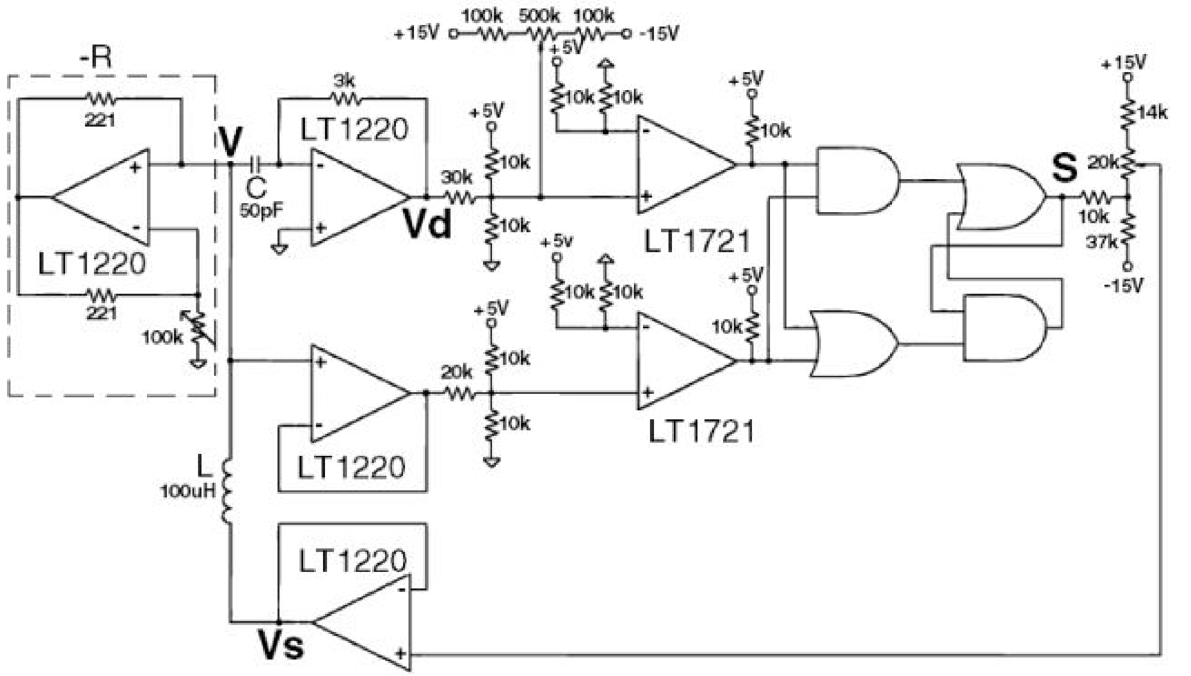

Figure 3.7: Electronic schematic for the high-frequency chaotic oscillator. Source: Adapted from [5]

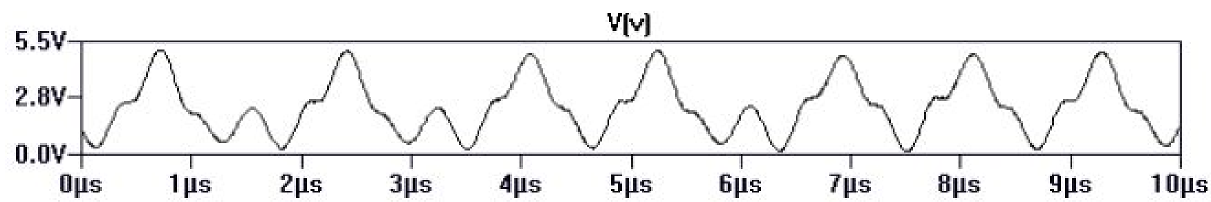

Figure 3.8: Output of the high-frequency chaotic oscillator. Source: Adapted from [5]

3.1.2 High-Frequency Oscillator in Electronics

Whereas the original system was derived and proved as a low-frequency system (roughly 84 Hz), a similar chaotic system has been realized at high-frequency, allowing for feasible use in communication systems. This high-frequency system was realized by Beal et al. [5] as an exact folded-band chaotic oscillator, with a similar theoretical derivation to the circuit developed by Corron et al. The schematic for the implementation in electronics of the high-frequency chaotic system is shown in Fig. 3.7.

Components were chosen for the specific purpose of high-frequency operation. All op-amps utilized in this design are Linear Technology LT1220, due to its favorable characteristics

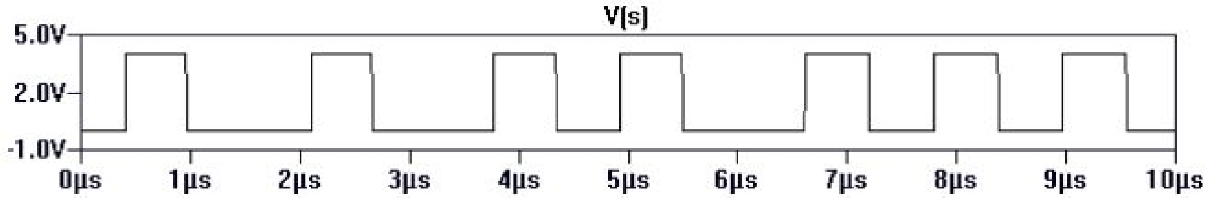

Figure 3.9: Symbolic content of the high-frequency chaotic oscillator. Source: Adapted from [5]

at the desired operating frequency of 1 MHz. The impedance converter used to simulate inductance in the low-frequency design was replaced with a real inductor, with a value of $100uH$ and a series resistance of 1Ω. Potentiometers were added to allow for tuning of various parameters to ensure the sensitive conditions that result in chaotic behavior are met. These adjustments included scaling the value of v_d and v_s, and adjusting the value of $-R$. The flip-flop used for the generation of the symbolic content in the low-frequency design was replaced with a network of logic gates. This network is fed by v and v_d converted to logic levels. The output of the oscillator is shown in Fig. 3.8 and the symbolic content is shown in Fig. 3.9. Additionally, a phase-space plot of the circuit simulated with real components is shown in Fig. 3.10.

Due to the circuit being simulated with non-ideal components, there are a few noticeable differences in the resulting phase plot as compared to the phase plot of the low-frequency oscillator. Perhaps the most obvious difference is the “wrinkling” of the plot near the folding action. This is likely caused by non-ideal switching within the logic network, causing some secondary oscillations in the resonant circuit. These oscillations are quickly damped and do not have a major effect on the desired operation of the system. Secondly, the folding condition does not take place exactly along the x-axis, and this can perhaps be attributed to hysteresis within the non-ideal active components present. Lastly, the attractors are no longer centered around ± 1; this is certainly caused by the lack of chaotic behavior during the simulation being remedied by an adjustment of the potentiometer controlling the levels

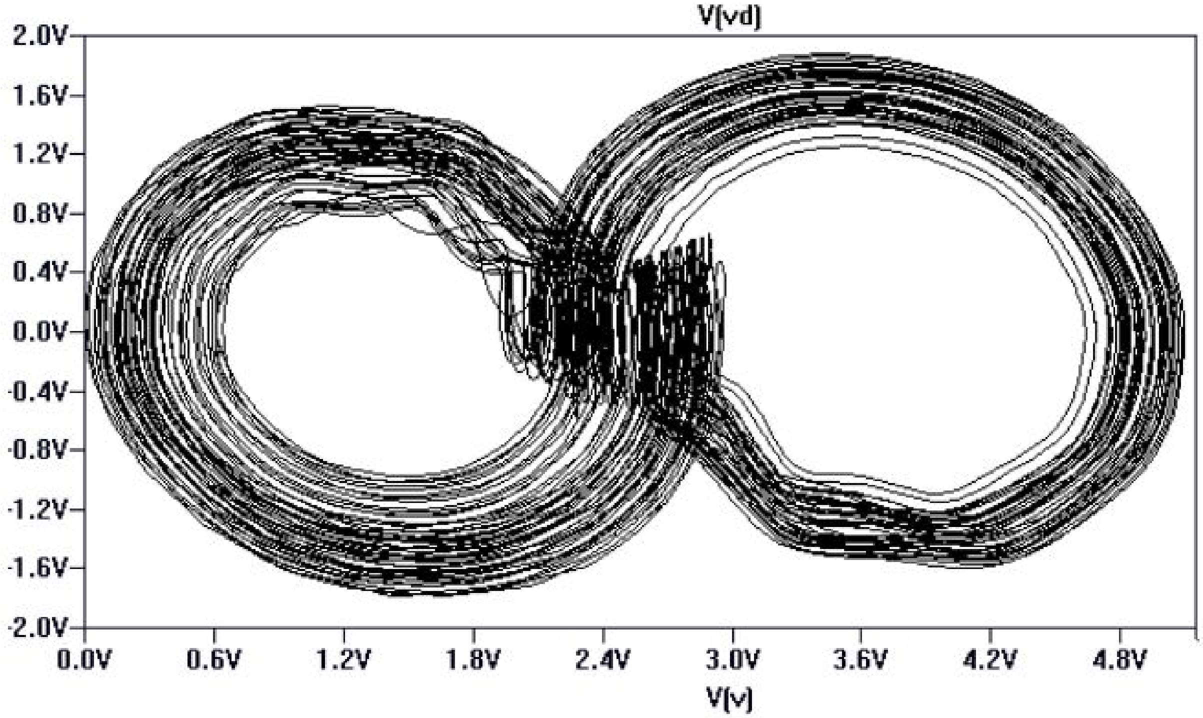

Figure 3.10: Phase-space representation of the high-frequency oscillator. Source: Adapted from [5]

of the symbolic content s. Overall, this system functions within the necessary parameters to allow for use in communication systems.

3.1.3 The Single-Transistor Chaotic Oscillator and its Implementation in Hardware

While the previously discussed high-frequency system simulated with real components is able to function properly within the simulation environment, implementation in hardware is somewhat difficult due to the narrow band of tuned parameters that allow for chaotic operations, and the inherent variance of virtually every component with respect to the ambient conditions. For this reason, a lower-frequency system was designed by Rhea et al. [45] to somewhat account for the variations in the ambient conditions in which the physical oscillator would be placed, and allow for the system to be re-tuned relatively infrequently.

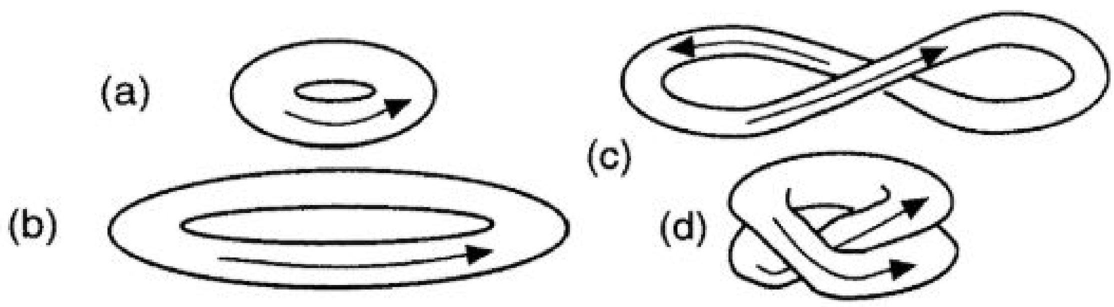

Figure 3.11: Stretch-twist-fold phenomenon. Source: Adapted from [44]

This system is based on the previously discussed exactly solvable chaotic system that can be considered a convolution of a linear basis function and a binary sequence. In this system, the two attractors are defined as ± 1. The design of this particular system is based on an easy-to-grasp description of chaotic behavior by E. Ott [26]. The process described is that of the manipulation of a circle. To begin, a circle is stretched until it reaches a certain size, then it is twisted into a figure-8, and finally folded back on itself, at which point the two stacked circles merge and become a single, smaller circle. A visual representation is shown in Fig. 3.11, and it is known that this phenomenon can describe the behavior of various systems. [44]. The "stretching" corresponds to the exponential sinusoidal growth of the system. In the implementation of this design, this is performed by a negative RLC circuit. A negative impedance implies the addition of energy into the system. This growth is unstable, so there must be a limiter in place to prevent failure of components. This is performed by a clock signal that is generated to trigger the "twisting" and "folding" processes, which are analogous to the changing of attractors and removal of energy from the system. A block diagram of this process is shown in Fig. 3.12.

A feature of this design is the replacing of the op-amp NIC with a single bipolar junction transistor (BJT) in common-base configuration. A schematic of the resulting resonant circuit is shown in Fig. 3.13. The relevant components were selected with respect to a resonant frequency of 18.4 kHz. The capacitors C1 and C2 function as both bypass capacitors for the

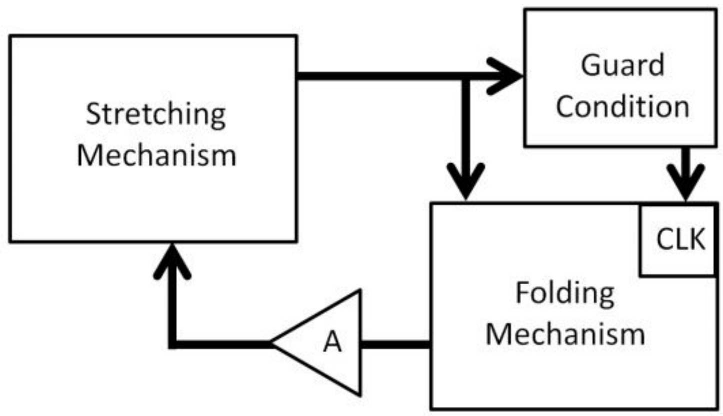

Figure 3.12: Block diagram of approach to oscillator design. Source: Adapted from [45]

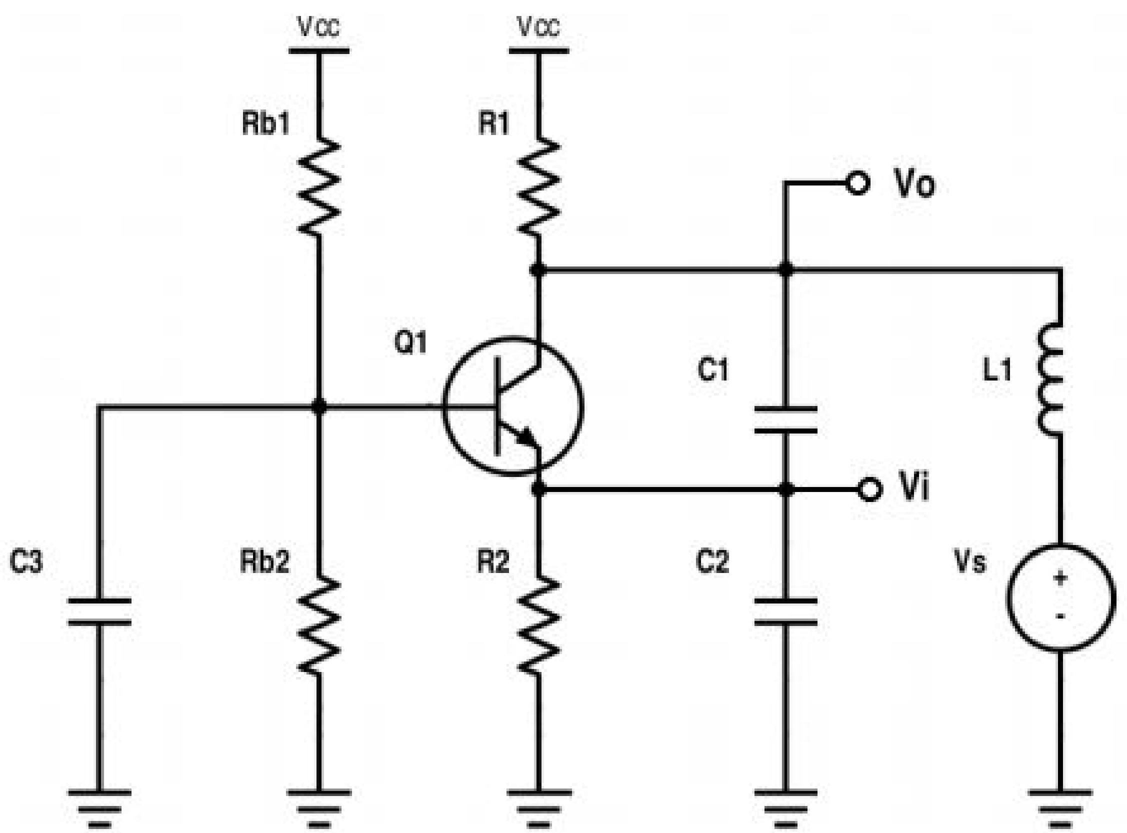

Figure 3.13: Schematic of 18.4 kHz resonant circuit. Source: Adapted from [45]

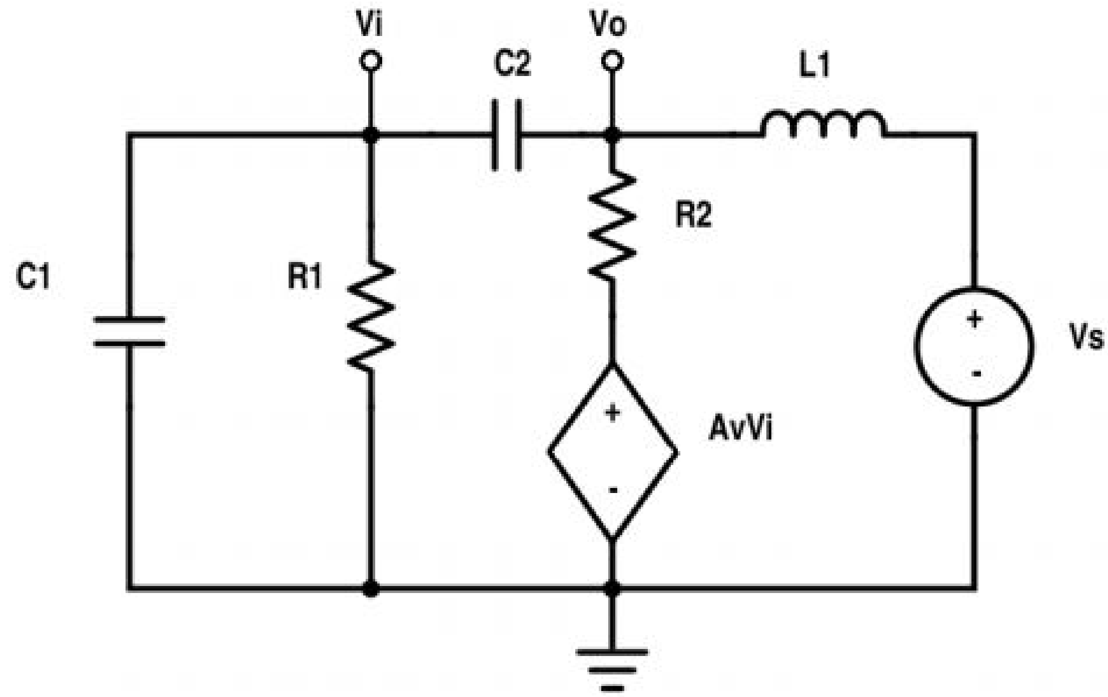

Figure 3.14: Equivalent circuit for single-transistor resonant circuit. Source: Adapted from [45]

bias network of the transistor, and as a component of the resonant circuit formed with L1. The voltage source V_s is defined by the piecewise linear function previously discussed, and it corresponds to the feedback voltage that serves to center the oscillation at one of the two defined attractors. Modeling the transistor in its application as a voltage-controlled voltage source, and knowing the states of the discrete-time signal, nodal analysis can be performed on the equivalent circuit, shown in Fig. 3.14, yielding

$$\frac{V_i}{\frac{1}{sC_1}} + \frac{V_i}{R_1} + \frac{V_i - V_0}{\frac{1}{sC_2}} = 0 \tag{3.13}$$

$$\frac{V_0 - V_s}{sL} + \frac{v_0 - V_i}{\frac{1}{sC_2}} + \frac{V_0 - A_0 V_i}{R_2} = 0 \tag{3.14}$$

where V_0 is the output taken from the collector of the BJT, V_i is the input to the emitter, and V_s is a known binary sequence. The transfer function for this circuit is described by

$$\frac{V_0}{V_s} = \frac{s(C_1 + C_2) + \frac{1}{R_1}}{s^3(LC_1C_2) + s^2(B) + s(C_1 + C_2 + \frac{L}{R_1R_2}) + \frac{1}{R_1}} \tag{3.15}$$

where

$$B = (\frac{C_2L}{R_1} + \frac{L(C_1 + C_2)}{R_2} - A_VC_2L). \tag{3.16}$$

One drawback to the design approach for this system is that the resonant circuit being replaced by the single BJT and the tank circuit constructed with a physical inductor and capacitor effectively make this system third-order, though the third order terms are many orders of magnitude smaller than the first and second order terms. To ensure this behavior would not render the previously developed approach to the design of a matched filter ineffective, the circuit was constructed and further simulated.

For evaluation, a SPICE model of the circuit was designed and tested. The circuit utilizes the single transistor model in combination with an LC tank circuit to create the required -RLC component. The resonant circuit effectively adds energy to the system, creating a growing oscillation. This oscillation is fed into a comparator with the negative reference terminal grounded. The output of the comparator is fed into a D-latch as the input D, which functions as the folding mechanism of the system. The clock signal of the latch is controlled via a network that determines the sign of the derivative of the resonant circuit output. This network acts as the guard condition and functions by feeding the output of the resonant circuit into an op-amp configured as a differentiator. To detect a zero-crossing of the derivative, two differentiators, one with the positive terminal as a reference and the other with the negative terminal as a reference, take the output of the differentiator and compare it to ground. The outputs of these comparators are fed into a NOR gate, which then acts as the clock signal for the latch. The function of this center trace is to create a pulse that will cause the latch to accept a new input from the comparator on the top trace. This pulse is created due to the hysteresis present in the dual-comparator configuration, which means that during any zero-crossing event, there will be a short time when both comparators are driven low, which will drive the output of the NOR gate high. A schematic of the full circuit is shown in Fig. 3.15, and the simulation results are shown as a time-domain plot (Fig. 3.16) and a phase plot (Fig. 3.17).

From visual inspection of the simulation results, it is not possible to detect the presence of third-order dynamics. The system behaves in a more desirable manner, in fact, as the transient periods and noticeable hysteresis present in the high-frequency design are not present in this design.

This circuit was implemented in hardware via a four-layer PCB. To ensure proper operation, the components were laid out with consideration to reducing the average trace length. Additionally, the second and fourth layers were configured as ground planes to isolate the components on the top layer from the power rails on the third layer. Potentiometers were placed at the base, collector, and emitter of the BJT, as well as on the SMA output of the oscillator, to ensure compatibility with transmitter and receiver components. Trimmer pots were also used on the V_s output to ensure the levels could always be tuned to ± 1V. These design considerations made the physical oscillator relatively easy to tune to account for changes in hardware component parameters due to ambient conditions. A time-domain oscilloscope capture is shown in Fig 3.18, and a phase space capture is shown in Fig. 3.19. The time-domain capture shows probes of the output V and the symbolic content V_s. This operation, along with the presence of the double-scroll signature in the phase-space capture, is consistent with the representation of the dynamics of the theoretical systems.

3.2 Oscillator Controller

To allow the oscillator to act appropriately upon a desired serial sequence of binary data, a controller was developed by Rhea et al [46]. The design approach to the controller is based on "steering" the oscillator output value toward a desired value using proportional control, based on the difference between the oscillator's output and the desired reference

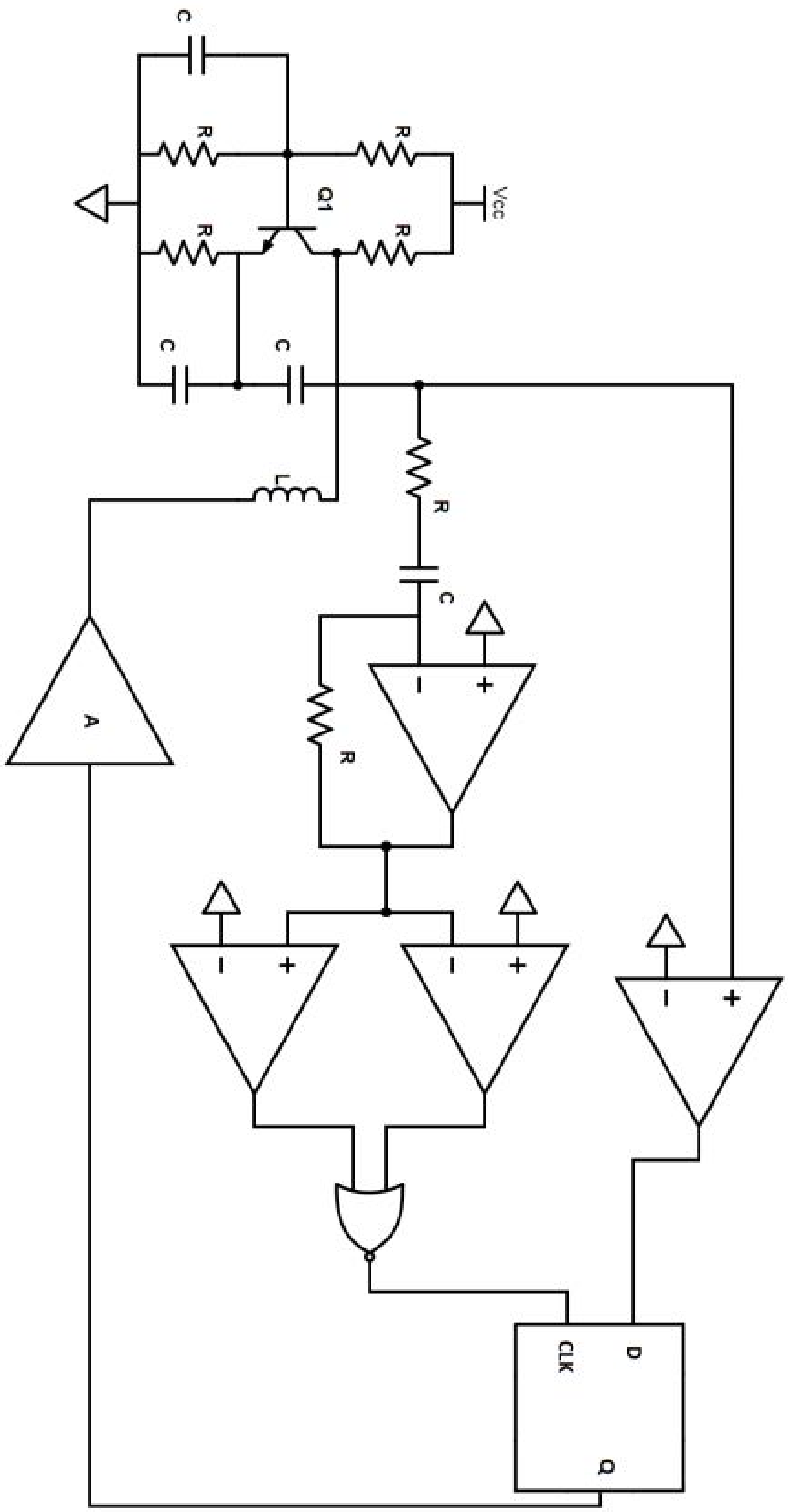

Figure 3.15: Schematic of single-transistor chaotic oscillator in SPICE. Source: Adapted from [45]

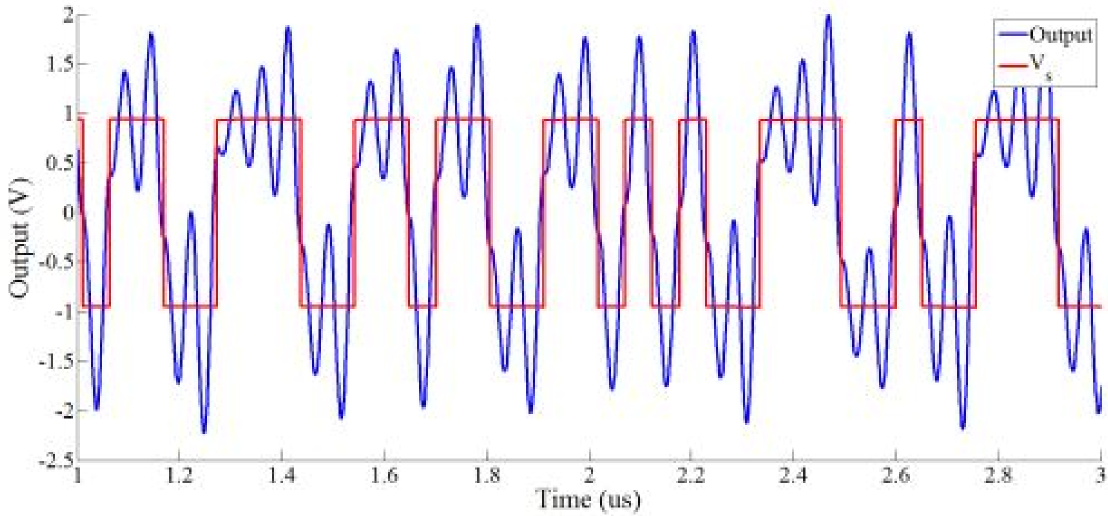

Figure 3.16: Time-domain output of oscillator. Source: Adapted from [45]

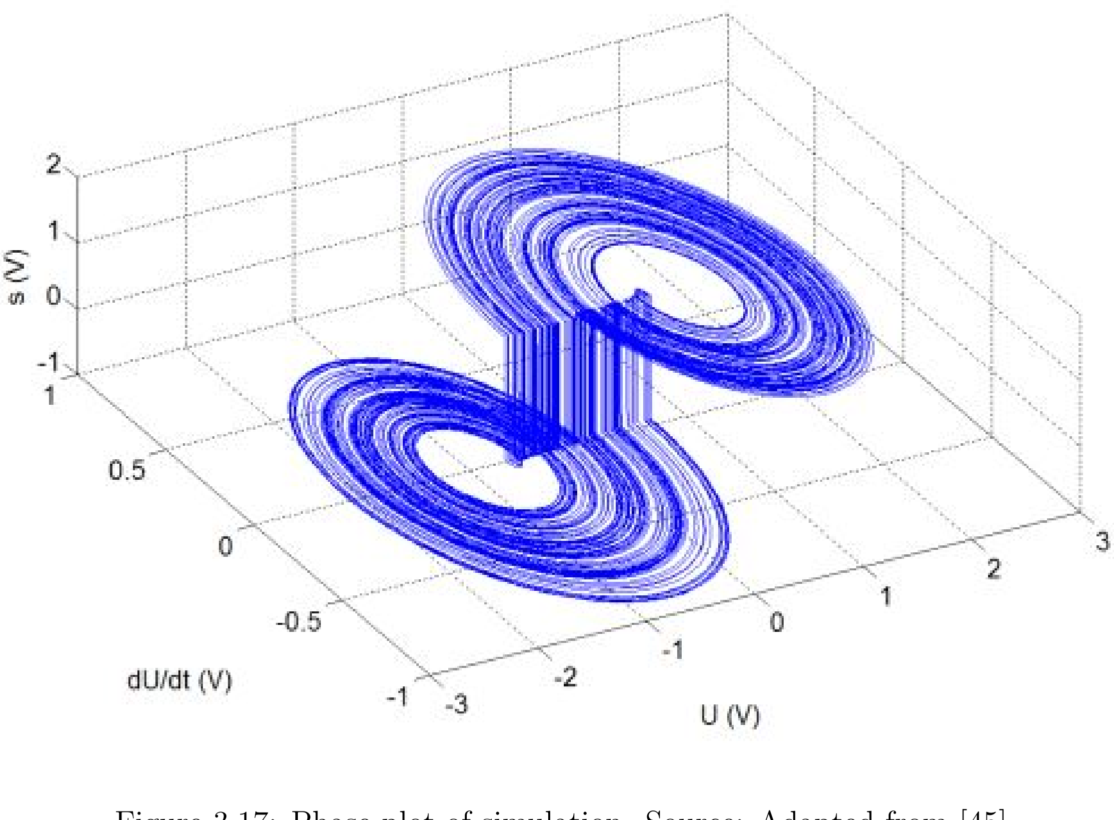

Figure 3.17: Phase plot of simulation. Source: Adapted from [45]

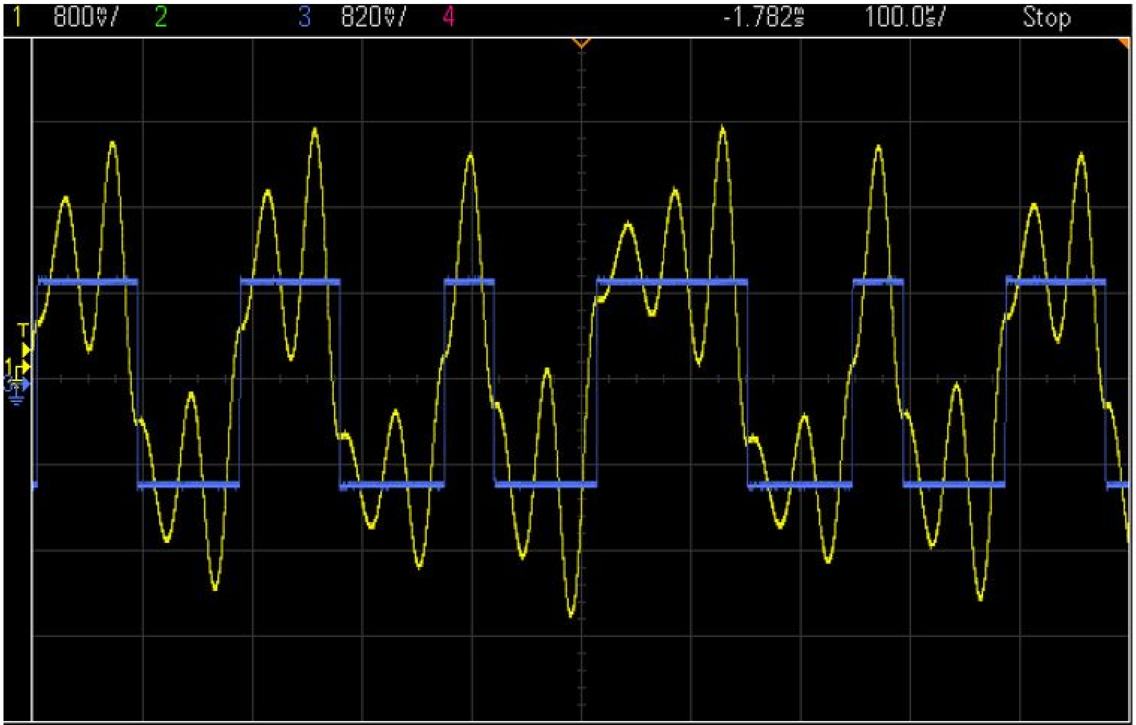

Figure 3.18: Time-domain oscilloscope capture of hardware oscillator output, overlaid with $s(t)$ (blue). Source: Adapted from [45]

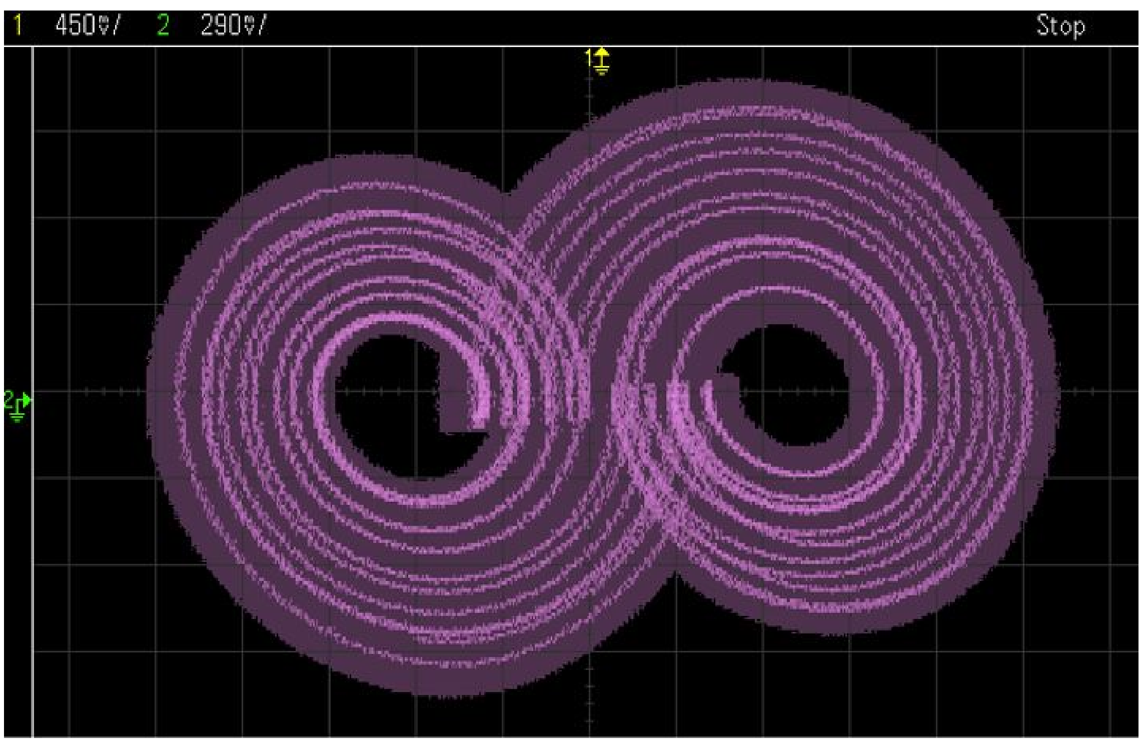

Figure 3.19: Phase-space oscilloscope capture of hardware oscillator. Source: Adapted from [45]

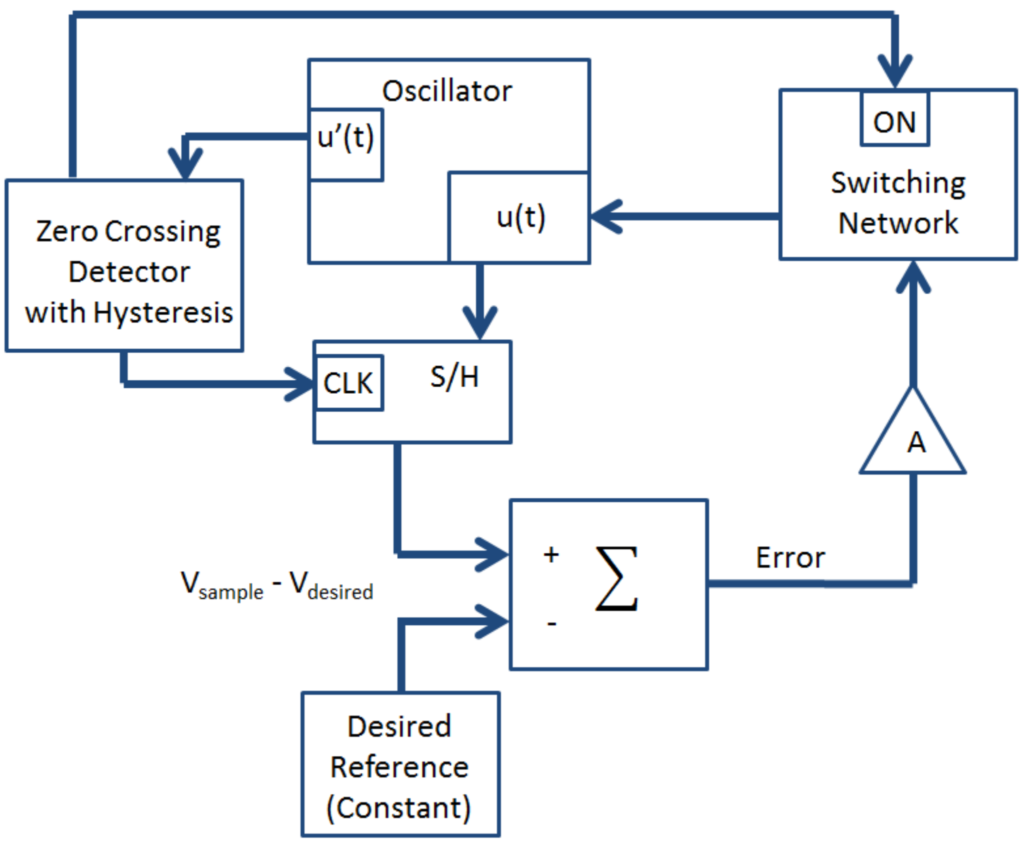

Figure 3.20: Oscillator controller block diagram.

voltage. This allows the reference voltage to be controlled externally, giving the system the ability to accept an input of symbolic data. This controller uses the oscillator output v and the output derivative v_d as inputs. A reference voltage is applied to the controller given the desired state of the symbolic portion of the oscillator $s(t)$. Both the reference voltage and the oscillator output voltage are buffered, and the reference voltage is scaled via an op-amp in a non-inverting configuration. A potentiometer is placed in the feedback path to allow for tuning of the value of the reference voltage. The oscillator output voltage and the reference voltage are then compared, and an error value is generated. Meanwhile, the derivative of the oscillator output is fed into two parallel amplifiers; one inverting and the other noninverting. The outputs of the parallel amplifiers are then used as the inputs to a NOR gate. This network takes advantage of the hysteresis present in the amplifier configuration to generate a short pulse as the derivative of the oscillator output crosses zero. This pulse is fed into the gate of an N-channel MOSFET, while the drain is driven by the previously generated error value. The output is taken at the source of the MOSFET, and is a pulse that varies in magnitude depending on the error magnitude and is allowed to conduct when given a pulse from the NOR gate. This pulse is fed into a non-inverting op-amp circuit, which contains a potentiometer to allow for the adjustment of pulse length. This effectively "steers" the oscillator back to the desired trajectory when the oscillator is nearing the triggering of the guard condition. It is necessary to adjust the pulse length somewhat infrequently due to the small variances that can appear in the oscillator output caused by changing ambient conditions. A block diagram for the controller is shown in Fig. 3.20, and the schematic is shown in Fig. 3.21. The controller was implemented in hardware via a 4-layer PCB, with attention paid to overall trace length. The second and fourth layers were configured as ground planes to isolate the components from the power rails, as well as to ensure the variation in the ground reference for all components was as small as possible. An oscilloscope capture of the controlled oscillator is shown in Fig. 3.22. This shows the reference voltage given to the controller (green), and the corresponding hardware oscillator output (yellow). A more detailed view of the controller function is shown in Fig. 3.23. This shows the hardware oscillator output (pink), the pulses generated at the points where the derivative of the output crosses zero (yellow) and the magnitude of the error (purple) when the MOSFET is allowed to conduct.

3.3 Matched Filter Realization

To allow for the extraction of the symbolic content of the waveform, a matched filter was developed by Werner et al. [36], using the theory for the matched filter derived by Corron et al [22]. The matched filter is designed to be the ideal filter for a particular system by correlating the filter input with a basis function known to represent the dynamics of the transmitting system. The derived equation for the intermediate stage of a matched filter, discussed previously, is

$$\eta(t) = \int v(t'+1) - v(t')dt'. \tag{3.17}$$

This corresponds to the signal being passed through a delay line that delays the signal by one period, then subtracts that delayed signal from the origninal signal, and integrates. The subtraction of the delayed signal with the current signal cancels most of the periodic content of the received waveform, leaving only the large shifts between attractors. The integration stage acts as a low-pass filter, further removing the higher-frequency information left over after the subtraction stage. This description of the filter allows for easy implementation in analog electronics. A schematic showing the various stages present in the matched filter is shown in Fig. 3.24. The received waveform v is input into a delay line, consisting of four amplifier stages with unity gain. These op-amps are configured as inverting first-order all-pass filters. Four stages were chosen due to the fact that the oscillator waveform contains various frequency components, and the change in delay with frequency leads to some distortion in the output. This is especially notable with two stages, where each stage must

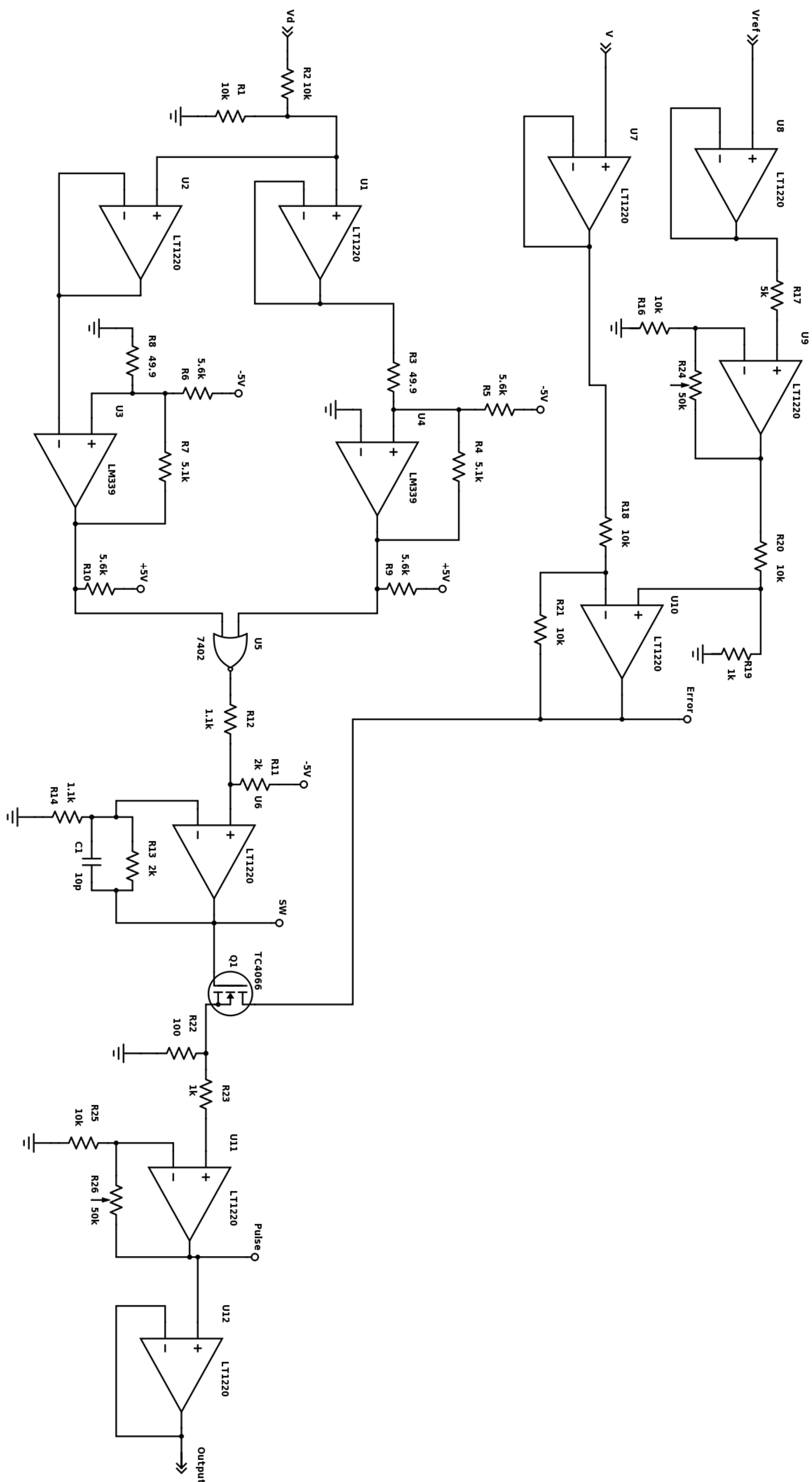

Figure 3.21: Oscillator controller schematic.

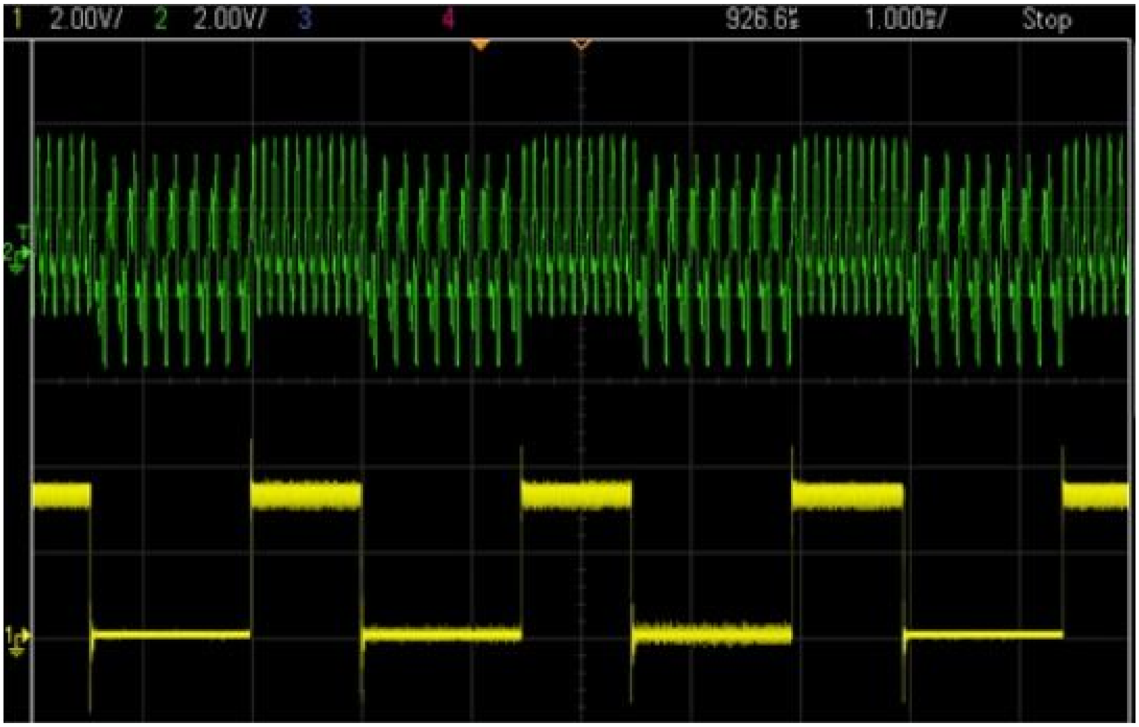

Figure 3.22: Oscilloscope capture of the oscillator output when the controller receives a 1-0-1-0 pattern. Source: Adapted from [46]

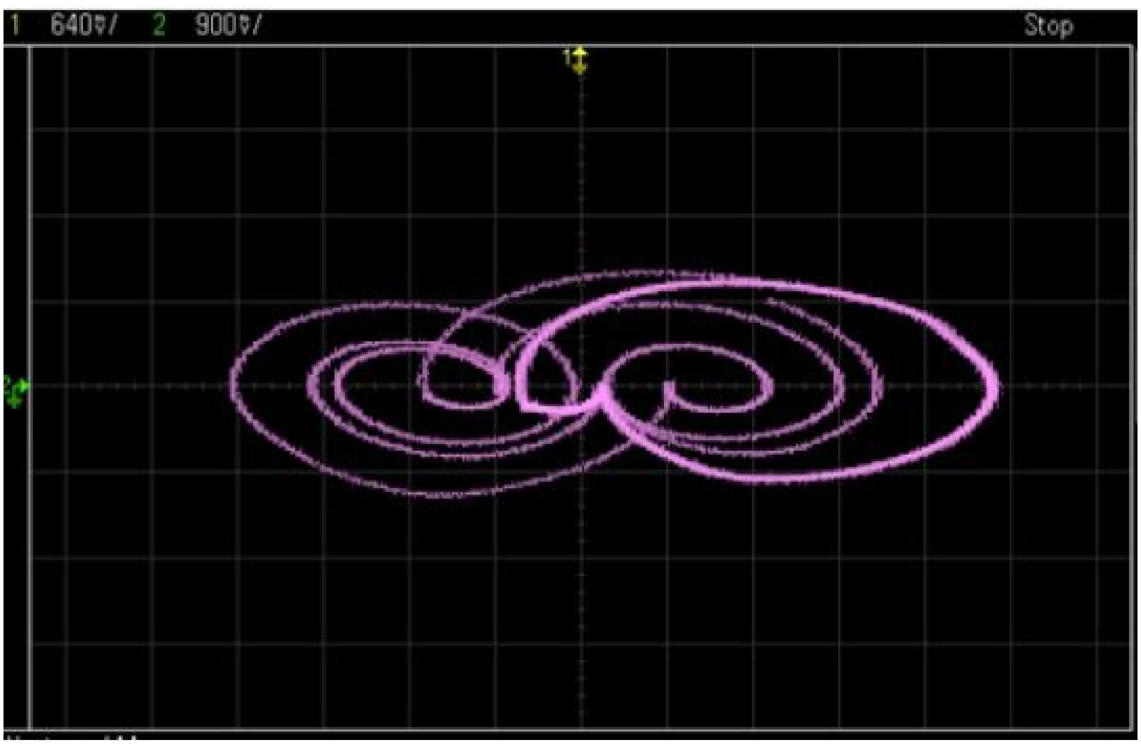

Figure 3.23: Phase-space capture of the oscillator receiving a 1-0-1-0 pattern. Source: Adapted from [46]

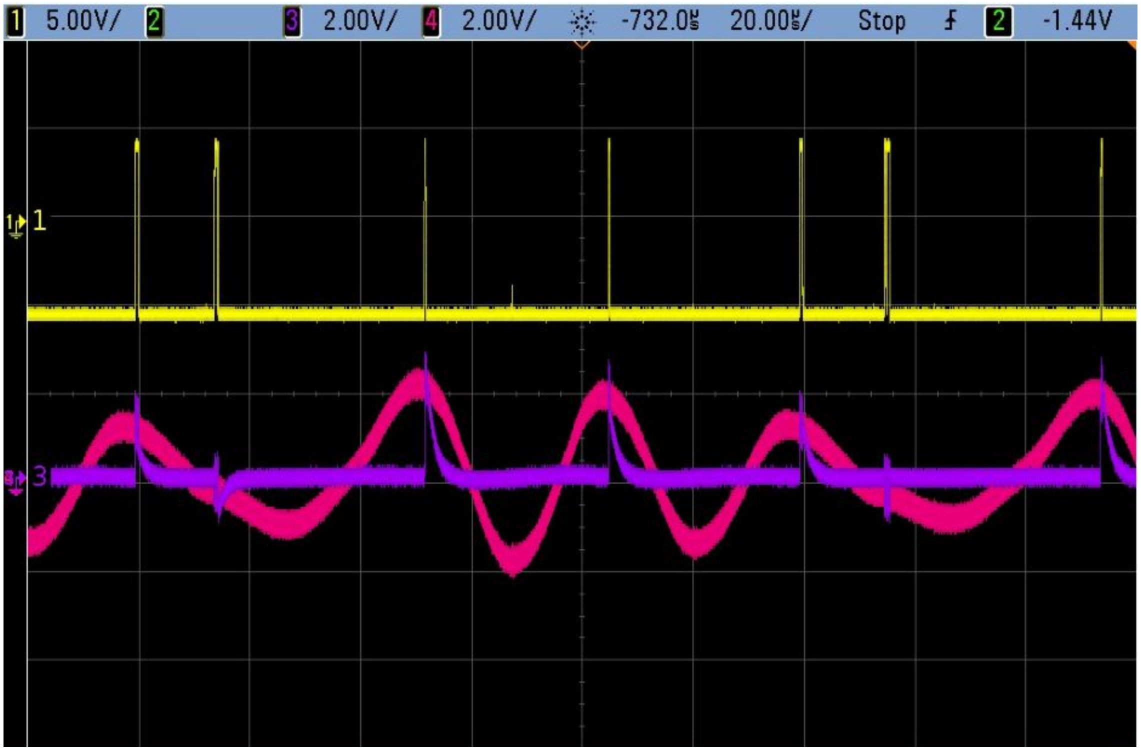

Figure 3.24: Oscilloscope capture of the function of the controller.

shift the signal through 180 degrees. To reduce distortion, the amount of delay required from each stage needs to be reduced. This leads to a small increase in complexity from the need for more stages, but the signal quality at the delay output is increased. This delayed signal, along with the original input, are fed into a difference amplifier, which subtracts the original input from the delayed input. This value is then fed to an op-amp configured as a low-pass filter, which acts as the transfer function $\frac{1}{s}$, which is integration in the frequency domain. The output of this integrator is η, the symbolic content of the original waveform. Additionally, a resonant circuit is present as the final stage of the design. This is used for verification purposes, to show the mathematically derived output ξ, which should contain the same oscillatory dynamics as the original chaotic system.

To verify this design and conjecture, a simulation of the electronic matched filter was created and tested using a simulated chaotic oscillator output mixed with noise. The results of this test are shown in Fig. 3.26. The original waveform (blue) is injected with noise

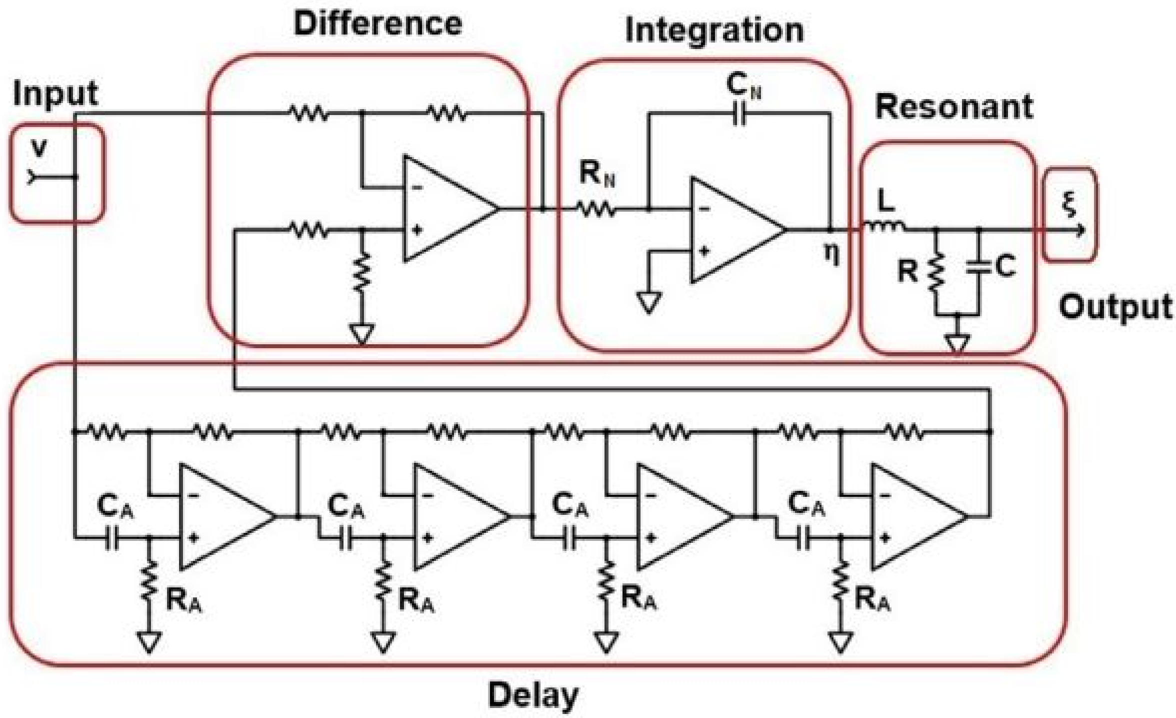

Figure 3.25: Generalized schematic of the analog matched filter. Source: Adapted from [36]

(red), and the matched filter is able to roughly produce a waveform with the original system dynamics. Discrepancies between the two waveforms are due to the fact that real filters have a limited passband, which will inevitably eliminate some high-frequency components of the spread-spectrum chaotic waveform, and also to the small variations in real components used to create the resonant circuit in the matched filter designed to replicate the resonant circuit in the oscillator. Additionally, the simulation was performed with deliberate alterations to the symbolic content of the waveforms using noise. The results of this test are shown in Fig. 3.27. The matched filter is able to remove the falsified data from the test waveform and reconstruct the original bitstream.

After demonstration of the successful function of the electronic design, the filter was constructed in hardware. To ensure the correct delay time, care must be taken in component selection. To ensure the fundamental frequency of the oscillator is not attenuated, C_N and R_N must be selected to ensure the gain bandwidth product is appropriate. To ensure each

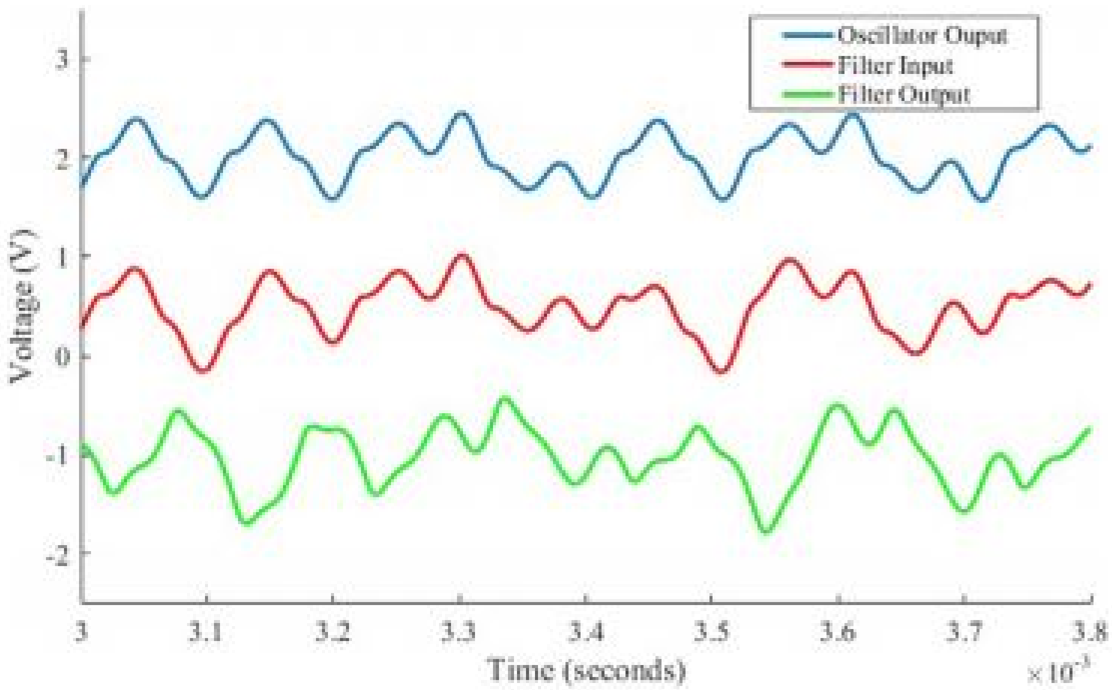

Figure 3.26: Results of the simulation of showing the output of matched filter when supplied with a noisy input signal. Source: Adapted from [36]

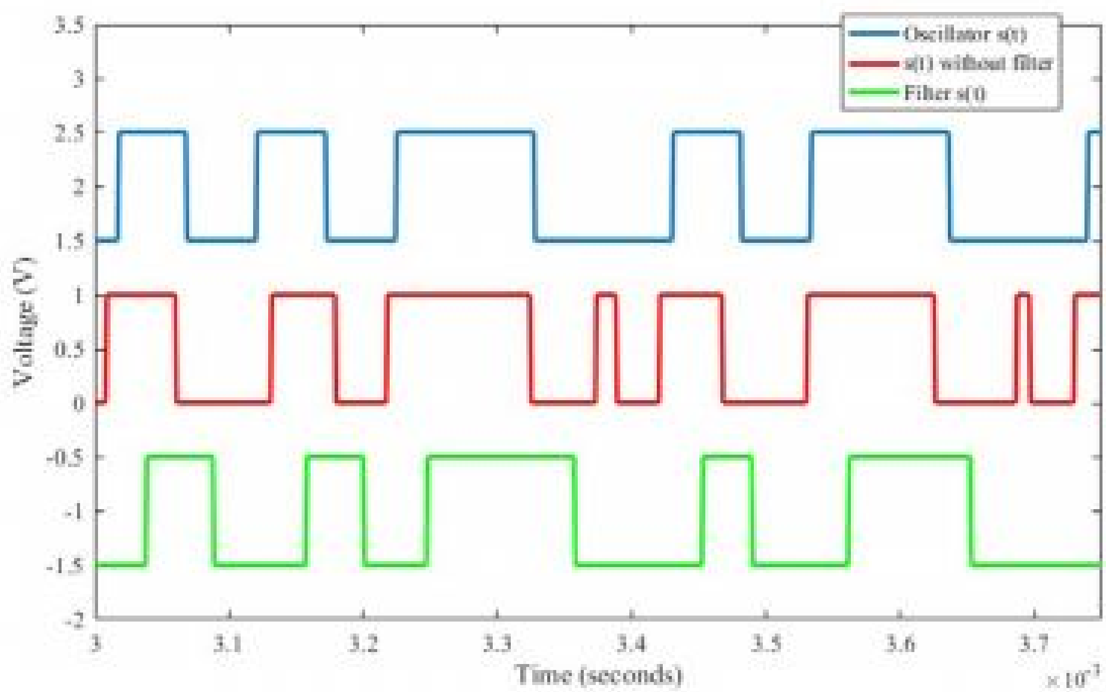

Figure 3.27: Simulation with falsified symbolic content and matched filter correction of falsified data. Source: Adapted from [36]

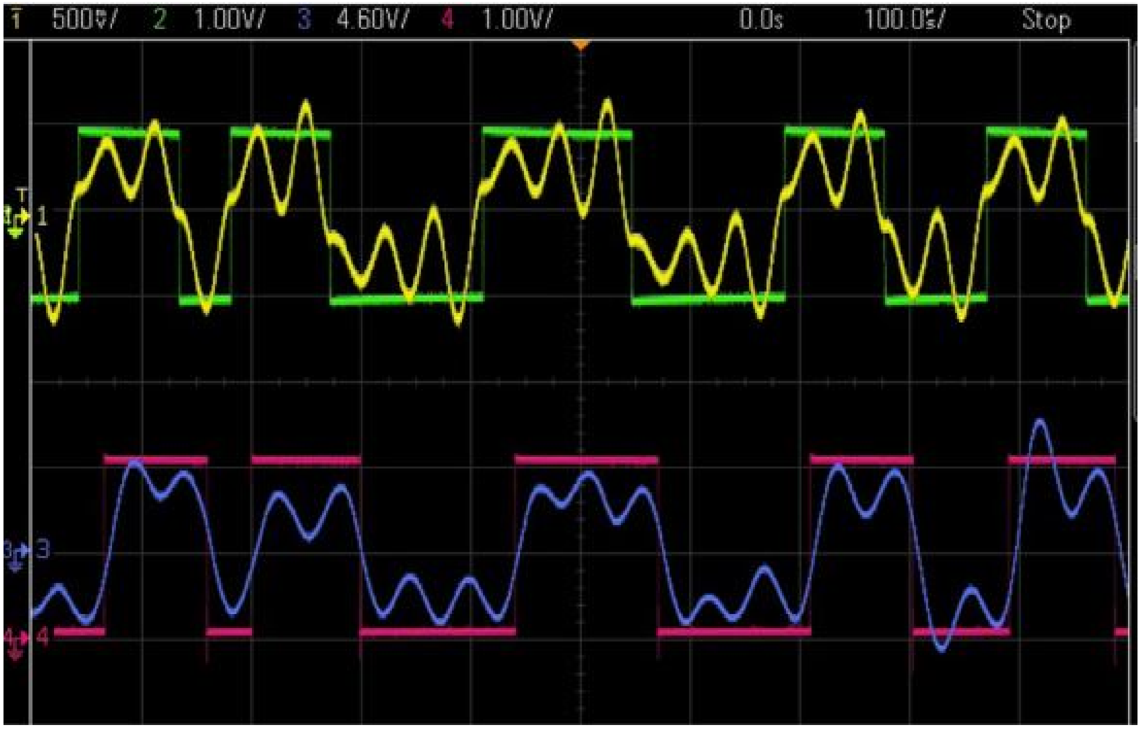

Figure 3.28: Oscilloscope capture of oscillator output overlaid with symbolic content, and the matched filter output ξ overlaid with symbolic output η. Source: Adapted from [36]

stage of the delay line produces a phase shift of 90 degrees, the physical components C_A and R_A must be selected to be as close to identical as possible Additionally, to recreate the original chaotic waveform as accurately as possible, the R, L, and C, present in the resonant circuit must match the resonant circuit of the oscillator. Given the oscillator's operation at 18.4 kHz, the component values were chosen to be $C_A = 0.1$, $R_A = 75\omega$, $C_N = 0.1$, $R_N = 84\omega$, $C = 0.5$, $R = 18k\omega$, and $L = 150$. This hardware matched filter was then tested with a hardware oscillator injected with a generated noise signal. Fig. 3.28 shows an oscilloscope capture of this test. The oscillator (yellow) is injected with a noise signal and its symbolic content (green) is altered. The matched filter is able to extract the correct symbolic data (red) and roughly reconstruct the original system dynamics (blue).

In addition to this analog filter, a digital matched filter was developed. This was done to increase the flexibility of the matched filter design for possible use in updated systems with oscillators that have different parameters to the one used in this system. The digital

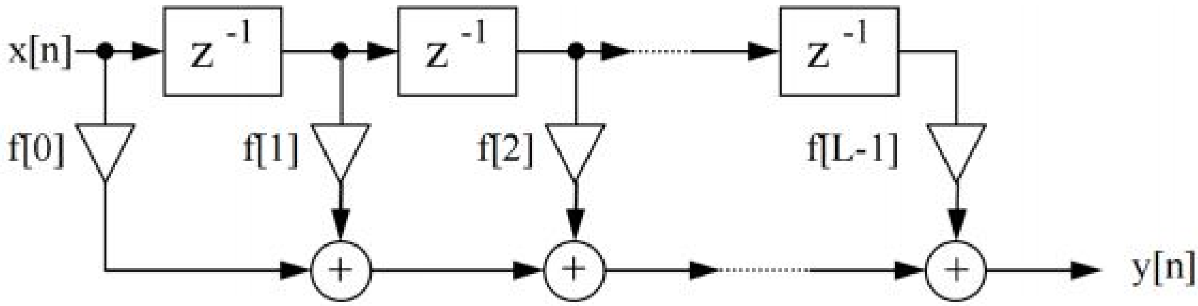

Figure 3.29: General diagram of an FIR filter. Source: Adapted from [50]

filter was implemented as a software-defined finite-impulse-response (FIR) filter. The general equation for an N^{th} order FIR filter is

$$Y[n] = X[n] * H[n] = \sum_{k=0}^{N-1} H[k] * X[n-k], \tag{3.18}$$

which is a sum of convolutions, and the exact function is described by the values of the FIR coefficients, H. [49]. A block diagram depicting the general form of an FIR is shown in Fig. 3.29. While this approach is implementable in hardware, it is less than desirable due to the use of multipliers; however, taking a software approach to this problem alleviates any issues using certain undesirable hardware components.

To lay out the algorithmic approach to the design of the matched filter in software, a MATLAB script was written to perform the operations required in an intuitive way. A Simulink model was designed to simulate the function of the oscillator. The model hierarchy is shown in Fig. 3.30 and Fig. 3.31. The simulated output of this oscillator was then decimated to simulate the waveform being sampled at a rate of 4 samples/second, which, at a fundamental frequency of 18.4 kHz, amounts to a sample rate of 36.8 kSamples/s; this sampling rate can be handled easily by commonly available hardware. Additionally, this sampling rate allows a sample to be stored until four more samples are taken, which will amount to a delay of 360 degrees. The decimated simulink output is shown in Fig. 3.32. This data is then scaled to an appropriate value to simulate a level-shifted signal that can

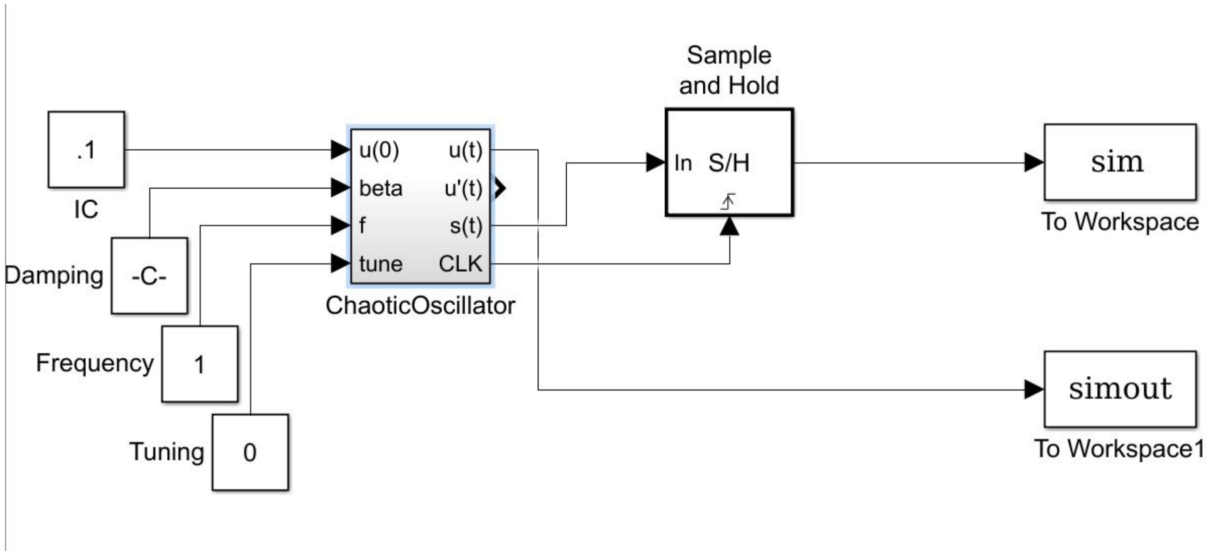

Figure 3.30: Simulink model of the oscillator with tunable parameters and outputs to the MATLAB workspace.

be interpreted properly by a 12-bit analog-digital converter (ADC). The array of data is then subtracted from the array shifted by 4 (for example, Arr[k-4]-Arr[k]) to simulate the current signal being subtracted from the period-delayed signal. The delayed signal is shown in Fig. 3.33 and the subtracted signal is shown in Fig. 3.34. The subtraction operation will show a spike when a change in attractor occurs, and will have a relatively small output when the attractor does not change between samples. To manipulate these spikes into the desired sequence of bits, a numeric integration stage is used. This creates a signal in which the spikes in the subtractor output cause the integrator's output to change, and the points at which no spikes occur cause little change in the final output. The output of the integrator stage compared with the subtraction stage is shown in Fig. 3.35. The output of the integrator is then compared to a threshold, and a digital output is generated. The digital output compared with the symbolic content of the initial waveform are shown in Fig. 3.36. By examining these results, it can be seen that the digital matched filter algorithm is able to successfully reproduce the symbolic content of the oscillator waveform. The MATLAB program used to demonstrate this algorithm is contained in Appendix A.

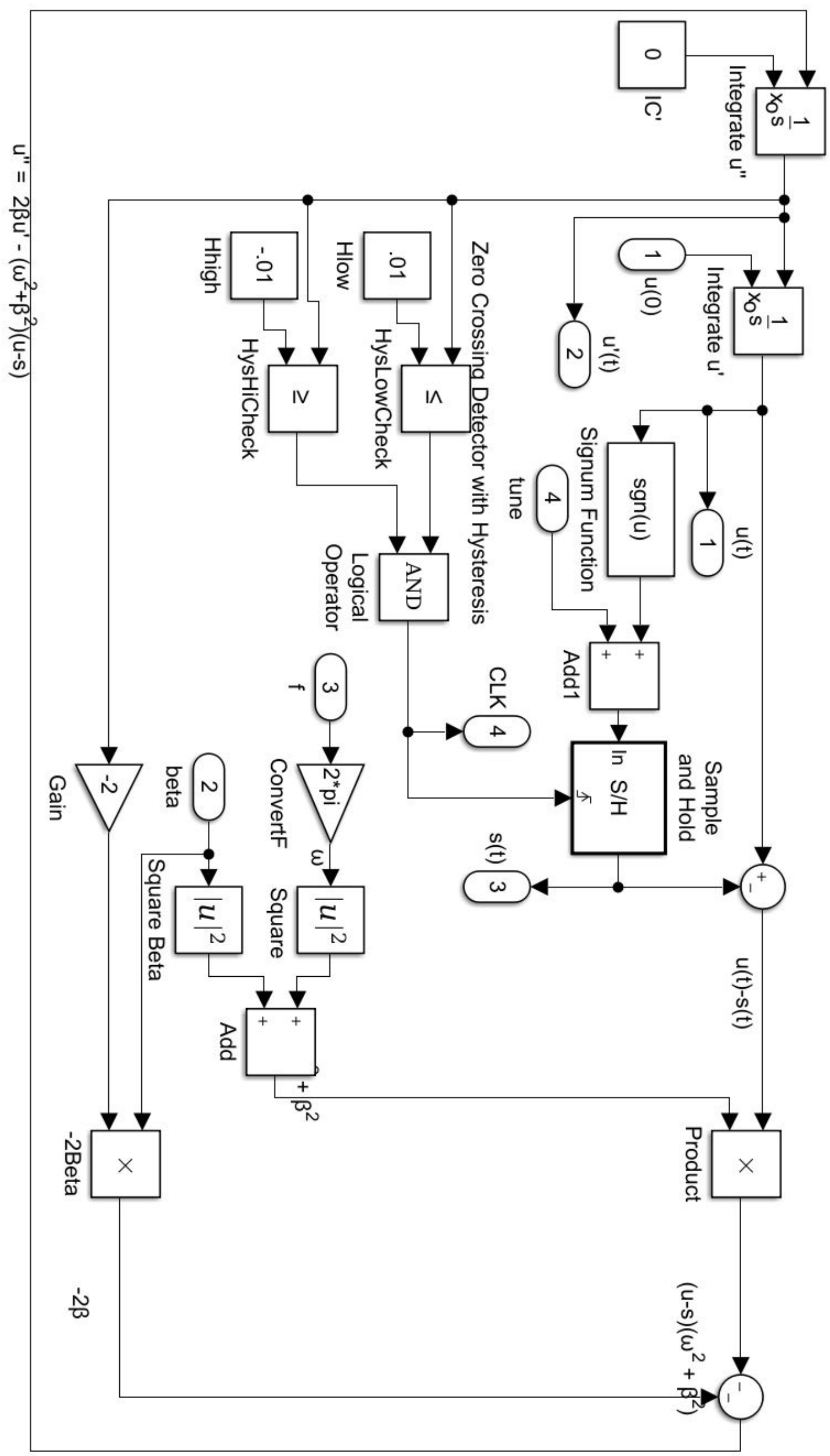

Figure 3.31: Simulink model of the chaotic equation block diagram.

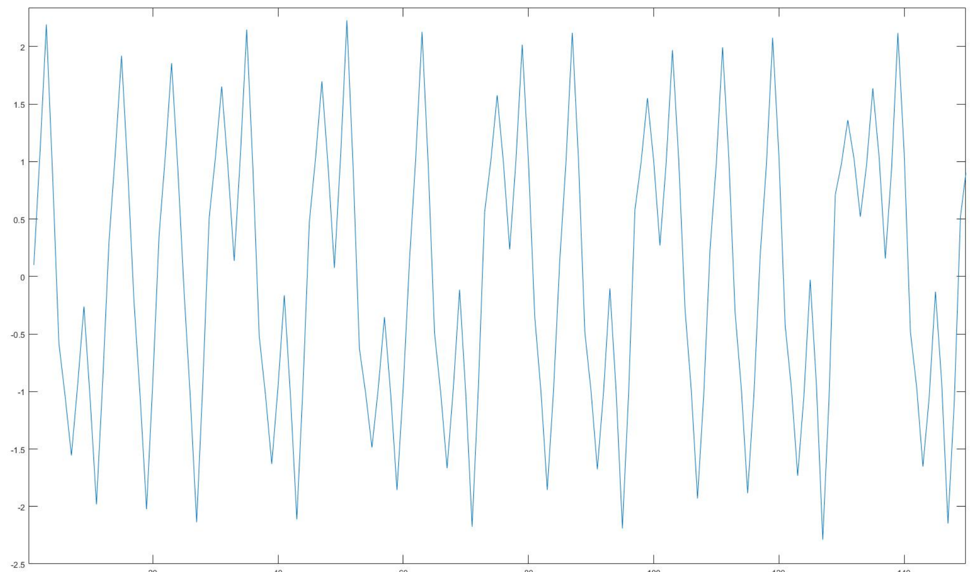

Figure 3.32: Decimated oscillator output to simulate sampling.

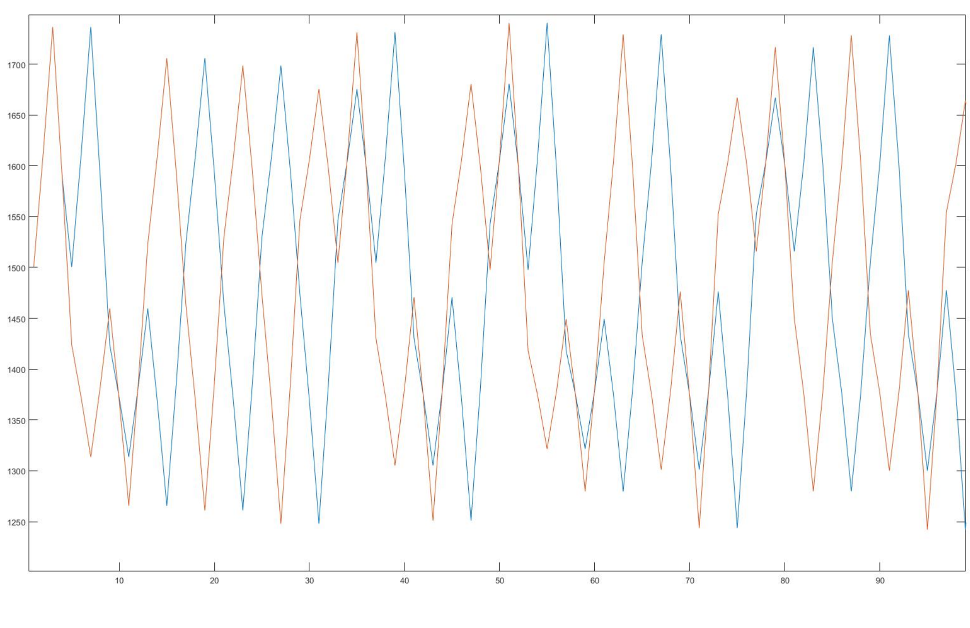

Figure 3.33: Sampled level-shifted and scaled oscillator signal (red) and delayed signal (blue).

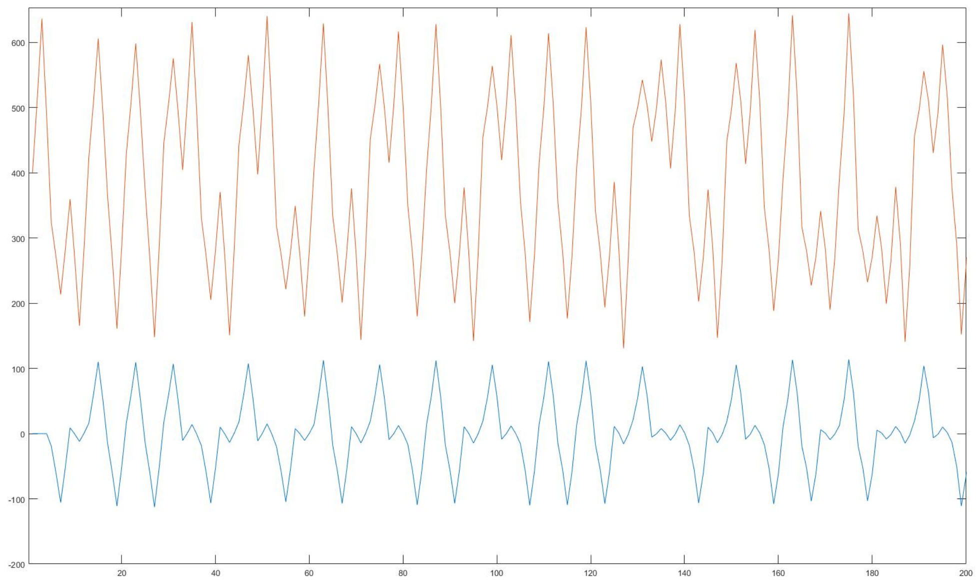

Figure 3.34: Sampled oscillator output (red) with output of subtraction operation.

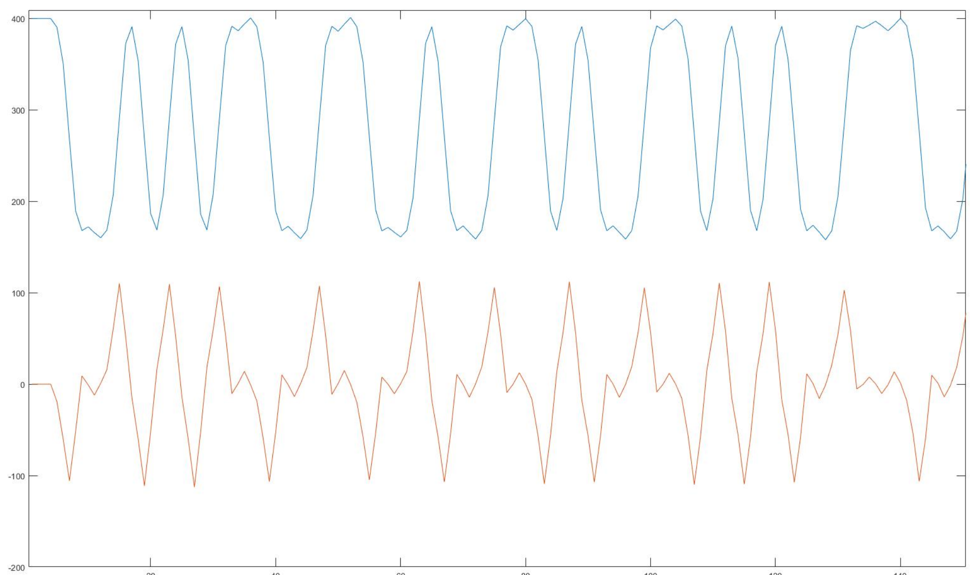

Figure 3.35: Output of subtractor (red) with output of integrator (blue).

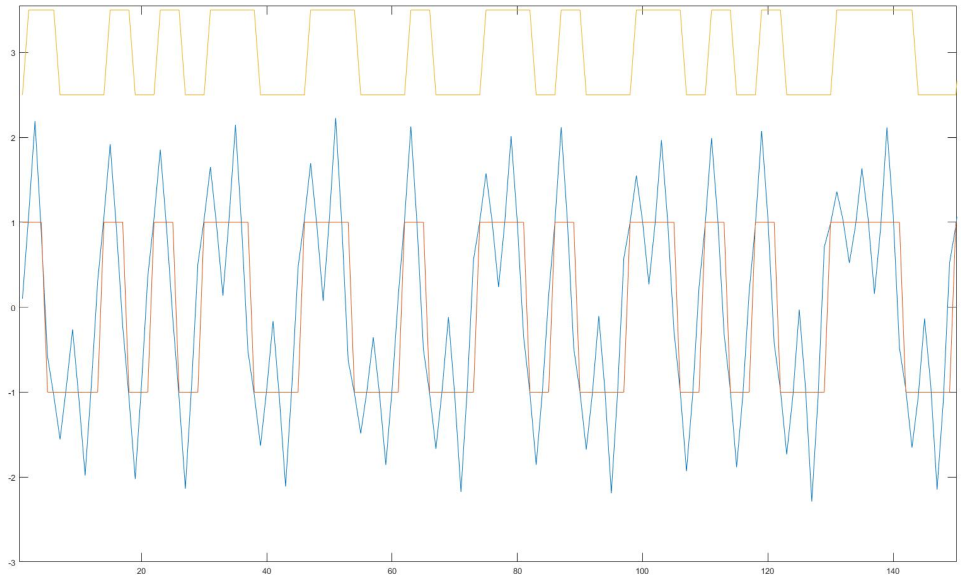

Figure 3.36: Sampled oscillator signal overlaid with symbolic content, and digital matched filter algorithm output (yellow).

While it may be relatively simple to implement this algorithm in an environment where nothing has to be done in real-time, far more considerations must be taken into account when implementing this system in real hardware. For this task, an ST microcontroller was chosen as the platform to implement the filtering algorithm. The STM32 Nucleo F446 was chosen for this task because of the onboard ADC, robust processor, and low cost. The development board is shown in Fig. 3.37. The ARMKeil MDK toolchain was used to set up the various layers of software and firmware; this toolchain has the advantage of easy instantiation of peripherals and other components necessary to the physical implementation of this design. The various programs used are included in Appedix B.

To allow the microcontroller's hardware to perform any operations on the signal, it must be converted to a digital waveform. This was accomplished by configuring a 12-bit ADC to sample the analog signal at a defined general purpose input-output (GPIO) pin on the microcontroller development board; the incoming signal was tested to ensure its compliance with the 0-3.3V operating range of the microcontroller hardware. An external timer was

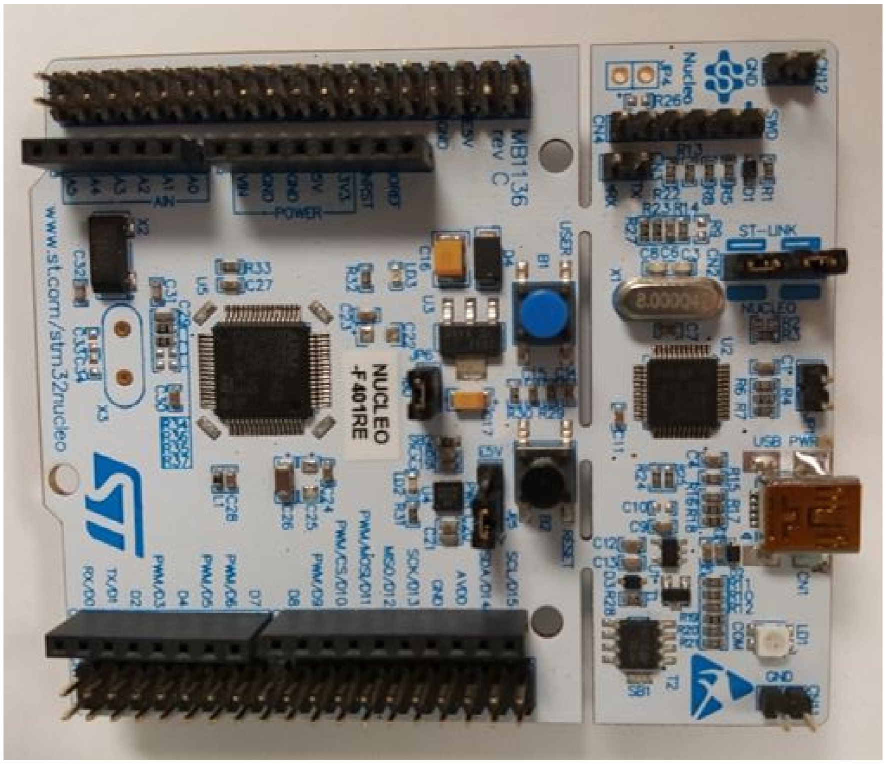

Figure 3.37: Development platform used to implement the software matched filter.

configured to trigger an ADC conversion at a rate of 73.6 kSamples/s. This was achieved by generating a counter that counts the cycles of an internal clock, and when the counter reaches the appropriate value, the output level of the timer transitions to the opposite logic level. The ADC notices the change in logic level, and it begins a conversion. When the conversion is complete, the ADC sets the end-of-conversion (EOC) bit in the appropriate register to HIGH. When the EOC bit is set, an interrupt request is sent to the microcontroller's nested vector interrupt controller (NVIC). Since the ADC interrupts are enabled, the processor calls an interrupt request handler, which reads the value from the ADC data register, clears the ADC interrupt flag, and calls a callback function. This callback function stores the ADC converted value as a variable into an element of an 8-element array. This array is used to store previously converted values for the last eight samples, to allow the oldest sample to be compared to the current, which acts as the delay and subtraction stages of the matched filter. After the current value is subtracted from the previous value, the previous value is overwritten by the current value in the array. A counter is used to select which element

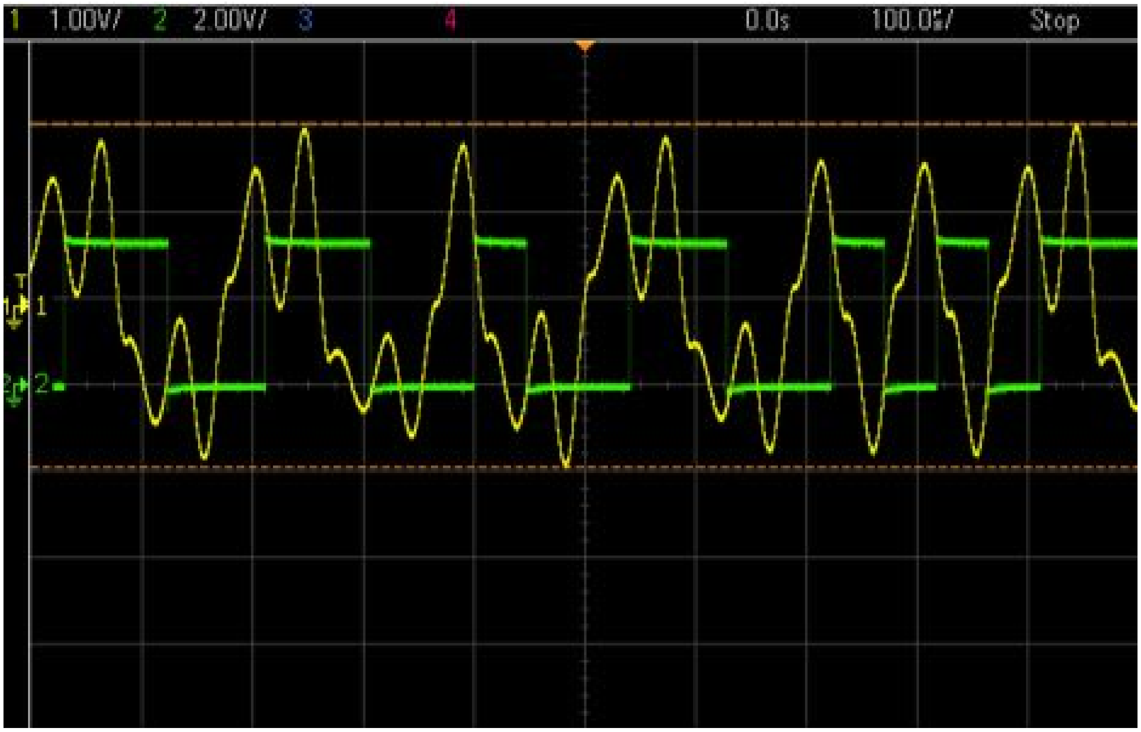

Figure 3.38: Software-defined matched filter (green) shown extracting correct symbolic data from oscillator output (yellow).

of the array is used for the subtraction and overwriting. The subtracted value is stored as a variable, and the variable is passed to a numeric integrator, which takes the sum of the current value and the previous value and divides by an integration constant. This value is then added to the previous output of the integrator. The integrator output is then compared to a set threshold and a GPIO pin configured as a digital output is set to a logic level. For testing and observation purposes, the output of the integrator is written to a 12-bit digital-analog converter (DAC), which is updated at the same rate at which the ADC sampling occurs. The function of the matched filter is shown in Fig. 3.38 and Fig. 3.39.

3.4 Encoding and Decoding

3.4.1 Encoding

To demonstrate the function of the communication system, data from an analog temperature sensor was used. As the temperature sensor itself was not within the scope of

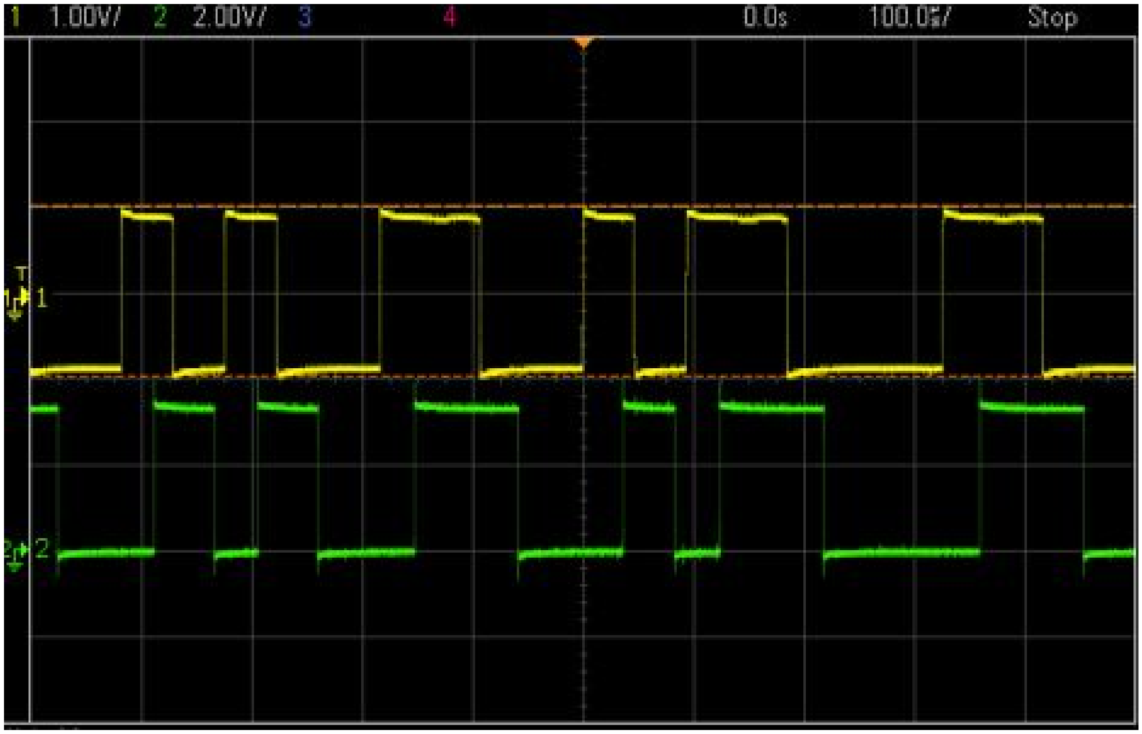

Figure 3.39: Symbolic content of oscillator (yellow) compared with software matched filter (green). Notice the small delay in the matched filter's output.

this project, the connection of the temperature sensor with the microcontroller was made as simple as possible. The microcontroller 5V power output and ground serve as the supply for the sensor, and the data output of the sensor is attached directly to a GPIO pin. Fig. 3.40 shows the temperature sensor connected with the encoder microcontroller.

It was discovered that the hardware implementation of the oscillator, when controlled by the hardware controller receiving an external input, is steered into two completely separate states, each with two orbits of their own. This is likely due to the small differences in the simulated oscillator and controller versus the realized versions, and the fact that these differences are greatly amplified by the use of a chaotic system. This led to larger amounts of energy being required from the controller to change the state of the oscillator, and noticeable transient periods were introduced [47]. To allow the communication system to function properly, these transient periods must be allowed to settle into the appropriate orbits so the matched filter can extract the binary data. This was accomplished by sending multiple

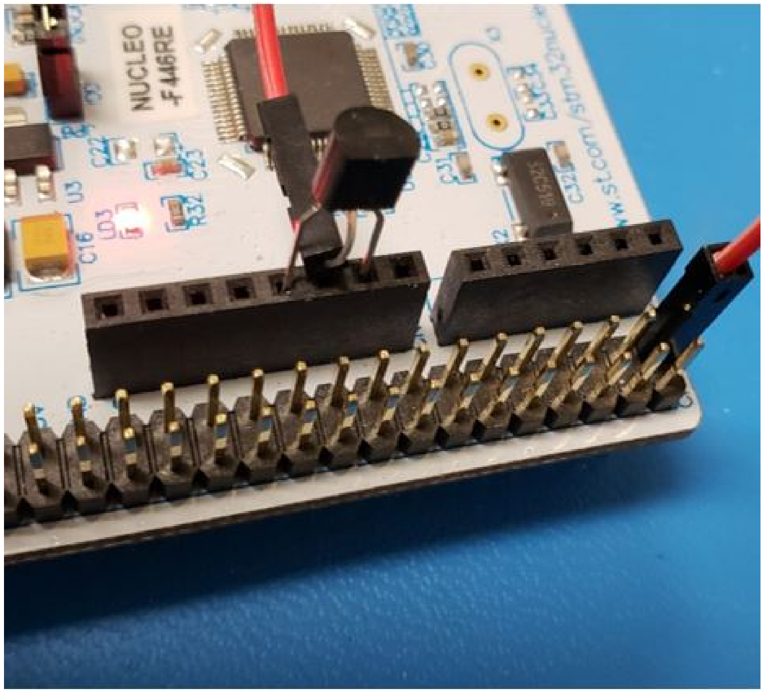

Figure 3.40: Temperature sensor connected to encoder microcontroller.

copies of the same bit to the controller. The transient periods can be seen in Fig. 3.41. The encoder determines the length of a single bit by monitoring the pulse given by the folding mechanism of the oscillator. The folding mechanism sends a pulse when the guard condition is met, and the controller is able to counter any switching of attractors with its own pulse. For a single given bit from the data input, the encoder has a set number of pulses it counts before moving to the next bit.

To send data from the sensor over the communication system, the sensor data is read through an 8-bit ADC on the encoder microcontroller. A block diagram of the encoder, along with its interfaces to the other components on the transmit side of the communication system, is shown in Fig. 3.42. The program used for the encoder is included in Appendix C, and a block diagram of the decoder, along with its interfaces with the other components on the receive side is shown in Fig. 3.43. The ADC was set up in the same manner as the ADC on the digital matched filter, using an external timer configured in output-compare mode to trigger an event. The external timer was configured to trigger a conversion once every

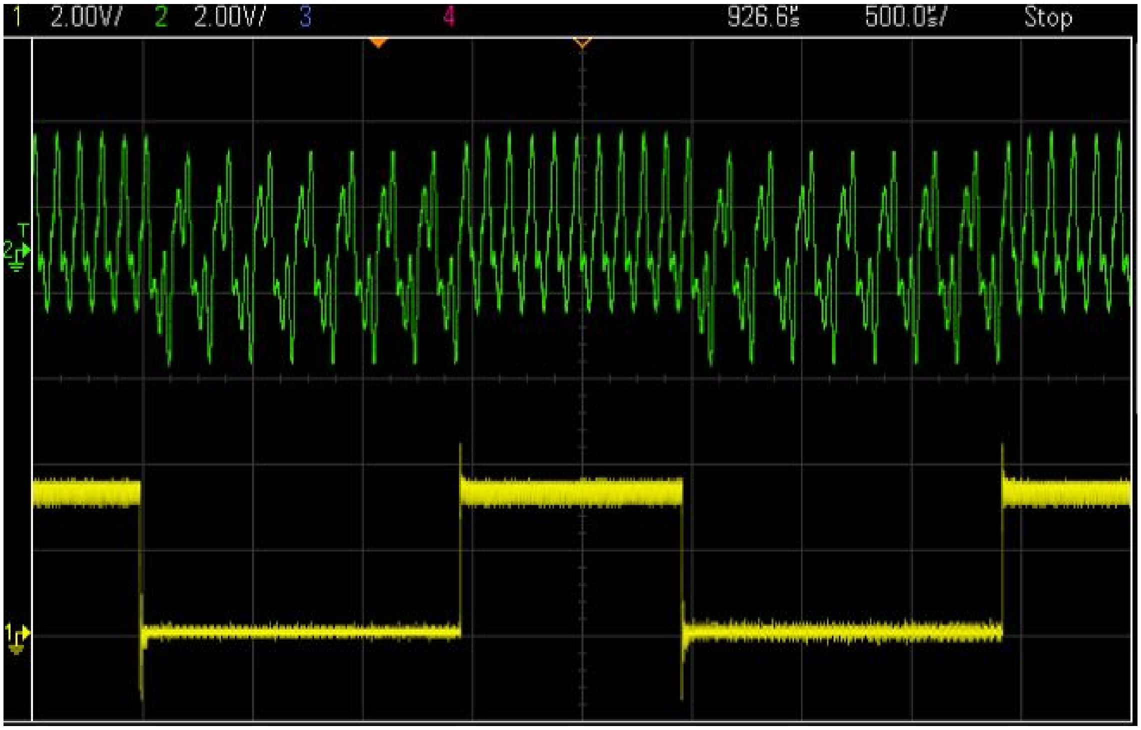

Figure 3.41: Controller input (yellow) and oscillator output given the binary sequence (green). Note the transient periods between oscillator states.

two seconds, allowing for easy viewing and debugging on an oscilloscope. Once a conversion of the temperature sensor data is complete, the conversion complete callback function is again used to perform the operations necessary to sending the data. The data from the temperature sensor is stored as an 8-bit value, and the value is parsed into an array of length eight, with each element containing an individual bit. Each element of the array is used to drive a GPIO pin to the appropriate logic level. The GPIO pin is driven to that level until the folding mechanism reaches the specified value of pulses, and the encoder moves to the next bit. This process occurs until all eight bits are sent. The microcontroller also sends the sampled temperature data to a computer via UART for verification purposes. To notify the receive side of incoming data, a start sequence consisting of directly sent high and low pulses are sent.

Initially, a midpoint sampling scheme was used to recover the serial data, but due to the chaotic nature of the system, the length of each individual period of unstable growth

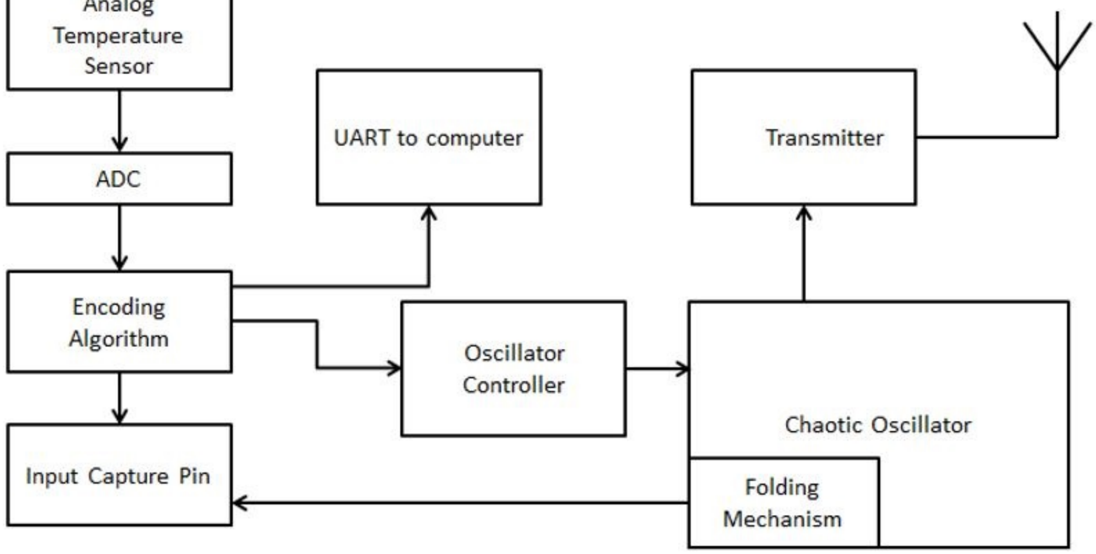

Figure 3.42: Block diagram of the encoder and its interfaces with the oscillator and controller.

in the system varies. This variation results in an inconsistent period for each bit, resulting in false data being recovered by the midpoint sampler. For this reason, an ad-hoc scheme was introduced to increase the reliability of the system. This solution can be described as "constant off-time pulse width modulation". The scheme treats a data one as "1-0" and a data zero as "1-1-0". This bypasses the small inconsistencies inherent to the chaotic system by rendering them relatively small compared to the variation in pulse length.

3.4.2 Decoder

Due to the changes in behavior of the transmit side of the system, the matched filter output resembles binary frequency shift-keying (BFSK) [48]; however, standard methods of decoding a BFSK signal are not reliable, as the frequencies shown are not consistent enough to be modulated and low-pass filtered successfully. The decoder scheme reads the matched filter output into a GPIO port, configured either for analog or digital data, depending on which matched filter is used. If the analog matched filter is used, the ADC is configured to sample the signal at an extremely high speed, using direct-memory access to attain the

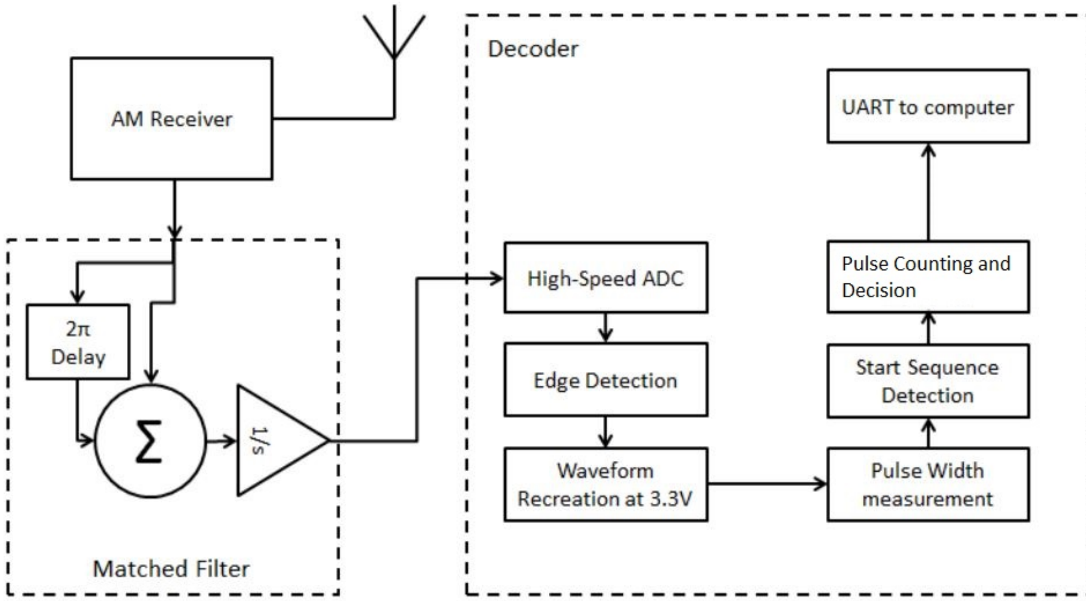

Figure 3.43: Block diagram of the decoder and its interface with the receive side.

highest speed possible. This signal is then recreated at digital logic levels by comparing the current ADC value with the previous value, and if they differ greatly, meaning a change in logic level is detected, a GPIO pin configured for digital output is updated to the appropriate level. This digital signal is then fed into another GPIO pin configured for digital input, and pulse length is measured using a configured timer. If the digital matched filter is used, this stage is bypassed and the digital matched filter output is fed directly to the digital input GPIO pin. The pulse length is then compared to a threshold, and a decision is made on the corresponding logic level. These pulses are counted until a logic zero is received, corresponding to the constant off-time portion of the encoding scheme. At this time, the number of pulses counted is compared to a threshold number, and a decision is made. If the pulse count is greater than the threshold, the decoder assumes the sequence 1-1-0 was sent, and an element of an array of length eight is set to zero. If the pulse count is less than the threshold, a 1-0 sequence is assumed and the element is set to one. The program used to perform the decoding operations is included in Appendix D.

The start sequence is treated in a different manner to the actual data, allowing for a simple implementation that requires next to no change in the encoder parameters. Each individual pulse of both the one and zero of the start sequence is counted. On the first zero pulse, the number of counted one pulses is compared to a set number. If they are equal, the decoder begins counting the zero pulses. If they are not equal, the zero pulses are not counted and the ones counter is reset. At the point of transition back to ones pulses, the number of zero pulses is counted and compared to a set number. If they are unequal, the following data transmission is ignored. If they are equal, the encoder begins accepting data. Since the number of pulses corresponding to each component of the start sequence will remain consistent, each can be compared to an exact number, greatly reducing the possibility of an erroneous start detection. This is further reduced by the fact that the start sequence high and low lengths are unequal to that of any encoded data.

After eight decisions are made, the decoder stops accepting data and prepares to send it to a computer via the UART. This is done by operating on the data in the opposite manner as the encoder–each element of the array containing the decoded data is added to a single 8-bit variable using a loop containing a logic AND operation with a left shift corresponding to the loop count value. This allows the variable to be used as an argument in a generated UART transmit function.

Chapter 4

Testing

To demonstrate the function of the communication system, a hardware test was performed. Initially, an AM transmitter and receiver were designed to modulate and demodulate the signal at 2.3 GHz, however, this system proved to be unreliable in practical use, as components used caused large nonlinearities in the received signal, overwhelming the matched filter. Additionally, the receiver contained no automatic gain control (AGC) system, and reliability was inconsistent at various distances. To remedy this, an off-the-shelf FM transmitter and receiver were introduced. This change greatly improved the quality of the signal and made evaluation and debugging more manageable. Three microcontrollers were required for this system: one for the encoder, one for the decoder, and one for the digital matched filter. It was intended to integrate the decoder and matched filter onto a single microcontroller, but the combined computational intensity of both the matched filter algorithm and the decoder algorithm were too great for one microcontroller to handle.

To ensure optimal functionality of the system, various parameters of the system require tuning. Generally, the tuning process begins with the oscillator. If the oscillator output is periodic, the potentiometer connected to the source of the MOSFET must be adjusted until chaotic behavior appears, either by inspection of the phase space or the time-domain output. (interestingly, it is possible to hear the oscillator and adjust it until the sound resembles that of an out-of-tune analog television.) Once the oscillator exhibits the desired behavior, $s(t)$ is examined and adjusted via potentiometers controlling magnitude and level shift to ensure operation at ± 1V. Once all oscillator parameters are as desired, the encoder and controller are connected, and the oscillator output is examined with a test pattern of 1-0-1-0. The controller and encoder rarely require adjustment, and may only need to be adjusted if a parameter on the receive side requires a change in the encoding process. After the transmit side is functioning properly, the received waveform is observed, along with the matched filter output. Often, small changes in the behavior of the oscillator will require an adjustment of $R\eta$ on the analog filter, or an adjustment of the threshold on the digital filter. The pulse length threshold on the decoder requires occasional adjustment due to the variations in the unstable growth rate in the oscillator.

The test was conducted in an indoor environment, with the transmitter and receiver spaced roughly one meter apart. Radio-frequency and baseband noise were observed in the environment by an external software-defined radio, indicating a good environment to demonstrate the system's desired performance in a noisy space. The testing environment is shown in Fig. 4.1. It was assumed that the sensitivity of the oscillator to initial conditions, and the small parameter changes in electronic components given varying external conditions, would cause a change in reliability if the environment was varied. This was tested by conducting two bit-error rate (BER) tests [52], one with the room's thermostat set to the maximum, and the other with the thermostat set to the minimum. Typically, the system is operated using two separate computers to verify the congruence of the transmitted data and the received data. For this test, a Python program was written to send a randomized ASCII character. Both the encoder and decoder UART outputs are connected to one computer, and the test program can determine if the character transmission is successful. Additionally, the program compares the 8-bits of the transmitted and received values individually, and counts the number of incorrect bits to constitute a true BER test. An oscilloscope capture of the overall function of the system is shown in Fig. 4.2.

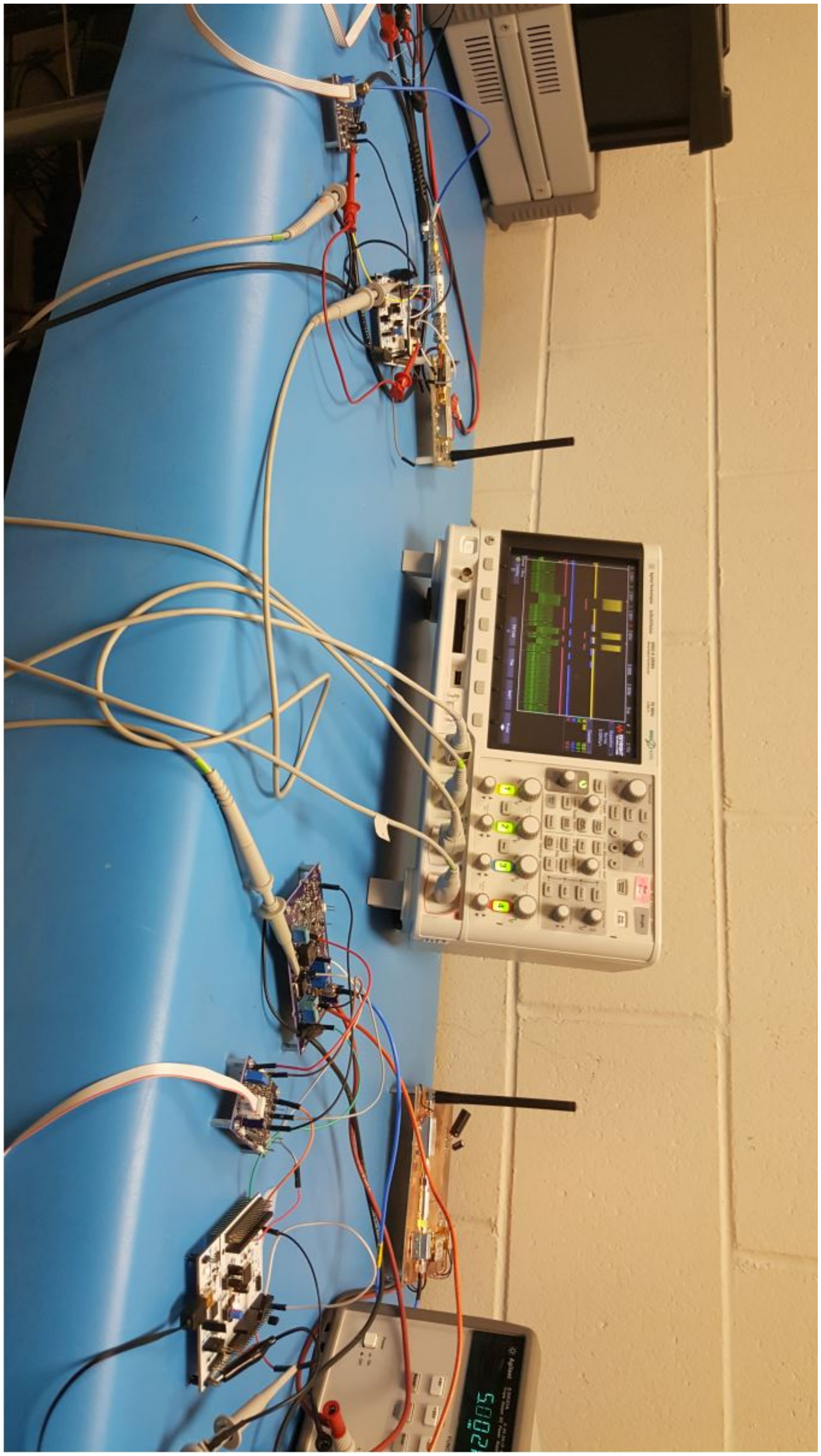

Figure 4.1: Testing environment for the full communication system. Transmit side (right) and receive side (left).

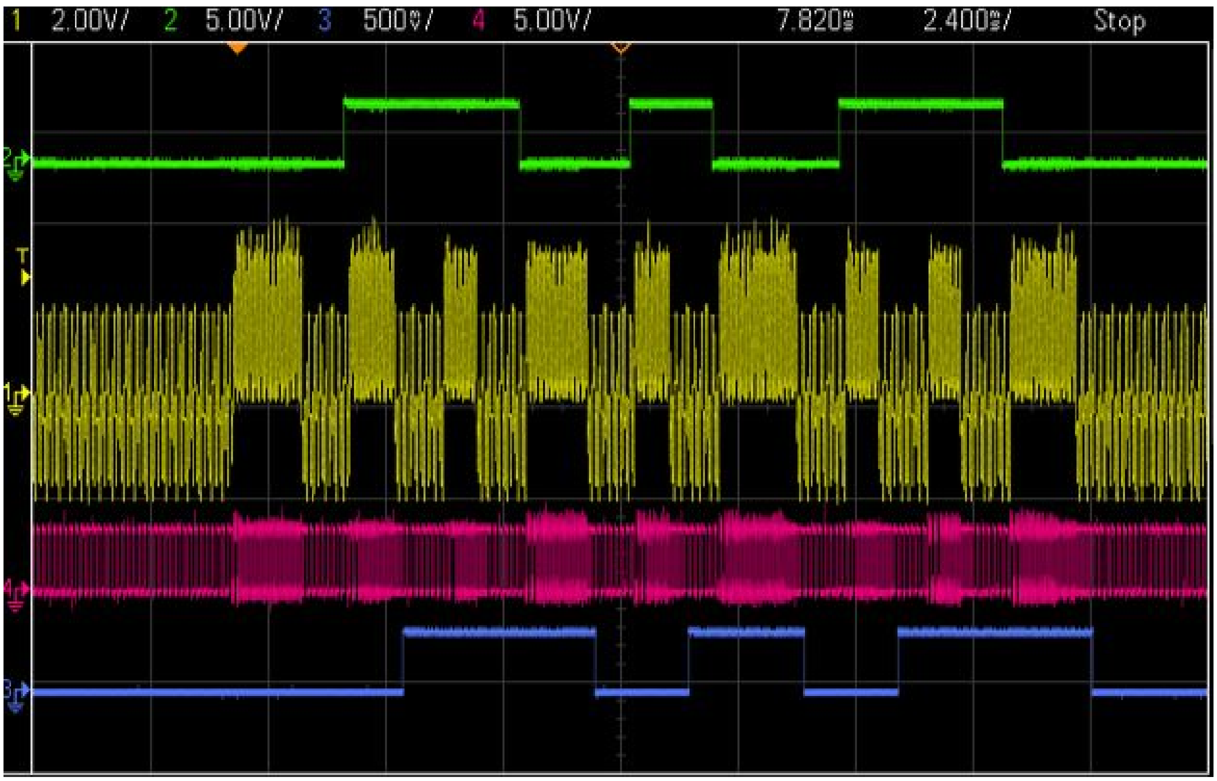

Figure 4.2: Oscilloscope capture of the function of the communication system. Oscillator output (yellow), binary data sent serially to oscillator controller (green), matched filter output (pink), and decoded binary data (blue).

Chapter 5

Results

The test was conducted twice, with the temperature sensor reading an average temperature of 25.9 degrees C and 22.1 degrees C, respectively. Ideally, the temperature variance would have been greater, but the HVAC system in the building in which the test was performed is only able to produce heating and air conditioning separately. However, the temperature variance that was present was still enough to show evidence for claims that varying the environment would cause changes in the system. The results of both BER tests are shown in Fig 5.1. The results show that the system is able to achieve a very low error rate, even without any formal error correction. Additionally, no NULLs were detected, indicating perfect reliability in the start sequence detection scheme; a false start sequence detection would result in either a decoded NULL or a timeout, in which case a NULL is sent.

BER Test Results	Number of Characters	Avg. Temperature (°C)	Success Rate
Test 1	1984	25.9 °C	99.24%
Test 2	1597	22.1	94.74%

Figure 5.1: Bit-error rate test results from both tests.

Chapter 6

Conclusion

Presented is a hardware demonstration of a wireless communication system that takes advantage of the spread spectrum characteristics of an exactly solvable chaotic oscillator operating at a baseband frequency of 18.4 kHz. The system is based on previous work that found an exact analytical solution and an equation for a matched filter to a chaotic system defined as a linear convolution of a fixed basis function and a discrete-time function. This communication system has been shown to have the ability to transmit and receive data from a temperature sensor with a low bit-error rate in a relatively uncontrolled environment. The system was implemented using an FM transmitter and an FM receiver, a controller, an oscillator, and a matched filter, all realized on custom-designed PCBs. The digital matched filter, and all encoding and decoding was implemented on STM32F446 breakout boards, which are based on the ARM Cortex M4 microcontroller architecture. All programs used were written in embedded C, and they take advantage of the KEIL toolchain and the abstraction layers it provides. Programs were written so that an individual with little software experience could more easily grasp their function and follow a straightforward manual to operate the entire system. Hardware testing shows that the received binary data from the temperature sensor can be encoded into two controlled orbits of the chaotic oscillator. It was found through examination of the received signal that noise was present in the environment; however, the matched filter was still able to produce the filtered signal in a manner that could be easily decoded. An ad-hoc encoding and decoding scheme was implemented to take advantage of observed properties of the system. This scheme proved to be very reliable, as no false data was recorded by the BER test (meaning no NULLs, showing no falsely detected start sequence). As expected, varying the ambient temperature caused the system to go out of tune, and the reliability decreased. This is likely caused by the matched filter threshold value becoming inaccurate, or by the decoder pulse threshold becoming inaccurate when compared with the incoming signal. One notable drawback to the implementation of this system is the data rate. As the oscillator operates at a baseband of 18.4 kHz, and multiple copies of the same bit along with a custom encoding scheme are both required to increase accuracy to acceptable levels, the current system's use is largely limited to remote sensing applications where fast data updates are not required. Overall, the system was able to function as the theoretical background indicated.

Chapter 7

Future Work

Future work will almost certainly involve increasing the baseband frequency of the chaotic oscillator to increase data transmission rate. This will likely involve further development of either the single-transistor oscillator or the HF oscillator, with a priority being tunability. Additionally, it may be possible to implement an automatic tuning system in either software or hardware to tune the oscillator before any transmission begins. Eliminating the problem of the transient periods between oscillator states is paramount to increasing the data rate. This may be achieved by redesigning the controller around a different control scheme–one that is able to remedy the issue observed in the phase-space plots of the controlled oscillator. This issue appears to be that the oscillator controller is unable to perturb the oscillator at the most desirable point, which is as near as possible to the ideal "fold" point. It was observed that the controller would occasionally perturb the oscillator near that point, and the transient period was not observable. There is also room for improvement in the trajectories to which the controller steers the oscillator to more closely resemble the ideal function. This would likely include a ground-up reworking of the encoding and decoding schemes, which take advantage of the deficiencies in the controller.

The receive side can likely be reduced in footprint by implementing all digital operations on an FPGA. This may allow for integration of the currently separate matched filter and decoder (when using the digital matched filter), as the ability of FPGAs to perform operations in parallel would likely yield improvements in performance. The system would certainly see an improvement in reliability if a standard error reduction method, such as a parity bit [51], was introduced.

Bibliography

[1] Wiener, N., *The Homogeneous Chaos*, American Journal of Mathematics, 1938.

[2] Grad, H., *Principles of the Kinetic Theory of Gases*, Thermodynamics of Gases, 1958.

[3] O'Toole, J. T.; Dahler, J. S., *On the Kinetic Theory of a Fluid Composed of Rigid Spheres*, J. Chem. Phys., 1960.

[4] Ergun, Salih,*Regional random number generator from a cross-coupled chaotic oscillator*, Circuits and Systems, 2011 IEEE 54th International Midwest Symposium on, pp. 1-4, Aug. 2011.

[5] Beal, A. N.; Bailey, J. P.; Hale, S. H.; Dean R. N.; Hamilton, M.; Tugnait, J. K.; Hahs, D. W.; Corron, N. J.,, *Design and simulation of a high frequency exact solvable chaotic oscillator*, Military Communications Conference, 2012 - MILCOM 2012 , vol., no., pp.1,6, Oct. 29 2012-Nov. 1 2012.

[6] Corron, N. J.; Blakely, J. N.; Pethel, S. D., *Communicating with exactly solvable chaos*, Henry Leung, ed., Chaotic Signal Processing, SIAM, pp. 48-84, 2013.

[7] Venkatasubramanian, V.; Leung, H., *A robust chaos radar for collision detection and vehicular ranging in intelligent transportation systems*, Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on, pp. 548-552, Oct. 2004.

[8] Zhang, Wenfang; He, Dake *Chaotic secure communication based on discrete-time chaos noise generator*, Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on, pp. 935-939, 27-29 Aug. 2003.

[9] Lau, Francis C. M.; Tse, Chi K.*Study of Anti-Jamming Capabilities of Chaotic Digital Communcation Systems*, Information Theory and Its Applications, International Symposium on, pp. 65-68, 2002.

[10] Saito, T.; Fujita, H.,*Chaos in a Manifold Piecewise Linear System*, Electronics and Communications Journal, vol. 64, no. 10, pp. 9-17, Oct. 1981.

[11] Corron, Ned J.,*An Exactly Solvable Chaotic Differential Equation*, Dynamics of Continuous, Discrete and Impulsive Systems, vol. 16, pp. 777-788, 2009.

[12] Corron, Ned J.; Blakely, Jonathan N.,*Exact folded-band chaotic oscillator*, Chaos, 22, 023113, 2012.

[13] Coullet, P.; Elphick, C.; Repaux, D., *Nature of Spatial Chaos*, Phys. Rev. Lett., 1987.

[14] Deakin, M., *Catastrophe and Chaos: Mathematical Views of the World*, Meanjin, 1978.

[15] May, R. M., *Nonlinear Phenomena in Ecology and Epidemiology*, Annals of the New York Academy of Sciences, 1980.

[16] Sinai, Y. G., *Dynamical Systems with Elastic Reflections. Ergodic Properties of Dispersing Billiards.*, Uspekhi Mat. Nauk, 1970.

[17] Oono, Y.; Takahashi, Y., *Chaos, External Noise and Fredholm Theory*, Progress of Theoretical Physics, 1980.

[18] Pesin, Y. B., *Characteristic Lyapunov Exponents and Smooth Ergodic Theory*, Uspekhi Mat. Nauk, 1977.

[19] Rosser, J. B., *Chaos Theory and the New Keynsian Economics*, The Manchester School of Economic and Social Studies, University of Manchester, 1990.

[20] Blakely, Jonathan N.; Corron, Ned J.,*Ambiguity in Range-Doppler Determination Using Waveforms of a Solvable Chaotic Oscillator*, Signal Processing, vol. 104, pp. 136-142, Nov. 2014.

[21] Corron, Ned J.; Stahl, Mark T.; Harrison, R. C.,*Acoustic Detection and Ranging Using Solvable Chaos*, International Conference on Theory and Application in Nonlinear Dynamics, vol. 23, pp. 213-223, Dec. 2013.

[22] Corron, Ned J.; Blakely, Jonathan N.; Stahl, Mark T.,*A Matched Filter For Chaos*, Chaos, 20, 023123, 2010.

[23] Rössler, O. E.,*An Equation for Continuous Chaos*, Physics Letters A, vol. 57, no. 5, 1976.

[24] Lorenz, Edward N.,*Deterministic Nonperiodic Flow*, Journal of Atmospheric Sciences, vol. 20, pp. 130-141, 1963.

[25] Corron, Ned J., *An Exactly Solvable Chaotic Differential Equation*, Dynamics of Continuous, Discrete and Impulzive Systems, A: Mathmatical Analysis 16, Watam Press, pp. 777-788, 2009.

[26] Ott, Edward, *Chaos in Dynamical Systems*, Cambridge University Press, second edition, 2002.

[27] Reartes, Walter *The Homoptopy Analysis Method in the Search for Periodic Orbits*, Actas Del XII Congreso Dr. Antonio A. R. Monteiro, 2013.

[28] Rodakoviski, Rodrigo B.; Dias, Nelson L., *Statistical Description of Rayleigh-Bénard Convection Using the Lorenz Equations*, American Journal of Environmental Engineering, 2018.

[29] Kolmogorov, A. N., *The Local Structure of Turbulence in Incompressible Viscous Fluid for Vry Large Reynolds Numbers*, Proceedings of the Royal Society A, 1941.

[30] Cartwright, M. L.; Littlewood, J. E., *On Non-Linear Differential Equations of the Second Order*, Journal of the London Mathematical Society, 1945.

[31] Feigenbaum, M., *Quantitative Universality for a Class of Nonlinear Transformations*, Journal of the London Mathematical Society, 1945.

[32] Boeing, J., *Visual Analysis of Nonlinear Dynamical Systems: Chaos, Fractals, Self-Similarity and the Limits of Prediction*, Systems, 2016.

[33] Li, T. Y.; Yorke, J. A., *Period Three Implies Chaos*, The American Mathematical Monthly, Vol. 82, No. 10, 1975.

[34] Katsura, S.; Fukuda, W., *Exactly Solvable Models Showing Chaotic Behavior*, Physica A: Statistical Mechanics and its Applications, 1985.

[35] Devaney, R. L., *An Introduction to Chaotic Dynamical Systems*, Addison-Wesley, 1989.

[36] Werner, F. T.; Rhea, B. K.; Harrison, R. C.; Dean, R. N., *Electronic implementation of a Practical Matched Filter for a Chaos-based Communication System*, Chaos, Solitons, and Fractals, 2017.

[37] Ott, E; Grebogi, C.; Yorke, J. A., *Controlling Chaos*, Phys. Rev. Lett 64, 1990.

[38] Turin, G., *An introduction to Matched Filters*, IRE Transactions on Information Theory, 1960.

[39] Valsakumar, M. C.; Satyanarayana S. V. M.; Sridhar V., *Signature of Chaos in Power Spectrum*, Pramana Journal of Physics, 1997.

[40] Hayes, S., *Chaos from Linear Systems: Implications for Communicating with Chaos, and the Nature of Determinism and Randomness*, Journal of Physics Conference series 23, 2005.

[41] Hayes, S.; Grebogi, C.; Ott, E., *Communicating with Chaos*, Phys. Rev. Lett 70, 1993.

[42] Corron, N. J.; Stahl, M. T.; Blakely, J. N, *Exactly Solvable Chaotic Circuit*, International Symposium on Circuits and Systems (ISCAS) 2010.

[43] Corron, N. J.; Pethel, S. D.; Hopper, B. A.;, *Controlling Chaos with Simple Limiters*, Phys. Rev. Lett 84, 2000.

[44] Childress, C.; Gilbert, A. D., *Stretch, Twist, Fold: The Fast Dynamo*, Springer, 1995.

[45] Rhea, B. K.; Beal, A. N.; Werner, F. T.; Dean, R. N., *Chaotic Oscillator Implementation Based on an Exactly Solvable Piecewise Linear Chaotic System Intended for Communication System Applications*, IMAPS 2017.

[46] Rhea, B. K.; Harrison, R. C.; Whitney, D. A.; Werner, F. T., *Hardware Implementation of Chaos Control Using a Proportional Feedback Controller*, ICAND 2018.

[47] Grebogi, C.; Ott, E.; Yorke, J. A., *Crises, Sudden Changes in Chaotic Attractors, and Transient Chaos*, Bell Labs Technical Journal, 1963.

[48] Bennett, W. R.; Rice, S. O., *Spectral Density and Autocorrelation Functions Associated with Binary Frequency-Shift Keying.*, Bell Labs Technical Journal, 1963.

[49] Nekeoi, F.; Kavian, Y.; Strobel, O., *Some Schemes of Realization Digital FIR Filters on FPGA for Communication Applications*, Microwave and Telecommunication Technology, 2010.

[50] Damian, C.; Lunca, E., *A low-area FIR Filter for FPGA Implementation.*, IEEE, 2011.

[51] Davey, M. C., *Error-Correction using Low-Density Parity-Check Codes*, University of Cambridge, 2000.

[52] Jeruchim, M., *Techniques for Estimating the Bit Error Rate in the Simulation of Digital Communication Systems*, IEEE Journal on Selected Areas in Communications, 1984.

Appendices

Appendix A

MATLAB program for digital matched filter demonstration

```
%If you're using an outside signal file you shouldn't need lines 1 through
%5 or the for loop in line 10. just make x in line 6 equal to the data you want to filter. Also make
%sure all of the array values match up
%f=18e3;
%fs=f*4;%sample at 4 times the frequency of the signal
%nCyl=40; %generate five cycles of sinusoid
%t=0:1/fs:nCyl*1/f; %time index
%x=.2+.05*cos(2.6*pi*f*t)+.3*sin(10*pi*f*t)+.05*sin(12.3*pi*f*t)+.05*sin(25*pi*f*t); %change this to input file or whatever
%for k=1:161 %this function generates the two states for the wave above. Comment out if using outside data.
   % if k<20
     %   x(k)=x(k)+.5;
    %elseif k<55
    %   x(k)=x(k)+.3;
    %elseif k<70
    %    x(k)=x(k)+.7;
   % elseif k<85
    %    x(k)=x(k)+.3;
   % elseif k<128
   %     x(k)=x(k)+.5;
  %  elseif k<156
   %     x(k)=x(k)+.3;
   % elseif k<180
   %     x(k)=x(k)+.7;
  %  else
```

```
  %      x(k)=x(k)-10;
  %  end
%end
%plot(t,x);
t = 1:401;
%plot(t,simout)
KI = 1; % this changes the amount of separation of the two states of the
   output
%plot(t,x)
title('Continuous sinusoidal signal');
xlabel('Time(s)');
ylabel('Amplitude');
%delay signal 360 degrees (was 180, changed according to Frank's paper on
%analog matched filter
%4 samples taken per cycle, so a delay of 360 is 4 samples
x = 1241*((simout')/11+1.2);
for h=1:401
    if h<5
    y(h)=x(h);
    else
        y(h)=x(h-4);
    end
end
plot(t,y,t,x)
pause;
%add negative of delayed signal back into original. This detects a change
%in value of the two samples. If the two values are the same or close, the
%output of the adder will not change much.
xy=x-y;
plot(t,xy/4,t,x-1100)
pause;
for j=2:401 %this is the integrator (LPF) stage. Uses trapezoidal Riemann
   sums
```

```
    x1 = xy(j);
    if j==1
        x2 = x1;
    else
        x2 = xy(j-1);
    end
    sum1(j) = (x1+x2);
    sum2(1)=0;
    sum2(j) = (sum2(j-1)+sum1(j))*KI;
    sumout(1) = 0;
    sumout(j) = (sum2(j)+sum2(j-1));
    if 1000+sum2(j)>0
        binout(j)=1;
    else
        binout(j)=0;
    end
end
plot(t,simout,t,sim,t,binout+2.5)
```

Appendix B

Program for Software-Defined Matched Filter

NOTE: This program was written in such a way that an individual unfamiliar with programming can make necessary changes to maintain the function of the communication system. Attention to efficiency and elegance yields to ease of access.

```
/**
  ******************************************************************************
  * File Name          : main.c
  * Description        : Main program body
  ******************************************************************************
  *
  * COPYRIGHT(c) 2017 STMicroelectronics
  *
  * Redistribution and use in source and binary forms, with or without modification,
  * are permitted provided that the following conditions are met:
  *   1. Redistributions of source code must retain the above copyright notice,
  *      this list of conditions and the following disclaimer.
  *   2. Redistributions in binary form must reproduce the above copyright notice,
  *      this list of conditions and the following disclaimer in the documentation
  *      and/or other materials provided with the distribution.
  *   3. Neither the name of STMicroelectronics nor the names of its contributors
  *      may be used to endorse or promote products derived from this software
```

```
  *      without specific prior written permission.
  *
  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS " AS IS"
  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
  *
  ******************************************************************************
  */
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx_hal.h"

/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

```

```
/* Private variables ----------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

DAC_HandleTypeDef hdac;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

/* USER CODE BEGIN PV */
/* Private variables ----------------------------------------------------------*/
uint32_t ADC_data[1];
int valueReady = 0;
int firstRun1 = 1;
int firstRun2 = 1;
int firstRun3 = 1;
int firstRun4 = 1;
int firstRun5 = 1;
int firstRun6 = 1;
int firstRun7 = 1;
int firstRun8 = 1;
int32_t sum[1];
int32_t lastSum[1];
int32_t store1[1];
int32_t store2[1];
int32_t store3[1];
int32_t store4[1];
int32_t store5[1];
int32_t store6[1];
int32_t store7[1];
int32_t store8[1];
```

```
int32_t lastxint2[1];
int lastStore = 4;
int firstSum = 1;
int32_t KI1 = 1; //integrator constant
int32_t KI2 = 1;
int32_t xint1[1];
int32_t xint2[1];
int32_t currentval[1];
int32_t xout[1];
uint32_t dacout[1];
int32_t timeavg_arr[36]; // # values used in time average is 50 with 1
    extra to shift, use 26 for 4 samples per period
int32_t avgval = 0;
int32_t avgsum = 0;
int num = 1;
int maxnum = 35; //use 35 for 4 samples per period, use 50 for 8 samples
    per period
int32_t avgout[1];
uint32_t dacavgout[1];
int shiftvalue = 3750;
uint32_t maxout[1];
uint32_t minout[1];
int currentstate = 0;
/* USER CODE END PV */

/* Private function prototypes
    -----------------------------------------------*/
void SystemClock_Config(void);
void Error_Handler(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);
static void MX_DAC_Init(void);
```

```
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);


/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/

/* USER CODE END PFP */

/* USER CODE BEGIN 0 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1)
{
  ADC_data[0] = HAL_ADC_GetValue(hadc1);
    if (lastStore == 4) //use this with 4 samples/period
    //if (lastStore == 8) //use this with 8 samples/period
    {
      lastStore = 1;
      if (firstRun1 == 0)
      {
        currentval[0] = ADC_data[0]/2; //divide adjusted to try and help with noise getting into signal somehow -- grounding issue, supply, etc
        sum[0] = currentval[0] - store1[0]; /* add current read value to negative of delayed value */
        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
        xint2[0] = xint1[0]+lastxint2[0];
        lastSum[0] = sum[0];
        xout[0] = xint2[0];
      }
      store1[0] = currentval[0];
```

```
      firstRun1 = 0;
      lastxint2[0] = xint2[0];
    }
    else if (lastStore == 1)
    {
      lastStore = 2;
      if (firstRun2 == 0)
      {
        currentval[0] = ADC_data[0]/2;
        sum[0] = currentval[0] - store2[0]; /* add current read value to
            negative of delayed value */
        xint1[0] = (sum[0] + lastSum[0])/KI1/KI2; //integrator stage
        xint2[0] = xint1[0]+lastxint2[0];
        lastSum[0] = sum[0];
        xout[0] = xint2[0];
      }
      store2[0] = currentval[0];
      firstRun2 = 0;
      lastxint2[0] = xint2[0];
    }
    else if (lastStore == 2)
    {
      lastStore = 3;
      if (firstRun3 == 0)
      {
        currentval[0] = ADC_data[0]/2;
        sum[0] = currentval[0] - store3[0]; /* add current read value to
            negative of delayed value */
        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
        xint2[0] = xint1[0]+lastxint2[0];
        lastSum[0] = sum[0];
        xout[0] = xint2[0];
      }
```

```
      store3[0] = currentval[0];
      firstRun3 = 0;
      lastxint2[0] = xint2[0];
    }
    else if (lastStore == 3)
    {
      lastStore = 4;
      if (firstRun4 == 0)
      {
        currentval[0] = ADC_data[0]/2;
        sum[0] = currentval[0] - store4[0]; /* add current read value to negative of delayed value */
        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
        xint2[0] = xint1[0]+lastxint2[0];
        lastSum[0] = sum[0];
        xout[0] = xint2[0];
      }
      store4[0] = currentval[0];
      firstRun4 = 0;
      lastxint2[0] = xint2[0];
     }
//  uncomment lines 181-245 for 8 samples per period
//    else if (lastStore == 4)
//    {
//      lastStore = 5;
//      if (firstRun5 == 0)
//      {
//        currentval[0] = ADC_data[0]/4;
//        sum[0] = currentval[0] - store5[0]; /* add current read value to negative of delayed value */
//        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
//        xint2[0] = xint1[0]+lastxint2[0];
//        lastSum[0] = sum[0];
```

```
//        xout[0] = xint2[0];
//      }
//      store5[0] = currentval[0];
//      firstRun5 = 0;
//      lastxint2[0] = xint2[0];
//     }
//    else if (lastStore == 5)
//    {
//      lastStore = 6;
//      if (firstRun6 == 0)
//      {
//        currentval[0] = ADC_data[0]/4;
//        sum[0] = currentval[0] - store6[0]; /* add current read value to negative of delayed value */
//        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
//        xint2[0] = xint1[0]+lastxint2[0];
//        lastSum[0] = sum[0];
//        xout[0] = xint2[0];
//      }
//      store6[0] = currentval[0];
//      firstRun6 = 0;
//      lastxint2[0] = xint2[0];
//     }
//    else if (lastStore == 6)
//    {
//      lastStore = 7;
//      if (firstRun7 == 0)
//      {
//        currentval[0] = ADC_data[0]/4;
//        sum[0] = currentval[0] - store7[0]; /* add current read value to negative of delayed value */
//        xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
//        xint2[0] = xint1[0]+lastxint2[0];
```

```
//         lastSum[0] = sum[0];
//         xout[0] = xint2[0];
//       }
//       store7[0] = currentval[0];
//       firstRun7 = 0;
//       lastxint2[0] = xint2[0];
//      }
//     else if (lastStore == 7)
//     {
//       lastStore = 8;
//       if (firstRun8 == 0)
//       {
//         currentval[0] = ADC_data[0]/4;
//         sum[0] = currentval[0] - store8[0]; /* add current read value to
    negative of delayed value */
//         xint1[0] = (sum[0] + lastSum[0])*KI1/KI2; //integrator stage
//         xint2[0] = xint1[0]+lastxint2[0];
//         lastSum[0] = sum[0];
//         xout[0] = xint2[0];
//       }
//       store8[0] = currentval[0];
//       firstRun8 = 0;
//       lastxint2[0] = xint2[0];
//      }


  dacout[0] = xout[0]-shiftvalue; //manually scale value for the DAC
     output to look right (0 to 4096)
     if (num <= maxnum) //use array values 1 to 50 (omitting 0)
     {
       timeavg_arr[num] = xout[0];
       avgsum = avgsum + timeavg_arr[num];
       avgval = avgsum/num;
```

```
      num++;
    }
    else
    {
      for (int j=1; j<= maxnum; j++)
      {
        timeavg_arr[j-1] = timeavg_arr[j];
      }
      timeavg_arr[maxnum] = xout[0];
      avgsum = avgsum + timeavg_arr[maxnum] - timeavg_arr[0];
      avgval = avgsum/maxnum;
    }
    avgout[0] = avgval-shiftvalue;
    dacavgout[0] = avgout[0]; //avoid issues with unsigned int vs int

    if (dacavgout[0] > maxout[0]) //adjust max and min
    {
      maxout[0] = dacavgout[0];
    }
    if (dacavgout[0] < minout[0])
    {
      minout[0] = dacavgout[0];
    }

  HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, avgout[0]);
  //HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, dacout[0]);

    //NOTE: actual 1 and 0 are opposite what they appear to be on the DACOUT. This part takes care of that.

    //ANOTHER NOTE: The new transciever doesn't have the problem of flipping the 1 and 0. Program adjusted to compensate.

```

```c
    if (dacavgout[0] > minout[0] + 600)  //hardcoded number here is a tuned number. change based on what scope shows you
    {
      HAL_GPIO_WritePin(BINOUT_GPIO_Port, BINOUT_Pin, GPIO_PIN_SET);
      if (currentstate == 0) //check to see if there is a switch to a new state
      {
        minout[0] = dacavgout[0];//reset max and min
        maxout[0] = dacavgout[0];
        currentstate = 1; //change state
      }
    }
    if (dacavgout[0] + 675 < maxout[0]) //hardcoded number here is a tuned number. change based on what scope shows you
    {
      HAL_GPIO_WritePin(BINOUT_GPIO_Port, BINOUT_Pin, GPIO_PIN_RESET);
      if (currentstate == 1) //check to see if new state
      {
        minout[0] = dacavgout[0]; //reset max and min
        maxout[0] = dacavgout[0];
        currentstate = 0; //change state
      }
    }

}
//this part is supposed to simulate hardware MF
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM3 && currentstate == 1)
  {
    HAL_GPIO_TogglePin(BFSKOUT_GPIO_Port, BFSKOUT_Pin);
  }
  else if (htim->Instance == TIM4 && currentstate == 0)
```

```c
  {
    HAL_GPIO_TogglePin(BFSKOUT_GPIO_Port, BFSKOUT_Pin);
  }
}
/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration
      ------------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
      Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_TIM2_Init();
  MX_DAC_Init();
  MX_TIM3_Init();
  MX_TIM4_Init();

  /* USER CODE BEGIN 2 */
  HAL_TIM_OC_Start_IT(&htim2, TIM_CHANNEL_1);
```

```
  HAL_TIM_Base_Start_IT(&htim3); //TIMER 3 is for fast pulses (period = 20
      us)
  HAL_TIM_Base_Start_IT(&htim4); //TIMER 4 is for slow pulses (period = 50
      us)
  //FHAL_TIM_Base_Start_IT(&htim2);
  //HAL_ADC_Start_DMA(&hadc1, ADC_data, sizeof(uint32_t)); DON'T USE THIS

  HAL_ADC_Start_IT(&hadc1); // comment out to see if pulses work
  HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

  }
  /* USER CODE END 3 */

}

/** System Clock Configuration
*/
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;

```

```
  __HAL_RCC_PWR_CLK_ENABLE();

  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 144;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
  {
    Error_Handler();
  }

  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
```

```
  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}

/* ADC1 init function */
static void MX_ADC1_Init(void)
{

  ADC_ChannelConfTypeDef sConfig;

    /**Configure the global features of the ADC (Clock, Resolution, Data
       Alignment and number of conversion)
    */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISINGFALLING
     ;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

```

```
    /**Configure for the selected ADC regular channel its corresponding
       rank in the sequencer and its sample time.
    */
  sConfig.Channel = ADC_CHANNEL_10;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }

}

/* DAC init function */
static void MX_DAC_Init(void)
{

  DAC_ChannelConfTypeDef sConfig;

    /**DAC Initialization
    */
  hdac.Instance = DAC;
  if (HAL_DAC_Init(&hdac) != HAL_OK)
  {
    Error_Handler();
  }

    /**DAC channel OUT2 config
    */
  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_2) != HAL_OK)
  {
```

```
    Error_Handler();
  }

}

/* TIM2 init function */
static void MX_TIM2_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  //htim2.Init.Period = 489; //use this if you want 8 samples per period (more processing load on the controller)
  htim2.Init.Period = 978; //use this for 4 samples per period
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_OC_Init(&htim2) != HAL_OK)
  {
```

```
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
     HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    Error_Handler();
  }

  HAL_TIM_MspPostInit(&htim2);

}

/* TIM3 init function */
static void MX_TIM3_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

```

```
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 0;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 1679;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_OC_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
      HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
```

```c
  if (HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    Error_Handler();
  }

  HAL_TIM_MspPostInit(&htim3);

}

/* TIM4 init function */
static void MX_TIM4_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 4199;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
```

```
  if (HAL_TIM_OC_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) !=
      HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_1) !=
      HAL_OK)
  {
    Error_Handler();
  }

  HAL_TIM_MspPostInit(&htim4);

}

/** Configure pins as
        * Analog
        * Input
        * Output
        * EVENT_OUT
```

```
      * EXTI
    PA2   ------> USART2_TX
    PA3   ------> USART2_RX
*/
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_EVT_RISING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : BFSKOUT_Pin BINOUT_Pin */
  GPIO_InitStruct.Pin = BFSKOUT_Pin|BINOUT_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pins : USART_TX_Pin USART_RX_Pin */
  GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
```

```
  GPIO_InitStruct.Alternate = GPIO_AF7_USART2;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, BFSKOUT_Pin|BINOUT_Pin, GPIO_PIN_RESET);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  None
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler */
  /* User can add his own implementation to report the HAL error return state */
  while(1)
  {
  }
  /* USER CODE END Error_Handler */
}

#ifdef USE_FULL_ASSERT

/**
   * @brief Reports the name of the source file and the source line number
   * where the assert_param error has occurred.
```

```
  * @param file: pointer to the source file name
  * @param line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
     number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
       line) */
  /* USER CODE END 6 */

}

#endif

/**
  * @}
  */

/**
  * @}
*/

/************************ (C) COPYRIGHT STMicroelectronics *****END OF
   FILE****/
```

Appendix C

Program for Encoder

```
/**
  ******************************************************
  * File Name          : main.c
  * Description        : Main program body
  ******************************************************
  *
  * COPYRIGHT(c) 2017 STMicroelectronics
  *
  * Redistribution and use in source and binary forms, with or without modification,
  * are permitted provided that the following conditions are met:
  *   1. Redistributions of source code must retain the above copyright notice,
  *      this list of conditions and the following disclaimer.
  *   2. Redistributions in binary form must reproduce the above copyright notice,
  *      this list of conditions and the following disclaimer in the documentation
  *      and/or other materials provided with the distribution.
  *   3. Neither the name of STMicroelectronics nor the names of its contributors
  *      may be used to endorse or promote products derived from this software
  *      without specific prior written permission.
  *
  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
```

```
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 ******************************************************
 */
/* Includes ------------------------------------------------------------------*/
//This program WILL NOT WORK if not hooked up to oscillator and controller !!!!
#include "stm32f4xx_hal.h"

/* USER CODE BEGIN Includes */
#define OFFSET 30
#include "ManchesterEncode.h"
/* USER CODE END Includes */

/* Private variables ---------------------------------------------------------*/
```

```
ADC_HandleTypeDef hadc1;

DAC_HandleTypeDef hdac;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

UART_HandleTypeDef huart2;
UART_HandleTypeDef huart3;

/* USER CODE BEGIN PV */
/* Private variables
   ---------------------------------------------------------*/
static int TestArray[ONES_REPEATED + ZEROES_REPEATED];
  //To prevent transients on the oscillator, multiple copies of the same
     symbol must be sent in a row.
  //This sequence is actually 101010101010
int ArrLen = sizeof(TestArray) / sizeof(int);
int ArrInd = 0;
int NextSymbol = 85, CurrentSymbol = 85;
//85 = invalid input, prevents accidentaly entering loops

uint8_t ADC_data[1]; //for reading temp sensor
int ConvCplt = 0; //this is used to begin transmit function after ADC
   takes a reading
uint8_t ConvData[1];
uint8_t buffer1[100];


volatile int togglecheck = 1; // flag for when to update nextt transmitted
    symbol
```

```
char PolarityCheck = 'R'; //used to update current edge polarity (Rise or
    fall)
int stop_ADC = 1;
int CaseNumber = 0; //used to (eventually) switch between data entry modes
int EdgeCount = 0; //counts edges detected to determine when to transmit
    next symbol

volatile uint32_t ADCConvertedValue; //holds last converted value
uint32_t LastValue = 5000; //invalid value to prevent invalid start

uint16_t CaptureIndex = 0;
uint32_t CapValue0 = 0;
uint32_t CapValue1 = 0;
uint32_t CapValue2 = 0;
uint32_t ValueDiff = 0;

uint32_t i = 0;

volatile uint32_t EntryModeFlag = 0;
volatile uint32_t StartFlag = 0;

uint8_t view1 = 0; // for viewing values parsed from ADC data
uint8_t view2 = 0;
uint8_t view3 = 0;
uint8_t view4 = 0;
uint8_t view5 = 0;
uint8_t view6 = 0;
uint8_t view7 = 0;
uint8_t view8 = 0;
int switch1 = 1;
int switch2 = 1;
/* USER CODE END PV */
```

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
void Error_Handler(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_DAC_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_USART3_UART_Init(void);
static void MX_TIM4_Init(void);
static void MX_USART2_UART_Init(void);

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);


/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/
void TestCaseOutput(void); //output function for testcase mode
void ReadCustomData(void);
/* USER CODE END PFP */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */
  uint16_t testindex = 0;
  uint16_t tmp_tot = ONES_REPEATED + ZEROES_REPEATED;
```

```
  for(testindex=0; testindex < ONES_REPEATED; testindex++){
    TestArray[testindex] = 1;
  }
  for(testindex = ONES_REPEATED; testindex<tmp_tot; testindex++){
    TestArray[testindex] = 0;
  }
  /* USER CODE END 1 */

  /* MCU Configuration
     ----------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
     Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_DAC_Init();
  MX_TIM2_Init();
  MX_TIM3_Init();
  MX_USART3_UART_Init();
  MX_TIM4_Init();
  MX_USART2_UART_Init();

  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT(&htim2);  //time base for adc sampling rate
  //HAL_ADC_Start_DMA(&hadc1, &ADCConvertedValue, sizeof(uint32_t));
  //HAL_TIM_OC_Start(&htim2,TIM_CHANNEL_1);
```

```
  HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
  HAL_TIM_OC_Start_IT(&htim2, TIM_CHANNEL_1);
  HAL_ADC_Start_IT(&hadc1);

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    //HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_SET);
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
//    while(!StartFlag){
//    }
    //StartFlag = 0;
    switch(CaseNumber){
      case(0) :
        if (switch1 == 1)
        {
          HAL_ADC_Stop_IT(&hadc1); //stop the ADC to keep its callback and interrupt out of the way
          switch1 = 0;
          switch2 = 1;
        }
        while(togglecheck){ //if it's time to switch to next symbol THIS MAKES NO SENSE.
          // If togglecheck is commented in line 205, test case works but cannot get out of this loop.
          TestCaseOutput(); //transmit
        }
```

```
        break;
      case(1) :
        if (switch2 == 1)
        {
          HAL_ADC_Start_IT(&hadc1); //Start the ADC to get temperature
             readings
          switch1 = 1;
          switch2 = 0;
        }
        ReadCustomData();

        break;
      case(2):
        break;
      default :
        break;
    }
  }
  /* USER CODE END 3 */

}

/** System Clock Configuration
*/
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;

  __HAL_RCC_PWR_CLK_ENABLE();

  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
```

```
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSICalibrationValue = 16;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLM = 8;
RCC_OscInitStruct.PLL.PLLN = 168;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 2;
RCC_OscInitStruct.PLL.PLLR = 2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
  Error_Handler();
}

RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                            |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV4;
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
{
  Error_Handler();
}

HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
```

```
}

/* ADC1 init function */
static void MX_ADC1_Init(void)
{

  ADC_ChannelConfTypeDef sConfig;

    /**Configure the global features of the ADC (Clock, Resolution, Data
        Alignment and number of conversion)
    */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_8B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

    /**Configure for the selected ADC regular channel its corresponding
        rank in the sequencer and its sample time.
    */
  sConfig.Channel = ADC_CHANNEL_10;
  sConfig.Rank = 1;
```

```
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }

}

/* DAC init function */
static void MX_DAC_Init(void)
{

  DAC_ChannelConfTypeDef sConfig;

    /**DAC Initialization
    */
  hdac.Instance = DAC;
  if (HAL_DAC_Init(&hdac) != HAL_OK)
  {
    Error_Handler();
  }

    /**DAC channel OUT1 config
    */
  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }

}
```

```c
/* TIM2 init function */
static void MX_TIM2_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 2624;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 63999;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_OC_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
     HAL_OK)
```

```
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    Error_Handler();
  }

  HAL_TIM_MspPostInit(&htim2);

}

/* TIM3 init function */
static void MX_TIM3_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_IC_InitTypeDef sConfigIC;
  TIM_OC_InitTypeDef sConfigOC;

  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 41;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 0xffff;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
```

```
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_IC_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_OC_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
     HAL_OK)
  {
    Error_Handler();
  }

  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
```

```
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_2) !=
     HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TIMING;
  if (HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_3) !=
     HAL_OK)
  {
    Error_Handler();
  }

  HAL_TIM_MspPostInit(&htim3);

}

/* TIM4 init function */
static void MX_TIM4_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
```

```
  TIM_OC_InitTypeDef sConfigOC;

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 671;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 62499;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_TIM_OC_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) !=
      HAL_OK)
  {
    Error_Handler();
  }

  sConfigOC.OCMode = TIM_OCMODE_TIMING;
  sConfigOC.Pulse = 0;
```

```
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    Error_Handler();
  }

}

/* USART2 init function */
static void MX_USART2_UART_Init(void)
{

  huart2.Instance = USART2;
  huart2.Init.BaudRate = 9600;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }

}

/* USART3 init function */
static void MX_USART3_UART_Init(void)
{

```

```c
  huart3.Instance = USART3;
  huart3.Init.BaudRate = 115200;
  huart3.Init.WordLength = UART_WORDLENGTH_8B;
  huart3.Init.StopBits = UART_STOPBITS_1;
  huart3.Init.Parity = UART_PARITY_NONE;
  huart3.Init.Mode = UART_MODE_TX_RX;
  huart3.Init.HwFlowCtl = UART_HWCONTROL_RTS_CTS;
  huart3.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart3) != HAL_OK)
  {
    Error_Handler();
  }

}

/** Configure pins as
        * Analog
        * Input
        * Output
        * EVENT_OUT
        * EXTI
*/
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin : B1_Pin */
```

```
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : TransistorSwitch_Pin */
  GPIO_InitStruct.Pin = TransistorSwitch_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init(TransistorSwitch_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, LD2_Pin|TransistorSwitch_Pin, GPIO_PIN_RESET);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

}

/* USER CODE BEGIN 4 */
void TestCaseOutput(void)
  {

  if(NextSymbol == 85){ //data validation for the initial transmission
```

```
    NextSymbol = TestArray[ArrInd];  //load the first transmission symbol in the array
    ArrInd++;
  }

  if(NextSymbol == 1){
    HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_SET);
    //HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
  }else if(NextSymbol == 0){
    HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_RESET);
  }

  if((ArrInd + 1) <= (ArrLen))
    { //prevent ArrInd going out of bounds
    CurrentSymbol = NextSymbol;
    NextSymbol = TestArray[ArrInd];
    ArrInd++;
    if(ArrInd == ArrLen)
      {
      ArrInd = 0;
    }
  }
  togglecheck = 0; //clear flag
}

void ReadCustomData(void)
  {
      //HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
  //    static int p=0; //,i = 0;
      //uint8_t buffer[100]; //for inputting data
      //uint8_t recbuff[1]; //for inputting data
```

```
    uint8_t sendbuff[8];
    //p = ADC_data[0];
    //HAL_UART_Transmit(&huart2, buffer, p, 50);
  if (ConvCplt == 1)
  {
      ConvData[0] = ADC_data[0];
      for(i=0;i<8;i++)
      {
        if (ConvData[0]%2 == 1)
        {
          sendbuff[i] = 1;
        }
        else
        {
          sendbuff[i] = 0;
        }
      ConvData[0] = ConvData[0]/2;
      }
      view1 = sendbuff[0];
      view2 = sendbuff[1];
      view3 = sendbuff[2];
      view4 = sendbuff[3];
      view5 = sendbuff[4];
      view6 = sendbuff[5];
      view7 = sendbuff[6];
      view8 = sendbuff[7];
//    p = sprintf((char *)buffer, "Please enter in your own input:\n"); for entering data from putty/serial
//    HAL_UART_Transmit(&huart2, buffer, p,50);
//    HAL_UART_Receive(&huart2, recbuff, 2, 10000);
    //p = 0;
    //recbuff[0] = ;
//    for(i=0;i<8;i++){
```

```
//       sendbuff[i] = (recbuff[0] & (0x1 << i)); //this might parse the recbuff (which is stored as a single element) into 8 bits
//       sendbuff[i] = sendbuff[i] >> i; //perhaps the best way to get ADC temp value to this state would be to create var ADCsendbuff[8] and do the same as is done to the value from UART
//       //p++;
//     }

  for(int k=0;k<7;k++)
      {
    //these two for loops create the start sequence of 1-1-0-0
      for(int z = 0; z<1; z++) //changed to z<1 for try of 1-0
      {
        for(int j=0;j<ONES_REPEATED;j++)
            { //Flag a start to the sequence
              while(!togglecheck){}
              HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_SET);
              //HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
              togglecheck = 0;
            }
        //HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_RESET);
      }
      for(int z = 0; z<1; z++)
      {
        for(int j=0;j<ZEROES_REPEATED;j++) //changed to z<1 for 1-0
        { //Flag a start to the sequence
              while(!togglecheck){}
              HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_RESET);
              //HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
```

```
                    togglecheck = 0;
            }
            //HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port,
                TransistorSwitch_Pin, GPIO_PIN_SET);
          }
  //this loop takes the 8 values in sendbuff and makes them into the
      bitstream to be sent
if (k%2 == 0 && k<5)
{
    for(i=0;i<8;i++)
    {
      if(sendbuff[i] == 1) //1 = 1-0
        {
        ManchesterOne();
        ManchesterZero();
        if(k>4)
        {
        sendbuff[i] = 0;
        }
        }
        else if(sendbuff[i] == 0) //0 = 1-1-0
        {
        ManchesterOne();
        ManchesterOne();
        ManchesterZero();
        if(k>4)
        {
        sendbuff[i] = 0;
        }
        }
    }
  }
else //null delimiter thing
```

```
{
  for(i=0;i<8;i++)
  {
    ManchesterOne();
    ManchesterOne();
    ManchesterZero();
    //ManchesterZero();
  }
}
HAL_Delay(10);
  }
    ConvCplt = 0;
  }
    while(!togglecheck){ //set back to low
    }
    HAL_GPIO_WritePin(TransistorSwitch_GPIO_Port, TransistorSwitch_Pin, GPIO_PIN_RESET);
    //HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
    togglecheck = 0;
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc1) //reads temperature sensor
{
    ADC_data[0] = HAL_ADC_GetValue(hadc1);
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); //shows conversion happening every 2 seconds
    static int p1=0;
    p1 = sprintf((char *)buffer1, "%d%c%d%c%d%c%c",ADC_data[0],0,ADC_data[0],0,ADC_data[0],0,0); // remember sprintf returns a length
    HAL_UART_Transmit(&huart2, buffer1, p1,50); //transmit data in buffer1 with length p1
```

```
    ConvCplt = 1;
}
//void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) //IGNORE THIS
   CALLBACK --UNUSED
//{
//  //HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); //pin toggle to ensure
   operation
//
//  if(LastValue <= 4095){
//    if((LastValue < LOWEDGE) && (ADCConvertedValue > HIGHEDGE)){
//      EdgeCount++;
//      if(HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1) != HAL_OK)
//        {
//          Error_Handler();
//        }
//    }
//
////    if(CurrentSymbol == 1){
////      if(EdgeCount >= HIGHCOUNT){
////        EdgeCount = 0;
////        togglecheck = 1;
////      }
////    }else if(CurrentSymbol == 0){
////      if(EdgeCount >= LOWCOUNT){
////        EdgeCount = 0;
////        togglecheck = 1;
////      }
////    }
//  }

//  LastValue = ADCConvertedValue; //store last state
//}

```

```
void HAL_TIM_IC_CaptureCallback( TIM_HandleTypeDef* htim ){ //this function writes the togglecheck back to 1 for the while loop in main
//this means the uc must have the oscillator folding mechanism on pin D12 to generate the 1010101010 pattern

  if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1){ //check that the proper timer channel interrupted
    if((PolarityCheck == 'R') && (CaptureIndex == 0)){ //if the next edge is rising and the first capture
      CapValue0 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
      CaptureIndex++; //increase index variable
      PolarityCheck = 'F'; //change capture polarity
    }else if(CaptureIndex == 1){
      CapValue1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
      CaptureIndex++;
      PolarityCheck = 'R';
    }else if(CaptureIndex == 2){
      CapValue0 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
      CaptureIndex--; //bring back to CapInd = 1
      PolarityCheck = 'F';//change polarity
      togglecheck = 1; //now that ---__-- low pulse detected, begin next transmission
    }
    if(PolarityCheck == 'R'){ //change IC capture polarity
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim3, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_RISING);
    }else if(PolarityCheck == 'F'){
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim3, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_FALLING);
    }
  }
}

```

```
void HAL_TIM_OC_DelayElapsedCallback( TIM_HandleTypeDef* htim ){
  HAL_TIM_OC_Stop_IT(htim, TIM_CHANNEL_1);
  if(EntryModeFlag){
    CaseNumber = 1;
    EntryModeFlag = 0;
    StartFlag = 1;
  }else if(!CaseNumber){
    ArrInd = 0;
    ArrLen = sizeof(TestArray) / sizeof(int);
    //HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
    StartFlag = 1;
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){


//  if((htim4.Instance->CR1 && 0x0001) == 0){
//    HAL_TIM_OC_Start_IT(&htim4, TIM_CHANNEL_1);
//  }else if((htim4.Instance->CR1 && 0x0001) == 1){
//    EntryModeFlag = 1;
//  }
  StartFlag = 1;
  CaseNumber++;
//  if(!CaseNumber){
//    HAL_TIM_IC_Stop_IT(&htim3, TIM_CHANNEL_1);
//    CaseNumber++;
//    ArrInd = 0;
//  }else{
//    CaseNumber = 0;
//    ArrLen = sizeof(TestArray) / sizeof(int);
//    ArrInd = 0;
//        HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
```

```
//  }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef * huart){

}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  None
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler */
  /* User can add his own implementation to report the HAL error return
     state */
  while(1)
  {
  }
  /* USER CODE END Error_Handler */
}

#ifdef USE_FULL_ASSERT

/**
   * @brief Reports the name of the source file and the source line number
   * where the assert_param error has occurred.
   * @param file: pointer to the source file name
   * @param line: assert_param error line source number
   * @retval None
   */
```

```
void assert_failed(uint8_t* file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
     number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
       line) */
  /* USER CODE END 6 */

}

#endif

/**
  * @}
  */

/**
  * @}
*/

/************************ (C) COPYRIGHT STMicroelectronics *****END OF
   FILE****/
```

Appendix D

Program for Decoder

```
/**
  ******************************************************************************
  * File Name          : main.c
  * Description        : Main program body
  ******************************************************************************
  ** This notice applies to any and all portions of this file
  * that are not between comment pairs USER CODE BEGIN and
  * USER CODE END. Other portions of this file, whether
  * inserted by the user or by software development tools
  * are owned by their respective copyright owners.
  *
  * COPYRIGHT(c) 2017 STMicroelectronics
  *
  * Redistribution and use in source and binary forms, with or without modification,
  * are permitted provided that the following conditions are met:
  *   1. Redistributions of source code must retain the above copyright notice,
  *      this list of conditions and the following disclaimer.
  *   2. Redistributions in binary form must reproduce the above copyright notice,
  *      this list of conditions and the following disclaimer in the documentation
  *      and/or other materials provided with the distribution.
  *   3. Neither the name of STMicroelectronics nor the names of its contributors
```

```
  *      may be used to endorse or promote products derived from this software
  *      without specific prior written permission.
  *
  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
  *
  ******************************************************************************
  */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32f4xx_hal.h"

/* USER CODE BEGIN Includes */
//#include "extern_declare.h"
```

```
#define ZEROPULSE 3400 //change this to change the threshold for 1 and 0
    pulse length
#define SAMPLE_DELAY 2550
#define EIGHT_TWO_MSDELAY 8200
#define INPUT_RISE 'R'
#define INPUT_FALL 'F'
/* USER CODE END Includes */

/* Private variables
    ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;
DAC_HandleTypeDef hdac;
TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
TIM_HandleTypeDef htim5;
TIM_HandleTypeDef htim6;
UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
/* Private variables
    ---------------------------------------------------------*/
uint32_t dbgflg = 0;
uint32_t dbg2 = 0;

int index_seq = 0;
int num = 0;

uint32_t IntegratorValue;
uint32_t ADCValue[1];
```

```
uint32_t OverSampleCounter = 0;
uint32_t SymbolCounter = 0;
uint32_t OnesCounter = 0;
uint32_t ZeroCounter = 0;
uint32_t ZeroDetected = 0;
uint32_t RUNONCEFLAG = 0;

uint32_t CurrentConvertedValue;
uint32_t PastConvertedValue;

uint32_t FallTimestamp;
uint32_t RiseTimestamp;
uint32_t TimeDiff;
uint8_t EdgeFlag;
uint32_t FilteredSignal;
uint32_t EdgeDetectFlag;

uint32_t difference = 0;

uint32_t ValueCheck;
uint32_t PulseArray[80];
uint32_t PulseIndex = 0;
uint32_t CountEntries = 0;
uint32_t SymbolIndex = 0;
uint32_t RecoveredSignal = 0;
uint32_t CollectFlag = 0;

char buffer[10];
int n;

uint32_t SerialModeFlag = 0;
uint32_t SymbolReceived;
uint32_t SequenceFlags;
```

```
uint32_t SequenceIndex;
uint32_t StartFlag = 0;
uint8_t SerialSequenceReceived[8]; //changed to 16 to accept Manchester
    data
uint32_t SerialIndex = 8; //changed to 16 to accept the Manchester data
int start = 0; //used with new encoding scheme (1-0 and 1-1-0)
int databegin = 0; //to make sure the pulse measurement takes place on the
     rising edge of the first data sequence
int arrayPos = 8; //used to store data bits in array


/* USER CODE END PV */

/* Private function prototypes
    -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_DAC_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM6_Init(void);
static void MX_TIM4_Init(void);
static void MX_TIM5_Init(void);
static void MX_TIM1_Init(void);

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);


/* USER CODE BEGIN PFP */
```

```
/* Private function prototypes -----------------------------------------------*/
void ADCBasicMode(void);
uint8_t EdgeDetect(uint32_t past_val, uint32_t curr_val, uint32_t* flag);
uint32_t PulseSymbolValidation(uint32_t rise_time, uint32_t fall_time);
uint32_t StartSequenceCheck(uint32_t last_symbol, uint32_t received_ptr, uint32_t index);
void OutputSymbol (void);
void SequenceCheck(void);
uint32_t IntegratorAdjust(uint32_t signal, uint32_t integrator);
/* USER CODE END PFP */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration----------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
```

```c
/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_ADC1_Init();
MX_TIM2_Init();
MX_USART2_UART_Init();
MX_DAC_Init();
MX_TIM3_Init();
MX_TIM6_Init();
MX_TIM4_Init();
MX_TIM5_Init();
MX_TIM1_Init();

/* USER CODE BEGIN 2 */
HAL_ADC_Start_DMA(&hadc1, ADCValue, sizeof(uint16_t));
HAL_TIM_Base_Start(&htim2);
HAL_TIM_IC_Start_IT(&htim4, TIM_CHANNEL_1);

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
```

```
  /* USER CODE BEGIN 3 */

  }
  /* USER CODE END 3 */

}

/** System Clock Configuration
*/
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;

    /**Configure the main internal regulator output voltage
    */
  __HAL_RCC_PWR_CLK_ENABLE();

  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /**Initializes the CPU, AHB and APB busses clocks
    */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 170;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
```

```
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Activate the Over-Drive mode
    */
  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Initializes the CPU, AHB and APB busses clocks
    */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV4;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**Configure the Systick interrupt time
    */
  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

    /**Configure the Systick
    */
```

```
  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}

/* ADC1 init function */
static void MX_ADC1_Init(void)
{

  ADC_ChannelConfTypeDef sConfig;

    /**Configure the global features of the ADC (Clock, Resolution, Data
       Alignment and number of conversion)
    */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

```

```
    /**Configure for the selected ADC regular channel its corresponding
       rank in the sequencer and its sample time.
    */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* DAC init function */
static void MX_DAC_Init(void)
{

  DAC_ChannelConfTypeDef sConfig;

    /**DAC Initialization
    */
  hdac.Instance = DAC;
  if (HAL_DAC_Init(&hdac) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

    /**DAC channel OUT1 config
    */
  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
```

```
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* TIM1 init function */
static void MX_TIM1_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig;

  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 84;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 0xffff;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  if (HAL_TIM_OC_Init(&htim1) != HAL_OK)
  {
```

```
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
  sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
  if (HAL_TIM_OC_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
  sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
  sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
  sBreakDeadTimeConfig.DeadTime = 0;
  sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
  sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
  sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
  if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) !=
     HAL_OK)
```

```
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* TIM2 init function */
static void MX_TIM2_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 45;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  if (HAL_TIM_OC_Init(&htim2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
```

```
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_TIMING;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
  if (HAL_TIM_OC_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  HAL_TIM_MspPostInit(&htim2);

}

/* TIM3 init function */
static void MX_TIM3_Init(void)
```

```
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 0;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 0xffff;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  if (HAL_TIM_OC_Init(&htim3) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
      HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
```

```
  }

  sConfigOC.OCMode = TIM_OCMODE_TIMING;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* TIM4 init function */
static void MX_TIM4_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_IC_InitTypeDef sConfigIC;

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 0xffff;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
```

```
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  if (HAL_TIM_IC_Init(&htim4) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel(&htim4, &sConfigIC, TIM_CHANNEL_1) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* TIM5 init function */
static void MX_TIM5_Init(void)
{
```

```
  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
  TIM_OC_InitTypeDef sConfigOC;

  htim5.Instance = TIM5;
  htim5.Init.Prescaler = 0;
  htim5.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim5.Init.Period = 4024;
  htim5.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  if (HAL_TIM_Base_Init(&htim5) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim5, &sClockSourceConfig) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  if (HAL_TIM_OC_Init(&htim5) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim5, &sMasterConfig) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
```

```
  sConfigOC.OCMode = TIM_OCMODE_TIMING;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_OC_ConfigChannel(&htim5, &sConfigOC, TIM_CHANNEL_2) !=
     HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* TIM6 init function */
static void MX_TIM6_Init(void)
{

  TIM_MasterConfigTypeDef sMasterConfig;

  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 0;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 56366;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) !=
     HAL_OK)
  {
```

```
    _Error_Handler(__FILE__, __LINE__);
  }

}

/* USART2 init function */
static void MX_USART2_UART_Init(void)
{

  huart2.Instance = USART2;
  huart2.Init.BaudRate = 9600;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{
  /* DMA controller clock enable */
  __HAL_RCC_DMA2_CLK_ENABLE();

  /* DMA interrupt init */
```

```
  /* DMA2_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);

}

/** Configure pins as
        * Analog
        * Input
        * Output
        * EVENT_OUT
        * EXTI
*/
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, LD2_Pin|DecodedOutput_Pin|DebugOutput_Pin,
      GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(FeedOut_GPIO_Port, FeedOut_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(TimerOut_GPIO_Port, TimerOut_Pin, GPIO_PIN_RESET);

```

```
/*Configure GPIO pin : B1_Pin */
GPIO_InitStruct.Pin = B1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : LD2_Pin */
GPIO_InitStruct.Pin = LD2_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : FeedOut_Pin */
GPIO_InitStruct.Pin = FeedOut_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
HAL_GPIO_Init(FeedOut_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : DecodedOutput_Pin DebugOutput_Pin */
GPIO_InitStruct.Pin = DecodedOutput_Pin|DebugOutput_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/*Configure GPIO pin : TimerOut_Pin */
GPIO_InitStruct.Pin = TimerOut_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(TimerOut_GPIO_Port, &GPIO_InitStruct);
```

```
  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

}

/* USER CODE BEGIN 4 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
  {
  if(htim->Instance == TIM4)
    {
    switch(EdgeFlag)
      {
      case INPUT_RISE:
        RiseTimestamp = __HAL_TIM_GET_COMPARE(&htim4, TIM_CHANNEL_1);
        TIM_RESET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1);
        TIM_SET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1,
            TIM_ICPOLARITY_FALLING);
        break;
      case INPUT_FALL:
        FallTimestamp = __HAL_TIM_GET_COMPARE(&htim4, TIM_CHANNEL_1);
        TIM_RESET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1);
        TIM_SET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1,
            TIM_ICPOLARITY_RISING);
        FilteredSignal = PulseSymbolValidation(RiseTimestamp,
            FallTimestamp);
        //if(SerialModeFlag == 0){
          switch(FilteredSignal)
            {
            case 0:
            if (ZeroCounter > 40) //weirdness, just reset everything
            {
```

```
            start = 0;
            ZeroCounter = 0;
            OnesCounter = 0;
            arrayPos = 8;
          }
          if (start == 0)
          {
            HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin
                , GPIO_PIN_RESET);
          }
          else
          {
            //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
                DecodedOutput_Pin, GPIO_PIN_SET);
          }
//              if (OnesCounter != 0)
//              {
//              HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET);
//              }
//              if (start == 0)
//              {
//              //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET);
//              }
//              ZeroCounter++;
        if((OnesCounter <= 16) && (start == 0)) //this is a case where
            some noise might have gotten through
        {
          OnesCounter= 0;
////                //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET); //show normal/test zero
        }
```

```
        if((OnesCounter > 5) && (ZeroCounter > 3) && (start == 0)) //this is when the ones and zeros are both high enough to signify the start sequence
        {                                                        //counts up how many ONE pulses from matched filter and ZERO pulses after to determine start sequence
//              //if((htim5.Instance->CR1 && TIM_CR1_CEN) == 0)                                    //altered to accept 1-0 start sequence
//                  //HAL_TIM_Base_Start_IT(&htim5);
//                  //RUNONCEFLAG++;
                start = 1;
                OnesCounter = 0; //keep the Ones count of the start sequence from messing with the first data bit
//                  OnesCounter = 0;
//                  ZeroCounter = 0;
//                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_SET);
        }
//          else if ((start == 1) && (databegin == 1)) //if data is being received -- this should be entered when the first return to zero happens.
//          {
//              if (arrayPos > 0) //data value received is between 1 and 8
//              {
  if (ZeroCounter == 0 && start == 1)
  {

        if (start == 1 && arrayPos > 1)
        {
            //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_SET);
```

```
                if (OnesCounter > 24)
                {
                  HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_RESET); //received 1-1-0, go low
                  SerialSequenceReceived[arrayPos - 1] = 0;
                }
                else
                {
                  HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_SET); //received 1-0, go high
                  SerialSequenceReceived[arrayPos - 1] = 1;
                }
            OnesCounter = 0;
            ZeroCounter++;
            arrayPos--;
          }
          else if (start == 1 && arrayPos == 1)
          {
            if (OnesCounter > 24)
                {
                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_RESET); //received 1-1-0, go low
                  SerialSequenceReceived[arrayPos - 1] = 0;
                }
                else
                {
                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin, GPIO_PIN_SET); //received 1-0, go high
                  SerialSequenceReceived[arrayPos - 1] = 1;
```

```
                }
            arrayPos = 8;
            start = 0;
            OnesCounter = 0;
            HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin,
                GPIO_PIN_SET); //little blip to show transmission
            OutputSymbol(); //after 8 bits received, send to COM port
            ZeroCounter++;
          }
          else if (start == 0)
          {
            //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port, DecodedOutput_Pin
                , GPIO_PIN_RESET);
            ZeroCounter++;
          }
          else
          {
            ZeroCounter++;
          }
  }
  else
  {
    ZeroCounter++;
  }

//                  OnesCounter = 0;
//                  ZeroCounter = 0;
//                  arrayPos--; //prepare for next data bit
//                  }
//                  else //end of 8 bits of data
//                  {
//                    OutputSymbol(); //finished receiving 8 bits, output
    the data
```

```
//                  databegin = 0; //reset begin and start to wait for
   start sequence again
//                  start = 0;
//                  arrayPos = 8; //reset array position to be ready for
   next set of data
//                  OnesCounter = 0;
//                  ZeroCounter = 0;
//                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET);
//                }
//              }
            break;


          case 1:
            if (OnesCounter > 12 && start == 1)
            {
            //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
                DecodedOutput_Pin, GPIO_PIN_SET);
            }
            else
            {
            //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
                DecodedOutput_Pin, GPIO_PIN_RESET);
            }

//              if (start == 1 && databegin == 0) //this should be the
   first "1" after the start sequence is verified
//              {
//                databegin = 1; //show that data is starting to be
   received
//                OnesCounter = 0;  //reset to make sure of an accurate
   count
```

```
//                  ZeroCounter = 0;
//                }
//                else
//                {
//                if (start == 0)
//                {
//                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET); //show the high of the start
   sequence or a test case
//                }
//                else
//                {
//                  //HAL_GPIO_WritePin(DecodedOutput_GPIO_Port,
   DecodedOutput_Pin, GPIO_PIN_RESET); //go low when getting a high data
   pulse
//                }
                  OnesCounter++;
                  ZeroCounter = 0;
//                }
              break;
//            }
          }
          break;
      }
  }
}

void HAL_ADC_ConvCpltCallback( ADC_HandleTypeDef* hadc){

  CurrentConvertedValue = ADCValue[0];

  EdgeFlag = EdgeDetect(PastConvertedValue, CurrentConvertedValue, &
      EdgeDetectFlag);
```

```
    if(EdgeDetectFlag == 1){
      EdgeDetectFlag = 0;
      switch(EdgeFlag){
        case INPUT_RISE:
          HAL_GPIO_WritePin(FeedOut_GPIO_Port, FeedOut_Pin, GPIO_PIN_SET);
             //detected rising edge, flipped to RESET for inverter effect
          break;
        case INPUT_FALL:
          HAL_GPIO_WritePin(FeedOut_GPIO_Port, FeedOut_Pin, GPIO_PIN_RESET);
              //see above
          break;
      }
    }
    PastConvertedValue = CurrentConvertedValue;
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
//  SerialModeFlag = 1;
//  HAL_TIM_IC_Stop_IT(&htim4, TIM_CHANNEL_1);
//  HAL_TIM_Base_Stop_IT(&htim6);
//  __HAL_TIM_SET_COUNTER(&htim4, 0);
//  __HAL_TIM_SET_COUNTER(&htim6, 0);
//  RiseTimestamp = 0;
//  FallTimestamp = 0;
//  TIM_RESET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1);
//  TIM_SET_CAPTUREPOLARITY(&htim4, TIM_CHANNEL_1, TIM_ICPOLARITY_RISING);
//  HAL_TIM_IC_Start_IT(&htim4, TIM_CHANNEL_1);
}


//uncomment the period elapsed callback function if you want to do
   midpoint sampling

```

```
//void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
//  if(htim->Instance == TIM5){

//    if(OverSampleCounter < 17){ //change this in accordance with
   conditional statement below
//      OverSampleCounter++;
//    }else{
//      SymbolCounter++;
//      OverSampleCounter = 0;
//    }
//
//    if(OverSampleCounter == 16){  //CHANGE THIS TO CHANGE SAMPLING
   LOCATION
//
//      // What does this do? Shows output
//      HAL_GPIO_TogglePin(DebugOutput_GPIO_Port, DebugOutput_Pin);

//      SerialSequenceReceived[SerialIndex-1] = FilteredSignal;
//      SerialIndex--;
//      if(SerialIndex == 0){
//        SerialIndex = 8; //changed to 16 for Manchester
//        StartFlag = 0;
//        OnesCounter = 0;
//        ZeroCounter = 0;
//        SymbolCounter = 0;
//        OverSampleCounter = 0;
//
//        // What does this do? STOPS the midpoint timer
//        while(HAL_TIM_Base_Stop_IT(&htim5) != HAL_OK); //stop the
   midpoint timer after 8 samples
//
//        OutputSymbol();
//
```

```
//         for(int i = 0; i<8; i++){ //changed to 16 for manchester
//           SerialSequenceReceived[i] = 0;
//         }
//
//       }
//     }
//   }
//}



void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim){

}

/* EdgeDetect
 * Looks at 2 sequential values to determine whether a valid
 * edge transistion has occured.
 *
 * Outputs R or F to show whether edge was fall or rise,
 * and sets a 1 using a pointer to a flag.
 */
uint8_t EdgeDetect(uint32_t past_val, uint32_t curr_val, uint32_t* flag){
  uint8_t edge = 0;

  if((past_val >= MIDPOINT) && (curr_val <= MIDPOINT)){
    edge = INPUT_FALL; //falling edge
    *flag = 1;
  }else if((past_val <= MIDPOINT) && (curr_val >= MIDPOINT)){
    edge = INPUT_RISE; //Rising edge
    *flag = 1;
  }

```

```
  return edge;
}

/* PulseSymbolValidation
 * Calculates a pulse length, then determines whether the length was
 * within a valid threshold to be a valid transmission.
 *
 * Returns a flag to indicate a symbol was recieved, and writes a 1
 * (only valid symbol pulses denote) to an address.
 */
uint32_t PulseSymbolValidation(uint32_t rise_time, uint32_t fall_time){
  uint32_t flag = 0;
  //____----_____
  if(rise_time < fall_time){ //difference calculation
    difference = fall_time -  rise_time;
  }else if(rise_time > fall_time){
    difference = ((0xffff - rise_time) + fall_time);
  }
  if(difference <= ZEROPULSE){ //determining symbol
    flag = 1;
  }
  return flag;
}


void OutputSymbol (void){
  /*  OutputSymbol
   *  This function toggles an output pin to indicate whether a
   *  1 or 0 has been received.
   *  It also writes 1 or 0 to the UART (seen through USB COM Port
  */
  int p=0;
  uint8_t buffer[100];
```

```
  uint8_t compressedsequence = 0;

//  p = sprintf((char *)buffer, "TESTING\r\n");
//  HAL_UART_Transmit(&huart2, buffer, p,50);

  for(int i = 0; i<8; i++){
    compressedsequence += SerialSequenceReceived[i] << (7-i);
  }
  if(compressedsequence == 0){  //if a null received
    p = sprintf((char *)buffer, "%c", compressedsequence);
    dbg2 = HAL_UART_Transmit(&huart2, buffer, p,50);
  }else{ //otherwise it's temperature data
    p = sprintf((char *)buffer, "%d", compressedsequence);
    dbg2 = HAL_UART_Transmit(&huart2, buffer, p,50);
  }

  //HAL_TIM_IC_Start_IT(&htim4,TIM_CHANNEL_1);
}

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @param  None
  * @retval None
  */
void _Error_Handler(char * file, int line)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
     state */
  while(1)
  {
```

```
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT

/**
   * @brief Reports the name of the source file and the source line number
   * where the assert_param error has occurred.
   * @param file: pointer to the source file name
   * @param line: assert_param error line source number
   * @retval None
   */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
     number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
       line) */
  /* USER CODE END 6 */

}

#endif

/**
  * @}
  */

/**
  * @}
*/
```

```
/************************ (C) COPYRIGHT STMicroelectronics *****END OF
   FILE****/
```