Study and implementation of FM-RDS communications protocol

UNDERGRADUATE STUDENT PROJECT
NAVAL PHYSICAL AND OCEANOGRAPHIC LABORATORY
DEFENCE RESEARCH AND DEVELOPMENT ORGANISATION, MINISTRY OF DEFENCE

Joshua Peter Ebenezer
Third Year Undergraduate
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Project Guide: Dr. P. Murali Krishna
Scientist F, Naval Physical and Oceanographic Laboratory

Abstract

In this project report, a software scheme is presented for the simulation of the modulation and demodulation processes of the Frequency Modulation Radio Data System (FM-RDS) communications standard protocol. RDS was developed for the purpose of embedding small amounts of digital information in conventional FM broadcasts. This information may include programme information, track and artist name, station name, time, and traffic information. The RDS protocol is implemented and simulated results are presented. An analysis of the robustness of the scheme to noise is also performed. MATLAB® is used as the programming language for the implementation. Simulink® models are also built to simulate the scheme.

Acknowledgements

I acknowledge the help and support from my project guide, Dr. P. Muralikrishna. Despite his busy schedule, he provided guidance and direction, and monitored my progress. He provided all that I needed to complete the task in time.

My sincere thanks to Director, NPOL for giving me this opportunity and the resources to do this project. I learnt a lot from the discipline and methodologies followed here, and I am grateful to all the staff and scientists who made the workplace environment pleasant and enjoyable.

I also acknowledge my family, who have always been a source of support, strength, and joy. Thanks to Dad, Mum, Christine, and my grandparents for always being there for me.

Thanks be to God, my saviour, without whose grace and love I would not have been able to start or finish this project, and who in his sovereign and good will guides my work and life.

Table of Contents

LIST OF FIGURES

1 Introduction

Radio has had a relatively short history, but has developed extremely rapidly over the past 150 years, along with the rest of science and technology. It was in 1864 that Maxwell showed, mathematically, that electromagnetic waves could travel through free space, publishing his work in 1865 [1]. Many scientists worked to further his ideas, and the experimental demonstrations of Hertz from 1886-88 that validated Maxwell's formulations provided further impetus to others. However, it was Marconi who turned what were essentially laboratory experiments into commercially viable technology in the late 19th and early 20th century. From then on, radio development and application were accelerated by a host of technological advancements, and it is still growing, in its scope and ingenuity, well into the 21st century.

Frequency modulation (FM) is the encoding of information in a carrier wave by modifying the instantaneous phase of the wave. In the second half of the 20th century FM was shown to have superior Signal to Noise Ratio (SNR) as compared to Amplitude modulation (AM), with the effect of distortion from electrical equipment and the atmosphere greatly reduced. However, the range of FM (30-40 miles) is significantly lower than that of AM, due the higher frequencies. In general, an FM wave would be of the form

$$x_c(t) = A_c \sin(\omega_c t + 2\pi f_\Delta \int_0^t x_m(\theta)\, d\theta) \tag{1.1}$$

where $x_m(\theta)$ is the message signal, ω_c is the angular carrier frequency, A_c is the amplitude of the carrier signal, f_Δ is the frequency deviation, and $x_c(t)$ is the carrier signal.

RDS (Radio Data System) was developed by public broadcasters collaborating within the European Broadcasting Union (EBU) from about 1975 [2]. The purpose of RDS is to send useful digital information in conventional analog broadcasts. The first specification was issued by the EBU in March 1984. RDS was inspired by an upcoming traffic broadcast identification system called ARI, jointly developed in Germany by the public broadcasting research centre, IRT, and the car radio maker Blaupunkt. Enhancements to the alternative frequencies functionality were added to the standard and it was subsequently published as a European Committee for Electrotechnical Standardization (CENELEC) standard in 1990. The CENELEC standard was updated in 1992 with the addition of Traffic Message Channel and in 1998 with Open Data Applications [3]. Radio Broadcast Data System (RBDS) is the official name used for the U.S. version of RDS [4]. The two standards are only slightly different. In 2000, RDS was published worldwide as IEC standard 62106 [5].

Most RDS implementations have been based on hardware. This is probably due to the very low cost of FM-RDS ICs that have enabled rapid commercialization of car radio devices that are capable of receiving and demodulating RDS data. Literature on software based implementations are scarce and recent. Shende published a thesis that was based on a software implementation of the receiving and demodulation of RDS data [6]. As radio evolves, there have already been strides in the direction of fully software based radios (software defined radios). RDS functionality is continuously being adapted for more functionality, a recent example being Li *et al*. [7] using RDS data for adaptive clock calibration in sensor networks. As the field of radio communications increasingly finds ways to exploit the vast opportunities and advantages that software based functionality provides, this project presents an implementation of RDS functionality on MATLAB®, a programming language with significant capabilities and that was especially developed for technical work.

2 Modulation of FM-RDS broadcasts

In this section, the RDS protocol is described. First an overview of stereophonic FM broadcast modulation is presented. After this, the data channel is described, followed by a description of the baseband coding (data-link layer).

2.1 Overview of modulation of FM-RDS broadcasts

Stereo FM broadcast technology was developed much after monophonic FM broadcast, and hence when stereo was introduced it was made necessary that stereo would be fully compatible with monophonic systems. Hence, in a typical stereophonic FM broadcast, the signals intended for the left and right speakers (henceforth referred to as 'left'(L) and 'right'(R) respectively) are added up and sent at baseband with a bandwidth of 15 kHz. Monophonic receivers play this added signal from a single speaker. The difference of the 'left' and 'right' signals (L-R) is sent on a 38 kHz subcarrier using the double sideband suppressed carrier technique (DSB-SC), with a bandwidth of 30 kHz. Thus the L-R signal is present between 23 kHz and 53 kHz. Stereo receivers demodulate both the L+R signal at baseband, as well as the L-R signal at 38 kHz, to find the 'left' and 'right' signals separately for the two speakers. A pilot tone is sent at 19 kHz to indicate the presence of the stereo L-R signal (which is at twice the pilot tone), as well as to enable synchronized demodulation of the L-R and RDS signals via carrier recovery (since the carriers for L-R and RDS are at twice and thrice the pilot tone frequency respectively). The RDS data is sent at 57 kHz (thrice the pilot tone), to minimize interference and intermodulation with the L-R signal and the pilot, and has a bandwidth of 4.8 kHz. This combined signal is then frequency modulated and sent in the 87.5 to 108.0 MHz portion of the spectrum. The schematic of the spectrum of a typical FM broadcast before frequency modulation is shown in Figure 2.1.

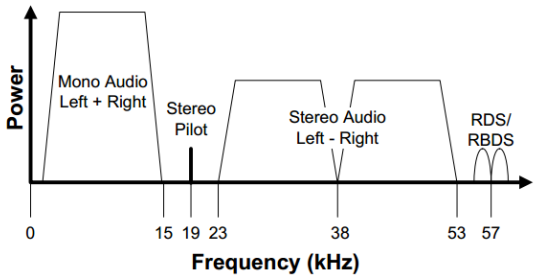

Figure 2.1. Spectrum of a typical FM-RDS broadcast.

Mathematically, if the audio signal intended for the left speaker is represented as $l(t)$, and the signal for the right speaker as $r(t)$, then the sum signal (sent at baseband) is given by

$$s(t) = l(t) + r(t) \tag{2.1}$$

and the difference signal, when DSB-SC modulated at 38 kHz, is given by

$$d(t) = [l(t) - r(t)]\cos(2\pi f_d t) \tag{2.2}$$

where $f_d = 38$ kHz. The Fourier transform of $\cos(\omega_c t)$ is

$$F(\omega) = \int_{-\infty}^{\infty} \cos(\omega_c t)\, e^{-i\omega t}\, dt = \frac{\delta(\omega-\omega_c)+\delta(\omega+\omega_c)}{2} \tag{2.3}$$

where $\delta(.)$ denotes the Dirac delta function.

So, if the Fourier transform of $l(t) - r(t)$ is $G(\omega)$, then multiplying $\cos(\omega_d t)$ with the difference $l(t) - r(t)$ corresponds to convolving $F(\omega)$ and $G(\omega)$. If the Fourier transform of this product (defined in (2.2)) is $D(\omega)$, then

$$D(\omega) = G(\omega) * F(\omega) = \frac{G(\omega-\omega_d)+G(\omega+\omega_d)}{2} \tag{2.4}$$

which corresponds to a signal mirrored around ω_d, and is the band-limited signal required. Similarly, the RDS signal is modulated by a 57 kHz carrier. The pilot tone added is $p(t) = \cos(\omega_p t)$ (if there is no phase offset). If the RDS signal at 57 kHz is represented as $q(t)$, then the signal, $m(t)$, that is to be frequency modulated, can be written as

$$m(t) = s(t) + d(t) + p(t) + q(t) \tag{2.5}$$

The schematic of the expected spectrum of $m(t)$ is shown in Figure 2.1, as mentioned earlier.

2.2 Data channel (Physical layer) modulation

The RDS data is sent at the rate of 1187.5 bits/s. As the carrier is at 57 kHz, there are 48 carrier cycles for each bit. Digital data is sent after encoding and pulse shaping, as shown in Figure 2.2. A description of each block of the RDS modulation system is presented in the following subsections.

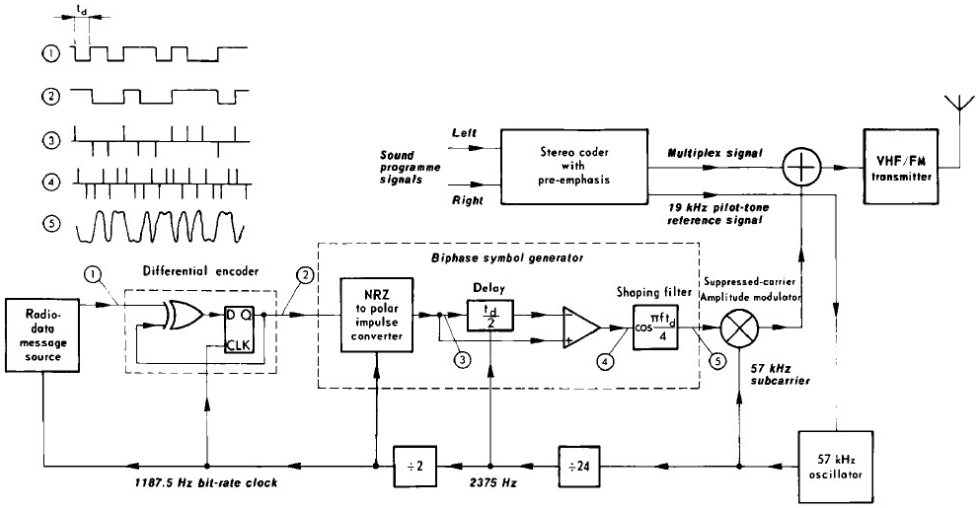

Figure 2.2. Modulation scheme of FM-RDS.

2.2.1 Differential encoder

The data is first differentially encoded. The reason behind this is that differential encoding ensures that even if the encoded stream is fully inverted, upon demodulation the stream will turn out to be the correct one. When data is transmitted over twisted-pair lines, it is easy to unintentionally introduce a half twist in the lines. This will cause complete inversion of the received data, called 180° phase ambiguity. Differential encoding protects against this. The encoded bit can be interpreted as the sum of the bit to be encoded and the previous encoded bit. The scheme used for the encoding is

$$e_k = e_{k-1} \oplus b_k \tag{2.6}$$

where $\oplus$ denotes the XOR (exclusive-OR) logic operation, b_k is the bit at the k^{th} index of the input stream that is to be encoded, and e_k is the encoded bit at the k^{th} index of the encoded stream. The truth table for the exclusive-OR operation is shown in Table 2-1.

Table 2-1

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

2.2.2 Non-return-to-zero to polar impulse converter

Non-return-to-zero refers to the differentially encoded bitstream, in the sense that the value is always either 0 or 1, and the bits never takes any value in between. A return-to-zero code would have a rest state in addition to the 0 and 1 states.

A polar impulse is a delta function when the bit is 1, and a delta function multiplied by -1 when the bit is 0. The delta function occurs at multiples of t_d, where t_d is the time that elapses between each bit and is equal to 1/1187.5 s, as the bit rate for RDS is 1187.5 bits/s. The bitstream's value is sampled right after t_d to produce the corresponding delta function. The polar impulse is labelled as the third waveform in Figure 2.2. The impulse train is up-sampled to enable shaping of the signal. This step is the precursor to the generation of the biphase symbols.

2.2.3 Biphase symbol generator

The power of the data near the 57 kHz carrier is minimized by coding each bit as a biphased symbol. This is achieved because the average value of biphase symbols is 0 (refer to Figure 2.5), and hence no DC power is transmitted. Atleast one transition occurs in each bit. This in contrast to how bits are generally represented, where a transition occurs only at the end of each bit period.

Each binary bit gives rise to an odd impulse pair. If the binary bit is logic 1, then the pair is

$$e(t) = \delta(t) - \delta(t + \tfrac{t_d}{2}). \tag{2.7}$$

If the binary bit is a logic 0, then

$$e(t) = -\delta(t) + \delta(t + \tfrac{t_d}{2}). \tag{2.8}$$

Here, $e(t)$ is the generated impulse pair, and $\delta(t)$ refers to the delta function. The impulse pairs are labelled as the 4th waveform in Figure 2.2. t_d, as described earlier, is the time that elapses between each bit, and is equal to 1/1187.5 s. This stream of impulse pairs is then fed to a cosine filter described by the equation

$$H(f) = \begin{cases} \cos(\frac{\pi t_d}{4}), & 0 < f \leq \frac{2}{t_d} \\ 0, & f > \frac{2}{t_d} \end{cases}. \tag{2.9}$$

The frequency response of the cosine filter is shown in Figure 2.3. This gives the required band-limited spectrum, shown in Figure 2.4. The time domain responses of the biphase symbols are shown in Figure 2.5 for logic 1 and logic 0.

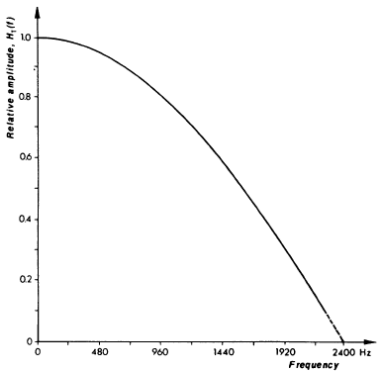

Figure 2.3. Frequency response of raised cosine filter.

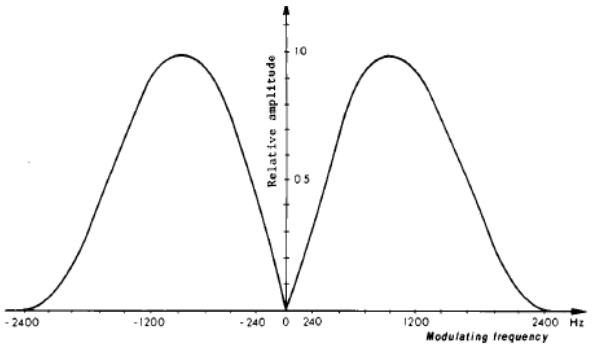

Figure 2.4. Typical Spectrum of RDS signal after shaping.

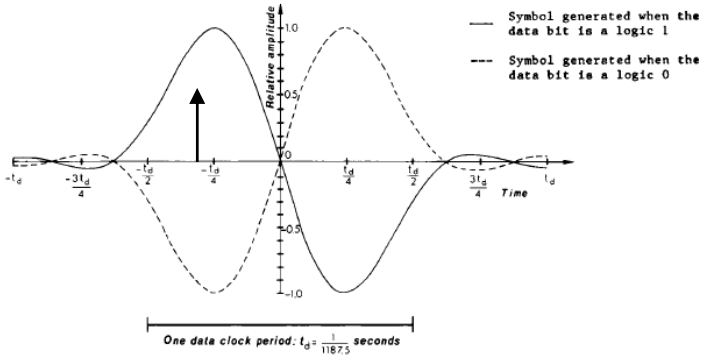

Figure 2.5. Biphase symbols for logic 1 and logic 0

This shaped signal can then be modulated by multiplying it with a carrier (tone) at 57 kHz, which would cause the baseband signal to be mirrored around 57 kHz for transmission.

2.3 Baseband coding (data-link layer)

The structure of the baseband message is shown in Figure 2.6. The broadcast is divided into groups, each of length 104 bits. Each group is further divided into 4 blocks, that are each of width 26 bits. Each block contains the word that carries the information, as well as the sum of a checkword and an offset word. The information word is 16 bits long, and the sum of the checkword and the offset word is 10 bits long.

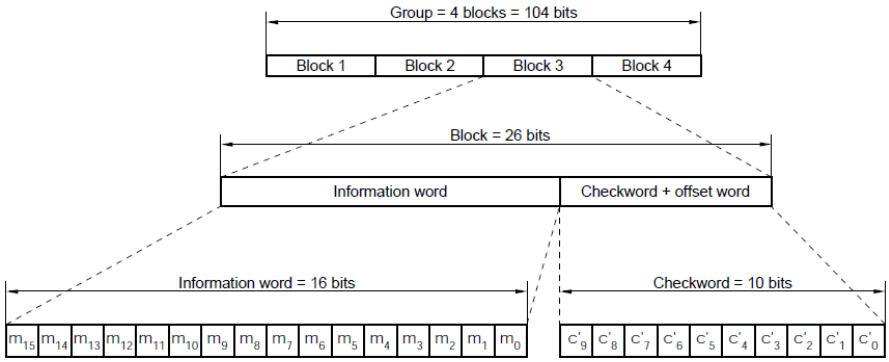

Figure 2.6. Baseband coding structure

2.3.1 Error protection

A significant function of the data link layer is to provide a means by which errors in the data can be detected. Based on the encoding scheme used, the receiver may be able to correct the error, or on detecting the error ask the transmitter to attempt sending the correct message again. In the RDS scheme, the checkword for the information is formed using a CRC (cyclic redundancy check) encoder, and is 10 bits long. The generator polynomial for the CRC is given by

$$g(x) = x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1 \tag{2.10}$$

The principle behind CRC encoding is based on binary arithmetic, and polynomial division. A binary sequence can be represented as a polynomial in a variable, say x, having coefficients of value unity for powers of x that are of the same value as the binary placeholder positions in the sequence that are non-zero. The coefficients of powers of x that correspond to positions in the sequence having value 0, are 0. For example, a sequence 110101 can be represented as $x^5 + x^4 + x^2 + 1$.

Following this scheme, say the message to be transmitted is represented as $p(x)$. If $p(x)$ is multiplied by x^k, where k is the degree of the generator polynomial, and the result is divided by the generator polynomial $g(x)$, a quotient $q(x)$ and a remainder $r(x)$ are generated, such that

$$x^k p(x) = q(x)g(x) + r(x) \tag{2.11}$$

where the degree of $r(x)$ is lesser than k, which is the degree of the generator polynomial. It must be remembered that the coefficients are not integers, but integers modulo-2, or binary numbers. The length of the coefficients of the polynomial $r(x)$ will hence not exceed k, and the coefficients themselves are the checkword. They are concatenated to the message sequence, which is shifted k bits to the left. In the RDS scheme, since the generator polynomial is of order 10, k is

equal to 10. The bits of the transmitted block will correspond to the polynomial $x^k p(x) + r(x)$. Addition in modulo-2 arithmetic is actually the same as subtraction as the carry is not taken into account (refer to Table 2-1). Hence, this polynomial can be rewritten as $x^k p(x) - r(x)$. From equation (2.6), this is the same as $q(x)g(x)$. In other words, the transmitted message must correspond to a polynomial that is divisible by $g(x)$. If the received block is not divisible by $g(x)$, an error must have occurred. If it is divisible by $g(x)$, note that this still may not guarantee that the message is error free. However, it can be proved that $g(x)$ of the form described in (2.5) will be able to detect all single bit errors, burst errors of length not exceeding a length of 10, and double bit errors. [8] [9]. If an error is found, it can be corrected with the generation of a syndrome that will indicate the error.

After the checkword is generated, an offset word is added to it. The offset word depends on the block type (refer below). The checkword itself is intended for error detection and correction. The offset word, on the other hand, is intended for synchronization of the group and the block, and is made use of via the error detection process, more fully described in section 3.2.1.
Hence, the process for the development of the last 10 bits of each block is as follows:

i. Find the remainder when the product of the word with x^{10} is divided by $g(x)$ under modulo-2 arithmetic.
ii. According to the block position within the group, add an offset (refer to Figure 2.7) to the remainder. This forms the last 10 bits for that block.

To simplify the calculation required to find the remainder in the first step of the above process, define the generator matrix

$$\mathbf{G} = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&1&1&1\\
0&1&&&&&&&&&&&&&&&0&1&0&1&1&1&0&0&1&1&1\\
0&&1&&&&&&&&&&&&&&0&1&1&1&0&1&0&1&1&1&1\\
0&&&1&&&&&&&&&&&&&0&1&1&0&0&0&0&1&0&1&1\\
0&&&&1&&&&&&&&&&&&0&1&1&0&1&0&1&1&0&0&1\\
0&&&&&1&&&&&&&&&&&0&1&1&0&1&1&1&0&0&0&0\\
0&&&&&&1&&&&&&&&&&0&0&1&1&0&1&1&1&0&0&0\\
0&&&&&&&1&&&&&&&&&0&0&0&1&1&0&1&1&1&0&0\\
0&&&&&&&&1&&&&&&&&0&0&0&0&1&1&0&1&1&1&0\\
0&&&&&&&&&1&&&&&&&0&0&0&0&0&1&1&0&1&1&1\\
0&&&&&&&&&&1&&&&&&0&1&0&1&1&0&0&0&1&1&1\\
0&&&&&&&&&&&1&&&&&0&1&1&1&0&1&1&1&1&1&1\\
0&&&&&&&&&&&&1&&&&0&1&1&0&0&0&0&0&0&1&1\\
0&&&&&&&&&&&&&1&&&0&1&1&0&1&0&1&1&1&0&1\\
0&&&&&&&&&&&&&&1&&0&1&1&0&1&1&1&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&0&1&1&1&0&0&1
\end{bmatrix}.$$

The i^{th} row of this matrix is the concatenated message and checkword for messages having 0's in all bits except the i^{th} position. For example, the message and checkword combination for the message 0000000000000001 is 00000000000000010110111001, which is the last row of the matrix. The 26 bit representation x for a given 16 bit message m is generated as

$$x = m\mathbf{G} \tag{2.12}$$

2.4 Message format

The format of the message is shown in Figure 2.7. Each block has a different offset added to its checkword. The first block of each group holds information regarding the PI code (program identification), and a checkword that is added to offset A (refer to Figure 2.7). The second block of each group holds information regarding the program type (PTY in short), with an offset B for

the checkword. The codes that are repeated most frequently, and require a very short acquisition time, occupy fixed positions in the first two blocks so that the information in the first two blocks can be decoded and retrieved independently of any other block.

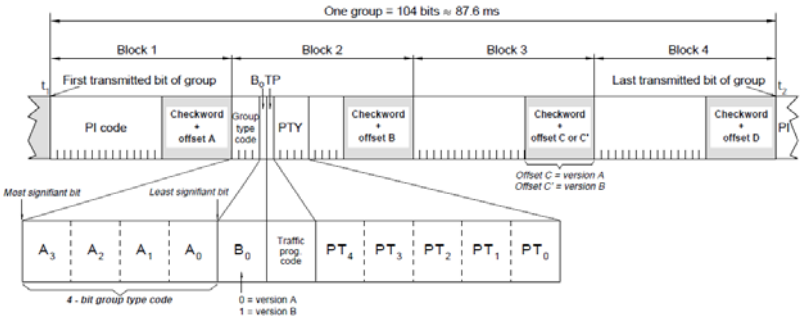

Figure 2.7. Message format

There are different types of groups, intended for different kinds of messages to be transmitted in blocks 3 and 4. However, the main features must remain the same, i.e.

a) The first block in every group must contain the Programme Identification (PI) code
b) The first four bits of the second block of every group are allocated to a 4 bit code for identifying the group type (application). Groups are referred to types 0 – 15 according to the 4 bit binary code when converted to decimal. For each type, ranging from 0-15, a version can be described. The version is specified by the fifth bit of the second block as follows:
    i. $B_0 = 0$; The PI code is inserted in block 1 only. This is called version A. So the group is referred to as 0A, 1A etc (with the type and the version)
    ii. $B_0 = 1$. The PI code is inserted in block 1 as well as in block 3 of all group types. This is called version B.

The PI, PTY and the Traffic Programme information (TPI) occupy fixed positions in block 1 and 2. Hence they can be decoded without referring to any other block. Version B groups have an offset C' added to the third block , that is different from the offset C added to the third block in version A groups. Hence B_0 need not be referenced at all to ascertain the version of the group once the offset for block 3 is known if the blocks are error free. Any mixture of A and B groups can be sent in a single broadcast.

3 Demodulation of FM-RDS broadcasts

The schematic for the demodulation of the broadcast is shown in Figure 3.1. Each block is explained in detail in the following sections.

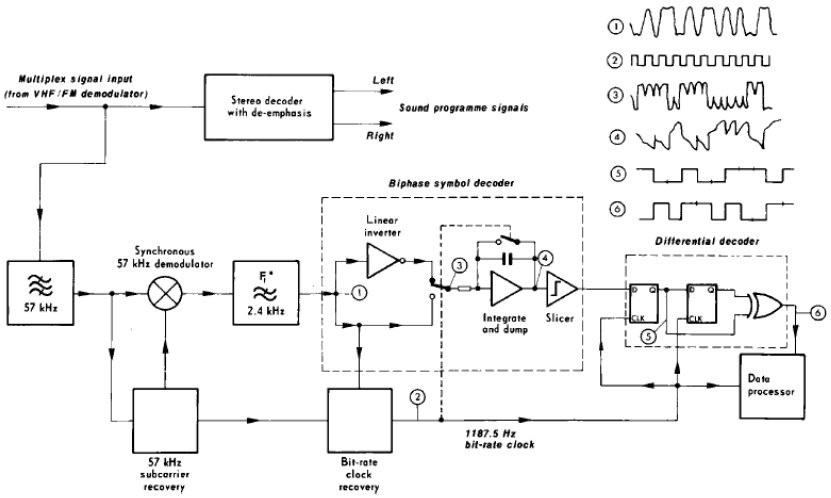

Figure 3.1. Demodulation scheme of the FM-RDS broadcast.

3.1 Stereo decoder

The signal from the FM demodulator is sent through a stereo decoder with de-emphasis. The L+R signal is recovered by passing the broadcast through a lowpass filter with a cutoff frequency of 15 kHz.

The L-R signal at 38 kHz is retrieved by passing the broadcast through a bandpass filter with center frequency 38 kHz and bandwidth of 30 kHz, mixing it with a tonal at 38 kHz to shift it back to baseband, and then applying a lowpass filter to the output of the mixer. It will be recalled that the Fourier transform of $\cos(\omega_c t)$ is given by (2.3). Hence mixing (multiplying) it with the difference signal at the 38 kHz band, say $d(t)$, gives

$$d(t)\cos(\omega_d t) \quad \overset{\mathcal{F}}{\leftrightarrow} \quad \frac{1}{2}[D(\omega-\omega_d)+\ D(\omega+\omega_d)]. \tag{3.1}$$

Referring to (2.4), the terms in the parentheses give

$$\begin{aligned} D(\omega-\omega_d)+\ D(\omega+\omega_d) &= \tfrac{1}{2}[G(\omega-2\omega_d)+G(\omega)]+\tfrac{1}{2}[G(\omega)+G(\omega+2\omega_d)] \\ &= G(\omega)+\tfrac{1}{2}[G(\omega+2\omega_d)+G(\omega-2\omega_d)] \end{aligned} \tag{3.2}$$

which when passed through a lowpass filter will give back the required L-R signal, since the engineering assumption is that the components centred at $2\omega_d$ are removed by the lowpass filter. Referring to (2.2), this can also be viewed as the result of multiplying $l(t)-r(t)$ with $cos^2(\omega_d t)$ (when both the modulation and demodulation processes are taken into account). This yields

$$d(t)\cos(\omega_d t) = [l(t)-r(t)]\,cos^2(\omega_d t) = \tfrac{1}{2}[l(t)-r(t)]\,[1+\cos(2\omega_d t)] \tag{3.3}$$

When this signal is passed through a lowpass filter with bandwidth 15 kHz, the $\cos(2\omega_d t)$ component is not retained as it is at a higher frequency than the bandwidth of the lowpass filter. This gives a time domain picture of the recovery process. Once the L-R and L+R signals are recovered, the L and R signals are acquired by half their sums and differences respectively.

3.2 Data channel (Physical layer) demodulation

3.2.1 RDS bandpass filter and mixer

The RDS signal is at 57 kHz and is hence first passed through a bandpass filter with centre frequency 57 kHz and a bandwidth of 4.8 kHz. It is then mixed with a tonal at 57 kHz to bring it back to baseband, after which it is passed through a low pass filter to retrieve the RDS signal, in the same way as the stereo difference signal is retrieved (described above). The only difference is that here the carrier is at 57 kHz, instead of at 38 kHz.

The lowpasss filter and the biphase symbol decoder together constitute a matched filter for the signal, corresponding to the cosine filter used in the transmitter side for shaping.

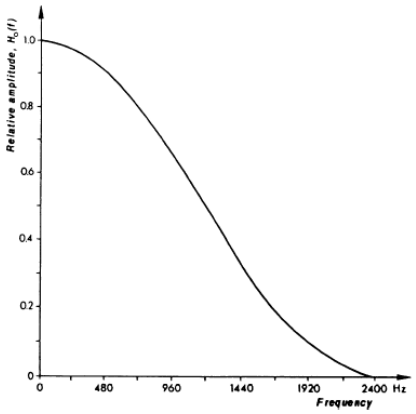

Figure 3.2. Combined response of receiver and transmitter filters

3.2.2 Bipolar sampler

The next block is the first phase of the process to decode the biphase symbols. The signal is sent through two paths. The first path inverts the signal, and the second passes the signal as it is. The sampler at the other end switches between these two paths at a frequency that is equal to the symbol frequency. This effectively acts like a bipolar sampler. Referring to Figure 2.5, it is clear that the output of the sampler will be positive for a biphase symbol representing logic 1, and negative for a biphase symbol representing logic 0. The output is symmetric about the $\frac{t_d}{4}$ time instance from the start of the bit for each bit.

3.2.3 Integrate and dump

The signal is then integrated over one time period of the bit. Symbols representing logic 1 will yield a positive output, while signals representing logic 0 will yield a negative output.

3.2.4 Slicer

A simple slicer is used to threshold the signal. The signal is sampled at every time period of the bitstream. Positive signals are are assigned logic 1, and negative signals are assigned logic 0.

3.2.5 Differential decoding

Differential decoding is done to recover the signal. The recovered signal is

$$r_k = e_k \oplus e_{k-1} \tag{3.4}$$

where r_k is the decoded, recovered bit at index k, and e_k is the received, encoded bit at index k. Referring to Table 2-1, this is equivalent to finding the difference between the two received bits, and hence even if both received bits are inverted the recoverd bit would be the same. The encoded, received bits at the transmission side are equivalent to the sum of the previous encoded bit in the stream and the current bit to be encoded, and it follows that the original information is recovered by finding the difference between the current and the previous received bit.

3.3 Baseband decoding (data-link layer)

At baseband, the first task is the synchronisation of the groups and blocks. This is achieved through the generation of a syndrome.

3.3.1 Syndrome calculation

For a received 26 bit stream (block) of data, say y, the syndrome is calculated as

$$s = y\mathbf{H} \tag{3.5}$$

where **H** is a parity-check matrix characterized by the equation

$$x\mathbf{H} = \mathbf{0} \tag{3.6}$$

where x is the 26 bit stream of data before the addition of the offset. **H** is given by

$$\mathbf{H} = \begin{vmatrix} 1000000000 \\ 0100000000 \\ 0010000000 \\ 0001000000 \\ 0000100000 \\ 0000010000 \\ 0000001000 \\ 0000000100 \\ 0000000010 \\ 0000000001 \\ 1011011100 \\ 0101101110 \\ 0010110111 \\ 1010000111 \\ 1110011111 \\ 1100010011 \\ 1101010101 \\ 1101110110 \\ 0110111011 \\ 1000000001 \\ 1111011100 \\ 0111101110 \\ 0011110111 \\ 1010100111 \\ 1110001111 \\ 1100011011 \end{vmatrix}.$$

or

$$[H] = \begin{Bmatrix} h_1 \\ \vdots \\ h_{26} \end{Bmatrix}$$

where h_n, n = 1, 2, 3, …26 are the rows of the matrix, each having 10 columns. The first 10 rows of **H** correspond to the Identity matrix. y is a row matrix with 26 columns, with the message bits in the first 16 columns, and the checkword + offset in the next 10 columns. x is a row matrix with 26 columns, with the message bits in the first 16 columns, and only the checkword in the next 10 columns, and is given by equation 2.12. **H** satisfies the property **GH = 0**, and hence from equation 2.12 $x\mathbf{H} = \mathbf{0}$.

Consider y as the received word, comprising of the information word and the checkword summed with the offset, and x as the transmitted codeword, that has the information and the checkword alone. The difference $y \oplus x$ is equivalent to the offset word that was added. Calculating the syndrome for the expression $z = y \oplus x$,

$$z\mathbf{H} = (y \oplus x)\mathbf{H} = y\mathbf{H} \oplus x\mathbf{H} = y\mathbf{H} \tag{3.7}$$

Hence, in effect, the syndrome calculated from the 26 bit word is equivalent to the syndrome calculated from the offset words if the transmitted block is error free.

3.3.2 Initial acquisition of group and block synchronization

For a syndrome generated from a received 26 bit word, if it matches with one of the offset words it is likely that the transmission was error free and the synchronization was perfect for that word. Synchronization is not certain because the match may also have been the result of an error that is equal to the offset, or due to parts of the information and codeword together corresponding to a recognized offset. The correspondence is shown in Table 3-1.

Table 3-1. Syndromes and corresponding offsets

Offset	Offset word $d_9, d_8, d_7, \ldots d_0$	Syndrome $S_9, S_8, S_7, \ldots S_0$
A	0011111100	1111011000
B	0110011000	1111010100
C	0101101000	1001011100
C'	1101010000	1111001100
D	0110110100	1001011000

To be almost certain of good synchronization, the following scheme is implemented. The syndrome is generated for every adjacent 26 bits as they arrive, shifting the start bit back by a single bit each time. If the offset matches one of the offset words shown in Table 3-1, a match is likely. If two matches are found, in a valid sequence for a group (i.e. A, B, C, (or C'), D) at a distance n x 26 bits apart (where n=1,2,3 etc) the blocks are said to be synchronized. Note that this condition for synchronization is not so strict as to inherently assume that all words are error free, which would render the checkwords meaningless. Synchronization is achieved using just two blocks that are assumed to be error free and that are a reasonable distance apart, as the probability

that a match would be found for two close blocks by chance is very small. The other blocks that are not used for synchronization may or may not be error free.

Once synchronization is achieved, the start bit is set for the groups and blocks, the corresponding order of offset words is noted and subtracted from each block's last 10 bits. The checkwords are thus received and error detection and correction is done on the blocks using syndrome calculation once again. This time, the syndrome will correspond to an actual error (for the blocks with errors) since the offset has been subtracted already.

3.3.3 Detection of loss of synchronization

The previous subsection dealt with acquisition of synchronization at the start of data reception. However, as the data is being received and processed after the initial synchronization, clock slips may still occur. The recommended way to deal with this is to use the PI (programme identification) code, which usually does not change on a given transmission. If a known PI code is received correctly, but is shifted by a bit, the clock slip can be corrected by compensating for it.

3.3.4 Programme identification code

The PI code is arranged in a way that is different from the United States' RBDS standard, so this is an important point to be noted. All codes are binary coded Hex numbers. Referring to the following figure,

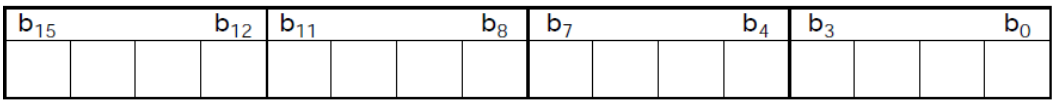

Figure 3.3. PI code layout

Bits b_{15}-b_{12} – Country code

India (ISO code) has a country PI symbol of 5 (hex), with Extended Country Code (ECC) F0. ECC codes must be sent atleast once every minute in to ensure that the PI code is rendered unique. They are carried in Variant 0 of type 1A groups.

Bits b_{11}-b_8 – Programme type in terms of area coverage

I: (International) The same programme is transmitted in other countries

N: (National) Transmitted throughout the country

S: Supra regional Transmitted throughout a large part of the country

R1 … R12: (Regional) Programme is available only in one region over one or more frequencies

L: (local) Local programme transmitted via single transmitter

Hex codes for the area coverage types are given in Table 3-2.

Bits b_7-b_0 – Programme reference number

These codes are determined by each country separately.

Table 3-2. Hex-coding rules for area coverage types

Area coverage code	L	I	N	S	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

3.3.5 Other miscellaneous codes and information

The first five bits of block two of every group describe the group type (refer to Figure 2.7). Group type essentially indicates the application of the data, and includes categories such as radio text, open data, clock and time, basic tuning and switching, etc. Each group type has its own format to be followed while sending information in blocks 3 and 4. The group types, and the formats in which block 3 and block 4 are filled, are described completely in [3].

15B groups carry some specific and basic information. The sixth bit of block 2 contains the traffic programme code, and the 12th bit carries the traffic announcement code. Different combinations of these two codes correspond to different ways in which the programme behaves, and the correspondence is shown in Table 3-3.

Table 3-3. Traffic codes and their applications

Traffic Programme code (TP)	Traffic Announcement code (TA)	Applications
0	0	This programme does not carry traffic announcements nor does it refer, via EON, to a programme that does.
0	1	This programme carries EON information about another programme which gives traffic information.
1	0	This programme carries traffic announcements but none are being broadcast at present and may also carry EON information about other traffic announcements.
1	1	A traffic announcement is being broadcast on this programme at present.

The 13th bit carries the music/speech bit. If the value is 0, the program carries speech. If the value is 1, the program is playing music.

The 14th bit carries the decoder identification control bit. The DI bit is transmitted as 1 bit in each type 15B group, for four groups. The 15th and 16th bits carry the address of the DI bit to be located, from among the four groups.

4 MATLAB® implementation and results

In this section, the results for each block or process is presented. The data is first generated and modulated, and then the corresponding inverse transforms are applied on the demodulating side to receive the signal. All codes that were written are available in the appendices, and are referred to when required.

The sampling frequency, F_s, for the entire implementation process is chosen as 237500 Hz. The value is chosen so that there are exactly 200 samples per symbol (logic bit). This is because the number of carrier cycles per bit is fixed by the protocol as 48 (refer Section 2.2). To satisfy the Nyquist criterion, at least 96 samples are required per bit. 200 samples per bit satisfy this criterion by a fair margin. Besides this, biphase symbol generation requires a samples-per-bit value that is divisible by four for digital implementation. This is because the bipolar sampler must switch every quarter of a time period of a bit. These two factors led to choosing 200 samples per bit.

The lengths (i.e. number of data points to be sampled and stored) is chosen to be the same for all the modulated data streams, the value being 950000 (L), while 4750 bits were generated for RDS. The values of frequency for which the frequency spectrum can be viewed correctly (without aliasing errors) is determined by the maximum frequency that is permissible according to the Nyquist criterion, which is half the sampling frequency, or $F_s/2$. So the frequency base ranges from $-F_s/2$ to $F_s/2$, with the number of data points on the frequency axis being L. To ensure that the frequency spectrum gives reasonably correct and precise values for amplitudes of spectra, the spacing between the points on the frequency axis would have to be such that small integers (1 or 2) are divisible by them, to represent different frequencies correctly. For example, 0.2 when multiplied with 4 gives 1, and is suitable to be used as the frequency spacing. 1 is not divisible by 0.3, however. If 0.3 is used as the frequency spacing, the value of the spectrum at 1000 Hz would not be represented in the digital implementation, and instead only the values at 999.9 and 1000.2 would be available, which are close but not exactly the same as the value at 1000 Hz. The frequency base spacing is $F_s/2\mathrm{L}$, which turns out to be 0.125 (if L is chosen as 9.5E5), which is a suitable value.

This choice of F_s and L were also such that the number of bits generated would be an integer. It is inconvenient and unrealistic to have to deal with a non-integer number of bits in simulation. There are 1187.5 bits per second according to the protocol. The spacing in the time domain being T_s=1/F_s, or , the number of seconds for which the simulation runs is LT_S, or 4 seconds. This is sufficient to be able to simulate an integer number of bits (4750 bits to be exact). Refer to the script 'main_rds.m' for the declaration of these parameters.

The audio signals $l(t)$ and $r(t)$ are modelled as simple sinusoidal 1000 Hz tones. A few runs were also done with smoothed streams of digital data to simulate lowpass signals. The RDS information (PI code, PTY code etc) is generated with random data, but the data-link layer (segmenting, checkwords and offsets) is properly implemented. An in-depth analysis of the results at each stage of the modulation is presented in the following subsections. Though signals are represented as analog, it must be kept in mind that they are in reality discrete, and sampled as described earlier.

4.1 Audio and pilot tone broadcast

The signals used as the audio for the left and right speakers are

$$l(t) = \sin(2000\pi t) \tag{4.1}$$

and

$$r(t) = \cos(2000\pi t) \tag{4.2}$$

The sum and the difference of these two signals at baseband, before modulation, are shown in Figure 4.1 for the first few sampling instances. DSB-SC modulation is done for the difference signal by multiplying it with a cosine signal at 38 kHz, as described in section 2.1. After modulation, the sum and difference signals are at baseband and mirrored around 38 kHz respectively. The pilot tone is given by

$$p(t) = \cos(38000\pi t) \tag{4.3}$$

The frequency spectrum of the simulated results of these operations in Figure 4.1 shows these signals clearly. The tones at 37 kHz and 39 kHz are due to the $l(t) - r(t)$ signal, while the tone at 1 kHz is due to the $l(t) + r(t)$ signal. The tone at 19 kHz is the pilot. The implementation is done using the function fm_modul.

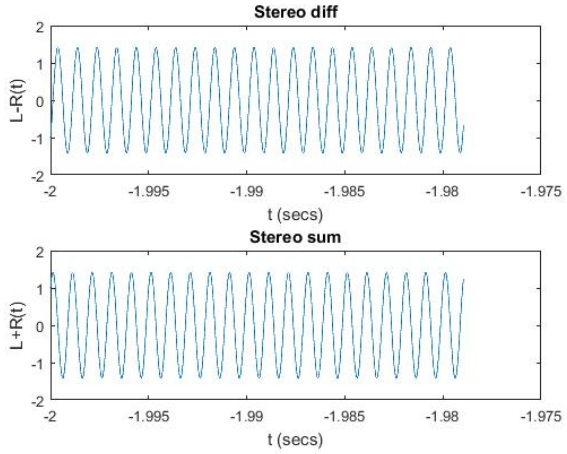

Figure 4.1. Sum and difference signals of simulated audio.

4.2 RDS broadcast

4.2.1 Data-link layer

The message and presentation layers are not shown as it is not necessary to actually generate meaningful PI (programme identification) and PTY (programme type) codes for this simulation. Note that group structure, block structure and checkword generation and offset addition are followed as per the protocol. The information is generated using the randi function (refer to generate_msgs.m in the appendices), and is divided into blocks and groups. The generator polynomial is used to generate the checkwords, and offsets are added to the corresponding blocks in each group, as described in Figure 2.7. Random information encoded in the format of block 1 is shown in Figure 4.2, as generated by an isolated snippet of the program. It will be recalled that there are four blocks, each of length 26 bits, in a group, and block 1 contains the PI code in the first 16 bits and has an offset word 0011111100 (called offset A) to be added to the CRC checkword in the next 10 bits. Inbuilt MATLAB® functions were used to generate the CRC code and add the offsets, using communication systems toolbox objects.

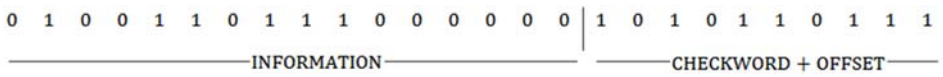

Figure 4.2. Block 1 with random information and proper CRC and offset

4.2.2 Physical layer implementation

Upon preparing the data, differential coding was performed using a communications systems toolbox object. The following figure shows the result of differential coding for a the first two blocks of an RDS bitstream (generated after the data link layer was implemented), using a snippet of MATLAB® code. Refer to lines 59-61 in main_rds.m for the code. The differential coding and decoding functions used were checked separately for their reliability and found to work as expected for test data.

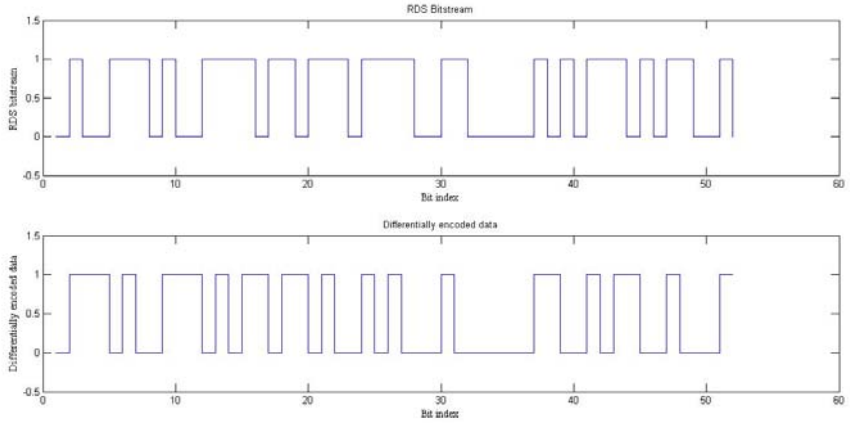

Figure 4.3. RDS bitstream and corresponding differentially encoded stream

Following this, NRZ to polar impulse conversion is done. First, the stream is converted from a string of 0s and 1s to a stream of -1s and 1s, respectively, so that it becomes easier to convert the data into positive and negative impulses. This is done by multiplying all the bits by 2 and subtracting 1 from each bit. As the bitstream is initially of length equal to the number of bits, one way to convert the stream of digital data into impulses is to upsample the stream by a factor of $\frac{T_b}{T_s}$, where T_b is the time delay between bits, and T_s is the sampling interval of the simulation. It will be recalled that there are 4750 bits, and hence the vector representing the bits is of length 4750. This factor is 200, as was chosen earlier. Hence upsampling causes the vector to become of length 9.5E5. Refer to main_rds.m and biphase_generator.m for the code.

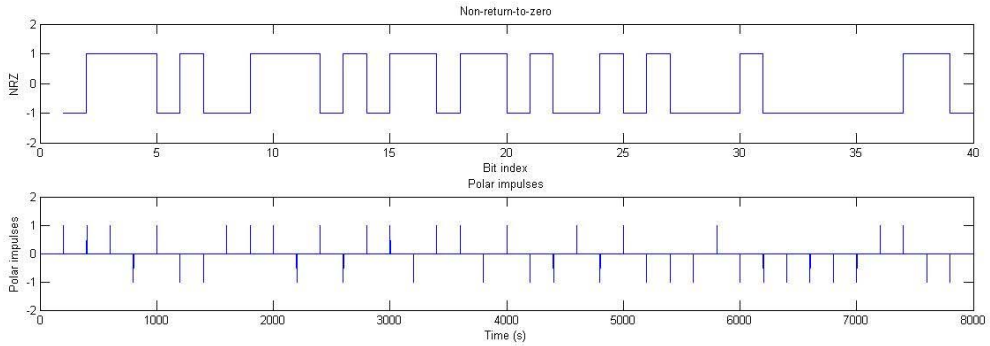

Figure 4.4. NRZ and polar impulses.

Note that with this process, the polar impulse signal can be treated as a continuous function in time that has been sampled discretely at a rate of $1/T_s$, in contrast to how bits are seen in terms of their indices and positions in words, and not their time instances. This is reflected in the x-axis labelling of Figure 4.4.

The impulses are now converted to odd impulse pairs, with an odd impulse pair representing each bit. The impulse train is delayed by 100 samples (half the number of samples per bit) and the delayed train of impulses is subtracted from the original. Refer to biphase_generator.m. The result is shown in Figure 4.5.

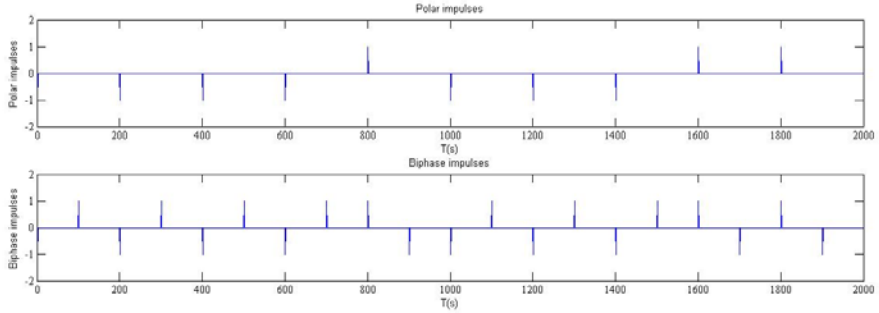

Figure 4.5. Polar and bipolar impulses.

The next step is pulse shaping. The impulse response and the frequency response of the pulse shaping filter are shown in Figure 4.6.

andFigure 4.7 respectively. It can be seen that the frequency response is nearly zero at frequencies greater than 2400 Hz, as required.

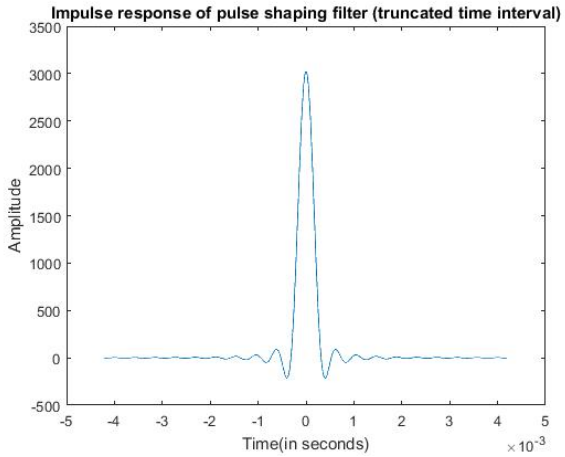

Figure 4.6. Impulse response of pulse shaping filter.

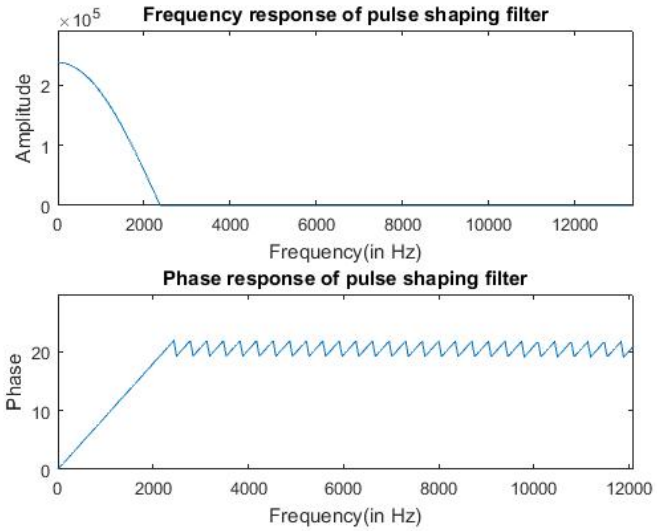

Figure 4.7. Frequency response of pulse shaping filter (magnitude and phase)

Biphase symbols are generated by convolving this filter with the biphase impulses. The filter is generated, and the biphase symbols produced, in the function pulse_shape.m. Some biphase impulses and their corresponding biphase impulses are shown together in Figure 4.8. The biphase symbols are labelled as RDS to indicate that these are the symbols that will be transmitted in the broadcast finally.

The spectrum of these biphase symbols is shown in Figure 4.9. This data is then multiplied with a carrier at 57 kHz. The time domain response of this signal is shown in Figure 4.10.

Now the audio signals, the pilot tone, and the RDS signal, are all added up to produce the signal intended for frequency modulation and subsequent transmission. The amplitude spectrum of the resultant signal is shown in Figure 4.11, and the power spectrum is shown in Figure 4.12.

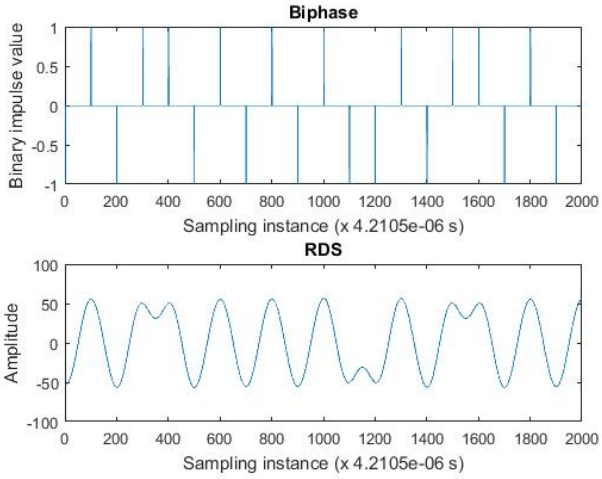

Figure 4.8. Biphase symbols and shaped signal

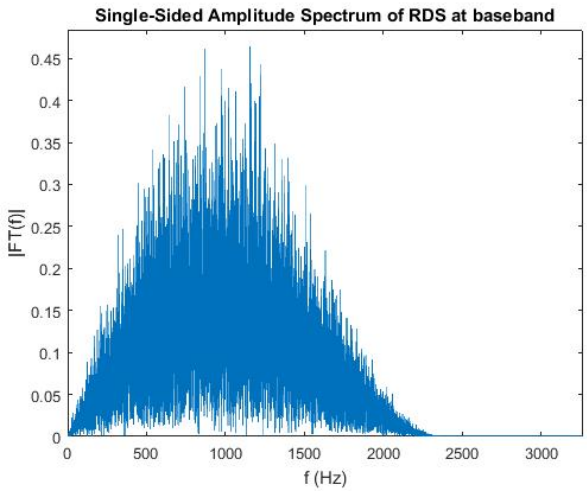

Figure 4.9. Spectrum of RDS data at baseband.

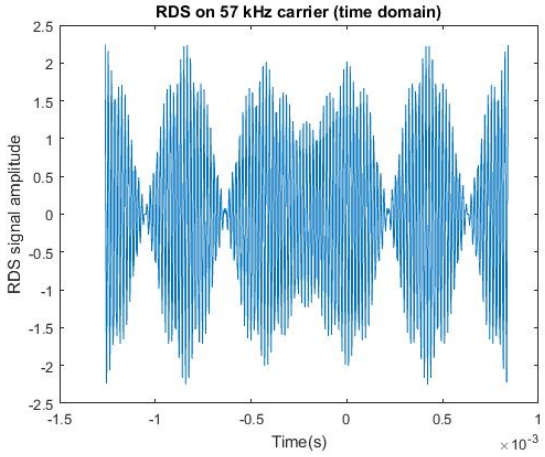

Figure 4.10. Modulated RDS data (time domain).

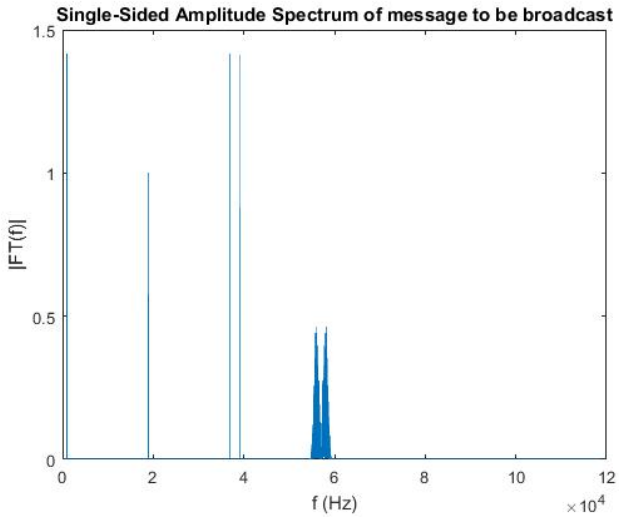

Figure 4.11. Single sided spectrum of modulated signal.

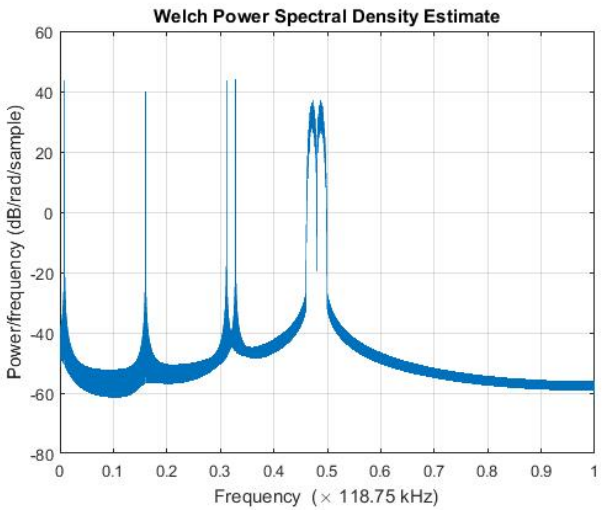

Figure 4.12. Power spectral density estimate of modulated signal.

4.3 Frequency modulation and demodulation

A frequency of 100 MHz is chosen, with a frequency deviation of 50, for frequency modulation. The signal is modulated and demodulated using the MATLAB® functions fmmod and fmdemod. It is observed that the frequency demodulated signal is identical to the one transmitted, so no further comments are required. Refer to lines 92-104 in main_rds.m for the implementation.

4.4 Demodulation of the broadcast

4.4.1 Audio demodulation and pilot recovery

The mono audio signal present at baseband is extracted by passing the received broadcast through a lowpass FIR filter with a cutoff frequency of 15 kHz. The code is found in fmrds_demod.m. The frequency response of this lowpass filter is shown in Figure 4.13.

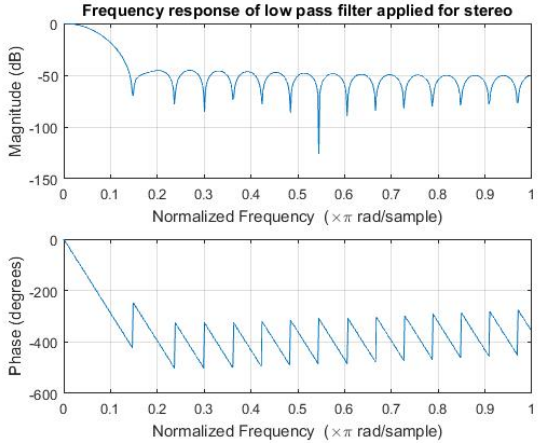

Figure 4.13. Frequency response of lowpass filter (magnitude and phase)

To extract the difference signal, pilot recovey and subsequent carrier recovery is required. A bandpass filter centred at 19 kHz, with a bandwidth of 2 kHz, is used to recover the pilot. The carrier is the second harmonic of the pilot tone, and thus a simple way of recovering it is to simply square the pilot tone, and subtract 1 (constant) from it. This follows from the trigonometric identity

$$\cos(2\theta) = 2\cos^2\theta - 1. \tag{4.4}$$

The carrier at 38 kHz is thus obtained, following the frequency and phase of the carrier used in the transmission. Similarly, the carrier at 57 kHz is approximately obtained by cubing the pilot signal, amplifying it by a factor of 4, and subtracting 3 times the pilot signal from it. This follows from the identity

$$\cos(3\theta) = 4\cos^3\theta - 3\cos\theta \tag{4.5}$$

The recovered pilot and carriers are shown in Figure 4.14.

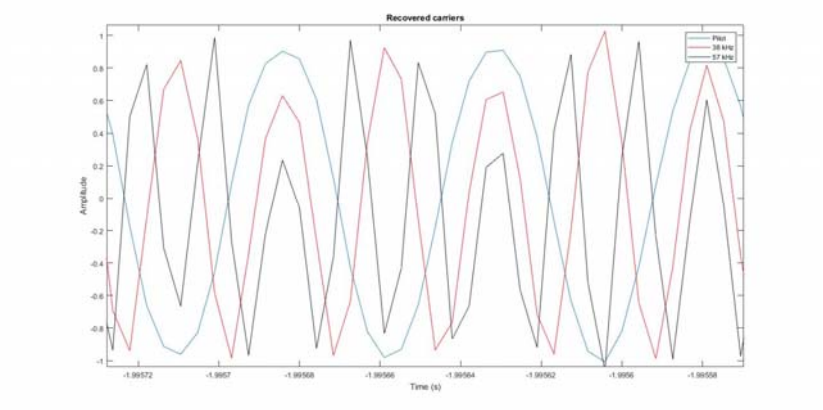

Figure 4.14. Recovered carriers

A phase-locked loop is unnecessary here because a separate pilot tone is provided, and the carriers are integer multiples of the frequency of the pilot tone. This allows for convenient application of trigonometric identities, by which frequency and phase offsets are accounted for. Comparison with the performance of a second order PLL shows that usage of the trigonometric identities gives better results. For a second order PLL with proportional gain constant α and integral gain constant β, it can be shown that for stability

$$0 \leq \alpha < 2 \tag{4.6}$$

$$0 \leq \beta < 1 - \frac{\alpha}{2} - \sqrt{\frac{\alpha^2}{2} - 1.5\alpha + 1} \tag{4.7}$$

A lower value of α would reject noise better, but result in a larger steady state phase error. A larger value of β is desirable to reduce stead state frequency errors, but it is limited by the stability criterion. A PLL was created with α = 0.05 and β = 0.006 to extract the 38 kHz and the 57 kHz carriers. Figure 4.15 shows the result for a frequency deviation of 100 Hz in the pilot. There is a phase error in the PLL output due to the bandpass filter used to extract the second harmonic from the squared pilot signal. Figure 4.16 shows the amplitude spectrum of the same. It is seen that there are spurious frequencies in the spectrum of the signal recovered by the PLL. This may be because the data is finite in time. Usage of the trigonometric identities overcomes these difficulties and makes use of the fact that the carriers are harmonics of the pilot.

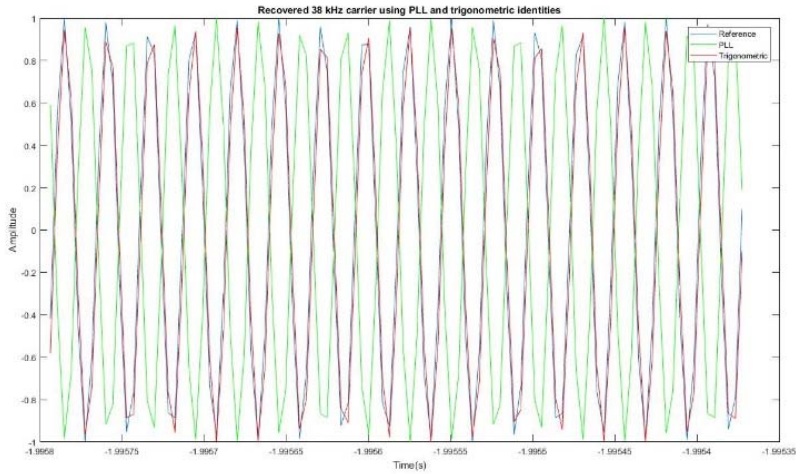

Figure 4.15. Recovered 38 kHz using PLL and trigonometric identities

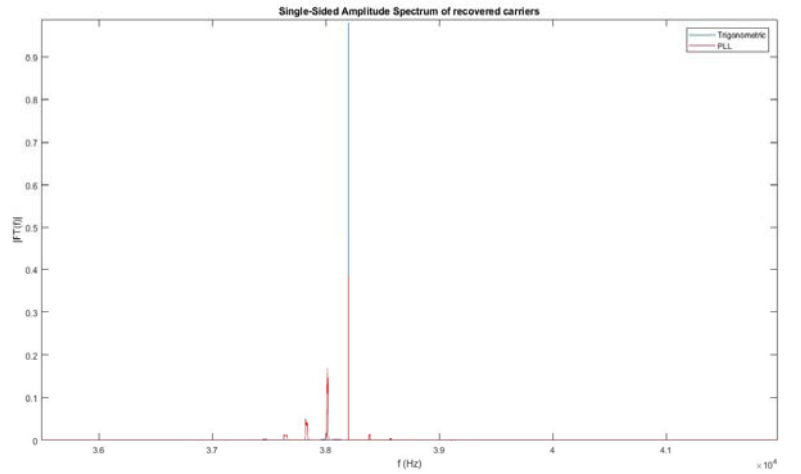

Figure 4.16. Amplitude spectrum of recovered signals

19 kHz is 16 times the clock frequency (1187.5 bits/s) according to the protocol, and hence the clock is also recovered from the pilot. The difference signal at 38 kHz is first extracted by an IIR bandpass filter. This is multiplied with the recovered 38 kHz carrier to shift it baseband, after which the lowpass filter used to extract the mono audio (refer to Figure 4.13) is again used to filter out the difference signal. The results (at baseband) of these operations are shown in Figure 4.17.

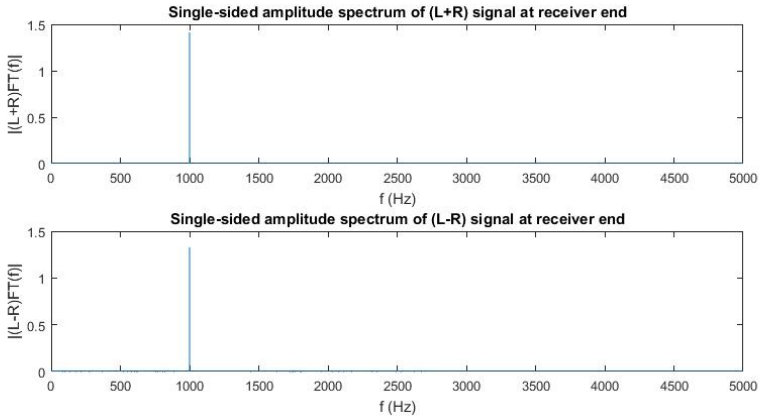

Figure 4.17. Amplitude spectra of sum and difference signals at baseband

4.4.2 RDS digital data demodulation and recovery

The RDS signal at 57 kHz is first extracted by an IIR bandpass filter centred at 57 kHz with a bandwidth of 4.8 kHz. This is then multiplied with recovered 57 kHz carrier, to shift it to baseband. An FIR lowpass filter with a cutoff frequency of 2.4 kHz is then used to remove other unwanted components of the signal. The implementation is found in fmrds_demod.m. The results at the receiver end showed an impulse in the spectrum at 2375 Hz. This is twice the frequency of the bit rate, but is equal to the rate of the impulse pairs. This is shown in Figure 4.18.

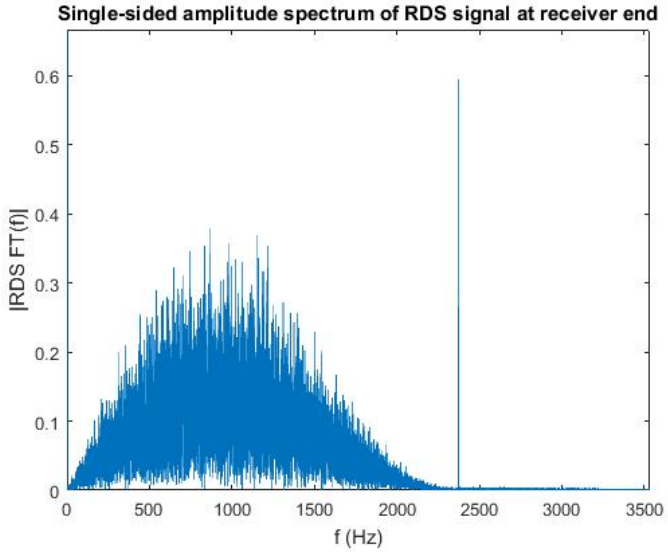

Figure 4.18. Baseband RDS spectrum at receiver end

The average value for the RDS signals that were transmitted are of the order of 1E-4. It is not exactly zero because the pulses in transmission are truncated on both sides (at the first bit and at the last bit), and also because the signal is digitally sampled. Though this is insignificant while transmitting, due to the non-ideal nature of the bandpass filter, the recovered carrier, and the lowpass filter, small distortions can cause the mean to increase slightly. For uncorrelated message symbols with pulse spectrum $P(f)$ and amplitude autocorrelation $R_a(n)$, with mean m_a and variance σ_a, it can be shown that [8]

$$G(f) = {\sigma_a}^2 r|P(f)|^2 + (m_a r)^2 \sum_{n=-\infty}^{n=\infty} |P(nr)|^2 \delta(f - nr) \tag{4.6}$$

where $G(f)$ is the power spectral density. Hence for signals with non-zero mean, peaks appear at multiples of the signaling rate r in the power spectral density. A similar phenomenon is seen here, in that there is a peak in the amplitude spectrum at 2375 Hz. In any case it is at the edge of the spectrum and not required for signal recovery, but it can be used for clock recovery.

The output for one of the simulation runs is shown in the first subplot of Figure 4.19. After this, bipolar sampling is performed. Since each bit is represented by an odd impulse pair, the period of the sampler must be equal to the period of the bits, which is equal to 1/1187.5 s. The implementation is found in bipolar_switch.m. The result of bipolar sampling for a few biphase symbols is shown in Figure 4.19.

.

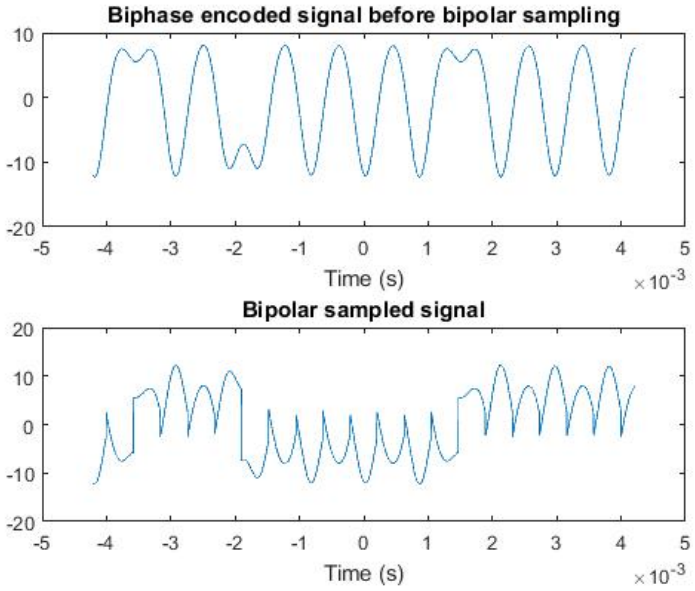

Figure 4.19. Bipolar sampling.

This is followed by integration and dumping to generate a positive value for logic 1 symbols, and a negative value for logic 0 symbols. Symbols are integrated over a period, and then the value of the integral is reset for the next symbol. The output is shown in Figure 4.20 for the data shown in Figure 4.19.

.

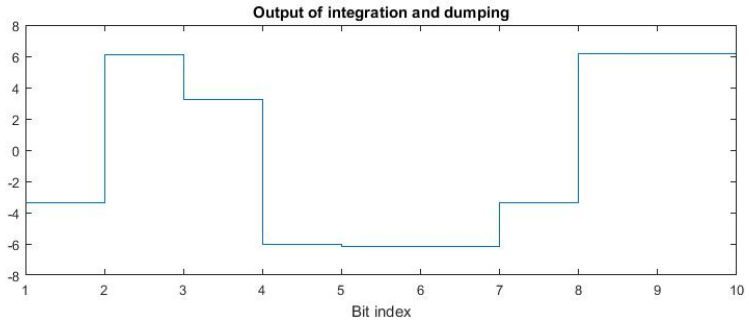

Figure 4.20. Result of integrate and dump

The output is then passed through a simple thresholder, that compares the signal with 0. The symbols are assigned a value of 1 if they are more than 0, and 0 if they are less than 0. The

values of this operation are ideally the differentially encoded information. The output is shown in Figure 4.21, below the data that was sent. It is clear that the two are identical, which means that the demodulation and recovery was successful.

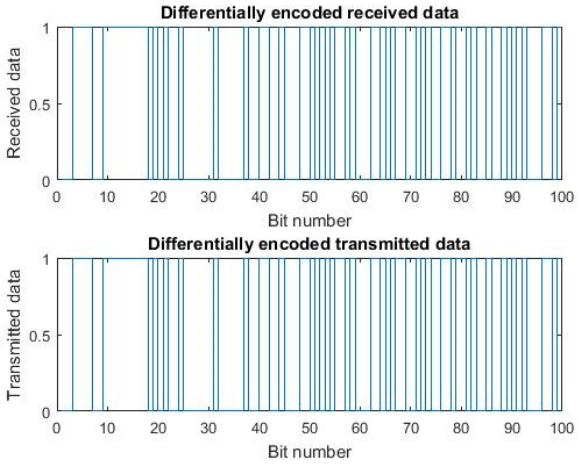

Figure 4.21: Transmitted and received encoded data

These bits are then differentially decoded using a MATLAB® communications system toolbox object. User defined routines were also written for differential encoding and decoding (refer to diff_deco.m and diff_enco.m) but were not used in the final simulation. Encoded data for one of the simulation runs (obtained after demodulation and biphase symbol extraction) and the data extracted from it by decoding, are shown together in Figure 4.22.

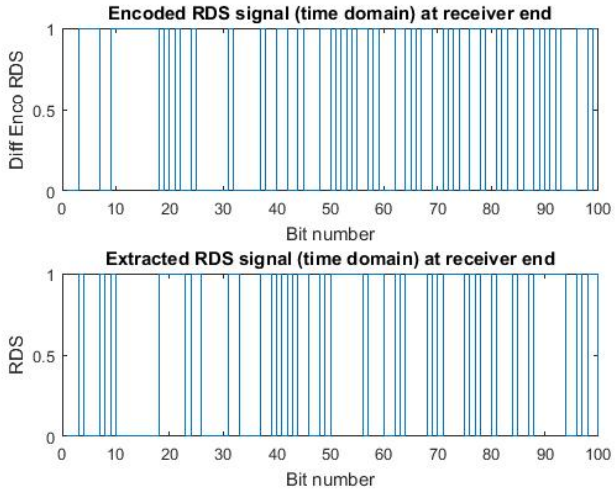

Figure 4.22. Differentially encoded and extracted data.

4.5 Noise analysis

The number of errors is consistently zero for different runs of the simulation when the above scheme is followed, for all 4750 bits. However, this is an ideal scenario. In an actual broadcast, there will always be noise in the channel. An analysis of the robustness of the scheme to noise is performed. Additive white Gaussian noise is added to the signal after modulation, with different SNRs (signal to noise ratios), where the SNR is measured with respect to the entire

broadcast (i.e. including the audio and pilot). Figure 4.23 shows the number of errors versus the noise level.

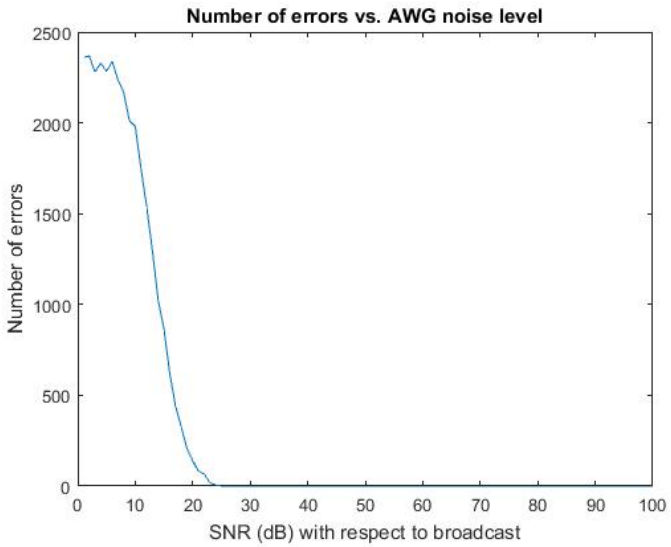

Figure 4.23. Number of errors versus AWG noise level

At at SNR of around 15 dB, acquisition of synchronization (refer to section section 4.6.1) becomes impossible, and it is impossible to correctly find the block position in the group, because there are no two blocks less than 72 bits apart that have an error free offset. Consequently, information extraction and error detection can no longer said to be meaningful below around 15 dB.

4.6 Data-link layer

4.6.1 Acquisition of synchronization

As the RDS stream arrives, it is broken up into groups of 104 bits that are adjacent to each other, each having four blocks as described earlier. The syndrome is calculated for each incoming block, and compared with the syndrome expected from the known offset words (A,B,C,C' and D). If a match is found, it is likely that the stream is synchronized. So, if a match is found, instead of moving to the next bit to study the block that starts from that bit, 26 bits are skipped to look for the next block in the structure. If a match is found for that block, for the offset that is expected for adjacent blocks (knowing the offset that was found for the first block) the nearest group (not block) start location is returned by the function synchronize.m.

In case the immediately adjacent block's syndrome does not match with the syndrome that is expected for the adjacent block, another 26 bits are skipped, and then the syndrome that corresponds to a block that is one block away from the initial block is searched for. In this way, a maximum of 3 blocks (26 x 3 = 72 bits) are skipped in search of a syndrome that corresponds to the location of that block with respect to the first block.

For example, if the syndrome corresponding to offset C is found for the block starting at the bit the loop is at in the current iteration, the algorithm skips to the next block (26 bits away) and checks if the syndrome for that block corresponds to offset D. If yes, a match is found, and the start bit for the nearest A block is returned. If no, another 26 bits are skipped, and that block is checked for the syndrome for offset A (as offset A is expected for the block that is one block away from an offset C block). If here, and at the next block (offset B), no match is found, the algorithm moves to the next bit, adjacent to the starting location of the block that had matched C in the first

place. In this simulation, the synchronized start bit was always found at 1 for SNRs that were above 20 dB. Errors cause the start bit to keep shifting until around an SNR of 15 dB, when no start bit is found in the stream (due to errors in the offsets), and the default start bit value of 1 is returned by synchronize.m. After this, depending on the AWGN it may or may not be possible to find a start bit, which is reflected in the peaks and falls in Figure 4.24, the falls almost certainly indicating that the default value of 1 was returned as the function could not find a start location.

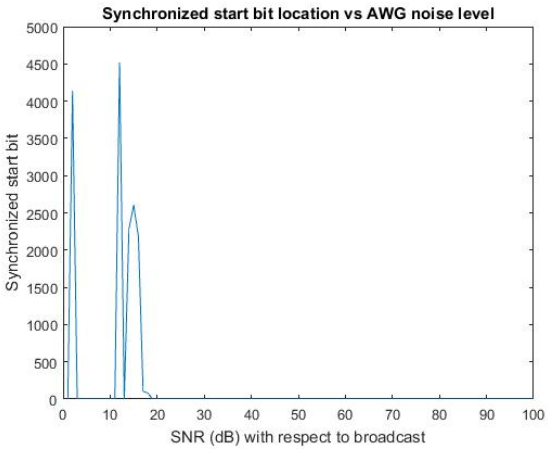

Figure 4.24. Synchronized start bit location vs SNR

4.6.2 Information extraction and error checking

Once synchronization is achieved, offsets are subtracted from each checkword. Syndromes are calculated for each checkword and errors are detected. This operation was implemented using the CRC system object provided by MATLAB® with appropriate segmenting of groups and blocks. In this simulation, there were no errors.

Correction of clock slips during streaming can also be done using the clock slip value returned by bit_slip.m, but for that a PI code would be required with an actual broadcast.

5 Simulink® implementation

The entire scheme is also implemented in Simulink®. The block diagram of the modulation scheme is shown in Figure 5.1.

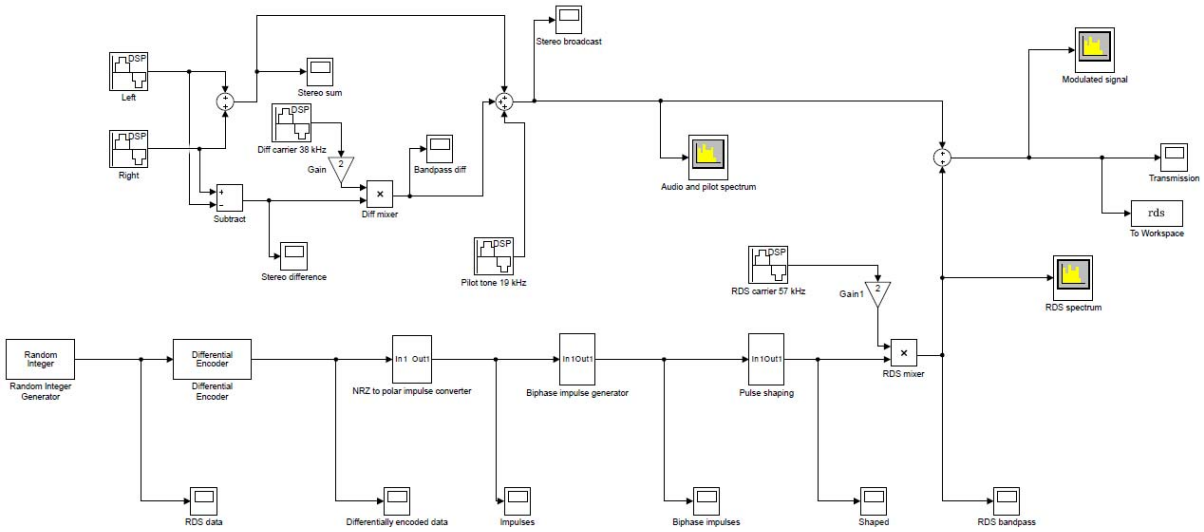

Figure 5.1. Simulink® model for modulation of signal

The NRZ to polar impulse subsystem is shown in Figure 5.2. The pulse shaping subsystem is shown in Figure 5.4.

The Simulink® model for demodulation is shown in Figure 5.5. The audio recovery subsystem is shown in Figure 5.6, and the RDS recovery subsystem is shown in Figure 5.7.

.

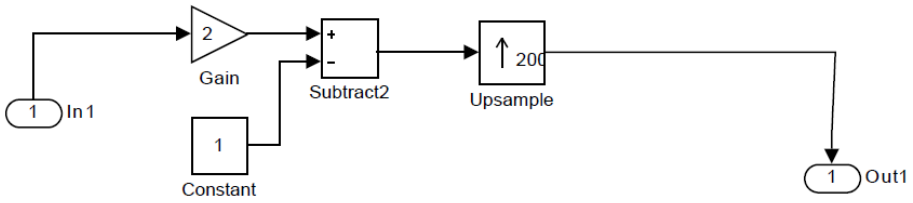

Figure 5.2. NRZ to polar impulse converter subsystem

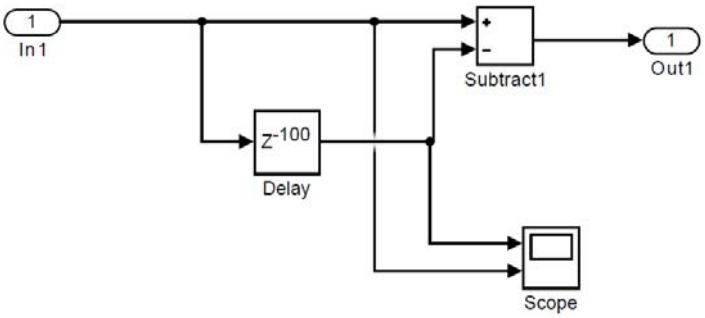

Figure 5.3. Biphase impulse generator

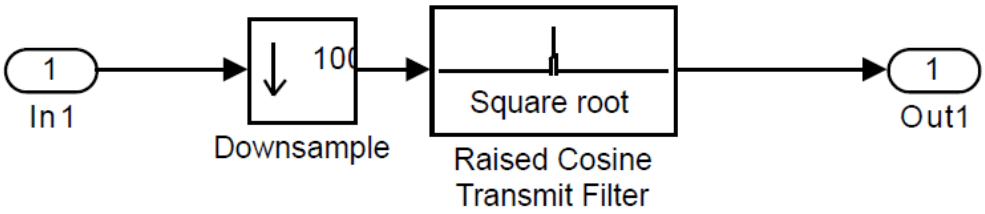

Figure 5.4. Pulse shaping sybsystem

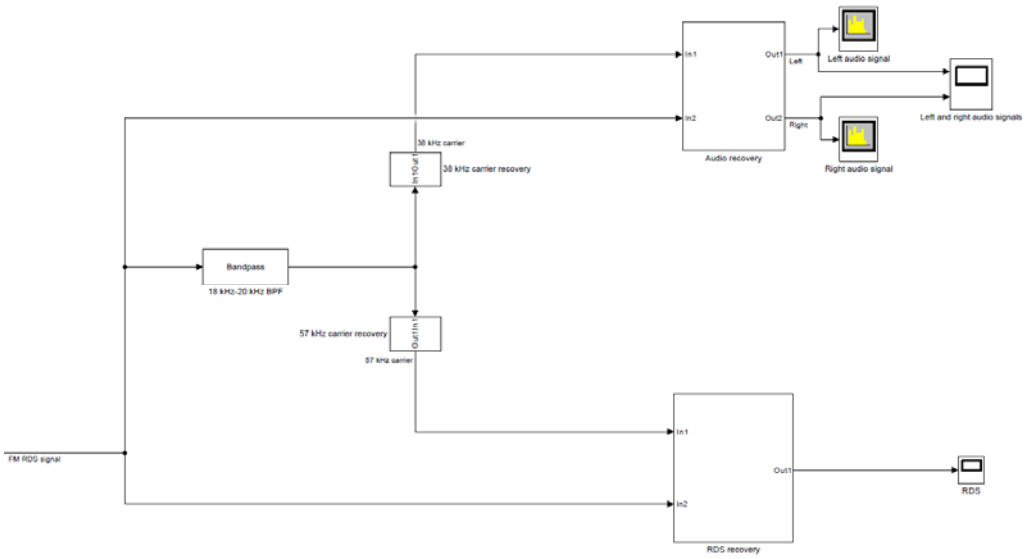

Figure 5.5. Simulink model for demodulation.

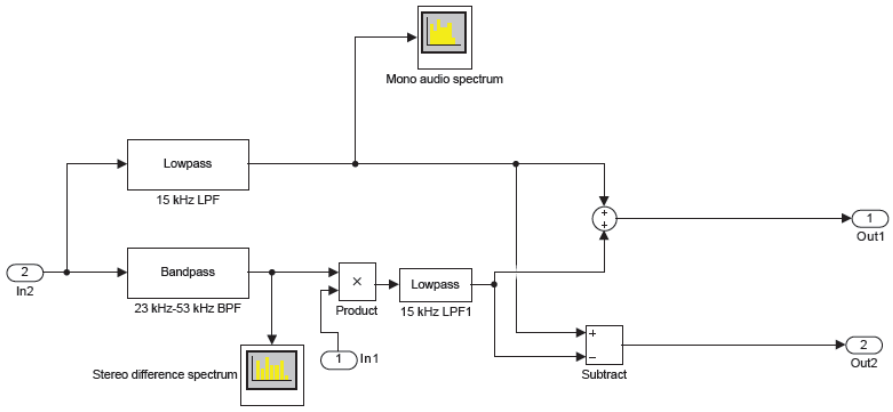

Figure 5.6. Audio recovery subsystem

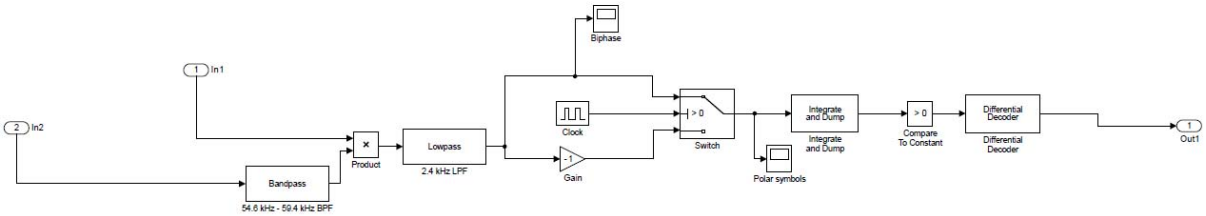

Figure 5.7. RDS recovery subsystem

6 Conclusion

The entire FM-RDS protocol is studied and implemented on MATLAB®. Results show that the software implementation is viable and robust. The performance of the scheme in the presence of noise is studied, which is an addition to literature. Carrier recovery is done using trigonometric identitities, without the need for a phase-locked loop. An in-depth discussion of the protocol and results is presented, and can be used to build applications in other areas as well.

7 References

[1] J. C. Maxwell, "A Dynamical Theory of the Electromagnetic Field," *Philosophical Transactions of the Royal Society of London 155,* pp. 459-512, 1865.

[2] The RDS forum, "March 2009: RDS is now 25 – the complete history," The RDS Forum, Geneva, Switzerland, 2009.

[3] CENELEC, "European standard EN 50067: Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz.," CENELEC, April 1998.

[4] N. R. S. Committee, "United States RBDS standard," 1998.

[5] IEC, "IEC 62106:2015: Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 MHz to 108,0 MHz," IEC, 2015.

[6] S. S. Shende, "A MATLAB®-Based FM Demodulator for the Radio Broadcast Data System," Boise State University Theses and Dissertations. 85, 2010.

[7] L. Li, G. Xing, L. Sun, W. Huangfu, R. Zhou and H. Zhu, "Exploiting FM radio data system for adaptive clock calibration in sensor networks," in *'Proceedings of the 9th international conference on Mobile systems, applications, and services', Pages 169-182*, 2011.

[8] A. B. Carlson, P. B. Crilly and J. C. Rutledge, *Communication systems*, Mc-Graw Hill, 2002.

[9] W. Peterson and D. Brown, "Cyclic code for error detection," *Proceedings of the IRE,* pp. 228-235, 1961.

APPENDICES

MATLAB® SCRIPTS AND FUNCTIONS

S.no.	File name	Purpose
1.	main_rds.m	Main script
2.	generate_msgs.m	Function to randomly generate information, append checkwords, arrange in blocks, and add corresponding offsets.
3.	diff_enco.m	Function to perform differential encoding.
4.	biphase_generator.m	Function to generate biphase symbols from polar impulses.
5.	pulse_shape.m	Function to shape the symbols for transmission.
6.	fmrds_modulate.m	Function to modulate monoaudio, stereo and RDS signals, and add to pilot tone.
7.	fmrds_demod.m	Function to demodulate received signal.
8.	bipolar_switch.m	Function to switch and sample received RDS pulses.
9.	synchronize.m	Function to ascertain scynchronized start bit.
10	syndrome.m	Function to generate syndrome for given block.
11	diff_deco.m	Function to perform differential decoding.
12	data_decode.m	Function to perform CRC checking and information extraction.
13	bitslip.m	Function to check for loss in synchronization while receiving.
14	message_display.m	Function to show information in readable format.
15	pll.m	Function to perform PLL carrier recovery.

```
%% ============================  main_rds.n ============================== %%
% Purpose of script:                  Simulation of Modulation and Demodulation of an↙
FM RDS broadcast
% User-defined Functions called:
%                                     generate_msgs
%                                     biphase_generator
%                                     pulse_shape
%                                     fmrds_modulate
%                                     fmrds_demod
%                                     synchronize
%                                     data decode
%                                     message_display
%
% Author:                             Joshua Peter Ebenezer
% Date of creation:                   1st December, 2017
%

%% ------------------------ Sequence of flow ------------------------- %%

% Script - main_rds.m
%   main function
%   Audio is generated
%   call function generate_msgs.m
%                   Messages are generated (random information with proper checkwords↙
and
%                   offsets)
%   Differential encoding
%   call function biphase_generator.m
%                    Converts NRZ to bipolar impulses
%   call function pulse_shape.m
%                    Shape the data using raised cosine filter
%   call function fmrds_modulate.m
%                    Modulate data with carriers and add audio and RDS
%   Frequency modulation & demodulation/ Noise
%   call function fmrds_demod.m
%                   Carrier recovery, demodulate data
%   Differential decoding of RDS data
%   call function synchronize.m
%                   Initial acquisition of synchronisation (start bit)
%   call function data_decode.m
%                   CRC decoding and error finding, information extraction
%   call message_display.m
%                   Display the information in a user-friendly format

%% ------------------------------------------------------------------------%%

clc;
clear;
close all;
%%
% pilot frequency is at 19kHz
pilot_freq = 19000;
% number of sampling instances
L = 9.5e5;
% frequency of sampling
```

```
Fs = 2.375e5;
% time step
Ts = 1/Fs;
% last sampling instance
tmax = (L/2)*Ts;
% sampling time array
t = (-L/2:L/2-1)*Ts;
% frequency
f = Fs*(0:(L/2))/L;
%% -------------------- Generation of audio signal ------------------- %%
% message signal - single tone

left = sin(2*pi*1000*t);
right = cos(2*pi*1000*t);

%% --------------  Generation of RDS data (data-link layer) --------------%
% clk freq of rds data is 3 times carrier frequency divided by 48 (1187.5
% bits/second)
rds_stream_length = round(2*tmax*pilot_freq*3/48);

% number of messages (that can be received/sent)
% bitstream size may not be equal to a multiple of the number of messages
no_msgs = floor(rds_stream_length/104);

% set option to 1 if reading from file
read_from_file = 0;

% group version - keeping it constant as version A
version = 'A';

[rds_bitstream,information] = generate_msgs(no_msgs,read_from_file,rds_stream_length,↙
version);

%% ----------------- Differential encoding of the bitstream ------------ %%
encode = comm.DifferentialEncoder;
tx_code = encode(rds_bitstream.');
tx_code = tx_code.';

%% ------------------ Data-channel spectrum shaping --------------------- %%
% convert NRZ to polar impulses
impulses = 2*tx_code-1;

% generate biphase symbols
biphase = biphase_generator(impulses);

% send the biphase symbols through the pulse shaper
rds = pulse_shape(biphase,L,Fs);

%% -------------------- Generate signal for transmission --------------- %%
fm_rds_signal = fmrds_modulate(left,right, rds, pilot_freq,L,Fs);

%% --------------------- Find and display fft of broadcast ------------- %%
fm_rds_Y = fft(fm_rds_signal);
% double sided spectrum
P2 = abs(fm_rds_Y/L);
```

```
% single sided spectrum
FM_RDS_FT = P2(1:L/2+1);

% recover exact amplitudes
FM_RDS_FT(2:end-1) = 2*FM_RDS_FT(2:end-1);

figure;
plot(f,FM_RDS_FT)
title('Single-Sided Amplitude Spectrum of message to be broadcast');
xlabel('f (Hz)');
ylabel('|FT(f)|');

%% ------------------------- Frequency modulation ---------------------- %%
% considering Fc as 100 MHz
fm_centre_freq = 100e6;
% sampling frequency is 4 times Fc
fm_sampling_freq = 400e6;
% frequency deviation
freqdev = 50;

% modulating signal to generate broadcast
% fm_broadcast = fmmod(fm_rds_signal,fm_centre_freq,fm_sampling_freq,freqdev);

%% ------------------------------ Noise ------------------------------ %%

% to hold bit indices at which errors occur for 100 SNR values
sig_error = zeros(1,100,4750);
% to hold number of errors for 100 SNR values
num_err = zeros(1,100);
% to hold average absolute error for 100 SNR values
avg_err = zeros(1,100);
% to hold the start locations for 100 SNR values
start = ones(1,100);

% SNR is fixed here as 50 dB. A for loop can be written here to observe
% outputs for 100 diff. SNR values (from 1 to 100)
snr = 50;

% add AWGN
fm_noisy_channel = awgn(fm_rds_signal,snr, 'measured');

%% ------------------------ Frequency demodulation --------------------- %%
% fm_received = fmdemod(fm_noisy_channel, fm_centre_freq, fm_sampling_freq, freqdev);

%% ------- View spectrum of received signal after FM mod&demod or noise --------- %%
fm_demod_fft = fft(fm_noisy_channel);
% double sided spectrum
P2 = abs(fm_demod_fft/L);
% single sided spectrum
FM_DEMOD_FT = P2(1:L/2+1);

% recover exact amplitudes
FM_DEMOD_FT(2:end-1) = 2*FM_DEMOD_FT(2:end-1);

figure;
```

```
plot(f,FM_DEMOD_FT)
title('Single-Sided Amplitude Spectrum of message after frequency↙
demodulation/noise');
xlabel('f (Hz)');
ylabel('|FT(f)|');


%% ---------------------- Demodulate received signal ------------------ %%
[left_rec, right_rec, pilot, rx_encoded] = fmrds_demod(fm_noisy_channel,Fs,L);

%% ---------------------- Error in Audio ------------------------------- %%
% find the error in the signal
aud_error = left-left_rec;

% average error
avg_aud_err = sum(abs(aud_error(:)))/length(aud_error);
% fprintf('Average error = %f for audio \n',avg_aud_err);

%% ---------------- Plot the received and transmitted signals ---------- %%
figure;
subplot(2,1,1);
stairs(rx_encoded(1:100))
title('Differentially encoded received data');
xlabel('Bit number');
ylabel('Received data');

subplot(2,1,2);
stairs(tx_code(1:100))
title('Differentially encoded transmitted data');
xlabel('Bit number');
ylabel('Transmitted data');

%% -------- Find the error in the encoded digitised received data ------ %%
code_error = rx_encoded - tx_code;
figure;
stairs(code_error);
title('Error in encoded received data');
xlabel('Bit number');
ylabel('Error in received signal');

%% ------------------------ Decode RDS data ---------------------------- %%
% perform differential decoding
decode = comm.DifferentialDecoder;
rx_bitstream = decode(rx_encoded.');
rx_bitstream = rx_bitstream.';

%% -------------------- Acquisition of synchronisation ----------------- %%
start_location = synchronize(rx_bitstream);

%% ---------------------- Data-link layer decoding -------------------- %%
[rx_information, info_error] = data_decode(rx_bitstream, start_location);

%% ------------------ Find and display error in information ------------ %%
figure;
stairs(info_error);
```

```
title('Error in received messages');
xlabel('Word number');
ylabel('Error in received messages');

%% Display the time-domain differentially encoded RDS and the RDS signals
figure;
subplot(2,1,1);
stairs(rx_encoded(1:100))
title('Encoded RDS signal (time domain) at receiver end');
xlabel('Bit number');
ylabel('Diff Enco RDS');

subplot(2,1,2);
stairs(rx_bitstream(1:100))
title('Extracted RDS signal (time domain) at receiver end');
xlabel('Bit number');
ylabel('RDS');

%% Find the error in the received data wrt transmitted data
% find the error in the signal
sig_error(1,100-snr+1,1:4750) = rx_bitstream-rds_bitstream;
% find number of errors
num_err(1,100-snr+1) = length(find(sig_error(1,100-snr+1,1:4750)~=0));
% average error
avg_err(100-snr+1) = sum(abs(sig_error(1,100-snr+1,1:4750)))/4750;
fprintf('Average error = %f and number of errors = %d \n',avg_err(100-snr+1), num_err↙
(100-snr+1));
start(100-snr+1) = start_location;


%% ------------------------ Message presentation layer ----------------- %%
message_display(rx_information);
```

```
function [ rds_bitstream, information ] = generate_msgs( no_msgs,read_from_file,↙
rds_stream_length,version )
%GENERATE_MSGS returns the RDS bit stream corresponding to a message signal
%   INPUT arguments:        number of messages, option to read
%                           message from file or to generate pseudo data,
%                           length of bitstream allowed, rds version (A or
%                           B)
%
%   OUTPUT arguments:       information words and checkwords concatenated,
%                           and information words separately
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 11th, 2017
%
%%

if (read_from_file == 1)
    rds_bitstream = load('rds_data.mat');
else
    % initialize bitstream
    rds_bitstream = zeros(1,rds_stream_length);

    msg_length = no_msgs*16*4;
    % generate random data
    rand_data = rand(1,msg_length);

    % initialize messages
    binary_data = zeros(1,msg_length);
    information = zeros(1,msg_length);

    % mapping random data to binary rds stream
    for i=1:msg_length
        if (rand_data(1,i)>0.5)
            binary_data(1,i) = 1;
        end
    end

    % generate data for each block separately
    blk1_info =  binary_data(1:msg_length/4);
    blk2_info = binary_data(msg_length/4 + 1:msg_length/2);
    blk3_info = binary_data(msg_length/2 + 1:3*msg_length/4);
    blk4_info = binary_data(3*msg_length/4 + 1:msg_length);

    % calculate CRC for each block and generate the blocks
    offsetA = [0 0 1 1 1 1 1 1 0 0];
    H1 = comm.CRCGenerator([10 8 7 5 4 3 0],'FinalXOR',offsetA,'ChecksumsPerFrame',↙
no_msgs);
    blk1_msg = H1(blk1_info.');
    blk1_msg = blk1_msg.';

    offsetB = [0 1 1 0 0 1 1 0 0 0];
    H2 = comm.CRCGenerator([10 8 7 5 4 3 0],'FinalXOR',offsetB,'ChecksumsPerFrame',↙
no_msgs);
    blk2_msg = H2(blk2_info.');
    blk2_msg = blk2_msg.';
```

```
    if (version == 'A')
        offsetC = [0 1 0 1 1 0 1 0 0 0];
        H3 = comm.CRCGenerator([10 8 7 5 4 3 0],'FinalXOR',↙
offsetC,'ChecksumsPerFrame',no_msgs);
        blk3_msg = H3(blk3_info.');
        blk3_msg = blk3_msg.';
    else
        offsetC2 = [1 1 0 1 0 1 0 0 0 0];
        H3 = comm.CRCGenerator([10 8 7 5 4 3 0],'FinalXOR',↙
offsetC2,'ChecksumsPerFrame',no_msgs);
        blk3_msg = H3(blk3_info.');
        blk3_msg = blk3_msg.';
    end

    offsetD = [0 1 1 0 1 1 0 1 0 0];
    H4 = comm.CRCGenerator([10 8 7 5 4 3 0],'FinalXOR',offsetD,'ChecksumsPerFrame',↙
no_msgs);
    blk4_msg = H4(blk4_info.');
    blk4_msg = blk4_msg.';

    % concatenate the information and error blocks
    for j=1:no_msgs
        rds_bitstream((j-1)*104+1:j*104) = cat(2,blk1_msg((j-1)*26+1:j*26),blk2_msg↙
((j-1)*26+1:j*26),blk3_msg((j-1)*26+1:j*26),blk4_msg((j-1)*26+1:j*26));
        information((j-1)*64+1:j*64) = cat(2,blk1_info((j-1)*16+1:j*16),blk2_info((j-↙
1)*16+1:j*16),blk3_info((j-1)*16+1:j*16),blk4_info((j-1)*16+1:j*16));
    end

end
end
```

```
function [ code ] = diff_enco( bitstream )
%% DIFF_ENCO Performs differential encoding of input bit-stream
%   INPUT arguments:        stream of bits to be encoded
%
%   OUTPUT arguments:       differentially encoded stream
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 4th, 2017
%
%%

% find the length of the stream
stream_length = length(bitstream);
% initialize the encoded stream
code = zeros(1,stream_length);
% initialize with first element
code(1) = bitstream(1);

% perform differential encoding
for k=2:stream_length
    code(k) = xor(code(k-1),bitstream(k));
end


end
```

```
function [ biphase ] = biphase_generator( impulses)
%BIPHASE_GENERATOR generates biphase symbols of RDS bitstream
%   INPUT arguments:        stream of differntially encoded RDS data
%
%   OUTPUT arguments:       biphase symbols
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 6th, 2017
%


% interpolate impulses so that values at multiples of td/2 can be accessed
% (for shifting)
impulses_interpl = upsample(impulses,200);

% initialize the shifted version of the signal
impulses_shift = zeros(1,length(impulses_interpl));

% delay index is the number of samples corresponding to td/2
delay_index = 100;

% delay the stream by td/2
impulses_shift(delay_index+1:end) = impulses_interpl(1:end-delay_index);

% biphase
biphase = (impulses_interpl - impulses_shift);



end
```

```
function [ rds ] = pulse_shape( biphase,L,Fs )
%PULSE_SHAPE Applies pulse shaping to biphase symbols (impulse pairs)
%   INPUT arguments:        biphase impulse pairs, frequency of sampling
%
%   OUTPUT arguments:       rds (shaped)
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 6th, 2017
%

% time step
Ts = 1/Fs;
% sampling time array
t = (-L/2:L/2)*Ts;


% frequency
f = Fs*(0:(L/2))/L;

% option 1 - hardcoding the filter by deriving the impulse response
% filter = 1/(2*pi) * (sin(4750*pi*(t+td/8))./(t+td/8) + sin(4750*pi*(t-td/8))./(t-td/8)) ;

% option 2 - designing the puslse shaping filter
% The rolloff is 100% (pure cosine)
rolloff = 1;

% Here each bit is represented by an impulse pair. Hence number of symbols
% must be twice the number of bits
no_of_symbols = 4*max(t)*19000*3/48;

% sps must be 100 because each bit has 200 samples and hence each symbol
% will have half of that
samples_per_symbol = 100;

filter = rcosdesign(rolloff,no_of_symbols,samples_per_symbol);

% impulse response of pulse shaping filter
figure;
zero_ind = find(t==0);
plot(t(zero_ind-1e3:zero_ind+1e3),filter(zero_ind-1e3:zero_ind+1e3));
title('Impulse response of pulse shaping filter (truncated time interval)');
xlabel('Time(in seconds)');
ylabel('Amplitude');
% plot frequency response
[H,f1] = freqz(filter,1,2048,Fs);

figure;
subplot(2,1,1);
plot(f1,abs(H));
title('Frequency response of pulse shaping filter');
xlabel('Frequency(in Hz)');
ylabel('Amplitude');
subplot(2,1,2);
plot(f1,unwrap(angle(H)));
title('Phase response of pulse shaping filter');
```

```
xlabel('Frequency(in Hz)');
ylabel('Phase');

% perform convolution
rds = conv(biphase,filter,'same');

% reduce the amplitude
gain = max(abs(filter(:)))/50;
rds = rds/gain;

figure;
subplot(2,1,1);
plot(biphase(1:2000));
title('Biphase');

subplot(2,1,2);
plot(rds(1:2000));
title('RDS');
%% --------------------- Find and display fft of RDS ------------- %%
fm_rds_Y = fft(rds);
% double sided spectrum
P2 = abs(fm_rds_Y/L);
% single sided spectrum
FM_RDS_FT = P2(1:L/2+1);

% recover exact amplitudes
FM_RDS_FT(2:end-1) = 2*FM_RDS_FT(2:end-1);

figure;
plot(f,FM_RDS_FT)
title('Single-Sided Amplitude Spectrum of RDS at baseband');
xlabel('f (Hz)');
ylabel('|FT(f)|');
end
```

```
function [ fm_rds_signal ] = fmrds_modulate( left, right, rds, pilot_freq,L,Fs)
%% TRANSMIT Function to modulate FM RDS broadcast (before FM stage)
%   INPUT arguments:        Left and right stereo audio, RDS bitstream,
%                           carrier frequency, length of time base, sampling frequency
%   OUTPUT arguments:       Frequency modulated broadcast
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 4th, 2017

% sum of left and right signals (main channel baseband)
stereo_sum = left+right;

% difference of left and right signals (centred at twice the carrier
% frequency)
stereo_diff = left-right;
% time step
Ts =1/Fs;
% sampling time array
t = (-L/2:L/2-1)*Ts;

figure;
subplot(2,1,1);
plot(t(1:5000),stereo_diff(1:5000))
title('Stereo diff');
xlabel('t (secs)');
ylabel('L-R(t)');

subplot(2,1,2);
plot(t(1:5000),stereo_sum(1:5000))
title('Stereo sum');
xlabel('t (secs)');
ylabel('L+R(t)')

% pilot frequency in radians (19kHz in radians)
carrier_omega = pilot_freq*2*pi;

% multiply the rds signal with the 57kHz carrier
rds_band = rds.*cos(3*carrier_omega*t);

figure;
zero_ind = find(t==0);
plot(t(zero_ind-3e2:zero_ind+2e2),rds_band(zero_ind-3e2:zero_ind+2e2));
title('RDS on 57 kHz carrier (time domain)');
xlabel('Time(s)');
ylabel('RDS signal amplitude');

% sub channel difference signal centred at 38kHz
sub_channel = stereo_diff.*cos(2*carrier_omega*t);

% pilot tone for synchronous detection
pilot_tone = cos(carrier_omega*t);
% sum the signals to get the broadcast
fm_rds_signal = stereo_sum + sub_channel + pilot_tone + rds_band;
end
```

```
function [left_rec, right_rec, pilot, rds] = fmrds_demod(fm_rds_signal,Fs,L)
% RECEIVE receive and demodulate fm rds broadcast
%   INPUT arguments:        FM RDS broadcast, sampling frequency, length of
%                           time base
%   OUTPUT arguments:       left stereo signal, right stereo signal, pilot
%                           tone, differential encoded RDS data
%   Functions called:       bipolar_switch
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 4th, 2017

% time base
t = (-L/2:L/2-1)/Fs;

% frequency
f = Fs*(0:(L/2))/L;

%% Extract pilot tone
% designing elliptical bandpass filter
filter_order = 2;
passband_ripple = 0.1;
stopband_atten = 80;

pilot_passband = [18e3 20e3]/(Fs/2);

ftype = 'bandpass';

% Zero-Pole-Gain design to avoid numerical instabilities
[zpilot,ppilot,kpilot] = ellip(filter_order,passband_ripple,stopband_atten,↙
pilot_passband,ftype);
% convert to second order sections for implementation
sos = zp2sos(zpilot,ppilot,kpilot);

pilot = sosfilt(sos, fm_rds_signal);

pilot_fft = fft(pilot);
% double sided spectrum
P2 = abs(pilot_fft/L);
% single sided spectrum
PILOT_FT = P2(1:L/2+1);

% recover exact amplitudes
PILOT_FT(2:end-1) = 2*PILOT_FT(2:end-1);

figure;
plot(f,PILOT_FT)
title('Single-Sided Amplitude Spectrum Pilot tone');
xlabel('f (Hz)');
ylabel('|FT(f)|');

% normalize tone
% pilot = pilot;

% generate carrier for 38 kHz stereo difference signal using trig identity
diff_carrier = 2*pilot.^2-1;
```

```
% generate carrier for 57 kHz RDS signal using trig identity
rds_carrier = 4*pilot.^3 - 3*pilot;


[~,pilot_ind] = max(PILOT_FT(:));
pilot_freq = max(f(pilot_ind));
% extract clock frequency for RDS from pilot tone
clk_freq = pilot_freq/16;

%% Low pass filtering to retrieve the stereo sum signal (L+R) (within 15kHz)
% Design low pass filter
% cutoff frequency normalized wrt Nyquist frequency
cutoff = 10e3/(Fs/2);

% order of filter
order = 64;

% impulse response of filter
lowpassh = fir1(order, cutoff);

figure;
% magnitude and phase response of the filter
freqz(lowpassh);
title('Frequency response of low pass filter');

% extract stereo sum signal
stereo_sum = conv(fm_rds_signal,lowpassh,'same');

%% Find and display FFT of the stereo sum signal
% fft of signal
stereo_sum_fft = fft(stereo_sum);
% double sided spectrum
P2 = abs(stereo_sum_fft/L);
% single sided spectrum
stereo_sum_FT = P2(1:L/2+1);
% recover exact amplitudes
stereo_sum_FT(2:end-1) = 2*stereo_sum_FT(2:end-1);

figure;
subplot(2,1,1);
plot(f,stereo_sum_FT)
title('Single-sided amplitude spectrum of (L+R) signal at receiver end');
xlabel('f (Hz)');
ylabel('|(L+R)FT(f)|');
%% Bandpass filtering to retrieve the stereo difference (L-R) signal centred at 38kHz
%
% designing elliptical bandpass filter
% filter order
filter_order = 4;
passband = [23e3 53e3]/(Fs/2);
ftype = 'bandpass';

% Zero-Pole-Gain design
[z,p,k] = butter(filter_order,passband,ftype);
sosdiff = zp2sos(z,p,k);
```

```
stereo_diff_band = sosfilt(sosdiff, fm_rds_signal);

% multiply signal with tonal to shift the stereo difference to baseband
base_stereo_diff = stereo_diff_band.*diff_carrier;

% Use the previous low pass filter
% extract stereo difference signal
stereo_diff = conv(base_stereo_diff,lowpassh,'same');

%% Find and display FFT of the stereo difference (L-R) signal
% fft of signal
stereo_diff_fft = fft(stereo_diff);
% double sided spectrum
P2 = abs(stereo_diff_fft/L);
% single sided spectrum
stereo_diff_FT = P2(1:L/2+1);
% recover exact amplitudes
stereo_diff_FT(2:end-1) = 2*stereo_diff_FT(2:end-1);

subplot(2,1,2);
plot(f,stereo_diff_FT)
title('Single-sided amplitude spectrum of (L-R) signal at receiver end');
xlabel('f (Hz)');
ylabel('|(L-R)FT(f)|');

%% Compute the left stereo signal and the right stereo signal

left_rec = (stereo_sum + stereo_diff)/2;
right_rec = (stereo_sum - stereo_diff)/2;

%% Find and display FFT of the LEFT signal
% fft of signal
left_fft = fft(left_rec);
% double sided spectrum
P2 = abs(left_fft/L);
% single sided spectrum
left_FT = P2(1:L/2+1);
% recover exact amplitudes
left_FT(2:end-1) = 2*left_FT(2:end-1);

figure;
subplot(2,1,1);
plot(f,left_FT)
title('Single-sided amplitude spectrum of left signal at receiver end');
xlabel('f (Hz)');
ylabel('|LEFT FT(f)|');

%% Find and display FFT of the RIGHT signal
% fft of signal
right_fft = fft(right_rec);
% double sided spectrum
P2 = abs(right_fft/L);
% single sided spectrum
right_FT = P2(1:L/2+1);
```

```
% recover exact amplitudes
right_FT(2:end-1) = 2*right_FT(2:end-1);

subplot(2,1,2);
plot(f,right_FT)
title('Single-sided amplitude spectrum of right signal at receiver end');
xlabel('f (Hz)');
ylabel('|RIGHT FT(f)|');

%% Display the time-domain received signals (both left and right)
zero_ind = find(t==0);

figure;
subplot(2,1,1);
plot(t(zero_ind-1e3:zero_ind+1e3),left_rec(zero_ind-1e3:zero_ind+1e3))
title('Left signal (time domain) at receiver end');
xlabel('t (secs)');
ylabel('left(t)');

subplot(2,1,2);
plot(t(zero_ind-1e3:zero_ind+1e3),right_rec(zero_ind-1e3:zero_ind+1e3))
title('Right signal (time domain) at receiver end');
xlabel('t (secs)');
ylabel('right(t)');

%% Extract RDS encoded bit-stream

% designing elliptical bandpass filter
filter_order = 2;
rds_passband = [54.6e3 59.4e3]/(Fs/2);
ftype = 'bandpass';
passband_ripple = 0.01;
stopband_atten = 100;

% Zero-Pole-Gain design
[z,p,k] = ellip(filter_order,passband_ripple,stopband_atten, rds_passband,ftype);
sos = zp2sos(z,p,k);

rds_band = sosfilt(sos, fm_rds_signal);

% multiply signal with tonal to shift the rds signal to baseband
base_rds = rds_band.*rds_carrier;

% Use a low pass filter
% Design low pass filter
% cutoff frequency normalized wrt Nyquist frequency
cutoff = 2.375e3/(Fs/2);

% order of filter
order = 64;

% impulse response of filter
lowpass_rds = fir1(order, cutoff);

% extract biphase encoded rds signal
```

```
biphase_code_rds = conv(base_rds,lowpass_rds,'same');

samples_per_symbol = Fs/clk_freq;

% sample the signal and it's inverted version with a bipolar switch
uni_rds = bipolar_switch(biphase_code_rds,samples_per_symbol);

% integrate and dump
rds_analog = intdump(uni_rds,samples_per_symbol);


figure;
stairs(rds_analog(end/2-4:end/2+5));
title('Output of integration and dumping');
xlabel('Bit index');

% slicer
rds = rds_analog>0;

%% Find and display FFT of the RDS baseband biphase encoded signal (o/p of filter)
% fft of signal
rds_fft = fft(biphase_code_rds);
% double sided spectrum
P2 = abs(rds_fft/L);
% single sided spectrum
rds_FT = P2(1:L/2+1);
% recover exact amplitudes
rds_FT(2:end-1) = 2*rds_FT(2:end-1);

figure;
plot(f,rds_FT)
title('Single-sided amplitude spectrum of RDS signal at receiver end');
xlabel('f (Hz)');
ylabel('|RDS FT(f)|');
end
```

```
function [ bipolar ] = bipolar_switch( signal, sps )
%BIPOLAR_SWITCH
%   samples signal for half a time period and it's inverted version for the
%   next half a time period
%   INPUT arguments:            Signal to be sampled, samples per
%                               symbol(must be even)
%
%   OUTPUT arguments:           Bipolar switching's sampled output
%
%   Author:                     Joshua Peter Ebenezer
%   Date of creation:           7th December, 2017

% initialize the output
bipolar = signal;

% number of symbols
symb_count = length(signal)/sps;

% flip sample whenever half the time period has elapsed
for i=1:symb_count
    bipolar((i-1)*sps+sps/4:(i-1)*sps+3*sps/4) = -signal((i-1)*sps+sps/4:(i-1)↙
*sps+3*sps/4);
end

%% plot the signal and the bipolar sampled signal

% number of sampling instances
L = 9.5e5;

% frequency of sampling
Fs = 2.375e5;
% time step
Ts = 1/Fs;
% sampling time array
t = (-L/2:L/2-1)*Ts;

zero_ind = find(t==0);

figure;
subplot(2,1,1);
plot(t(zero_ind-1e3:zero_ind+1e3),signal(zero_ind-1e3:zero_ind+1e3));
title('Biphase encoded signal before bipolar sampling');

subplot(2,1,2);
plot(t(zero_ind-1e3:zero_ind+1e3),bipolar(zero_ind-1e3:zero_ind+1e3));
title('Bipolar sampled signal');

end
```

```
function [ start_location ] = synchronize( rx_bitstream )
%SYNCHRONIZE Performs initial acquisition of synchronization of receieved
%               data by calculation of the syndrome
%   INPUT arguments:        received bitstream
%
%   OUTPUT arguments:       location of start of synchronized bitstream

start_location = 1;
% find length of received stream
rx_stream_length = length(rx_bitstream);
offset = zeros(1, rx_stream_length-26);
% move bit by bit through the stream
for i=1:rx_stream_length-26
    % check the block (of 26 bits) starting from the current index i
    current_block = rx_bitstream(i:i+25);
    % generate offset index for the block (read syndrome.m for
    % correspondence between offset, offset index, and syndrome)
    [offset(i),~] = syndrome(current_block);

    % if the offset is valid
    if (offset(i)<10)
        % matrix to check if adjacent blocks are valid
        check_next = zeros(1,4);
        % check for validity of blocks nx26 bits apart, where n=1,2,3
        for n = 1:3
            % find the offset for the next block
            if (i+(n+1)*26-1 < length(rx_bitstream))
                [check_next(n),~] = syndrome(rx_bitstream(i+n*26:i+(n+1)*26-1));
                % find out if the offset is correct and in the valid order
                if (check_next(n) < 10 && check_next(n) == mod(offset(i)+n,4))
                    % if yes then keep the start location at the block whose
                    % expected offset is A

                    % try to find the nearest start location of the group↙
corresponding
                    % to the matched block
                    if (i - (offset(i)-1)*26 > 0)
                        start_location = i - (offset(i)-1)*26;
                        % if that is not possible keep it in the next group's
                        % expected starting location
                    else
                        start_location = i + (5-offset(i))*26;
                    end
                    return;
                end
            else
                start_location = 1;
                return;
            end
        end
    end
end
```

```
function [ offset, version ] = syndrome( block )
%SYNDROME Finds offset corresponding to the input block
%   INPUT argument:     block of length 26 bits
%
%   OUTPUT argument:    offset word index corresponding to block
% version default - null
version = [];
% create an identity matrix
id_ten = eye(10);
% generate the lower part of the parity check matrix
lower = [1 0 1 1 0 1 1 1 0 0;
    0 1 0 1 1 0 1 1 1 0;
    0 0 1 0 1 1 0 1 1 1;
    1 0 1 0 0 0 0 1 1 1;
    1 1 1 0 0 1 1 1 1 1;
    1 1 0 0 0 1 0 0 1 1;
    1 1 0 1 0 1 0 1 0 1;
    1 1 0 1 1 1 0 1 1 0;
    0 1 1 0 1 1 1 0 1 1;
    1 0 0 0 0 0 0 0 0 1;
    1 1 1 1 0 1 1 1 0 0;
    0 1 1 1 1 0 1 1 1 0;
    0 0 1 1 1 1 0 1 1 1;
    1 0 1 0 1 0 0 1 1 1;
    1 1 1 0 0 0 1 1 1 1;
    1 1 0 0 0 1 1 0 1 1];
% generate the parity check matrix by concatenating the two
H = cat(1, id_ten,lower);
% generate syndrome
syndrome_calc = block * H;
% convert to modulo-2 form
syndrome = mod(syndrome_calc,2);
    % check if syndrome matches any of the offsets
    if (isequal(syndrome,[1 1 1 1 0 1 1 0 0 0]))
        offset = 1; % offset A [0 0 1 1 1 1 1 1 0 0]
    elseif (isequal(syndrome,[1 1 1 1 0 1 0 1 0 0]))
        offset = 2; % offset B [0 1 1 0 0 1 1 0 0 0]

    elseif (isequal(syndrome,[1 0 0 1 0 1 1 1 0 0]))
        offset = 3; % offset C [0 1 0 1 1 0 1 0 0 0];
        version = 'A';
    elseif (isequal(syndrome,[1 1 1 1 0 0 1 1 0 0]))
        offset = 3; % offset C'[1 1 0 1 0 1 0 0 0 0]
        version = 'B';

    elseif (isequal(syndrome,[1 0 0 1 0 1 1 0 0 0]))
        offset = 4; % offset D [0 1 1 0 1 1 0 1 0 0]
    else
        offset= 10;
    end
end
```

```
function [ bitstream ] = diff_deco( code )
%DIFF_DECO Performs differential decoding of input bit-stream
%   INPUT arguments:        stream of bits to be decoded
%
%   OUTPUT arguments:       decoded stream
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 4th, 2017
%

%%

% find the length of the stream
stream_length = length(code);
% initialize the decoded stream
bitstream = zeros(1,stream_length);

% first bit needs to be the same
bitstream(1) = code(1);

% perform differential decoding
for k=2:stream_length
    bitstream(k) = xor(code(k),code(k-1));
end


end
```

```
function [ rx_information,info_error ] = data_decode( rx_bitstream, start_location )
%DATA_DECODE Performs CRC checking of rx_bistream
%   INPUT arguments:        received and demodulated bitstream
%
%   OUTPUT arguments:       information words and errors
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 11th, 2017
%

synchronized = rx_bitstream(start_location:end);

% number of messages
no_msgs = floor(length(synchronized)/104);

rx_information = zeros(1,no_msgs*4*16);

info_error = zeros(1,no_msgs*4);

blk1_msg = zeros(1,no_msgs*26);
blk2_msg = zeros(1,no_msgs*26);
blk3_msg = zeros(1,no_msgs*26);
blk4_msg = zeros(1,no_msgs*26);

for j=1:no_msgs
    grp = synchronized((j-1)*104+1:j*104);

    blk1_msg((j-1)*26+1:j*26) = grp(1:26);
    blk2_msg((j-1)*26+1:j*26) = grp(27:52);
    blk3_msg((j-1)*26+1:j*26) = grp(53:78);
    blk4_msg((j-1)*26+1:j*26) = grp(79:104);
end

err3 = zeros(1,no_msgs);

% calculate CRC for each block and generate the blocks
offsetA = [0 0 1 1 1 1 1 1 0 0];
H1 = comm.CRCDetector([10 8 7 5 4 3 0],'FinalXOR',offsetA,'ChecksumsPerFrame',↙
no_msgs);
[blk1_info, err1] = step(H1,blk1_msg.');
blk1_info = blk1_info.';

offsetB = [0 1 1 0 0 1 1 0 0 0];
H2 = comm.CRCDetector([10 8 7 5 4 3 0],'FinalXOR',offsetB,'ChecksumsPerFrame',↙
no_msgs);
[blk2_info, err2] = step(H2,blk2_msg.');
blk2_info = blk2_info.';

for j=1:no_msgs
    [~, version] = syndrome(blk3_msg((j-1)*26+1:j*26));

    if ( version == 'A')
        offsetC = [0 1 0 1 1 0 1 0 0 0];
        H3 = comm.CRCDetector([10 8 7 5 4 3 0],'FinalXOR',offsetC);
        [blk3_info((j-1)*16+1:j*16), err3(j)] = step(H3,blk3_msg((j-1)*26+1:j*26).');
```

```
    else
        offsetC2 = [1 1 0 1 0 1 0 0 0 0];
        H3 = comm.CRCDetector([10 8 7 5 4 3 0],'FinalXOR',offsetC2);
        [blk3_info((j-1)*16+1:j*16), err3(j)] = step(H3,blk3_msg((j-1)*26+1:j*26).');
    end
end


offsetD = [0 1 1 0 1 1 0 1 0 0];
H4 = comm.CRCDetector([10 8 7 5 4 3 0],'FinalXOR',offsetD,'ChecksumsPerFrame',↙
no_msgs);
[blk4_info, err4] = step(H4,blk4_msg.');
blk4_info = blk4_info.';


for j=1:no_msgs
    rx_information((j-1)*64+1:j*64) = cat(2,blk1_info((j-1)*16+1:j*16),blk2_info((j-1)↙
*16+1:j*16),blk3_info((j-1)*16+1:j*16),blk4_info((j-1)*16+1:j*16));
    info_error((j-1)*4+1:j*4) = cat(2,err1(j),err2(j),err3(j),err4(j));
end

end
```

```
function [ slip ] = bitslip( block, pi_code )
%BITSLIP Alerts program to slip in clocking of data. Returns slip to be
%        compensated
%   INPUT arguments:        block (of data), pi_code
%
%   OUTPUT arguments:       slip
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 12th, 2017


[cross_correlation,lags] = xcorr(block,pi_code);

[~, lag_index] = max(abs(cross_correlation));

slip = lags(lag_index);
end
```

```
function [  ] = message_display( rx_information )
%MESSAGE_EXTRACTION Function takes in the received information and displays
%                   the message in a readable format
%   INPUT arguments:        received RDS information
%
%   OUTPUT arguments:       -
%
%   Author:                 Joshua Peter Ebenezer
%   Date of creation:       December 12, 2017

% number of messages
no_msgs = floor(length(rx_information)/64);

for i=1:no_msgs
    % taking out one group at a time
    group = rx_information((i-1)*64+1:i*64);

    % splitting up the blocks' information
    info1 = group(1:16);
    info2 = group(17:32);
    info3 = group(33:48);
    info4 = group(49:64);

    % PI code
    pi_hex = dec2hex(bin2dec(int2str(info1)));
    fprintf('\nPI code is given by %s', pi_hex);



    % group type
    group_type = bin2dec(int2str(info2(1:4)));
    fprintf('\nGroup type is %d',group_type);

    % version (0-A 1-B)
    if (info2(5)==0)
        version = 'A';  % PI code is only inserted in block 1
    else
        version ='B';   % PI code is inserted in blocks 1 and 3
    end
    fprintf('\nVersion is %s',version);

    % traffic programme code
    traffic_code = info2(6);
    if (traffic_code == 0)
        fprintf('\nTraffic programme information is not present');
    else
        fprintf('\nTraffic programme information is present');
    end

    % programme type code
    pty = info2(7:11);
    fprintf('\nProgramme type code is');
    disp(pty);

    % traffic announcement
```

```
    ta = info2(12);
    if (ta == 0)
        traff = 'OFF';
    else
        traff = 'ON';
    end
    fprintf('\nTraffic announcements are %s', traff);

    % music or speech
    ms = info2(13);
     if (ms == 0)
        mors = 'Speech';
    else
        mors = 'Music';
    end
    fprintf('\nMusic/Speech: %s\n', mors);

end
end
```

```
function [ carrier ] = pll( pilot, harmonic )
%PLL Takes pilot tone (19 kHz) and harmonic of carrier and returns phase-locked
%carrier
%   INPUT arguments:    Pilot signal and harmonic (1st,2nd,3rd..)
%
%   OUTPUT arguments:   Carrier signal (phase locked to harmonic of pilot)
%
%   Author:             Joshua Peter Ebenezer
%   Date:               December 26th, 2017


% number of sampling instances
L = 9.5e5;
% frequency of sampling
Fs = 2.375e5;
% time step
Ts = 1/Fs;

% Ideal carrier frequency
fVCO = harmonic * 19000; %free running oscillating freq of VCO
KVCO = fVCO*0.1;    % gain of VCO (voltage to freq transfer coeff.) [Hz/V]
G1 = 0.05;        % Prop gain term of PI controller
G2 = 6e-3;      % integral gain term of PI controller

% PLL LPF (for integration of error, removal of noise from error)
fc = 1e3;  % frequency of LPF
fil_coeff_num = 500; %number of filter coeffs of LPF

% BPF after squaring or cubing pilot signal
harm_fil_order = 400;    %order of filter
harm_fil_freq_range = [fVCO-1e3,fVCO+1e3]; %pass band
% bpf initialization
bpf = fir1(harm_fil_order,harm_fil_freq_range/(Fs/2));
%FIR filter coefficients
harm_fil_buffer = zeros(1,harm_fil_order+1);
harmonic_input_filtered = zeros(1,L);

% PLL initialization
pll_fil = fir1(fil_coeff_num,fc/(Fs/2)); % design FIR filter coeff
VCO = zeros(1,L); % VCO signal array
phi = zeros(1,L); % VCO angle array
error = zeros(1,L); % error array
int_error = zeros(1,L); % error array
filter_buffer = zeros(1,fil_coeff_num+1); % initialize PLL LPF buffer
error_mult = zeros(1,L); % initialize error signal
reference = zeros(1,L);
PI_error = zeros(1,L);
carrier = zeros(1,L);
input_harmonic = zeros(1,L);

%% Begin sampled time simulation %%
for n=1:L
    t = (-L/2 + (n-1))*Ts;

    % Find harmonic of received signal
```

```
    input_harmonic(n) = pilot(n)^harmonic;

    % send through BPF to extract harmonic term alone
    harm_fil_buffer = [harm_fil_buffer(2:harm_fil_order+1),input_harmonic(n)];
    harmonic_input_filtered(n+1) = fliplr(bpf)*harm_fil_buffer.'; %BPF operation
    reference(n) = harmonic_input_filtered(n); % signal that goes to PLL

    % PLL calculations
    error_mult(n) = harmonic_input_filtered(n)*VCO(n); % multiply VCO x signal input↙
to get raw error signal

    % LPF raw error signal
    filter_buffer = [filter_buffer(2:fil_coeff_num+1),error_mult(n)]; % update PLL LPF↙
buffer
    error(n+1) = 4*fliplr(pll_fil)*filter_buffer.';

    % process filtered error signal through PI controller
    int_error(n+1) = int_error(n)+G2*error(n)*Ts;
    PI_error(n+1) = G1*error(n+1)+int_error(n+1);

    % update VCO
    phi(n+1) = phi(n)+2*pi*PI_error(n+1)*KVCO*Ts; % update phase of VCO
    VCO(n+1) = -sin(2*pi*fVCO*t+phi(n+1));   % compute VCO signal

    % VCO_shifted(n+1) = sin(2*pi*fVCO*t+phi(n+1)-pi/2); % shift output of PLL by 90↙
degrees
    carrier(n) = cos(2*pi*fVCO*t+phi(n));

end
end
```