RSPduoEME
Software Description

D Warwick G4EEV

December 2020

The Application is written in Qt V5.1 using QT Creator V4.6 on a Windows10 platform. The interface to the RSPduo is provided through the sdrplay_api.dll (X86) version 3.06 or later. The application comprises of 5 C++ classes; main, RSPduoInterface, MainWindow, ProcessThread, and DSPthread. In addition two header files; sdrplay_api.h and filters.h, define the sdrplay_api and DSP filter definitions.

Architecture:

The sdrplay_api provides functions to control the RSPduo and callback functions to transfer data and status. The sdrplay_api functions are compiled for C calling conventions and have to be defined as [extern "C"] when calling from C++. In addition the callback functions have to be defined and run outside of any C++ class. For this reason the callback functions are defined before any class definitions in rspduointerface.cpp.

The data returned by the sdrplay_api callback functions is processed to reduce the data rate for each of the two input channels from 2M samples/sec to either 192K or 96K samples/sec. As this is not an integer reduction, multi-rate techniques are used in the DSP routines to achieve this. As significant time critical DSP processing is required, the DSP, Processing and MainWindow classes, operate in separate threads. The operation of the application is described below in the descriptions of each of these C++ classes.

MainWindow:

This is the main display window which comprises widgets to take input parameters, a status display for messages, and when in dual UDP output modes, a phase and oscilloscope display. On start-up the constructor starts the ProcessThread in a separate thread and also start the RSPduoInterface and Timer. Saved settings are also read and the input widgets initialised with these values.

The UpdateParameters function is called when key inputs are modified and before starting processing. UpdateParameters sets the operating flags for the ProcessingThread and DSPThread, based upon the required operating mode. These parameters are only updated while processing is stopped, with one exception, the selection of the soundcard output channel in dual output mode. Additionally, changes to LNA and IF gain can be made directly to the RSPduoInterface while the processing is in operation.

Processing is started when the Start button is pressed and on_StartButton_clicked slot is called. The RSPduoInterface Start slot is called with the required frequency (converted to Hz) and LNA/IF gain settings. The mode parameters are set with a call to UpdateParameters and the DSP oscillator phase is reset to zero for both input channels by emitting a signal to the DSPThread GenerateSineCosTable. Finally the processing is started by emitting a StartProcessingThread signal to the Processing Thread and the Start button text is changed to Stop. When Stop is pressed, the StopProcessingThread signal is emitted and the RSPduoInterface Stop function is called.

In Dual Output modes the Phase indicator and Oscilloscope display are shown in an extension at the bottom of the MainWindow. The purpose of these displays is to view and set the relative phase between channel A and B. The display extension is activated by calling a screen resize with appropriate parameters from the UpdateParameters function. The Timer (100mS) is activated when processing is active and calls the 'update' function to cause the screen to be re-painted by a call to the paintEvent slot. The paintEvent function draws the backgrounds, reads the channel output data from the DSP thread Circular OutputBuffers (TIMF2 IP data as pointed to by LatestOutputDataIndex), calculates the phase, and displays the data in real time (100mS) on the Phase and Oscilloscope displays. During this process the average phase error is calculated and the phase correction calculated by subtracting the error from the RequiredPhase value. When the Set button is pressed it is disabled and a signal is emitted to the DSPThread GenerateSineCosTable function, which calculates a new table with the required starting phase between channels A and B. When the Set button is disabled, each Timer timeout increments the PhaseDisplayTimeout; after reaching 10 (1000mS), the input data is frozen allowing the display to be read. Finally the paintEvent function checks the OverloadFlag, set by the RSPduoInterface object, and if set, draws the text 'Overload' to the display.

In order to monitor the process timing and alert when insufficient CPU time is available, the on_Timer function calculates the AverageTime by reading the ProcessTime list produced by the DSPThread. If the AverageTime is > 35mS for each 40mS of data, a Debug message is created indicating the percentage of time taken.

RSPduoInterface:

The SDRplay API/Hardware Interface comprise of a hardware driver service and the sdrplay_api.dll. These are linked and need to be from the same version. Typically these are downloaded when SDRuno is installed or can be downloaded separately. The sdrplay_api.dll (X86 version, for my QT Creator version) needs to be copied to where the RSPduoEME.exe is executed from to be found at runtime. The current API version is 3.07 dated 14/4/20 (previous 3.06 dated 25/11/19). This application checks the API version numbers which need to be grater than 3.06. It is anticipated the future releases of the API will be backwards compatible, however, this may be a cause for future issues?

This code is based upon the example code provided by the sdrplay_api documentation. The definitions for the variables and functions are contained in the header file sdrplay_api.h. At the head of the rspduointerface.cpp file some global definitions are made which need to be defined outside of any class, these include the three callback functions and their variables.

StreamACallback and StreamBCallback are called by the sdrplay.api when data is ready to be transferred from the A and B channels. The data is saved in a two dimensional array as blocks of 80000 [INPUT_BUFFER_SIZE] * 10 [BUFFERS]. Each of the 10 buffers holding 80000 samples or 40mS of type 'float' data. When a buffer is completely filled the variables A_LastBuffer and B_LastBuffer are set to the index of the last completed buffer, indicating the state of the input buffers for reading by the application. The buffers wrap around and form a circular buffer system.

The EventCallback function handles the event messages from the sdrplay_api and provides messages which are added to the MessageList for reading and displaying by the MainWindow. In addition the OverloadFlag is set or reset for display by the MainWindow.

The RSPduoInterface Class defines four public functions; Start, Stop, ChangeIFGain and ChangeLNAgain. These functions are called from the MainWindow to Start / Stop the RSPduo and change the gain settings. In addition a private function, on_Timeout, is used to read the MessageList and send status messages to MainWindow via the Status signal.

ProcessThread:

The ProcessThread Class is responsible for starting the DSPThread and outputting data to the Soundcard and the Network (UDP). The Start function initialises the Soundcard and Network TxSocket as required, taking the current values of SampleRate and SelectedOutputDevice to configure the Soundcard output device. The Soundcard output buffer size is selected to be five times larger than that required for one 40mS block of output data; this is because the output device will signal for new data when one fifth of the data has been sent. The Qt soundcard output is not perfect in timing and a sufficiently large buffer is therefore required. For 96000 sample/sec data, 16 bit stereo (I/Q), the buffer size is 76800 (bytes) or 200mS. Before starting the soundcard output device the input buffer is pre-filled with zeros to allow full buffering. The DSPThread processing is initiated by passing relevant variables, including the LastInput buffer value, and setting the Finished flag to 1. The Timer is also started with a timeout of 40mS.

The Timer timeout signal calls the on_Timeout slot, which in turn calls the ProcessData function; this is required for network output. Additionally, the on_AudioOutput_Notify slot also calls the ProcessData function when more audio output data is required. (The AudioOutput_Notify may not be relied upon, hence, also calling the ProcessData based upon the timer timeout signal.)

When Linrad RAW16 UDP output is selected, a TCP server is started listening for Linrad Mode Requects on port 49812. When Linrad receives the UDP RAW16 stream Linrad sends a Moder Request which is responded to by the server. This is required by Linrad to start processing the data. There is the potential that if Linrad is also outputting data (TIMF2 to MAP65) there is a clash of ports and Linrad has to be run on a separate machine.

The ProcessData function is responsible for outputting processed data to the Soundcard output device and Network (UDP) as required. The output data is read from the respective

CircularOutputBuffers of the DSPThread. The amount of data output is calculated by taking into account the data available (difference between InPoint and respective OutPoint pointers) and that which the Soundcard Output Buffer requires or that required to make complete UDP packets. The Soundcard output data is converted from type 'float' floating point format to integer 16 bit and packed into four bytes for output.

The UDP TIMF2 output data is packed into packets of 1416 bytes, containing a 24 byte header and 1392 byte payload. The packet format is compatible with the Linrad TIMF2 format (except the header 'ptr' value is not used and is set to 0). It was found that sending lots of UDP packets at once in a loop overwhelmed the network (average data rate for MAP65 input is about 12.5Mb/s), this was slowed down by using a 'usleep' command in the loop.

The Linrad RAW16 UDP output is also packed into packets of 1416 bytes, containing a 24 byte header and 1392 byte payload. The data is derived from the Soundcard format Circular Output Buffer as the spectrum correction applied to the TIMF2 data is not required. The floating point data is converted to 16 bit integer for sending.

Various status messages from the network and Soundcard outputs are handled via appropriate signals and slots and emit a signal to the MainWindow Status Display.

DSP Thread:

The DSPthread class is invoked from the ProcessThread class and is responsible for reading the input channel data from the RSPduoInterface input buffers and converting this to suitable formats and sample rates for output to the soundcard and network (UDP). The input channel data is at 2M sample/sec per channel Real data in 'float' format, centred on an IF of 450KHz. The required output is either 96000 or 192000 samples/sec complex baseband I/Q format. The input channels therefore need to be tuned to a centre IF frequency, converted to baseband I/Q, and rate changed to 96000 or 192000 samples/sec. The DSPthread operates as a separate processing thread to process data in blocks of 40ms [INPUT_BUFFER_SIZE]. The Timer timeout signal calls onTimer every 20mS to see if the previous conversion has finished and new data is ready for processing.

The Constructor of this class initiates all the required data arrays required to hold intermediate data from the various filter and decimation stages as well as the output data for the soundcard and network (TIMF2) output. The Destructor is responsible for deleting these arrays. The DSP FIR filter coefficients and delay line arrays are defined in the filters.h file. The FIR filters were designed using Matlab/Octave and their code is also contained in the filters.h file for reference.

In order to tune the input channels to the centre of the 1MHz bandwidth of the 2M sample/sec Real data streams and convert to Complex baseband I/Q format, a digital oscillator is required. Due to the high sample rates involved this comprises of a look-up table of pre-calculated sine and cosine values. In order to tune 450KHz to 500KHz it was found that a circular table of 40 values incremented by 0.45*Pi was required. In addition, to align the phase of the incoming channels, the starting phase of one channel can be offset. The GenerateSineCosTable function is responsible for calculating these tables.

Depending whether one or two channels are required for processing there are two processing functions, PrpocessBufferA and ProcessBufferAB. These operate identically, except that only one buffer is processed in the former, saving valuable processing time. In order to reduce and change the sample rate a series of decimation and rate conversion is employed, these processes require FIR filters to prevent alias and image products. These FIR filters employ the Polyphase technique to gain efficiency and avoid unwanted calculations associated with the decimation and interpolation process. The operation of the DSP process is listed below.

- Take data at 2M samples/sec and complex multiply with the tuner oscillator to centre the IF and produce 1MHz bandwidth baseband I/Q in _Buffer.
- Decimate by 2 (filter D2A), 1M sample/sec data now in _Buffer[indexed 2].
- If the required sample rate is 96000, decimate by 2 (filter D2B), 500K sample/sec data now in _Buffer[indexed 4]
- Decimate by 5 (filter D5), 100K/200K samples/sec data now in D5FilterOut.
- Perform up-sample by 6 (filter US6), 600K/1200K data now in US6FilterOut.
- Decimate by 5, 120K / 240K sample/sec data in US6FilterOut[indexed 5].
- Perform up-sample by 4 (filter US4), 480K/960K sample/sec data now in US4FilterOut.
- Decimate by 5, 96K/192K sample/sec data in US4FilterOut[indexed 5].

The US4FilterOutput is copied to the SoundCardOut buffer for use in Soundcard and UDP(RAW16 format) output.

The US4FilterOutput data is also rotated by 180 degrees by complex mutilation with Pi. This is to bring the output format in line with the TIMF2 format required by Map65. This is performed in-place using the US4FilterOut[indexed 5] buffer.

The Soundcard and Network format data are copied to the respective CircularOutputBuffers and the circular buffer pointer InPoint is updated with wrap-around. The ProcessThread uses separate output pointers so only one thread ever writes to a memory location. As both threads can read InPoint, this needs to be declared as a “volatile” variable. Finally, the finished flag is set and the next onTimer event will start another sequence provided the input buffer pointer, LastBuffer, has incremented.

The DSPthread is the most time critical and it was found that with dual channels being processed, this could be achieved in about 20ms for a 40ms block of data. (AMD A10-7800 Radeon R7, 12 Compute Cores 4C @ 3.5GHz The overall timing on Windows 10 Task Manager being about 14%) In order to monitor this, the process time is measured and appended to the ProcessTime list for use by the MainWindow if required.