COE3DY4 Project
Real-time FM Receiver: SDR for Mono/Stereo Audio and RDS

Objective

The main objective of the project is to navigate a complex, industry-level specification and address the challenges involved in implementing it within a real-time computing system operating in a form factor-constrained environment.

Preparation

- Revise the material from labs 1 to 4

Project Overview

The 3DY4 project leverages front-end radio-frequency (RF) hardware, such as RF dongles [1] based on the Realtek RTL2832U chipset [2], and single-board computers, such as the Raspberry Pi 4 [3]. The objective is the real-time implementation of a software-defined radio (SDR) system for the reception of frequency-modulated (FM) mono/stereo audio [4], as well as the reception of digital data transmitted via FM broadcast using the radio data system (RDS) protocol [5].[1] Due to the affordability of RF hardware, numerous open-source SDR projects have emerged over the past decade, including GNU Radio [6] and Gqrx [7]. However, the goal of the 3DY4 project is not to duplicate these open-source initiatives. Rather, this real-time SDR project serves to consolidate the knowledge acquired so far and to provide experience in navigating from first principles to understanding the complex interrelationships between seemingly disparate, yet practically related, topics in electrical and computer engineering.

Each FM channel occupies 200 KHz of the FM band and is symmetric around its center frequency, which can range from 88.1 MHz to 107.9 MHz in Canada. Note that not all FM channels are used in the same geographic region; therefore, in some areas, it is common for two FM stations to broadcast 400 KHz apart or at an even higher multiple of 200 KHz. When analyzing the positive frequencies of a demodulated FM channel (0 to 100 KHz), three sub-channels within each FM channel are of interest to this project. The mono sub-channel spans 0 to 15 KHz, the stereo sub-channel spans 23 to 53 KHz, and the RDS sub-channel spans 54 to 60 KHz. Above 60 KHz, there *might* be other sub-channels utilizing SCA sub-carriers, where SCA stands for Subsidiary Communications Authorization. These sub-carriers allow FM stations to broadcast additional services but are not standardized. The focus of the 3DY4 project will be on the mono audio, stereo audio, and RBDS, as clearly labeled in Figure 1.

[1] Since RDS was initially developed in Europe, the official name in North America is Radio Broadcast Data System (RBDS). Except for some digital codes specific to geographic locations, the two standards are virtually identical; therefore, the terms RDS and RBDS will be used interchangeably.

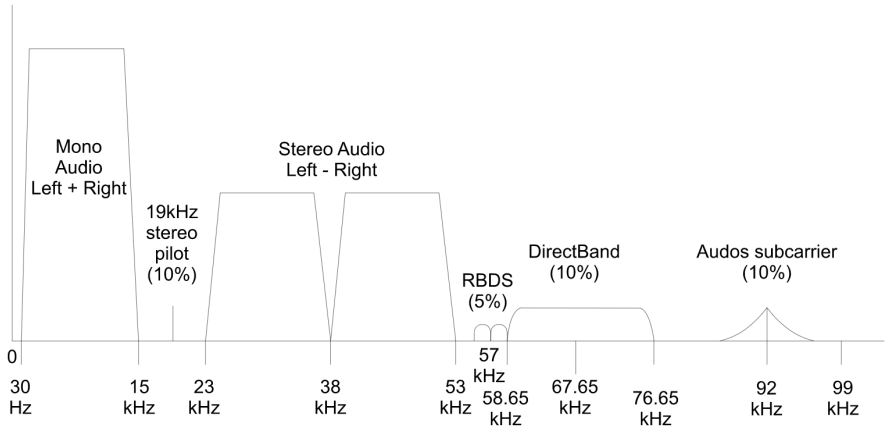


Figure 1: Spectrum of an FM channel [4].

Between the mono and stereo sub-channels, a 19 KHz stereo pilot tone is used to synchronize the stereo sub-carrier to the second harmonic of this tone, i.e., 38 KHz. This mechanism was essential for broadcasting stereo audio when receivers were built with analog technologies. While the RDS sub-channel is typically centered around the third harmonic of the pilot tone, i.e., 57 KHz, some FM stations do not necessarily lock the RDS sub-carrier to the pilot tone. As a result, carrier recovery and downconversion for RDS will use a different approach than that used for stereo audio. It is also worth noting that the RDS approach resembles the methods followed by many modern digital communication standards.

While FM stations provide the data that feeds the SDR system implemented in this project, all essential FM-related information relevant to the project is provided through the previous labs, forthcoming class discussions, and this project document. For more detailed information on FM technology, you are referred to academic textbooks [8, 9] or industry white papers, such as [10, 11]. As a final note, since some terms can be inferred from context, we will refer to the sub-channels of an FM channel simply as channels, i.e., the mono channel (0 to 15 KHz), the stereo channel (23 to 53 KHz), and the RDS channel (54 to 60 KHz).

The big picture of the 3DY4 project is captured in Figure 2, and it is summarized below:

- The RF hardware is responsible for acquiring the RF signal through an antenna and converting it into the digital domain by producing an 8-bit sample for the in-phase (I) component and another 8-bit sample for the quadrature (Q) component of the RF signal. This process involves analog downconversion through mixing and filtering using a local oscillator. Since the software interface to the RF hardware has been developed by a third party [2], the key point to consider is that the I/Q data is transferred to the host in an **interleaved** format as I/Q 8-bit sample pairs—i.e., an 8-bit I sample followed by the corresponding 8-bit Q sample, then the next I/Q pair, and so on.
- The RF front-end block of the SDR system extracts the FM channel through low-pass filtering, followed by decimation to an intermediate frequency (IF) and FM demodulation. Conceptually, it is the same as the RF front-end used in the previous lab; however, for this project, there will be **four** modes of operation.

  It is critical to note that, for the remainder of this document, all figures and examples are described as if there are only two modes of operation. While mode 0 is the same for all project groups, modes 1, 2, and 3 will be custom for each

group. The unique settings for these modes will be uploaded to your group's GitHub repository after it has been created. Therefore, mode 1 in this document is "fictive" and does not apply to any project for any group. The descriptions of mode 1 in this document, including its sample rates (e.g., 2.5 Msamples/sec for the front-end and 250 Ksamples/sec for the intermediate stage), are only meant to highlight the differences between the custom modes 1, 2, and 3 and mode 0, as well as to explain the conceptual challenges involved in resampling.

By default, your SDR software should operate in mode 0, with only the mono path being exercised. If command-line arguments are provided, the first argument specifies the mode (0, 1, 2, or 3), and the second specifies the most advanced path to be exercised (`m` for mono, `s` for stereo, and `r` for RDS). Note that if the `r` path is exercised, it implies that both the mono and stereo paths will also be exercised. For further details, refer to your group's GitHub project repository for the custom sample rate settings for each mode.

- The mono block extracts the mono audio channel (0 to 15 KHz) and reduces the sample rate to 48 KSamples/sec. The input consists of FM demodulated data, which should be represented in real format (either 32-bit float or 64-bit double) with an IF sample rate of either 240 KSamples/sec or 250 KSamples/sec, depending on the mode of operation. The output should be in 16-bit signed integer format, suitable for transfer to an audio player. Note that if only mono audio is implemented, the 16-bit values represent the sum of the left and right audio channels. Since the intermediate data is in floating-point format, the SDR software developer must select an appropriate internal scale factor for the output audio data to fully utilize the dynamic range provided by the 16-bit signed integer format.
- The stereo block extracts the 19 KHz pilot tone through band-pass filtering to enable the downconversion of the stereo channel. The downconverted stereo channel is then filtered and combined with the mono audio data to produce the left and right audio channels. This is achieved by combining the data from the mono and stereo channels, where the mono channel contains the sum of the left and right audio channels, and the stereo channel contains the difference between the left and right audio channels. The input and output formats are the same as those for the mono path.
- The RDS block first recovers the subcarrier from the RDS channel, downconverts it using a digital communication approach, and resamples it before performing clock and data recovery. Once the bitstream is generated, frame synchronization is performed in the data link layer, after which the extracted information words are passed to the RDS application layer. The input to this block is the same as that for the mono and stereo paths. However, the output consists of bits and words of radio data.

The remainder of this document will outline the motivation and objectives of each processing block in the SDR receiver for FM mono, stereo, and RDS, along with its most critical details. *More specific details, such as the design methodology, including implementation choices, trade-offs, and other considerations, will be discussed during class sessions. Feedback on your unique approach (and thought process in general) will be provided during project meetings.*[2]

Before proceeding with the description of each processing block for this project, the following is a summary of the most important takeaways from the lab work:

[2]For general information on topics such as the Raspberry Pi, RTL-SDR, FM, and RDS that are not specific to this course, you should refer to third-party literature.

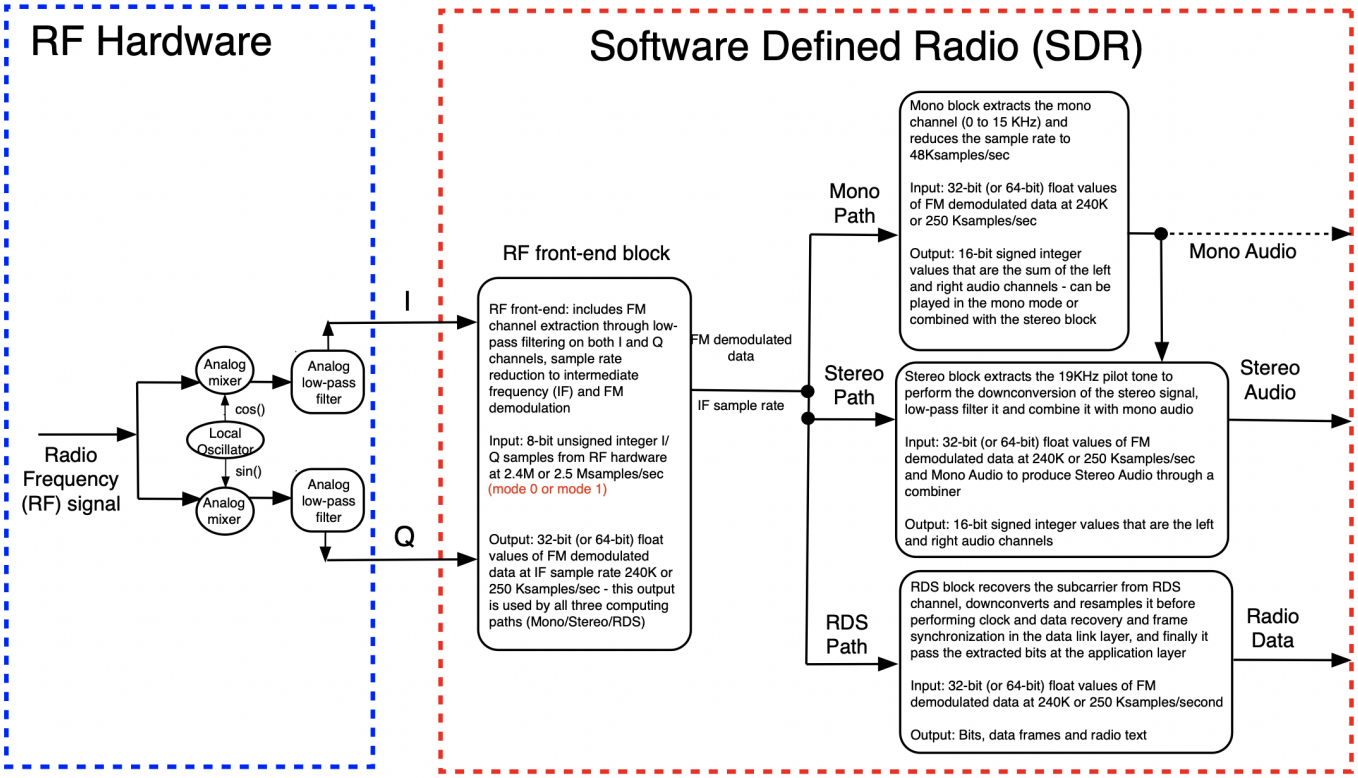


Figure 2: High-level overview of the project.

- Filters are fundamental building blocks in any digital signal processing (DSP) application and can be implemented by convolving the filter's impulse response with the input sequence (a discrete-time sequence of digital samples).
- The impulse response coefficients can be derived using the *sinc* function, based on the Fourier transform relationship between the rectangular window — used to define the passband in the frequency domain — and the *sinc* function in the time domain.
- In practice, ideal (or brick-wall) filters cannot be implemented. Therefore, finite impulse response (FIR) filters, which guarantee a linear phase response, are commonly used. For a given sample rate, the number of FIR taps (input sample delay/coefficient pairs) affects both the stopband attenuation and the width of the transition band between the passband and stopband.
- The filter coefficients depend on the number of FIR taps, the cutoff frequency for lowpass filters (or the start and stop frequencies for bandpass filters), and the sample rate.
- For most practical applications, including SDRs, filtering cannot be performed in a single pass over the entire sequence of input samples. Instead, the input sequence is processed in blocks to avoid excessive latency during data acquisition and large memory requirements. This adds to the implementation challenge. Nonetheless, it is a common issue that must be addressed similarly across a wide range of real-time computing systems.
- A collection of digital filters and other signal processing blocks are connected in a *signal-flow graph*, an abstract representation used to model DSP applications. Depending on the software language, its libraries, and the underlying hardware platform, the execution time

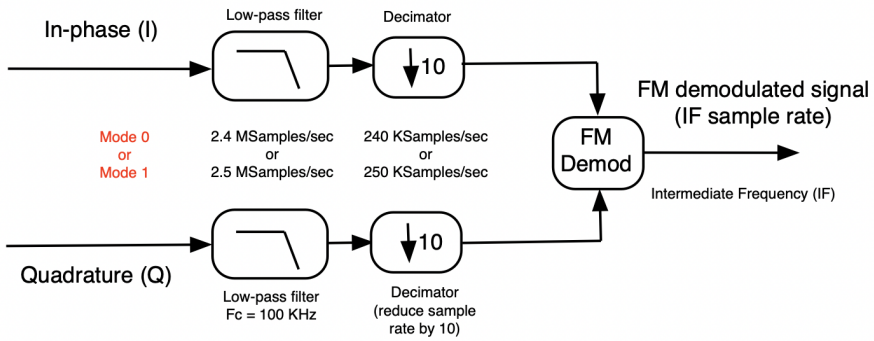


Figure 3: RF front-end processing (in software).

for the same signal-flow graph with identical parameters can vary substantially. Signal-flow graphs should first be modeled in a high-level language, e.g., `Python` with its scientific libraries `SciPy/NumPy`. Only after the functional requirements have been met should they be re-implemented in another language, e.g., `C/C++`, to enable real-time execution.

- The methods used to verify the correctness of an implementation when bringing up a complex system can vary depending on the tasks to be performed and the issues at hand. For example, when dealing with the interface to the physical world, there is no reference data with exact values to match. Therefore, one must rely on visual inspection of power spectra to judge whether the code has been implemented correctly. Conversely, when refactoring `C++` code to make the implementation feasible for real-time execution - e.g., converting convolution from single-pass to block processing - bit equivalence is expected to ensure that no implementation artifacts have been introduced during refactoring.

To conclude the project overview, note that for the machine code compiled from `C++` to operate in real-time, special considerations for optimizing the source code will be necessary. Some of the generic high-level approaches will be discussed during class sessions. However, the specific implementation choices will be left to the discretion of each project group.

RF Front-End Processing

Figure 3 illustrates the signal-flow graph for the RF front-end block. Note the following points:

- Conceptually, the signal-flow graph is identical to the front-end signal-flow graph from the third lab, with one critical difference: there are now two modes of operation for the SDR system (reminder: check the note on page 2 concerning modes of operation). In mode 0, the input sample rate (i.e., RF sample rate) is 2.4 MSamples/sec, while in mode 1, the RF sample rate is 2.5 MSamples/sec. Although this difference is inconsequential to the modeling of the RF front-end block—since the decimation scale factor remains 10 regardless of the mode—it has a ripple effect on resampling in the other processing blocks. It is worth noting that, in any real-time implementation, any filter followed by a decimator $\downarrow$ should compute **only** the samples that will be retained **after** decimation.
- The I/Q samples fed to your software by the RTL-SDR driver [2] will be represented as 8-bit unsigned integers (`unsigned char` in `C++`). If you plan to reuse the software model for this block from the last lab, you will need to normalize these 8-bit unsigned integer values to the range of -1 to +1 in real numbers. The lab code assumes the use of the `float` data type

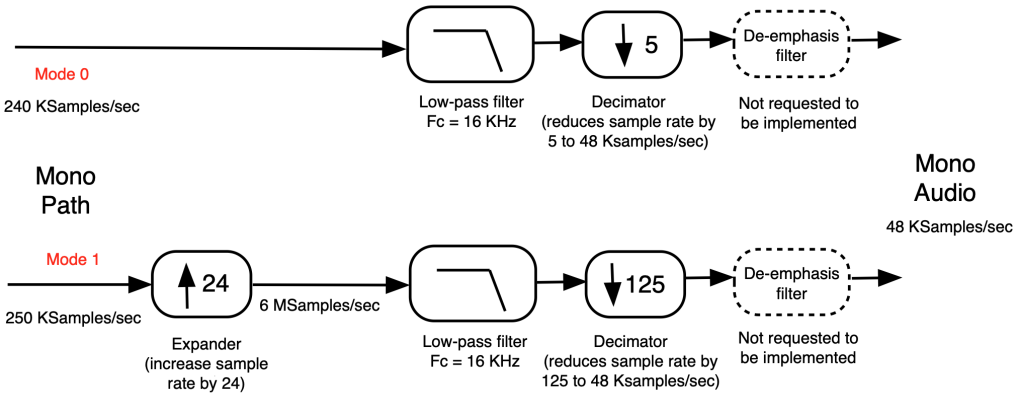


Figure 4: Mono processing path.

in `C++`, representing real values on 32 bits. While the dynamic range and precision provided by the `float` type are usually sufficient, you are advised to consider the `double` data type, which represents real numbers on 64 bits, depending on your design approach.

Using `double` instead of `float` doubles the memory requirements for all blocks that use real numbers. However, it provides finer precision when working with very small values. Since the memory overhead is acceptable, the `Python` code (starter code from the third lab) has been adapted to normalize data to 64-bit real values. This is consistent with the settings in `CMakeLists.txt` (from the fourth lab), which define a macro for conditional compilation of the `C++` code using the `double` type.

- The only constraints on the data types are the 8-bit unsigned integer values for the inputs (I/Q samples) and the 16-bit signed integer values for the outputs (audio samples). The internal data representation is left to the discretion of each project group. The above discussion regarding normalization to the -1 to +1 real range and the choice between `float` and `double` is *a suggestion rather than a requirement.*

Mono Processing

Figure 4 shows the signal-flow graph for mono processing. Note the following points:

- Because background noise in the very high frequency (VHF) range of electromagnetic waves can affect frequencies toward the higher end of the audio spectrum, FM broadcasting stations apply pre-emphasis, i.e., they boost the higher frequency components during transmission. Consequently, receivers are commonly equipped with a de-emphasis filter in the final stage of FM audio decoding. This de-emphasis filter is **ignored** in this project because its absence is hardly noticeable from a user perception perspective. Additionally, the available project time is considered better utilized for developing an understanding of other important implementation details, such as multi-threading, in greater depth.
- Depending on the mode of operation, different types of digital filters and sample rate conversions will be used. In mode 0, the RF sample rate is 2.4 MSamples/sec, the IF sample rate (input to the mono processing) is 240 KSamples/sec, and the audio sample rate (output from the mono processing) is 48 KSamples/sec. This signal-flow graph is the same as the one used in the last lab for mono channel extraction, filtering, and downsampling to the audio sample rate.

- In mode 1 (reminder: check the note on page 2 concerning modes of operation), the RF sample rate is 2.5 MSamples/sec, the IF sample rate is 250 KSamples/sec, and the audio sample rate is 48 KSamples/sec. The fundamental difference from mode 0 is that no integer scale factor can downsample the IF data to the audio data. Instead, a fractional resampler — an upsampler followed by a downsampler — is required.

  Assuming an integer upscale factor of U for upsampling, the first step involves implementing an expander $\uparrow$, which inserts $U-1$ zeros between any two input samples. This produces an output sequence with a sample rate of $U \times \text{IF}$, which is U times larger than the input sample rate of the resampler. Since the IF sample rate is 250 KSamples/sec in mode 1, the output sample rate from the expander is $U \times 250$ KSamples/sec. For example, if $U = 25$, the sample rate at the output of the expander is 6 MSamples/sec, as shown in Figure 4.

  The decimator $\downarrow$ applies an integer downscale factor of D by removing every $D-1$ samples from the output sequence. This reduces the sample rate from $U \times \text{IF}$ to $\frac{U}{D} \times \text{IF}$.

  Since the expander $\uparrow$ needs to be followed by a low-pass filter for *anti-imaging*, and the decimator $\downarrow$ needs to be preceded by a low-pass filter for *anti-aliasing*, these two filters are combined into a single low-pass filter **between** the expander and the decimator. This filter has a maximum cutoff frequency equal to the minimum of $\{\frac{U}{D} \times \frac{\text{IF}}{2}, \frac{\text{IF}}{2}\}$. Note that this cutoff frequency is relative to the signal's sample rate between the expander and the decimator, which is $U \times \text{IF}$.

  It is important to note that, due to the increase in the sample rate caused by zero-padding, **both the size of the impulse response, i.e., the number of filter taps, and the filter's gain must be scaled up by a factor of** U. However, despite this increase in filter size and gain, the number of non-zero partial products contributing to the output signal will remain reasonable. This is because, in the expanded stream, each non-zero value is followed by $U-1$ zeros due to zero-padding. Consequently, the number of non-zero contributions is still approximately equal to the default impulse response size used when no resampling is performed (e.g., 101 taps).

- To reduce the sample rate at which the low-pass filter operates in mode 1, U is chosen as the ratio between the output sample rate (48 KSamples/sec) and the greatest common divisor (GCD) of the input and output sample rates (250 KSamples/sec and 48 KSamples/sec, respectively, for mode 1). This results in $U = 24$ for mode 1. Similarly, $D = 125$, which is the ratio between the input sample rate and the GCD of the input and output sample rates.

  Simple calculations confirm that the maximum cutoff frequency for the low-pass filter between the expander $\uparrow$ and the decimator $\downarrow$ in mode 1 is 24 KHz. However, to extract only the mono channel, we select a lower cutoff frequency of 16 KHz.

- Implementing this filter is particularly challenging because the sampling rate at its input is $U \times \text{IF}$, i.e., 24×250 KSamples/sec $= 6$ MSamples/sec. A large number of filter taps is required to reduce the transition band width and increase attenuation between the passband and stopband. A straightforward implementation would be very computationally demanding. However, by leveraging certain properties of the input sequence to the filter—e.g., the $U-1$ zeros introduced by the expander $\uparrow$ between non-zero samples or eliminating computations for the $D-1$ output samples that are removed by the decimator $\downarrow$—the execution time can be significantly reduced. This optimization technique will be one of the algorithmic methods explored to make real-time implementation feasible.

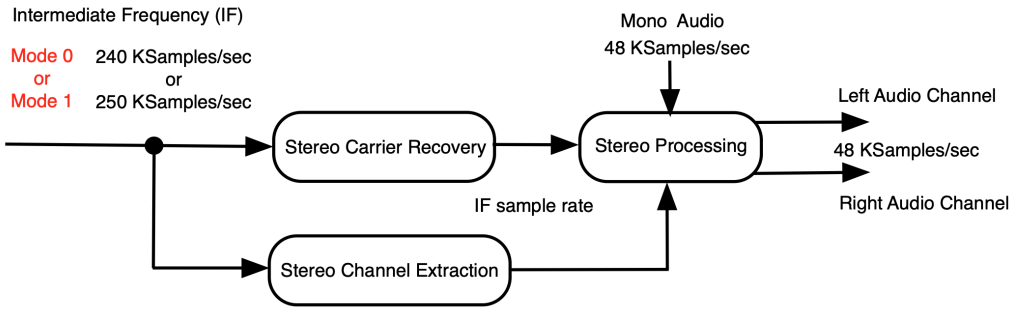


Figure 5: Stereo processing path.

Stereo Processing Path

In FM broadcasting, the stereo signal is modulated onto a 38 KHz subcarrier using Double Sideband Suppressed Carrier (DSBSC) modulation. Since the carrier is suppressed, a 19 KHz pilot tone — at half the subcarrier frequency — is transmitted to enable the receiver to regenerate the 38 KHz carrier for proper demodulation. While the mono signal carries the **sum** of the left and right audio channels (L+R), the stereo signal carries their **difference** (L-R). The bandwidth of the stereo signal is 15 KHz, the same as that of the mono signal, with the two sidebands symmetric around the 38 KHz subcarrier.

The high-level overview of the stereo processing path is shown in Figure 5. The FM demodulated signal at the intermediate frequency (IF) sample rate - either 240 KSamples/sec or 250 KSamples/sec, depending on the mode (reminder: check the note on page 2 concerning modes of operation) - is passed to two separate sub-blocks: stereo carrier recovery and stereo channel extraction. The outputs of these two sub-blocks are fed into another sub-block for stereo processing, which performs digital downconversion by mixing the recovered carrier with the stereo channel. The output of the mixer is then converted to the audio sample rate using the same approach as in the mono processing path. Finally, the stereo data is combined with the mono data to produce the left and right audio channels.

Stereo Channel Extraction and Carrier Recovery

The sub-blocks for stereo carrier recovery and stereo channel extraction are shown in Figure 6. However, before explaining these sub-blocks, it is important to elaborate - using simplified formalism - *why* their outputs need to be used together as inputs to the mixer in the stereo processing sub-block shown on the right-hand side of the figure, to better understand their purpose.

First, it is important to clarify that Double Sideband Suppressed Carrier (DSBSC) is a distinct modulation technique from FM. The stereo signal is upconverted *within* the FM channel at 38 KHz using DSBSC. Note that to avoid any confusion, all baseband data from the FM channel's subchannels are subsequently frequency modulated before being transmitted by the FM broadcaster.

The basic idea of DSBSC modulation is to mix, i.e., multiply, the message signal with the carrier signal as follows:

$$A_m \cos(2\pi f_m t) \times \cos(2\pi f_c t),$$

where A_m is the amplitude of the message signal, f_m is the frequency of the message, and f_c is the carrier frequency. (For simplicity, we ignore the amplitude of the carrier and the phases of the two cosine waves.)

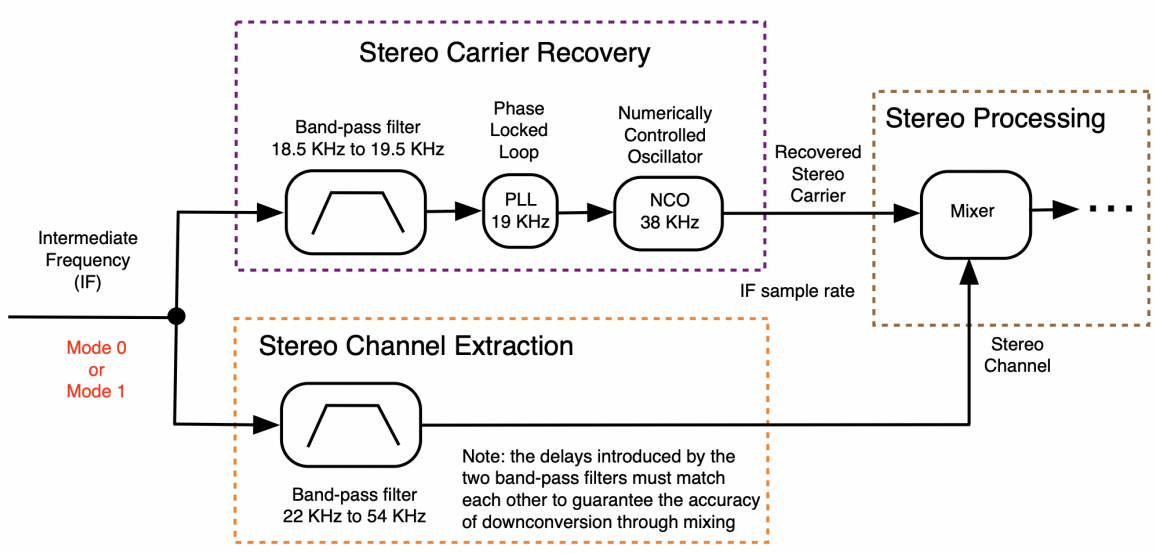


Figure 6: Stereo channel extraction and carrier recovery.

Using trigonometric identities, the multiplication above translates to the following DSBSC-modulated signal:

$$\frac{A_m}{2}\left[\cos(2\pi(f_c + f_m)t) + \cos(2\pi(f_c - f_m)t)\right].$$

In the receiver, demodulation is performed by applying the same type of mixing, i.e., multiplying the received DSBSC-modulated signal by the carrier signal, which produces:

$$\frac{A_m}{4}\left[2\cos(2\pi f_m t) + \cos(2\pi(2f_c + f_m)t) + \cos(2\pi(2f_c - f_m)t)\right].$$

It can be seen that the first term represents the original message signal (with its amplitude scaled down by a factor of 2), while the other terms represent high-frequency components, assuming that f_c is much larger than f_m. Therefore, if we apply a low-pass filter to the output of the mixer - on the right-hand side of Figure 6 - we recover the original message. The gain of the low-pass filter must be adjusted to compensate for the amplitude loss caused by the mixing process during transmission.

To perform the above-described downconversion accurately, we must first extract the stereo channel. This can be achieved using a bandpass filter with a start frequency (f_{low}) of 22 KHz and an end frequency (f_{high}) of 54 KHz. This is the purpose of the stereo channel extraction sub-block shown at the bottom of Figure 6. Equally important is the need to mix the stereo channel at the receiver with a carrier frequency that is **synchronized** with the subcarrier frequency used by the *mixer at the transmitter*.

Since DSBSC suppresses the carrier to save transmit power, FM stations broadcast a 19 KHz stereo pilot tone. The second harmonic of this pilot tone is used as the subcarrier for mixing at the transmitter. Note that the 19 KHz pilot tone is transmitted as a standalone signal, positioned between the mono and stereo sub-channels within the FM channel.

As shown in the Stereo Carrier Recovery sub-block from Figure 6, the first step is to extract the 19 KHz pilot tone using a band-pass filter. Next, the pilot tone is synchronized using a phase-locked loop (PLL). Finally, the output of the PLL is multiplied by a scale factor of 2 using a numerically controlled oscillator (NCO) to generate the recovered stereo carrier for mixing.

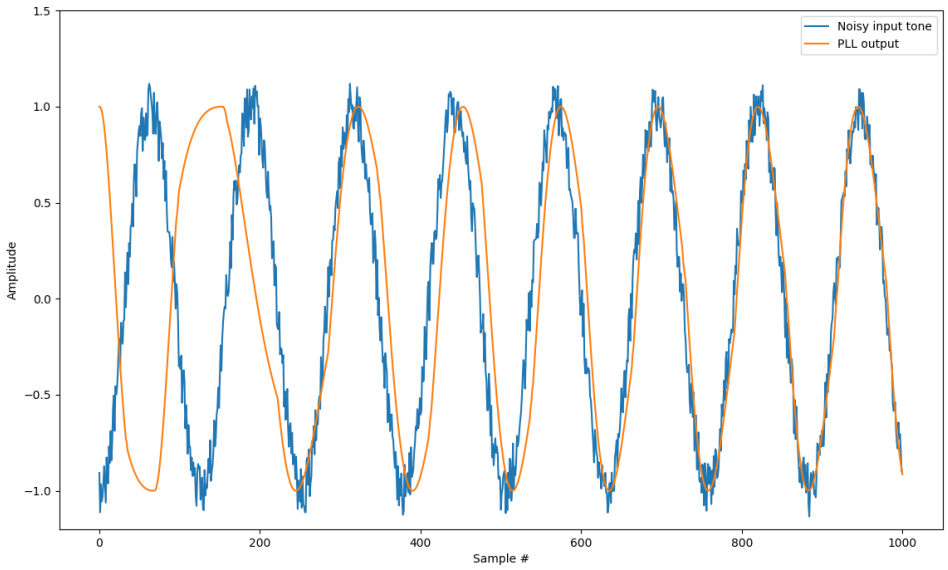


Figure 7: Phase-locked loop (PLL).

It is important to clarify that phase-locked loops (PLLs) are phase-tracking devices traditionally implemented using analog integrated circuits. However, software PLLs can be reliably implemented today for reasonable frequency ranges, even on embedded computing devices. The applications of PLLs in electrical and computer engineering are extensive, ranging from controlling electrical machines to clock distribution in microprocessors and maintaining synchronization in wireless communications and data transmission.

Elaborating on the inner workings of a PLL is beyond the scope of this project document. Nonetheless, at an intuitive level, it is worth mentioning that an essential goal of a PLL is to produce a *clean* output signal - e.g., filtered and amplified if the input signal weakens - that remains locked to a noisy input. A PLL continuously tracks the phase of the input signal and adjusts the output frequency and phase to maintain synchronization.

An inherent trade-off in PLL design is that the faster a PLL can lock, the more phase jitter - rapid, short-term fluctuations in phase - appears in the output. This increased jitter can degrade the quality of the output signal. Conversely, if the PLL prioritizes reducing jitter, it may take longer to lock onto the input signal, potentially making it less effective at tracking rapidly changing inputs. Additionally, a PLL that locks quickly may struggle to maintain synchronization with weak or highly noisy signals.

These trade-offs are particularly important in applications such as FM receivers, where both fast locking and low jitter are desirable but difficult to achieve simultaneously. Careful tuning of PLL parameters - such as loop bandwidth and damping - is required to balance these competing requirements effectively. Therefore, to ease development, the backbone code for a software PLL/NCO will be provided in `Python`.

To visualize how a PLL works, Figure 7 illustrates how a PLL can lock to the phase of a periodic but noisy input signal and output a “clean” version, which is then passed to the numerically controlled oscillator (NCO) for frequency scaling. It is also worth noting that the waveforms in Figure 7 were generated using a noisy input tone at 19 KHz, oversampled at 2.4 MSamples/sec. As shown, the PLL is capable of locking within fewer than 1,000 samples.

Both the stereo carrier recovery and stereo channel extraction rely on bandpass filtering. There are two important points to note:

- The number of filter taps for the two bandpass filters in Figure 6 should be matched to ensure the same delay from the FM demodulated data to both inputs of the mixer.
- The pseudocode for deriving the impulse response coefficients follows the structure and notation from the first lab, where pseudocode was provided for deriving the impulse response coefficients for a lowpass filter.

Input: Filter parameters: f_{low} (low cutoff frequency), f_{high} (high cutoff frequency), f_s (sampling rate), N_{taps} (number of filter taps)
Output: Coefficient weights for the FIR filter $h(k)$, where $k = 0, \ldots, N_{\text{taps}} - 1$

$center \Leftarrow (N_{\text{taps}} - 1)/2$ ▷ Center index for the *sinc* function
$f_{\text{mid}} \Leftarrow (f_{\text{low}} + f_{\text{high}})/2$ ▷ Mid-frequency of the passband
$scale\_factor \Leftarrow 0$ ▷ Initialize the scale factor

for $k \in [0, N_{\text{taps}} - 1]$ **do**
  $n \Leftarrow k - center$
  if $n = 0$ **then**
    $h(k) \Leftarrow \frac{2f_{\text{high}}}{f_s} - \frac{2f_{\text{low}}}{f_s}$ ▷ Handle center tap
  else
    $h(k) \Leftarrow \frac{\sin\left(2\pi f_{\text{high}} n/f_s\right)}{\pi n} - \frac{\sin(2\pi f_{\text{low}} n/f_s)}{\pi n}$ ▷ Compute the *sinc*-based response
  end if
  $h(k) \Leftarrow h(k) \cdot \left(\frac{1}{2} - \frac{1}{2}\cos\left(\frac{2\pi k}{N_{\text{taps}}-1}\right)\right)$ ▷ Apply the Hann window
  $scale\_factor \Leftarrow scale\_factor + h(k) \cdot \cos\left(\frac{2\pi n \cdot f_{\text{mid}}}{f_s}\right)$ ▷ Accumulate the scale factor
end for

for $k \in [0, N_{\text{taps}} - 1]$ **do**
  $h(k) \Leftarrow h(k)/scale\_factor$ ▷ Normalize the coefficients
end for

Stereo Processing: Downconversion, Filtering and Combining

The final sub-block of the stereo path, shown in Figure 8, performs downconversion through mixing, sample rate conversion, and stereo combining. The principle of downconversion through mixing has been explained at the beginning of the previous subsection. From a computational standpoint, mixing is a low-demand task because the two input streams are multiplied sample by sample (pointwise multiplication).

However, the output of the mixer places a significantly higher computational demand on the system, as it follows the same principles and computational path for digital filtering and sample rate conversion as for the mono channel. Note that the mixer must be followed by a lowpass filter, which is "absorbed" in the digital filtering sub-block. Recall that the gain of the lowpass filter after mixing must be scaled by a factor of two to compensate for the amplitude loss caused by the mixing process.

The stereo data at the audio sample rate is then combined with the mono audio to produce the left and right audio channels, which can subsequently be fed to an audio player.

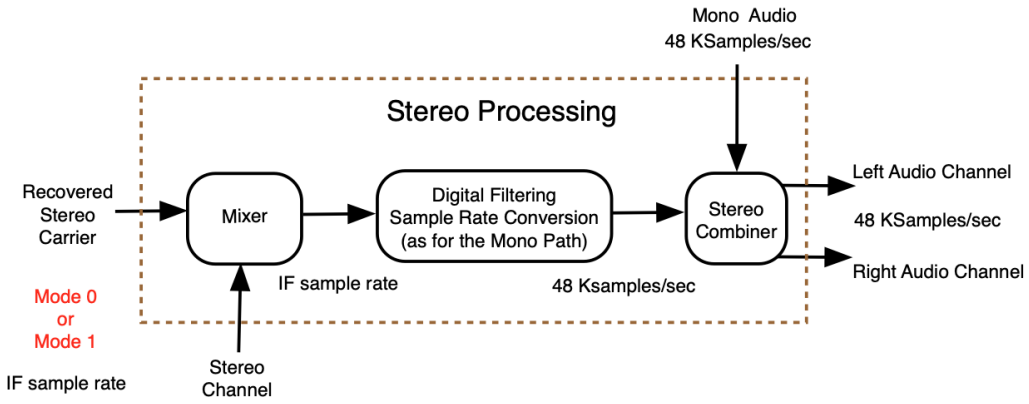


Figure 8: Stereo downconversion, filtering and combining.

Allpass Filter on the Mono Path to Match the Delay on the Stereo Path

In the context of this SDR project, the phase delay introduced by signal processing blocks is only significant when the signal "branches" at a stem node in the signal flow graph and then "reconverges" at a later point. As shown in Figure 8, the samples processed by the mono path are **combined** with the samples processed by the stereo path. Since both paths originate from the FM demodulator output in Figure 3, any mismatch in delays between the two paths will cause the recombined samples to be out of phase. This phase misalignment can lead to audio artifacts during playback, which will be particularly noticeable for content where the difference between the left and right audio channels is significant.

For most FM broadcasts, however, these artifacts are subtle or even negligible because the stereo signal typically has a much lower amplitude than the mono signal. This is because the mono path carries the sum of the left and right audio channels (L+R), while the stereo path carries their difference (L-R). As a result, when the left and right channels contain nearly identical content, the signal strength in the stereo path is minimal. Since this applies to most real-life content, phase misalignment can easily go unnoticed when relying on typical signals acquired from the "airwaves." In fact, even if the PLL is incorrectly implemented, one could mistakenly conclude that the stereo path is functioning correctly if the (L+R) signal is significantly stronger than the (L-R) signal, as it is the case for typical audio content. [3]

To address the phase misalignment between the mono and stereo paths - caused by the additional phase delay in the stereo path - an allpass filter must be introduced in the mono path, as illustrated in Figure 9. This step should be taken after the stereo path has been fully implemented. Further details on the rationale and the approach to be followed are provided next.

A fundamental property in signal theory states that FIR filters with symmetrical coefficients around the center coefficient exhibit a linear phase response. The phase delay introduced by such filters results in a real-time delay, measured in seconds, and is given by:

$$\text{Phase Delay} = \frac{N-1}{2F_s},$$

where N is the number of filter taps and F_s is the sample rate of the input signal fed into the FIR filter.

[3] To ensure that the stereo path can be stress-tested under more challenging conditions, a couple of recordings with noticeable differences between the left and right channels will be provided.

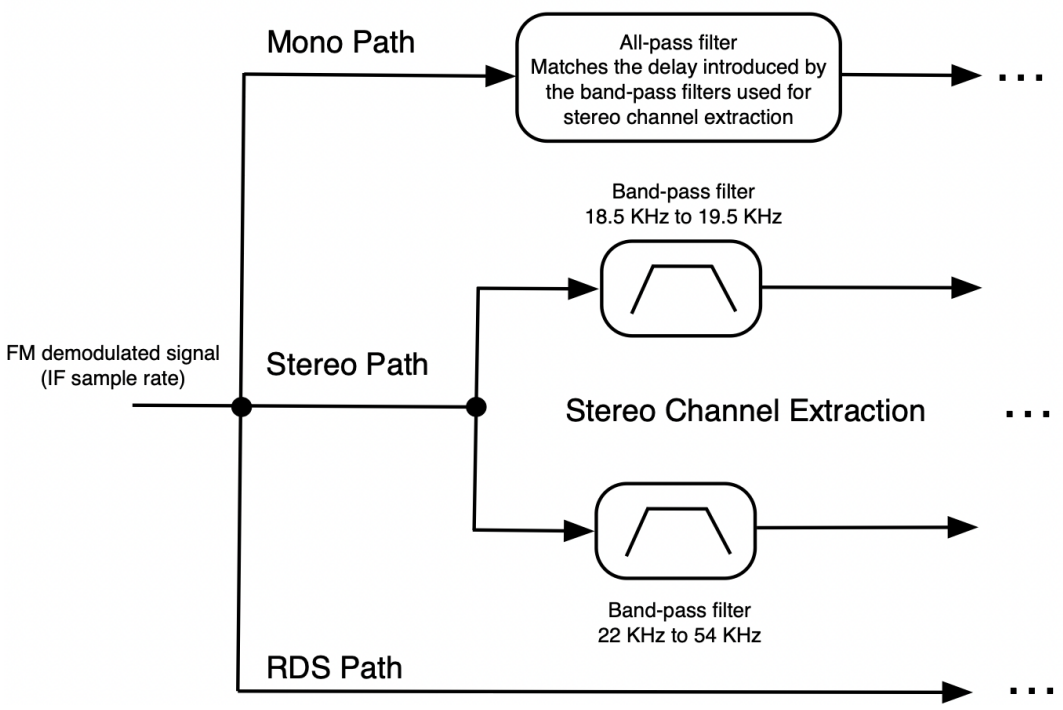


Figure 9: Introducing an allpass filter to delay the mono path to match the sample delay caused by the bandpass filters used for stereo recovery.

To ensure matching delays when the signal branches and reconverges, both the bandpass filter used to extract the stereo channel and the bandpass filter for the 19 KHz pilot tone must have the same number of taps. This guarantees that the phase delay on both branches will be identical when they are recombined at the mixing point shown in Figure 6.

Instead of using a full filter to replicate this phase delay on the mono path, a simpler and more efficient solution is to introduce a delay block. This block can be implemented using the same principles as a shift register in hardware, with the number of delayed samples equal to $(N-1)/2$, where N is the number of taps in the stereo path's filters.

For practical implementation, the delay block on the mono path should be set to match the cumulative delay caused by the two bandpass filters in the stereo path. For example, if each bandpass filter has 101 taps, the required delay at the input of the mono path would be:

$$\text{Delay} = \frac{101-1}{2} = 50\,\text{samples}.$$

It is also important to consider the timing of this implementation. The delay block should only be added to the mono path **after** the stereo path has been fully implemented and verified. This ensures that the combined audio output - consisting of both mono and stereo components - is correctly synchronized and free of phase artifacts. Adding the delay block prematurely can complicate debugging and validation of the overall system.

Finally, it is worth noting that no similar delay block is needed for the RDS path. Unlike the content on the mono and stereo paths, RDS data - e.g., radiotext and other metadata - does not need to be accurately synchronized with audio samples. Since there is no point where the RDS path reconverges with any of the audio paths, similar delay blocks on the RDS path are unnecessary.

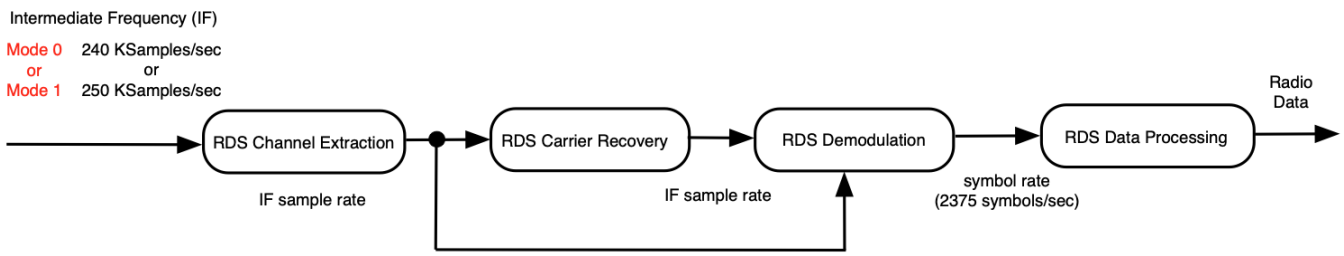


Figure 10: RDS processing path.

RDS Processing Path

The RDS processing path is responsible for extracting the RDS channel (54 to 60 KHz, as shown in Figure 1), downconverting it to baseband without explicit synchronization to a pilot tone, and demodulating it to extract the digital bitstream. Although the digital data rate is low, this processing path presents a significant computational challenge because the RDS signal is relatively weak compared to both the mono and stereo signals.

Figure 10 shows the high-level diagram of the RDS processing path. The input to the RDS path is the same FM-demodulated signal that drives the mono and stereo paths (see Figure 2). This signal is at the intermediate frequency, sampled at 240 KSamples/sec in mode 0. Similar to the stereo processing path, the first step is extracting the RDS channel. Subsequently, the carrier is recovered; however, unlike in the stereo path, the recovery is **not** based on locking explicitly to a harmonic of the pilot tone. Instead, consistent with most digital communication standards, the carrier is extracted directly from the signal, as elaborated in the following sub-section.

The main processing step concerns RDS demodulation, which involves downconversion, sample rate conversion to a rate that is a multiple of the symbol rate, matched filtering, and timing recovery. Finally, the extracted symbols, at a rate of 2,375 symbols/sec, are passed to the RDS data processing stage, where they are converted to a bitstream at 1,187.5 bits/sec. In the final step, frame synchronization is performed before extracting the radio data.

RDS Channel Extraction and Carrier Recovery

The RDS channel is upconverted at the transmitter using the same principle as the stereo channel. When extracting the channel with a bandpass filter, the RDS channel has a center frequency of 57 KHz and a bandwidth of 6 KHz. All the information is subsequently extracted after downconversion from the positive sideband, where the digital data is binary-phase (bi-phase) coded at 2,375 symbols/sec. Since the RDS signal is relatively weak compared to the mono and stereo channels, precise filtering is essential to maintain synchronization with the RDS sub-carrier. As shown in Figure 11, the bandpass filter for RDS channel extraction operates at the IF sample rate, and the data from the RDS channel is passed to the RDS carrier recovery sub-block.

The RDS carrier recovery sub-block facilitates downconversion to baseband by mixing the recovered carrier with the RDS channel. The key difference from the stereo path lies in **not** locking to the third harmonic of the pilot tone. This is because not all FM stations synchronize the RDS sub-channel to the pilot tone, and this approach demonstrates a digital-centred method for carrier recovery. Unlike analog methods, this digital approach extracts the carrier directly from the signal using techniques robust to phase and frequency variations.

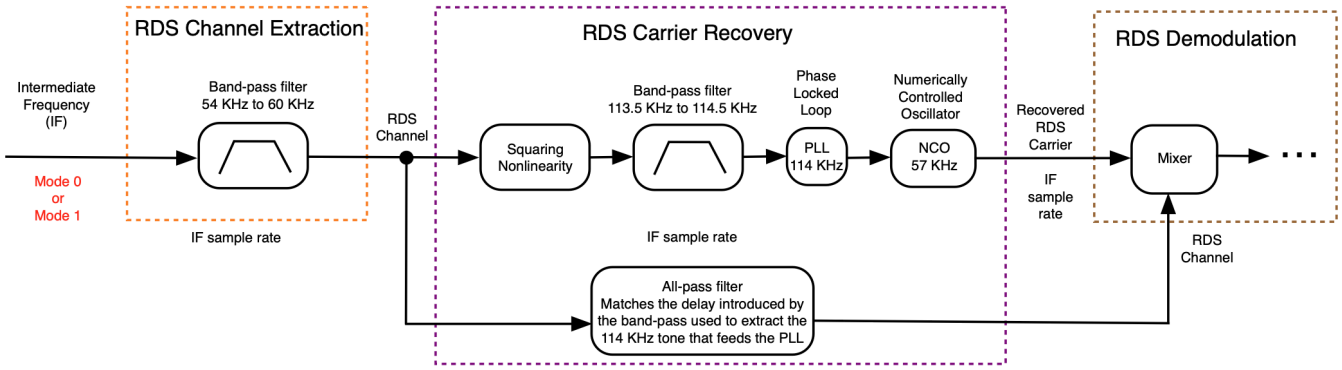


Figure 11: RDS channel extraction and carrier recovery.

The core idea of this approach is based on the fact that squaring the message signal $A_m \cos(2\pi f_m t + \phi)$ results in:

$$\frac{A_m^2}{2}\left(1 + \cos(4\pi f_m t + 2\phi)\right).$$

The squared signal has a frequency of $2f_m$ and a phase of 2ϕ. In bi-phase coding, commonly used in digital communication, each bit period contains at least one phase transition. This is achieved by encoding '0' and '1' with opposite phase shifts (e.g., $\phi = 0$ for '0' and $\phi = \pi$ for '1'). When the signal is squared, the phase is doubled, resulting in either 0 or 2π, which corresponds to no net phase shift. This property is key to simplifying carrier recovery since the squared signal becomes phase-aligned with itself at double the frequency.

This concept is implemented using the squaring non-linearity block from Figure 11, which performs a pointwise multiplication of each sample with itself. The output of this block is passed through a narrow bandpass filter to extract a 114 KHz tone, which is double the center frequency of the RDS channel, i.e., 57 KHz.

Although drawn separately, the PLL and NCO can be integrated into a single module to produce a 57 KHz signal, which is used to mix the RDS channel as part of the downconversion process in the RDS demodulator. The entire RDS carrier recovery sub-block operates at the IF sample rate. Since the input to the PLL is a 114 KHz tone, it becomes clear why the IF sample rate in either of the two supported modes has been chosen to exceed 228 KSamples/sec. This ensures that the Nyquist criterion is satisfied, allowing processing of the 114 KHz input tone.

It is important to note that for the mixing process to work correctly, the RDS channel must be delayed by the same amount as the bandpass filter that extracts the 114 KHz tone. This is the purpose of the allpass filter block at the bottom of Figure 11. [4] Without this delay, phase misalignment between the RDS channel and the recovered carrier would invalidate the downconversion process.

Although it is not required at runtime, during learning, modeling, troubleshooting, and tuning of the carrier recovery block, it is recommended to process both the in-phase (I) and quadrature (Q) components of the RDS channel. This can be achieved by using two outputs from the numerically controlled oscillator (NCO): the default in-phase output and the quadrature output, which is a 57 KHz tone delayed by $\frac{\pi}{2}$.

[4] Following the same reasoning as in Figure 9, this allpass filter can be implemented by delaying the input by $(N-1)/2$ samples, where N is the number of filter taps in the bandpass filter that extracts the 114 KHz tone.

While many advanced digital modulation techniques require simultaneous in-phase and quadrature mixing, this is not strictly necessary for binary phase-shift keying (BPSK) modulation, which is referred to as bi-phase coding in the RBDS standard [12]. In BPSK, the modulated signal carries information solely on the phase of the in-phase component, with phase shifts of 0 or π representing binary symbols. Therefore, the quadrature path can be disabled during real-time execution once the system is tuned. However, during the tuning phase, the quadrature path is useful for adjusting the NCO's phase output. This is done by analyzing the phase shifts and symbol transitions observed in the constellation diagram, which can be visualized after timing recovery. These adjustments help ensure that the in-phase component used for mixing aligns correctly with the RDS carrier phase, improving the accuracy of data demodulation. Details on this tuning process will be illustrated in the following sub-section.

RDS Demodulation

Figure 12 shows the top-level diagram of the RDS demodulator. Each stage will be elaborated below.

Mixing and Lowpass Filtering

The front end of the RDS demodulator is similar to that of the stereo path, as the baseband signal is extracted after mixing through a lowpass filter. For RDS, the cutoff frequency of this filter is set to 3 KHz, which is sufficient to capture the full bandwidth of the 2,375 symbols/sec digital data stream.

Rational Resampler

After filtering, the next step involves a rational resampler, which adjusts the IF sample rate to a new rate that is an integer multiple of the symbol rate. This ensures that each symbol can be represented by a consistent number of samples, which is needed for timing recovery. The resampler operates on the same principles as the mono path, using both expansion and decimation (see Figure 4 for details).

Assume an IF sample rate of 240 KSamples/sec and that the number of samples per symbol (SPS) is set to 24. The desired output sample rate for the resampler would be:

$$\text{Output Sample Rate} = 24 \times 2,375 = 57,000\,\text{Samples/sec}.$$

To determine the expansion and decimation ratios for the resampler, the greatest common divisor (gcd) between 240,000 and 57,000 is computed as:

$$\gcd(240,000, 57,000) = 3,000.$$

Thus, the expansion and decimation factors become:

$$\text{Expansion Factor} = \frac{57,000}{3,000} = 19, \quad \text{Decimation Factor} = \frac{240,000}{3,000} = 80.$$

The anti-imaging/anti-aliasing filter within the resampler operates at an internal sample rate of:

$$240\text{KSamples/sec} \times 19 = 4.56\,\text{MSamples/sec}.$$

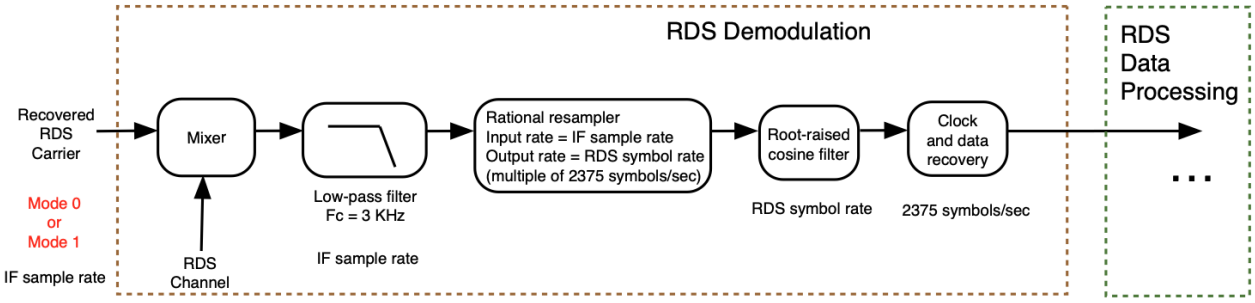


Figure 12: RDS demodulation.

Now, consider the case where the SPS is set to 33. The desired output sample rate becomes:

$$\text{Output Sample Rate} = 33 \times 2,375 = 78,375\,\text{Samples/sec}.$$

The gcd between 240,000 and 78,375 is:

$$\gcd(240,000, 78,375) = 375.$$

The corresponding expansion and decimation factors are:

$$\text{Expansion Factor} = \frac{78,375}{375} = 209, \quad \text{Decimation Factor} = \frac{240,000}{375} = 640.$$

The internal sample rate for the anti-imaging/anti-aliasing filter in this case becomes:

$$240\text{KSamples/sec} \times 209 = 50.160\,\text{MSamples/sec}.$$

Reminder: Check your group's project repository for the specific SPS assigned to your project. This value will define the output sample rate of the resampler when the RDS path is enabled.

Root-Raised Cosine Filter

Without elaborating on the detailed theory of pulse shaping in digital communications, it is important to clarify key concepts to understand the role of the next sub-block in the RDS demodulator: the root-raised cosine (RRC) filter [13]. The primary purpose of this filter is to reduce inter-symbol interference (ISI), which occurs when overlapping digital pulses interfere with one another, leading to errors in symbol detection.

In digital communications, pulses are "shaped" before transmission to reduce spectral bandwidth and maximize data transmission rates. However, limiting the spectral bandwidth in the frequency domain leads to infinite pulse duration in the time domain, which can increase ISI. To address this, digital pulses are typically shaped using a *Nyquist filter*. A Nyquist filter is designed to produce an impulse response that is nonzero at one specific sampling point (typically the center of a symbol period) and zero at all other symbol-spaced sampling points. This ensures that symbols do not interfere with each other at these critical sampling intervals, thereby minimizing ISI.

The root-raised cosine (RRC) filter plays a key role in this process. It is important to note that the RRC filter is **not** a Nyquist filter when implemented on its own. However, when an RRC filter is applied for spectrum shaping at **both** the transmitter and the receiver, the combined effect results

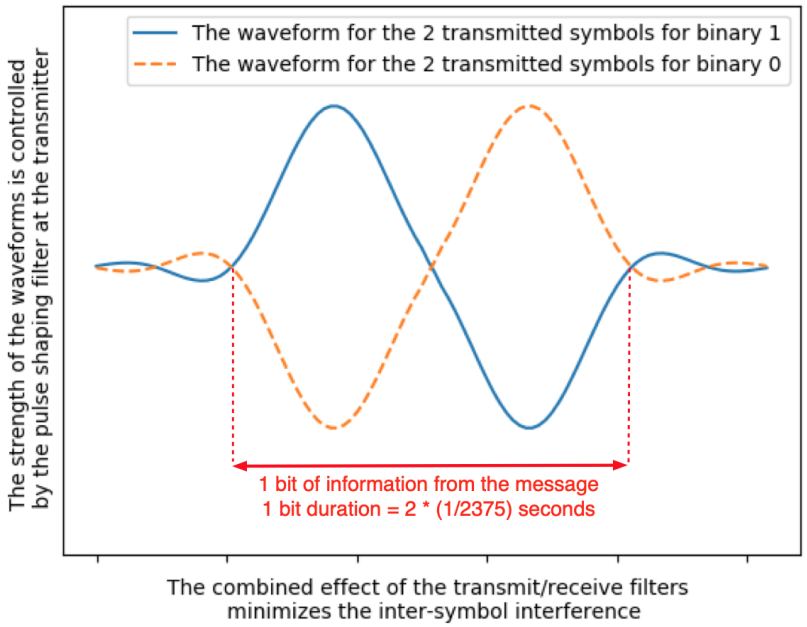


Figure 13: From bits to waveforms.

in a raised-cosine filter, which **is** a Nyquist filter. This configuration achieves the desired zero ISI condition at symbol-spaced intervals, while still controlling the signal's spectral bandwidth.

In summary, the RRC filter in the RDS demodulator is crucial for ensuring that symbols can be reliably separated at the receiver by reducing ISI[5]. The filter operates on the incoming signal after it has been resampled to a rate that is an integer multiple of the symbol rate.

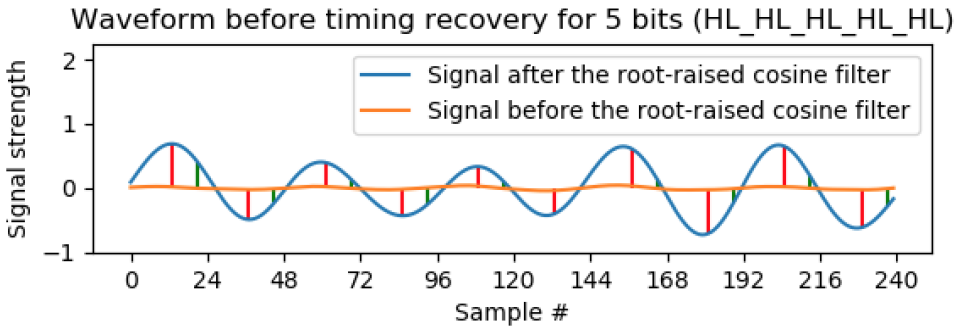
Clock and Data Recovery

Before transmission, RDS data is Manchester encoded, meaning that each binary bit is represented by **two** symbols. Specifically, a binary 1 is translated into a High followed by a Low symbol (labelled as HL), while a binary 0 is translated into an Low followed by a High symbol (labelled as LH). This encoding scheme is illustrated in Figure 13, which shows the shape of the **two** symbols corresponding to each binary value. Since there are two symbols per bit, the symbol rate is 2,375 symbols/sec, while the bitrate is half of that, i.e., 1,187.5 bits/sec.

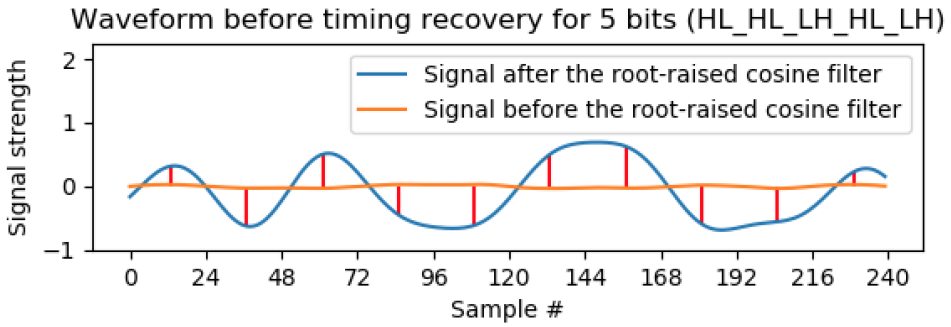
It is important to note that in a receiver, the RRC filter is also used to provide gain to the weak RDS signal extracted from the FM channel. The relative size of the input and output of the RRC filter is illustrated in Figure 14. This additional gain is necessary for timing recovery (also referred to as clock and data recovery), which is a critical block in any digital communication system.

Timing recovery involves determining the optimal sampling instants for each received symbol. While knowing the expected shape of the transmitted symbols is essential, it is not sufficient on its own to correctly recover binary data. The receiver must also decide when to sample **each** symbol to minimize errors. Figure 14a demonstrates two potential choices for sampling times (assuming 24 samples per symbol). The sampling times indicated by the red vertical lines clearly result in better symbol detection compared to those indicated by the green vertical lines.

[5]A `Python` model for generating the RRC impulse response will be provided.



(a) Waveform of the signal at the receiver for five consecutive bits equal to binary 1 (each HL pair is a logic 1 bit); the red lines show a good choice whereas the green lines show a poor choice for sampling the symbols.



(b) A different waveform for binary sequence 11010 illustrating the shape of waveforms for real data that was bi-phase coded, as observed at the receiver.

Figure 14: From waveforms to bits.

The main challenge in timing recovery lies in devising an algorithm that can automatically adjust the sampling instants. This algorithm must adapt to variations in the received signal, including potential changes in symbol timing due to noise or jitter.

There are a few important points to be made regarding the signal processing flow. First, only the in-phase signal (i.e., the signal obtained by mixing the RDS channel with the in-phase output of the NCO) is shown in Figures 14a and 14b. This explains, in part, the variation in pulse intensity, as even for a well-tuned PLL/NCO, a portion of the signal energy will appear in the quadrature component. This phenomenon is observable in the constellation diagrams from Figures 15 and 16.

It is important to note that after the PLL/NCO has been tuned to account for the filter delay used for carrier recovery, only the in-phase component is sampled for BPSK data. In Figure 14b, it can be seen that for BPSK-modulated data, a logic 0 (represented by the symbols LH) is simply a phase-shifted version of a logic 1 (symbols HL), with a phase shift of π radians between them.

For real-world RDS data, symbols are often incorrectly sampled due to factors such as noise, jitter, or frequency offsets. These errors result in bit errors that are eventually detected—and possibly corrected—later in the processing flow through error-checking mechanisms such as cyclic redundancy checks (CRC) or forward error correction (FEC). It is important to note that while this project includes the detection of bit errors as part of RDS data processing, the correction of these errors is beyond the scope of the project.

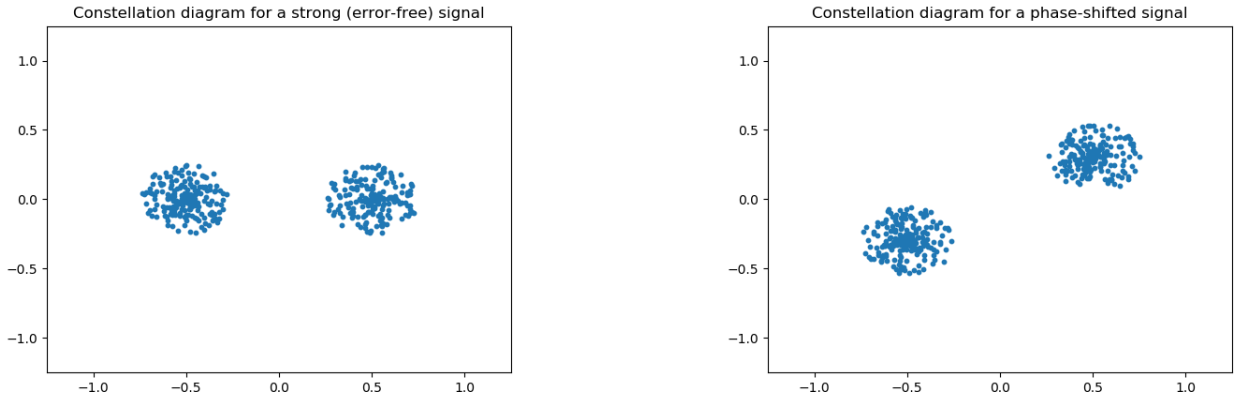


Figure 15: Constellation diagrams for idealized signals.

Another point worth emphasizing for the RDS demodulator is the importance of monitoring signal constellations during the modelling and tuning phase, as illustrated in Figures 15 and 16. Constellation diagrams visually represent the demodulated symbols (after timing recovery) as a scatter plot on the complex plane, displaying **both** the in-phase and quadrature components of the signal. These diagrams provide valuable insights into the signal's phase and amplitude characteristics.

As mentioned earlier, processing both the in-phase and quadrature components is essential for more advanced digital modulation techniques, such as quadrature phase-shift keying (QPSK) and quadrature amplitude modulation (QAM). For BPSK, however, only the in-phase component carries useful information, since binary data is encoded by shifting the carrier's phase between 0 and π radians.

Nonetheless, during the modeling phase, monitoring the constellation diagram remains crucial to tune the PLL/NCO. The objective is to transfer most of the signal energy from the RDS channel to the in-phase component after downconversion. Any residual signal energy appearing in the quadrature component can indicate phase misalignment that needs adjustment. An example of such a phase shift that requires correction is shown on the right-hand side of Figure 15, where the distribution of points deviates from the ideal axis alignment. Adjusting the PLL/NCO parameters ensures that the in-phase symbols are properly centered, improving data demodulation accuracy.

It is important to note that the constellation diagrams in Figure 15 are idealized, meaning they have been artificially created with a clear separation between all the sampled `H` and `L` symbols. In practice, constellation diagrams for real RDS data captured from the airwaves often resemble those shown in Figure 16. Various channel impairments, such as noise, interference, or weak signal strength, can cause the sampled symbols to deviate significantly from their ideal positions.

If the signal is heavily corrupted - e.g., due to a weak transmission - meaningful data extraction becomes difficult or even impossible. This is illustrated on the left-hand side of Figure 16, where the symbols are scattered without any discernible separation, indicating poor signal quality. Similar degraded constellations can occur due to receiver-side issues, such as an incorrectly implemented phase-locked loop (PLL), faulty downconversion, or errors in timing recovery.

However, when the receiver is correctly tuned and the demodulated signal is sufficiently strong, the constellation diagram improves, as shown on the right-hand side of Figure 16. Here, the `H` and `L` symbols exhibit better separation, allowing for reliable symbol detection and bit extraction. In this case, data frames can be correctly identified and decoded, as explained in the next section.

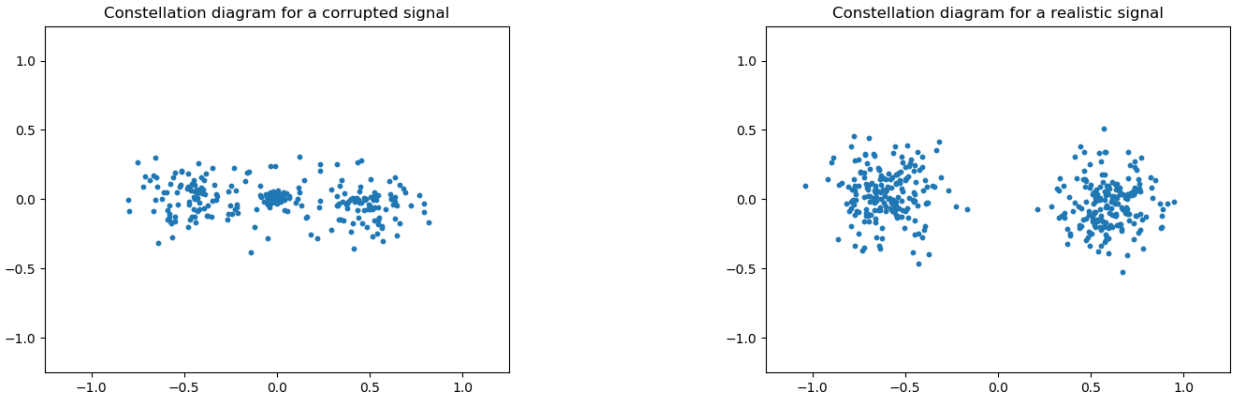


Figure 16: Constellation diagrams for bi-phase coded signals extracted from the RDS sub-channel broadcast by an FM station.

RDS Data Processing

Once the waveforms have been translated into symbols, the final step in the RDS processing path involves converting symbols to bits and identifying data frames within the bitstream. In RDS, these frames are organized into structures called groups, which are further divided into blocks. After the "framed bits" are extracted, they are parsed, and the corresponding text messages are displayed.

The application layer (also referred to as the presentation/session layers in the RBDS standard [12]) is responsible for digital message formatting. This layer handles the interpretation of data structures such as program service (PS) names, radio text (RT), and other metadata transmitted through the RDS channel, ensuring that the information is properly formatted and presented to the end user.

Manchester and Differential Decoding

The first step within the RDS Data Processing block, shown in Figure 17, is to convert the stream of symbols obtained from the clock and data recovery sub-block into a bitstream. Since the transmitted data is Manchester encoded (as discussed earlier with Figure 13), where logic 1 is represented by the symbol pair (HL) and logic 0 by (LH), the symbol rate is 2,375 symbols/sec. Consequently, the extracted bitstream has a bitrate of 1,187.5 bits/sec, since there are two symbols for each bit.

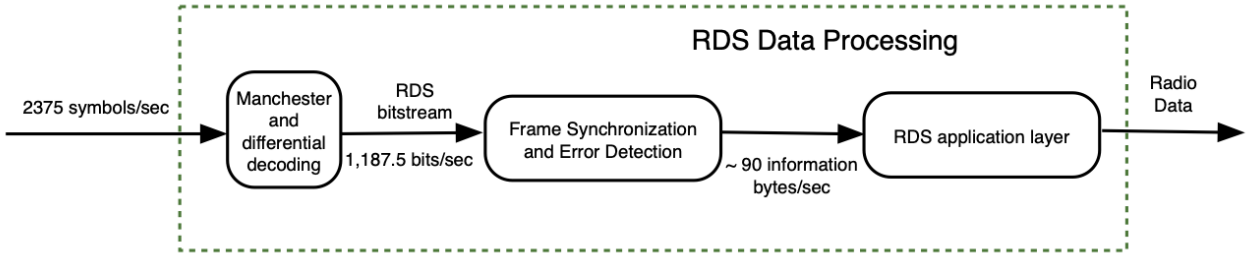


Figure 17: RDS data processing.

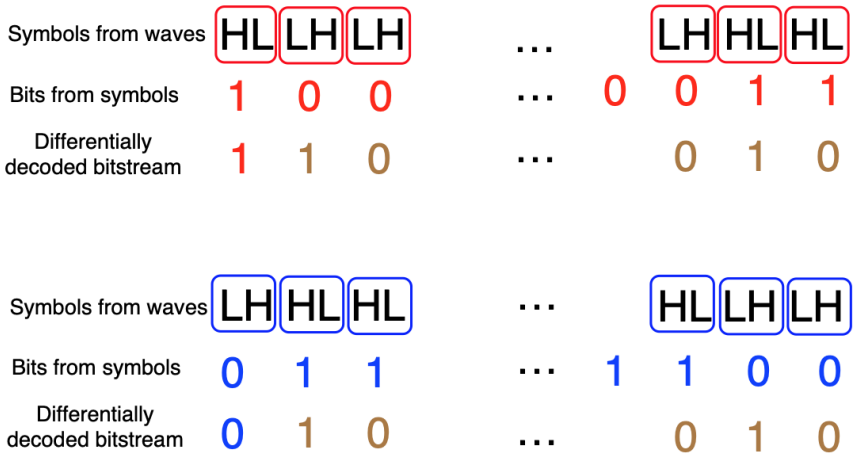


Figure 18: Differential decoding.

It is also important to note that the bitstream is *differentially encoded* at the transmitter before being converted to symbols. Differential encoding involves performing an exclusive OR (XOR) operation between the current bit and the previous bit, resulting in a differentially encoded stream. Therefore, to correctly reconstruct the original bitstream, the receiver must apply **differential decoding**.

The motivation for using differential coding is closely tied to the method of carrier synchronization in the RDS receiver. As discussed earlier, the receiver's phase-locked loop (PLL) locks to an extracted 114 KHz tone, which is double the frequency of the RDS sub-carrier (57 KHz). Because of this, the output of the numerically controlled oscillator (NCO) can exhibit two possible phase shifts relative to the RDS channel data: 0 radians or π radians. If the output phase is inverted (i.e., shifted by π radians), it would cause a complete inversion of the symbol stream. Differential coding ensures that such phase inversions do not affect the interpretation of the binary data, since only the relative phase difference between consecutive bits matters.

A detailed example of differential decoding is shown in Figure 18. The diagram illustrates how, when symbols are inverted, only the first bit in the decoded bitstream differs. This results in a single-bit error that affects only the first data frame, after which the remaining data is correctly recovered.

Frame Synchronization

After Manchester and differential decoding, frame synchronization must be performed to confirm that the received data adheres to the bitstream formatting defined by the RBDS standard [12]. This synchronization process ensures that the data is correctly aligned, allowing for extraction and decoding of the packets of data that are continuously broadcast through the RDS sub-channel. Figure 19 provides a high-level overview of the frame synchronization process.

In RDS, any broadcast digital information is embedded within a *block* of data containing 26 bits: 16 information bits and 10 checkword bits. Note that within the context of this SDR project, the checkword bits are used **only** for synchronization purposes, and no error correction is required. These checkwords play a central role in enabling the receiver to maintain synchronization by ensuring that data blocks are correctly aligned with the expected frame structure.

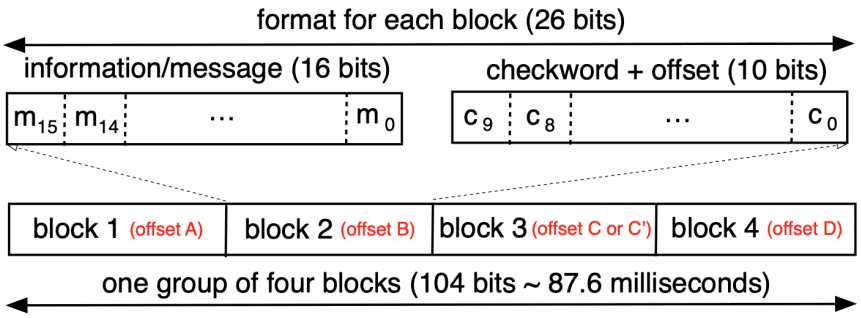


Figure 19: Groups and blocks.

Blocks are identified by unique **offset words** - A, B, C (or C'), and D - which are added to the checkword at the transmitter. Four blocks are grouped together to form a complete **group** of data, which is repeatedly broadcast. Each group carries various types of RDS information, such as program service (PS) names, radio text (RT), and metadata.

At the receiver, the 26 bits from each block are processed to compute a **syndrome**, a value that verifies whether the block is correctly synchronized. If the syndrome is valid, the 16 information bits are extracted and passed on for further processing. This process ensures that only correctly synchronized data is used, avoiding data corruption in subsequent processing stages.

Frame synchronization is required each time the receiver tunes to a new FM station. During the synchronization process, whenever a new bit is received from the differential decoder (recall that the bitstream has a low data rate of 1,187.5 bits/sec), a new **syndrome** must be calculated using the most recent 26 bits. This can be accomplished by moving a sliding window of size 26 across a larger segment of the bitstream, as illustrated in Figure 20, where the message bits are assumed to be all zeros. This brute-force search continues until a sequence of syndromes matching valid offset words is detected 26 bits apart.

Once synchronization is established, subsequent block syndromes are computed only after another complete block of 26 bits has been assembled. To maintain synchronization, the receiver must verify that blocks within a group follow the expected sequence. Specifically, block 1 (with offset word A) must be followed by block 2 (offset B), which is then followed by block 3 (offset C or C'), and finally block 4 (offset D). The next group starts again with block 1 (offset A), and this sequence repeats continuously.

Loss of synchronization can be detected if the sequence order of blocks within a group is violated. For example, if block 2 is not followed by block 3, or if block 4 is missing, it indicates potential synchronization failure. Such failures often occur due to a weakened signal or increased noise. When the error rate in block order becomes too high, the receiver may need to reinitiate a brute-force search for frame synchronization.

For the theory regarding the modified shortened cyclic code used for frame synchronization in RBDS, please refer to Appendix B of the official standard document [12]. For additional background on error detection and correction, as well as cyclic redundancy check (CRC) codes, a good starting point is the reference document [14]. Note that, for self-containment purposes, this document provides the parity check matrix used for syndrome computation, along with the error-free syndromes corresponding to each valid block. These details can be found in the Appendix of this document.

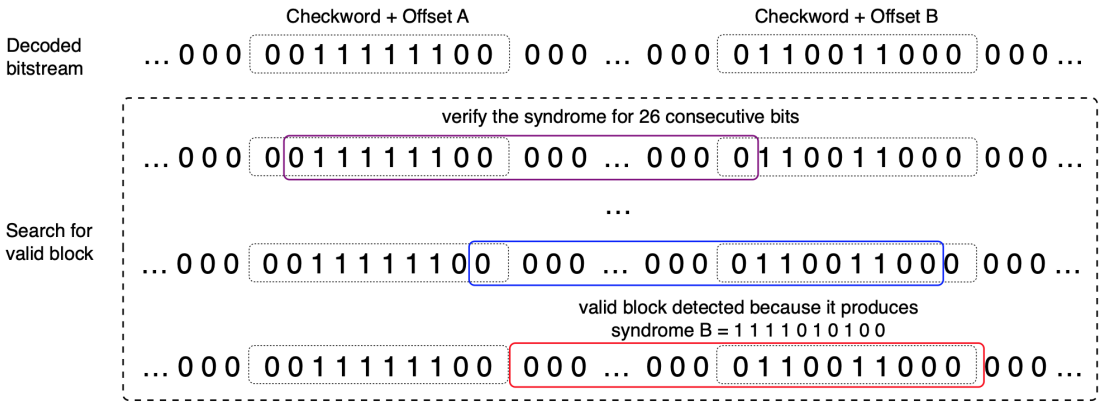


Figure 20: Block search during frame synchronization.

RDS Application Layer

The final step of RDS Data Processing is extracting the message from each block's information bits. The details can be found in the RBDS standard document [12]; an example of the message format from the standard is provided in Figure 21.

In RDS, there are 16 different group types, each identified by the group type code in block 2. These group types are used to organize different types of information broadcast by FM stations. For the purposes of this project, it is sufficient to extract the following key elements: **program identification (PI) code**, which is found in block 1. This 16-bit code uniquely identifies the FM station and its broadcast region. More details on how FM broadcasters prepare these PI codes can be found in [15]; **program type (PTY)**, which is found in block 2. This code categorizes the type of programming (e.g., News or Sports) and is standardized in Annex F of the RBDS document [12]; **program service (PS) name**, which is broadcast using group type 0A. This typically contains an 8-character name representing the station's branding or call sign and is detailed in section 3.1.5.1 of the standard [12]; **radio text (RT)**, which is transmitted using group type 2A. This text typically provides dynamic information such as song titles or artist names. The message format for radio text is explained in section 3.1.5.3 of the RBDS standard.

Although not explicitly required for this project, those interested can explore the clock, time, and date information, which is transmitted using group type 4A. Further details on clock/time synchronization are available in section 3.1.5.6 of the RBDS document [12].

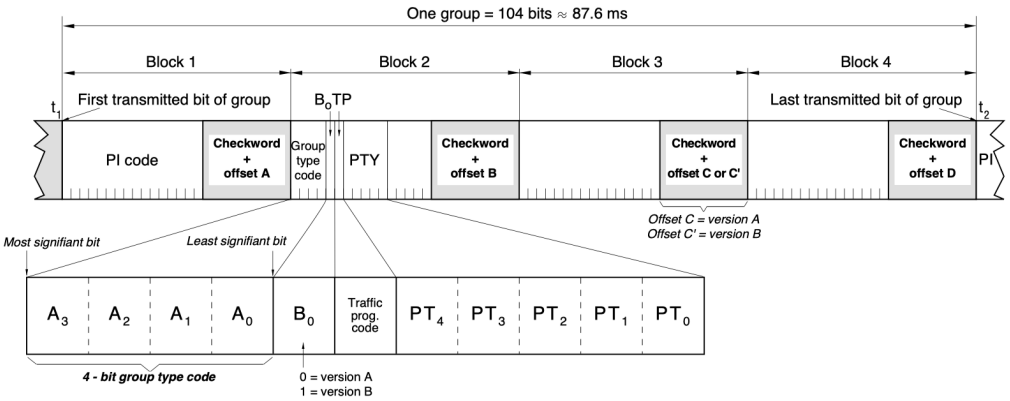


Figure 21: RDS message format (from [12]).

Multi-threading

Due to the nature of this SDR application, many degrees of freedom can be leveraged to facilitate real-time execution on an embedded computing platform [3]. While some optimizations can be explored at the signal-flow graph level - e.g., integrating decimation and filtering - others are in the implementation space. While suggestions will be provided during classes and project meetings, one particular topic is worth planning for in advance: threading.

As both the codebase and the compute load increase, partitioning the application into several independently maintainable threads becomes necessary. Figure 22 shows the suggested thread partitioning. The RF front-end operates at the highest sample rate, generating FM-demodulated data for all the other processing paths. Therefore, it can be a standalone thread that fills a queue of data blocks, which are processed independently by the other threads. While the mono and stereo paths do most of the processing independently, their outputs are eventually recombined; hence, these two audio paths can be integrated into a single thread. Since the output of the RDS path is independent of the audio paths, it should be implemented in its own thread.

It is also worth mentioning that the audio and RDS threads always lag behind the RF front-end thread in terms of block count; however, they can get ahead of each other arbitrarily. As long as the synchronization queue is limited in size, this divergence between the audio and RDS paths will not impact the application in a tangible way.

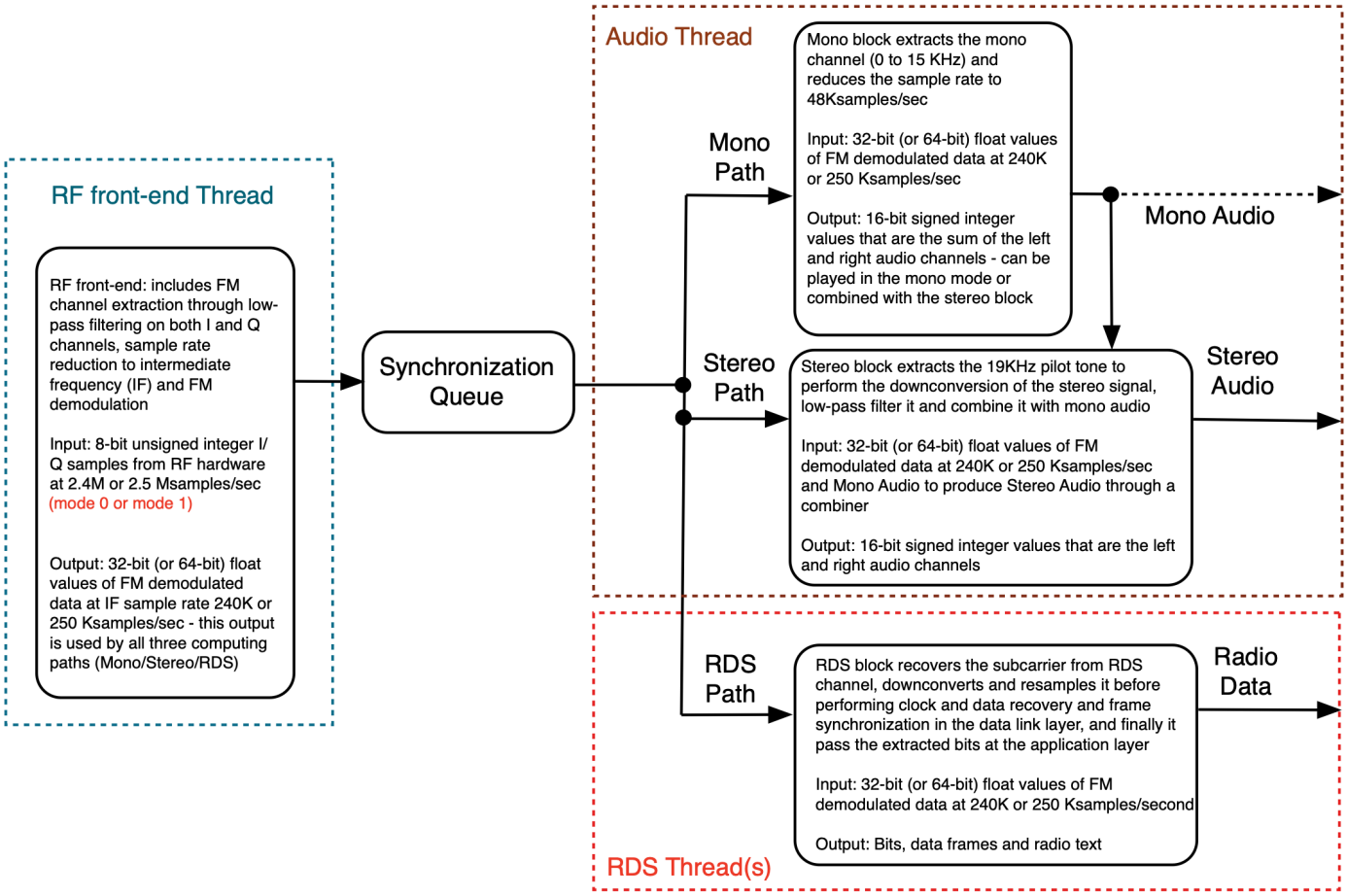


Figure 22: Thread partitioning to leverage the parallelism offered by a multi-core processor.

References

[1] NESDR SMArt Series. https://www.nooelec.com/store/sdr/sdr-receivers/smart.html. Accessed: 2025-02.

[2] Osmocom RTL SDR. https://osmocom.org/projects/rtl-sdr/wiki/Rtl-sdr. Accessed: 2025-02.

[3] Raspberry Pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Accessed: 2025-02.

[4] FM Broadcasting. https://en.wikipedia.org/wiki/FM_broadcasting. Accessed: 2025-02.

[5] Radio Data System. https://en.wikipedia.org/wiki/Radio_Data_System. Accessed: 2025-02.

[6] GNU Radio. https://wiki.gnuradio.org/. Accessed: 2025-02.

[7] Gqrx SDR. https://gqrx.dk/. Accessed: 2025-02.

[8] PySDR: A Guide to SDR and DSP using Python. https://pysdr.org/. Accessed: 2025-02.

[9] Signal Processing for Communications. https://www.sp4comm.org/webversion.html. Accessed: 2025-02.

[10] FM Stereo/RDS Theory. http://rfmw.em.keysight.com/wireless/helpfiles/n7611b/Content/Main/FM_Broadcasting.htm. Accessed: 2025-02.

[11] Frequency Modulation (FM). https://www.ni.com/en-ca/innovations/white-papers/06/frequency-modulation--fm-.html. Accessed: 2025-02.

[12] Specification of the Radio Broadcast Data System (RBDS). https://avenue.cllmcmaster.ca/d2l/le/content/633404/viewContent/4993194/View?ou=633404. Accessed: 2025-02.

[13] Square-Root Raised Cosine Signals (SRRC). https://gssc.esa.int/navipedia/index.php/Square-Root_Raised_Cosine_Signals_(SRRC). Accessed: 2025-02.

[14] Error Detection and Correction. https://www.mathworks.com/help/comm/ug/error-detection-and-correction.html. Accessed: 2025-02.

[15] The Technical Advisory Committee on Broadcasting (B-TAC) - Program Information Codes for Radio Broadcasting Stations. https://www.ic.gc.ca/eic/site/smt-gst.nsf/eng/h_sf08741.html. Accessed: 2025-02.

Enjoy!

Appendix

The full details of the theory and implementation of the modified shortened cyclic code used in RBDS are provided in Appendix B of the official standard document [12]. For self-containment purposes, the parity check matrix from section B.2.1 of [12], which is used for frame synchronization in the receiver, is reproduced below.

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

A 26-bit block containing the information word (16 bits) and its checkword (10 bits), produced at the transmitter and combined with an offset word (A, B, C, C', or D) depending on the block type, will pre-multiply the parity check matrix in a Galois field (GF(2)). In GF(2), pointwise multiplication corresponds to a logic AND operation between two bits, while pointwise addition corresponds to an exclusive OR (XOR) operation. This vector/matrix multiplication in GF(2) produces a 10-bit **syndrome**, which confirms the block type. The table below lists the error-free syndromes for the four block types.

Offset type	Offset word (added at transmitter)	Syndrome word (generated at receiver)
A	0011111100	1111011000
B	0110011000	1111010100
C	0101101000	1001011100
C'	1101010000	1111001100
D	0110110100	1001011000