QRSS Grabber

Abstract

This project uses an mbed as the core of a QRSS receiver – an RF receiver designed to digitize a very small bandwidth of RF signals, and provide them to a server for processing into a spectrum image for real-time display on a website.

QRSS is one of many modes of communications and experimentation used by radio amateurs. It involved the transmission of very low speed Morse (3 second long ‘dits’) from very low power transmitters, and their reception on special ‘visually’ receivers called ‘grabbers’. Because of the slow transmission rates very low level signals can still be received, even if they cannot be hear by ear. The grabbers receive a small bandwidth of the RF spectrum, convert it to an image (spectrum vs time), and make it available on a webpage. Reception of a signal can therefore be viewed in (almost) real-time from receiver stations all over the world.

The normal hardware involved with a grabber receive system requires a very stable radio receiver, a PC (with soundcard), and an internet connection. The receiver audio is feed into the PC soundcard. Software then converts the audio to an image. Further software makes these images available on a webpage. Each Grabbers is hosted on its own webpage, so finding your signal can therefore require the inspection of a number of websites.

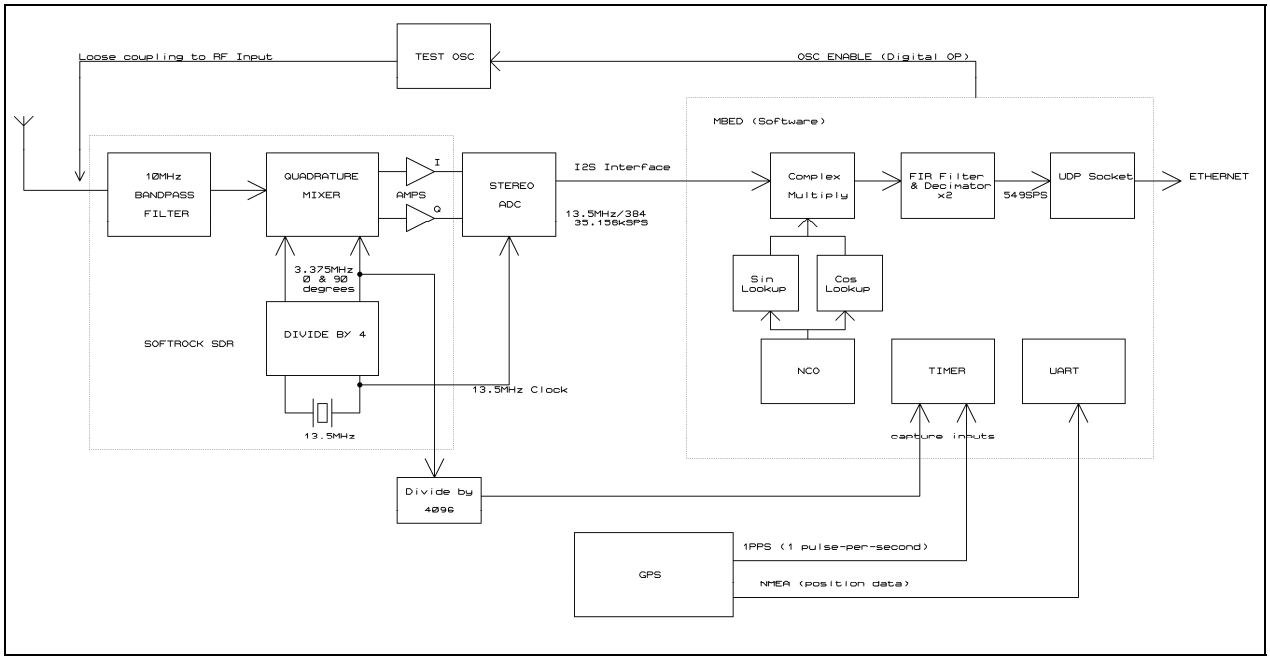
This project. - the ‘QRSS-Rx’ - is designed to improve the QRSS grabbers system. The idea is for a central server to receive spectrum data from all over the world, and then make it available on a single website. The QRSS-Rx is the receiver part of this system. Many of these can be located around the world. They receive and digitize the RF spectrum of interest, and pass the samples data to the server for further processing and display. The QRSS-Rx is designed to be a cheap and simple unit that will allow the setup of a receiver without tying up expensive receivers and PCs.

The QRSS-Rx also includes a GPS receiver. This is used for location (to report this to the server) but more importantly for calibration of the QRSS-Rx frequency components to ensure that it is receiving exactly on frequency. The mbed timer capture feature is used to implement this functionality.

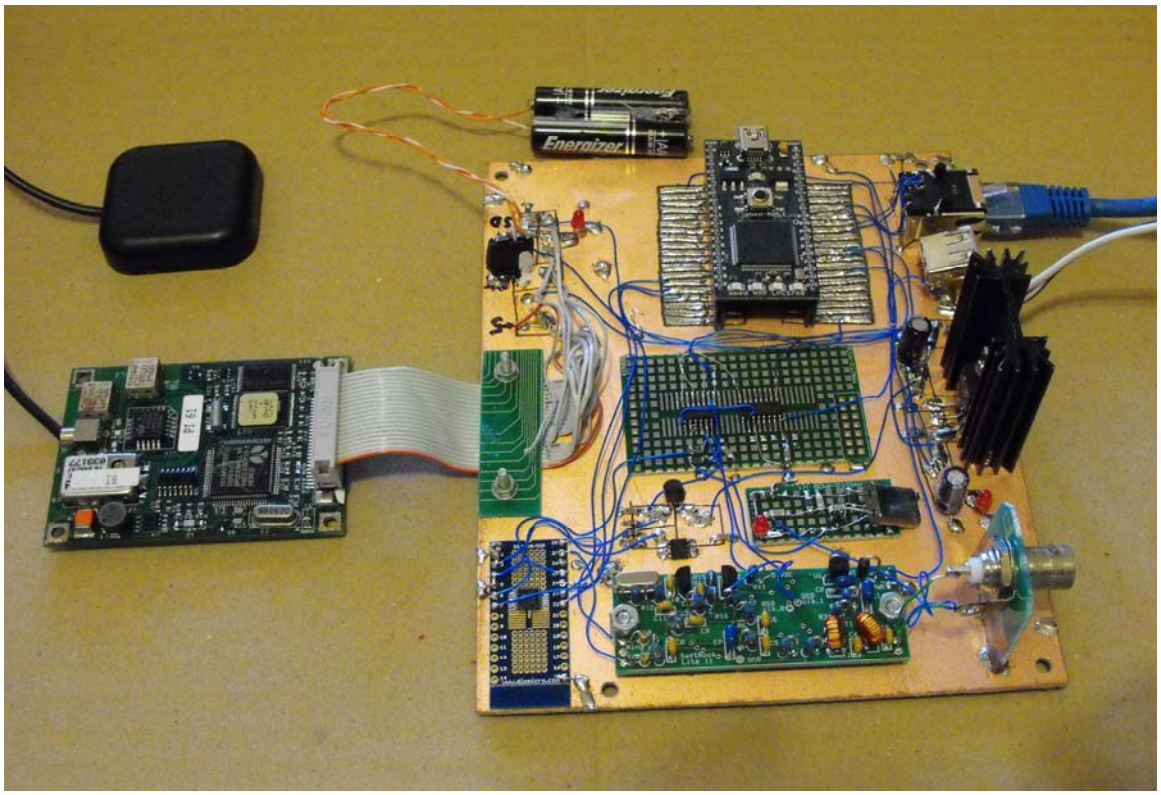
The QRSS-Rx makes use of a ‘Softrock’ software defined radio (SDR) module. The output of this is digitized and passed into the mbed via the I2S interface. In software the samples are further down-converted, filtered, and decimated (sample rate reduction). They are then ready to be passed to a server for processing into grabber images.

The QRSS system works well. RF signals are successfully down converted and are clearly visible in the ADC samples output. The QRSS-Rx development did not include the server development, but a simple test application allow the QRSS-Rx to be exercised, and the sample outputs captured and plotted in Excel to clearly show the down-convetred output.

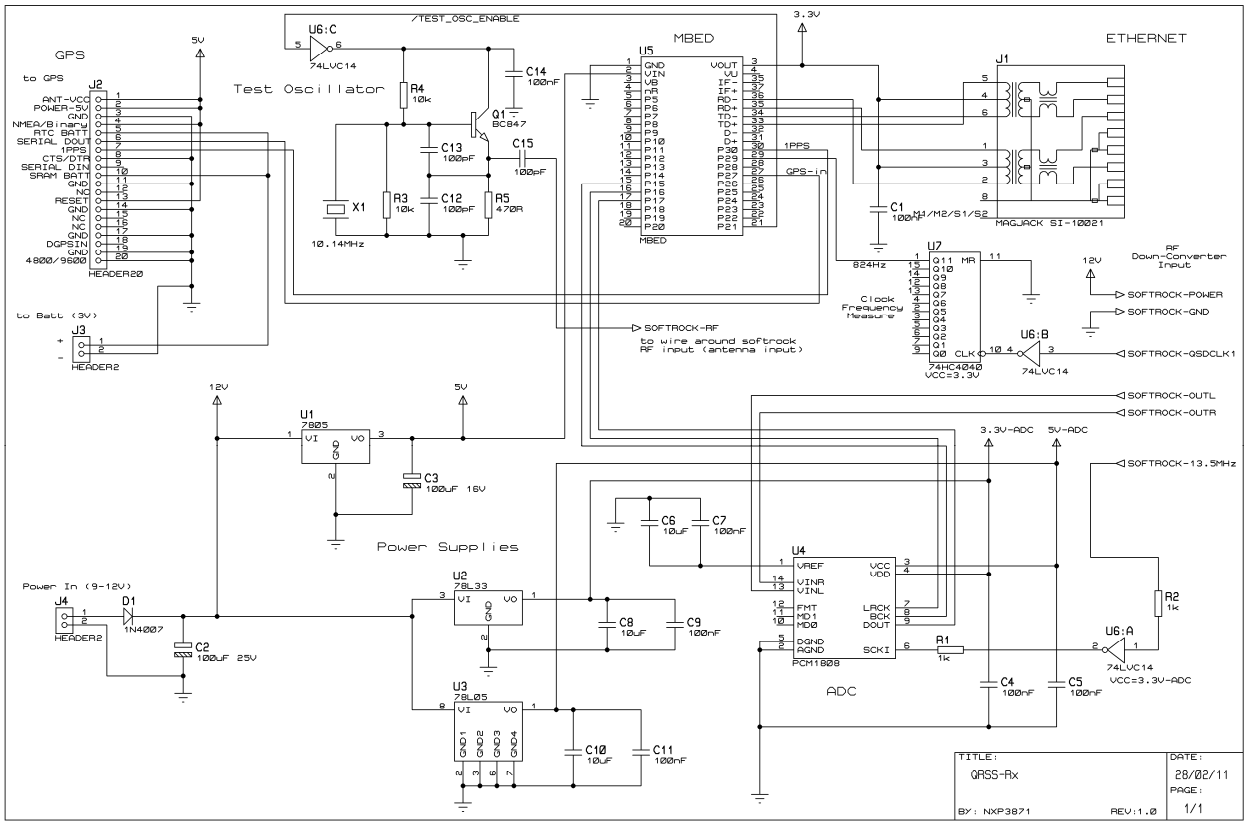
Block Diagram



Project Photo



Circuit Diagram



Code Sample

```
//--------------------------------------------------------------------------
//  DSP Methods
//--------------------------------------------------------------------------

//--------------------------------------------------------------------------
//
//  Set up NCO Increment from a frequency value
//
void TDSPProcessor::NCOFrequency( int32_t iFreq )
{
    int64_t     llFreq;
    int64_t     llInc;

    // Inc (32bit) = Freq * 2^32 / SampleRate
    // Use long long (64 bit int) for calculations to prevent overflow and get best resolution
    llFreq = iFreq;
    llInc = llFreq * 0x10000 * 0x10000 / SAMPLE_RATE;
    // Convert back to 32 bit unsigned via integer type
    uiMixerPhaseIncrement = (uint32_t)((int32_t)llInc);
    printf( "LO Freq set to %d - NCO Inc set to %u\r\n", iFreq, uiMixerPhaseIncrement );
}


//--------------------------------------------------------------------------
//
//  Reset processing
//
void TDSPProcessor::Reset()
```

```
{
    uiMixerPhaseAccumulator = 0;
    bLPFPartailsValid = false;
    Release();
}

//---------------------------------------------------------------------------
//
//  Mix samples with LO (local oscillator)
//
bool TDSPProcessor::MixLO()
{
    int ii;
    int iLen = Length();
    TDataSample * pSample;
    int16_t     iIlo;
    int16_t     iQlo;
    int32_t     iIin;
    int32_t     iQin;

    for ( ii=0,pSample=SamplePtr(); ii<iLen; ii++,pSample++ )
    {
        // generate quadrature oscillators. LO I=cos Q=sin
        iIlo = aiSinTable[ (64+(uiMixerPhaseAccumulator>>24))&0xFF ];     // COS
        iQlo = aiSinTable[ (uiMixerPhaseAccumulator>>24) ];               // SIN
        // inc NCO
        uiMixerPhaseAccumulator += uiMixerPhaseIncrement;
        // scale samples (they are only 24 bits)
        iIin = pSample->iIData / 256;
        iQin = pSample->iQData / 256;
        // complex multiply sample and LO
        // (A + Bi) * (C + Di) = (AC - BD) + (BC + AD)i
        pSample->iIData = (iIin * iIlo) - (iQin * iQlo);
        pSample->iQData = (iQin * iIlo) + (iIin * iQlo);
    }

    return true;
}

//---------------------------------------------------------------------------
//
//  LPF processing
//
bool TDSPProcessor::LPF()
{
/*
    We just code this up for the parameters defined.
    It could be dynamically coded, but for simplicity hard-coding will be used
*/
#if (BUFFERSYS_SIZE!=512)
  #error BUFFERSYS_SIZE has changed from 512
#endif
#if (DSP_FIR_COEFFICIENTS!=511)
  #error DSP_FIR_COEFFICIENTS has changed from 511
#endif
#if (DECIMATION_RATIO!=64)
  #error DECIMATION_RATIO has changed from 64
#endif
#if (LPF_OUTPUTS_SIZE!=8)
  #error LPF_OUTPUTS_SIZE has changed from 8
#endif

    int64_t     iSum;
    int         ii;
    bool        bRet = false;

    // Outputs
    if ( bLPFPartailsValid )
    {
        for ( ii=0; ii<8; ii++ )
        {
```

```
            iSum = MAC_Samples( asLPFPartials[ii].iIData, &(SamplePtr(0)->iIData),
&(aiFIRCoefficients[448-(ii*64)]), (ii*64)+62 );
            asLPFOutputs[ii].iIData = ConvertToOutput( iSum );
            iSum = MAC_Samples( asLPFPartials[ii].iQData, &(SamplePtr(0)->iQData),
&(aiFIRCoefficients[448-(ii*64)]), (ii*64)+62 );
            asLPFOutputs[ii].iQData = ConvertToOutput( iSum );
        }
        bRet = true;
    }
    // Partials
    for ( ii=0; ii<7; ii++ )
    {
        asLPFPartials[ii].iIData = MAC_Samples( 0, &(SamplePtr((ii*64)+64)->iIData),
&(aiFIRCoefficients[0]), (7-ii)*64 );
        asLPFPartials[ii].iQData = MAC_Samples( 0, &(SamplePtr((ii*64)+64)->iQData),
&(aiFIRCoefficients[0]), (7-ii)*64 );
    }
    // Partials[7] = 0
    asLPFPartials[7].iIData = 0;
    asLPFPartials[7].iQData = 0;
    bLPFPartailsValid = true;

    return bRet;
}
```