AEROSPACE REPORT NO.
ATR-2010(5482)-1

# A Flexible Software Based EMWIN/HRIT Prototype Solution for the GOES-R Transition

October 10, 2009

Jeremy Roberson, Esteban Valles, Konstantin Tarasov, Eugene Grayver, Kevin King
Digital Communication Implementation Department, Communication Systems Implementation Subdivision

Prepared for:

GOES-R Program Office
National Oceanic and Atmospheric Administration

Contract No. DG133E07CQ005

Authorized by: Civil and Commercial Operations

AEROSPACE REPORT NO.
ATR-2010(5482)-1

# A Flexible Software Based EMWIN/HRIT Prototype Solution for the GOES-R Transition

October 10, 2009

Jeremy Roberson, Esteban Valles, Konstantin Tarasov, Eugene Grayver, Kevin King
Digital Communication Implementation Department, Communication Systems Implementation Subdivision

Prepared for:

GOES-R Program Office
National Oceanic and Atmospheric Administration

Contract No. DG133E07CQ005

Authorized by: Civil and Commercial Operations

AEROSPACE REPORT NO.
ATR-2010(5482)-1

# A Flexible Software Based EMWIN/HRIT Prototype Solution for the GOES-R Transition

October 10, 2009

Jeremy Roberson, Esteban Valles, Konstantin Tarasov, Eugene Grayver, Kevin King
Digital Communication Implementation Department, Communication Systems Implementation Subdivision

Prepared for:

GOES-R Program Office
National Oceanic and Atmospheric Administration

Contract No. DG133E07CQ005

Authorized by: Civil and Commercial Operations

# A Flexible Software Based EMWIN/HRIT Prototype Solution for the GOES-R Transition

Approved by:

Philip A. Dafesh, Director
Digital Communication Implementation Department
Communication Systems Implementation Subdivision
Communications and Networking Division
Engineering and Technology Group

M. Christian Wallisch, Systems Director
GOES-R and NOAA Architecture Support
Civil and Commercial Operations

## Contents

## Figures

## Tables

# 1. Introduction

The Geostationary Operational Environmental Satellite R-series (GOES-R) program [1] is a collaborative development and acquisition effort between the National Oceanic and Atmospheric Administration (NOAA) and the National Aeronautics and Space Administration (NASA). There are many signals of interest transmitted on the GOES satellite constellation [2]. This document focuses on two signals in particular, the Emergency Managers Weather Information Network Signal (EMWIN) [3][4], and the Low Rate Information Transmission, or LRIT [and its follow-on in the GOES-R era, High Rate Information Transmission (HRIT)] [5][6].

In the current GOES I-M era, EMWIN-I (FSK modulated) and LRIT are being transmitted over the GOES-11 (West) and GOES-12 (East) satellites (Table 1). In the GOES N-O-P era, the EMWIN-N signal will be transmitted on GOES-13 through GOES-15. The EMWIN-I and LRIT signals have their own carriers but share transponders, while the EMWIN-N and LRIT signals have their own transponders. There are commercially available EMWIN receivers currently available [7].

In the GOES-R era, the EMWIN and LRIT information bit-stream will be combined before signal modulation and transmission. The new LRIT signal, with embedded EMWIN, will have a transmission baud rate of 927 kilo symbols-per-second (ksps). Because the new combined signal will be at a higher rate, its name will be changed from LRIT (from the GOES N/O/P era) to HRIT[1] in the GOES-R era.

Table 1. Signals of Interest from NOAA's GOES Satellites

| Spacecraft | Launch Year | Signal | Modulation | Throughput [sps] |
|---|---|---|---|---|
| GOES-I (8) | 1994 | WEFAX | FSK | 9600 |
| GOES-J (9) | 1995 | WEFAX | FSK | 9600 |
| GOES-K (10) | 1997 | EMWIN-N | OQPSK[2] | 17,970 |
| GOES-L (11) | 2000 | EMWIN-I / LRIT | FSK / BPSK | 9600 /293,000 |
| GOES-M (12) | 2001 | EMWIN-I / LRIT | FSK / BPSK | 9600 /293,000 |
| GOES-N (13) | 2006 | EMWIN-N / LRIT | OQPSK / BPSK | 17,970 / 293,000 |
| GOES-O (14) | 2009 | EMWIN-N / LRIT | OQPSK / BPSK | 17,970 / 293,000 |
| GOES-P (15) | 2010 | EMWIN-N / LRIT | OQPSK / BPSK | 17,970 / 293,000 |
| GOES-Q (16) | Cancelled | | | |
| GOES-R (17) | 2014 | HRIT | BPSK | 927,000 |

The current EMWIN and LRIT receivers will not be able to receive the transmission off of GOES-R when the signal goes active, thus motivating the design of a low-cost EMWIN/HRIT prototype receiver that maintains the spirit of the EMWIN service of maximum accessibility. The prototype receiver can be used now to receive the current EMWIN and LRIT signal, and will be capable of receiving the GOES-R HRIT transmission without any changes to the major components of the system, making it an ideal solution to ease the transition. The prototype is not intended to be commercially available, but rather lay the groundwork for a commercial product lines.

This document is primarily intended for developers and covers system, hardware, and software design considerations. It is important to note how this document is organized before reading further.

1 The name HRIT implicitly refers to both the HRIT and EMWIN data-streams.
2 The GOES-K satellite was updated to transmit the EMWIN-N (OQPSK) signal.

Section 2 presents an overview of the product description. The main hardware and software blocks are introduced. In Section 3, a detailed explanation of the hardware component of the proposed solution is presented. Similar analysis is done for the software components in Section 4. Performance results for the integrated receiver are presented in Section 5. Software and hardware appendices with detailed information for future developers appear in Sections 6 and 7. Conclusions are presented in Section 8.

# 2. EMWIN/HRIT Prototype Solution Description

## 2.1 System Overview

The EMWIN/HRIT prototype receiver is a software-based radio that is designed around an open source framework called GNURadio [8][9]. (An EMWIN-I and EMWIN-N solution designed by Avtec is also a software based radio [10].) Most of the receiver is implemented in software running on a Windows-based PC. However, there is some additional hardware required for the system to work. Figure 1 shows an example of the complete solution. Every system requires a powered satellite dish that is at least 1 meter. Such dishes are commercially available for as low as $500 [11]. All EMWIN-I/N users that currently have a dish will not be required to purchase a new one to receive HRIT. The dish is followed by a Low Noise Block (LNB) Down-converter that amplifies the signal and down-converts from L-band (~1690 MHz) to an IF frequency around 140 MHz. The Gain over Temperature (G/T) Figure of Merit (FOM) of the combined dish and LNB/LNA[3] must be at least -0.3 dB/K.

The bandwidth of EMWIN-I/N is a narrowband signal that allows existing commercial receivers to utilize a PC's sound-card to digitize the data and send it to the computer for further demodulation. The new HRIT signal's bandwidth is too wide for any typical PC component to digitize the signal. New hardware is therefore required with the sole purpose of digitizing the analog signal and streaming the data to the computer for further processing.

Currently, there exists a commercially available device called the Universal Software Radio Peripheral (USRP) designed by Ettus Research LLC that is capable of digitizing the HRIT signal [12][13]. The hardware digitizer designed for the prototype receiver is similar in nature to the USRP design, however unlike the USRP, the prototype hardware was not developed for direct commercial use. This device is referred to as the Aerospace Intermediate frequency Digitizer (AID).

Although not shown in Figure 1, ground systems require that the satellite dish be powered. The AID as designed for this prototype solution does not supply power to the satellite dish, therefore an alternate means of powering the dish is required. The LNB devices used by some ground systems may be unable to supply power to drive the AID board. An inline amplifier will be required in these situations.

## 2.2 Hardware Overview

The AID is a mixed signal hardware board with components that are typical of any communication system. The general characteristics of the board are summarized in Table 2. The input signal power must be between -70 and -5 dBm. The hardware can process signals that have a baud-rate of 10 mega-symbols per second, with bandwidths as large as 20 MHz. However, the software components of the receiver limit the achievable system throughput to much lower values. The current board design is meant to handle downconverted-signals between 130 and 150 MHz. The cost of printing the board and purchasing all of the components is under US $100. A summary of the major components is shown in Table 3.

[3] LNA stands for Low-Noise Amplifier

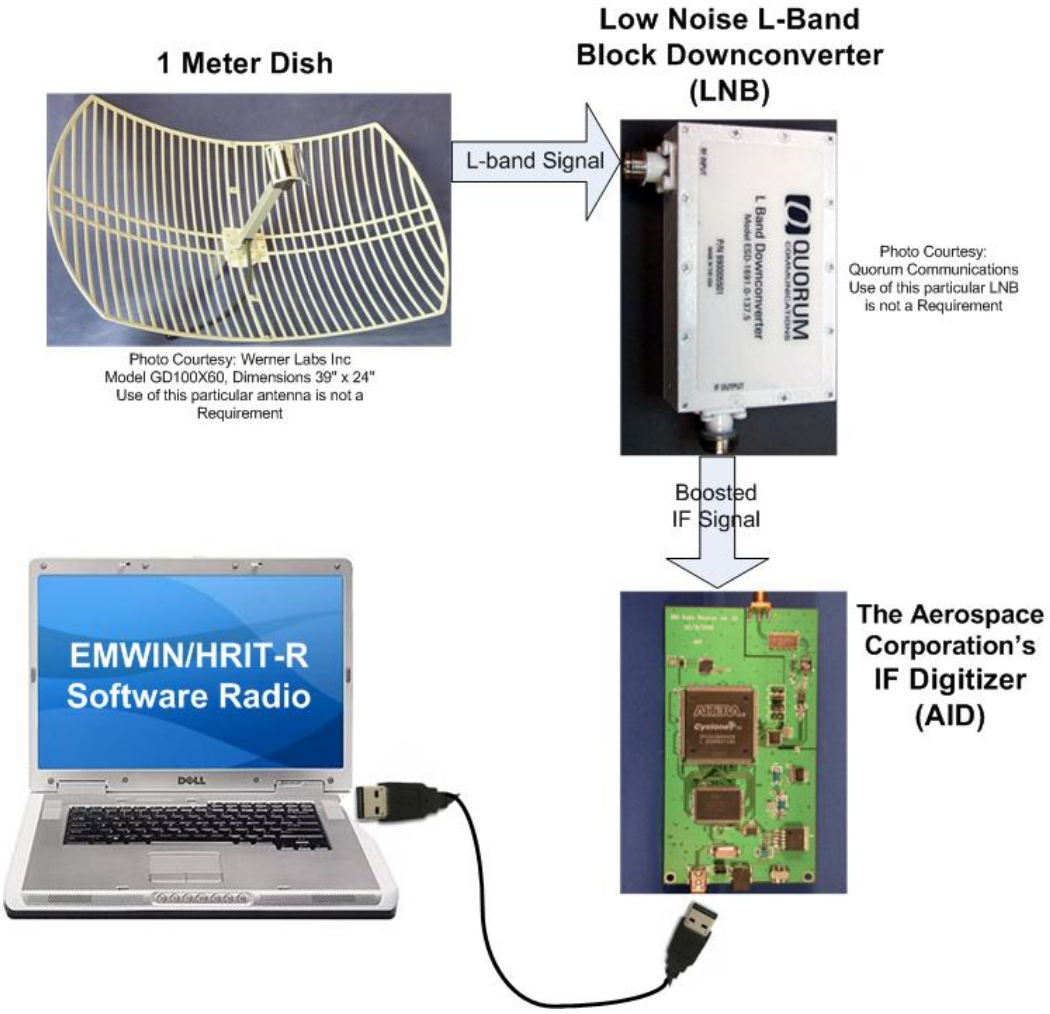


Figure 1. EMWIN/HRIT system diagram.

Table 2. AID Specifications

| | |
|---|---|
| **Part Cost** | Approximately $100 |
| **Sensitivity** | -70 dBm to -5 dBm |
| **IF Range** | 130-150 MHz |
| **Power** | 1.5 Watts, USB Powered |
| **Maximum Signal Bandwidth** | Up to 20 MHz |
| **Maximum Signal Baud Rate** | Up to 10 Msps |

Table 3. AID List of Major Components

| | |
|---|---|
| **Anti-Aliasing Filter** | 20 MHz Bandwidth, fc=140 MHz, Insertion loss 11 dB |
| **AGC** | Maximum Output (13 dBm) |
| **ADC ( Analog to Digital Conv.)** | 64 Mega-samples /second |
| **FPGA** | 5980 Logic Elements |
| **USB Controller** | 24 MHz Input Clock |

### 2.2.1 AID Front End

The front-end analog portion of the AID performs filtering and amplification, while the digital portion samples the analog waveform, down converts it from IF to baseband, and finally streams the digital samples to the computer via the USB controller. The data is sampled with the Analog-to-Digital Converter (ADC) while the down-conversion and down-sampling of the digital samples are performed by a Field Programmable Gate Array (FPGA).

The signal at the AID's SMA[4] input is assumed to have been received by a satellite dish down-converted from RF to IF in the range of 140 ± 6 MHz. The input signal is also assumed to be DC blocked. The board was specifically designed to operate at 140 MHz to be compatible with legacy EMWIN solutions such as EMWIN-N and EMWIN-I.

To suppress out-of-band thermal noise and non-HRIT signals, a 20-MHz wide Surface Acoustic Wave (SAW) filter, with a center frequency of fc = 140 MHz, was used as a first element in the receiver chain. The bandwidth of the filter was chosen to be larger than the bandwidth of the signal to accommodate an array of different weather satellite signals. Typically, the large insertion loss (11 dB) introduced by the SAW filter would affect the overall power; however, the board was designed to receive a signal that has already been boosted. The only amplification capability on the board is the Automatic Gain Controller (AGC) that is driven by the output of the SAW filter. The output of the AGC goes to the ADC for conversion.

The ADC used in the design has 12-bit resolution and is sampled at 64 MHz[5]. Because the sampling frequency of the ADC is less than twice the bandwidth of the IF-signal, the hardware is performing IF-sampling (sub-sampling) instead of direct sampling. The principles of IF-sampling will be detailed further in Section 3.1.3.1.

Although the FPGA and ADC are efficient enough to down-convert a communication signal with very high baud rates, the USB device presents a communication bottleneck of 256 Mega bits per second (Mbps) for transferring output bits to a PC. This bottleneck restricts the maximum input signal baud rate to just over 10 Mega-symbols per second (Msps). Since the HRIT will have a symbol rate of less than 1 Msps, the USB interface provides sufficient throughput for this application.

### 2.2.2 AID Power Meter

The AID has an LED power meter that indicates the power of the input signal as shown in Figure 2. The LED is driven by the output of the AGC. Below is a photo of the AID board, with enclosure and LED power meter. The power meter is intended to provide a visual aid for users while aligning the satellite dish to maximize signal strength.

[4] SMA is a SubMinature version A connector
[5] The original design of the AID box had an ADC running at 48 MHz.


Figure 2. LED example.

### 2.2.3 Power Consumption

The AID can be powered completely by the USB port, requiring no additional AC power sources. The board takes in 5 Volts from USB port and uses 270mA, thus consuming under 1.5 Watts of power.

## 2.3 Software Overview

The EMWIN/HRIT software radio continuously grabs data off of the USB port and demodulates it in real time. As stated earlier, most of the signal processing is done in software, maximizing flexibility and minimizing the amount of additional hardware necessary to process the desired signals. A simplified block diagram of the software is shown in the bottom portion of Figure 3. Although not shown, the first operation by the software is to acquire the carrier frequency. The operation only happens occasionally, thus the first persistent processing block of the software radio is a filter to isolate the exact signal of interest and reject all other signals. After that, the software performs the core functions of digital demodulation: phase tracking, matched filtering, and timing tracking. In the case of EMWIN-N, LRIT, and HRIT, the tracked digital data is sent to a Viterbi decoder for decoding. After that, the data has finally been converted from an IF analog signal, at the output of the LNB, into binary data. From there, additional processing, like Reed-Solomon (RS) decoding, is done to convert the bit-stream into text and images to be viewed using a visualization graphical user interface (GUI).

Table 4 summarizes the EMWIN/HRIT software. The software requires a computer system equipped with a dual-core Intel Processor with Streaming SIMD Extensions version 3 (SSE3)[6] and approximately 1 GB of RAM. The computer also needs at least one free USB 2.0 port. The software was built around multiple libraries including: GNU Radio 3.1.2, the FFTW library "Fastest Fourier Transform in the West", Intel Performance Primitives, and USB libraries. These libraries are of interest to designers, but are completely irrelevant to the end-user. The software currently achieves a maximum speed of over 1.5 Msps, resulting in plenty of computer processing margin.

[6] In 2009, all dual-core Intel processors had SSE3 extensions.

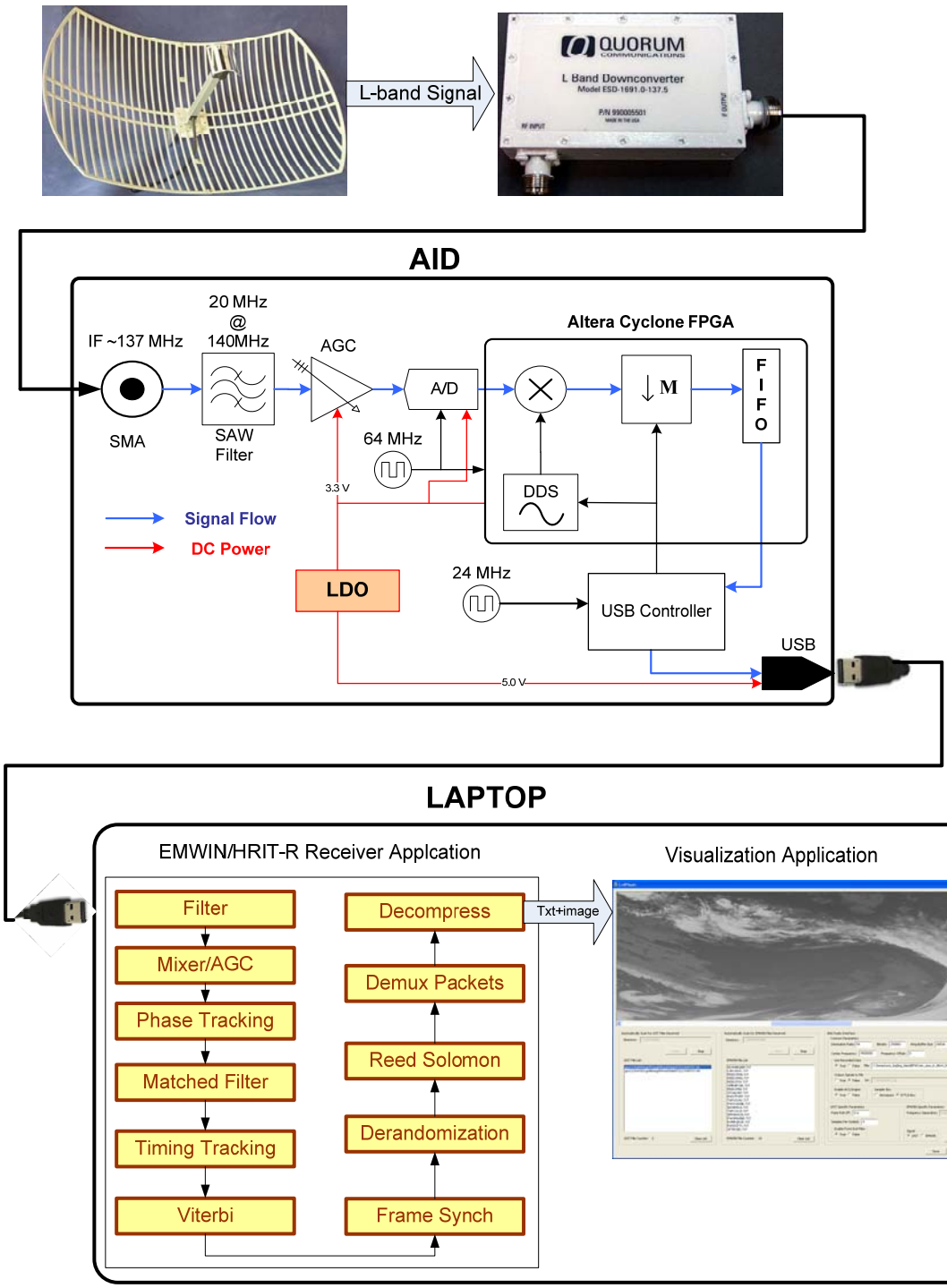


Figure 3. Detailed EMWIN/HRIT system diagram.

Table 4. Software Specifications

| | |
|---|---|
| **Operating System** | Windows XP |
| **Interface** | USB 2.0 |
| **Max Throughput (as of today )** | 1.5 Msps |
| **EMWIN/HRIT Requirements** | Intel Dual Core Processor. 1GB RAM |
| **Software Libraries Used** | GNU Radio 3.1.2, FFTW, Win-libUSB32, Intel Performance Primitives |

# 3. Hardware Description and Performance

## 3.1 Hardware Design

Section 2.2 provided a brief overview of the AID hardware board. This section will go into much greater detail about the hardware board. For a detailed description of how the hardware interfaces with the software, and a detailed look at the schematics, please see Section Appendix B.

For convenience, the AID diagram is shown in Figure 4, where three distinct circuits are highlighted: clock, signal, and power.

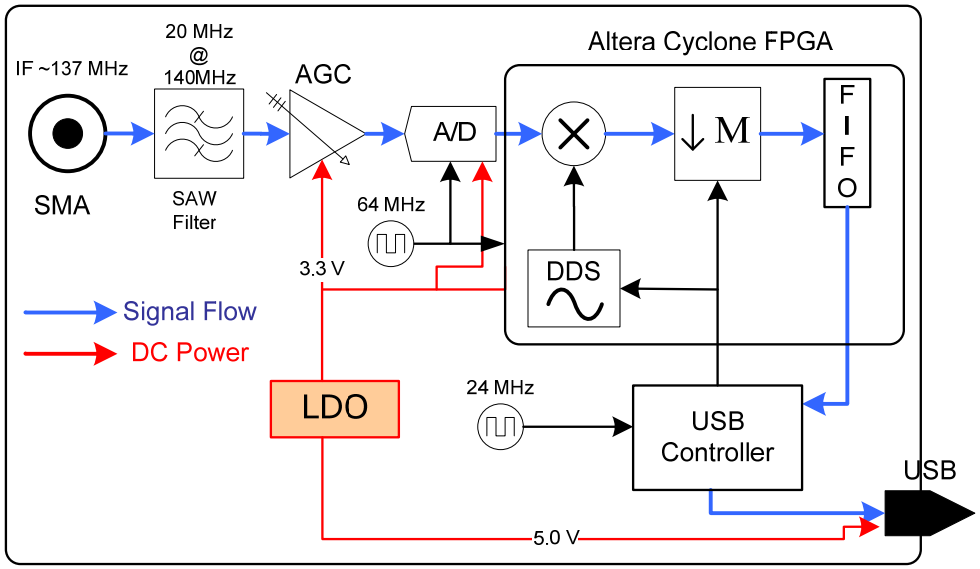


Figure 4. AID diagram.

### 3.1.1 AID: Power Circuit

The AGC, ADC, and FPGA all operate at 3.3 V, while the USB device runs at 5V. A Low-dropout regulator (LDO) is used to produce two voltage levels across the board.

### 3.1.2 AID: Clocking Circuit

A 64 MHz crystal is currently used to drive the FPGA and the ADC, while a 24 MHz crystal is used to drive the USB controller.

### 3.1.3 AID: Signal Circuit

The signal line starts at the SMA connector input and ends at the USB output connector. The signal path represents a typical flow for A/D sampling. The following subsections will analyze in detail the methods used for digitizing and processing the received signals.

#### 3.1.3.1 AID: IF Sub-sampling

The new HRIT signal will have a baud rate of less than 1 Msps, resulting in a Nyquist sampling rate of 2 Mega-samples/second (Msamps/s). If the signal was sampled at baseband, we would only need an ADC operating at above 2 MHz. As shown in Figure 4, the AID samples directly at IF (~140 MHz), instead of shifting the signal down to baseband first. This form of sampling is referred

to direct IF-sampling, or sub-sampling, since the ADC sampling frequency is less than twice the highest frequency (280 MHz).

The bandwidth of interest spans from 134 to 146 MHz, which results in a Nyquist rate of 24 Msamps/s  The reason the bandwidth of interest is so large is because signals of interest for GOES-R users might be at several distinct IF frequencies across the frequency span.  The input bandwidth is restricted by the SAW bandpass filter (BPF).  The exact frequency response of the filter is shown in Figure 5.

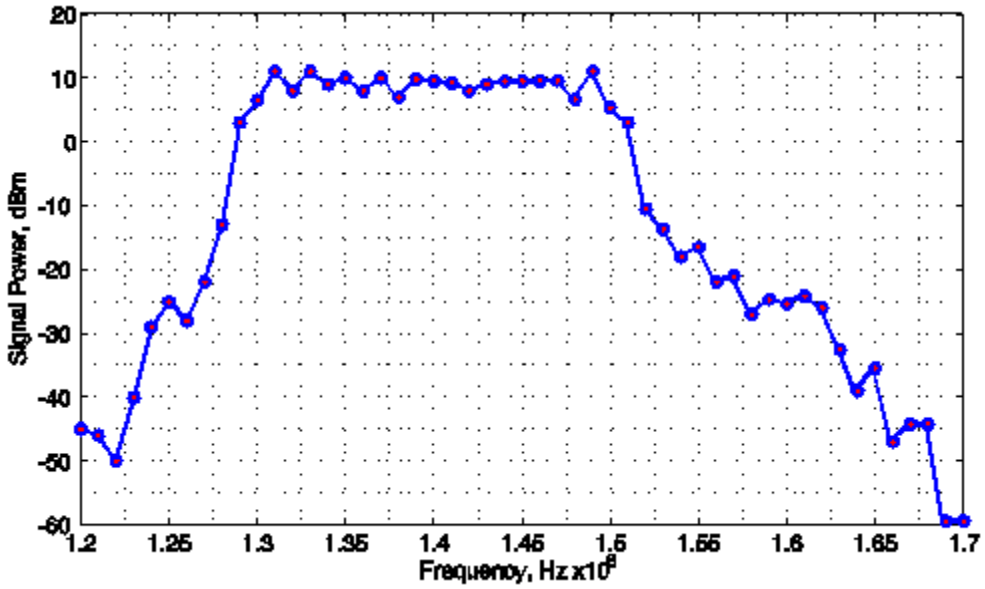


Figure 5. SAW filter frequency response.

Sub-sampling uses the general principle of sampling theory [14].  Figure 6 shows a block diagram of the signal processing done by the AID board. The first block is the SAW BPF, followed by signal processing blocks implemented by the FPGA, which include: the mixer, the decimator, and the low-pass filter. The AGC has no implicit role in sub-sampling and is thus omitted from the figure.

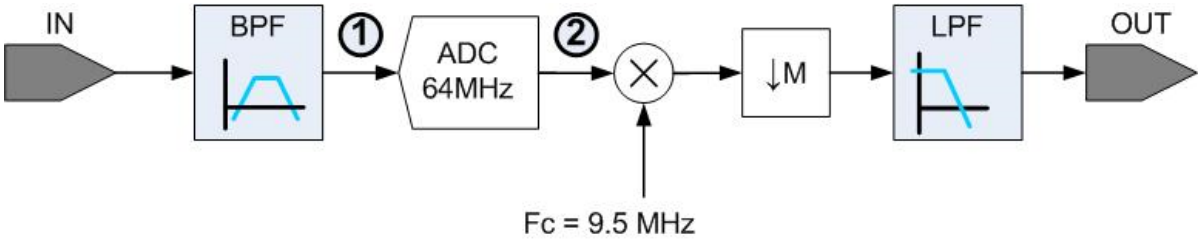


Figure 6. IF-Sampling block diagram.

There are two signals of interest shown in Figure 6, the output of the BPF and the output of the ADC. To illustrate the theory behind IF-sampling, the following example illustrates how sub-sampling works for two signals in our band of interest.

Example: Sub-sampling

Let the input signal be a real waveform with a bandwidth of 400 kHz and a center frequency of 137.5 MHz (these are actually similar characteristics to the LRIT signal). Figure 7 shows the spectrum of Signal-1. The full spectral representation shows spectral content at ±137.5 MHz. The positive spectral image is colored **red**, while the negative content is colored **gray**. The SAW-filter filters this spectrum such that the bandwidth of the signal being sampled does not exceed 20 MHz. In this example, the signal is much less than 20 MHz so the input and output of the BPF are equivalent.

The spectrum of the sampled waveform, Signal-2, is shown in Figure 8. This spectrum follows the standard laws of sampling. Copies of the "negative" and "positive" peaks are spaced every 64 MHz, which is the assumed sampling frequency. The negative copies of the spectrum have aliases at {…, -137.5, -73.5, ***-9.5***, 54.5, 118.5, 182.5, …} MHz, while the positive copies fall at {…, 137.5, 73.5, ***9.5***, -54.5, -118.5, -182.5,…} MHz. The figure shows that the negative and positive images are separated by 19 MHz, thus there is no aliasing between the negative and positive images, despite the fact that the signal was not sampled at the direct Nyquist rate of 280 MHz.

As illustrated in Figure 8, the closest signals to baseband are images centered around 9.5 MHz. The mixer, shown in Figure 6, with frequency Fc = 9.5 MHz, brings the images down to baseband. If the IF frequency of interest, *f*, is between 136 and 144 MHz, the mixer frequency needs to be set to Fc = *f* – 2*64 MHz, to bring the IF signal to baseband.■

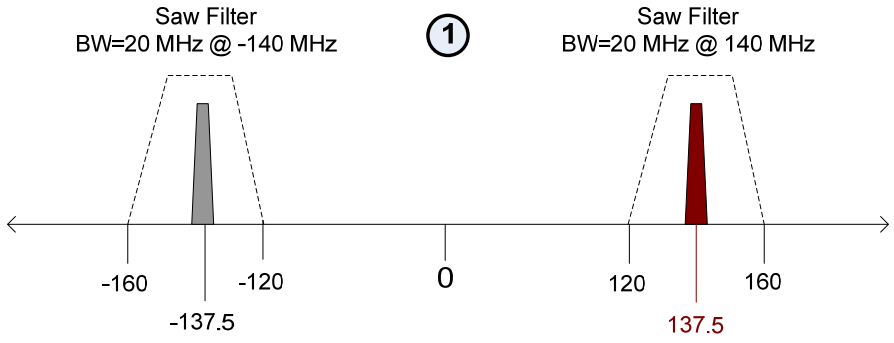


Figure 7. IF Sampling Signal 1.

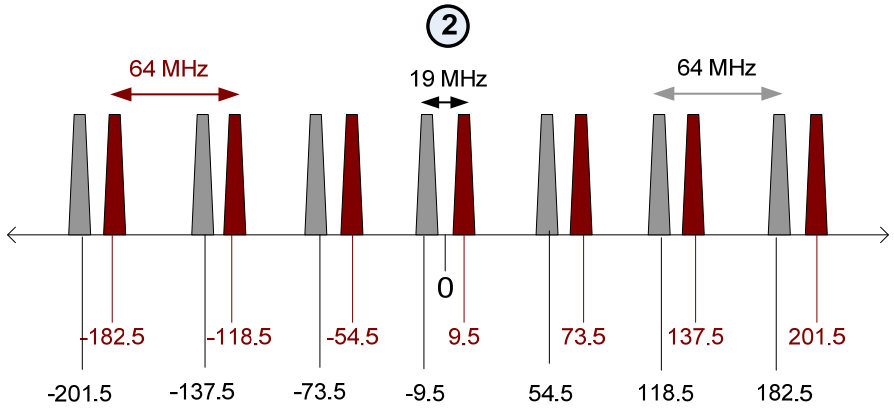


Figure 8. IF Sampling Signal 2.

### 3.1.3.2 Choosing the Sampling Frequency for the AID

The USB controller has the capability of producing a 48 MHz clock output by multiplying a local 24 MHz reference clock by a factor of two, and by using a phase-locked loop. When the AID box was originally designed, in order to minimize component costs, the same 48-MHz signal was initially used to control the ADC.

The sampling approach described in the example above worked great for EMWIN/LRIT legacy signals, running either at 48 or 64MHz. However, the final GOES-R frequency plan, with the HRIT signal at 1697.4 MHz generated some aliasing problems when using a 48-MHz clock.

On Figure 9, we show a plot of the spectrum for the GOES-R signals Global Re-Broadcast signal (GRB) and HRIT after converting from L-band to an IF frequency. The GRB signal is centered at 136.5 MHz and is 12-MHz wide, while HRIT is centered around 143.9 MHz and is around 1.35-MHz wide. When this signal is sampled at 48 MHz with the approach described above, the HRIT images will interfere with each other causing a significant degradation on the overall performance.

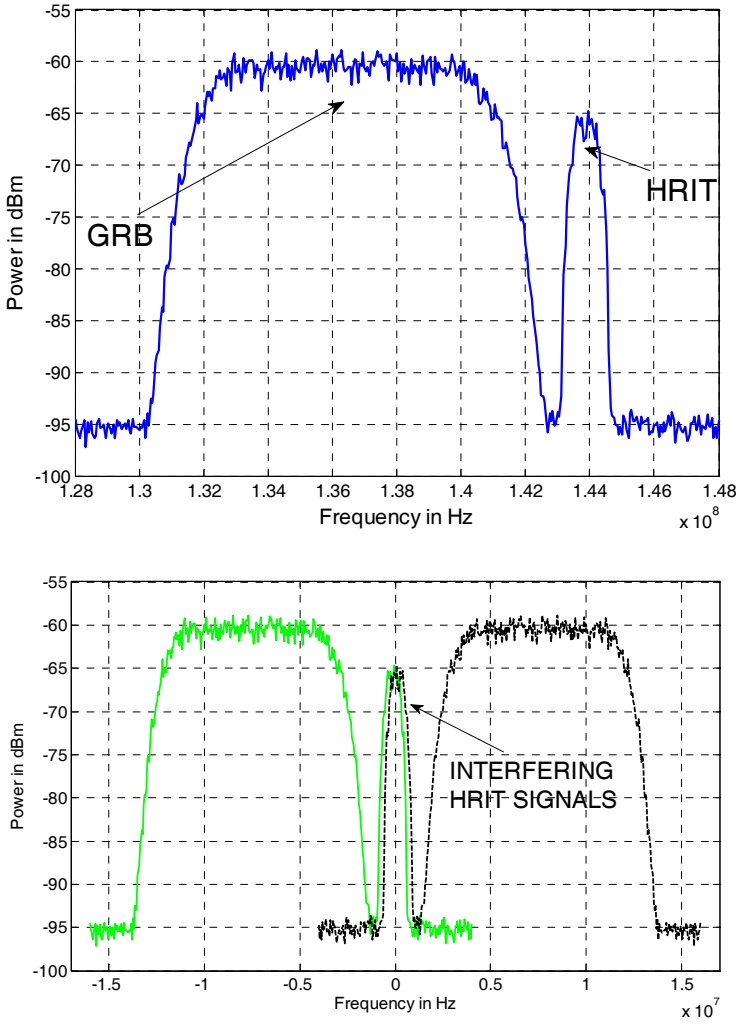


Figure 9. Spectrum of GOES-R image using 48 MHz sampling. The HRIT signals are interfering.

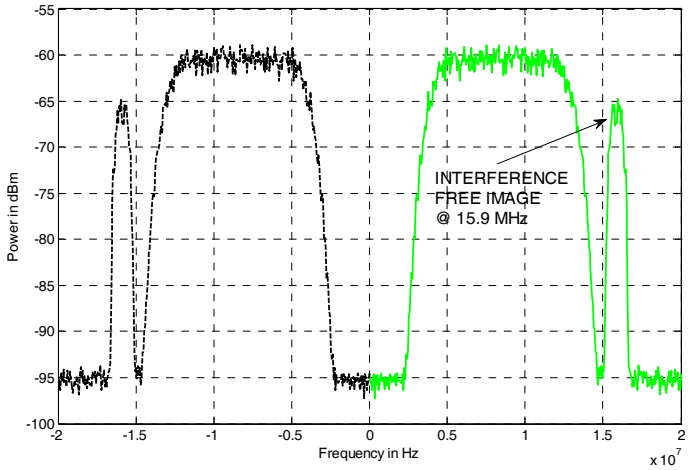


Figure 10. Spectrum of GOES-R image using 64 MHz sampling. The HRIT signals are not interfering.

The GOES-R signals do not interfere when driving the ADC with a 64 MHz clock. This was done by simply replacing the clock that feeds the ADC on the AID board. The current FPGA firmware design and software accommodates a 64 MHz clock, so no changes were needed when changing the ADC clock to 64 MHz. In general, arbitrarily changing the ADC clock will require simple changes to the FPGA firmware design and software.

#### 3.1.3.3 AID: Filtering and Decimation

In traditional digital communication systems, the only step left after sampling a signal would be to digitally low-pass filter (LPF) the baseband signal and send the digital data out for further processing. Because we are performing IF-sampling, there is one additional step to consider before sending the data out, and that is decimation.

Without decimation, the output digital signal would have a rate of 64 Msamps/s and with an ADC resolution of 12 bits, the resultant USB output sample rate would be 576 Msamps/s. Streaming 576 Msamps/s over a USB interface is not possible. Even if the USB interface could sustain that data rate, the software demodulator would not be able to keep up (using a typical consumer grade PC). The purpose of decimation is to reduce the data rate as much as possible before sending it to the PC.

In general, let R be the symbol rate of the signal under study. Let *Fs* be the sampling frequency of the ADC, and M be the decimation rate as shown Figure 6. The number of samples per symbol is simply Fs/R/M.

For the AID device, *Fs* is fixed at $64 \times 10^6$. The symbol rate of the signal of interest is usually known, so the decimation rate can be chosen accordingly to achieve a desired number of samples/symbol. A demodulator should have at least 2 samples/symbol to be able to correctly recover the transmitted signal. At the same time, processing a higher number of samples per symbol reduces the overall throughput, measured in symbols per second (sps). If decoding speed is not a critical issue, as a rule of thumb for a BPSK transmission, we set the samples/symbol to four. When trying to maximize the possible throughput, we set the samples/symbol to two.

The HRIT software demodulator can work with any number of samples/symbol greater than two, but there is no reason to overburden the software with more samples per symbol if it won't increase the

quality of the received data. The samples/symbol must be as close to an integer value as possible in order for the software to work properly.

The decimation range of the AID is any even integer between 2 and 256. When demodulating the LRIT or EMWIN/HRIT signal, the software chooses a decimation rate that results in 2 samples/symbol by default.

The following example illustrates how decimation works. Let the input signal be 293,883 sps. Without down-sampling, there would be $64x10^6/293,883 = 217.78$ samples/symbol. Decimating by 108 would result in 2.01 samples per symbol, and is the default decimation rate used by the prototype receiver to digitize the current LRIT signal. Table 5 shows the parameters for receiving LRIT, EMWIN-N, and EMWIN-I.

Table 5. EMWIN/LRIT Configuration

| **Product** | **HRIT** | **LRIT** | **EMWIN** | **EMWIN** |
|---|---|---|---|---|
| **Modulation** | **BPSK** | **BPSK** | **OQPSK** | **FSK** |
| RF Frequency [MHz] | 1697.4 | 1691 | 1692.7 | 1690.725 |
| IF Frequency [MHz] | 143.9 | 137.5 | 139.2 | 137.225 |
| Data Rate (R) [sps] | 927,000 | 293,883 | 17,970 | 9,600 |
| ADC Sampling Frequency (Fs) [MHz] | 64 | 64 | 64 | 64 |
| Mixer Frequency (Fc) [MHz] | 15.9 | 9.5 | 11.2 | 9.225 |
| Decim. Rate (M) / Samples per Symb. | 1 / 69.04 | 1 / 217.77 | 1 / 3561.5 | 1 / 6666.6 |
| | 16 / 4.3 | 54 / 4.03 | 88 / 40.47 | 74 / 90.09 |
| | 22 / 3.13 | 72 / 3.02 | 178 / 20 | 98 / 68.02 |
| | 34 / 2.04 | 108 / 2.01 | 222 / 16 | 136 / 49.01 |

Now that the principles of sub-sampling have been covered, the real FPGA implementation will be treated.

### 3.1.3.4 Implementation of a Digital Down-Converter on an FPGA

The FPGA digital down converter is shown in Figure 11.

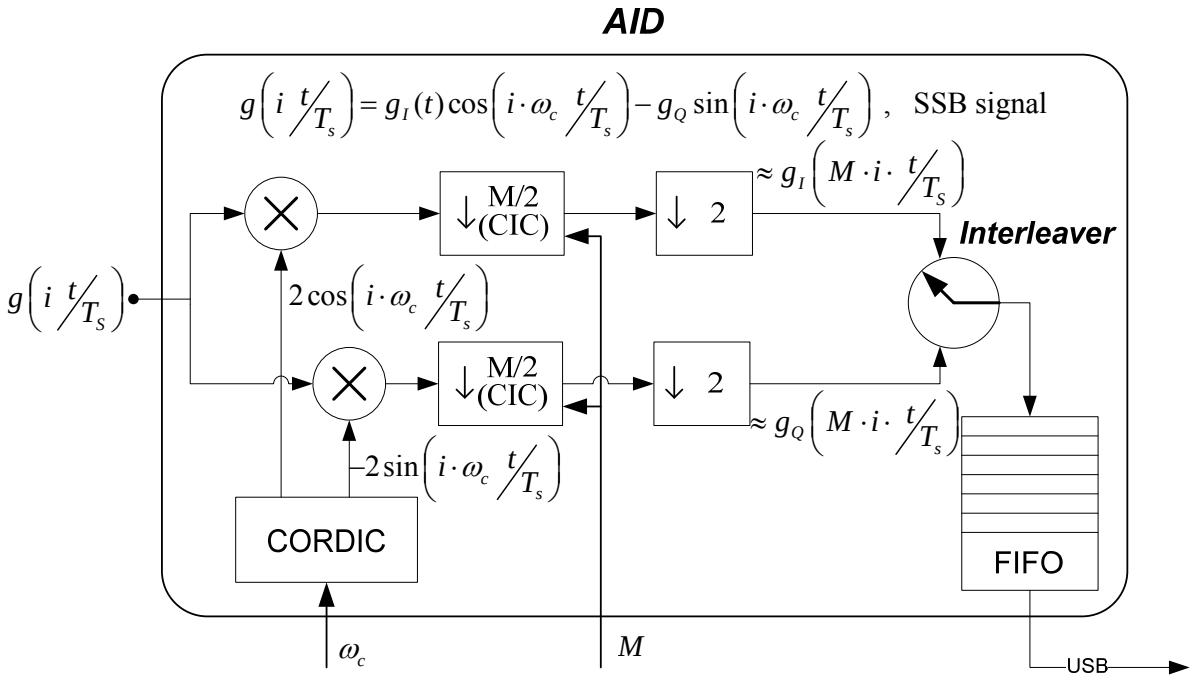


Figure 11. FPGA digital down-converter implementation.

Recall that the mixer, down-sampling and low-pass filtering are all implemented on the FPGA chip on the AID. The user can control the mixer frequency, $f_c$, and the decimation rate M. The low pass filter coefficients are automatically computed given the user choice of M, such that the cutoff frequency is ($64x10^6$/M/2).

The mixer is implemented via the CORDIC algorithm which multiplies incoming IF waveforms by a complex valued exponential [15]. Decimation, which includes down-sampling (dropping samples) and filtering, are done together in practical digital systems. One of the more efficient low-pass and down-sampling filtering algorithms is the Cascaded Integrator Comb (CIC) filter [16] which is implemented on the AID board. Lastly, there is a First-Input-First-Output (FIFO) buffer between the USB controller and FPGA to manage the data transfer. The FIFO depth is not programmable by the user.

The FGPA code was designed to operate in multichannel mode with two transmit and two receive channels. The AID only needed to operate one receive channel. This allowed us to implement a single channel version of the original FGPA code. Our single channel design is much smaller, and fits on an FPGA that is cheaper and consumes less power than the original FPGA. The FPGA used is a Cyclone EP1C6, which contains 5980 logical elements.

### 3.2 Hardware Performance

There are number of factors that influence the performance of the device, e.g., the sampling frequency of the ADC, the noise figure of the system, the noise introduced into the system due to clock jitter, the anti-aliasing filter, and other aspects. The best measure of board performance is the overall *noise figure*. The noise figure of the AID was estimated to be around 18 dB. The single best way to improve performance would be to get a better clock to drive the FPGA and ADC, however, the clock that was chosen was adequate to meet system specifications.

In order to characterize the performance of our IF-sampler, we measured the amount of folded noise in the band of interest that occurred as a result of sampling. To conduct this experiment, a live signal from GOES-12 was used. Figure 12 illustrates a plot of the captured signal. The blue line shows the frequency response of the LNB, which has a bandwidth of 50 MHz, while the green line shows the frequency response of the signal after being filtered with a SAW filter. The performance degradation due to folding was minimal.

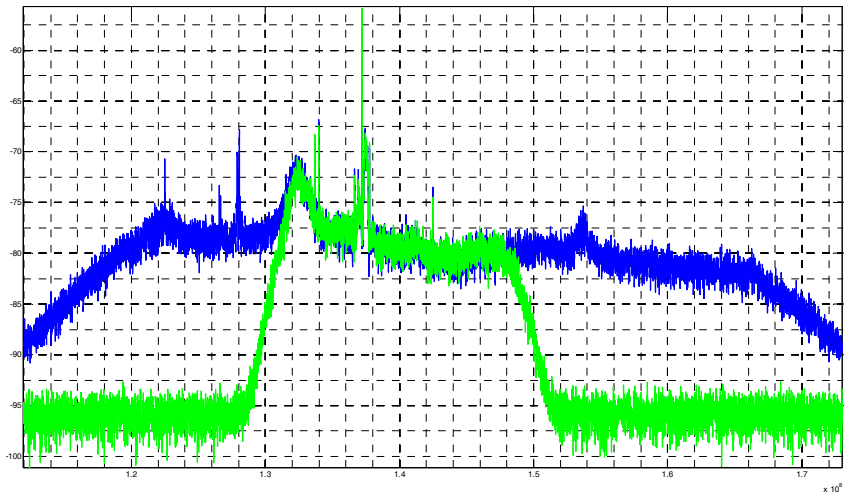


Figure 12. Spectrum of live GOES-12 signal.

# 4. Software Description

In this section, we describe the signal processing blocks designed to demodulate LRIT, EMWIN-I, and EMWIN-N. Throughout this section, we will reference the different file names of the source code used to implement the different processing blocks. For example, when a finite impulse response (FIR) filter-block is shown, it will referred to as *gr_fir_filter*, because that block is implemented using the file *gr_fir_filter.cpp*.

We begin this section by describing the general software framework used to design the EMWIN/HRIT software receiver in Section 4.1. We then describe in Section 4.2, a carrier acquisition algorithm that is performed before demodulating any communication signal After that we investigate the LRIT transmission specification and the LRIT software receiver architecture in Sections 4.3 and 4.4, respectively. The EMWIN-N and EMWIN-I are likewise treated in the subsequent sections. A brief description of the Soft-decision Viterbi decoder appears in Section 4.7.

## 4.1 Software Framework

As stated earlier, the software was developed with a free open source software development suite called GNU Radio [8]. The GNU Radio framework provides an environment that allows a simple interconnection of different C-written subroutines. The GNU Radio paradigm allows different programmers to write custom digital-communication blocks that can be easily configured and connected using basic C++ syntax. Once all the blocks are configured and connected, GNU Radio automatically streams data from beginning to end without the need for explicit user control. The user-controlled initial state of the software blocks can be found on Table 6.

The environment of connecting the blocks and streaming data in GNU Radio is resource-efficient, making GNU Radio a good framework for designing real-world radio applications. The main bottlenecks in GNU Radio come from the implementation of the actual communication blocks, for example, filtering, error correcting decoding, etc. In order to take advantage of the powerful GNU Radio framework but also produce an application with high performance, it is best for users to write their own communication blocks, or simply optimize the blocks that already exist in the GNU Radio library. In order to develop the EMWIN/HRIT solution, several GNU Radio-blocks were optimized and many new blocks were created in order to build a system that works according to the specifications. In many cases, blocks were written using the Intel® Performance Suite 5.1 [17]. This library allows many mathematical operations to be performed in parallel, using specific low-level Intel-processor instructions. Vector multiplications and matrix operations can be efficiently pipelined in order to increase the overall system throughput. The system diagram in Figure 13 shows the logic behind the software radio components of the EMWIN/HRIT solution. Each of the software-blocks that are part of this solution are described in the following sections.

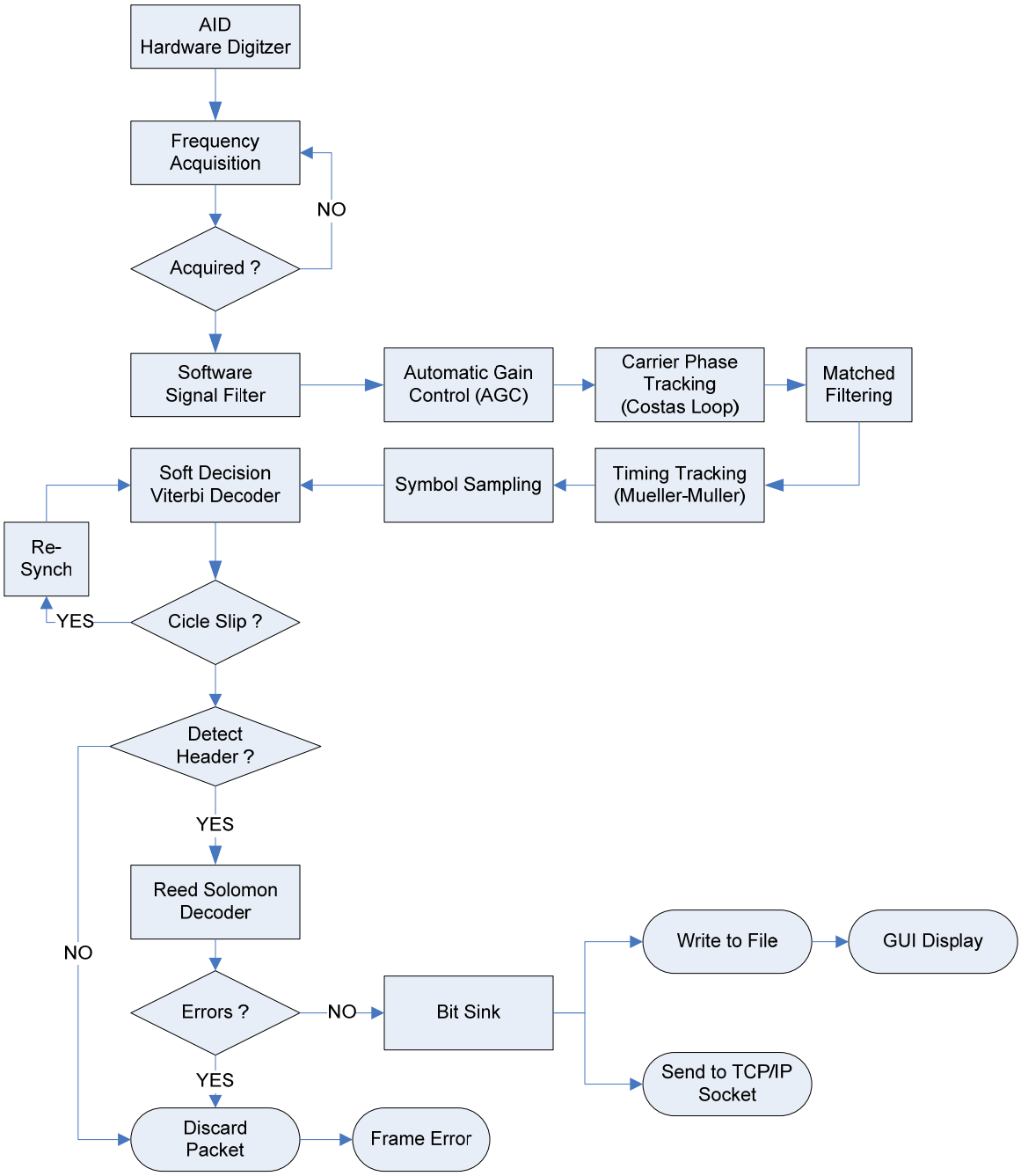


Figure 13. Overall EMWIN/HRIT system block diagram.

Table 6. AID User-Defined Input Parameters

| Parameter | Command Flag | Default Value | Description |
|---|---|---|---|
| Enable File Sink | -wr | 0 | Write raw USB samples directly to disk. Output file will be located in c:\temp\usrp_log\ |
| Decimation Rate | -dr | Depends on Signal range: [2-256] | Sets the decimation rate on the FPGA. See Table 5 |
| Center IF-Frequency | -fc | | For the LRIT Signal, default values is 137.5 MHZ, for EMWIN-I, 137.225 MHz, and for EMWIN-N is 137.9 MHz. 143.9 MHz for HRIT |
| Enable Software Filter | -lfilter | 1 | Enable software rejection filter |
| Repeat Incoming Waveform | -repeat | 0 | If the radio is run using a recorded user-file, this flag enables the receiver to repeat the incoming waveform once it reaches the end of file This is a useful feature for bit error rate testing |
| Roll off Factor | -roll | 0.5 Range: [.1 - .9] | Roll-off factor for square root-raised cosine pulse shaping |
| Bit Rate | -br | Depends on Signal | 293,883 for LRIT, 17970 for EMWIN-N, 9600 for EMWIN-I, 927,000 for HRIT. |
| Buffer Size | -buff | 64 [kbytes] | For multi-threaded applications, this parameter defines the size of the buffer between threads |
| Use Recorded Data? | -rec | 0 | Allows software radio to run reading a pre-recorded file stored on a network drive instead of using data from the USB port of the AID box. |
| Timing Bandwidth | -tbw | 3000[LRIT] 1000[EMWIN-N] | Indicates the factor that divides the symbol rate of the desired signal to determine the timing-loop bandwidth for LRIT or HRIT and EMWIN-N only |
| Phase Bandwidth | -pbw | 100[LRIT] 200[EMWIN-N] | Indicates the factor that divides the symbol rate of the desired signal to determine the phased-locked-loop bandwidth for LRIT or HRIT and EMWIN-N only |

## 4.2 Frequency Acquisition

The purpose of the block is to estimate the center frequency of a sampled communication waveform. The constituent blocks of the acquisition engine are shown in Figure 14. The core of this software engine is based on averaging the successive Fast Fourier Transforms (FFT) of the incoming signal. The frequency-search window (that depends on the initial frequency uncertainty), the bin-resolution, and the amount of bin-averaging are all runtime programmable. The command-line frequency acquisition options and default values are described in Table 7. A fourth-law detector has been provided to lock onto the BPSK subcarrier. For the offset quadrature phase-shift keying (OQPSK) case a square-power detector is used. The search window size and bin resolution values determine the FFT size.

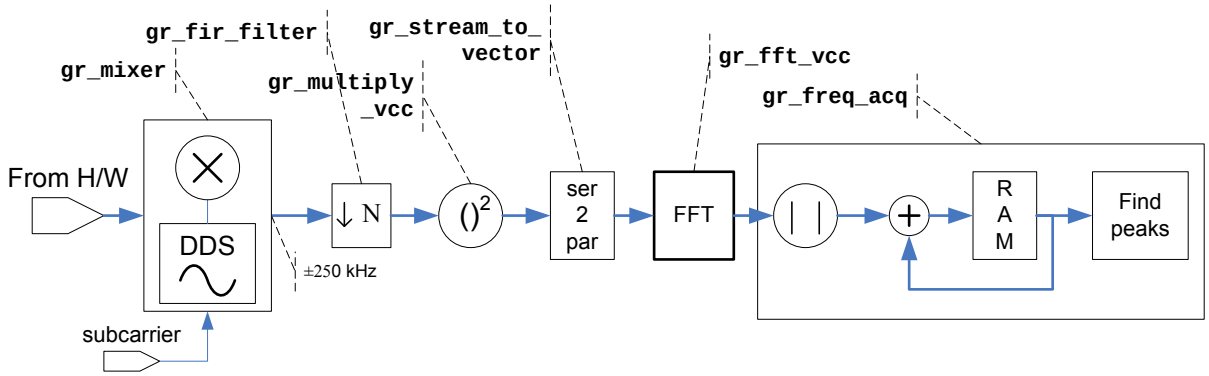


Figure 14. Block diagram of software-based frequency acquisition.

Table 7. Configuration for Frequency-Acquisition Subroutine

| Parameter | Command Flag | Default Value | Description |
|---|---|---|---|
| Enable / Disable Freq.Acq | -acq | 1 | Frequency acquisition is enabled by default. |
| Bin resolution | -fa_res | 50 Hz | Frequency bin-resolution |
| Freq. Range | -fa_range | 50 kHz | Initial frequency uncertainty to search for |
| Detection Rule<br>Square-Law<br><br>4th order Law<br><br><br>Absolute Power | <br>-fa_2<br><br>-fa_4 | <br>0 or 1<br><br>0 or 1 | <br>Depends on signal of interest and SNR. Default method for OQPSK.<br>Depends on signal of interest and SNR. Default method for BPSK.<br>f Square and 4th Power are disabled, then this method is enabled. Used in some low SNR scenarios. Default for FSK. |
| FFT Averages | -fa_numfft | 16 | Number of times FFT algorithm is run to measure the frequency content of a specific bin |
| Forced Frequency Offset | -foff | 0 | User can set a desired frequency offset that adds to the value that the frequency algorithm obtains. Or one can turn acquisition off, and enter in a known offset manually. |

A peak-search algorithm is run once the desired number of averaged FFTs have been computed. The acquisition engine will return a set of bins whose integration-value determine the likelihood of the signal of interest having frequency contents on that particular bin. Each detected peak will therefore have an associated power metric. Let $\vec{P}_0(N)$ be the length-N vector of measured power levels. The acquisition engine creates a vector of sorted metrics $\vec{P}_s(N)$ where, without loss of generality, we assume that $P_s(0) \le P_s(1) \le \cdots \le P_s(N)$. The peak-frequency is determined by computing the center of mass for the largest bin and the adjacent eight bins. Let $P_0(Max)$ be the bin in the un-sorted power metric vector that is mapped to the sorted metric $P_s(N)$. The power metric can then be defined as:

$$P_m = \frac{\sum_{i=-4}^{4} P_0(Max - i)}{\frac{4}{N}\sum_{i=0}^{N/4} P_s(i)}$$

The denominator can be thought of as the noise power, while the numerator is the signal power. Recall that when a BPSK signal of the form $\pm Ae^{j\omega_c t}$ is sent through a square-law detector, the output is ideally an impulse function. This implies that in order to measure the signal power one would ideally require summing over only a single bin on the numerator. However, due to the imperfect size of the FFT-bins, our experiments show that considering eight adjacent bins is a better indicator of the true signal level. At low SNRs, squaring or quadrupling the value of a particular bin can have severe squaring-losses. Therefore the SNR operating point must be considered before choosing a specific frequency-acquisition method.

## 4.3 LRIT Transmitter Specifications

The digital LRIT is an international standard for data transmission that was developed by the Coordination Group for Meteorological Satellites (CGMS) in response to the recommendation for digital meteorological satellite broadcasts. The CGMS Global Specification provides the standard that is supported by all operational geostationary meteorological satellites to be flown by the United States, European agencies, Japan, China, and Russia. The NOAA and other world meteorological agencies have developed subsequent system specifications, designs, and implementations of their specific LRIT systems. The origin of the name LRIT is due to its initial low data rate of 293 ksps.

The full LRIT specification can be found at [5][6]. The packetization process of LRIT data follows a standard developed by the Consultative Committee on Space Data Systems (CCSDS). Below is a summary of steps for converting image data into an LRIT waveform.

### 4.3.1 LRIT Transmission Process

- Compress image data with Rice coding [18]
- Add CRC (Cyclic Redundancy Check)
- Create Virtual Channel Data Unit (VCDU) data payload out of compressed image data
- Assemble Coded Virtual Channel Data Unit (CVCDU) packet
  - Reed Solomon encode VCDU data payload
  - Add VCDU deader
- Assemble the Channel Access Data Unit (CADU) packet
  - Randomize CVCDU packet
  - Add synchronization header
- Viterbi encode CADU packets continuously without termination
- Modulate Viterbi encoded data using BPSK modulation

For a more detailed description of the transmission process, please see the specification. [5][6]. In Table 8, key LRIT signal parameters are presented.

Table 8. Key LRIT Transmission Parameters

| Parameter | Value |
|---|---|
| Satellite Service | GOES-11 (West), GOES-12 (East), GOES-13, GOES-14 |
| Modulation | BPSK |
| Center Frequency | 1691 MHz |
| Typical (140 MHz) IF Frequency | 137.5 MHz |
| Measured Symbol Rate | 293,883 symbols/second |
| Pulse Shape | Root Raised Cosine (α= 0.5) |
| Forward Error Correction Encoder | ▪ Type: Convolutional (Soft-Decision Decoder)<br>▪ Rate: ½<br>▪ Constraint Length: 7 bits<br>▪ Generator:<br>G1 = 1111001; G2 = 1011011<br>▪ Symbol Inversion: none<br>▪ Puncturing: none |
| Synchronization Header | 1ACFFC1D |
| Randomizer Polynomial | $h(x) = 1 + x^3 + x^5 + x^7 + x^8$ |
| Reed Solomon | (255, 223) code symbol interleaving I = 4 |
| CRC Check Polynomial | $g(x) = 1 + x^5 + x^{12} + x^{16}$ |

### 4.4 LRIT Software Receiver Architecture

The steps required to process an LRIT file from the transmitted data are shown in Figure 15. As stated earlier, the first step to demodulate LRIT is carrier acquisition. After carrier acquisition, the signal is low-pass filtered to reject all adjacent signals. The signal rejection filter can be turned on and off using the flag specified in Table 6. After bringing the signal down to baseband and filtering out interference, phase tracking, matched filtering, timing recovery, and convolution decoding via the Viterbi algorithm are performed. The data link layer takes the output of the Viterbi decoder consisting of blocks of 8160 bits, with each block being preceded by a 32 bit header for synchronization. These 8160 data bits have been randomized to assure sufficient bit transitions using the polynomial $h(x) = 1 + x^3 + x^5 + x^7 + x^8$. The de-randomized bits form a Coded Virtual Channel Data Unit (CVCDU) as shown in Figure 15. The first 892 octets of the 1020 octets in the CVCDU correspond to the header and data bits of a systematic (255,223) RS code [20]. By using a systematic code, the 892 octets are not modified by the RS channel code. This implies that under high SNR values the VCDU data could be used without RS decoding. The network layer forwards the RS decoded packets to the transport layer that takes care of reassembling the LRIT files that were subdivided before transmission. The session layer is next, where the CRC field checks for packet integrity using the polynomial $g(x) = 1 + x^5 + x^{12} + x^{16}$. The presentation layer will finally convert the decompressed LRIT files into user data files. The type of files that will be received include: image data files, service messages, alphanumeric files and Global Telecommunication System (GTS) messages.

The BPSK signal of interest is defined by the bit rate, $f_b$, and the subcarrier frequency, $f_c$:

$$X(t) = m(t, f_b)\cos(2\pi(f_c + \Delta + f_0)t + \phi(t))$$

where $f_0$ is a frequency offset introduced by the channel, $\Delta$ is the difference between the transmitter and receiver frequencies, $\phi(t)$ is a phase offset and $m(t, f_b)$ is the information message to be recovered. A conventional BPSK demodulator circuit with separate phase and symbol tracking loops is used for demodulation, as shown in Figure 16. The signal at the input of the software radio has

already been frequency-shifted from $f_c$ to baseband and decimated in the digitizer to reduce the computational requirements:

$$Y(t) = m(t, f_b)\cos(2\pi(f_0 + \Delta)t + \varphi(t))$$

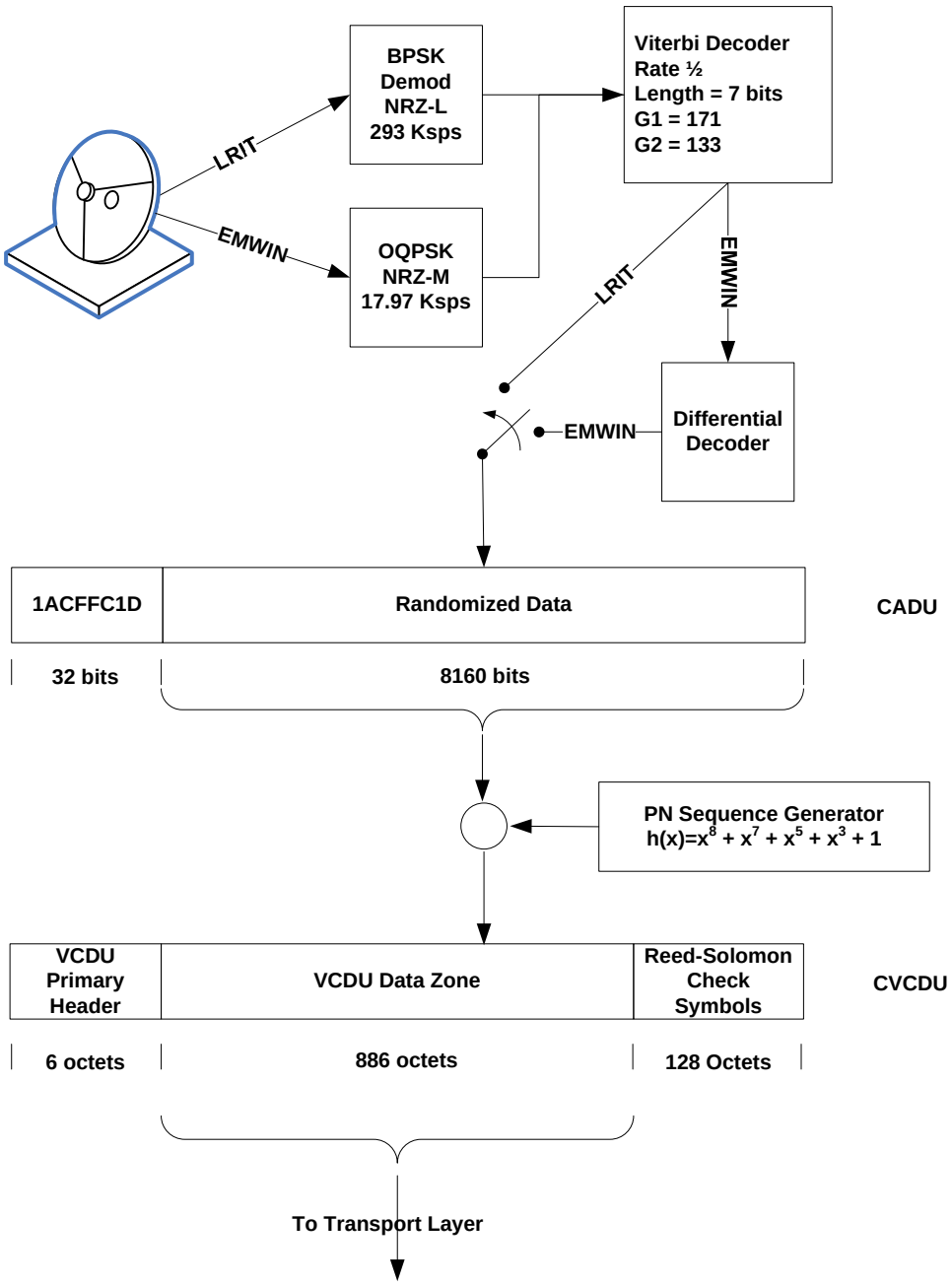


Figure 15. The LRIT and EMWIN-OQPSK packet structure.

An optional software-decimator can be added to further reduce the bit rate. For this particular implementation, we only use the decimator at the digitizer and use the data rates shown in Table 5. An additional low-pass software filtering stage is added to the input of the radio to remove undesired neighbor frequency contents.

The LNB from the hardware interface translates the incoming frequency of the LRIT signal, from 1691 MHz down to $f_c = 137.5$ MHz. However, due to the sub-sampling method used, the effective center frequency is $f_c' = 9.5$ MHz, giving $\Delta_{\max} = (9.5 \times 10^6)(200 \times 10^{-6}) = 1.9$ kHz, assuming a 200 ppm frequency tolerance on both the transmitter and receiver oscillator references. Phase tracking by means of a Costas loop [19], can be applied to remove the dynamic frequency offset, $f_0$, introduced by the channel and the offset due to unmatched reference clocks. Following this stage, a filter matched to the root raised cosine pulse shape is used for maximizing the SNR in the presence of additive stochastic noise. A Mueller-Müller timing error detector [26] is used to recover the symbol timing of the signal.

Once the timing properties of the signal have been recovered, a convolutional Viterbi decoder is used for error correction. Building a software Viterbi decoder that was able to handle the projected data rates of 900 ksps was one of the greatest challenges in this project. Since one of the ideas of GNU Radio is to be able to reuse signal processing blocks, we chose to build a flexible decoder that could handle any constraint length, rate and polynomial choice. Since the full specifications of the future GOES broadcast signal are not completely defined, we decided that the practical advantages of having a flexible decoder were better than designing a specific decoder for a particular channel code rate that could perhaps attain a higher throughput. Our initial design was able to maintain a throughput of around 300 ksps. Since this was not sufficient to handle the desi red rates, we carefully vectorized the algorithm using Intel IPP functions [17]. This redesign allowed us to increase our throughput to 1.89 Mbps when running the decoder on a dual processor Intel Xeon CPU at 3.7 GHz. In Figure 16, the signal flow in the receiver is shown, together with the name of GNU Radio blocks that are used in the implementation. Note that, for the case of the Viterbi decoder, the block has been labeled *ar_viterbi* to indicate that this block was never a member of the GNU Radio library. The remaining blocks use the prefix *gr*, to indicate that they have been derived from the original GNU Radio library. This does not imply that these blocks remain unchanged from their original versions. In fact, all of them have been optimized using the Intel IPP libraries to enhance their throughput. The final signal-processing stage is a threshold slicer that generates output bits that are fed to the data link layer for image extraction.

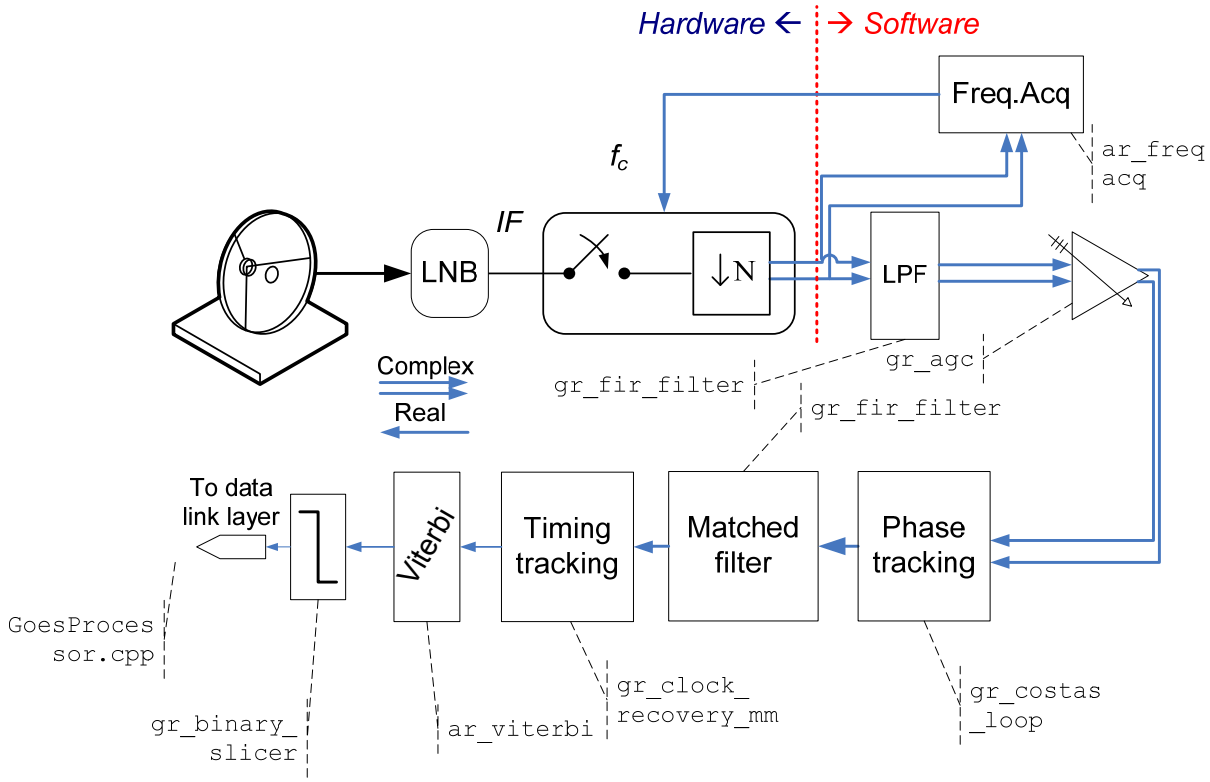


Figure 16. Block diagram showing the different signal processing blocks in an LRIT BPSK receiver.

Table 9. Soft Decision Viterbi Decoding Parameters

| Parameter | Command Flag | Default Value | Description |
|---|---|---|---|
| Minimum Threshold | -vthresh | 0.1 | Minimum bit error rate at the output of the Viterbi block, to declare that the decoder has solved its phase ambiguity |
| Test for Cycle Slip | -vperiod | 10 | Indicates number of blocks before the Viterbi subroutine verifies that it is locked to the correct phase. For BPSK there are 2 possible states only. For OQPSK there are four. |

## 4.5 EMWIN-N Transmission Specifications

The full EMWIN specification can be found at [3][4]. Below is a summary of steps for converting EMWIN products into an EMWIN waveform.

### 4.5.1 EMWIN-N Transmission Process

- Generate gif, jpg, zip, and txt products
- Create VCDU data payload out of EMWIN products
- Assemble CVCDU packet
  - Reed Solomon encode VCDU data payload
  - Add VCDU header

- Assemble CADU packet
  - Randomize CVCDU packet
  - Add synchronization header

- Viterbi encode CADU packets continuously without termination

- Apply binary differential encoding on the Viterbi encoded data

- Modulation data using Offset Quadrature Phase Shift Keying (O-QPSK)

Table 10 summarizes the key EMWIN-N signal parameters.

Table 10. EMWIN-N Key Parameters

| Parameter | Value |
|---|---|
| Satellite Services | GOES-13, GOES-14 |
| Modulation | OQPSK |
| Center Frequency | 1692.7 MHz |
| Typical (140 MHz) IF Frequency | 139.2 MHz |
| Measured Symbol Rate | 17970 symbols/second |
| Pulse Shape | Root Raised Cosine (□= 0.5) |
| Forward Error Correction Encoder | Type: Convolutional (Soft-Decision Decoder)<br>Rate: ½<br>Constraint Length: 7 bits<br>Generator:<br>G1 = 1111001; G2 = 1011011<br>Symbol Inversion: none<br>Puncturing: none |
| Binary Encoding | Differential Encoding |
| Synchronization Header | 1ACFFC1D |
| Randomizer Polynomial | $h(x) = 1 + x^3 + x^5 + x^7 + x^8$ |
| Reed Solomon | (255, 223) code symbol interleaving I = 4 |

### 4.6 EMWIN-N Software Receiver Architecture

The EMWIN software receiver described herein was developed to satisfy GOES-N broadcast requirements. The user data format is shown in Figure 15. OQPSK modulation is a variant of phase-shift keying modulation using four different values of the phase to transmit information across a channel [19][21]. Taking four values of the phase (two bits) at a time to construct a QPSK symbol can allow the phase of the signal to jump by as much as 180° at a time. When the signal is low-pass filtered (as is typical in a transmitter), these phase-shifts result in large amplitude fluctuations which has an undesirable effect in communication systems. By offsetting the timing of the odd and even bits by half a symbol-period, $T/2$, the in-phase and quadrature components will never change at the same time. This will limit the phase-shift to no more than 90° at a time which yields much lower amplitude fluctuations than non-offset QPSK. The demodulation chain is shown in Figure 17.

The $T/2$ fixed symbol timing offset is removed after the frequency/phase has been completely recovered by the Costas loop. As shown in Figure 17, the phase/frequency corrected signal gets split into a parallel I-Q data stream. The I-data is then delayed by $T/2$, and then recombined with the Q-data to form an offset free phase tracked QPSK signal. Because the data has been recombined, the timing tracking and match filtering blocks operate on complex I-Q data, instead of working on I-data and Q-data independently.

Another difference with Figure 16 is that the in-phase and quadrature signals require a de-interleaving block prior to their Viterbi decoding. This de-interleaving works by taking the in-phase vector $\vec{I} = \{I_0, I_1, I_2, \ldots\}$, and the quadrature vector $\vec{Q} = \{Q_0, Q_1, Q_2, \ldots\}$ and combining them into a single vector, $\vec{R} = \{I_0, Q_0, I_1, Q_1, I_2, Q_2 \ldots\}$. Finally, since the EMWIN-N bits are differentially encoded, a differential decoding stage is added before the hard-decision detection at the binary slicer. After slicing, the binary data goes to the data link layer processor for header synchronization and Reed-Solomon decoding.

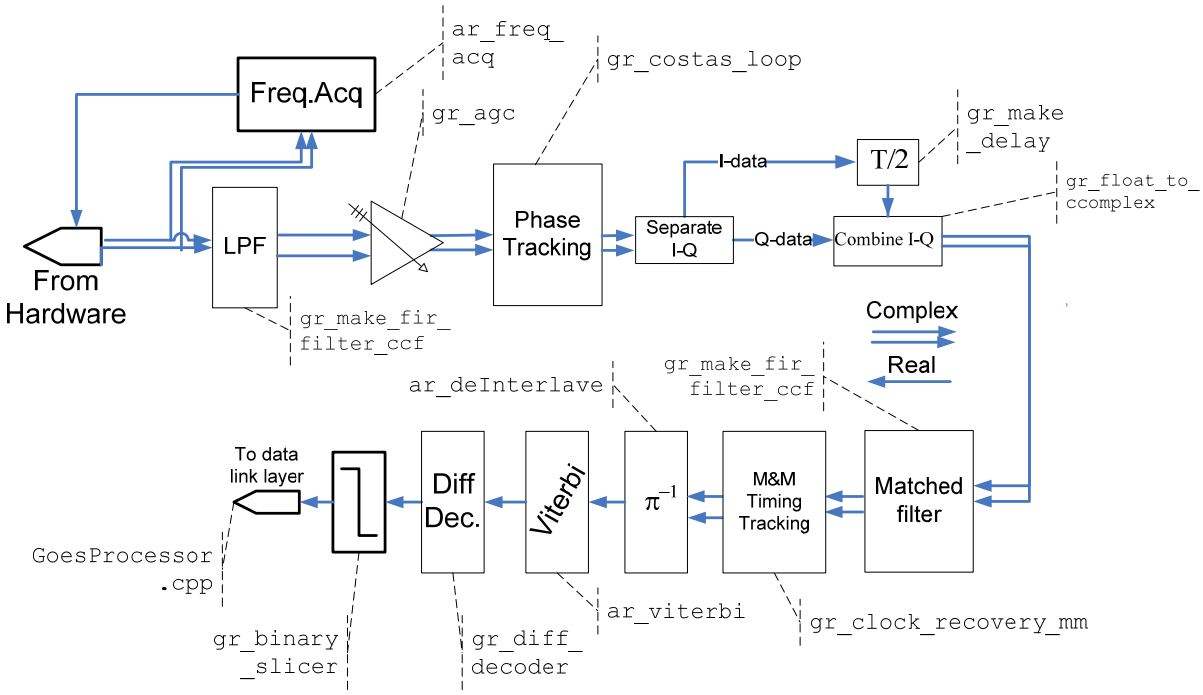


Figure 17. Block diagram showing the different signal processing blocks in an EMWIN OQPSK receiver.

### 4.7 Soft-Decision Viterbi Decoder

A convolutional code is a type of error-correcting code in which each an m-bit information symbol is encoded into an n-bit symbol, where m/n is the code rate. The transformation is a function of the last k information symbols, where k is the constraint length of the code. Several algorithms exist for decoding convolutional codes. For relatively small values of k, the Viterbi algorithm is universally used as it provides maximum likelihood performance and is highly parallelizable. Viterbi decoders are thus easy to implement in digital integrated circuits and in software.

Low complexity high-throughput Viterbi decoders utilize hard-decisions to recover the information bits. However, the decoder performance suffers 3 dB loss when using hard decisions. In order to recover the 3 dB performance loss while maintaining high throughput, quantized soft-decisions are used instead of full 32-bit precision floating point numbers. Our soft-decision Viterbi decoder quantizes the 32 bit soft-decisions into 3-bit numbers (8-levels). This quantization technique allows the decoder to avoid using floating point multiplications, which significantly improves speed while still maintaining a high coding gain. Vectorized manipulation of data using the Intel Performance Primitives library also contributed to increasing the overall throughput.

The EMWIN-N, LRIT and HRIT protocols all use a rate ½ (where for each transmitted bit, an additional parity bit is transmitted) convolutional code whose parameters are described in both Table 8 and Table 10.

Viterbi decoding is more computationally intensive than all other processing blocks combined. In order to maintain a high throughput, the Viterbi decoder had to be implemented on one thread while the BPSK demodulation was implemented on a second thread. By doing this, one can take advantage of dual core CPUs, and the total system throughput is equivalent to the throughput of the slowest thread alone. Splitting the Viterbi decoder onto a separate thread is not necessary for the EMWIN-N signal due to its slow data rate.

### 4.8 EMWIN-I Signal Specifications

The EMWIN software receiver described in this section corresponds to a legacy FSK signal developed to satisfy GOES-I broadcast requirements in [22]. This data stream consists of National Weather Service (NWS) weather products and other data files. Each product or file, whether ASCII text or binary data, is divided into 1 Kbyte packets and sent as a series of asynchronous 8-bits, with no parity bits, one start and one stop bit:(8,N,1)[23]. The asynchronous bit-stream follows the RS-232 standard. This standard specifies that an 8-bit data byte is preceded by one start bit (0), and is ended with an idle sequence, which is represented by sequence of stop bits (1). As an example, consider the representation of 3 bytes of data.

A synchronous binary representation of **0** base 10**, 255** base 10, and **170** base 10, is

00000000111111111010101 0.

An example of an asynchronous binary representation of the same three bytes is

**0**00000000**1111110**11111111**10**10101010**111**.

For easy readability, the start bits and stop bits (idle sequences) are in bold red.

There is always one start bit before every byte, however the length of the stop bit sequence (idle sequence) is indeterminate. The byte representing **0** base 10 begins with a start bit, but continues with an idle sequence of length 6, instead of just a single stop bit. After that idle time, there is a start bit, then the byte representing **255** base 10, and then a single stop bit. After that there is another start bit, the byte representing **170** base 10, and finally another idle sequence of length 3. When parsing the data for text files, the asynchronous nature of the bit-stream must be accounted for.

The EMWIN-I satellite broadcasts are transmitted as asynchronous $(9600,8,N,1)$. The FSK signal has no formal CCSDS packet structure and no error correction codes. The signal has a rudimentary packet structure that has a header marker of 6 bytes of zeros. The header follows the same asynchronous principles as the actual data, so there is a start bit followed by an indeterminate idle sequence after each byte of zeros. Further details on the specification of this signal can be found [24].

Sometimes EMWIN-I is described as Direct FSK or DFSK, not to be confused with differential FSK. The term DFSK is obsolete and the type of modulation used by EMWIN-I is simply referred to as FSK by today's standards.

Table 11. EMWIN-I Key Parameters

| Parameter | Value |
|---|---|
| Satellite Service | GOES-11 (West), GOES-11 (East) |
| Modulation | FSK |
| Center Frequency | 1690.725 MHz |
| Typical (140 MHz) IF Frequency | 137.225 MHz |
| Measured Symbol Rate | 9600 symbols/second |
| Frequency Separation | 3600 Hz |
| FEC | none |
| Synchronization Header | 0x**000000000000** (hexadecimal) |
| Randomizer Polynomial | $h(x) = 1 + x^3 + x^{20}$ (V.36) |

## 4.9 EMWIN-I Software Receiver Architecture

Frequency-shift keying (FSK) is a modulation scheme that transmits information through discrete frequency changes of a carrier wave [25]. A binary FSK signal can be defined by two tone frequencies: $f_0$ and $f_1$, corresponding to '0' and '1' bits. The demodulator is designed to support programmable tone frequencies.

The tone frequencies may be rather large, while the symbol rate is typically very small. The Nyquist sampling [19] requirement dictates a sampling rate, $f_s$, of at least $2|f_1|$ to prevent aliasing. System throughput and efficiency can be improved by translating the tones such that they are centered around DC. The translation is accomplished by first mixing the received signal with a tone at a frequency of $f_c = 1/2(f_1 + f_0)$. The new tones are then given by: $f_0' = 1/2(f_0 - f_1)$ and $f_1' = 1/2(f_1 - f_0)$, as shown in Figure 18.

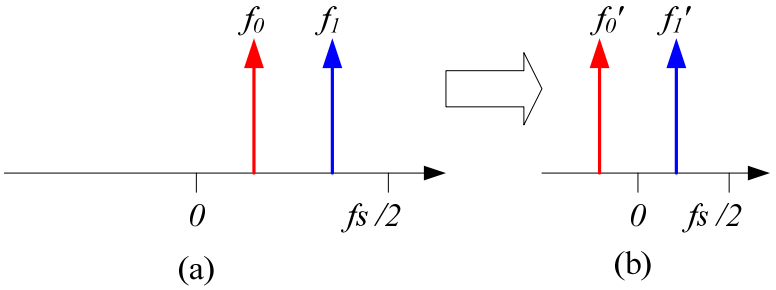


Figure 18. Down-converted FSK Frequency Diagram

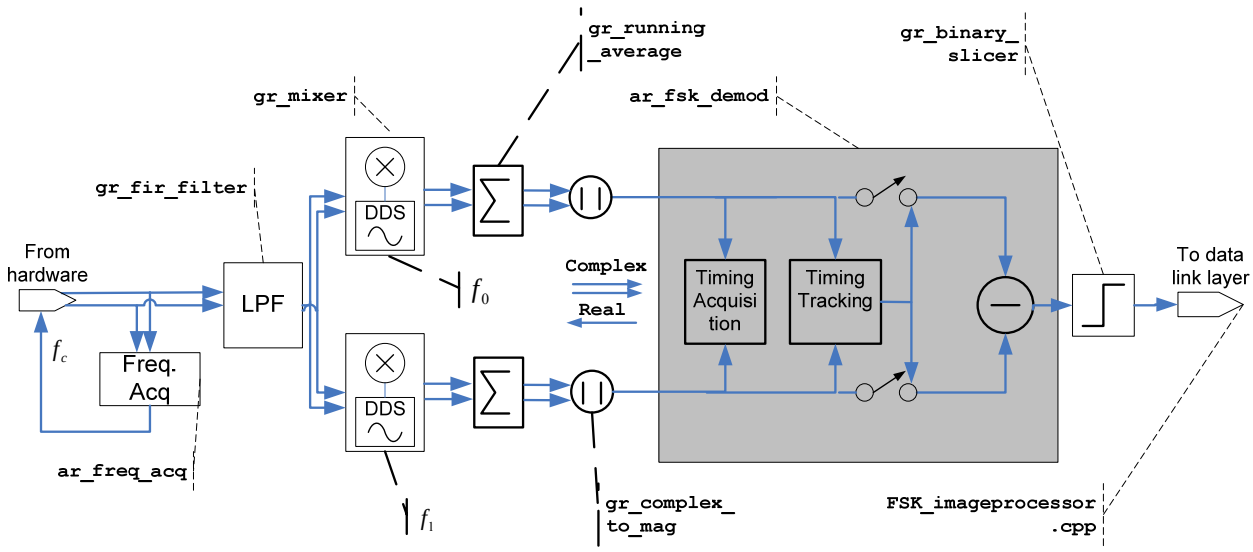


Figure 19. Block diagram showing the different signal processing blocks in an EMWIN-I FSK receiver.

The FSK demodulation block diagram is shown in Figure 19. The center frequency $f_c = 1/2(f_1 + f_0)$ is estimated by the carrier frequency acquisition block as shown in the figure. A low pass filter is applied to reject out-of-band interference after bringing the tones down to baseband. For simplicity, the baseband tones will be referred to as $f_0$ and $f_1$, instead of $f_0'$ and $f_1'$ for the remainder of the discussion.

From there, the data is correlated using two locally generated tones, corresponding to $f_0$ and $f_1$. Both correlation output streams go through a running average block, where the averaging window period is the width of a symbol period. The running average is not an integrate and dump process. If there are 100 samples in there will be 100 samples out. After the running average, the data goes through a magnitude block which converts the data from complex to real. These two real-valued data streams then go into an acquisition and tracking engine for further processing.

The timing acquisition and timing tracking blocks that reside inside the *ar_fsk_demod* block are described in sections 4.9.1 and 4.9.2 respectively. The acquisition block acquires the initial time offset, while the symbol timing tracking block maintains timing lock. The output of the timing tracking block is an estimate of the timing error, *err* as shown in Figure 21. The timing error drives the samplers for both the $f_0$ and $f_1$ correlation output streams. Each output of the sampler represents the total energy over the $f_0$ and $f_1$ bases over an entire symbol period. If the difference of these two outputs is positive, then the correlation against $f_0$ is larger than the correlation against $f_1$, and thus the bit slicer would produce 0.

The bit decisions are sent to a data-link layer block for further processing, which is defined in *fsk_imageprocessor.cpp*. The first step in data processing is descrambling the asynchronous FSK data bits.

The scrambler used is similar to the one in the International Telecommunication Union (ITU) V.35 standard (currently replaced by ITU-V.36)[27]. The polynomial used for scrambling is $h(x) = 1 + x^3 + x^{20}$ and, unlike the V.35 recommendation, no circuit to detect an adverse state is implemented. Note that unlike the scrambler shown in Figure 15, which can actually be thought of as a data-independent bit mask, the EMWIN-FSK descrambler is a data dependent shift register. After the data has been properly descrambled, a header consisting of six asynchronous null-bytes (with its corresponding start and stop bits) indicates the beginning of the data field. After synchronizing the

FSK data, the packet is parsed, and the appropriate data is written to disk or sent over a socket according to the EMWIN-I specification.

### 4.9.1 Symbol Timing Acquisition

The FSK symbol timing is initially acquired by looking for a "01" sequence transition [26]. The outputs of the two running average blocks (for $f_0$ and $f_1$) are monitored over the duration of two symbol periods. For each sample on each correlation output stream, the current value is compared to the maximum value seen so far. If a new maximum is found, three values are recorded:

1. New maximum.
2. Current sample number, relative to some arbitrary starting point.
3. Value of the other channel.

Referring to Figure 20 when one of the channels is at the maximum, the other channel should be at the minimum. Assuming a "01" sequence, there will be one large value for each channel. Let the maximum value for channel 0 be A, while the value for channel 1 at the same time is B. Likewise, let the maximum value for channel 1 be C, while the value for channel 0 at the same time is D. A lock condition is declared if the following four conditions are met:

1. $A > \beta, C > \beta$
2. $A > \alpha B$
3. $C > \alpha D$
4. $0.87 < A < 1.15C$

The value, $\beta$, is set to the number of samples/symbol squared and divided by 4. Condition-1 requires that the peaks be high enough such that if the input AGC is set to 1.0, at least $1/4$ of the power is in each peak. The value of $\alpha$ is a programmable threshold, greater than 1. Conditions 2 and 3 require that the peak be at least $\alpha$ times larger than the trough. Condition-4 requires that the peaks on channel 0 and 1 be approximately the same. Once the lock condition is satisfied, the optimal sampling point is computed by taking the average of the offsets where the maximum values occurred for channels 0 and 1.

If a lock is not detected for the observation window, the window is shifted by one full symbol and the operation is repeated. The actual symbol detection is started from the next symbol. The initial sample counter is set to T-x, such that a new symbol is declared $x$ samples after the current time, where T represents the symbol duration. The value of $x$ is estimated by averaging the times of the two peaks, $p_0$ and $p_1$ as $x = 1/2\,(p_0 + p_1 - T)$.

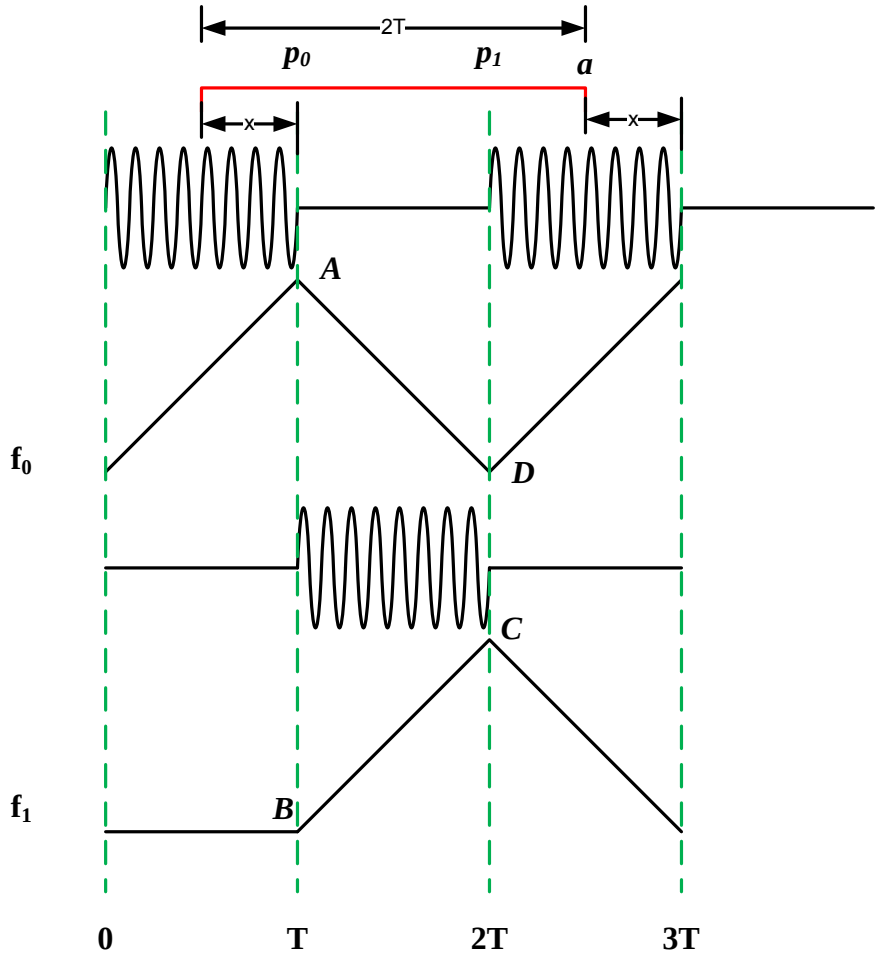


Figure 20. Detailed diagram of FSK timing acquisition block.

### 4.9.2 Symbol Timing-Tracking

A classic early-late timing tracking structure is used for the EMWIN-I tracking mechanism loop as shown in Figure 21 [19]. In an FSK signal, at any given time, if one tone correlation output represents signal, the other output represents noise. The loop should be driven by signal, not noise. Because of that, the first operation of the timing tracking loop is to select which correlation output represents signal, and drive the loop with just that output. The output of the loop, *err* , drives the sampler as shown towards the end of Figure 19.

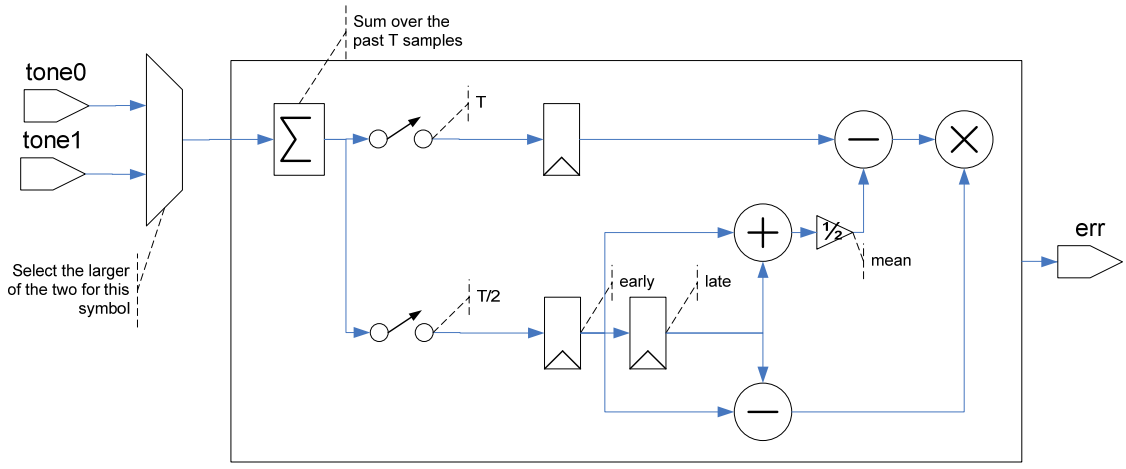


Figure 21. Detailed diagram of FSK symbol timing block.

## 4.10 The Data Link Layer

After all four data signals, EMWIN-I, EMWIN-N, LRIT, and HRIT, are demodulated and decoded, they are sent to a data-link layer processor, as shown in Figure 16, Figure 17, and Figure 19 respectively, for final processing. This processing layer begins by synchronizing the data to a header sequence, and ends by either writing the LRIT and EMWIN file products to disk, or by sending bytes of data over a socket for further processing using a separate application.

### 4.10.1 EMWIN-N, HRIT and LRIT CCSDS Packet Processing

The data–link layer process is very similar for EMWIN-N, LRIT, and HRIT because they use a similar CCSDS packet structure as shown in Figure 15. The data file that processes these CCSDS packets in our C++ software solution is GOESProcessor.cpp. The first step of the process is synchronizing the frames to the 32 bit CADU header. After synching to the CADU header, the payload is de-randomized and then RS decoded. The RS decoder determines whether the decoded packet is error free if it still has errors, in which case it is discarded.

**4.10.1.1.1 CADU Frame Synchronization.** CADU frame synchronization is performed by sliding a window of 32 bits and checking the sequence against the CADU packet header. If a bit sequence that matches the header is found, then frame synchronization is declared. In an ideal world, one would force all 32 bits to match the CADU header sequence in order to declare that the frame is synched and valid, however, this could cause many packets to be missed under high noise conditions. It is important to set a suitable frame-synchronization threshold (lower than 32 bits).

Let the number of differences between the header sequence and a portion of the data sequence be denoted by

$$s = \sum xor(\vec{r}(1,\ldots,32), 0x1ACFFCID)$$

The 32-bit section of the data sequence that is being evaluated for frame-synchronization is represented as $\vec{r}(1,\ldots,32)$. The summation represents the total number of differences between the 32 bit section of processed data and the ideal header. For instance, if the 32 bit section of data differs in 5 of 32 places from 0x*1ACFFCID*, then $s = 5$. A frame is detected or missed based on the value of *s* and the frame-synchronization threshold: $th$.

$$\textit{Frame Synch Found:} \quad s \le th$$
$$\textit{No Frame Present:} \quad s > th$$

Over extremely adverse channels, some frame headers have $s = 20$ (which ultimately means that 20 out of 32 bits in the header are incorrect), yet the packet can still be decoded after RS decoding. To handle such cases, there are four states of frame-synchronization lock. The value $th$ varies based on the lock-state.

Lock State-0 is defined as being completely unlocked, and thus has the most stringent frame synchronization threshold requirement to prevent a false detection. On the other hand State-3 is defined as being ideally locked, and thus the least stringent threshold. On Figure 22 a synchronization state machine diagram is shown. Each state has a lock status number and its associated frame-synch threshold. The figure also shows the transition cases to go from one state to the next.

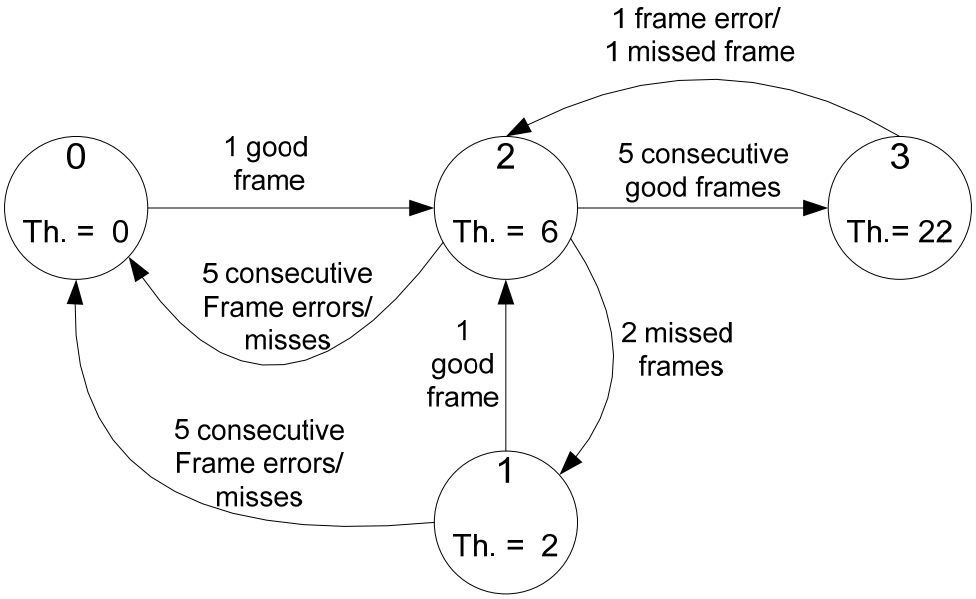


Figure 22. CADU Frame synchronization lock state graph.

The lock state remains unchanged until an event happens that causes a state transition. The receiver begins in State-0, and thus the 32-bit sequence must perfectly match the CADU header to declare a frame-synch. This stringent requirement remains in place until a frame is found and successfully decoded without error. From there, the state machine goes immediately to State-2. In this state, we assume that successive headers can be found with exact synchronization, i.e., exactly 1024 bytes away from the previous header. While in State-2, if five consecutive frames are found with uncorrectable errors in all five frames, then State-2 is downgraded back to State-0. If five consecutive frames are detected and decoded error-free, then the state machine is upgraded to State-3.

State-3 assumes that you are perfectly locked, and thus the threshold of tolerable errors is set to a very high number. The default number of tolerable errors in the header for this state is set to 22. The user can set its desired threshold for State-3 by setting the flag "–thresh x" at the command prompt, where "x" is the desired threshold. A command of "–thresh 32" would guarantee that all packets are forwarded to the RS decoder when in State-3. If a frame isn't found, then the lock state is downgraded to State-2.

Synchronization can be lost unexpectedly for a number of reasons, including RF impairments such as an abrupt frequency drift, fades in the received signal, etc. It is common for a packet to appear unlocked in severe noise conditions (due to excessive errors in the header sequence) but this does not always imply that the software receiver has lost lock. This is the reason why, for State-3, the threshold is set to a very high value to ensure that lock is maintained in high noise conditions. The State-machine can still detect when lock is truly lost since as soon as a single frame is found to be in error, State-3 is immediately downgraded to State-2, and from there, it can easily restart the state machine and go back to State-0.

When in State-2 or 3, the next frame in the transmission should be exactly 1024 bytes away. If a frame is not found exactly 1024 bytes ahead of the previous frame header, then the software radio advances 1024 bytes and searches for the next header in the exact location is expects to find it. No attempt is made to search in the middle of a frame while in-lock since it would reduce throughput and increase the chance of locking to a false header. If headers cannot be found in the positions they are expected to be in, the state machine moves to State-1.

The threshold in State-1 has a threshold of two errors allowed. With the threshold set to such a stringent level, it is safe to search byte by byte (in the middle of a packet) to try to find the next header. After scanning 3072 bytes (three packets) without success, the state machine assumes it lost all time reference and returns to State-0 to begin all over again. Note that two missed frames are pre-requisite to entering State-1, thus if an additional three frames are missed in State-1, then five total consecutive frames have been missed.

There is one additional caveat to the state diagram. The EMWIN-N signal has a differential decoder which removes the polarity ambiguity; however the LRIT signal does not have such a feature. When the LRIT receiver is in State-0, It will alternatively try to synch to the assumed header and its inverted (180°) version.

**4.10.1.1.2 RS Decoding.** Once the CADU frames are synched, they are sent to the (223,255)-RS decoder. As shown in Figure 15, the CADU payload contains 1020 bytes of data. The 1020 bytes are comprised of four interleaved RS frames of 255 bytes each. The RS decoder de-interleaves the four RS frames and checks to see if each of the four syndromes are correct. If one of the frames is in error, the entire CADU packet is dropped.

**4.10.1.1.3 VCDU Product Identification.** The first step after successful RS-decoding is to read the VCDU counter. The VCDU counter increments sequentially. If a frame is missed, the difference between the current VCDU marker and the previous VCDU marker is larger than one. An error is reported to the users if missed CADU frames are detected.

The following steps occur after obtaining the VCDU marker: read the APIDs, extract the file names, and write the files to the disk. In the case of LRIT, the extracted data file goes through a cyclic redundancy check (CRC). LRIT images are compressed and must be decompressed using Rice decompression before the images can be displayed.

Each CADU frame represents a single part of an EMWIN or LRIT product. Some products, like short text messages, require two parts, while other files, like EMWIN JPEG images, can require over 80 parts. Every part of the file contains file name information, the current product part number, and the total number of parts the product possesses. If subsequent parts of a given file (i.e., TRKINUS.GIF) are not received (i.e., if parts 13 and 15 are received, but 14 was never accounted for), we alert the user with an error, empty all the buffers that store the temporary files, and the complete file is discarded.

### 4.10.2 EMWIN-N Data Socket Output

The EMWIN/HRIT prototype solution has preliminary support for outputting data to a local socket. The intention for sending data over the socket is to allow third party vendors to develop the weather data user interface. The system has only been verified with one third party application: Wx Weather Message [28].

To activate the socket output, run the executable from the command line with the flag, -socket "*port number."* For example, if you want the solution to output data to port 18000, then run the executable with –socket 18000. Once the program executes, it will send a message stating that it has established a server at port 18000 and is waiting for a client to connect. At this point, the third party application should request a connection and begin interchanging data. The program does not have a time out period, so if no third party application is there to connect, the program will wait indefinitely.

The data sent over the socket is a continuous stream of raw data bytes. The bytes are not manipulated, and there is no advanced handshaking or network protocols used to transmit the data. Once the connection is initially established, the stream of bytes is sent over the socket at real-time signal transmission rates. If the information data rate for HRIT is 450 Kilo-info-bits/second, then the byte stream being sent over the socket will also be that rate.

The data cannot be sent over the web, or over a physical device port, or to a user defined IP address. The data is always transmitted to localhost IP address 127.0.0.1. The only flexibility is the port associated with localhost. That means that the third party software must be run on the same machine as the EMWIN/HRIT solution and must always establish a connection to localhost.

The data being sent over the socket is a portion of a CVCDU packet, which is shown at the bottom of Figure 15. This packet has three zones, the VCDU primary header, the VCDU data zone and the RS check symbols. The VCDU data zone is 886 bytes, however the first 2 bytes in the VCDU data zone are the Multiplexed Protocol Data Unit (M_PDU) headers as described in [6]. These 2 bytes are discarded and the remaining 884 bytes of the VCDU data zone are transmitted over the socket.

### 4.10.3 HRIT and LRIT Data Socket Output

LRIT and embedded EMWIN data can also be sent over a socket. Interaction between the EMWIN/HRIT prototype solution and a 3rd party LRIT GUI has not been verified.

Because LRIT possesses two distinct products, LRIT and EMWIN, our system utilizes two sockets. The flag, –socket "port number" will send the embedded EMWIN data over "port number", while the flag –socket_lrit "port number" will send the LRIT data over a different port number. Thus to send embedded EMWIN data over port 10000, and LRIT data over port 18000, one would set the flags to –socket_lrit 18000, -socket 10000.

The first 892 bytes of the CVCDU header is sent over the socket for the LRIT data. Every packet is sent over the socket, so it is up to the third party software to handle the data accordingly by reading the VCID and APID markers. Some of the packets will be fill packets, and some packets will contain EMWIN data.

The embedded EMWIN data will be handled exactly as described in the previous section. Unlike the LRIT data, interaction between embedded EMWIN data and 3rd party GUI software has been verified.

### 4.10.4 EMWIN-I Data Link Layer Processor

The EMWIN-I signal does not have CCSDS packets, and its transmission specification is explained, with references, in Section 4.8. The data link layer processor follows the transmission specification and is done in file FSK_Imageprocessor.cpp in the EMWN/HRIT solution. Because there is no advanced coding or CCSDS packet structure, there is no elaborate frame-synchronization process.

Unlike the HRIT and LRIT and EMWIN-N signal, the first step in the process is not frame-synchronization, but instead bit-level de-randomization. The output of the de-randomized data still contains asynchronous start bits and idle sequences. This asynchronous behavior must be accounted for when performing frame synch, and when processing the packet.

After bit-level de-randomization, the receiver must synch to the 6-byte null character header. If the 6-byte null header cannot be found, the receiver tries to synch to the inverted sequence in order to resolve the polarity ambiguity.

Once a header is found, the data is parsed for a file name, date and time, and most importantly, the check sum flag. The check sum flag that is sent is used for error detection. The check sum for FSK, as explained in [24], is the sum of all the ones within the 1116 byte FSK packet. Once the entire data sequence is processed, the sum of all the ones received is compared to the check sum flag transmitted at the beginning of the packet. If these numbers do not agree, an error is sent to the user and the file is not written to disk.

### 4.10.5 EMWIN-I Data Socket Output

The data socket handling protocols and options are identical to the HRIT and LRIT and EMWIN-N signals. The entire 1116 byte EMWIN-I FSK packet is sent over the socket, including null headers. Before sending data over the socket, the asynchronous 8-bit (8,N,1) format is removed, so there are no longer any start and stop bits encapsulating the EMWIN-I data.

# 5. EMWIN/HRIT Prototype Solution Performance

## 5.1 Noise Performance

### 5.1.1 Testing Methodology

There are three ways that the EMWIN/HRIT software receiver is tested for noise performance.

1. A live satellite signal degraded with IF noise
2. A controlled signal using a signal generator degraded with IF noise.
3. A controlled signal using a file source including signal impairments.

#### 5.1.1.1 Testing Methodology for a Live Signal

Figure 23 illustrates the live signal test environment. The satellite dish used to perform the test was a 2.3m dish with a G/T specification of 10.5 dB/K. The dish was pointed at GOES-12 for LRIT and EMWIN-I testing, and GOES-10 for EMWIN-N testing.

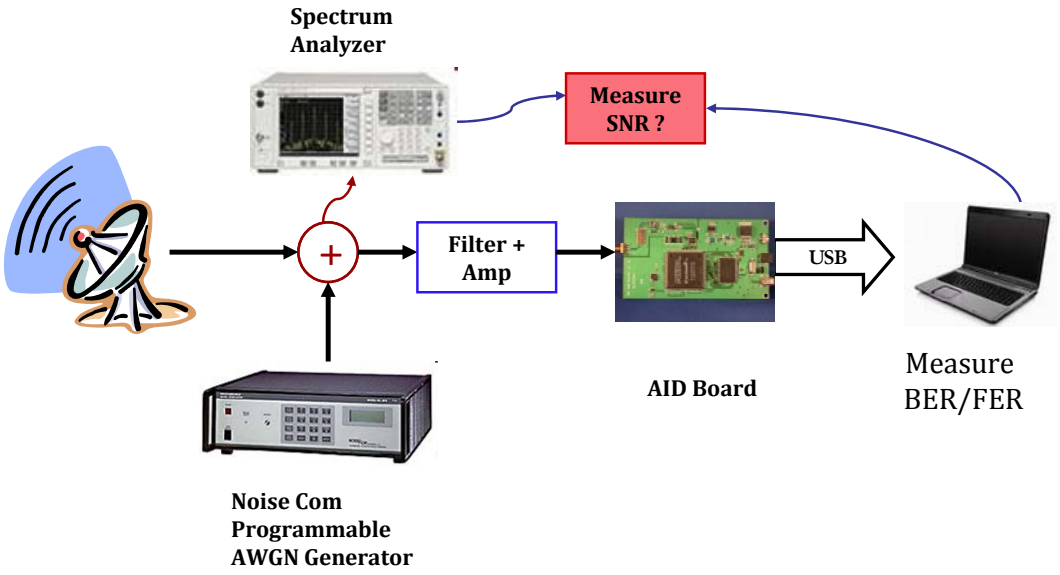


Figure 23. Testing setup for live signal feed experiment.

The dish had an RF front end integrated in it that provided filtering, amplification, and downconversion from L-band to 140 MHz IF using the Quorom Communications LNB model no. 1691.10-137.5, with a stated noise figure of 0.5 dB, and an oscillator accuracy of 2.5 ppm. The GOES-R IRD requires that the link be closed with a G/T of -0.3 dB/K. To simulate a system with a lower G/T spec, we degraded the signal using an IF noise generator supplied by Noise/Com Inc. model number UFX.

At the output of the signal + IF noise combiner, there is an additional filter and amplifier. These were there to make sure that the levels going into the AID board were sufficient and to filter the wideband noise being generated by the Noise Comm IF noise generator. The Noise Comm generates a 3-GHz wideband noise signal, and in practice, the AID board would not experience noise with this bandwidth due to filtering provided by the LNB.

The performance metrics for the system are CADU Frame Error Rate (FER) at the output of RS decoding and Bit Error Rate (BER) at the output of Viterbi decoding. The RS decoder automatically reports when one of the RS frames possesses an uncorrectable error within a CADU frame. If one of the four RS frames is in error, then the entire 8192 bit CADU frame is declared erroneous. Missed CADU frames also contribute to the FER, and they are computed by examining the VCDU markers as described in Section 4.10.1.

To estimate the BER of an unknown signal, we assume that the output of the RS decoder is error free if there are no uncorrectable errors detected in the frame. We compare this perfect sequence to the output of the Viterbi decoder (input to the RS decoder) to estimate the BER at the output of Viterbi decoding. We do not approximate the BER at the output of RS, but instead focus solely on FER.

#### 5.1.1.2 SNR Definition

The SNR was measured two ways. The first way was with the spectrum analyzer, and the second way was with the EMWIN/HRIT software receiver using the input to the Viterbi decoder. To measure the SNR using the spectrum analyzer, we would measure the in-band carrier power of the signal. To measure power, we integrated the power spectral density over a band of 450 and 40 kHz for LRIT and EMWIN respectively. Because this signal contains both noise, $N$, and signal, $C$, the measurement reported by the scope is actually $C+N$. In order to estimate $C/N_0$, we had to measure the noise density and the total in-band noise power. Generally speaking, $N = N_0 + 10\log 10(BW)$, where BW stands for bandwidth, and can be either 450 or 40 kHz, depending on which signal is being measured. We made no assumptions that the noise was perfectly flat, thus N was never computed analytically, but rather it was directly measured with the spectrum analyzer. Once there was a good measurement for $C+N$, $N$, and $N_0$, $E_b/N_0$ could be computed using the following formula.

$$C = 10\log_{10}[10^{(C+N)/10} - 10^{N/10}]$$
$$E_s/N_0 = 10\log_{10}[C/N_0] - 10\log_{10}[SymbolRate]$$
$$E_b/N_0 = E_s/N_0 - 10\log_{10}[CodeRate*M]$$

In the above formula, M is the number of bits/symbol for the modulation constellation, and the code rate is simply the rate of the error correction code. For LRIT signals, the symbol rate, code rate, and M were 293883 symbols/second, 1/2(*223/255) = 0.437, and M = 1 respectively. For EMWIN, the symbol rate, code rate, and M were 17970 symbols/second, 0.437, and M=2 respectively.

When working with FER, the code rate was ½*(223/255) because it takes into account both codes, while the code rate for BER was only ½ since it takes into consideration just the Viterbi decoder.

The other way to measure SNR was to measure $E_s/N_0$ directly in software after demodulation. In this case, the SNR estimate is not being defined at the input to the EMWIN/HRIT receiver, but instead at the input of the software-based Viterbi decoder. Figure 24 is a plot of a signal constellation at the input of the Viterbi decoder. The histogram of the data shown in Figure 24 would represent two Gaussian distributions centered around the software AGC levels of ±10. Denote the mean of one Gaussian distribution as $\mu$ and the variance $\sigma_n^2$. The estimate for SNR is

$$E_s/N_0 = 10\log 10(\mu^2/2\sigma_n^2)$$

We use an iterative signal processing algorithm called the expectation maximization algorithm to optimally determine $\mu$ and $\sigma_n^2$ of the Gaussian distributions.

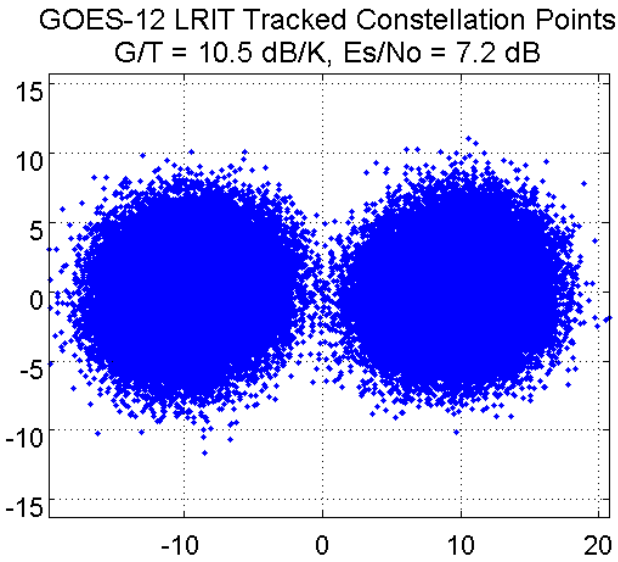


Figure 24. LRIT constellation points.

Measuring SNR at the input to the Viterbi decoder obviously assumes that the EMWIN/HRIT solution has no implementation loss from the input to the AID board to the input of the Viterbi decoder. Indeed, in most cases the difference between the SNR measured at the input of AID board and the input of Viterbi decoder were less than 0.2 dB. Using the digital data to estimate SNR instead of a spectrum analyzer is helpful for two reasons. First, the measurement accuracy and precision are better. Second, the digital method allows you capture the time varying nature of the SNR more effectively.

There are several ways that measurement inaccuracies can be introduced when measuring the SNR of a live signal. The best measurement accuracy is achieved when you can turn modulation off while doing the measurements. Modulation was not turned off while testing with the live signal. There was no way to measure just the noise in-band without measuring the carrier power, so an assumption was made that the noise was perfectly flat, thus the noise power at 137.5 MHz was equivalent to the noise at 139 MHz. In this case, we could measure the noise power over 139 MHz, where no signal was present, and assume that it held true for the band of interest of 137.5 MHz.

It was observed at times this was not necessarily true. There is no guarantee that the noise estimate did not posses undesirable signals that were slightly above the noise floor or if the noise measured at 139 MHz was precisely equal to the noise at 137.5 MHz. Sometimes, the noise floor appeared uneven causing measurement inaccuracies.

When measuring the carrier power, $C + N$ , the signal might contain distortions or interferers that cannot be fully accounted for.

The rapid fluctuations in a real signal meant that significant averaging had to be used in order to get reasonable measurements. This introduced many other potential measurement problems. Long averaging blurs time dependent characteristics that can greatly affect the performance of the system. For instance an interferer can corrupt a signal over a short burst of time, affecting instantaneous FER, but can be averaged out over time as far as SNR measurements are concerned. The performance of the down-converter might fluctuate sporadically over time as well and that might be averaged as well. In general, the SNR is a function of time. It was observed that the SNR changed 1.5 dB over a 3 hour

period without any changes to the lab setup. All of these issues make it difficult to measure SNR performance consistently over a long period of time.

Lastly, estimating Es/No using the data at the input of the Viterbi decoder can produce inaccuracies in three main ways. The first way is that the system may have had a large implementation loss due to an interferer, signal distortion, or RF impairments that invalidate the assumption of near zero implementation loss between the input of the AID board and the input of the software Viterbi decoder. The second way is when the Es/No is so low that the noise estimation algorithm has a non-negligible estimation error. Lastly, if the noise is non-Gaussian, for example if there is residual phase noise present, then the Expectation Maximization algorithms assumption of additive white Gaussian noise (AWGN) does not hold, and the estimate accuracy will suffer.

Despite all of these potential measurement inaccuracies, the SNR was measured very carefully, and the data presented in this section is reliable.

#### 5.1.1.3 Testing Methodology for a Signal Generator-based Signal Source

Figure 25 illustrates the test setup for the signal generator tests. The live feed was replaced with a controlled signal source using a signal generator. One thousand valid CADU frames, containing 8192 bits each, of clean CCSDS EMWIN and LRIT binary data was loaded into the signal generator, then the signal generator was set to the appropriate modulation, power and frequency to generate the signal IF waveform. In order to compare the results to the live signal, the original data rate for LRIT of 293,883 symbols/second was used instead of the proposed 927 ksymbol/second rate for HRIT in the GOES-R era. The SNR was for the most part known, but it was also measured with a spectrum analyzer and with the digitized demodulated data to ensure accuracy. FER and BER were measured.

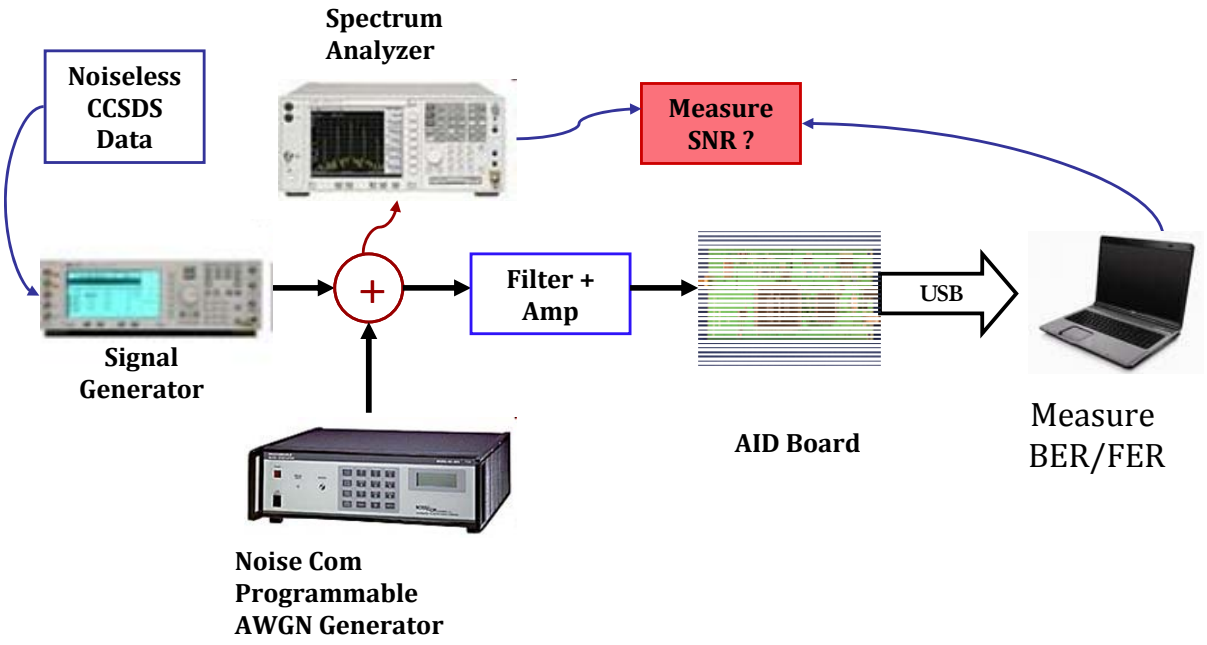


Figure 25. Diagram of signal generator testing setup.

#### 5.1.1.4 Testing Methodology Using a File Source

Because the EMWIN/HRIT receiver is software based there is an option to drive it with a file instead of driving it with a stream of USB samples from the AID board. The data files contain modulated EMWIN and LRIT data corrupted by AWGN, phase noise, and carrier drift. Figure 26 illustrates the test setup for testing with a file source. There is no hardware involved in this process. A mathematical software tool called Matlab was used to generate the waveform and write the data to a file. The EMWIN/HRIT software application used this data file as an input. The motivation for using a file source was to characterize the receiver performance over a wide range of known noise and RF impairment levels.

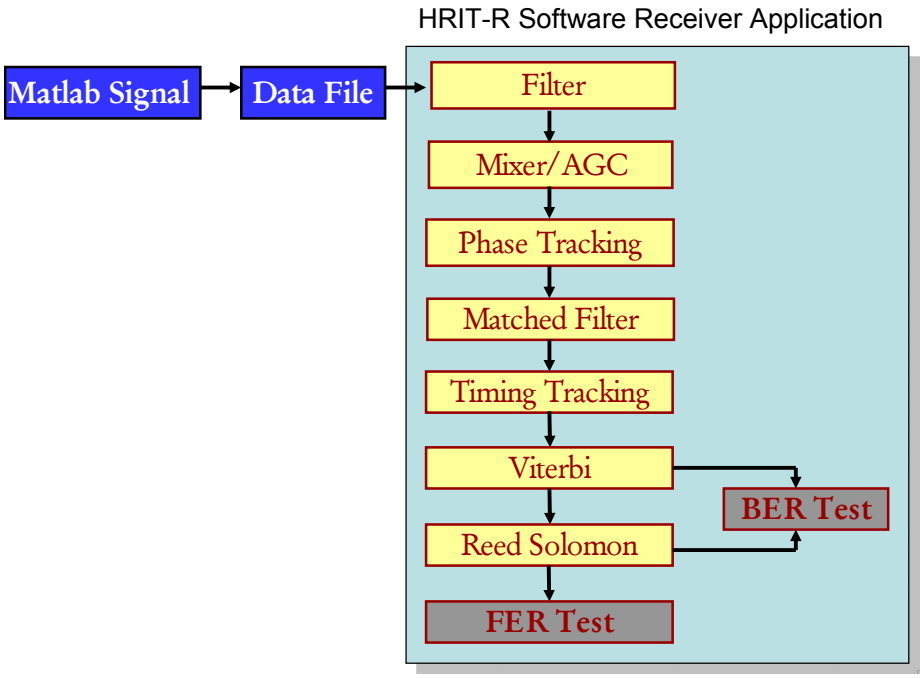


Figure 26. Diagram of testing using a file source.

The software generated waveform used perfect CCSDS LRIT and EMWIN binary data and then modulated them according to the modulation parameters. Denote the clean digital waveform as $x(t)$.

After $x(t)$ is generated, AWGN, phase noise, and carrier drift were added to the waveform, and the final waveform was then written to a file and read by the software receiver.

The phase noise was added first. It can be expressed in the following equation

$$x_{pn}(t) = x(t)e^{j\theta(t)}$$

Phase noise is modeled by a Wide-Sense Stationary (WSS) random process $\theta(t)$. Because it is WSS, it has a time domain autocorrelation function $\theta(\tau)$. The Fourier transform of $\theta(\tau)$ generates the spectrum $\Phi(\omega)$. Figure 27 shows the spectrum of the phase noise used in the test. The phase noise level was set using the 2σ level. Thus, a phase noise of 16° would mean that the random process $\theta(t)$ would be scaled such that $\frac{180}{\pi}(2\sqrt{E\{\theta(t)\theta^*(t)\}}) = 16$. An example of BPSK with a phase noise of 16° and SNR of 20 dB is shown in Figure 28.

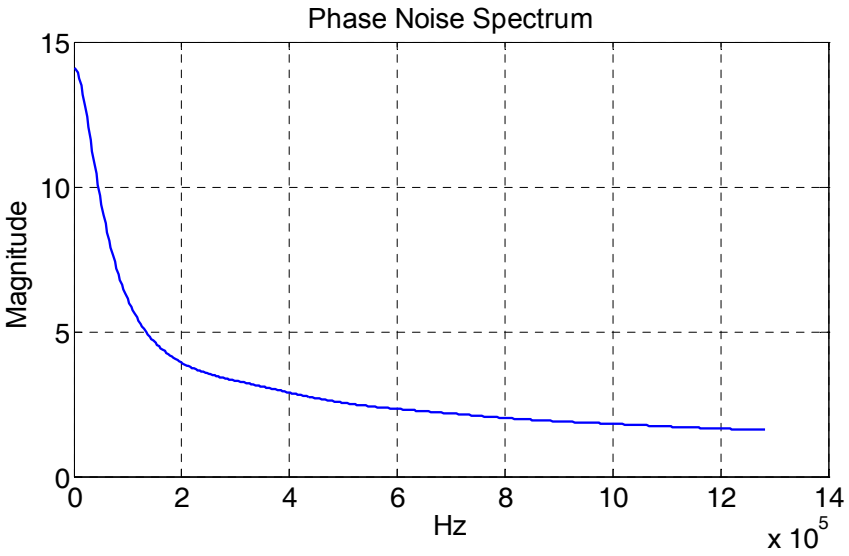


Figure 27. Phase noise spectrum of waveform generated by a file source.

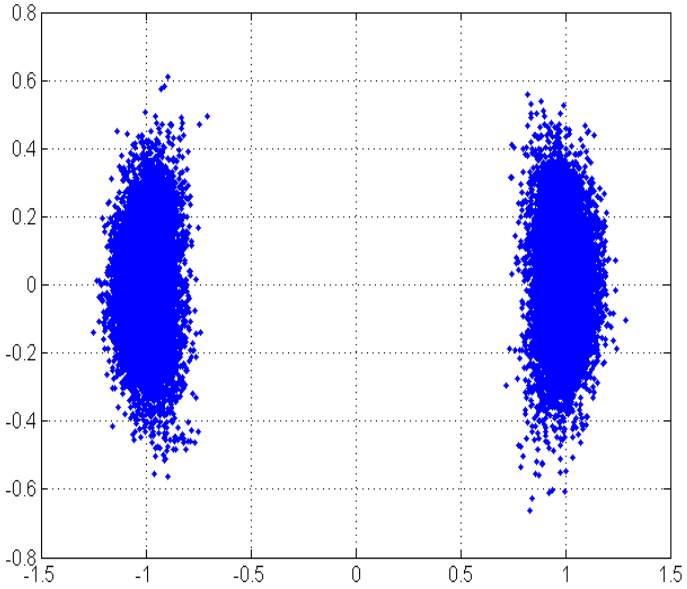


Figure 28. File source based LRIT signal with 16° of phase noise and SNR = 20 dB.

The carrier drift was added after adding phase noise. It was reported that EMWIN and LRIT systems can face a peak-to-peak drift rate of 5 kHz over 20 minutes. That means that the frequency drifts up 5 kHz in 10 minutes, then back down 5 kHz in 10 minutes. To model this drift profile we used a cosine wave instead of a linear drift profile to better approximate real conditions. If a linear profile is used, a 5 kHz rate over 20 minutes peak-to-peak would result in a constant drift rate of 8.3 Hz/second. Because the profile is a cosine shape, its rate exceeds 8.33 Hz/second at some points, and approaches 0 Hz/second at other times. Below is an equation that represents the drift profile.

$$dr(t) = \frac{f_d}{2}[\cos(\frac{2\pi}{1200}t) - 1]$$

In the above equation, $f_d$, represents the peak-to-peak drift over 20 minutes. The maximum instantaneous drift rate is $\pi f_d/1200$, which is about 1.5 times (π/2) faster than the instantaneous drift rate of the linear drift profile.

The final data stream, $x_r(t)$, was created by adding the drift, the carrier frequency $f_c$, and AWGN denoted by $n(t)$.

$$x_r(t) = x_{pn}(t)e^{j2\pi[f_c t + \int dr(t)]} + n(t)$$

Because the waveform was created digitally, the $E_b/N_o$ levels could be tightly controlled.

### 5.1.2 Noise Performance Results

All three testing methodologies are used for both the EMWIN and LRIT signals. The $E_b/N_o$ metric is estimated as discussed in the previous section. Recall that when using a file source, the $E_b/N_o$ is known precisely and added digitally. For the live signal, it is estimated with a spectrum analyzer, and for the signal generator, it is estimated with the spectrum analyzer and analytically computed using known parameters to ensure consistency.

#### 5.1.2.1 LRIT BPSK Results

The BER performance results of the EMWIN/HRIT solution using the various testing methodologies are shown in Figure 29. Recall that the BER is defined at the output of the Viterbi decoder, and thus does not take into consideration the RS error-correction performance. The full LRIT link, including RS decoding, is typically closed when the BER at the output of Viterbi is $10^{-3}$.

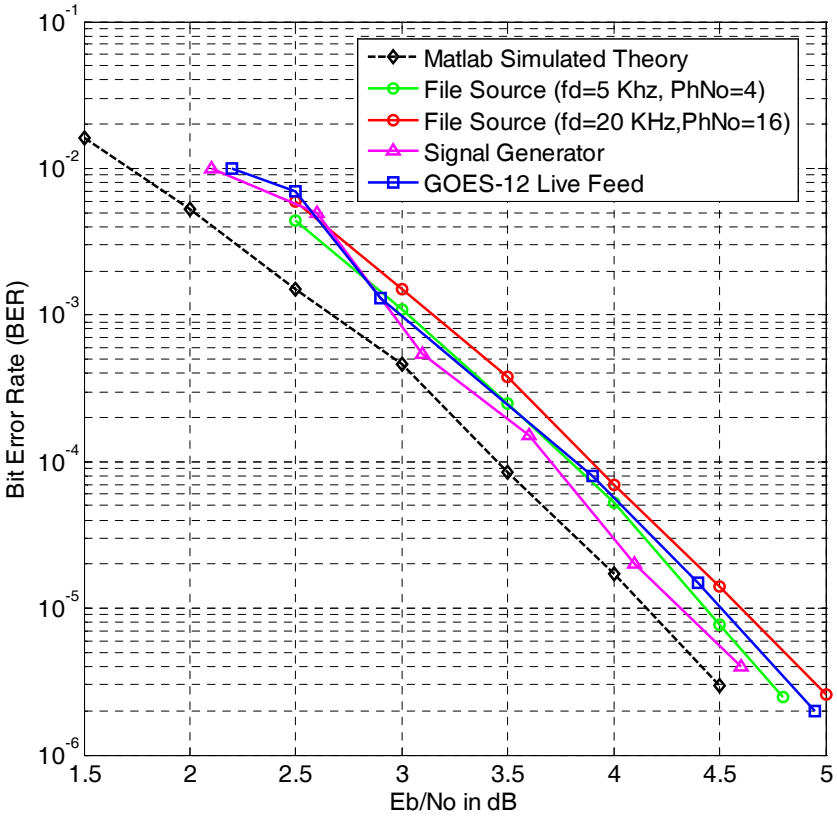


Figure 29. LRIT BER Performance Test Results Using All Test Methodologies.

The testing results show that the implementation loss of the EMWIN/HRIT system can be as low as 0.3 from theory in terms of BER performance. There are two file-source tests, one with a 5 KHz peak-to-peak drift rate over 20 minutes and a phase noise of 4°, and the other with a drift rate and phase noise of 20 KHz and 16° respectively. The less stressed file source test showed results worse than the signal generator test but better than the live signal. The more stressed file source experiment actually performed worse than the live signal, which implies that the live signal did not have that much phase noise or drift present.

The Matlab curves and theory are very smooth, while the live signal and signal generator curves are reasonably smooth. All of the curves are within 0.25 dB of each other. According to the BER plot illustrated in Figure 30, the implementation loss for our system is in the 0.3 dB range, however, implementation loss can only be officially determined when measuring the FER.

There did not appear to be much of a tradeoff between drift-rate tolerance and BER when testing with the file source. The phase bandwidth and timing bandwidth are always set to 3 kHz and 100 Hz respectively. These bandwidths are adequate to track dynamics within system specifications. If these bandwidths are significantly altered, the performance will suffer.

The FER is very difficult to measure, especially for the live signal. The tests must be run over a long period of time with random fluctuations, and the theoretical FER curve is very sharp. It goes from $10^{-2}$ to less than $10^{-5}$ in less than 1 dB. Error events that contribute to the FER are burst errors, cycle slips in the phase and/or tracking loops, undetected CCSDS synchronization headers, falsely detected CCSDS synchronization headers, in addition to standard AWGN. These events do not affect the BER nearly as drastically as FER, if at all. Add in measurement inaccuracies and it becomes apparent that FER curves would not be terribly accurate.

Instead of making a curve, we have generated tables that have three basic data points. The first data point is where the FER is extremely high and the link is totally unusable. The second point is where the FER is moderate leading to lost data, but still fairly usable. The final point is where the FER is confidently less than the system specification of $10^{-4}$. This data is shown in Table 12 for all three testing methodologies.

Table 12. LRIT Frame Error Rate Performance for all Testing Methodologies

| Signal Generator | | Live Signal | | File Source Stressed | |
|---|---|---|---|---|---|
| Eb/No | FER | Eb/No | FER | Es/No | FER |
| 2.7 | 1.60E-01 | 2.8 | 0.12 | 2.7 | 1.50E-01 |
| 3.2 | 1.40E-03 | 3.1 | 1.00E-02 | 3.1 | 8.00E-04 |
| 3.7 | <1e-4 | 3.7 | <1e-4 | 3.6 | <1e-4 |

#### 5.1.2.2 EMWIN O-QPSK Results

The EMWIN-N BER performance using a live-signal test, signal-generator test, and one file source test that has a drift of 1 kHz and phase noise of 4°, is shown in Figure 30. The live signal feed used to test the EMWIN signal was GOES-10. The live signal and the signal generator tests result in nearly identically performance. This implies that the additional RF impairments present in the live signal do not significantly degrade BER performance. The file source test shows noticeably better performance compared to the other testing methods. Again, implementation loss is established by looking at the FER instead of BER. However, the BER implementation loss appears to be a little under 0.5 dB and 1 dB at $10^{-4}$ for the file source and live signal respectively when using the BER metric.

Five different file sources that represent varying degrees of RF-impairments are compared in Figure 31. The first three curves compare performance across drift rates of 1, 3, and 4 kHz, while the last two curves compare performance across phase noise levels of 4º and 12°. The EMWIN-N receiver cannot tolerate as much drift as the LRIT signal, thus we limit the experiment to a drift rate of 4 kHz peak-to-peak over 20 minutes. Also, because EMWIN-N is an OQPSK signal, it is much more susceptible to degradation due to phase noise. For the LRIT signal, a single loop bandwidth is good over a wide range of frequency drift rates, however the EMWIN-N signal requires one loop bandwidth for small levels of drift and a different loop bandwidth for other levels of drift. The carrier recovery loop-bandwidth for drift rates of 1 kHz or less is 120 Hz, and 180 Hz otherwise. The clock recovery loop bandwidth is fixed at 18 Hz.

At a BER rate of $10^{-5}$, four of the curves have approximately the same performance, with a loss of about 0.4 dB compared to theory. The final curve, which represents 12° of phase noise, has a loss of about 0.8 dB.

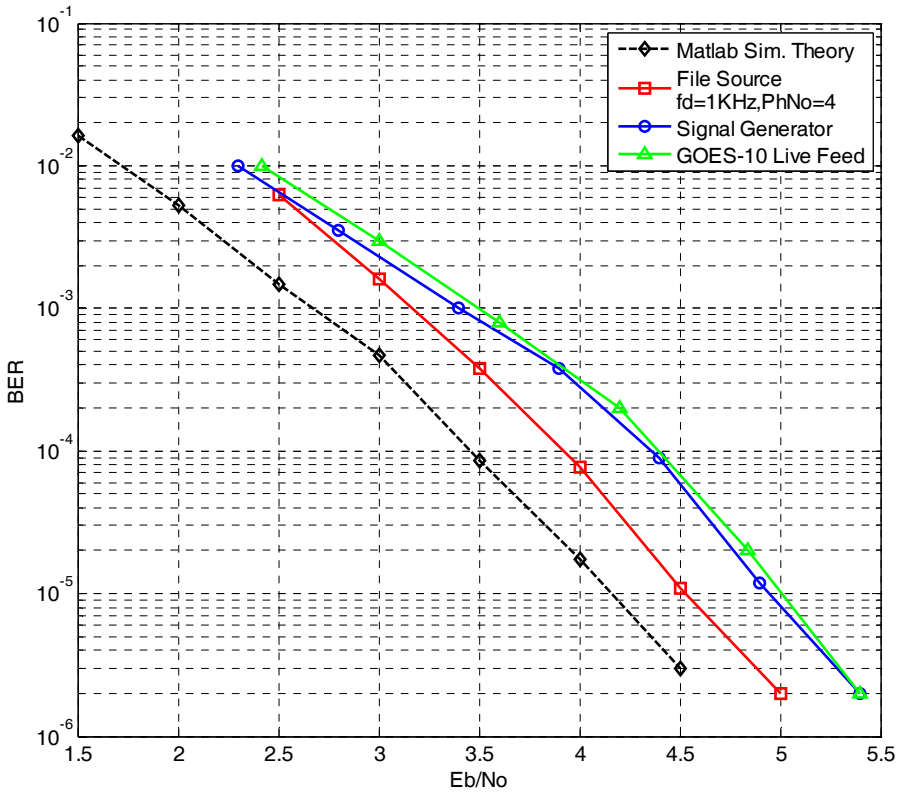


Figure 30. EMWIN-N bit error rate test results using all test methodologies.

Finally, the FER performance is shown in Tables 13, 14, and 15. The FER for the EMWIN-N signal was considerably more difficult to characterize. The carrier power seemed to be very dynamic compared to the LRIT signal when testing with the live GOES-10 feed, therefore it was very difficult to estimate FER vs $E_b/N_o$ precisely. The test would begin at 2 dB, then vary widely between 1.5 and 2.5 dB throughout the test run. On a given day, the link could be closed as low as 2.6 dB, but the following day, the link required an additional dB to close.

When testing EMWIN-N with the signal generator, the transition region was also difficult to find. At one point, the FER is unacceptably high. At a level 0.25 dB higher, sometimes you would get fairly

reasonable FER for awhile before the tracking loops would run into successive cycle slips and FER would degrade significantly. At 0.25 dB higher than that, the performance was great and the link was closed. For these reasons the FER isn't tabulated for the live GOES-10 feed or the signal generator tests.

As one can see in Table 13, we claim that the link can be closed in the range of 2.9 to 4.2 dB. The reasoning for this behavior is because over an extended period of time, the system ran error free with an estimated $E_s/N_o$ of 2.9 dB on a given day, but required 4.2 dB on another day. As it was mentioned before, one of the reason for this behavior is that the overall BER performance over an extended period of time (required to get meaningful statistics) is dominated by the lowest $E_s/N_o$ of the testing period. If the channel had a theoretical AWGN behavior then these variations would not occur. However a better characterization as a fading channel would be able to predict such behavior. The signal generator was able to close the link at 3.4 dB, and this is the level that the live GOES-10 typically closes the link under.

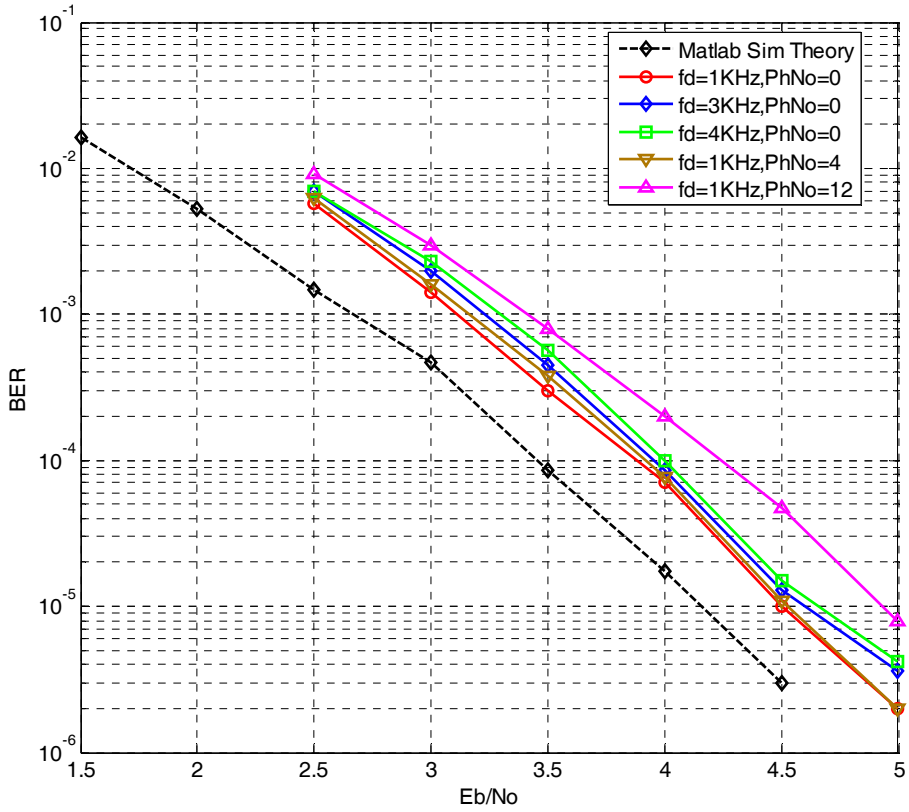


Figure 31. EMWIN-N BER performance comparison using file sources with varying levels of RF impairments.

Table 13. EMWIN-N FER Performance Over GOES-10 Live Feed and Signal Generator Testing

| Signal Generator | | Live Signal (LA) | |
|---|---|---|---|
| Eb/No | FER | Eb/No | FER |
| 2.9 | 2.00E-01 | 2.2-2.9 | >1e-2 |
| 3.4 | <1e-4 | 3.4-4.2 | <1e-4 |

Table 14 and Table 15 both represent testing with a file source. In Table 14, the phase noise was fixed at a moderate level of 4° while the carrier drift was varied from 1 kHz to 4 kHz. At 1 kHz, the link is closed very well at 3.6 dB, leaving a lot of margin. While at 3 and 4 kHz, it closes at 4.6 dB, which is right on system specification limits.

Table 14. EMWIN-N FER Performance Using a File Source with Varying Levels of Drift Rate with a Fixed Phase Noise of 4°.

| Drift = 1 kHz | | Drift = 3 kHz | | Drift = 4 kHz | |
|---|---|---|---|---|---|
| Eb/No | FER | Eb/No | FER | Es/No | FER |
| 2.7 | 1.00E-01 | 3.1 | 1.00E-01 | 3.1 | 1.40E-01 |
| 3.1 | 1.00E-02 | 4.1 | 1.00E-03 | 4.1 | 1.40E-02 |
| 3.6 | <1e-4 | 4.6 | <1e-4 | 4.6 | <1e-4 |

In Table 15, the carrier drift is fixed at 1 kHz, while the phase noise varies from 4 to 16°. At a moderate level of 8°, the link is closed at 4.1 dB leaving 0.5 dB margin left. At 12°, the link closes at 4.6 dB, which coincides with the system limits. Finally, at 16°, the link does not close at a reasonable level, and the implementation loss is 1.5 dB larger than the allocated maximum in system specifications. A phase noise of 16° is excessive, and if users are using reasonable LNBs, like the ones manufactured by Quorum (that was used for testing), phase noise will rarely exceed these levels.

Table 15. EMWIN-N FER Performance Using a File Source with Varying Levels of Phase Noise with a Fixed Drift of 1 KHz

| Ph No = 4 | | Ph No = 8 | | Ph No = 12 | | Ph No = 16 | |
|---|---|---|---|---|---|---|---|
| Eb/No | FER | Eb/No | FER | Es/No | FER | Es/No | FER |
| 2.7 | 1.00E-01 | 3.1 | 1.00E-01 | 3.1 | 1.00E-01 | 3.1 | 1.00E-01 |
| 3.1 | 1.00E-02 | 3.6 | 1.00E-03 | 3.6 | 1.00E-02 | 4.1 | 1.00E-03 |
| 3.6 | <1e-4 | 4.1 | <1e-4 | 4.6 | <1e-4 | 6.1 | <1e-4 |

In conclusion, both LRIT and EMWIN-N close the link with the live signal with an implementation loss of under 2 dB, and thus meet system specifications. The EMWIN-N signal is very sensitive to drift and phase noise, however, the live signal feed and Quorom down-converter used in our experiment never produced excessively high levels of drift or phase noise that caused the system to have an implementation loss of more than 2 dB.

#### 5.1.2.3 EMWN-I FSK Results

The Performance of EMWIN-I FSK was measured using the GOES-12 live signal feed, and the results compared to theoretical FERs computed via computer simulation. Since the EMWIN-I signal does not use any type of error correcting code protection, we only measure the FER, and not the BER. To measure FER, we compared the check sum transmitted during each 1116 byte packet, and the check sum computed from the received bits as specified in Section 4.10.4. The FER is plotted in Figure 32.

The theoretical curves were generated using Matlab, where the program's built in modulator and demodulators were used over AWGN noise, using packets of 1116*8 = 8928 bits. Due to the low transmission rate of the FSK signal, the results for low frame-error rates took many hours of simulation.

As shown in the Figure 32, in the waterfall region of the FER curve, we experienced an implementation loss of ~2 dB when using a Signal Generator and ~3 dB for the live signal. For the live-signal case an error floor at $2.10^{-3}$ appears for $Es/N_0$ values above 17 dB. For the simulated environment using a Signal Generator, performance appears to begin to floor around $Es/N_0$= 18.5 dB, however no errors were found after simulating 45,000 packets at $Es/N_0$= 20 dB.

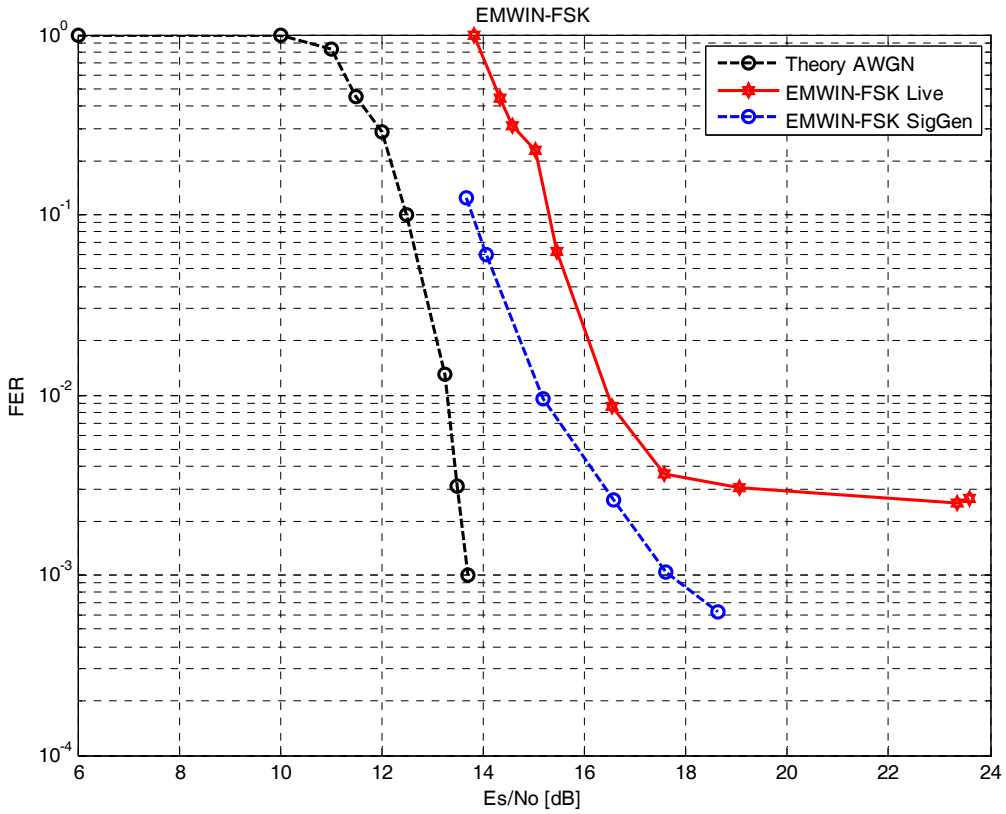


Figure 32. EMWIN-I (FSK) Frame Error Rate (FER) performance over GOES-12 compared to an ideal FSK modulation over AWGN noise. Packet size is 8128 bits.

### 5.1.3 GOES-R Frequency Plan Noise Performance Tests

In this section, we conduct an all signal generator test to emulate the GOES-R waveforms. In GOES-R, there will be a Global Re-Broadcast signal (GRB) located at 1690 MHz (136.5 MHz IF using Quorum down-converter) and HRIT will be centered at 1697.4 MHz (143.9 MHz IF). As shown in Figure 33, there are two signal generators, one for GRB and one for HRIT, that are combined before adding noise. The purpose of this experiment is to better represent the real frequency environment of the GOES-R signal. Ideally, there should be no difference in performance between this experiment and the standard frequency as tested before.

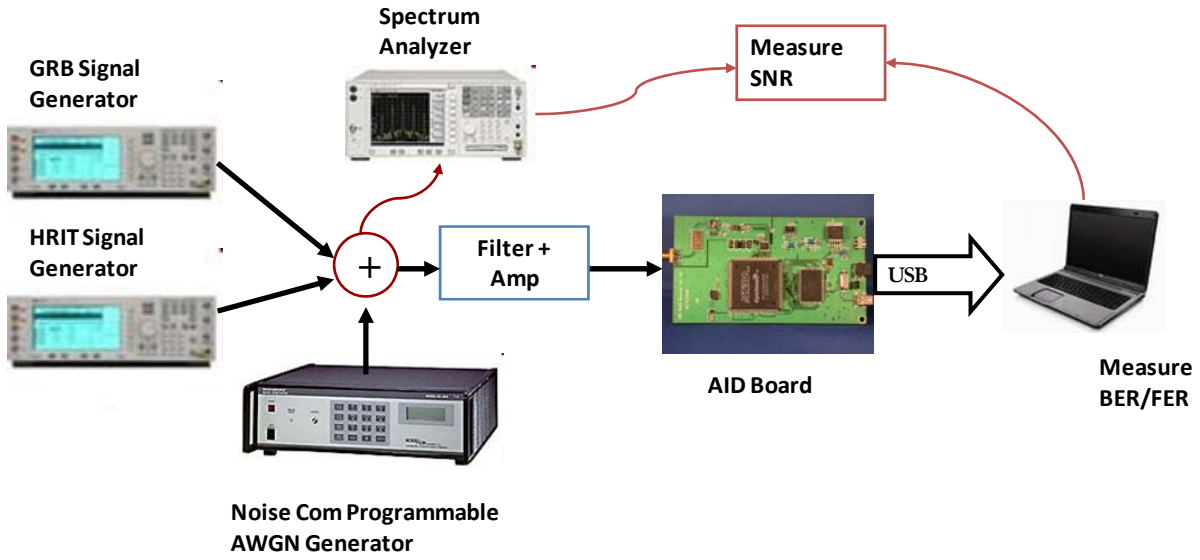


Figure 33: Diagram of GOES-R frequency emulation testing setup including HRIT and GRB signals using AID board.

The GRB signal has a proposed Equivalent Isotropically Radiated Power (EIRP) of 105.25 dBm per polarization while the EIRP for HRIT was 89.25 dBm. Because of this, the carrier power is set to 16 dB higher for the GRB signal than the HRIT signal on the signal generator. The GRB signal is emulated using a QPSK signal with a symbol rate of 9 Mega-symbols/second, and root raised cosine pulse shape of $\alpha = 0.25$, and a resulting bandwidth of 12 MHz. The HRIT signal has a symbol rate of 927 Kilo-symbols/second, and an $\alpha = 0.50$ resulting in a bandwidth of 1.39 MHz [7]. In the previous tests, LRIT was tested, not HRIT, thus the rate was only 293,883 symbols/second and centered at 137.5 MHz, with no GRB signal present. Below is a screen capture of the spectrum analyzer of the GRB and HRIT signals used to conduct the BER/FER tests in this section.

[7] The HRIT-GRB test was conducted using a roll-off factor of 0.5, however the latest official GOES-R IRD specifies a roll-off factor of 0.3. This discrepancy would not significantly alter the results of the test.

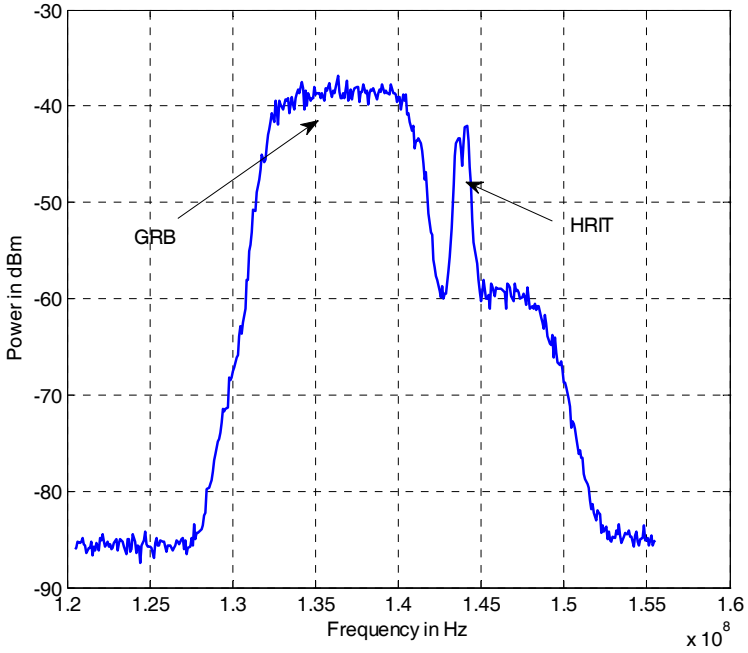


Figure 34. Spectrum analyzer screen capture of the input HRIT + GRB signal including noise and SAW filter response.

The FER and BER results for the HRIT+GRB signal generator simulation are identical to the results tabulated in the LRIT signal generator simulation tests. This implies that the EMWIN/HRIT solution closes the link on current generation LRIT signals, and will be able to do the same for the future HRIT signal in the future, without much performance degradation.

## 5.2 Throughput

Because HRIT will be operating at 927 ksps, it is important that the throughput of our software receiver be at least 1 Mbps. A throughput test must be conducted in order to ensure that our receiver can operate at higher rates. Because the data structure and modulation properties of the new HRIT signal will be equivalent to today's LRIT signal, throughput tests only need to be conducted on LRIT.

In order to achieve bitrates that are over 927 ksps, our software splits the demodulator across two threads. As it was explained on Section 4.7, the first thread consists of the software blocks from the input of the AID up to the Viterbi decoder. The second thread takes care of the Viterbi decoder and the remaining tasks, like frame synchronization and RS decoding. Throughput performance using an Intel® Core 2 CPU running at 2.66 GHz is shown in Table 16. To produce these measurements, we recorded a live feed of the LRIT 293 ksps signal into a local hard drive. We then ran the decoder to process the recorded data instead of a live satellite feed. A speed increase of about 1.8 times was achieved due to multi-threading.

All dual core Intel processing based computers (laptop or desktop) that have been tested have sustained throughputs greater than the proposed 927 ksps HRIT data rates.

Table 16. Throughput Measurements for EMWIN-N and LRIT

| Signal | Samples/Symbol | Single Thread | Mult. Thread |
|---|---|---|---|
| LRIT | 2 | 1030 Ksps | 1890 |
| EMWIN-N | 11 | 191 ksps | N/A |

## 6. Conclusion

The EMWIN/HRIT Prototype Solution uses a low-cost hardware digitizer that interfaces with a standard Windows based PC. This software solution allows processing of the EMWIN-I, EMWIN-N, LRIT, and also the future HRIT signal. The proposed system-solution uses the same antenna and LNB as current generation EMWIN and LRIT solutions. The part costs of the new AID board are about $100, providing a worldwide-affordable solution. The proposed system operates using a PC's USB interface for both data and power supply, eliminating the need for an additional power source.

The future HRIT signal will have a similar waveform definition to the LRIT signal, with the exception that its data-transmission rate will be increased from 293 to 927 Ksps. The latest throughput measurements using the AID box and an HRIT-simulated sequence allowed processing rates of more than 1.7 Msps using a portable laptop-machine. This demonstrates how users will be able to receive the different EMWIN / HRIT products using The Aerospace Corporation's software defined radio, once the HRIT signal becomes available, sometime after 2013.

The link budget for the new HRIT signal will allocate up to 2 dB of implementation loss margin for our solution. Our proposed AID solution shows an implementation loss of about 1.8 dB, which achieves the system-requirement.

The hardware and software are designed to work together providing a low-cost flexible solution. In this way, users have the flexibility of receiving a whole family of EMWIN/HRIT signals with a single hardware/software solution. Furthermore, compatibility with future transmission-protocols could also be added by means of a simple software update.

# 7. Acronym List

| | |
|---|---|
| ADC | Analog to Digital Converter |
| AGC | Automatic Gain Control |
| AID | Aerospace IF Digitizer |
| ARD | Aerospace RF Digitizer |
| AWGN | Additive White Gaussian Noise |
| | |
| BER | Bit Error Rate |
| BPF | Bandpass Filter |
| BPSK | Binary Phase Shift Keying |
| BW | bandwidth |
| | |
| CADU | Channel Access Data |
| CCSDS | The Consultative Committee for Space Data Systems |
| CGMS | Coordination Group for Meteorological Satellites |
| CIC | Cascaded Comb Filter |
| CPU | Central Processing Unit |
| CRC | Cyclic Redundancy Check |
| CVCDU | Coded Virtual Channel Data Unit |
| | |
| DDFS | Direct Digital Frequency Synthesizer |
| | |
| Eb/No | Energy per bit over noise density |
| EEProm | Electrical Erased Programmable Read Only Memory |
| EIRP | Equivalent isotropically radiated power |
| EMWIN | Emergency Managers Weather Information Network |
| Es/No | Energy per symbol over noise density |
| | |
| fc | center frequency |
| FER | Frame Error Rate |
| FFT | Fast Fourier Transform |
| FFTW | Fastest Fourier Transform in the West |
| FIFO | First Input First Output |
| FIR | Finite Impulse Response |
| FOM | Figure of Merit |
| FPGA | Field Programmable Gate Array |
| fs | sampling frequency |
| FSK | Frequency Shift Keying |
| | |
| G/T | Gain over temperature figure of merit |
| GNU | GNU's not Unix |
| GOES | Geostationary Operational Environmental Satellite |
| GPL | General Public License |
| GRB | Global ReBroadcast System |
| GUI | Graphical User Interface |
| | |
| HRIT | High Rate Information Transmission |
| | |
| I2C | Inter-Integrated Circuit (interface) |
| IF | Intermediate Frequency |

| | |
|---|---|
| IO | Input and Output |
| ITU | International Telecommunications Union |
| Ksps | Kilo-symbols per second |
| LDO | Low Drop Out (voltage regulator) |
| LED | Light Emitting Diode |
| LNA | Low Noise Amplifier |
| LNB | Low Noise Block Down-converter |
| LPF | Low Pass Filter |
| LRIT | Low Rate Information Transmission |
| M&M | Mueller- Müller (timing tracking) |
| M_PDU | Multiplexed Protocol Data Unit |
| Mbps | Mega-bits per second |
| MFC | Microsoft Foundation Class Library |
| Msamps/s | Mega samples per second |
| Msps | Mega-symbols per second |
| NOAA | National Oceanic and Atmospheric Administration |
| NWS | National Weather Service |
| OQPSK | Offset Phase Shift Keying |
| PLL | Phase-locked loop |
| RF | Radio Frequency |
| RS | Reed Solomon (decoder) |
| samps/s | samplers per second |
| SAW | Surface Acoustic Wave |
| SMA | SubMiniature version A connector |
| SNR | Signal to Noise Ratio |
| SPI | Serial Peripheral Interface |
| sps | symbols per second |
| USB | Universal Serial Bus |
| USRP | Universal Software Radio Peripheral |
| VCDU | Virtual Channel Data Unit |
| Vpp | Peak-to-Peak Voltage |

# 8. Bibliography


[1] The GOES-R Program Office, "GOES-R main website", [Online] Available: http://www.goes-r.gov/

[2] National Oceanic and Atmospheric Administration (NOAA), "Geostationary operational environmental satellite (GOES)," [Online]. Available: http://www.oso.noaa.gov/goes

[3] National Oceanic and Atmospheric Administration (NOAA), "EMWIN-N Overview," [Online]. Available: http://www.weather.gov/emwin/index.htm

[4] National Oceanic and Atmospheric Administration (NOAA), "EMWIN-N Prototype Receiver Specifications," [Online]. Available: http://www.weather.gov/emwin/transition/EMWIN-OQPSK Specifications Final.doc , May 2005

[5] National Oceanic and Atmospheric Administration (NOAA), "LRIT Overview," [Online]. Available http://noaasis.noaa.gov/LRIT

[6] National Oceanic and Atmospheric Administration (NOAA), "LRIT Transmitter Specification," [Online]. Available http://noaasis.noaa.gov/LRIT/pdf-files/LRITSTUDY4.pdf

[7] National Oceanic and Atmospheric Administration (NOAA), "EMWIN-N Commercial Vendors," [Online]. Available:, http://www.weather.gov/emwin/winven.htm

[8] GNU Radio, "The GNU Software radio." [Online]. Available: **http://gnuradio.org/trac/wiki**

[9] Dickens, M. L., B. P. Dunn, J. N. Laneman, "Design and Implementation of a Portable Software Radio," *IEEE Communications Magazine*, vol. 46, no. 8, pp. 58 – 66.

[10] Avtec Systems, "NOAA/GOES Emergency Management Weather Information Network," [Online]. Available: **http://www.avtec.com/customers/customer_case_studies/emwin_case_study**

[11] Werner Labs Inc, "The Antenna, model GD100X60," [Online], Available: **http://www.wernerlabsinc.com/antenna.htm**

[12] Ettus Research LLC, "The USRP" [Online]. Available: http://www.ettus.com

[13] GNU Radio, "USRP Documentation." [Online]. Available: **http://www.gnuradio.org/trac/wiki/USRP**

[14] Oppenheim, A.V., R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, 1989.

[15] Volder, J., "The CORDIC trigonometric computing technique," IRE Transactions on Electronics Computers, vol. EC-8, no. 3, pp. 330-334, Sept. 1959.

[16] Hogenauer, E. B., "An economical class of digital filters for decimation and interpolation." *IEEE Trans. On Acoustics, Speech and Signal Processing*, ASSP, vol. 20, no. 2, pp. 155-162, 1981.

[17] Intel Software Network, "Intel integrated performance primitives," [Online]. Available: **http://www.intel.com/cd/software/products/asmo-na/eng/302910.htm**.

[18] R. F. Rice, "Some practical universal noiseless coding techniques," JPL-PUB-79-22; NASA-CR-158515, Pasadena, CA, 1979.

[19] J. G. Proakis, Digital Communications, 3rd ed., McGraw-Hill, 1995.

[20] S. Lin. And J. D. Costello, Error Control Coding: Fundamentals and Applications. Prentice-Hall., 1983.

[21] Wikipedia, The Free Encyclopedia, "Phase shift keying." [Online]. Available: **http://en.wikipedia.org/wiki/Phase-shift-keying**

[22] National Oceanic and Atmospheric Administration (NOAA), "EMWIN-I Overview," [Online]. Available: **http://www.weather.gov/emwin/wintip.htm**

[23] Wikipedia, The Free Encyclopedia, "Asynchronous serial communication," [Online], Available: http://en.wikipedia.org/wiki/Asynchronous_start-stop

[24] National Oceanic and Atmospheric Administration (NOAA), "EMWIN-I Transmitter Specification," [Online]. Available: **http://www.weather.gov/emwin/winpro.htm**

[25] Wikipedia, The Free Encyclopedia, "Frequency shift keying." [Online]. Available: **http://en.wikipedia.org/wiki/Frequency-shift-keying**.

[26] K. Mueller and M. Müller, "Timing recovery for digital synchronous data receivers," IEEE Trans. On Comm., vol. 24, no. 5, pp. 516-531, Jun. 1976.

[27] International Telecommunication Union (ITU), "Recommendation V.36. Data Communication over the Telephone Network," pp. 7-9, 1993.

[28] Weather Message. [Online]. Available: http://www.wxmesg.com

# Appendix A. Software Design Document

## A.1 EMWIN/HRIT Software Solution Overview

This appendix details the design architecture of the prototype EMWIN/HRIT software radio. It is intended for designers who are interested in designing a commercial system or doing independent research or analysis based off of the prototype design. The objective of this document is to guide designers through the process of re-engineering our software solution using the source code that is provided by the GOES-R program office.

The core design of the software is completely based off of GNU Radio version 3.1.2. GNU Radio is an open source product and subject to the general purpose library GPL. GNU Radio is constantly being updated by the user community. Further information can be found in [8].

The Aerospace Corporation modified GNU Radio version 3.1.2 to work in the Windows environment and to take advantage of multiple-core computers. Since our software development efforts began, the GNU Radio development community has added built-in functionality to work in the Windows environment and support multiple-core processors. The newer versions of GNU Radio create a separate thread for every communication block present in a given project.

When our software receiver was compiled using these new versions, the large amount of threads present (over 12) generated an excessive amount of handshaking in the software. The large amount of buffers and controllers between threads caused the overall throughput of the project to decrease from the speeds achieved using only two threads. Overall a careful design where software radio blocks are distributed smartly to balance the CPU's load proved to be a more effective way of designing an effective software radio. The GNU Radio environment described in this section is based off of our implementation of GNU Radio version 3.1.2, and is not current with the latest GNU Radio release.

In order to open and recompile the source code, the user will need Microsoft Visual Studio 2003, the Intel® Compiler version 10, Intel® Performance Primitives 5.1, and Microsoft Windows XP Service Pack 2.

To begin, download GOESRadio.zip, and unzip its contents to the root C:\ drive. Notice that there are several directories inside the parent directory, C:\GOES Radio\.

The following folders can be found in the parent directory GOES Radio\

- \Common
- \Documents
- \EMWIN_I
- \EMWIN_N
- \Filters
- \Firmware
- \gnuradio
- \GUI
- \EMWIN_HRIT

The directories: EMWN_HRIT, EMWIN_I, EMWIN_N, and GUI are all directories that contain a Visual Studio Project file that can be used to generate a new executable. The HRIT project creates EMWINHRIT.exe, which is a software receiver for HRIT and LRIT. The EMWIN_I project creates

EMWINI.exe which is a software receiver for the EMWIN-I FSK signal, and finally the EMWIN_N project creates EMWINN.exe for the EMWIN-N OQPSK signal.

The \common folder has static libraries for the FFTW and LibUSB32 package, and it also has files that are shared by more than one project. The frequency acquisition class is a good example of shared class that would be located in this folder.

The \Filters folder contains low pass filter coefficients of varying decimation rates for the LRIT and EMWIN-N signal. The current filter coefficient path is set to \temp\usrp_log\input_data. This path can be changed to reflect your compilation, or you can move the filter coefficients into that directory.

The \Firmware folder contains windows drivers for the IF digitizer USB device, usrp.inf. It also contains USB and FPGA firmware, std.ihx and std_2rxhb_2tx.rbf respectively. A new windows system variable, USRP_PATH, must be created and must point to the firmware folder.

The last folder is the \gnuradio folder. The \gnuradio folder is a stripped down version of the folder from GNU Radio 3.1.2. Although the file structure is very similar, almost all of the original GNU Radio files that are used in the Visual Studio projects have been modified. In addition to the original file structure, there is a new folder called \arradio-core. This folder contains completely new files that never existed in GNU Radio 3.1.2. There are many critical files in this folder, but the most extensive ones are ar_viterbi.cc, ar_viterbi_lib.cc, ar_fsk_demo_cf.cc, and ar_freq_acq.cc. The files, ar_viterbi.cc and ar_viterbi_lib.cc, implement the entire Viterbi decoder. The file ar_fsk_demo_cf.cc implements most of the FSK demodulation chain, while a significant portion of the frequency acquisition engine was implemented in ar_freq_acq.cc.

The GUI is a separate project written using Microsoft Foundation Class Library (MFC). The GUI takes user inputs and executes EMWINHRIT.exe, EMWINI.exe, and EMWINN.exe with various configuration flags. This is equivalent to executing the different software-receivers from the command prompt using the flags described in Table 6, Table 7 and Table 9. These applications create products such as gif, jpg, txt, zip files, and store these files in two folders C:\EMWINTEMP and C:\LRITTEMP. The GUI has a file dialog box that shows the files contained in these temporary folders and allows users to display the files. The GUI also allows users to set various radio options. For more information about executing the software radio using the command prompt or the GUI, please see the User Guide Document on the GOES-R website.

## EMWIN-HRIT Visual Studio Project Description

To open the project,

- Browse to C:\GOES Radio\.
- Open EMWIN-HRIT.sln

The solution view ofEMWIN-HRIT.sln should show the major folders

- Source Files
- Header Files
- Resource Files

The header and resource files will not be covered in this document, only the source code. Inside the Source Files folder, there are the following sub folders

- gnuradio
- Main

The source files within the \Main subfolder contain most of the major processing code. The generic communication signal processing blocks, such as `gr_agc_cc.cc`, are provided by the GNU Radio 3.1.2 package and contained inside the \gnuradio folder and its subfolders. The low level code that facilitates the GNU Radio development environment is in the gnuradio\gnuradio-cores\runtime folder. The top level source file that possesses the function main() is main_emwin_hrit.cpp, which is located within the \Main folder of the solution.

## EMWIN-HRIT Solution Source Code Organization

In Figure 35 we show the main source code flow from main_emwin_hrit.cpp.

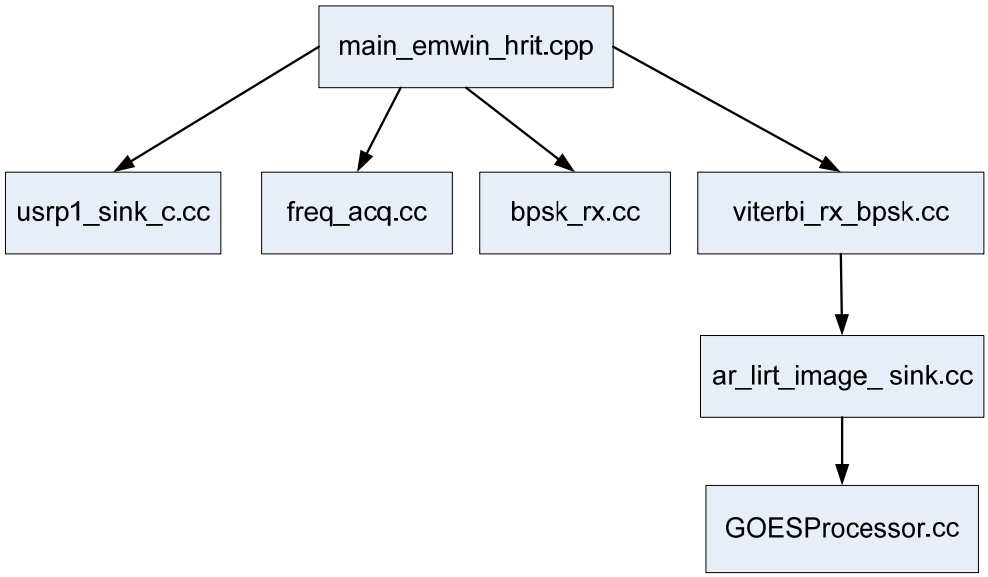


Figure 35. Flow of major software blocks in the EMWIN/HRIT C++ project

All of the functionality that interfaces the hardware to the software is contained in the gnuradio\gnuradio-cores\usrp folder. If the digital portion of the hardware is significantly altered, the code in the usrp folder must be changed. The most important file is `usrp1_source_c.cc.`

The frequency acquisition algorithm is implemented with a frequency-acquisition class described in freq_acq.cc within the Main\arradio-functions\ folder. This class uses another frequency acquisition sub-class that is defined in the solution file \gnuradio\Arradio-cores\`ar_freq_acq.cc`.

The Solution Subfolder data link layer contains code that implements the data link layer blocks as specified in Section 4.10.1. These blocks include Frame Synch, De-randomization, RS decoding, decompression, and final file assembly. The main data link layer code is contained in `GOESprocessor.cpp`. GOESprocessor was not written in the GNU Radio environment, so the GNU Radio block ar_lrit_image_sink.cc was written to create a bridge. The lrit_image_sink class creates and instantiates the GOESprocessor class.

The RF Digitizer Board Configuration subfolder has a file called `front_end_ctrl.cpp`. This file allows the software to configure the hardware front end for the RF board as specified in Section B.4 via a Serial Peripheral Interface (SPI) interface. The gain of the mixers and the frequency of the DDFS can be controlled using the functions defined in that class.

The other folder within \Main is BPSK demod and Viterbi. This folder contains code that performs the physical layer demodulation blocks (shown in Figure 16), the files `bpsk_rx.cpp` and

`viterbi_rx_bpsk.cpp`. In single threaded mode, the file `viterbi_rx_bpsk.cpp` is not used, as the Viterbi decoder is already implemented in `bpsk_rx.cpp`. In multi-threaded mode, which is the default mode, `bpsk_rx.cpp` implements everything in the Physical Layer except Viterbi decoding and data link layer processing. The Viterbi decoding and data link layer processing are done on its own thread in the file `viterbi_rx_bpsk.cpp`.

## A.2 EMWIN-HRIT Solution Signal Source Code Details

### A.2.1 Introduction to GNU Radio Syntax

Before covering source code details, a short overview of our GNU Radio environment syntax is required. There are four general steps to building a communication system using the modified GNU Radio environment

1. Create a block space
2. Define individual blocks within a block space
3. Connect the individual blocks together into a chain
4. Pass data from block to block within a block space

An example of syntax that creates a block space is

```
vector<gr_block_sptr> bpsk_blocks;
```

This syntax creates a vector of GNU Radio blocks named `bpsk_blocks`. A GNU Radio block is implemented using a gr_block class, which is defined in `gr_block.cc.`

To create an individual block, one can use the multitude of GNU Radio based blocks or create a custom block of class gr_block or gr_synch_block. A tutorial for building a GNU Radio compliant block is provided on the GNU Radio webpage. The following is an example of how to define an AGC block using the GNU Radio file `gr_agc_cc.cc.`

```
float agc_rate          = 1e-5;
gr_agc_cc_sptr agc      = gr_make_agc_cc (agc_rate, 10);
```

In the above example, an AGC block, named "agc" is created. It is configured to have an averaging window of 100,000 samples, and to maintain an AGC level of 10. All GNU Radio block classes possess a make-function that creates, configures, and assigns a block to a variable.

After making one or more blocks, they can be connected using the following syntax

```
connect_blocks(block 1, output port, block 2, input port, block
space).
```

The connect_blocks function is defined in `gnuradio.cpp`.

The following is an example of connecting blocks. In our system, there is a phase tracking block named "phase_track", and a matched-filtering block, named "match_filt". To connect the first output of "phase_track"to the first input of the "match_filt" the following code is executed

```
connect_blocks(phase_track, 0, match_filt, 0, d_blocks)
```

The first output and input ports are referenced by 0 based on our convention. The block space used for this code is called `d_blocks`. Most blocks have one input and one output, but some blocks have multiple inputs and multiple outputs.

Lastly, after connecting all of the blocks, the data is passed from one block to the next, within a block space labeled `d_blocks,` by starting a scheduler using the following syntax.

```
gr_single_threaded_scheduler_sptr scheduler =
start_scheduler_thread(d_blocks, true);
```

The scheduler function is defined in the file `gr_single_threaded_scheduler.cc.`

### A.2.2 Description of Source Code Functions

One of the key tasks performed in `Main()` is to form a connection between hardware and software using the GNU Radio file `usrp1_source_c.cc.` Any hardware that is designed to work with GNU Radio software should be designed to utilize `usrp1_source_c.cc`. If a contractor develops hardware that is not compatible with `usrp1_source_c.cc,` then significant changes to the software will be required.

To instantiate the hardware one should declare and instantiate the usrp_source_c class. This is done in `Main()` using the following syntax.

```
usrp1_source_c_sptr          usrp_source;
usrp_source= usrp1_make_source_c ( which_board, decim_rate,nchan, mux,mode, 0,0,"","")
```

The only variable that must be set when establishing a connection to the hardware is the decimation rate within the FGPA based down-converter. The frequency of the mixer can be set to *fc* using the following code.

```
usrp_source->set_rx_freq (0,fc);
```

As described in Section 4.2, the first action of the software radio is frequency acquisition. Frequency acquisition does not need to be run concurrently with demodulation, so it is more efficient to allocate all of the computer resources to either frequency acquisition or signal demodulation and decoding, but not both. To do this, a hardware source is instantiated for the purpose of frequency acquisition only. After the frequency acquisition engine determines the frequency offset present in the received signal, the hardware is released and is re-instantiated later. Below is a sequence of code representing frequency acquisition.

```
freq_acq_config                 freq_acq_config0;
freq_acq_config0.Fs             =  config.Fs;
freq_acq_config0.Fc             = 0;
freq_acq_config0.resolution     = 50;
freq_acq_config0.freq_range     = 50e3;
freq_acq_config0.num_fft_avg    = 16;
freq_acq_config0.square_law     = true;
freq_acq_config0.fourth_law     = false;

//Configure USRP
freq_acq_config0.usrp_which_board       = which_board;
freq_acq_config0.usrp_decim_rate        = decim_rate;
freq_acq_config0.usrp_nchan             = nchan;
freq_acq_config0.usrp_mux               = mux;
freq_acq_config0.usrp_mode              = mode;
freq_acq_config0.usrp_fc                = fc;

freq_acq acq(freq_acq_config0, false);
acq.run();
```

In the example above, a frequency acquisition configuration struct, freq_acq_config0, is formed. This configuration struct has two major groups of variables: hardware configuration and frequency acquisition configuration. Frequency acquisition variables such as sampling frequency, the number of FFTs, and other parameters are configured in the top half, while the second half is used to configure the hardware, which must be instantiated and then released during this process. After performing frequency acquisition, the next steps are BPSK demodulation, Viterbi decoding, and data link layer processing as discussed in sections 4.4, 4.7, and 4.10 respectively.

The HRIT solution was designed to be capable of using multiple threads available on a PC. Viterbi decoding and data link layer processing run on one thread while BPSK demodulation runs on a second thread. In order to do this, two separate block spaces must be created, one for Viterbi decoding and one for BPSK demodulation. This is done with the following code.

```
vector<gr_block_sptr> bpsk_blocks;
          vector<gr_block_sptr> viterbi_blocks;
```

BPSK demodulation,Viterbi decoding and data link processing are abstracted away from `main()`. The bpsk_rx class, defined in `bpsk_rx.cpp`, is created to define and connect the blocks associated with demodulation. The BPSK demodulator is instantiated by calling the `bpsk_demod()` function of the `bpsk_rx` class.

```
bpsk_rx     bpsk_demod(bpsk_blocks, config);
```

`Main()` interfaces with this class through the public class functions `get_input_block()`, `get_soft_output()`, and a configuration struct called `bpsk_config`. Several configuration options are passed from `main()` to the bpsk_rx class `bpsk_demod` using the following code.

```
bpsk_config    config;
config.Fs                   = 64e6/decim_rate;
config.bit_rate             = atoi(in_bitrate);
config.shaping_rolloff      = atof(in_shaping_rolloff);
```

```
config.PBW                = atof(PhaseBW);
config.TBW                = atof(TimeBW);
config.AGC                = atof(in_AGC);
config.vthresh            = atof(vit_thresh);
config.resynch_thresh     = atoi(resynch_thresh);
config.resynch_period     = atoi(resynch_period);
config.sink_data          = atoi(sink_data);
```

The bit rate, pulse shaping, sampling frequency, timing tracking loop bandwith, and other parameters are transferred from `main()` to the bpsk_rx class.

`Main()` creates the first two blocks, the hardware source block called `usrp_source`, and the digital rejection filter block called `lrit_filter`. The next several demodulation blocks are performed within the bpsk_rx class. The following code shows the syntax for passing the processing from `main()` to the bpsk_rx class, bpsk_demod.

```
connect_blocks(usrp_source,     0, lrit_filter, 0, bpsk_blocks);
connect_blocks(lrit_filter, 0, bpsk_demod.get_input_block(), 0, bpsk_blocks);
```

Within the bpsk_rx class (`bpsk_rx.cpp`), several blocks are created and connected as follows.

```
connect_blocks(agc, 0, phase_track, 0, d_blocks);
connect_blocks(phase_track, 0, match_filt, 0, d_blocks);
connect_blocks(match_filt, 0, timing_track, 0, d_blocks);
connect_blocks(timing_track, 0, slicer_real, 0, d_blocks);
```

As defined in the header file for the bpsk_rx class (`bpsk_modem.h`), the agc block is connected to the outside world using the public function `get_input_block(),` and the slicer_real block is likewise connected to the outside world using `get_soft_output().`

After the bpsk demodulator finishes processing the data, it is sent to the Viterbi decoder. The Viterbi decoder is instantiated by calling the `viterbi_dec()` function of the `viterbi_rx` class.

```
viterbi_rx  viterbi_dec(viterbi_blocks, configV);
```

The Viterbi class is configured using the following code.

```
viterbi_config configV;
configV.AGC                         = atof(in_AGC);
configV.resynch_thresh          = atoi(resynch_thresh);
configV.vthresh                 = atof(vit_thresh);
configV.resynch_period          = atoi(resynch_period);
configV.socket_flag             = atoi(in_socket);
```

Several variables are passed to the Viterbi decoder such as the AGC level, the Viterbi synchronization threshold, and the time between Viterbi re-synchronization periods. The data link layer variables such as CADU frame synchronization thresholds and socket output options are also passed to the Viterbi_rx class.

As stated earlier, Viterbi processing is done on one thread over the `viterbi_blocks` block space, while BPSK processing is done on another one over the `bpsk_blocks` block space. To pass data

from the output of BPSK processing to the input of Viterbi processing, the following code within `main()` is executed.

```
connect_blocks(bpsk_demod.get_softout(), 0, buff_sink, 0, bpsk_blocks);
connect_blocks(buff_source, 0, viterbi_dec.get_input_block(), 0, viterbi_blocks);
```

The first `connect_blocks()` call uses the block space `bpsk_blocks` while the second `connect blocks()` function call uses the block space `viterbi_blocks.` Using the our modified GNU Radio environment, `connect_blocks()` is not designed to be inherently multithreaded. To pass data from one thread to the next, circular buffers are used.

The blocks, `buff_sink` and `buff_source`, are circular buffers defined over the block spaces `bpsk_blocks` and `viterbi_blocks` respectively. The two buffers point to the same address in memory and are designed to be thread safe. This allows data to be passed between threads safely while still using syntax designed for single threads.

The Viterbi_rx class performs the following code within the file `viterbi_bpsk_rx.cpp`.

```
connect_blocks(viterbi_decoder,    0, viterbi_bytes, 0, d_blocks);
  connect_blocks(viterbi_bytes,      0, lrit_sink, 0, d_blocks);
```

Viterbi decoding is represented by the block `viterbi_decoder`. The block `viterbi_bytes` converts a bit-stream into a byte-stream. The block, lrit_sink, creates an instantiation of the GOESprocessor class, which take the Viterbi output byte-stream and performs frame synchronization and RS decoding.

The last major execution within `main()` begins the scheduler for both the `viterbi_blocks` and `bpsk_blocks` block spaces.

```
        gr_single_threaded_scheduler_sptr       scheduler =
            start_scheduler_thread(bpsk_blocks, true);
    gr_single_threaded_scheduler_sptr       scheduler_viterbi =
           start_scheduler_thread(viterbi_blocks, true);
```

That concludes the illustration of the code flow in `main().` This is just a general illustration of the code flow, not a software source code manual. In order to alter the code, an experienced software and communications engineer will need to review the actual source code.

# Appendix B. Hardware Implementation

## B.1 Overview

The Aerospace Corporation designed two boards, an IF board (AID) and an L-band board referred to as the Aerospace RF Digitizer (ARD). Both designs were designed to demonstrate that the GOES-R HRIT signal could be digitized using a "low cost" digitizer board. The boards were not designed for direct commercial use. Both boards are heavily based on the Ettus Research LLC USRP board and firmware developed by the GNU Radio project. The boards are copyright protected by Ettus Research LLC copyright, the FPGA firmware and receiver software are subject to the General Purpose License. The boards were not designed to compete with the USRP or any other commercially available design. This report does not recommend or favor any specific digitizer design for the purposes of commercial reception of the GOES signals. For more information on the USRP schematic design, please see the GNU Radio webpage.

The AID works with data at 140 MHz while the ARD works with L-band frequencies between 1668 – 1708 MHz. The performance of the IF board is better than the RF board, and thus it is the preferred board. All of the testing, details, and analysis in this report were conducted using the IF board. A brief description of the RF board is provided in Section B.4. The schematics for the IF board and RF board are available from the GOES-R program office.

## B.2 Aerospace IF Digitizer Layout

In order to minimize interference from the noisy digital components, the analog portion of the board was placed as far as possible from the digital side. The analog and digital grounds are joined by the ferrite beads, which improve noise isolation. Many beads were used in parallel to reduce the inductance and therefore reduce the ground bounce. The power planes were separated by Low Drop-Out Voltage regulators (LDOs), especially in the case of power supply to the RF side of the board for the ARD. The general layout of the AID board is shown in the Figure 36.

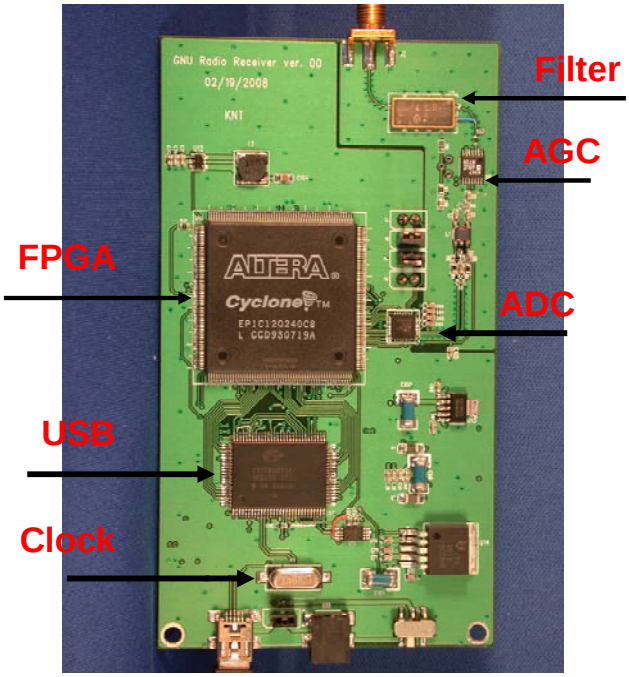


Figure 36. AID board layout.

### B.2.1 Components

Below is a list of all of the components present in the board together with their price.

Table 17. Component List and Prices

| Device | Part# | Supplier | # | Price |
|---|---|---|---|---|
| USB Controller | CY7C68013A-100AXC | http://www.mouser.com/cypress | 1 | $8.28 |
| FPGA | EP1C6-Q240 | http://www.altera.com | 1 | $15.50 |
| FPGA | EP1C12-Q240C8 | http://www.altera.com | 1 | $35.50 |
| ADC | LTC2226 | | 1 | $4.55 |
| AGC(Digital) | AD8367(45dB range) | http://www.analog.com | 1 | $5.88 |
| Crystal (24MHz) | XC742CT-ND | http://www.digikey.com | 1 | $0.43 |
| Crystal (64 MHz) | CWX813 | | 1 | $1 |
| Transformer | MA/COM1-1-13(balun) | www.macom.com | 1 | $0.90 |
| EEprom | 24LC02B/ST | http://www.digikey.com | 1 | $0.22 |
| BPF(SAW) | TB0320A/TB0203A/TB0438A | http://www.golledge.com | 1 | $12.77 |
| LDO 3.3V (1A) | TPS78633KTTT | http://www.ti.com | 1 | $2.80 |
| DC-DC | LM3671MF-1.5 | http://www.national.com | 1 | $1.50 |
| DC-DC | PT5521-1.5(1.5A) | http://www.ti.com | 1 | $15.00 |
| Capacitor | Various Capacitors | http://www.digikey.com | 104 | $8.32 |
| Resistor | Various Resistors | http://www.digikey.com | 30 | $0.41 |
| Inductor(HP) | CDRH5D18NP-1ØØN | http://www.digikey.com | 1 | $0.46 |
| Inductor | MI1206K601R-10 | http://www.digikey.com | 4 | $0.07 |
| Transistor | 2N2907 | www.mouser.com | 1 | $0.23 |
| SMA | J500-ND | http://www.digikey.com | 2 | $2.54 |
| Fuse(500mA) | P11361TR-ND | http://www.digikey.com | 1 | $0.11 |
| LED | L62405CT-ND | http://www.digikey.com | 2 | $0.16 |
| USB connector | WM17113-ND | http://www.digikey.com | 1 | $0.40 |
| JUMPER | A26228-ND | http://www.digikey.com | 5 | $0.15 |
| Board | 4 layer 4"x4"(1K, 4weeks) | http://www.4pcb.com | 1 | $5.86 |
| Population | Board | Ingko@dec-assembly.com | 1 | |
| Power fem. Con | WM4900 | http://www.digikey.com | 1 | $0.39 |
| Power male Con | WM2900 | http://www.digikey.com | 1 | $0.19 |
| | | **Total** | | $76.61 |

#### B.2.1.1 Clocking

There are two crystal oscillators on the board: a 24 MHz clock driving the USB controller, and a 64 MHz crystal which controls the ADC and the digital signal processing portion of the FPGA. The 24 MHz clock has a frequency stability of 100 ppm and a jitter of 116 psec. The 64 MHz clock is not temperature controlled, and has a frequency stability of 25 ppm and a jitter of 1 psec.

The FGPA FIFO that streams data to the USB controller is clocked using a 48 MHz clock. The USB device takes the 24 MHz input clock, and uses an internal clock multiplier and Phase Locked Loop (PLL) to generate the 48 MHz clock for the FPGA FIFO.

#### B.2.1.2 Power

For the AID board, power is provided using the same USB interface used for data communication. For the ARD board, an external power supply is necessary. The total power consumption for the AID and ARD board is 1.5 W and 4.8 W respectively.

#### B.2.1.3 ADC

An LTC 2228 A/D converter was used in the project. The maximum sampling frequency of this device is 65 MHz, with an input signal bandwidth of 500MHz. The ADC SNR is about 71 dB at 140 MHz. The digital output of this device is in the 2's complement format and is set by providing ¾ Vdd to one of the pins. Aside from the data format, the device has no other major configuration parameters. The noise figure was computed to be around 37dB. The input dynamic range is $2V_{p-p}$.

#### B.2.1.4 AGC

An AD8367 AGC was used in the project to ensure proper signal levels before entering the ADC. This AGC takes in a single ended input, conditions it, and using a balun transformer produces a differential signal for the ADC. The AGC has a 7.5 dB noise figure at its maximum gain of 42.5 dB for our frequency of interest.

#### B.2.1.5 Power Measuring Circuit

The power measuring circuit lights a series of LEDs based on the estimated signal strength. The signal strength is estimated by evaluating how much amplification the AGC needs to boost the input signal to 13 dBm. The AGC gain is fed into a circuit that drives the individual LEDs. This circuit consists of a simple resistive ladder which produces six voltages between 0 and 0.85 V. The voltage steps required to light successive LEDs is not linear since only six LEDs are used to show the power levels across a broad range.

#### B.2.1.6 SAW Filter

The Golledge TB0439 IF anti-aliasing filter was used in the design. The center frequency is 140 MHz, the bandwidth is 20 MHz, and the insertion loss is 11 dB.

This filter allows a wide pass band to accommodate multiple IF frequencies, as well as sharp rolloff to reject out-of-band noise and signals. The stop band provides over 30 dB in rejection, and the maximum ripple in the pass band is less than 1dB.

## B.3 Hardware-Software Interface

### B.3.1 Loading the USB and FPGA Firmware

The heart of the original USRP design, and consequently the AID design, is the digital processing that is provided by the USB controller and the FPGA. Both of these digital devices utilize intricate firmware.

When the AID is first plugged into the computer, the FPGA and USB firmware is not loaded. When the board is plugged into a USB port, a sequence of events takes place. First, one of the LEDs begins to blink with frequency of 3 Hz. This blinking means that the USB controller has awakened and loaded a simple program from the onboard EEProm. The EEProm is loaded using an I2C interface that is available on the board. The EEProm routines provides necessary device IDs that the PC uses to recognize the USB device. If the device's driver has been properly installed, a "USRP filter" message will pop up on the computer, along with USB device name.

The USB drivers are based on an open source USB package called LibUSB32. The drivers are located in GOES Radio\gnuradio\gnu_usrp_files\usrp.inf.

When a user creates

```
usrp_source = usrp1_make_source_c
(which_board,decim_rate,nchan,mux,mode, 0,0,"","")
```

within the main software radio code, it executes a chain of commands that go all the way down to low level USB calls, as provided by the LibUSB32 library. The first thing this function does is load two firmware files: one is std_main.ihx for the USB controller, and the other is std_2rxhb_2tx.rbf, which is the firmware for the FPGA. The naming convention was based on the original USRP design which had 2 half band Rx channels and 2 Tx channels, however our solution only utilizes one Rx channel.

The C++ file that facilitates the firmware loading sequence is usrp_prims.cc. In order to successfully load the firmware, the application requires a firmware PATH. The USRP_PATH variable must be set as an environment variable, with its value indicating the directory where the firmware is located. This directory is \GOES Radio\gnuradio\gnu-usrp\gnu_usrp_files\rev4.

### B.3.2 USB Controller Firmware

The USB firmware runs on a USB controller called Cypress CY C68013. The USB firmware can be recompiled using the Small Devices C Compiler (SDCC). The top level source code needed to recompile the USB controller is located in GOES Radio\gnuradio\usrp\firmware\src\usrp2\usrp_main.c. Compilation of this file and its dependencies will produce an ihx binary file. The source code creates an instance of the 8051 embedded controller architecture.

Once loaded with the powerful firmware, the USB controller is capable of supporting many of the digital functions on the board including: programming the FPGA, supplying SPI and I2C interfaces to configure all SPI/I2C compatible onboard devices, and streaming the high speed data from the FPGA to the computer.

### B.3.3 FPGA Firmware

The GNU Radio webpage provides FPGA source code as well as documentation on its overall architecture. The FPGA firmware can be recompiled using the Altera Quartus II software. Quartus II is an open source development tool for Altera FPGA devices. The top level source code for the FPGA firmware is located in the directory GOES Radio\ gnuradio\usrp\fpga\toplevel\usrp_std\usrp_std.v.

To alter the FPGA design, open usrp_std.v in the Quartus II environment. Once a design is open, the files can be altered manually, or reconfigured using the configuration files. Common configuration files, e.g. common_config_bottom.vh, can be found in \GOES Radio \gnuradio\usrp\fpga\toplevel\include\. More specific configuration files, like config.vh, are available in the same directory as the top level source file.

After altering the design as necessary, a designer can recompile the project to generate a new binary rbf file. The rbf file is loaded onto the FGPA during every runtime execution of a GNU radio software application.

Altering the FPGA design can usually be avoided by utilizing the existing registers to extend the functionality of any board design. These registers are defined in `fpga_reg_commons.h` and `fpga_reg_standard.h`, and are also summarized on the GNU Radio website.

For the ARD board, we needed to control the gains of few onboard devices and for that we used I/O registers that are available for general use. There are two I/O registers with a bit width of 16 bits. The Output register is accessed by first enabling it with _write_oe(which_board, 0x0000,0xffff) and then by writing a value to it using **`usrp_source->write_io(which_board, 0x0900, 0x0c00)`**. To understand how to modify and/or utilize the entire FPGA design, designers will need to follow the documentation and source code available by GNU Radio.

### B.3.4 GNU Radio Files Associated with Hardware Instantiation

The entire GNU Radio architecture cannot be summarized in this document. A designer that wants to modify the digital back-end of the hardware will ultimately need to examine the source code and the documentation that is available on the GNU Radio website. This section intends to give a very brief overview of the files that enable GNU Radio to interface with the hardware.

It is advisable to visit http://gnuradio.org/doc/doxygen.html where all the classes can be easily traced. GNU Radio software is highly hierarchical. The GNU Radio software architecture was designed to keep the signal processing algorithms and the low level hardware routines separated. There are only four high level classes that handle the hardware connection: usrp1_source_c, usrp1_source_s, usrp1_sink_c, and usrp1_sink_s. The usrp1_source_c and usrp1_source_s classes create a hardware receiver that supports complex and short data types respectively. The sink classes create hardware transmitters. Because we are working with complex digital modulation data, and we are using the hardware as a receiver, we only use usrp1_source_c. The following is an example of the inheritance chain for usrp1_source_c:

usrp1_source_c→ usrp1_source_base → usrp_standard.cpp → usrp_basic → usrp_prims, fusb.cc → libusb.dll (lowest level USB function calls)

The FPGA and USB firmware are loaded into the hardware in the usrp_prims.cc file.

As an example of how the hardware configuration parameters are passed down through varying levels of software, let's examine how setting the decimation rate on the FPGA's down-sampling block flows through the GNU Radio software.

The high level call,

```
usrp1_make_source_c (which_board,decim_rate,nchan,mux,mode, 0,0,"","")
[defined in usrp1_source_c.cc]
```

invokes

```
usrp_standard_rx(which_board,decim_rate,nchan,mux,mode, 0,0,"","")
[defined in usrp_standard.cc]
```

which calls

```
      set_decim_rate (decim_rate) [defined in usrp_standard.cc]
```

which calls

```
  int v = has_rx_halfband() ? d_decim_rate/2 - 1 : d_decim_rate - 1;
  bool ok = _write_fpga_reg (FR_DECIM_RATE, v); [defined in
usrp_basic.cpp]
```

FR_DECIM_RATE is an FPGA register defined in the `fpga_reg_commons.h`

The function _write_fpga_reg() calls

```
usrp_write_fpga_reg (d_udh, regno, value) [defined in usrp_prims.cc]
```

which calls,

```
usrp1_fpga_write (udh, reg, value); [defined in usrp_prims.cc]
```

which calls,

```
  buf[0] = (value >> 24) & 0xff;
  buf[1] = (value >> 16) & 0xff;
  buf[2] = (value >>  8) & 0xff;
  buf[3] = (value >>  0) & 0xff;

  return usrp_spi_write (udh, 0x00 | (regno & 0x7f),
                  SPI_ENABLE_FPGA,
                  SPI_FMT_MSB | SPI_FMT_HDR_1,
                  buf, sizeof (buf));  [defined in usrp_prims.cc]
```

`SPI_ENABLE_FPGA` is specified in usrp_spi_defs.h

The function usrp_spi_write() propagates the value of *buf* to write_cmd(), and write_cmd() then calls

```
usb_control_msg (udh, requesttype, request, value, index,
                     (char *) bytes, len, 1000);
```

Usb_control_msg() is a low level USB device primitive supplied in the libusb.dll dynamic library provided by LibUSB32.

In summary, setting the FPGA's decimation rate requires data to propagate down 8 levels. If a contractor alters the analog front end to the AID, then there will likely be no change to the software. Adding additional registers to the FPGA would not require the software to be rewritten since there are already open registers defined in the file `fpga_reg_standard.h`. If a contactor wishes to alter the serial controller, or the interaction between the serial controller and the FPGA, or the existing SPI interfaces on the board, very significant changes to the software would be required.

The GNU Radio design already provides a flexible bidirectional SPI and I2C interface, a solid connection to a sufficiently powerful FPGA, and a more than adequate USB interface to support the GOES-R application. It is strongly recommended that designers work within the framework already provided by GNU Radio and the original USRP design. Utilizing the USRP design will require contractors to resolve copyright issues.

## B.4 Aerospace RF Digitizer Overview

The Aerospace RF digitizer is very similar to the IF digitizer except that it has four new major components: An L-band filter, a mixer, an additional oscillator, and a Direct Digital Frequency Synthesis unit (DDFS). These four components are used to down-convert the signal from L-band to IF frequencies. The rest of the board is completely unchanged after the down-conversion from L-band to IF.

The mixer is implemented using the RFMD 2460 mixer chip. This chip has a variable gain that is controlled by the FPGA's SPI interface, and can be as high as 25 dB. The noise figure of the device is about 1.5 dB.

The DDFS was implemented using the ADF4360-3 synthesizer chip. Any frequency can be generated in software through the SPI interface in order to properly down-convert the L-band signal to a reasonable IF. The DDFS chip is driven by a 27 MHz crystal with frequency stability of 2.5 ppm.

The Aerospace RF digitizer is shown in Figure 37. As shown in the figure, the SMA input goes to the mixer chip, and then to the successive blocks that are present in the AID board.

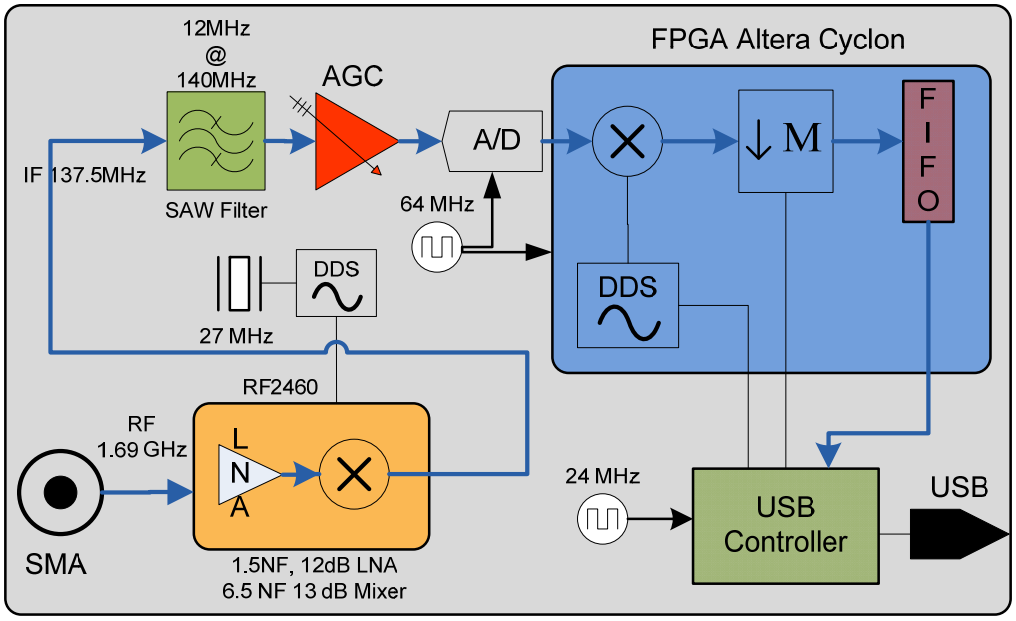


Figure 37. Aerospace RF (L-band) digitizer board block diagram