COE3DY4 Project
Real-time SDR for mono/stereo FM and RDS

Objective

The project's main objective is to navigate a complex (industry-level) specification and understand (and rise to) the challenges that must be addressed for a real-time implementation of a computing system that operates in a form factor-constrained environment.

Preparation

- Revise the material from labs 1 to 3

Project Overview

The 3DY4 project leverages front-end radio-frequency (RF) hardware, like RF dongles [1] based on the Realtek RTL2832U chipset [2] and single-board computers, like Raspberry Pi 4 [3]. The goal 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 sent through FM broadcast using the radio data system (RDS) protocol [5] [1]. Due to the affordability of RF hardware, a broad number of open-source SDR projects have emerged over the past decade or so, like GNU Radio [8] or Gqrx [9], only to name a couple. The goal of the 3DY4 project is not to duplicate this type of open-source initiative. Rather, the real-time SDR project in this course is used primarily to consolidate the knowledge acquired so far and learn how to navigate from the first principles to the complex interrelationships between seemingly disparate yet practically related topics from electrical and computer engineering.

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

[1] Since RDS was initially developed in Europe [6], the official name in North America is Radio Broadcast Data System (RBDS) [7]. Except for some digital codes specific to geographic locations, the two standards are virtually identical, and therefore, the terms RDS and RBDS will be used interchangeably.

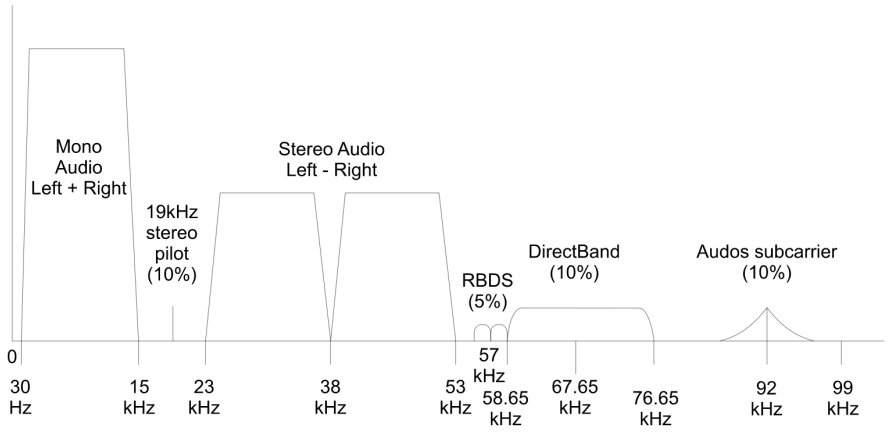


Figure 1: Spectrum of an FM channel [4]

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 needed to broadcast stereo audio when the receivers were built with analog technologies. While the RDS sub-channel is expected to be centred 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; hence the carrier recovery and downconversion for RDS will follow an approach that is different from the carrier recovery and downconversion for stereo audio; consequently, it is worth noting that the approach used for RDS is resemblant to the approach followed by many modern digital communication standards.

While FM stations drive the data that feeds the SDR system implemented in this project, all the essential FM-related information relevant to the project is provided between the last lab, forthcoming class discussions, and this project document. For more details on FM technology, you are referred to academic textbooks or industry white papers, such as [10, 11, 12]. As a final note, because some terms can be implied from the context, we will refer to sub-channels of an FM channel also 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 concerned with the acquisition of the RF signal through an antenna and translating 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 (through a process of analog downconversion via 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 that must be accounted for 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, followed by the next I/Q pair and so on.
- The RF front-end block from the SDR system extracts the FM channel through low-pass filtering followed by decimation to intermediate frequency (IF) and FM demodulation. It is conceptually the same as the RF front-end for the last lab; however, for the project, there will be **four** modes of operation.

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

and their unique settings will be uploaded into your group's repo in GitHub after it has been created. Hence, mode 1 from this document is "fictive", and it does not apply to any project for any group. The descriptions concerning 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 to highlight the differences of the custom modes 1, 2 and 3 from mode 0 and to articulate the conceptual challenges faced during resampling.

By default, your SDR software should operate in mode 0, and only the mono path should be exercised. If command line arguments are provided, the first one is the mode (0, 1, 2 or 3), and the second one is the most advanced path that is exercised (`m` for mono, `s` for stereo and `r` for RDS); note that if `r` path is exercised, it is implied that both mono and stereo paths will be exercised as well. For further details, see your group's project repo in GitHub for custom settings for the sample rates for each mode.

- The mono block extracts the mono audio channel (0 to 15 KHz), and it reduces the sample rate to 48 KSamples/sec. The input is FM demodulated data that should be represented in floating point format with an IF sample rate at either 240 KSamples/sec or 250 Ksamples/sec, depending on the mode of operation. The output should be in 16-bit signed integer format that can be transferred to an audio player. Note that if only mono audio is implemented, the 16-bit values are the sum of the left and right audio channels. Since the intermediate data is in floating-point format, it is up to the SDR software developer to choose an internal scale factor for the output audio data to adequately use the dynamic range offered by the 16-bit signed integer format.
- The stereo block extracts the 19KHz pilot tone through band-pass filtering to perform the downconversion of the stereo channel, which is subsequently filtered and combined with the mono audio data to produce the left and right audio channels. This can be achieved by combining the data from the mono and stereo channels because the mono channel contains the sum of the left and right audio channels. In contrast, the stereo channel contains the difference between the left and right audio channels. The input and output formats are the same as for the mono path.
- The RDS block first recovers the subcarrier from the RDS channel, downconverts it through a digital communication approach, and resamples it before performing clock and data recovery. Once the bitstream is generated, the frame synchronization is done in the data link layer before the extracted information words are passed to the RDS application layer. The input to this block is the same as for the mono and stereo paths. However, the output is now in terms of bits and words of radio data.

The remainder of this document will provide the motivation and objectives of each processing block in the SDR receiver for FM mono/stereo/RDS and its most critical details. *The more specific details, such as the design methodology, including implementation choices, trade-offs, ..., will be discussed during classes and 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, below is a summary of the most important takeaways from the lab work:

[2] For general details concerning Raspberry Pi, RTL-SDR, FM, RDS, ... that are not specific to this course you should refer to third-party literature.

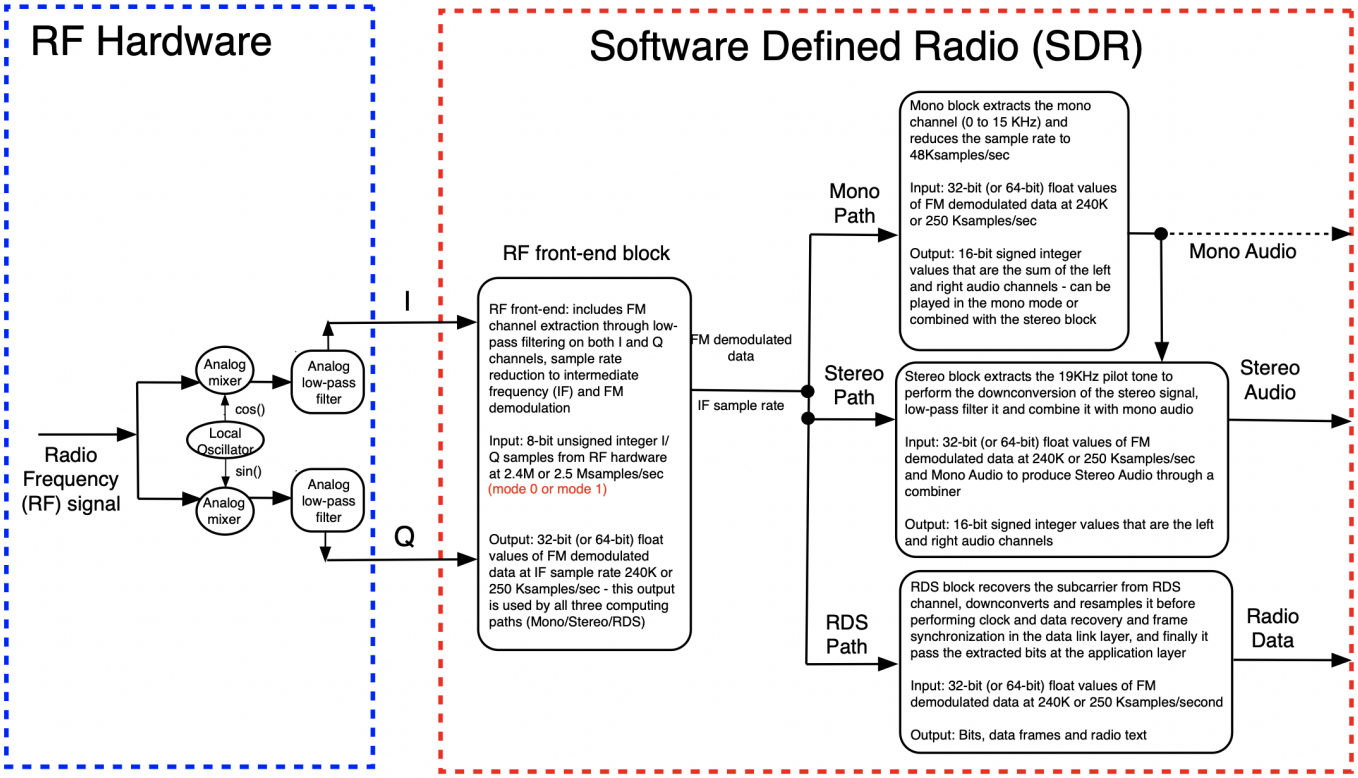


Figure 2: Project overview

- Filters are core building blocks in any digital signal processing (DSP) application, and they can be implemented using the convolution of the filter's impulse response with the input sequence (discrete time sequence of digital samples).
- The impulse response coefficients can be derived using the *sinc* function by relying on the Fourier transform relationship between the rectangular window, used to gate the pass band in the frequency domain, and the *sinc* function in the time domain.
- In practice, idealized (or brick-wall) filters cannot be implemented; hence, finite-impulse response (FIR) filters that guarantee linear phase are commonly used; for a given sample rate, the number of FIR taps (input sample delay/coefficient pairs) will have an impact on both the stop band attenuation and the width of the transition band.
- The filter coefficients depend on the number of FIR taps, the cutoff frequency (or frequencies for pass-band) **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. The input sequence is processed in blocks because of the need to avoid both an excessive latency for data acquisition and large memory requirements. This adds to the implementation challenge. Nonetheless, this is a common issue that needs to be tackled with the same mindset across a broad spectrum 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 for

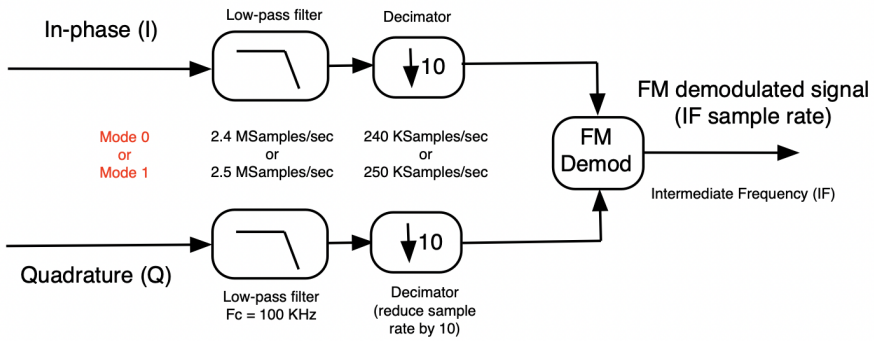


Figure 3: RF front-end processing (in software)

the same signal-flow graph with the same parameters can vary substantially. The signal-flow graphs should be modelled first in a high-level language. e.g., `Python` using its scientific libraries `SciPy/NumPy`, and only when the functional requirements have been met they are to be re-implemented in another language, e.g., `C/C++`, to facilitate real-time execution.

- The methods used to check the correctness of implementation while bringing up a complex system can vary depending on the tasks to be performed and issues at hand; for example, when dealing with the interface to the physical world, there is no reference data in terms of exact values to be matched, and therefore one has to rely on visual inspection of power spectra to pass judgment if the code has been implemented correctly; conversely, when re-factoring the `C++` code to make the implementation feasible for real-time execution, i.e., converting convolution from single-pass to block processing, bit equivalence is expected to guarantee that no implementation artifacts have been introduced during refactoring.

As a final note, for the machine code compiled from `C++` to work in real-time, special considerations for optimizing the source code will be needed, and some of the generic ones will be discussed during lectures and tutorials.

RF Front-End Processing

Figure 3 illustrates the signal-flow graph for the RF front-end block. Take note of 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 because now there are two modes (reminder: check note from page 2 concerning modes of operation) of operation for the SDR system: in mode 0 the input sample rate, i.e., RF sample rate, is 2.4 MSamples/sec; in mode 1 the RF sample rate is 2.5 Msamples/sec. While this is inconsequential to the modelling of the RF front-end block (because the decimation scale factor is kept at 10 regardless of the mode of operation), it does have a ripple effect on the re-sampling in the other processing blocks. Nevertheless, 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 kept **after** decimation.
- The I/Q samples that will be fed by the RTL-SDR driver [2] to your software will be represented as 8-bit unsigned integers (`unsigned char` in `C++`). If you are 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 -1 to +1 range of real numbers. Note that the code from the labs assumes the

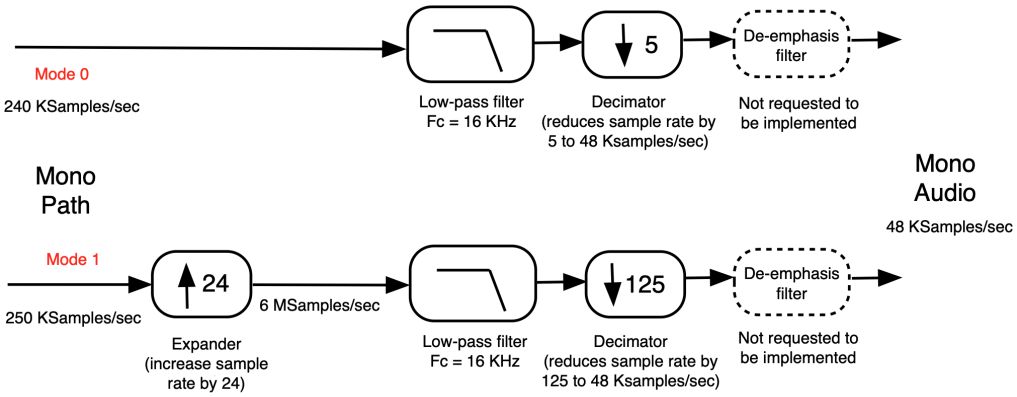


Figure 4: Mono processing

`float` data type for real numbers in `C++`, representing real values on 32-bits. While both the dynamic range and the precision offered by the `float` data type is normally sufficient, depending on your design approach, you might consider using the `double` data type, which represents real numbers on 64-bits. In the context of this SDR application for this project, the advantage of `double` over `float` is extra precision, which may or may not be needed because it depends on your unique parameters in the signal-flow graph and your project-specific implementation choices; however, this would come at the expense of doubling the memory requirements for all the blocks represented as real numbers and, depending on your unique parameters and project-specific choices, there is a possibility of a small performance overhead, which, in most cases, is expected to be tolerable.

- It is critical to emphasize that the only constraints on the data types are the 8-bit unsigned integer values for the inputs to your SDR software (I/Q samples) and 16-bit signed integer values for the outputs (audio samples) to be fed to an audio player. Note that the output of the RDS path is also clearly defined. However, the internal data representation is at the discretion of each project group and the above discussion concerning normalization to the -1 to +1 real range or `float` vs `double` is *a suggestion rather than a requirement.*

Mono Processing

Figure 4 shows the signal-flow graph for mono processing. Take note of the following points:

- Because the background noise in the very high frequency (VHF) range of electromagnetic waves can affect frequencies toward the higher range of the audio spectrum, the FM broadcasting stations pre-emphasize, i.e., 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 the user perception perspective, and the available time during the project phase is deemed to be better utilized to develop the understanding for other important implementation techniques, like multi-threaded software, in greater depth.
- It is critical to notice that depending on the mode of operation, there will be two different types of digital filters and sample rate conversions. 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 is

the same signal-flow graph from the last lab for the mono channel extraction, filtering and downsampling to the audio sample rate.

- In mode 1, 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 needed. Assuming an integer upscale factor of U for upsampling, the first step is implementing an expander $\uparrow$ where $U-1$ zeros are padded between any two input samples. This will produce a sequence at the output of the expander whose sample rate is $U \times IF$, which is U times larger than the sample rate at the input of the resampler (since IF is 250 KSamples/sec in mode 1, the sample rate at the output of the resampler is 25 $\times$ 240 KSamples/sec = 6 MSamples/sec, as shown in Figure 4). The decimator $\downarrow$ uses an integer downscale factor of D and it will remove every $D-1$ samples from the output sequence, which reduces the sample rate from $U \times IF$ to $\frac{U}{D} \times IF$.

  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*, the two low-pass filters are collapsed into a single filter **in between** the expander $\uparrow$ and the decimator $\downarrow$ with a maximum cutoff frequency equal to the minimum of $\{(\frac{U}{D} \times \frac{IF}{2}), \frac{IF}{2}\}$. Note that this cutoff frequency is relative to the signal's sample rate between the expander and the decimator, which is $U \times IF$. It is important to note that because of the increase in the sample rate caused by zero-padding, **both the impulse response's size, i.e., the number of filter taps, and gain of this filter are to be scaled up by a factor of** U. Note, however, despite the increase in the filter size and its gain, considering that in the expanded stream for each non-zero value there are $U-1$ zeros due to zero padding, the number of non-zero partial products that contribute to the strength of the output signal will still be in a reasonable range (i.e., equal to the default value for the size of the impulse response to be used for the filters when no resampling is done, e.g., 101).

- 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) between the input and output sample rates (250K and 48K respectively for mode 1). This results in U=24 for mode 1. Similarly, D=125 (ratio between the input sample rate and the gcd between the input and output sample rates). Simple math confirms that the maximum cutoff frequency for the low-pass filter between the expander $\uparrow$ and decimator $\downarrow$ for mode 1 is 24 KHz. However, to extract only the mono channel, we choose an even lower cutoff frequency at 16 KHz.

- What will make implementing this filter particularly challenging is that the sampling rate at its input is $U \times IF$, i.e., 24 $\times$ 250 KSamples/sec = 6 MSamples/sec. One has to choose a huge number of filter taps to reduce the transition band width and increase attenuation from the pass to the stop band. A straightforward implementation would be very computationally demanding, however, when leveraging some inherent properties of the input sequence to the filter, e.g., $U-1$ zeros introduced by the expander $\uparrow$ in between any two non-zero samples, or cancelling computations for the $D-1$ output samples that are removed by the decimator $\downarrow$, one can speed up the execution time substantially. This will be one of the algorithmic optimization techniques to be learned while making the real-time implementation feasible.

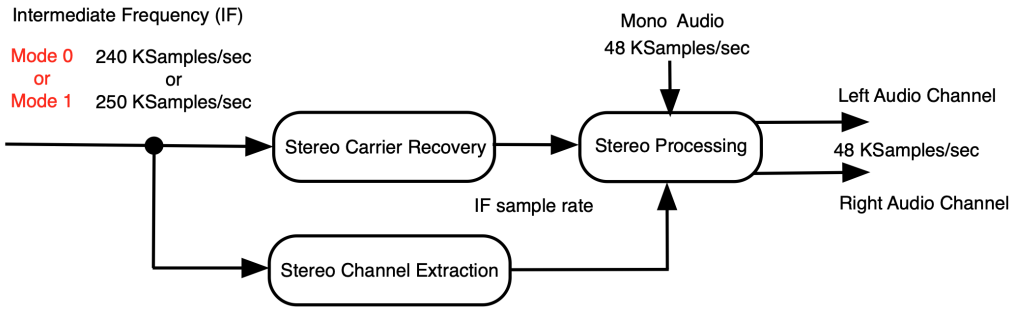


Figure 5: Stereo processing path

Stereo Processing Path

The stereo signal carries the **difference** between the left and the right audio channels, and for this reason, its strength is commonly lower than the signal from the mono channel, which carries the *sum* between the two audio channels. The stereo signal is modulated onto a 38 KHz subcarrier, which is locked to the second harmonic of the 19 KHz pilot tone that is present in each FM channel. During transmission, after upconversion the sub-carrier is suppressed, i.e., the stereo signal is modulated using Double Side Band Suppressed Carrier (DSBSC). The bandwidth of the stereo signal is 15 KHz, with the two sidebands being symmetric with respect to 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) 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 will subsequently be converted to the audio sample rate through the same line of reasoning followed in the mono-processing path. Finally, the stereo data will be combined with mono data to produce the left and right audio channels.

Stereo Channel Extraction and Carrier Recovery

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

First, it is important to clarify that DSBSC is a distinct modulation technique from FM, and the stereo signal is upconverted *within* the FM channel at 38 KHz using DSBSC (note, to avoid any confusion, all the baseband data from all the sub-channels from the FM channel are subsequently frequency modulated before hitting the airwaves from the FM broadcaster). The basic idea of DSBSC modulation is to mix, i.e., multiply, the message signal with the carrier signal as follows: $A_m cos(2\pi f_m) \times cos(2\pi f_c)$, where A_m is the amplitude of the message, f_m is its frequency and f_c is the carrier frequency (for the sake of this discussion we ignore the amplitude of the carrier and phases of the two cosines). Using trigonometric identities, the above multiplication translates to the following DSBSC modulated signal: $\frac{A_m}{2}[cos(2\pi(f_c + f_m)) + cos(2\pi(f_c - f_m))]$. In the receiver, the same type of mixing is done for demodulation purposes, i.e., the carrier frequency is multiplied by the received signal, which has been DSBSC modulated. This mixing in the receiver produces

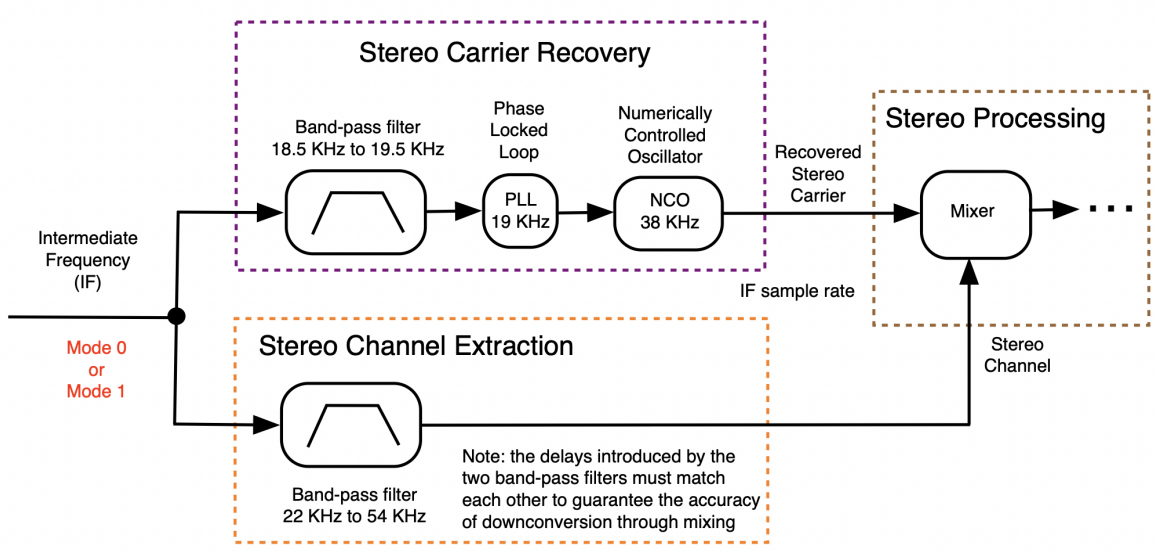


Figure 6: Stereo channel extraction and carrier recovery

$2\frac{A_m}{4}cos(2\pi f_m) + \frac{A_m}{4}[cos(2\pi(2f_c + f_m)) + cos(2\pi(2f_c - f_m))]$. It can be seen that the first term is the original message (whose amplitude is scaled down by 2), and the second term has a much higher frequency than the first term, assuming f_c is larger than f_m. Based on the above, if we low-pass filter the output of the mixer from the right-hand side of Figure 6, we will recover the original message (we can scale the gain of the low-pass filter to make up for the loss in message strength due to mixing at the transmit side).

To perform the above-described downconversion accurately, first, we have to extract the stereo channel, which can be done through a band-pass filter with its start (or beginning frequency f_b) at 22 KHz and end frequency (f_e) at 54 KHz. This is the purpose of the stereo channel extraction sub-block from the bottom of Figure 6. Of equal importance is 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*. Considering that DSBSC suppresses the carrier to save transmit power, all the FM stations broadcast a 19 KHz stereo pilot tone whose second harmonic is used for mixing at the transmitter; note that this 19 KHz pilot tone is sent as a standalone signal in between the mono and stereo sub-channels within the FM channel. As seen in the Stereo Carrier Recovery sub-block from Figure 6, we first extract the 19 KHz pilot tone through a band-pass filter; then we synchronize it using a phase-locked loop (PLL); finally, we multiply the output of the PLL by a scale factor of 2 using a numerically controlled oscillator (NCO) to produce the recovered stereo carrier for mixing.

It is important to clarify that 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 to keeping things orderly in wireless communications (or data transmission in general). Elaborating on the inner workings of a PLL is beyond the scope of this project document (note that the backbone code for a software PLL/NCO will be provided in `Python`). Nonetheless, at the intuitive level, it is worth mentioning that an essential goal of PLLs is to produce a *clean* output, e.g., filtered (and amplified if the input signal weakens), in lock to a noisy input. An inherent trade-off for PLLs is that the faster a PLL can lock, the more phase jitter will exist in the output and the worse the ability of the PLL to lock to a weak input. To

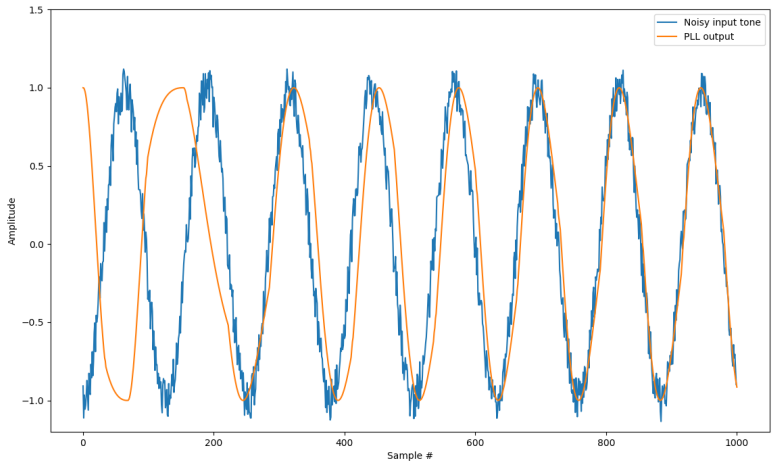


Figure 7: Phase-locked loop (PLL)

articulate the importance of PLLs in FM receivers, the example from Figure 7 illustrates how a PLL is capable of locking to the phase of a periodic yet noisy input signal and passing a "clean" copy to the NCO for frequency scaling. It is also worth noting that the waves from Figure 7 have been produced using a noisy input tone at 19 KHz oversampled at 2.4Msamples/sec; as it can be observed, the PLL can lock within less than 1 KSamples.

Both the stereo carrier recovery and stereo channel extraction rely on band-pass filtering. There are two points worth making: (i) the number of filter taps for the two band-pass filters from Figure 6 should be matched to each other to guarantee the same delay from the FM demodulated data to both inputs of the mixer; (ii) the pseudocode for deriving the impulse response coefficients follows the structure and notation from the first lab, where pseudocode was given for deriving the impulse response coefficients for a low-pass filter.

Input: Filter parameters: f_b, f_e, f_s and N_{taps} ▷ f_b is the beginning of the pass band
▷ f_e is the end of the pass band
▷ f_s is the sample rate
▷ N_{taps} is the number of filter taps

Output: Coefficients of the impulse response for a band pass filter $h(i)$ with $i = 0 \dots N_{taps} - 1$

$Norm_{center} \Leftarrow \frac{(f_e+f_b)/2}{f_s/2}$ ▷ define explicitly the normalized center frequency $Norm_{center}$
$Norm_{pass} \Leftarrow \frac{f_e-f_b}{f_s/2}$ ▷ define explicitly the normalized pass band $Norm_{pass}$

for $i \in [0, N_{taps} - 1]$ **do**
  if $i = (N_{taps} - 1)/2$ **then**
    $h(i) \Leftarrow Norm_{pass}$ ▷ avoid division by zero in *sinc* for the center tap when N_{taps} is odd
  else
    $h[i] \Leftarrow Norm_{pass} \frac{sin(\pi(Norm_{pass}/2)(i-(N_{taps}-1)/2))}{\pi(Norm_{pass}/2)(i-(N_{taps}-1)/2)}$
  end if
  $h(i) \Leftarrow h(i)cos(i\pi Norm_{center})$ ▷ apply a frequency shift by the center frequency
  $h(i) \Leftarrow h(i)sin^2(\frac{i\pi}{N_{taps}})$ ▷ apply the Hann window
end for

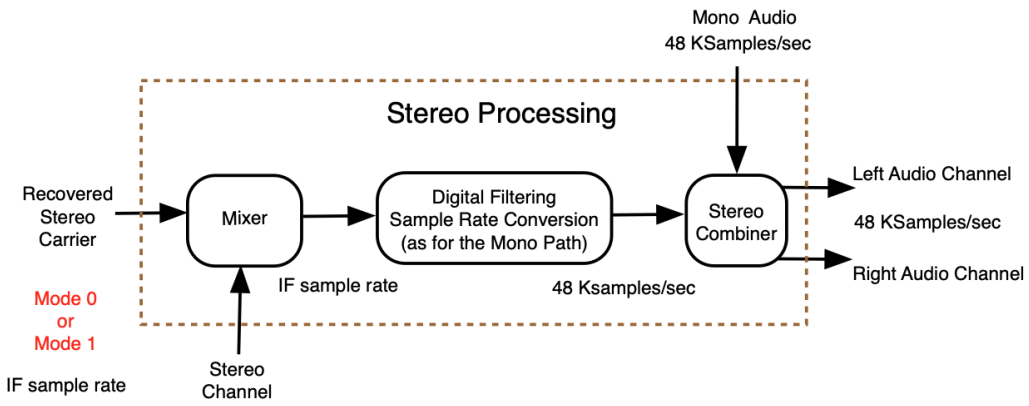


Figure 8: Stereo downconversion, filtering and combining

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 elaborated at the beginning of the previous subsection. From the computational standpoint, mixing is a low-demand task because the two input streams are multiplied sample by sample (pointwise multiplication). The output of the mixer will, however, have a significantly higher computational demand because it follows precisely the same sample 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 low-pass filter, which is "absorbed" in the digital filtering sub-block. The stereo data at the audio sample rate (48 Ksamples/sec) will be combined with the mono audio to produce the left and right audio channels that can be subsequently fed to an audio player.

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 the signal processing blocks matters only when the signal "branches" at a stem node in the signal flow graph and then the branches "reconverge." As shown in Figure 8 the samples processed by the mono path are **combined** with the samples processed on the stereo path. Considering that the source to both paths is the signal at the output of the FM demodulator from Figure 3 unless the delays on the mono and stereo paths are matched, the samples that are recombined will be out of phase. For most content that FM stations broadcast, the audio artifacts caused by this delay are unnoticeable because the strength of the signal in the stereo path is lower than the one in the mono path. Recall that the mono path holds the sum of the left and right channels, whereas the stereo path holds the difference; this explains why the strength of the signal in the stereo channel is very low whenever the content that is broadcast does not have a significant difference between the left and the right audio channels [3].

To address the above concern, an allpass filter must be introduced at the beginning of the mono path, as shown in Figure 9 and described next. A well-known property in signal theory is that for FIR filters, whose coefficients are symmetrical around the centre coefficient, the phase delay is (1/Fs)*(N-1)/2, where Fs is the sample rate, and N is the number of taps. The FIR filters in the

[3] A couple of recordings with a noticeable difference between left and right channels will be shared to ensure an adequate stress test of the stereo path.

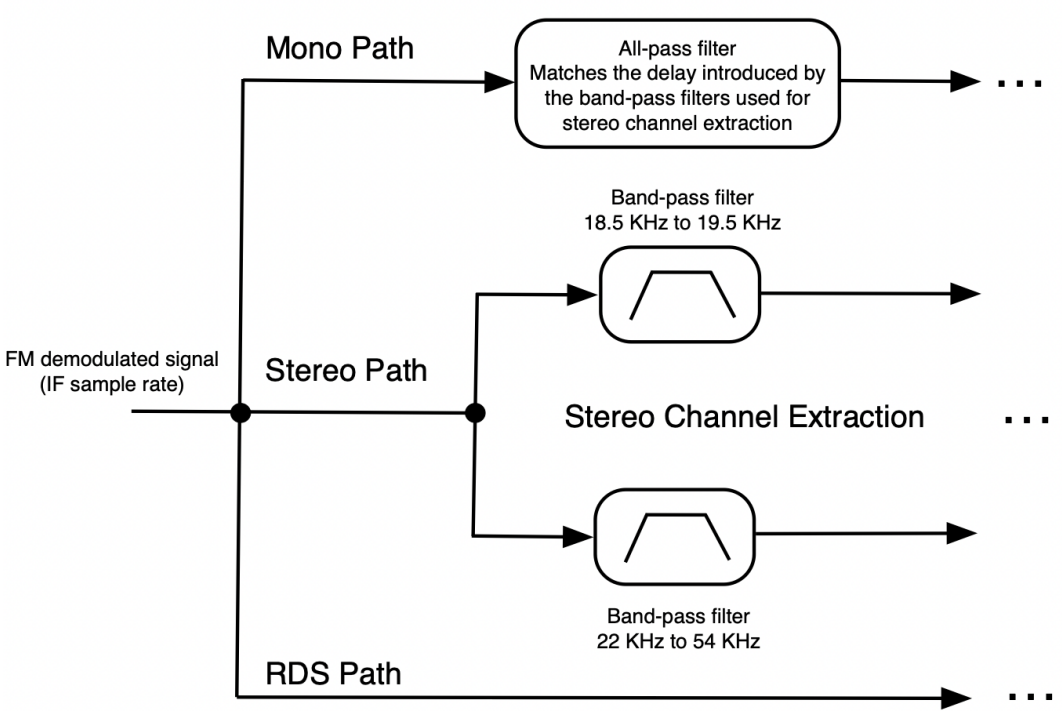


Figure 9: Introducing an all-pass filter to delay the mono path to match the sample delay caused by the bandpass filters used for stereo recovery

SDR project have this linear phase property. For example, to match delays when signal branches and reconverges, the bandpass filters used for extracting the stereo channel and the 19 KHz pilot tone must have the same number of filter taps to match the phase delay on the two branches that will reconverge at the mixing point from Figure 6. Note that instead of using a filter to generate this phase delay (with a passband from DC to the Nyquist rate), one can use a simple delay block (i.e., a shift register in hardware), where the number of samples that the signal is delayed by equals to (N-1)/2, where N is the number of filter taps in the filter whose delay the all-pass filter is supposed to match.

To deal with the unmatched delays on the mono and stereo paths that affect the samples that are recombined at the output of the stereo path, there is a need to introduce a delay block at the input of the mono path that matches the delay introduced by the two bandpass filters from the stereo path, as shown in Figure 9. For example, if each of the two bandpass filters from Figure 6 have 101 taps, then the delay to be introduced at the input of the mono path should be (101-1)/2 samples. Note that, to simplify the implementation and validation of the mono path, the introduction of the delay block on the input of the mono path should be done only **after** the implementation of the stereo path is complete, and the samples from the mono path are recombined with the samples from the stereo path to produce the left channel/right channel audio samples. Note also that one can ignore adding the same type of delay at the input of the RDS path because the audio samples and the radiotext are not sent in sync, and there is no reconvergence between the RDS path and any of the audio paths.

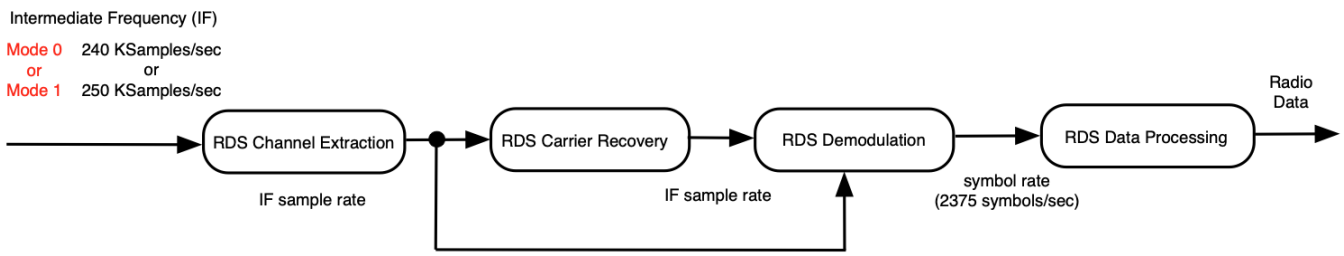


Figure 10: RDS processing path

RDS Processing Path

The RDS processing path is concerned with extracting the RDS channel (54 to 60 KHz from Figure 1), downconverting it to baseband without explicit synchronization to a pilot tone, and demodulating it for extracting the digital bitstream. Although the digital data rate is low, this processing path is particularly challenging from the computational standpoint because the RDS signal is relatively weak compared to both the mono and the 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 the stereo paths (see Figure 2). This signal is at the intermediate frequency, 240 KSamples/sec in mode 0. Similarly to the stereo processing path, extracting the RDS channel is the first step. Subsequently, the carrier is recovered, however, unlike the case for the stereo path, the recovery is **not** based on locking explicitly to a harmonic of the pilot tone. 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 multiple of the symbol rate, matched filtering and timing recovery. Finally, the extracted symbols, at a rate of 2,375 symbols/sec, are passed for RDS data processing, where they are converted to a bitstream at 1,187.5 bits/sec. In the last step, frame synchronization will be needed 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 using a band-pass filter, the RDS channel has a center frequency of 57 KHz, and the channel bandwidth is 6 KHz; all the information will be subsequently extracted after downconversion from the positive sideband where the digital data is binary-phase (bi-phase) coded at 2,375 symbols/sec. As shown in Figure 11, the band-pass filter for the 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 aims to facilitate downconversion to the 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 of a couple of reasons: not all FM stations synchronize the RDS sub-channel to the pilot tone, and it provides the opportunity to learn about a digital-centred approach for carrier recovery. The core idea of this approach lies in the fact that by squaring the message signal $A_m cos(2\pi f_m + \phi)$, one would end up with $\frac{A_m^2}{2}(1 + cos(4\pi f_m + 2\phi))$. The square signal has a frequency $2f_m$ and a phase 2ϕ, which is either 0 or 2π when using bi-phase coding of digital data.

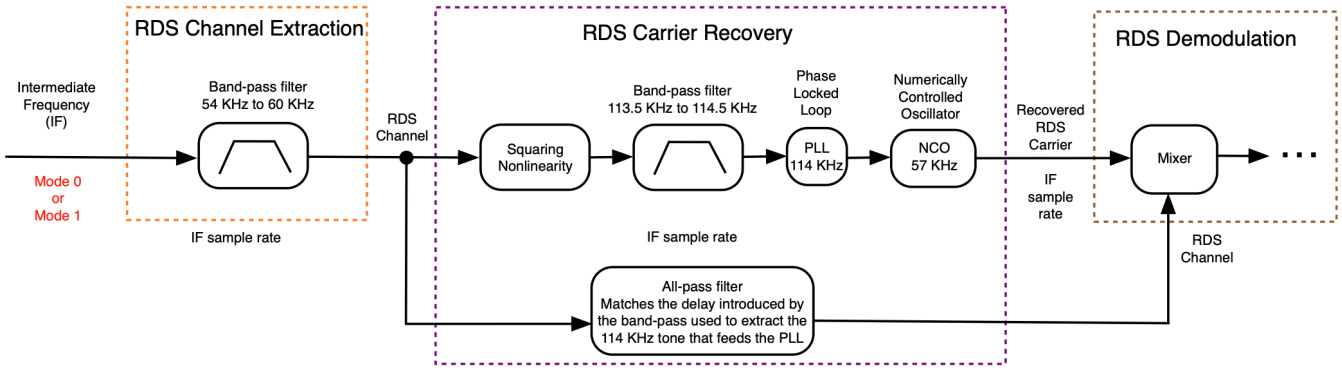


Figure 11: RDS channel extraction and carrier recovery

The squaring non-linearity from Figure 11 is a simple pointwise multiplication of each sample with itself. This result is used as an input to a narrow band-pass 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 the NCO can be integrated into a single module to produce a 57 KHz signal used for mixing the RDS channel as part of the downconversion process in the RDS demodulator. The entire RDS carrier recovery sub-block is operating at the IF sample rate. Considering that the input to the PLL is a 114 KHz tone, it is now clear why the IF sample rate in either of the two supported modes has been chosen to be above 228 KSamples/sec.

It is important to note that for the mixing to work, it is essential to delay the RDS channel by the same delay as in the band-pass filter that extracts the 114 KHz tone, which is the purpose of the allpass filter box at the bottom of Figure 11 [4]. Because this delay depends on the number of filter taps, to avoid any additional computations, it is recommended that in the tuning phase, RDS demodulation is done on both the in-phase and quadrature components of the RDS channel. This can be achieved using two outputs from the NCO (the default in-phase output and the quadrature output, which is the in-phase 57 KHz tone delayed by $\frac{\pi}{2}$). Note, while for more sophisticated digital modulation techniques in-phase/quadrature mixing is needed, in binary phase-shift keying (BPSK) modulation, i.e., bi-phase coding as labelled in the RBDS standard [7], this quadrature path can be disabled during real-time execution; it will be used only while tuning the phase of the NCO output used for mixing based on the phase shift observed in the constellation diagrams that can be visualized after timing recovery (as it will be illustrated in the following sub-section).

RDS Demodulation

Figure 12 shows the top-level diagram of the RDS demodulator. Its front end is similar to the stereo path because, after mixing, the baseband signal is extracted through a low-pass filter (note, for RDS, the cutoff frequency is set at 3 KHz). This is followed by a rational resampler that changes the IF sample rate to a sample rate that is an integer multiple of the symbol rate, which is 2,375 symbols/sec. The rational re-sampler can be implemented using the same principle as the mono path (see the discussion concerning expansion and decimation for Figure 4). Assuming an IF sample rate of 240 KSamples/sec and the samples per symbol (SPS) equal to 24, the resampler's output rate can be set to 24 x 2,375 = 57 KSamples/sec. By computing the gcd between 240K and 57K, i.e., 3,000, the expansion and decimation ratios for the resampler would be 19 and 80, respectively (with the sample rate for the anti-imaging/anti-aliasing filter equal to 240K * 19 =

[4]Based on the same line of reasoning from 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.

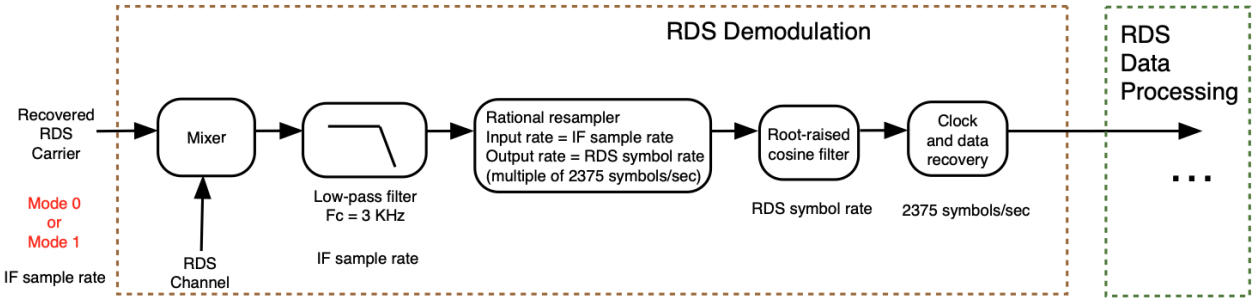


Figure 12: RDS demodulation

4.56 MSamples/sec). If SPS changed to 33, the resampler's output rate can be set to 33 x 2,375 = 78.375 KSamples/sec, and the gcd between 240K and 78.375K becomes 375, and the expansion and decimation scale factors become 209 and 640, respectively (the sample rate for the anti-imaging/anti-aliasing filter within the resampler becomes 240K * 209 = 50.016 MSamples/sec). **Reminder**: check your group's project repo for the number of samples per symbol for your group, which will define output rate of this resampler for the modes when the RDS path is enabled.

Without elaborating on the detailed theory of pulse shaping in digital communications, it is necessary to make a few clarifications to appreciate better the importance of the next sub-block in the RDS demodulator, i.e., the root-raised cosine (RRC) filter [13]. The purpose of this filter is to reduce inter-symbol interference (ISI). Digital pulses are "shaped" before transmitting to decrease the spectral bandwidth and maximize data transmission rates. Because finite bandwidth in the frequency domain causes infinite duration in the time domain, it is desirable to shape the transmitted digital data using Nyquist pulses to minimize ISI. The essential property of Nyquist pulses is that, given the known duration of digital symbols, their impulse response is nonzero only at one particular sample time and zero at all the other sampling instants. It is important to note that the root-raised cosine filter [13] is **not** a Nyquist filter when implemented on its own. However, by implementing it for spectrum shaping at **both** the transmitter and the receiver, the **combined** effect of the two filters produces a raised-cosine filter, which is a Nyquist filter.

Before transmission, RDS data is Manchester encoded, i.e., a single-bit for binary 1 is translated into **two** symbols (High followed by Low, labelled as HL); similarly, binary 0 is translated into LH. This is illustrated in Figure 13, where the shape of the **two** symbols for each of the binary values 1 and 0 is illustrated. Note that because there are two symbols per bit, the symbol rate is 2,375 symbols/sec, whereas the bitrate is half of it, i.e., 1,187.5 bits/sec.

It is important to take note that in a receiver the RRC filter is also used to add gain to the weak RDS signal that has been extracted from the FM channel (note: `Python` model for the RRC impulse response will be provided). The relative size of the input vs the output of the RRC filter is captured in Figure 14. This extra gain is needed for timing recovery (also referred to as clock and data recovery), which is a critical block in any digital communications system. Knowing the expected shape of the transmitted symbols is necessary but not sufficient to recover the binary data. One has to decide when to sample **each** received symbol. Figure 14a shows two choices for sampling times (assume 24 samples per symbol); clearly, the sampling times indicated by the red vertical lines are a better choice than the sampling times indicated by the green vertical lines. The main challenge lies in devising an algorithm to automatically adjust the sampling time.

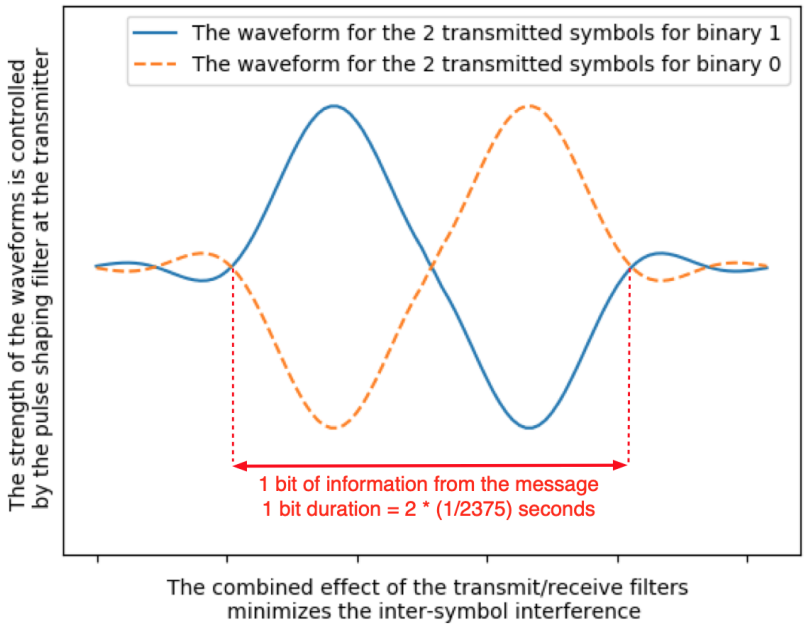
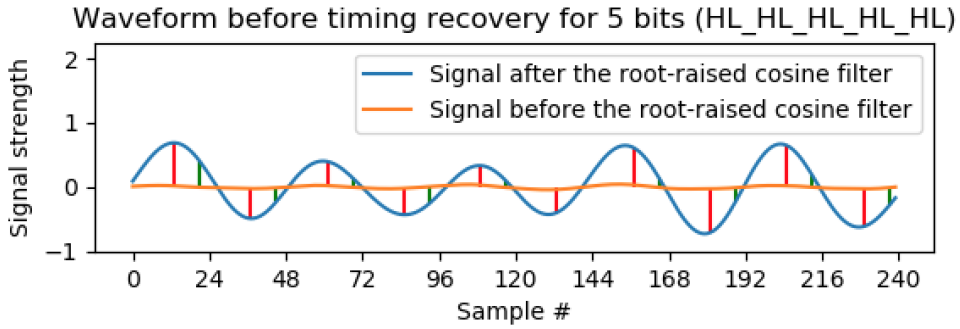
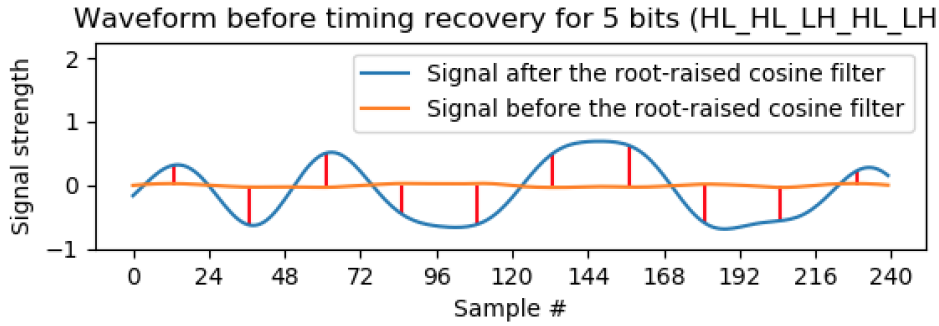


Figure 13: From bits to waveforms



(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

There are a few important points to be made. First, only the in-phase signal (one obtained by mixing the RDS channel with the in-phase component of the NCO) is shown in Figures 14a and 14b. This explains, in part, the variation in the intensity of the pulses because even for a well-tuned PLL/NCO, part of the signal energy will appear in the quadrature component, as it will be observed in the constellation diagrams from Figures 15 and 16. Note, however, that after the PLL/NCO has been tuned to match the filter delay used for carrier recovery for BPSK data **only**, the in-phase signal is sampled. It can also be observed in Figure 14b how for BPSK modulated data a logic 0 (symbols LH) is a phase-shifted version (by π radians) of logic 1 (symbols HL). Also, for real RDS data, symbols can be incorrectly sampled, leading to bit errors that are eventually identified (and possibly corrected) later in the processing flow.

Another point worth making for the RDS demodulator is the importance of monitoring signal constellations during the modelling/tuning phase, as illustrated in Figures 15 and 16. Constellation diagrams visually represent the demodulated symbols (after timing recovery) as a scatter plot in the complex plane that contains **both** the in-phase and quadrature components. As mentioned earlier, processing both the in-phase and quadrature components is needed for more advanced digital modulation techniques (e.g., QPSK, QAM, etc.). For BPSK, only the in-phase component carries useful information; nonetheless, during the tuning phase, it is important to understand how to adjust the PLL/NCO to transfer most of the energy from the RDS channel to the in-phase component after downconversion. An illustrative example of a phase shift that needs to be adjusted is shown on the right-hand side of Figure 15.

As a final point, the constellation diagrams from Figure 15 are idealized, i.e., they have been artificially created with a clear separation between all the sampled H and L symbols. The constellation diagrams look closer to the ones illustrated in Figure 16 for real RDS data captured from the airwaves. If the signal was corrupted, i.e., a weak signal due to channel impairments, it is unlikely that any meaningful data can be extracted from the sampled symbols as shown on the left-hand side of Figure 16. The same type of constellation diagrams can be observed for an inadequately tuned receiver (e.g., incorrect implementation of the PLL or the down-conversion or timing recovery...). Nevertheless, the bits can be extracted for the correct implementation, and a reasonably strong yet realistic, demodulated signal is shown on the right-hand side of Figure 16, and data frames can be identified, as explained in the next section.

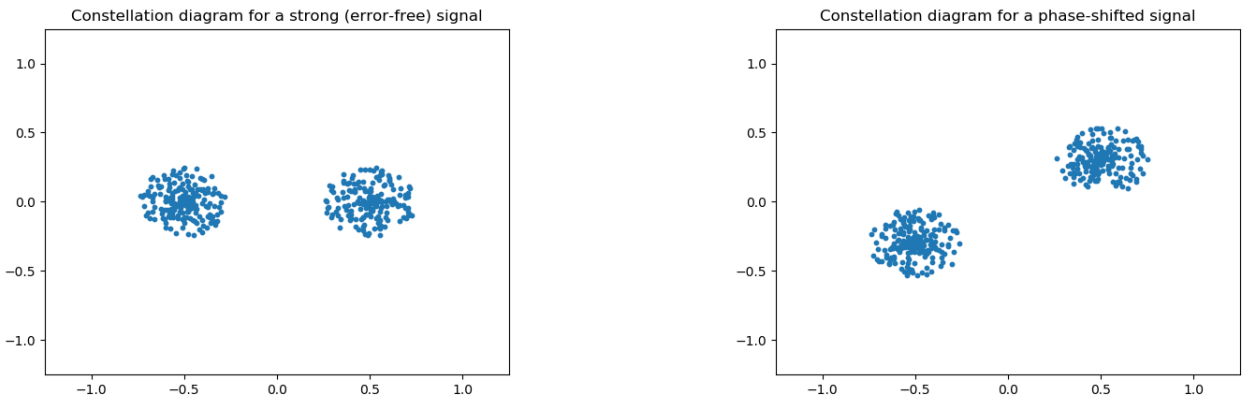


Figure 15: Constellation diagrams for idealized signals

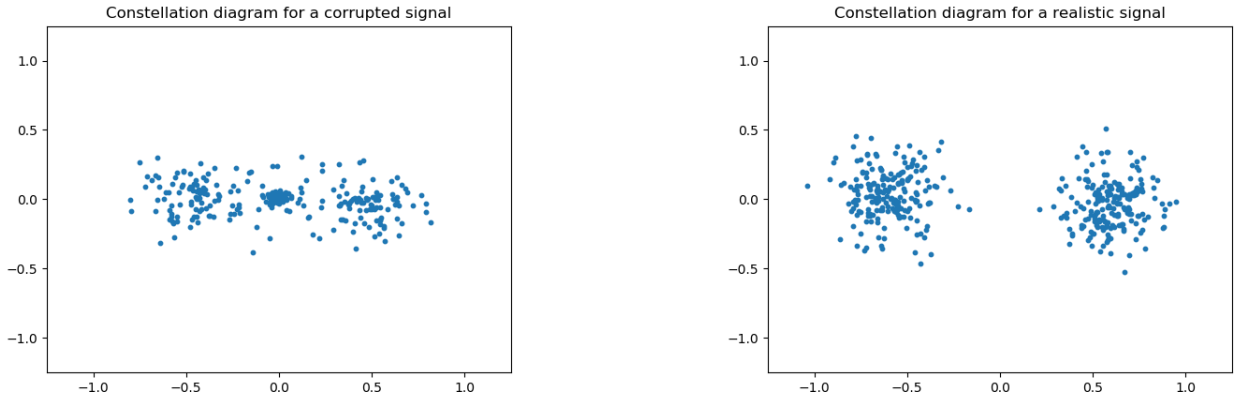


Figure 16: Constellation diagrams for real bi-phase coded signals extracted from the RDS sub-channel broadcast by an FM station

RDS Data Processing

Once the waveforms have been translated into symbols, the last step of the RDS processing path is concerned with converting symbols to bits and identifying data frames within bitstreams (called groups, and blocks within groups, in RDS) before these "framed bits" are parsed and the text messages are displayed. The application layer (referred to also as presentation/session layers in the RBDS standard [7]) is concerned with digital message formatting.

The first step within the RDS Data Processing block shown in Figure 17 is to convert the stream of symbols obtained by the clock and data recovery sub-block into a bitstream. Note, because the logic bits are Manchester encoded before being pulse-shaped at the transmitter (i.e., logic 1 is HL and logic 0 is LH as illustrated earlier using Figure 13), the symbol rate is 2,375 symbols/sec, whereas the extracted bitstream will have a bitrate equal to 1187.5 bits/sec. It is important to note that at the transmitter, the bitstream is *differentially encoded* before being converted to symbols - differential coding performs exclusive OR of the current bit with the previous one. Consequently, it must be **differentially decoded** at the receiver. The motivation for differential coding stems from the RDS receiver using a local oscillator for carrier synchronization (see the discussion concerning PLL/NCO). Since the oscillator within the PLL locks to the extracted 114 KHz tone, i.e., double the frequency of the RDS sub-carrier, the output of the NCO can have two possible phase shifts to the RDS channel data: either 0 or π radians; the differential coding prevents inversion of waves used for mixing from affecting the binary data. A detailed example is shown in Figure 18, where it can be seen that only the first bit differs if the symbols are inverted (note, this is the type of single-bit error that affects only the first data frame).

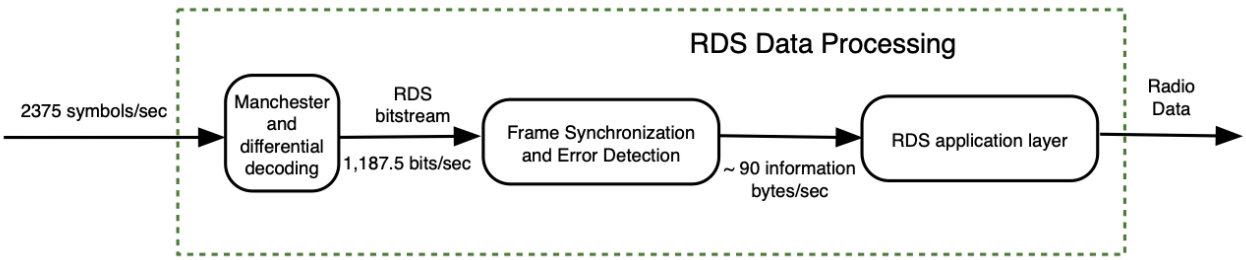


Figure 17: RDS data processing

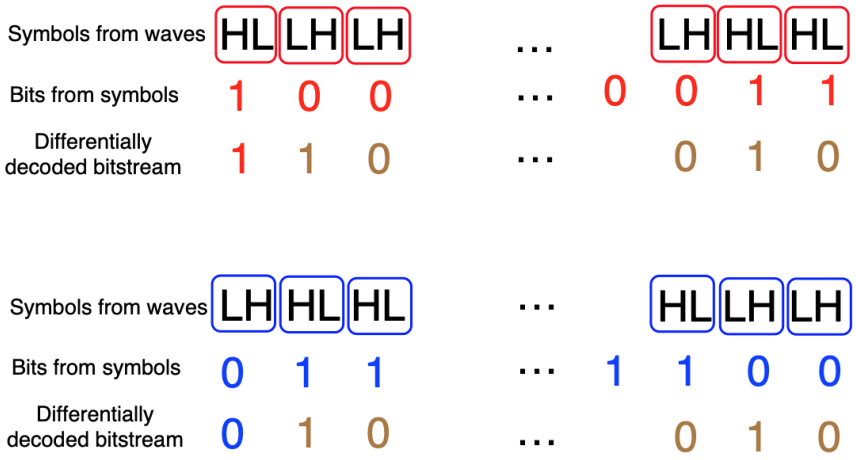


Figure 18: Differential decoding

Subsequently, frame synchronization must be done, which confirms that the received data is consistent with the bitstream formatting defined by the RBDS standard [7]. Figure 19 shows the high-level overview for frame synchronization. Any digital information that is broadcast is embedded into a *block* of data that contains 26 bits: 16 information bits and 10 checkword bits; note, within the context of this SDR project, the checkword bits are concerned **only** with synchronization and no error handling needs to be performed. Sequences of blocks, identified by unique offset words (A, B, C or C', and D) that are added to the checkword at the transmitter, are sent repeatedly in *groups* of four blocks. At the receiver, the 26 bits from a block are used to compute a **syndrome**, which, if valid, the information bits are extracted for further processing.

Frame synchronization is needed each time when tuning to an FM station. During frame synchronization, when a new bit has been received from the differential decoder (recall the bitstream has a low data rate of 1,178.5 bits/sec), a new syndrome must be calculated using the most recent 26 bits; note, the same can be achieved by moving a sliding window of size 26 across a larger volume of data from the bitstream (as shown in Figure 20, where the message bits are assumed to be all zeros). This brute-force check needs to be performed until a sequence of syndromes corresponding to valid offset words is found 26 bits apart. Subsequently, block syndromes are computed only after another block of 26 bits has been assembled. Detecting the loss of synchronization can be done if the consecutive blocks within a group do not follow the expected order, i.e., block 1 (with offset A) must be followed by block 2 (with offset B), which must be followed by block 3 (with

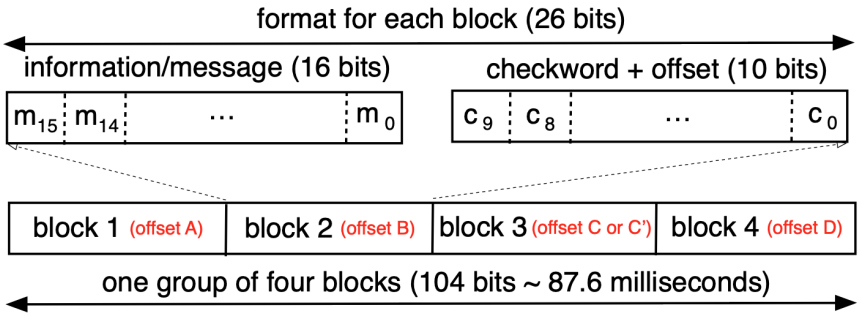


Figure 19: Groups and blocks

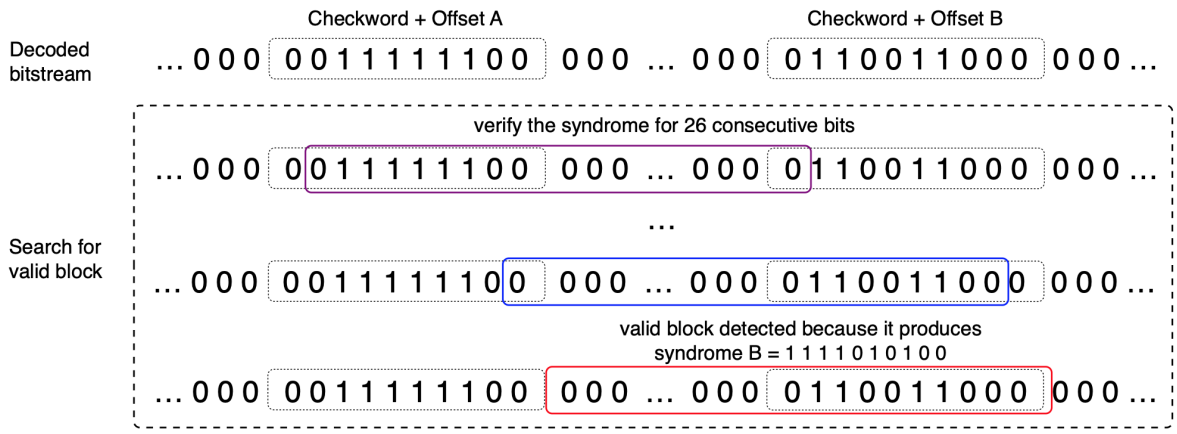


Figure 20: Block search during frame synchronization

offset C or C'), which must be followed by block 4 (with offset D), which must be followed by block 1 from the next group (with offset A), and so on. If the error rate for the sequence order increases due to a weakened signal another brute-force search for frame synchronization will be needed.

For the theory for the modified shortened cyclic code used for frame synchronization in RBDS please refer to Appendix B of the official standard document [7]. A good starting point for more background on error detection and correction and cyclic redundancy check codes is [15]. Note also, for self-containment purposes, the parity check matrix used for syndrome computation and the error-free syndromes for each valid block are provided in the Appendix of this document.

The final step of RDS Data Processing is extracting the message from each block's information bits. The details are in [7]; an example for the message format from [7] is provided in Figure 21. There are 16 different group types (the group type is identified in block 2) and in this project it is sufficient to extract: program identification code from block 1 (check [14] for how FM broadcasters prepare 16-bit codes), program type from block 2 (check Annex F from [7]), the program service name from group type 0A (section 3.1.5.1 from RBDS standard [7]), and radio text from group type 2A (section 3.1.5.3). Although not explicitly requested, the curious minds can search for the clock/time/date from group 4A (section 3.1.5.6), assuming the FM station is broadcasting it.

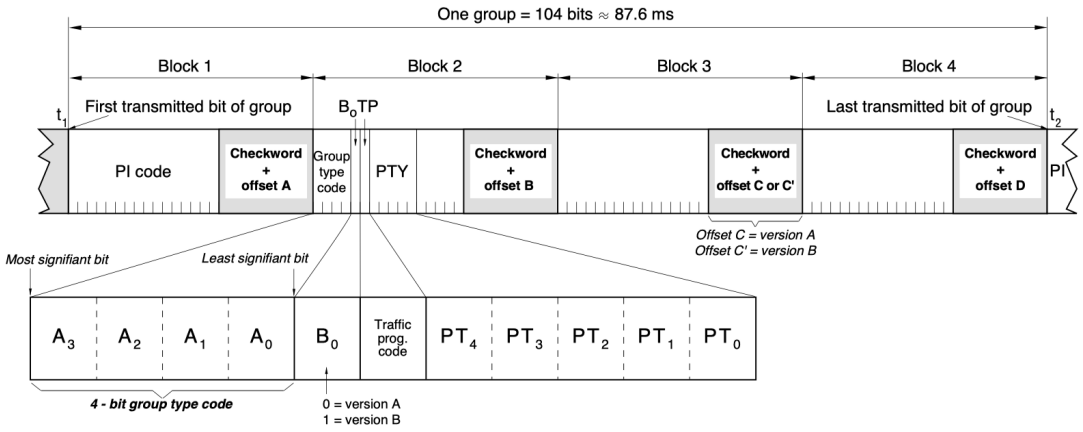


Figure 21: RDS message format (from [7])

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., changing the filter parameters to trade-off runtime vs signal strength or 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 the FM demodulated data for all the other processing paths. Therefore, it can be a standalone thread that fills in a queue of data blocks processed independently by the other threads. While 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; note, however, depending on the specific computational load for each thread, it might be needed to divide the RDS path into a couple of threads (e.g., decouple timing recovery and frame synchronization from the rest of the RDS path). It is also worth mentioning that the audio/RDS threads are always behind (in terms of block count) with respect to the RF front-end thread; however, they can get ahead of each other arbitrarily. So long as the synchronization queue is limited in size, this divergence between audio/RDS paths will not impact the application in a tangible way.

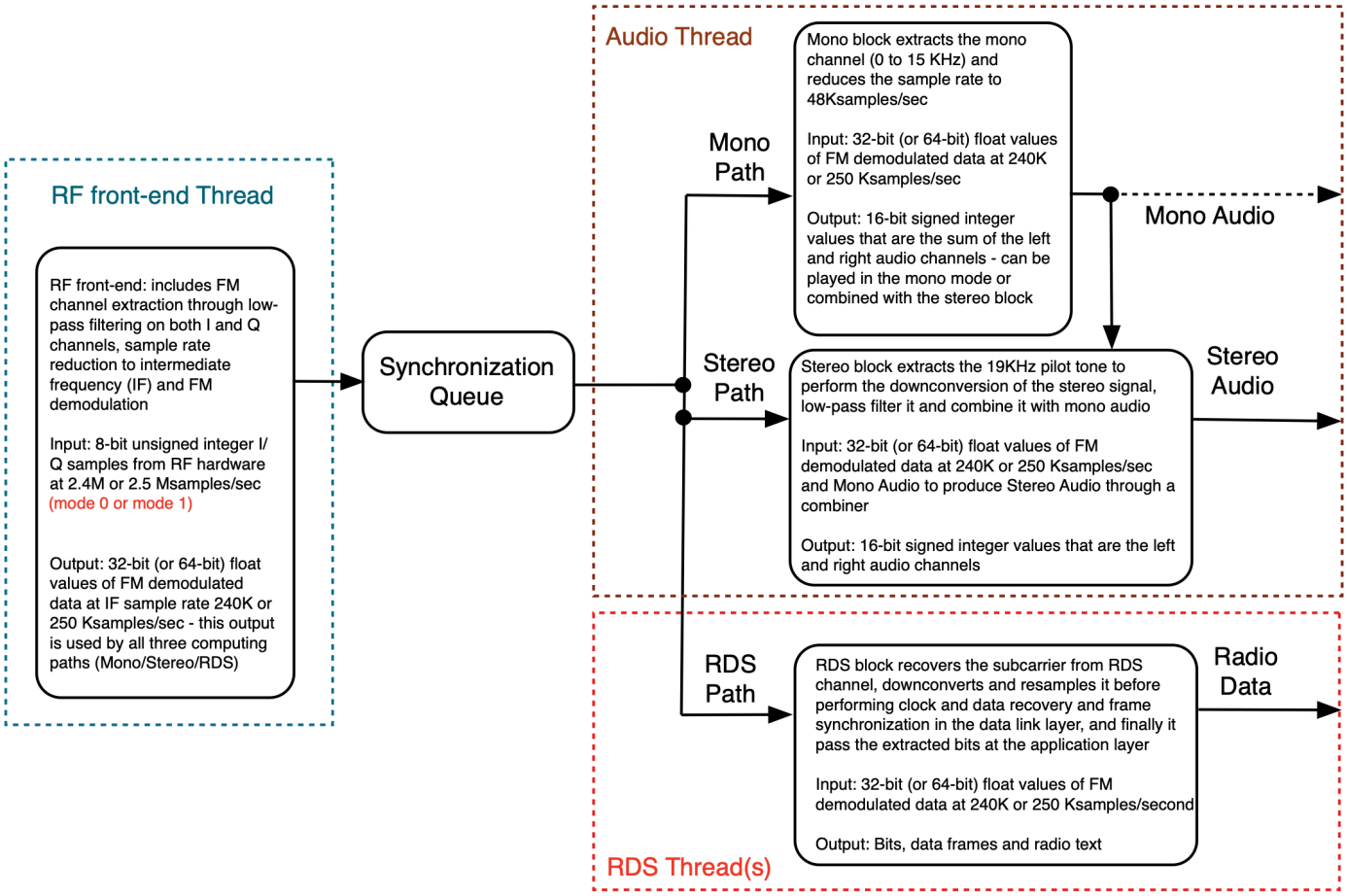


Figure 22: Thread partitioning to leverage the parallelism offered by a multi-core processor

References

[1] NESDR SMArt Series. https://www.nooelec.com/store/sdr/sdr-receivers/smart.html. Accessed: 2021-02.

[2] Osmocom RTL SDR. https://osmocom.org/projects/rtl-sdr/wiki/Rtl-sdr. Accessed: 2021-02.

[3] Raspberry Pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Accessed: 2021-02.

[4] FM Broadcasting. https://en.wikipedia.org/wiki/FM_broadcasting. Accessed: 2021-02.

[5] Radio Data System. https://en.wikipedia.org/wiki/Radio_Data_System. Accessed: 2021-02.

[6] Specification of the Radio Data System (RDS) for VHF/FM Sound Broadcasting in the Frequency Range from 87,5 to 108,0 MHz. http://www.interactive-radio-system.com/docs/EN50067_RDS_Standard.pdf. Accessed: 2021-02.

[7] United States RBDS Standard Specification of the radio broadcast data system (RBDS). https://www.nrscstandards.org/standards-and-guidelines/documents/standards/nrsc-4-b.pdf. Accessed: 2021-02.

[8] GNU Radio. https://wiki.gnuradio.org/. Accessed: 2021-02.

[9] Gqrx SDR. https://gqrx.dk/. Accessed: 2021-02.

[10] FM Stereo/RDS Theory. http://rfmw.em.keysight.com/wireless/helpfiles/n7611b/Content/Main/FM_Broadcasting.htm. Accessed: 2021-02.

[11] Introduction to FM-Stereo-RDS Modulation. https://www.advantest.com/documents/11348/7898f05e-0a52-4e68-9221-3b8b75595436. Accessed: 2021-02.

[12] Frequency Modulation (FM). https://www.ni.com/en-ca/innovations/white-papers/06/frequency-modulation--fm-.html. Accessed: 2021-02.

[13] Square-Root Raised Cosine Signals (SRRC). https://gssc.esa.int/navipedia/index.php/Square-Root_Raised_Cosine_Signals_(SRRC). Accessed: 2021-02.

[14] 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: 2021-02.

[15] Error Detection and Correction. https://www.mathworks.com/help/comm/ug/error-detection-and-correction.html. Accessed: 2021-02.

Enjoy!

Appendix

The full details for the theory and implementation of the modified shortened cyclic code used in RBDS are in Appendix B of the official standard document [7]. The parity check matrix from section B.2.1 from [7] (used for frame synchronization in the receiver) is reproduced below for self-containment purposes.

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

A 26-bit block containing the information word (16 bits) and its checkword produced at the transmitter (10 bits) that is added with an offset word (A, B, C, C' or D), which depends on the block type, will pre-multiply the above parity check matrix in a Galois field (GF(2)). In GF(2), a pointwise multiplication is logic AND between two bits, and pointwise addition is an exclusive OR. This vector/matrix multiplication result in GF(2) is a 10-bit syndrome that confirms the block type. In the table below, you can find 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