3DY4 Project - Custom Settings for Group 35

For audio processing (both mono and stereo), the four modes of operation are defined in terms of: the front-end (input) sample rate (RF Fs) used to acquire the raw I/Q samples from the RF dongle; the intermediate frequency sample rate (IF Fs) used at the input/output of the FM demodulator - note that the three processing paths (mono, stereo, and RDS) receive data from the FM demodulator at the IF Fs; and the audio (output) sample rate (Audio Fs) used to output the audio samples from your SDR software to an audio player. In the project specification for your group 35, the following values apply to RF Fs, IF Fs, and Audio Fs in the four different modes of operation.

Settings for group 35	Mode 0	Mode 1	Mode 2	Mode 3
RF Fs (KSamples/sec)	2400	960	2400	1152
IF Fs (KSamples/sec)	240	160	240	128
Audio Fs (KSamples/sec)	48	40	44.1	44.1

Mode 0 uses the same sample rate values as those used in lab 3 for audio processing. Your group 35 has a unique combination of settings for modes 1, 2, and 3. Therefore, the values described for mode 1 in the project document need to be replaced with those in the above table for modes 1, 2, and 3.

In modes 0 and 2, the IF Fs equals 240 KSamples/sec at the input of the RDS path. Since the RDS path does not use the audio Fs, modes 0 and 2 will be differentiated by changing the number of samples per symbol (SPS) used for clock and data recovery. The project specification for your group 35 will have to support the following two custom values for SPS.

Settings for group 35	Mode 0	Mode 2
Samples per symbol (SPS)	26	39

Note that, depending on the mode of operation, the RDS symbol rate (in samples/sec) at the output of the rational resampler from the RDS demodulator (see Figure 12 from the project document) will be the SPS value from the above table multiplied by 2,375 (symbols/sec).

By default, a packet of data transferred from one module in the signal flow graph to another should carry as many samples as are buffered in 40 ms of real-time acquisition (there will be one exception, explained on the next page). The number of partial products accumulated for an output sample for any filter should be in the range of 85 to 115. This is the number of taps for most filters, except for resamplers, where the number of taps is scaled by the expansion scale factor. Nonetheless, the number of non-zero partial products for the resampler should still meet the above constraint.

To stay organized, the judicious approach is to start with a model before proceeding to the implementation. Therefore, you must provide `Python` models for mode 0 of operation for both stereo and RDS paths if your `C++` code is to be assessed. For the same reason, you are also required to provide unit tests for your downsampler and resampler implementations, which prove that the slow/inefficient version (derived from the first principles of filtering with decimation and expansion implemented as pre/post-processing functions) is equivalent to the fast version needed for your code to run in real-time on the Raspberry Pi.

Based on the same principle as above, for incomplete submissions, any `C++` function that is claimed to be completed must have unit tests that prove its correctness and demonstrate how it has been evaluated.

Different use cases for your program and its interface to the `rtl_sdr` and `aplay` are provided on the following pages.

Interfacing your program with `rtl_sdr` and `aplay` through UNIX pipes

When going live, to enable UNIX pipes that stream data into your program from `rtl_sdr` and stream the output of your program to an audio player, such as `aplay`, you should run it as follows from the terminal:

`rtl_sdr` -f 107.1M -s 2400K - | ./`project` 0 m | `aplay` -f S16_LE -c 1 -r 48000

It is critical to note that for Unix piping to work as shown above, it is assumed that your program (called `project` in the above example) prints any information for the user using `std::cerr`, and **only** the audio samples are redirected to `stdout`. For the command line arguments for `rtl_sdr` and `aplay`, type the corresponding command in a Linux terminal. Your program should have two command line arguments: mode (0, 1, 2, or 3) and the most advanced processing path enabled for the current run of the program (m for mono, s for stereo, and r for RDS). For example, to work in mode 3 with only mono audio enabled, you should run it as follows:

`rtl_sdr` -f 107.1M -s 1152K - | ./`project` 3 m | `aplay` -f S16_LE -c 1 -r 44100

After the stereo path has been implemented, you can run it in mode 3 as follows:

`rtl_sdr` -f 107.1M -s 1152K - | ./`project` 3 s | `aplay` -f S16_LE -c 2 -r 44100

Note that in the above usage, you will stream twice as much data from your program, and therefore, `aplay` needs to be aware that its input has two channels.

After implementing the RDS path, you should run it only in modes 0 or 2. For example, in mode 2, run it as follows:

`rtl_sdr` -f 107.1M -s 2400K - | ./`project` 2 r | `aplay` -f S16_LE -c 2 -r 44100

When the RDS path is enabled, all audio **must** also be processed in stereo mode; hence, your program must stream its output to `aplay` in two channels. Note, however, that the RDS processing path can be ignored when your software runs in modes 1 and 3. In other words, when operating in modes 1 and 3 and the second command line argument is r, you can print a message confirming that RDS is not supported in the respective mode.

By default, in all modes of operation, you will need to scale your blocks to be proportional to 40 ms of real-time data. **Only after** the software has been proven to work correctly with the default block size for **all** modes of operation you should support another command line argument for the RDS path. This argument specifies the number of I/Q pairs to be processed in a block and it must be an integer between 48,000, i.e., equivalent to 20 ms of real-time data for mode 0 or 2, and 120,000, i.e., 50 ms.

`rtl_sdr` -f 107.1M -s 2400K - | ./`project` 0 r 57600 | `aplay` -f S16_LE -c 2 -r 48000

The above example should process 57,600 I/Q pairs in a block, which amounts to 24 ms in mode 0. The integer that follows `r` is relevant only when the RDS path is exercised (which implies that audio will also be processed). If not provided, the block size defaults to 40 ms, which is equivalent to processing 96,000 I/Q pairs when the RF Fs is 2.4 MSamples/sec. It is important to note that the custom block size feature enabled by this extra command line argument will not be assessed **unless** RDS mode was established to run correctly with the default block size.

Emulating streaming during development

While developing your code, you can emulate the streaming of I/Q samples from `rtl_sdr` by redirecting the pre-recorded I/Q samples using the `cat` command in UNIX. For example, to stream the pre-recorded samples from *iq_samples.raw* to your program that runs in mode 1 for the mono path, do as follows:

`cat` *iq_samples.raw* | ./`project` 1 m | `aplay` -f S16_LE -c 1 -r 40000

Note that you must ensure that the I/Q samples have been acquired at the appropriate sample rate, e.g., 960 KSamples/sec for your project settings for mode 1. To record five seconds, do as follows when the RF dongle is plugged in:

`rtl_sdr` -f 107.1M -s 960K -n 4800000 - > *iq_samples.raw*

For modes 0 and 2, the I/Q samples from lab 3 can be reused because they have been acquired at 2.4 MSamples/sec, and they are sufficient to get you started with troubleshooting (even with the resampler that is needed on the mono path in mode 2). If you wish to reuse the I/Q samples from lab 3 for other modes at another sample rate, you can change their sample rate using the Python script `fmRateChange.py` from the `model` sub-folder.