Digital Video and Telemetry Transmitter using the DVB-S2 Standard

By
Matthew B. Zachary

A Graduate Paper Submitted in Partial
Fulfillment of the Requirements for the degree of:
MASTERS OF SCIENCE
in
Electrical Engineering

Approved by:

____

Dr. Dorin Patru, Associate Professor
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

____

Dr. Sohail Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
August 2017

Graduate Paper Release Permission Form

Title:
Digital Video and Telemetry Transmitter using the DVB-S2 Standard

Rochester Institute of Technology
Kate Gleason College of Engineering

I, Matthew B. Zachary, hereby grant permission to the Wallace Memorial Library to reproduce my graduate paper in whole or part.

____
Matthew B. Zachary

____
Date

Acknowledgments

This project would not have been possible without the vision and guidance of Dr. Dorin Patru. I'd like to thank him for providing this opportunity and for mentoring me throughout the process. I'd also like to thank my family for their support and patience as I worked on this project. In addition, the small group at my church has been a constant source of encouragement and prayers during this summer. Technical support has been provided so graciously by faculty Mark Indovina, Jim Stefano, and Ken Snyder, and by peers Connor J. Goldberg and Lincoln D. Glauser. In addition, classmate and friend Adam Steenkamer's DVB-S2 modulation code was integral to the functionality of this project, and I appreciate his hard work on it. Finally, thank you to RIT for the invaluable engineering knowledge and experience that so greatly contributed to my success.

Abstract

Digital Video and Telemetry Transmitter using the DVB-S2 Standard

Matthew B. Zachary

Supervising Professor: Dr. Dorin Patru

This system builds upon a sister project [1] to implement a low power digital video and data transmitter that operates over Amateur Radio bands. The sister project developed a 16 APSK DVB-S2 transmitter in hardware, and this project focuses on integrating this modulator into a complete system. The system was designed to capture and compress HD video, and transmit that video alongside telemetry data as a transport stream over DVB-S2. This paper covers the system design, component and software selection, and individual subsystem designs required to create a fully integrated data transmission solution. Receivers were also selected 'off-the-shelf' to allow any amateur radio operator with a compatible satellite receiver to observe video and data streams sent from this transmitter. This paper also compares a software-based system and this hardware-accelerated system in which DVB-S2 modulation is done within a FPGA, in terms of possible transmission bitrate and system power draw. The hardware-accelerated system was found to be 48.57% more efficient than the software based system. Therefore, significant benefits were obtained by moving the modulation to the FPGA.

Table of Contents

List of Tables

List of Figures

1. Introduction

The target application for this project is a remote sensing application, particularly a high-altitude balloon instrumentation platform used for scientific research. In this specific application, it is critical to have a live video downlink to monitor the progress of the flight and the experiment. In previous applications, this video downlink was performed as analog television. However, this technology, while readily available and simple to use, has significant limitations especially in light of newer video transmission technologies. Following the trend of television broadcast, this project focuses on the use of Digital Video Broadcasting (DVB) standards to transmit live video.

By transmitting digitally, we can take advantage of well-established and used video compression algorithms (i.e. MPEG2, MPEG4) and data error correction systems (i.e. LDPC, FEC). By using video compression (which often compresses by a factor of ~80-100) we can either increase the resolution of the stream or decrease the required bandwidth. By introducing error correction techniques, we can use a lower transmit power and trust that the error correction built-in to the receiver will correct the additional losses suffered along the transmission medium due to the lower power. These two benefits to digital video transmission make it ideal for a remote sensing application in which both downlink bandwidth and power are at a premium. Finally, given that video is transmitted digitally, in theory one could transmit data alongside it, provided the standard supports it. This would allow for the addition of a telemetry channel with a high data rate, which would be useful for sensors where a high sample rate is required (i.e. inertial measurement unit).

For nearly all of these DVB standards, available transmitters are expensive and power hungry, which do not lend themselves well to portable remote sensing applications. Receivers, on the other hand are extremely easy to come by and relatively cheap, since they are used by consumers for viewing digital television. Therefore, this project focuses on the development of a low power and low cost transmitter that a common off-the-shelf receiver is compatible with. In this way, we can minimize the required development costs and setup costs for those who wish to take advantage of the technology. This would make it particularly attractive for the amateur radio community.

In this area, the best current solution is the DATV-Express board [2]. It has the benefits obtained by transmitting video digitally (higher resolution, lower bandwidth, lower power, targets off-the-shelf receivers) but suffers from a few drawbacks. The first is that it doesn’t take advantage of the latest in video compression techniques, as it uses MPEG2 over H.264 (MPEG4 Part 10). H.264 offers approximately double the video quality for the same bitrate, or half the bitrate for the same video quality as MPEG2, and is supported by modern satellite receivers. In addition, the DATV-Express was built using the DVB-S video broadcast standard, and since then a second version has been released (DVB-S2). DVB-S2 adds new coding and modulation schemes, and allows for more flexible data transfer using Generic Stream Encapsulation (GSE). The overall performance is also increased by 30%. Finally, the DATV-Express system does not currently support any data/telemetry transmission.

The following diagram summarizes the goal of this project. A camera capturing live video feeds uncompressed video into a video compression system. From there, the compressed video is multiplexed with the data and they both undergo DVB modulation. This DVB modulated signal is then transmitted over the air.

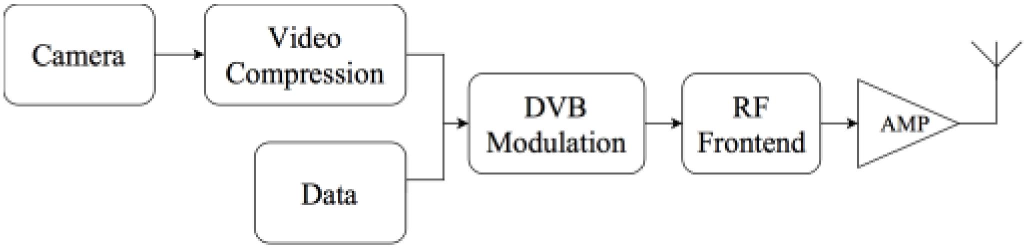


Figure 1: High Level System Diagram

2. Background

This section describes, at a high level, how the individual pieces of the video and data transmission system will be designed in order to meet the design specifications given above.

2.1 DVB Modulation

The Digital Video Broadcasting consortium [3] defines numerous standards for broadcasting digital video, including over cable (DVB-C), wirelessly along the surface of the earth (terrestrial - DVB-T), and wirelessly from a satellite (DVB-S). Each of these standards is specifically designed to optimize data transmission given the specific medium. For example, DVB-T acknowledges the challenges that occur when transmitting close to earth, namely the reflecting of the signals off buildings, and interference. DVB-C acknowledges that a cable is a low-loss medium (relative to air) and is not as susceptible to interference. Therefore, it contains less error correction than its terrestrial and satellite counterparts. DVB-S was specifically designed for satellite use, in which there is significant loss in the path due to the long distances, but where line of sight is generally guaranteed and reflections aren’t of concern [4]. For this reason, DVB-S is used, and specifically the second revision of the standard, DVB-S2. The target application is a high-altitude balloon platform which will fly at ~100,000 feet and have a line of sight with our receiving base station. The platform will essentially act as a DVB-S2 satellite transmitter, albeit from a much shorter distance than typical satellites.

The DVB-S2 modulation process is generally split into a 7-parts, and is fairly computationally expensive. The block diagram for this process is shown below:

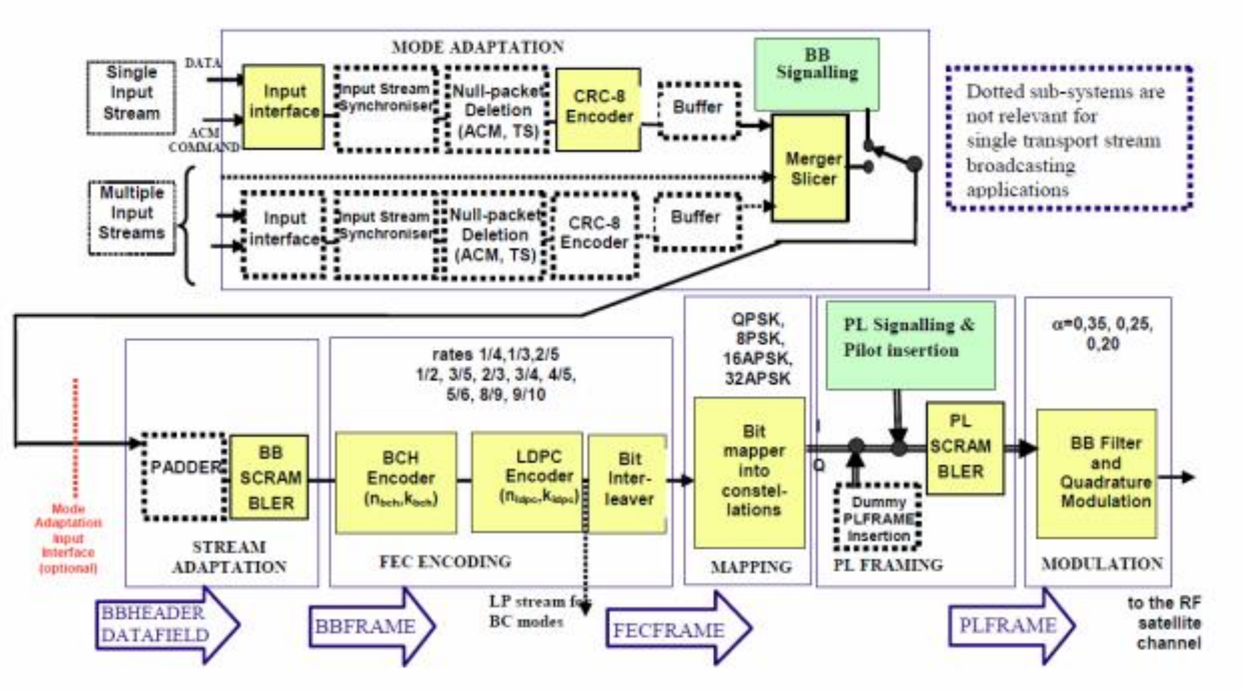
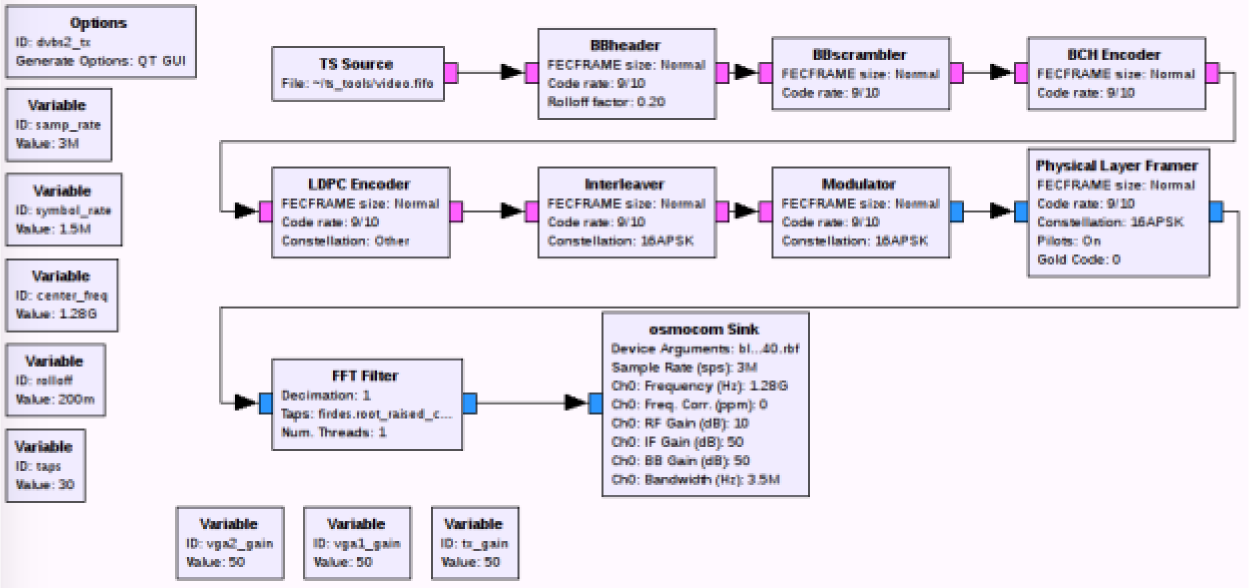

MODE ADAPTATION
Single Input Stream
DATA
ACM COMMAND
Input interface
Input Stream Synchroniser
Null-packet Deletion (ACM, TS)
CRC-8 Encoder
Buffer
BB Signalling
Merger Slicer
Multiple Input Streams
Dotted sub-systems are not relevant for single transport stream broadcasting applications
Mode Adaptation input interface (optional)
PADDER
BB SCRAM BLER
STREAM ADAPTATION
rates 1/4,1/3,2/5 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10
BCH Encoder
LDPC Encoder
Bit Inter-leaver
FEC ENCODING
QPSK, 8PSK, 16APSK, 32APSK
Bit mapper into constel-lations
MAPPING
PL Signalling & Pilot insertion
Dummy PLFRAME Insertion
PL SCRAM BLER
PL FRAMING
α=0,35, 0,25, 0,20
BB Filter and Quadrature Modulation
MODULATION
BBHEADER DATAFIELD
BBFRAME
LP stream for BC modes
FECFRAME
PLFRAME
to the RF satellite channel

Figure 2: DVB-S2 Modulation Block Diagram [5]

Details about the DVB-S2 modulation process can be found in the sister project [1]. In keeping with the goals of the project, the processor that performs this part of the chain must be low power and small enough to fit on the platform. Three options became evident. The first is purchasing a DVB-S2 modulation specific IC. However, these are expensive and not easily obtained. Alternatively, modulation could be performed in software using a small single board computer (i.e. Raspberry Pi or Beaglebone). If these computers do not contain enough computing power, the modulation could be moved to a FPGA, which would have to be sized appropriately to fit the area needs of the modulation scheme. The second option, specifically, a Raspberry Pi 3 was chosen first to experiment with, as it seemed it might be able to handle the modulation thanks to its 1.2 GHz quad-core ARM processor [6]. Fortunately, a DVB-S2 modulation suite has been created by amateur radio operator Ron Economos (W6RZ), made available as a suite of GNU Radio blocks [7]. The setup used was a DVB-S2 transmitter modulated at 16APSK, with a ‘normal’ frame size of 16,200, a code rate of 9/10, a roll-off factor of 0.2, a filter with 100 taps, and pilot symbols enabled. The image below shows the GNU Radio block diagram that was executed on the Pi.



Options
ID: dvbs2_tx
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 3M

Variable
ID: symbol_rate
Value: 1.5M

Variable
ID: center_freq
Value: 1.28G

Variable
ID: rolloff
Value: 200m

Variable
ID: taps
Value: 30

TS Source
File: ~/ts_tools/video.fifo

BBheader
FECFRAME size: Normal
Code rate: 9/10
Rolloff factor: 0.20

BBscrambler
FECFRAME size: Normal
Code rate: 9/10

BCH Encoder
FECFRAME size: Normal
Code rate: 9/10

LDPC Encoder
FECFRAME size: Normal
Code rate: 9/10
Constellation: Other

Interleaver
FECFRAME size: Normal
Code rate: 9/10
Constellation: 16APSK

Modulator
FECFRAME size: Normal
Code rate: 9/10
Constellation: 16APSK

Physical Layer Framer
FECFRAME size: Normal
Code rate: 9/10
Constellation: 16APSK
Pilots: On
Gold Code: 0

FFT Filter
Decimation: 1
Taps: firdes.root_raised_c...
Num. Threads: 1

osmocom Sink
Device Arguments: bl...40.rbf
Sample Rate (sps): 3M
Ch0: Frequency (Hz): 1.28G
Ch0: Freq. Corr. (ppm): 0
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 50
Ch0: BB Gain (dB): 50
Ch0: Bandwidth (Hz): 3.5M

Variable
ID: vga2_gain
Value: 50

Variable
ID: vga1_gain
Value: 50

Variable
ID: tx_gain
Value: 50

Figure 3: GNU Radio Software Implementation Block Diagram

Running this suite on the Raspberry Pi 3 yielded a maximum output symbol rate of ~1.5 MSymbols/s before the CPU was unable to keep up. The following equation was developed and is just a product of the efficiencies of each block in the chain. It is used to calculate the input bit rate that would be possible given the maximum output symbol rate that was just found experimentally.

$$Input\ Bit\ Rate = Output\ Symbol\ Rate * (BB\ Header\ Efficiency) * (BCH\ Efficiency) * (LDPC\ Efficiency) * (Modulation\ Efficiency) * (PL\ Framer\ Efficiency)$$

Equation 1: Input Bit Rate versus Output Symbol Rate

Using the efficiency equations in [1], the above equation was reduced to the following:

$$Input\ Bit\ Rate = Output\ Symbol\ Rate * 3.4826800911$$

Equation 2: Input Bit Rate versus Output Symbol Rate (solved)

and the input bit rate is found to be ~5.224 Mbit/s. This, while considerable, could definitely be improved upon. The sister project to this one, titled 'DVB-S2 Transmitter: FPGA Implementation' [1], focused on performing the entire DVB-S2 modulation chain in hardware, specifically a FPGA, which is able to meet the desired throughput requirements for this project. This project uses the code that was developed in Verilog and verified through test vector simulation as a 'black-box', except for some small changes that were made, which will be discussed later on. At the end, the software implementation (GNU Radio) will be compared to the hardware implementation (FPGA) in terms of both the possible bit rate and total system power consumption.

2.2 Transport Streams

The input data to this DVB-S2 transmitter 'black box' shown in Figure 1 is a transport stream. A transport stream is a container format generally used to carry video, but has been extended by the DVB-DATA standard to carry data as well through the use of Multiprotocol Encapsulation (MPE) [8]. Alternatively, one could use Generic Stream Encapsulation (GSE), an exceptionally robust and flexible container format that fully replaces a transport stream. This standard has been defined by DVB [9] to natively carry both video and data, however, it isn't widely supported yet. A paper written discusses these various options in detail [10]. For this project, the transport stream container was chosen, as it is more widely used. This transport stream carries both the live video and telemetry data shown in Figure 1. The format for a transport stream has been specified by ISO/IEC 13818-1 [11].

Each transport stream packet is 188 bytes long. The first 4 bytes consist of a header, the format of which is specified below, followed by a variable length adaptation field (used to hold a program clock reference (PCR) and associated data), followed by the payload which takes up the remainder of the 188 bytes. More details on the transport stream structure can be found in [11].

Name	Sync	TEI	PUSI	Transport Priority	PID	TSC	AFC	CC
Width (bits)	8	1	1	1	13	2	2	4
Typ. Value	0x47	-	-	-	-	0x0	-	-

TEI = Transport Error Indicator

PUSI = Payload Unit Start Indicator

PID = Packet Identifier

TSC = Transport Scrambling Control

ADC = Adaptation Field Control

CC = Continuity Counter

Table 1: Transport Stream Packet Header Structure

2.3 Video Capture and Compression

Ideal video capture for this remote sensing application (high-altitude balloon) would take place at high-definition (1280x720p) and at 15 fps or higher, and be compressed to under 4 Mbit/s. Many off-the-shelf options exist to capture video such as USB webcams and IPTV security camera, however, the most appealing is the Raspberry Pi Camera Module V2. This camera is capable of capturing video at 1920x1080p resolution, at 30 fps, which far exceeds the specifications of the project [12]. Furthermore, all Raspberry Pis natively support H.264 video compression on the onboard GPU for video received from the camera, which can be connected to the Pi using its onboard camera connector. This setup provides a perfect candidate for this project. Video is captured at HD resolution, compressed on the Raspberry Pi's internal GPU, packetized into transport stream packets alongside the telemetry data, and sent to the FPGA to be modulated according to the DVB-S2 standards.

2.4 RF Front End

With regards to the RF front-end shown in Figure 1, many options exist. First, an RF transmitter IC or PCB could be sourced, and then integrated separately with the FPGA to develop the transmit chain. Or, a Software Defined Radio (SDR) board such as the bladeRF or LimeSDR can be purchased which contain both the FPGA and RF front-end on one PCB. For the sake of prototyping and to reduce initial development efforts, the bladeRF SDR board was purchased. The RF front-end on this board is a LMS6002D, which has a frequency range of 300 MHz – 3800 MHz, and a bandwidth of 28 MHz [13]. Typical satellite transmission is done at 950 – 3000 MHz (post-LNB) with a maximum bandwidth of 27 MHz, so this board does suffice. This board contains a 40kLE Altera Cyclone IV FPGA and it has been proven that the DVB-S2 transmitter code could fit comfortably on the FPGA [1].

3. Design and Development

This section focuses on the design details for each of the portions of the system described above. For the DVB-S2 modulation aspect of the system, the focus was placed on integrating the HDL written in [1] into the bladeRF FPGA architecture, and ensuring compatibility with the LMS6002D RF front-end for data transmission and with the Raspberry Pi for data input. The video compression, thanks to the work done by the developers behind the Raspberry Pi, is already functional, so the main task was to get their API to capture and compress video data from the camera. Capturing telemetry is outside the scope of this project as it is application dependent. However, during this portion of the paper a generic data transfer scheme that works over a transport stream is being introduced and has been integrated into the system, so the end user can use it to send any data they wish. Once video and data systems have been built, the focus shifted towards encapsulating these into transport streams and multiplexing the two streams together.

3.1 DVB-S2 Modulation

The DVB-S2 module written in [1] is used here, with some slight modifications. This specific DVB-S2 transmitter was designed to use 16-APSK modulation, with a frame size of 16,200 (normal), a code rate of 9/10, and pilot symbols enabled, and this project’s use of it is consistent in those settings. Given the clocks that serve as inputs to the system are provided as designed and at the correct rates, data is required to be inputted at a rate of 4 Mbit/s, and symbols will be outputted at a rate of 1.1485 MSymbols/s. This is constant and is not configurable unless the input clock rates (2 MHz, 4 MHz, 8 MHz, and 48 MHz) are changed. The following figure shows the input and output interfaces for this DVB-S2 system, including the clocks:

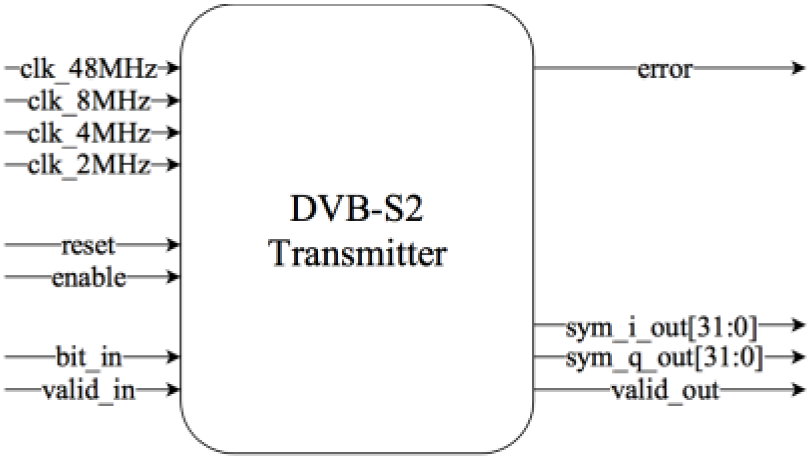


Figure 4: DVB-S2 Transmitter Interface

This transmitter accepts data serially, meaning that 1 bit is pumped in at a time. The input bit rate is 4 Mbps, though a valid bit does not need to be present precisely every 4 MHz. Thanks to the valid_in signal, a bit is only read when it is specified as valid. Despite this flexibility, one must make sure that the average bit rate into the system is exactly 4 Mbps or there will be errors. Symbols are outputted from the system in complex (I/Q) floating point (IEEE 754) format at a rate of 1.1485 MSymbols/s.

Based on amateur radio guidelines, 3 MHz of bandwidth can be allocated to this transmitter at an operating frequency of 1.28 GHz (23cm band). According to [14], the RF bandwidth is correlated to the symbol rate as follows:

$$RF\ Bandwidth = 1.2 * Symbol\ Rate$$

Equation 3: RF Bandwidth versus Symbol Rate

Therefore, with an RF bandwidth of 3, the symbol rate can be as high as 2.5 MSymbols/s. This is more than double than the 1.1485 MSymbols/s that the current DVB-S2 module is designed for. Therefore, the module was modified to output symbols at 2.297 MSymbols/s (meaning an input bitrate of 8 Mbit/s) in order to fully utilize this RF bandwidth. This was done by doubling all the clock speeds. In addition, the module has been modified to output symbols in SC16Q11 format, as required by the LMS6002D front end. SC16Q11 is just a 12-bit twos-complement binary representation of the I/Q symbol value. I/Q symbols have a range of [-1, 1], and the SC16Q11 symbols represent these fractions in a range of [-2047, 2048]. Given the following I/Q symbol diagram for 16APSK taken from [5]:

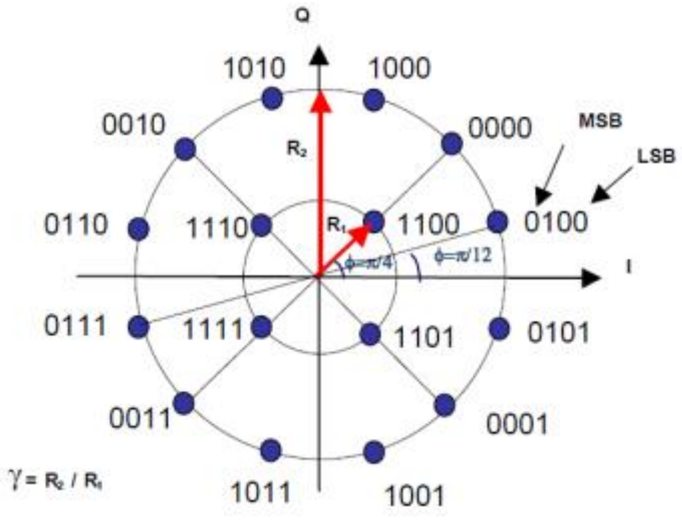


Figure 5: 16APSK Modulation

the following table has been constructed, showing the fractional and Q11 representations of each of the 16 possible I/Q symbols.

Symbol	Fractional – I	Q11 – I (hex)	Fractional – Q	Q11 – Q (hex)
0000	0.707106	0x586	0.707106	0x586
0001	0.707106	0x586	-0.707106	0xa58
0010	-0.707106	0xa58	0.707106	0x586
0011	-0.707106	0xa58	-0.707106	0xa58
0100	0.965926	0x7ba	0.258819	0x212
0101	0.965926	0x7ba	-0.258819	0xdee
0110	-0.965926	0x846	0.258819	0x212
0111	-0.965926	0x846	-0.258819	0xdee
1000	0.258819	0x212	0.965926	0x7ba
1001	0.258819	0x212	-0.965926	0x846
1010	-0.258819	0xdee	0.965926	0x7ba
1011	-0.258819	0xdee	-0.965926	0x846
1100	0.275139	0x233	0.275139	0x233
1101	0.275139	0x233	-0.275139	0xdcd
1110	-0.275139	0xdcd	0.275139	0x233
1111	-0.275139	0xdcd	-0.275139	0xdcd

Table 2: 16APSK I/Q Symbol Values

The updated diagram is shown below, showing the faster clock speeds and the 12 bit I/Q symbol output:

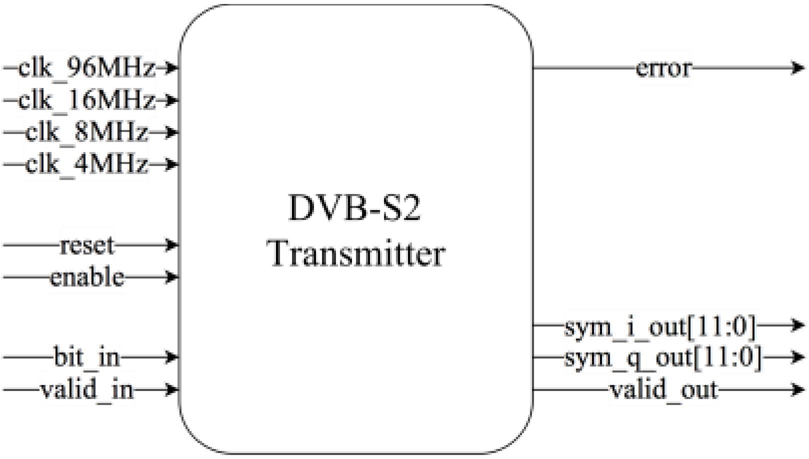
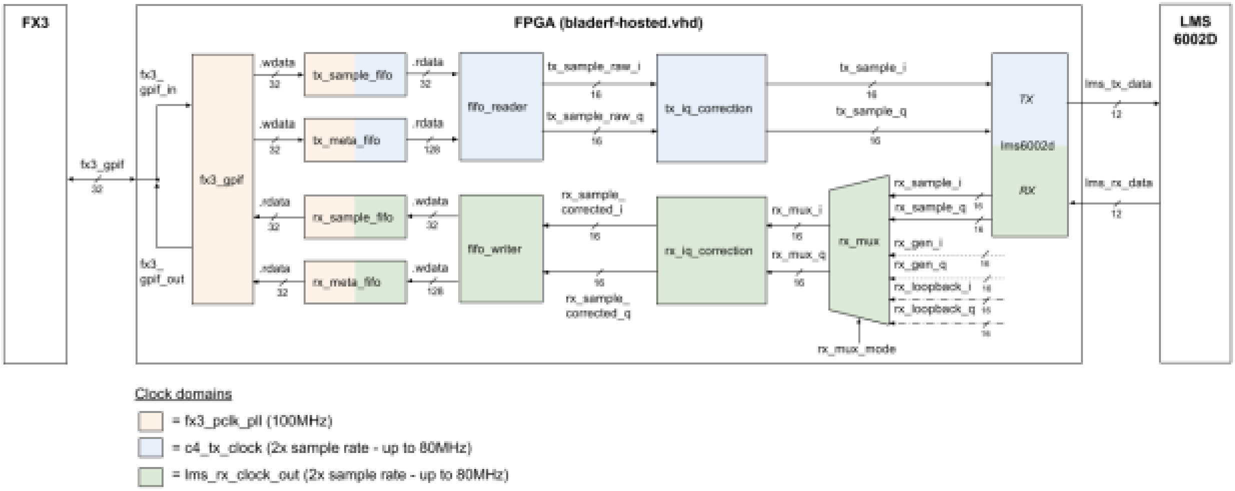


Figure 6: DVB-S2 Transmitter Interface (after updates)

The next step is to adapt this to fit in the FPGA on the bladeRF board. The current 'stock' architecture pre-programmed into the FPGA is shown in the figure below, from [15]. This architecture simply does I/Q symbol pass-through, meaning that I/Q symbols must be presented over USB as an input, where they will then be transmitted. The goal of this project is to present transport stream data as an input to the FPGA and have the DVB-S2 block generate the I/Q samples from this transport stream data within the FPGA, where they will then be transmitted.



FX3 = USB3 Interface

LMS6002D = RF Front End

Figure 7: BladeRF Default FPGA Architecture [15]

The custom DVB-S2 modulation code must be inserted somewhere in the TX chain. After studying the contents of the 'fifo_reader' block, it was determined that this block could be removed and replaced with custom code. USB data (transport stream packets) are received on the 32 bit .rdata bus. In order to adapt this 32 bit input to the bit serial data input on the DVB-S2 block, a custom module was written named 'dvb_fifo_reader'. This block must be able to read the 32 bit USB data from the previous FIFO ('tx_sample_fifo'), and output the transport stream data one bit at a time. In addition, this block must be able to adjust for underflows, or unavailability, of data coming in via USB. If there isn't enough data to pass to the DVB-S2 block at 8 Mbps, the block must insert a 'null' transport stream packet. The ISO/IEC 13818-1 standard [11] defined a null packet with PID=0x1fff, and the payload of the packet is blank data consisting of all '0xff'. The diagram below illustrates now null packets are inserted between video and data packets, when neither of those are available for transmission.

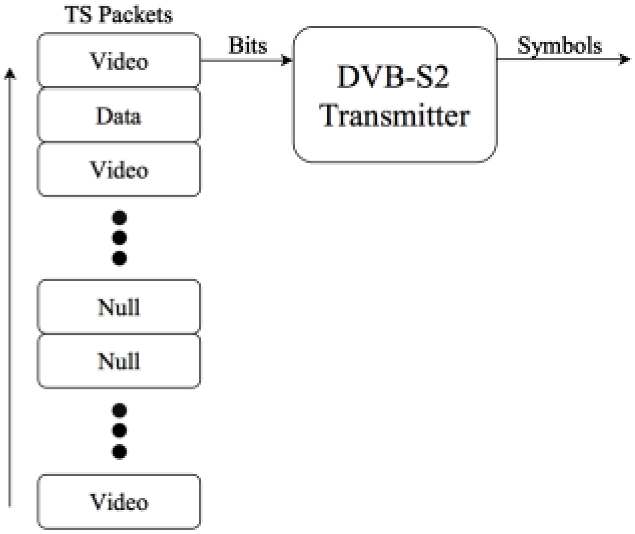


Figure 8: Null Packet Insertion Operation

The interface for this new ‘dvb_fifo_reader’ block is shown below, which replaces the ‘fifo_reader’ block shown in Figure 7.

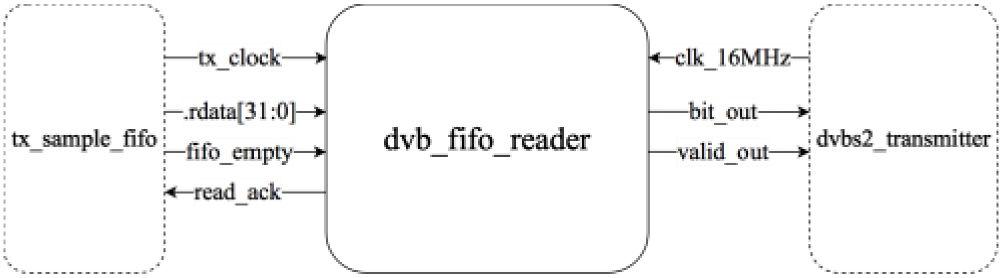


Figure 9: Custom Block ‘dvb_fifo_reader’ Interface

Based on the characteristics of the USB transmit data fifo (tx_sample_fifo), and the characteristics of the input to the DVB-S2 transmitter, the following high-level flow diagram was developed. Notice that the module stores an entire transport stream packet (188 bytes) before it is transmitted.

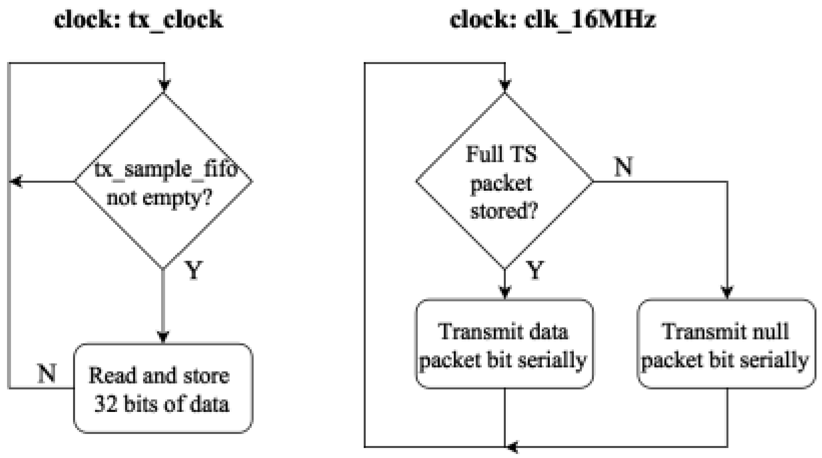

clock: tx_clock
tx_sample_fifo not empty?
Y
N
Read and store 32 bits of data
clock: clk_16MHz
Full TS packet stored?
N
Y
Transmit data packet bit serially
Transmit null packet bit serially

Figure 10: Custom Block 'dvb_fifo_reader' Operation

This module makes it possible to use the existing bladeRF API by transmitting the transport stream data in place of I/Q symbols. The module corrects the endianness of the data so that the user simply needs to specify a transport stream file (typically .ts) in place of the usually binary (.bin) file containing I/Q symbols.

Now that the input to the DVB-S2 module has been properly mated to the current bladeRF architecture through the module dvb_fifo_reader, it is time to focus on the output of the DVB-S2 module. As discussed, the module outputs I/Q symbols at 2.297 MSymbols/s. The problem with this is that it's not synchronized to any clock, and therefore there could be a slight mismatch between the symbol rate and the clock rate. This could cause symbols to be dropped or the system to underflow because it does not have enough symbols to transmit. The designers of DVB-S2 modulation scheme took this into consideration, and allow for the insertion of dummy PL frames in the last stage of the modulation process (PL framing – see Figure 2). The structure of a dummy PL frame is given below, as defined by the DVB-S2 standards [5]:

Name	**PL Header**	**Dummy Frame**	**Pilot Symbol**	**Dummy Frame**	**Pilot Symbol**	**Dummy Frame**
# Sym	1 slot	16 slots	36 symbols	16 slots	36 symbols	4 slots
		← SCRAMBLED →				

1 slot = 90 symbols

For every pilot symbol and every dummy symbol: I=0.707 and Q=0.707

Table 3: Dummy PLFrame Structure

Based on this structure, each dummy PL frame has 3402 symbols in it, as opposed to a normal PL frame which has 16,686. To insert dummy PL frames in between normal PL frames when underflows occur, two things need to happen. First, the end of each normal PL frame need to be recognized so that either another normal PL frame (if data is ready) or a dummy PL frame (if data is not ready) can be inserted. Second, the output symbol rate needs to be increased to allow time for dummy frames to be transmitted while still keeping output data received from the PL framer block from backing up.

With those in mind, the following figure shows the high-level interface of the new ‘output_sync’ block implements this functionality. It is attached to the output of the PL framer block, passes I/Q symbols divided into PL frames (with dummy frames inserted) to the FIR filter, which interfaces directly to the ‘tx_iq_correction’ block in the existing bladeRF FPGA architecture shown in Figure 7.

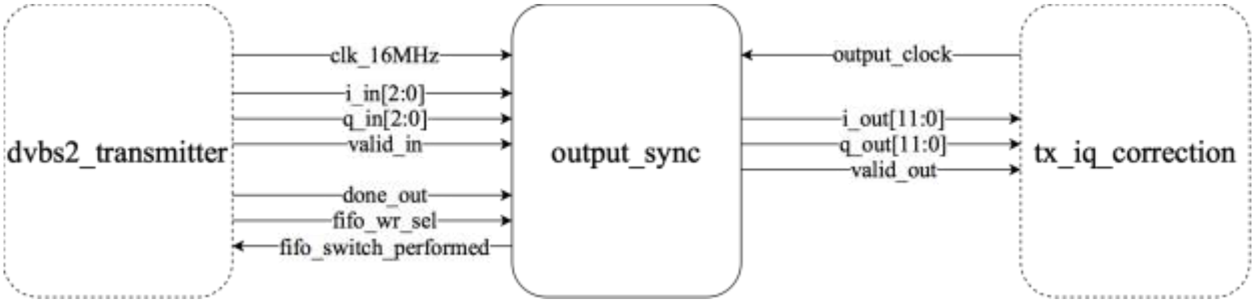


Figure 11: Custom Block 'output_sync' Interface

Three new signals were added to the PhyFramer block. The first is 'done_out', which is sent high every time a full PL frame is transmitted. The 'output_sync' block must set the 'fifo_switch_performed' signal high to acknowledge this done flag, after which the 'done_out' signal is reset by the PhyFramer block. The signal 'fifo_wr_sel' is used to tell which FIFO the 'output_sync' block should write to. There are two FIFOs within 'output_sync', each capable of storing a whole PL frame for the purposes of minimizing downtime and therefore the number of dummy PL frames that need to be transmitted.

The follow diagram shows the operation of the 'output_sync' block, which is to be described in Verilog HDL.

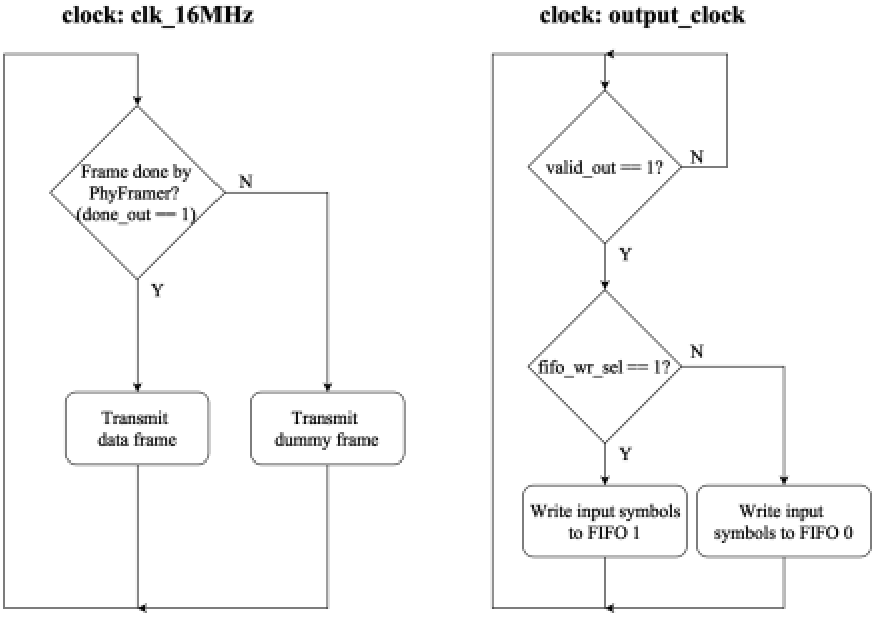


Figure 12: Custom Block 'output_sync' Operation

This module solves the problems identified above by inserting dummy PL frames into the stream of I/Q symbols (that are already organized into PL frames). To store 2 full PL frames, a significant amount of memory is required. In order to reduce this memory requirement, each I/Q symbol in all blocks that use I/Q symbols as their base symbol (bit mapper, physical framer, and output_sync) are converted from a 12 bit to a 3 bit representation, based on the table shown below. In the 'output_sync' block, they are converted back to 12 bits for transmission. To make room for these dummy frames, the effective symbol rate is increased to 2.6 MSymbols/s. However, only 2.297 MSymbols/s will be actual data, the remainder will be dummy frames. This symbol rate corresponds to a sample rate of 5.2 MSamples/s.

Original 12-bit representation	**New 3-bit symbol**
0x5a8	000
0x7ba	001
0x212	010
0x233	011
0xdcd	100
0xdee	101
0x846	110
0xa58	111

Table 4: Simplified Representation of Possible I/Q Symbol Values

Given all that was discussed, the following is a list of all the changes/additions made to the DVB-S2 transmitter module:

General:

- Addition of the dvb_fifo_reader input block
- Addition of the output_sync output block
- Doubled all clock speeds
- Changed all flip-flops to asynchronous reset

Bitmapper:

- Converted I/Q symbols from 32-bit floating point (IEE754) to 12-bit Q11 symbols
- Converted 12-bit Q11 symbols to the 3-bit values specified in Table 2

Phyframer

- Converted I/Q symbols from 32-bit floating point (IEE754) to 12-bit Q11 symbols
- Converted 12-bit Q11 symbols to the 3-bit values specified in Table 2
- Addition of signals to aid in dummy PL frame insertion

The DVB-S2 transmitter module, with all its changes, is now compatible to be integrated into the bladeRF FPGA architecture show in Figure 7. As mentioned, the module replaces the existing ‘fifo_reader’ block. In addition, all blocks instantiated specifically for receiving and for metadata have been removed to make room in the FPGA for the additional logic and memory elements.

Quartus was used to develop and compile this complete transmitter system, and generate the FPGA image that needs to be loaded onto the bladeRF’s Cyclone IV FPGA. With the FPGA image generated (‘withdvb.rbf’), the following command can be used with the bladeRF API to load the FPGA and to feed in a transport stream file for transmission.

```
bladeRF-cli -e \                          // Open the bladeRF API in execute mode
        "load fpga withdvb.rbf; \         // Load the FPGA with the specified image
        set frequency 1280000000; \       // Set the transmit frequency to 1.28 GHz
        set bandwidth 3500000; \                   // Set the bandwidth to 3.5 MHz
        set samplerate 5200000; \            // Set the sample rate to 5.2 MSamples/s
        tx config file=video.ts; \             // Input file is now a transport stream
        tx config repeat=1; \                   // Repeat the transport stream once
        set txvga1 -4; \                                              // VGA1 gain
        set txvga2 10; \                                              // VGA2 gain
        tx start; \                                            // Start transmission
        tx wait;"                             // Block until transmission is finished
```

Command 1: BladeRF Transmission

Onboard the bladeRF PCB, the LEDs (LED1 - LED3) and expansion ports (mini-exp1 and mini-exp2) are configured to provide debug information as follows:

I/O	**Configuration**
LED1	dvb_error (DVB transmission error)
LED2	Heartbeat synchronized to clk_16MHz
LED3	Always on when FPGA image loaded
mini_exp1	dvb_valid_in (valid bit in to DVB transmitter)
mini_exp2	tx_sample_raw_valid (valid symbol generated by DVB transmitter)

Table 5: BladeRF GPIO and LED Functionality

The following image of the bladeRF board shows the exact locations of these debug I/Os:

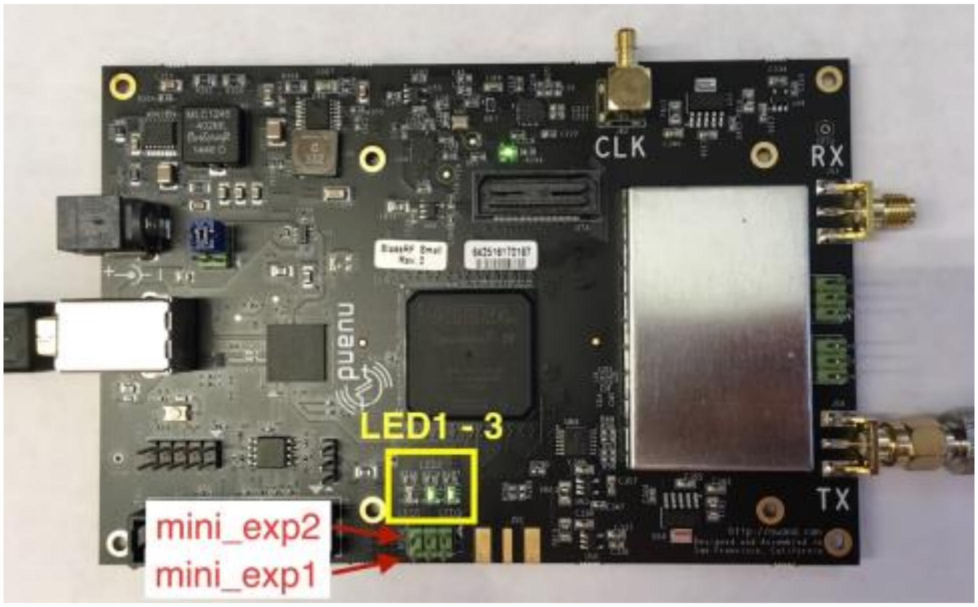


Figure 13: BladeRF GPIO and LED Locations

With a working transmission chain, we can now focus on the actual data we wish to transmit, which must be contained in a transport stream.

3.2 Video Capture and Compression

As mentioned, the Raspberry Pi contains a full video capture and compression system. Specifically, the Raspberry Pi 3 has an onboard camera connector, which connects to the Raspberry Pi Camera Module V2. The Raspberry Pi Camera Module V2 contains an 8 Megapixel sensor, capable of capturing video at up to 1920x1080p resolution, and up to 60 fps (at 720p) [12]. It communicates with the Pi 3 using a MIPI CSI2 interface. Onboard the Pi 3, the Broadcom VideoCore IV GPU contained within the BCM2837 SoC is capable of performing H.264 compression on this captured video [6]. The GPU runs at 400 MHz and shares 1 GB of RAM with the CPU. The image below shows the Raspberry Pi 3 setup with its camera connected.

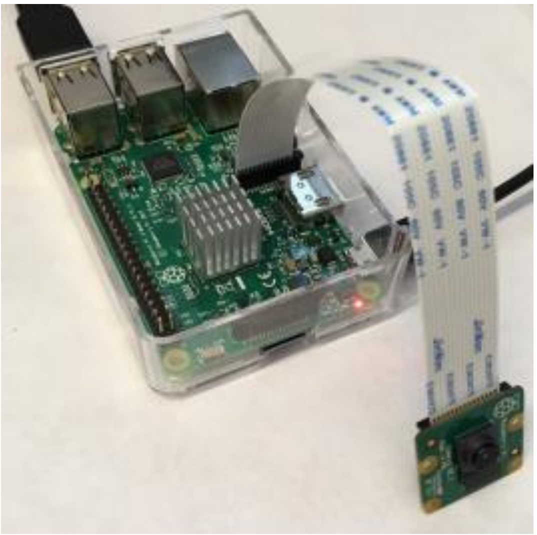


Figure 14: Raspberry Pi 3 Capturing Video through the Camera Module V2

Capturing video is performed using Command 2, the executable of which is preinstalled on recent distros of Raspbian. The command below was used to capture and compress video, and outputs raw H.264 packets (not contained in a transport stream) to standard out.

```
raspivid                        // Capture video from the Rasperry Pi camera
        -n \                          // Don’t output the video to the display
        -t 0 \                                     // Capture video infinitely
        -fps 25 \                           // Capture at 25 frames per second
        -w 1280 \                                // Video width of 1280 pixels
        -h 720 \                                 // Video height of 720 pixels
        -pf high \                           // Use the H.264 ‘High’ profile
        -vf \                          // Flip the video vertically (optional)
        -hf \                        // Flip the video horizontally (optional)
        -b 1000000 \                        // Output of approximately 1 Mbps
        -o -                     // Output compressed video to standard out
```

Command 2: ‘raspivid’ Command Capturing and Compression Video

A resolution of 720p was chosen, because it has a larger field of view than 1080p, as shown in the figure below pulled from [16]:

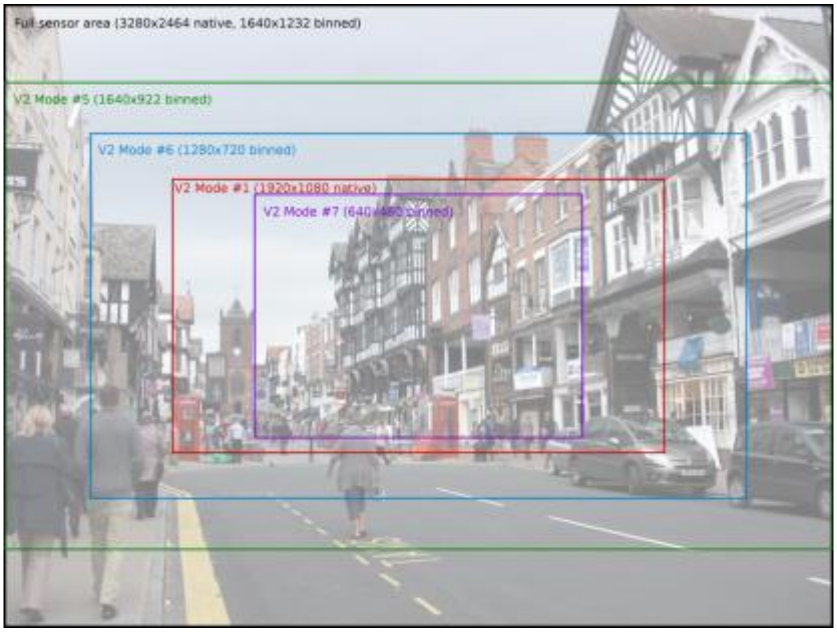

Full sensor area (3280x2464 native, 1640x1232 binned)
V2 Mode #5 (1640x922 binned)
V2 Mode #6 (1280x720 binned)
V2 Mode #1 (1920x1080 native)
V2 Mode #7 (640x480 binned)

Figure 15: Raspberry Pi Camera Module V2 Field of View Options

A bit rate of 1 Mbps, as specified in Command 2, outputs a video of sufficient quality that reacts well to rapid changes in scenery without distortion.

3.3 Transport Stream Encapsulation

This H.264 video data, which is supported natively by most satellite receivers, must be encapsulated or contained in a transport stream to be properly transmitted and received. The video conversion suite FFMPEG contains an output format specifier for transport streams called ‘mpegts’, which is used to perform this encapsulation. The following FFMPEG command was used. FFMPEG documentation provides information on how to use the command and its associated options [17].

```
ffmpeg \
        -re \                          // Receive data as it is produced on the input
        -an \                                                          // No audio
        -i - \                                      // Receive data from standard input
        -vcodec copy \             // Don’t perform any video compression, just copy
        -f mpegts \                                      // Encapsulate in TS packets
        -mpegts_flags system_b \                         // Conform to DVB standards
        -                                       // Output TS packets to standard out
```

Command 3: Transport Stream Encapsulation using ‘ffmpeg’

Combining this command with the video capture command specified in Command 2, the following command was synthesized to capture video, perform H.264 compression, and transport stream encapsulation:

```
raspivid -n -t 0 -fps 25 -w 1280 -h 720 -pf high -vf -hf -b 1000000 -o - | ffmpeg -re -an -i -
-vcodec copy -f mpegts -mpegts_flags system_b - > video.fifo
```

Command 4: Combined Video Capture, Compression, and Encapsulation

The transport stream packets are output to the ‘video.fifo’ file. This file has been setup as a named pipe in linux, generated using the ‘mkfifo’ command. For details on how named pipes work see [18]. This file is essentially a transport stream file, so it can be read using the bladeRF-cli command developed in Command 1.

```
bladeRF-cli -e "load fpga withdvb.rbf; set frequency 1280000000; set bandwidth 3500000;
set samplerate 5200000; tx config file=video.fifo; tx config repeat=1; set txvga1 -4; set
txvga2 10; tx start; tx wait;"
```

Command 5: BladeRF Transmitting Video

As video data is generated, the bladeRF-cli command reads it in and transmits it through the bladeRF board. Because a named pipe is being used, bladeRF-cli does not exit when it stops seeing data at ‘video.fifo’; it simply keeps waiting until there is data present. The transmitter will not suffer errors during this period of inactivity, as the extra bandwidth is filled (about 8 Mbps – 1 Mbps due to video = 7 Mbps) with null packets in the ‘dvb_fifo_reader’ block.

3.4 Data Generation

A multiprotocol encapsulation (MPE) is used to transmit data alongside video within a transport stream. More details about how multiprotocol encapsulation works can be found in [8]. The complexity of this protocol pushes it slightly outside of the scope of this project to be fully developed from scratch, however, an implementation using OpenCaster open source software demonstrates the capabilities of MPE. OpenCaster is a free and open source MPEG2 transport stream data generator and packet manipulator. It contains many utilities used in encapsulating various types of data in transport stream packets. The original OpenCaster repository is located at [19], however, some slight modifications had to be made to these utilities, and a new one had to be added. Therefore, a fork of this repository was made. A summary of the utilities used and the changes made are summarized below:

mpe2sec

Summary: opens an IP tunnel, such that any data presented to the tunnel is converted to an intermediate “SEC” format

Changes: None

sec2ts

Summary: converts this intermediate “SEC” format to a transport stream

Changes: Fixed an issue that prevented single sections from being transmitted live

ts2sec

Summary: converts a transport stream to the intermediate “SEC” format

Changes: Allowed input from stdin as an option

sec2mpe (added to the suite)

> Summary: opens an IP tunnel, such that any data found in the intermediate “SEC” file is presented to the IP tunnel for reading
>
> Changes: N/A

With this collection of utilities, ‘mpe2sec’ and ‘sec2ts’ are being used on the transmitter side. The ‘mpe2sec’ utility opens an IP tunnel, and by piping the output of ‘mpe2sec’ to ‘sec2ts’, any data placed on the tunnel by an application is encapsulated as MPE and then placed into a transport stream. This transport stream is then transmitted over DVB-S2, and received again as a transport stream.

On the receiving computer, the received transport stream is piped into the input of ‘ts2sec’, the output of which is piped into the command ‘sec2mpe’. This will open an IP tunnel, and any MPE data found on the transport stream is presented to the tunnel. Any application can read the data presented on the tunnel.

The following diagram illustrates this setup:

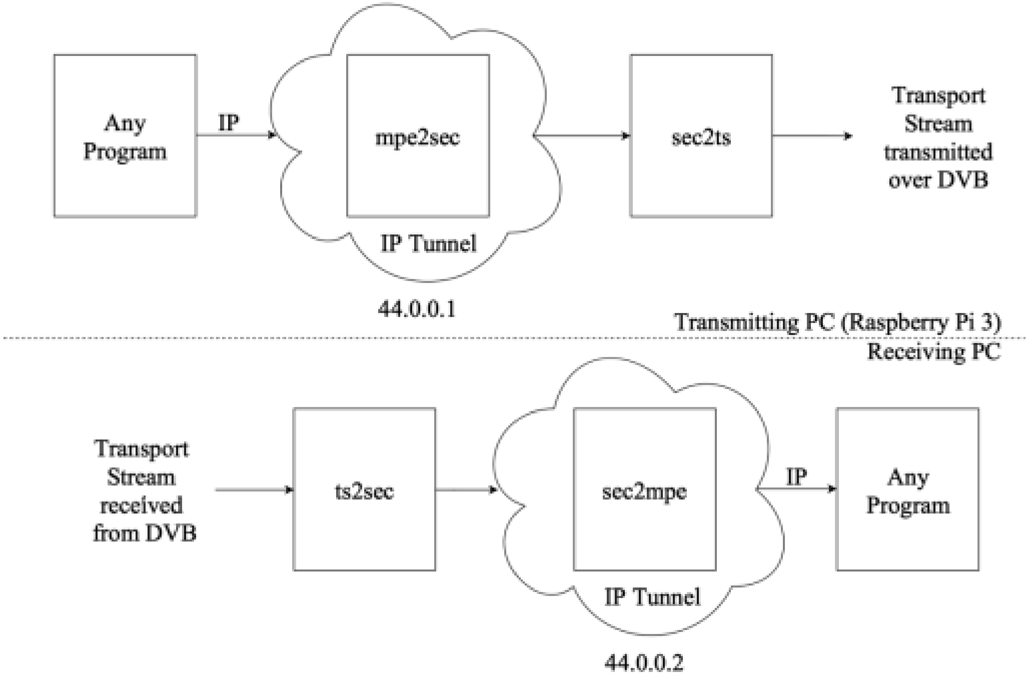

Any Program
IP
mpe2sec
IP Tunnel
44.0.0.1
sec2ts
Transport Stream transmitted over DVB
Transmitting PC (Raspberry Pi 3)
Receiving PC
Transport Stream received from DVB
ts2sec
sec2mpe
IP
Any Program
IP Tunnel
44.0.0.2

Figure 16: Data Transmit Block Diagram

The following command is used on the Raspberry Pi 3 to open an IP tunnel, such that data presented to the tunnel appears as MPE data within the transport stream “data.ts”:

```
sudo mpe2sec \          // Create an IP tunnel, generate ‘SEC’ data from data placed there
    dvbtx \                                                      // Name the tunnel ‘dvbtx’
    | sec2ts \            // Pipe data to sec2ts, which converts the ‘SEC’ data to TS data
    430 \                                                   // PID of the data is 430 = 0x1ae
    -s \                                                                  // Section the data
    > data.ts                                          // Pipe the transport data to ‘data.ts’
```

Command 6: Generate a Transport Stream with MPE Encapsulation Data

On the receiving PC, a similar command is used to convert the received transport stream into an IP tunnel:

```
cat data.ts | ts2sec \            // Pipe the received TS file to ts2sec to convert to ‘SEC’
        - \                                        // Have ts2sec pull from standard in
        430 \                                        // Filter PID 430 from the TS file
        | sudo sec2mpe \            // Pipe the ‘SEC’ data and generate an IP tunnel
        dvbrx                      // Present the data on an IP tunnel named ‘dvbrx’
```

Command 7: Interpret a Transport Stream with MPE Encapsulation Data

3.5 Multiplexing Transport Streams

With a video transport stream generated using FFMPEG, and a data transport stream generated using OpenCaster, the two must now be ‘multiplexed’ or merged. A simple utility was written in C to do this, called ‘stream_mux’. This utility simply interleaves transport stream packets of video and data with each other, where each transport stream packet is 188 bytes. Since video is generated at a constant bit rate (~1.0 Mbps), this serves as the reference bit rate for adding data packets. Data can be generated instantaneously (in the case of a file transfer), and so there needs to be a way to limit the speed in which data is sent such that it doesn’t hog the entire 8 Mbps of possible transmit bandwidth, leaving no room for the 1 Mbps of video.

This was achieved by limiting the number of data packets that can be sent for each packet of video. Currently it is setup (as a constant in the C program) to transmit 2 data packets for each video packet. This, in theory, could be increased up to 6 data packets for each video packet. Transmission of video packets is blocking, meaning that if there isn’t a video packet available, ‘stream_mux’ will stall until there is. However, transmission of data packets is non-blocking, meaning that if there isn't a data packet available the program just passes through and goes back to checking for a video packet.

An operational flow diagram for 'stream_mux' is shown below:

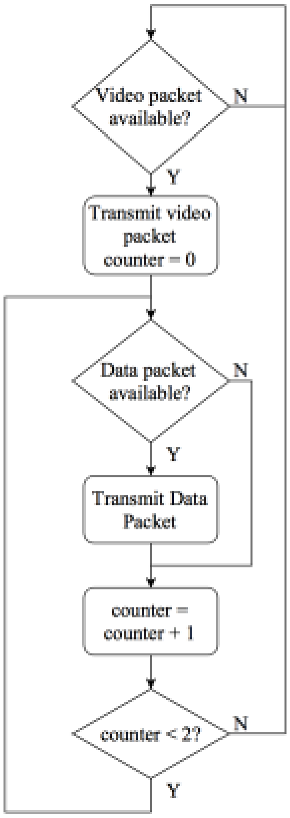


Figure 17: Custom C Code 'stream_mux' Operation

This utility is used (after compiling) as follows:

```
./stream_mux > combined.ts
```

Command 8: Custom C Code 'stream_mux' Usage

The input transport streams (data and video) must be named pipes in the same directory as the executable and are specified as constants in the code as ‘data.fifo’ and ‘video.fifo’, respectively. The entire system for transmitting data and video is now complete, and is summarized in the flow diagram below:

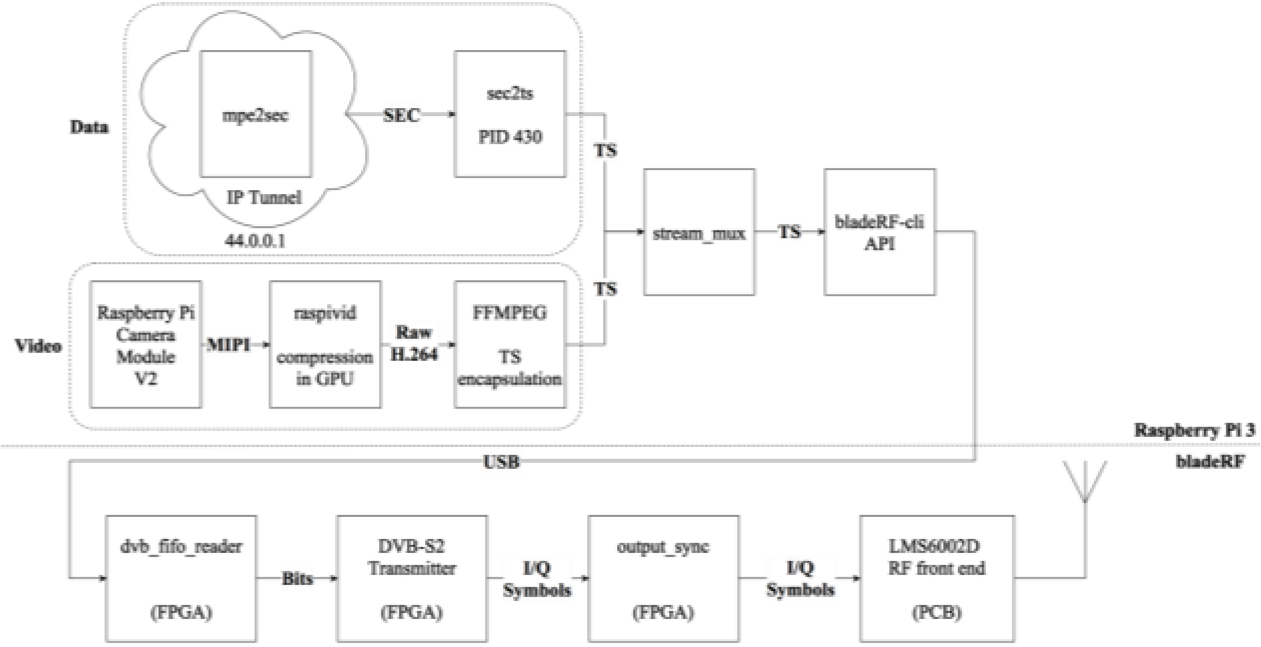


Figure 18: Detailed System Diagram

The following Experimental Setup section will explain how to use the commands mentioned in this section together to create the integrated system shown in the figure above. That section will also discuss how to configure off-the-shelf receivers to receive these streams.

4. Experimental Setup

This section explains how all pieces discussed above are setup as a transmission system, and how the receiving hardware is setup to receive the transmitted video and data. The following commands, when executed in order and from the same directory on the Raspberry Pi 3 with the bladeRF connected via USB, captures, compresses, and transmits live video and data via DVB-S2. The FPGA image ‘withdvb.rbf’ must be in the same directory, and named pipes ‘combined.fifo’, ‘video.fifo’, and ‘data.fifo’ must be created using the ‘mkfifo’ linux utility.

```
bladeRF-cli -e "load fpga withdvb.rbf; set frequency 1280000000; set bandwidth 3500000; set samplerate 5200000; tx config file=combined.fifo; tx config repeat=1; set txvga1 -4; set txvga2 10; tx start; tx wait;"

./stream_mux > combined.fifo

raspivid -n -t 0 -fps 25 -w 1280 -h 720 -pf high -vf -hf -b 1000000 -o - | ffmpeg -re -an -i - -vcodec copy -f mpegts -mpegts_flags system_b - > video.fifo

sudo mpe2sec dvbtx | sec2ts 430 -s > data.fifo
```

Command 9: Commands Required to Create a Transmission System

Finally, one must setup the IP tunnel as an IP interface, using the following command:

```
sudo ip addr add 44.0.0.1/24 broadcast 44.0.0.255 dev dvbtx;

sudo ip link set dvbtx up;

sudo ip link set dvbtx arp off;
```

Command 10: Setup the Transmitting IP Tunnel as an IP Interface

With these four commands running actively in the background on the Pi, the transmitter setup functions as expected. Video is captured and compressed using the ‘raspivid’ command, and encapsulated into a transport stream using the ‘ffmpeg’ command. Data is encapsulated into a transport stream using the ‘mpe2sec’ and ‘sec2ts’ commands. The transport streams are muxed together using ‘stream_mux’, the output of which is read by the bladeRF API and sent to the FPGA over USB. The FPGA is configured to perform DVB-S2 modulation onboard, and transmit the symbols using the LMS6002D front end.

To capture the video stream, a satellite receiver compatible with DVB-S2 and H.264 compression was purchased. Many low-cost options are available online, the one purchased (iSmart F1HD) [20] was designed to receive free-to-air satellite broadcasts. The DVB-S2 transmitter implemented mimics a free-to-air satellite TV station. Normally, when receiving a satellite TV station, a satellite dish is required, which receives frequencies on the order of 10-11 GHz. This 10 – 11 GHz signal is then down-converted to 950 – 2150 MHz before it reaches the receiver through the use of a low noise block downconverter (LNB). So, by transmitting at 1.28 GHz, the system falls in the range of the post-LNB receiving frequencies, and within the range of the 23cm amateur radio band. 23cm antennas are required for wireless transmission, but for bench testing a coaxial cable with a 10dB attenuator connects the output of the bladeRF transmitter to the receiver. The image below shows the system connected together:

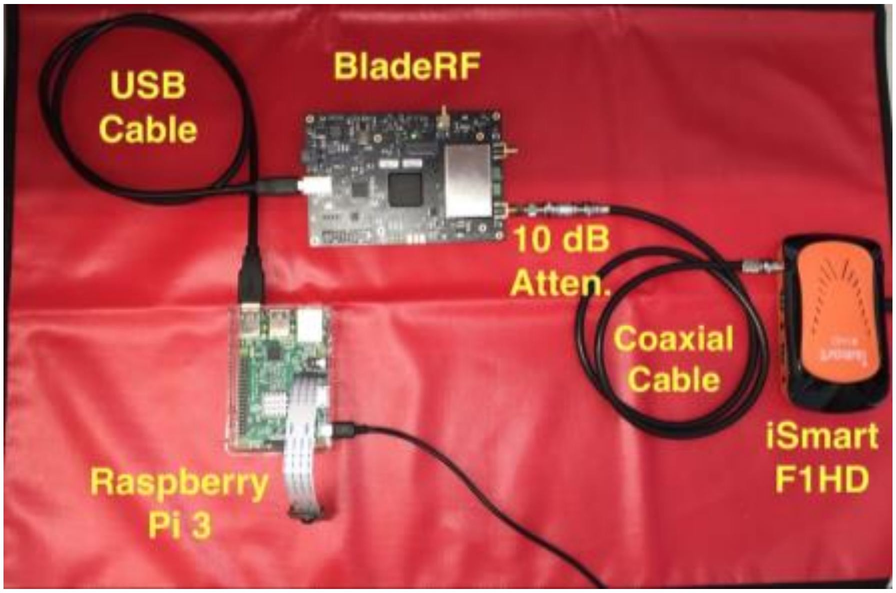
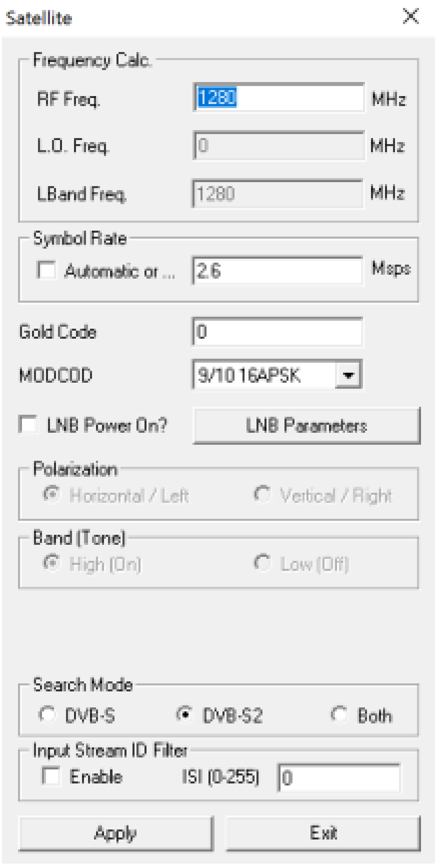


Figure 19: Transmit System Connected to the Receiver

The iSmart F1HD or equivalent satellite receiver must be configured to receive at a frequency of 11.030 GHz (the pre-LNB frequency that corresponds to a post-LNB frequency of 1.28 GHz) and a symbol rate of 2600 kSymbols/s.

To receive transmitted data encapsulated as MPE, a separate and more specialized satellite receiver must be used. The Novra S300V [21] was purchased, as it supports data receiving in the form of MPE. The S300V has an Ethernet interface that allows it to pass data received from the DVB-S2 transmission to a host computer. The following dialog shows the required settings for the satellite receiver:



Satellite

Frequency Calc.

RF Freq. 1280 MHz

L.O. Freq. 0 MHz

LBand Freq. 1280 MHz

Symbol Rate

□ Automatic or ... 2.6 Msps

Gold Code 0

MODCOD 9/10 16APSK

□ LNB Power On?

Polarization

● Horizontal / Left ○ Vertical / Right

Band (Tone)

● High (On) ○ Low (Off)

Search Mode

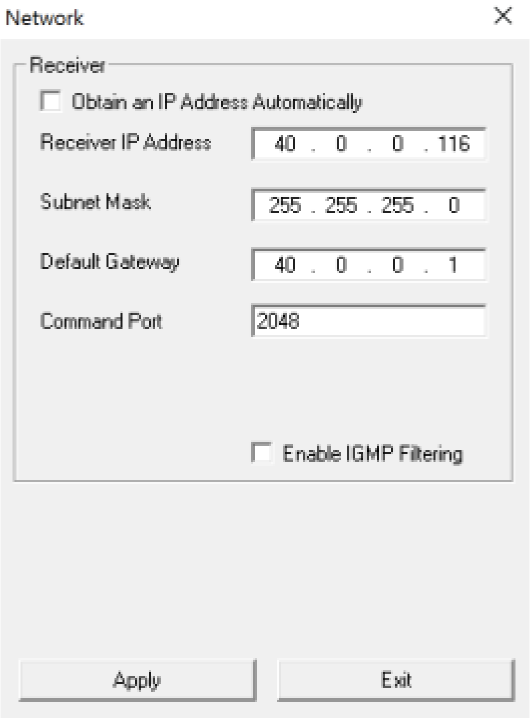
○ DVB-S ● DVB-S2 ○ Both

Input Stream ID Filter

□ Enable ISI (0-255) 0

Figure 20: Data Receiver Satellite Configuration

In addition, the receiver is configured to have an IP address of 40.0.0.116 and with a gateway of 40.0.0.1:



Network

Receiver

□ Obtain an IP Address Automatically

Receiver IP Address 40 . 0 . 0 . 116

Subnet Mask 255 . 255 . 255 . 0

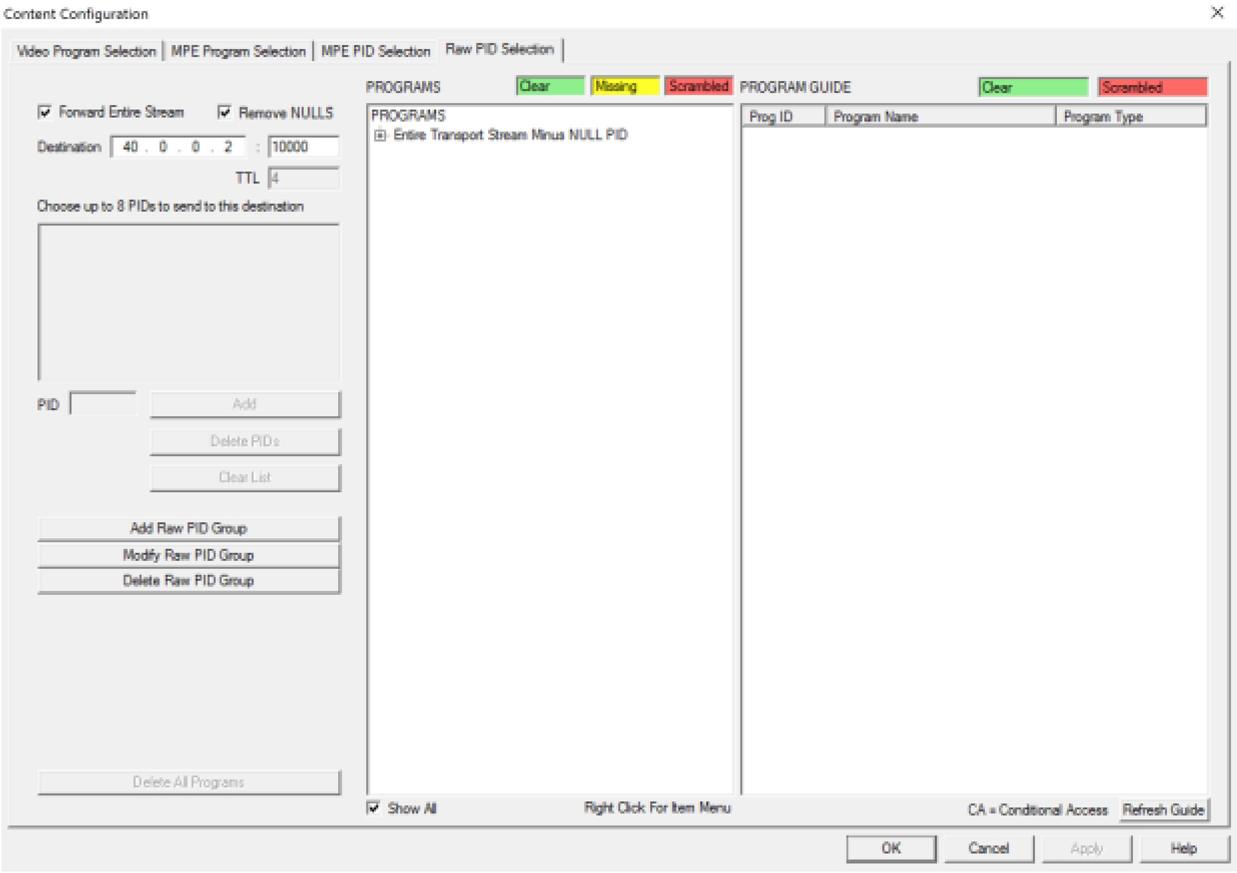
Default Gateway 40 . 0 . 0 . 1

Command Port 2048

□ Enable IGMP Filtering

Figure 21: Data Receiver Network Configuration

Finally, all received transport stream data (video and data) is forwarded to port 10000 of the host PC for processing (40.0.0.2 in this case, the PC must be setup with this static IP), as shown below:



Content Configuration

☑ Forward Entire Stream ☑ Remove NULLS

Destination 40 . 0 . 0 . 2 : 10000

TTL 4

Figure 22: Data Receiver Content Configuration

On the host PC, one can view the video by listening on port 10000 for UDP packets using VLC media player. Alternatively, one can receive transmitted data using the OpenCaster tools. Netcat (nc) [22] is used to listen on port 10000 of the ethernet interface, and the data received is piped to the OpenCaster utilities. Received data is presented on the dvbrx tunnel.

```
nc -ul 10000 | ts2sec - 430 | sudo sec2mpe dvbrx
```

Command 11: Commands Required to Setup a Receive System

To make this data accessible, the dvbrx IP tunnel was setup as an IP interface as shown below, and any packets sent over the link to the IP address 44.0.0.2 via this IP interface are received on this host computer.

```
sudo ip addr add 44.0.0.2/24 broadcast 44.0.0.255 dev dvbrx;
sudo ip link set dvbrx up;
sudo ip link set dvbrx;
```

Command 12: Setup the Receiving IP Tunnel as an IP Interface

As far as the host computer is concerned, this new IP interface is a direct connection to a remote computer (the transmitting Raspberry Pi 3 in this case). The receiving host PC has an IP of 44.0.0.2 on this virtual dvbrx link, so any packets sent to that IP address are accepted by this PC. See Figure 16. For example, if a simple message is sent over UDP on port 1234 using netcat on the transmitting Raspberry Pi as shown below:

```
echo “Test message from the Pi 3” | nc -u -s 44.0.0.1 -p 4000 44.0.0.2 1234
```

Command 13: Send a Test Message

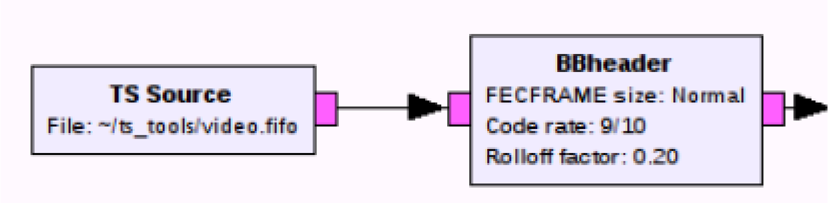
One could receive this message on the receiving host PC by listening on port 1234 using netcat.

```
nc -ul 1234
```

Command 14: Receive a Test Message

Everything regarding the link behind this data transfer (transport stream over DVB-S2) is abstracted away. As far as both the transmitting and receiving PC are concerned, there’s a direct, unidirectional link from the transmitting PC (Pi 3) to the receiving PC (host). However, only UDP packets are supported as there is no return link, which is a requirement of TCP. This setup could be extended to data transfer applications easily, as any application that can transmit solely using UDP would be compatible.

For the software implementation of this transmitter, the GNU radio block diagram shown in Figure 3 was run on the Raspberry Pi 3. A custom block was developed, called transport stream source or 'tssrc' [23], to provide transport stream data to BBheader. This block reads transport stream data in from a specified file, and provides it as an output to the GNU radio blocks. When the file is empty, the block inserts null transport stream packets until there is data available. This block was setup to read video data from the named pipe 'video.fifo' that the video capture/compression/encapsulation command given in Command 4 writes to. The figure below shows the configuration for this block:

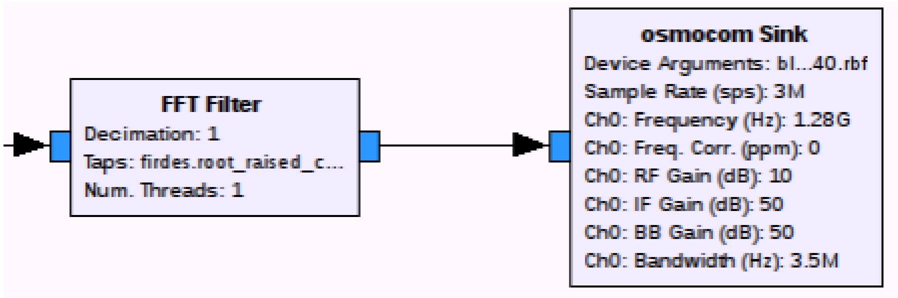


TS Source
File: ~/ts_tools/video.fifo

BBheader
FECFRAME size: Normal
Code rate: 9/10
Rolloff factor: 0.20

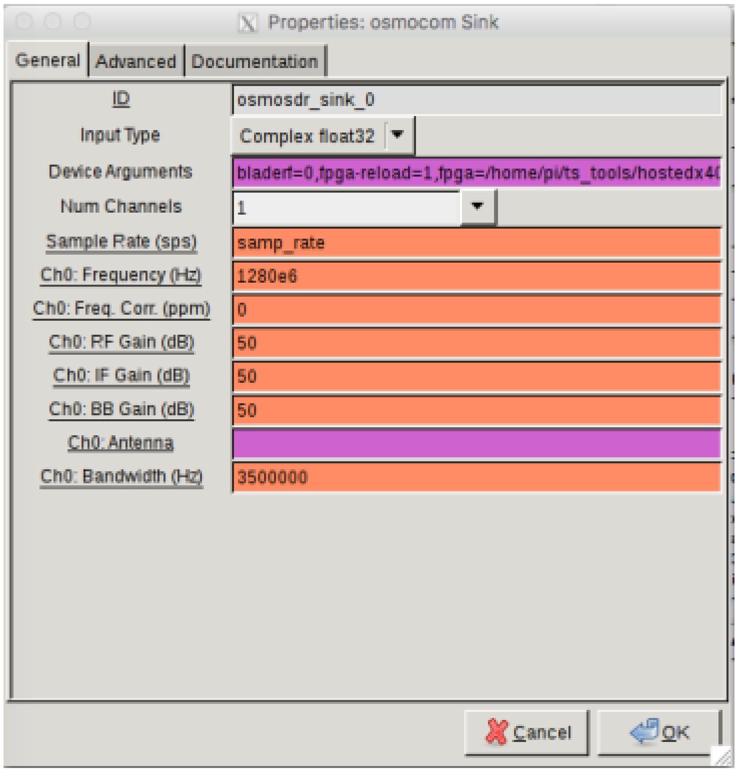
Figure 23: Custom GNU Radio Block 'TS Source' Implementation

Since all the DVB-S2 modulation is done in software, the factory image 'hostedx40-latest' [24] can be used for the FPGA on the bladeRF. The architecture is essentially I/Q symbol pass-through, as shown in Figure 7. In order for GNU Radio to communicate with the bladeRF and transmit these I/Q symbols, the GR-OsmoSDR [25] block is needed. The block contains all the same configuration information that was specified in the bladeRF-cli command (Command 5) used for the hardware implementation. The figures below show the implementation and configuration for this block:


FFT Filter
Decimation: 1
Taps: firdes.root_raised_c...
Num. Threads: 1

osmocom Sink
Device Arguments: bl...40.rbf
Sample Rate (sps): 3M
Ch0: Frequency (Hz): 1.28G
Ch0: Freq. Corr. (ppm): 0
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 50
Ch0: BB Gain (dB): 50
Ch0: Bandwidth (Hz): 3.5M

Figure 24: BladeRF Transmitter Block ‘osmocom Sink’ Implementation



ID	osmosdr_sink_0
Input Type	Complex float32
Device Arguments	bladerf=0,fpga-reload=1,fpga=/home/pi/ts_tools/hostedx4
Num Channels	1
Sample Rate (sps)	samp_rate
Ch0: Frequency (Hz)	1280e6
Ch0: Freq. Corr. (ppm)	0
Ch0: RF Gain (dB)	50
Ch0: IF Gain (dB)	50
Ch0: BB Gain (dB)	50
Ch0: Antenna	
Ch0: Bandwidth (Hz)	3500000

Figure 25: BladeRF Transmitter Block ‘osmocom Sink’ Configuration

With all these pieces in order, running GNU Radio with the bladeRF plugged in via USB creates the video transmission system as was done in hardware. The final GNU radio block diagram is shown in Figure 3. As discussed, a maximum symbol rate of 1.5 MSym/s is possible, which yields an input symbol rate of 5.224 MSymbols/s. To maximize this speed, the Python script that GNU Radio generates and runs during transmission was executed directly. This was done to eliminate having to open the GNU Radio GUI which takes up valuable CPU resources. The Python script had to be modified to remove the GUI portions, as shown in Appendix D – Modifying the GNU Radio Python Script.

In the following section, a comparison between the software (GNU radio) and hardware (bladeRF FPGA) implementations will be done, comparing the power and speed of both options.

5. Test Results and Discussion

Now that both software and hardware implementations have been developed, the two are compared in terms of power and speed. For both setups, the following stay constant:

1. Raspberry Pi 3 used to capture, compress, and encapsulate video from the Raspberry Pi Camera Module V2
2. 16 APSK DVB-S2 modulation
3. 9/10 code rate
4. Normal (16200) frame size
5. 100 tap filter with a roll-off of 0.2
6. Maximum bladeRF transmission power (+6 dBm = 3.98 mW)

In both the hardware and software implementation, the bladeRF SDR was used. However, in the software implementation the onboard FPGA was used as I/Q symbol pass through, and the DVB-S2 modulation was performed in GNU Radio on the Raspberry Pi. In the hardware implementation, the Raspberry Pi 3 was used solely for video capture purposes, and the DVB-S2 modulation was performed on the FPGA. The following table shows the speed capabilities of both systems:

Implementation	Output Symbol Rate (MSym/s)	Input Bit Rate (Mbit/s)
Software	1.5	5.224
Hardware	2.297	8

Table 6: System Data Rates

The hardware implementation can transmit at a bit rate that is 53.13% faster than the software implementation. The following table shows the power draw results of the two systems:

Implementation	bladeRF Power Draw (W)	Raspberry Pi 3 Power Draw (W)	Total Power Draw (W)
Software	3.1	3.198	6.298
Hardware	3.1	1.86	4.96

Table 7: System Power Draws

The hardware implementation uses 21.24% less power than the software implementation. Given the total power draw and the bitrate, a transmit energy value was calculated for each system, expressed in the energy used to transmit one megabyte of data or video. The following equation was developed:

$$Transmit\ Energy = \frac{Power\ Draw}{Maximum\ Input\ Bit\ Rate/8}$$

Equation 4: Transmit Energy versus Power and Bit Rate

This equation was applied to both systems' results:

Implementation	Transmit Energy (J/Mbyte)
Software	9.644
Hardware	4.96

Table 8: Average Transmit Energy Required per Byte Transmitted

Based on these results, the hardware implementation uses 48.57% less power per megabyte transmitted than the software implementation. Table 7 shows that the power draws of both bladeRF systems (software and hardware implementations) were the same. This is most likely because the

DVB-S2 modulation code that runs on the FPGA is running at such a slow rate. Most of the power draw comes from the RF front end and the amplifiers it uses to transmit the signals. Even if the DVB-S2 modulation clock rates were doubled, and the possible bit rate was therefore also doubled to 16 Mbit/s, the power draw for the bladeRF would most likely only marginally change. This would further decrease the transmit energy value for the hardware implementation, as double the bits could be transferred for approximately the same energy. However, this cannot be done for this application, as bandwidth is limited to ~3 MHz by amateur radio guidelines.

6. Conclusion & Future Work

This system-integration project succeeded in developing a video and data transmission system that uses DVB-S2 modulation. Video capture, compression, and encapsulation was done on a Raspberry Pi 3. Video capture was done at 1280x720p resolution and 25 fps with the Raspberry Pi Camera Module V2. H.264 Compression was performed on the Raspberry Pi 3's GPU. FFMPEG was used to encapsulate this H.264 video into a transport stream. This transport stream was multiplexed with data transport stream packets, and was sent over USB to the bladeRF software defined radio (SDR) board. On the bladeRF's Cyclone IV FPGA DVB-S2 modulation was performed, and the I/Q symbols were sent to the LMS6002D front-end for transmission at 1.28 GHz. The system block diagram is show below:

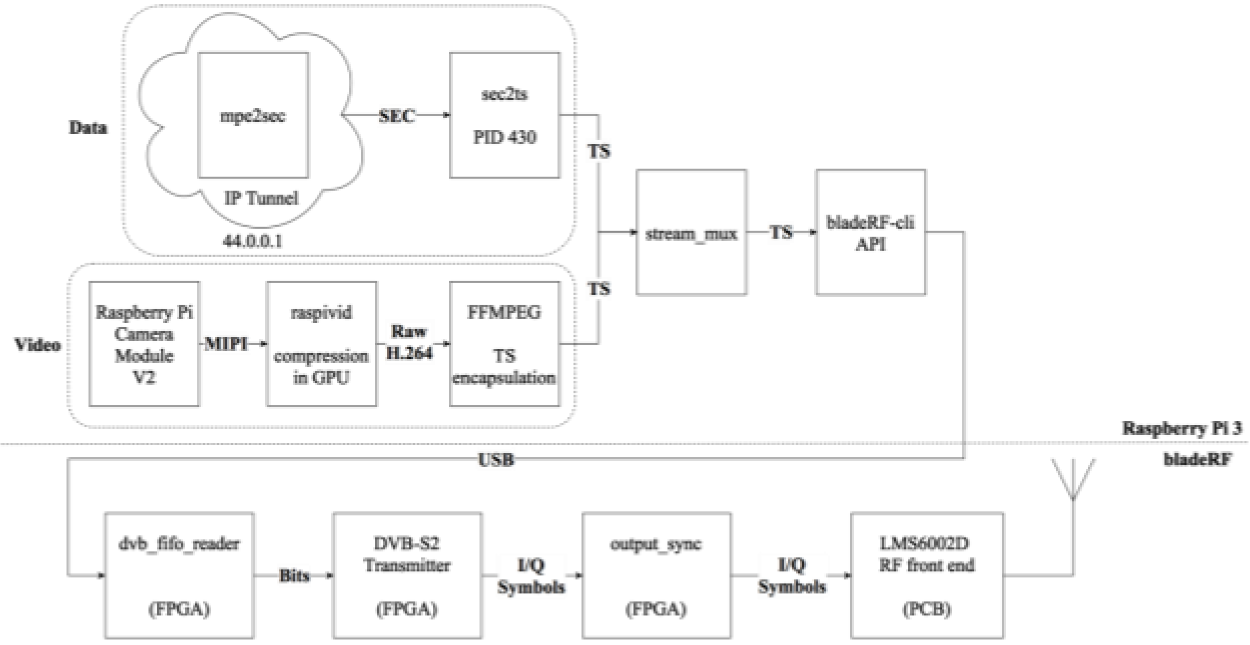


Figure 26: Detailed System Block Diagram

This system was compared to the base software implementation, in which DVB-S2 modulation was done in software on the Raspberry Pi 3 instead of the FPGA. The hardware implementation transmitted 53.13% faster and used 21.24% less power than the software implementation. Overall, the hardware implementation was 48.57% more efficient in transmitting data than the software implementation. The hardware implementation could, most likely, transmit at an even faster rate for the same power usage, but that would push the system outside the 3 MHz recommended transmit bandwidth on the 23cm amateur radio band.

To adapt this system to work in a remote sensing application, (i.e. high altitude balloon) amplifiers and antennas must be added on both the transmit and receive end. Transmission is done at 1.28 GHz which lies in the 23cm band, so the antennas and amplifiers must be optimized for use at this frequency. The transmit power that the transmit amplifier is designed for will be directly related to the required distance for the signal to be receivable at. DVB-S2 has error correction built in, so the required transmit power is relatively less compared to conventional analog signals or digital signals without error correction.

In addition, the current data transfer scheme (shown in Figure 16) must be improved upon and made more robust. As of now, if a piece of data is transmitted that requires more than one transport stream packet, the receive system will not interpret it correctly and the data will be lost. This is a bug that needs to be worked out. In addition, software should be developed to transmit the desired application-specific telemetry/data over UDP as described. This would involve a program on the transmit end running on the Raspberry Pi 3, and a program on the receive end to interpret this data.

Currently, this setup is configured to send 1 video stream encapsulated into a transport stream, which will be recognized by a satellite TV receiver. However, typical satellite TV streams have multiple (i.e. hundreds or thousands) of channels. This setup could be extended to communicate with a second Raspberry Pi (a Raspberry Pi Zero would suffice) to capture a second set of video, and transmit it over the same DVB-S2 link. This would involve configuring the second video stream as a second channel. In doing this, one could capture and transmit different views or angles of the same scene of the remote area that is being observed. On the receiving end, using an off-the-shelf satellite receiver, the user would only have to tune to the second channel to see the alternate view.

This project was successful in developing a system that conformed to the system requirements. Video and data are capable of being transmitted at a total rate of 8Mbit/s over a wireless link. This provides an extremely flexible, capable, and robust platform to transmit data unidirectional from a remote sensing system to a receiving base station.

References

[1] A. D. Steenkamer, “DVB-S2 Transmitter: FPGA Implementation.” RIT, Rochester, NY, 2017.

[2] DATV-Express, “DATV-Express,” 2017. [Online]. Available: https://datv-express.com. [Accessed: 08-Aug-2017].

[3] DVB, “Digital Video Broadcast,” 2017. [Online]. Available: https://www.dvb.org. [Accessed: 08-Aug-2017].

[4] W6HHC, “The Digital Fork in the Road,” *TechTalk*, Orange County, p. 10, May-2009.

[5] DVB, “Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2.” DVB, Geneva, p. 80, 2014.

[6] Raspberry Pi, “Raspberry Pi 3 Model B,” 2017. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Accessed: 08-Aug-2017].

[7] R. Economos, “A DVB-S2 and DVB-S2X transmitter for GNU Radio,” *Github*, 2014. [Online]. Available: https://github.com/drmpeg/gr-dvbs2. [Accessed: 08-Aug-2017].

[8] DVB, “DVB Document A027: DVB specification for data broadcasting.” DVB, Geneva, p. 75, 2016.

[9] DVB, “DVB Fact Sheet: Generic Stream Encapsulation.” DVB, p. 2, 2011.

[10] G. Fairhurst, “A Network-Layer Interface to the Second Generation standard for DVB over Satellite,” Aberdeen, UK.

[11] DVB, “Information technology — Generic coding of moving pictures and associated audio information: Systems.” ISO, Geneva, p. 164, 2000.

[12] Raspberry Pi, “Camera Module - Raspberry Pi Documentation,” 2016. [Online]. Available: https://www.raspberrypi.org/products/camera-module-v2/. [Accessed: 08-Aug-2017].

[13] Nuand, “bladeRF x40,” 2017. [Online]. Available: https://www.nuand.com/blog/product/bladerf-x40/. [Accessed: 08-Aug-2017].

[14] K. Konechy, “DATV - Looking at DVB-S2 Modulation,” *TechTalk*, Orange County, p. 9, Mar-2011.

[15] S. Biereigel, “FPGA Development,” *Nuand Github*, 2017. [Online]. Available: https://github.com/Nuand/bladeRF/wiki/FPGA-Development. [Accessed: 08-Aug-2017].

[16] D. Jones, “Camera Hardware,” 2017. [Online]. Available: https://picamera.readthedocs.io/en/release-1.13/fov.html.

[17] FFmpeg, “FFmpeg Formats Documentation,” 2017. [Online]. Available: https://ffmpeg.org/ffmpeg-formats.html#Authors.

[18] A. Vaught, “Introduction to Named Pipes,” *Linux J.*, 1997.

[19] Avalpa, “OpenCaster 3.2.2: the free digital tv software,” 2017. [Online]. Available: http://www.avalpa.com/the-key-values/15-free-software/33-opencaster. [Accessed: 08-Aug-2017].

[20] ISmart, “iSmart-F1HD,” 2017. [Online]. Available: http://www.ismartt.com/f1hd.html. [Accessed: 01-Mar-2017].

[21] Novra, “S300V Satellite Data and Video Receiver,” 2017. [Online]. Available: http://novra.com/products-page/ip-satellite-data-receiver/s300v/. [Accessed: 08-Aug-2017].

[22] Tutorials Point, “nc - Unix, Linux Command,” 2017. [Online]. Available: http://www.tutorialspoint.com/unix_commands/nc.htm. [Accessed: 08-Aug-2017].

[23] M. Zachary, “Transport Stream Source,” *Github*, 2017. [Online]. Available: https://github.com/mattzgto/gr-tssrc. [Accessed: 08-Aug-2017].

[24] Nuand, “bladeRF FPGA images,” 2017. [Online]. Available: http://novra.com/products-page/ip-satellite-data-receiver/s300v/. [Accessed: 08-Aug-2017].

[25] D. Stolnikov, “osmocom Gnu Radio Blocks,” *osmocom*, 2013. [Online]. Available: https://osmocom.org/projects/sdr/wiki/GrOsmoSDR. [Accessed: 08-Aug-2017].

Appendix A – User Guide

A. Raspberry Pi 3 setup:

1. Install Raspbian 8.0 to a Raspberry Pi 3.
    a. The following guide from Raspberry Pi is helpful in doing so: https://www.raspberrypi.org/documentation/installation/installing-images/

    b. The Raspbian downloads can be found here: https://www.raspberrypi.org/downloads/raspbian/

       The version 'With Desktop' is recommended. This will allow easy setup using a mouse, keyboard, and monitor

    c. The default username is 'pi' and the default password is 'raspberry'

2. Once Raspbian has been installed, connect a mouse, keyboard, and monitor to the Pi, insert the SD card, and provide a power source.

3. Once Raspbian is booted, run Raspberry Pi configuration (raspi-config).
    a. Enable SSH

    b. Enable the camera

    c. Change the password from the default to prevent unauthorized access.

4. Connect to a wireless network with internet access. If you will be operating remotely, connect to the same network as your remote computer. Also, if you will be operating remotely, you will need the IP address of the Pi. This can be found by executing 'ifconfig', in the 'inet addr' field or 'wlan'.

5. The Pi is now completely configured for either local (with mouse and keyboard) or remote (via SSH on a remote computer) operation. The following instructions are applicable to both.

6. Copy the project files to the user directory of the Pi (/home/pi or ~/), either using a USB flash drive or SCP.

B. Install required packages:

```
sudo apt-get install cmake
sudo apt-get install libusb-1.0
sudo apt-get install libboost1.50-all
sudo apt-get install libcppunit-dev
sudo apt-get install Doxygen
sudo apt-get install liblog4cpp5-dev
sudo apt-get install python-qt4
```

C. Install the bladeRF API:

```
sudo apt-get install libbladerf-dev
```

```
git clone https://github.com/Nuand/bladeRF.git
cd bladeRF/host
mkdir -p build
cd build
cmake -DENABLE_BACKEND_LIBUSB=ON ../
make
sudo make install
sudo ld config
```

D. Install video/data tools:

1. FFmpeg is provided pre-compiled for the Raspberry Pi 3. Install it as follows:

```
cd FFmpeg/build
sudo make install
sudo ldconfig
```

If this doesn’t work, it will have to be compiled and installed manually

2. Install the opencaster suite of tools:

```
cd opencaster
autoreconf -fi
./configure
make
sudo make install
```

3. Compile the stream multiplexer and place it in the root directory:

```
cd ts_mux
gcc stream_mux.c –o stream_mux
mv stream_mux ../
```

4. Place both FPGA images in the root directory

```
mv Software\ Implementation/hostedx40.rbf ~
mv Hardware\ Implementation/withdvb.rbf ~
```

E. Software Implementation:

1. Install GNU Radio using the following command:

```
sudo apt-get install gnuradio --fix-missing
sudo apt-get install gnuradio-dev
```

2. Install the GR-OsmoSDR GNU Radio block, which provides access to the bladeRF in GNU Radio:

```
sudo apt-get install gr-osmosdr
```

3. Install the Transport Stream source GNU Radio block

```
cd gr-tssrc
mkdir build
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

4. Install the DVB-S2 blocks

```
git clone https://github.com/drmpeg/gr-dvbs2

cd gr-dvbs2
mkdir build
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

5. Copy the blocks to the GNU Radio directory

```
sudo cp /usr/local/share/gnuradio/grc/blocks/dvbs2_*
/usr/share/gnuradio/grc/blocks/

sudo cp /usr/local/share/gnuradio/grc/blocks/tssrc_tssrc_bb.xml
/usr/share/gnuradio/grc/blocks/

sudo cp -r  /usr/local/lib/python2.7/dist-packages/dvbs2/
/usr/local/lib/python2.7/dist-packages/dvbs2_swig/
```

6. Open GNU Radio

```
sudo su
export XAUTHORITY=/home/pi/.Xauthority
sudo gnuradio-companion
```

7. Within GNU Radio, open the 'dvbs2_tx.grc' block diagram file from the 'Software Implementation folder'.

8. Plug the bladeRF into the Pi via USB.

9. Make a named pipe for the streams

```
mkfifo ~/video.fifo
mkfifo ~/data.fifo
mkfifo ~/combined.fifo
```

10. Within GNU Radio, execute the flow, and make sure the setup is working. Setup the satellite receiver to receive at a frequency of 11.030 GHz (pre-LNB) or 1.28 GHz (post-LNB), with a symbol rate of 2.6 MSymbols/s.

11. A python script 'dvbs2_tx.py' has been generated, which when run will start the DVB-S2 transmitter. Modify it per Appendix D – Modifying the GNU Radio Python Script.

12. Execute the Python script without the GUI by executing:

```
sudo python dvbs2_tx.py
```

13. Provide video and data to the transmitter by executing Command 4, Command 6, and Command 8.

F. Hardware Implementation:

1. No additional setup is required. Provide video and data to the transmitter by executing the series of commands shown in Command 9.

Appendix B – File System

- "bladeRF-dvbs2" – Contains the IP (Verilog and VHDL) and the Quartus project for the DVB-S2 modulator and the bladeRF architecture
    - "fpga"
        - "ip" – Contains the IP, separated by developer
        - "platforms"
    - "quartus" – Contains the Quartus project and generated output files
        - "signaltap"
        - "work" – Contains the Quartus project file ("bladerf.qpf")
            - "greybox_tmp"
            - "output_files" – Contains the generated output files, generated upon compilation (including the FPGA programming file, "withdvb.rbf")

  Alternatively, pull from https://github.com/mattzgto/bladerf-dvbs2

- "FFmpeg" – The fully compiled FFmpeg source code for the Raspberry Pi 3

- "gr-tssrc" – Custom GNU Radio block – Transport Stream Source - source code

  Alternatively, pull from https://github.com/mattzgto/gr-tssrc

- "Hardware Implementation" – Contains all the files required for the hardware implementation (in this case, just the FPGA programming file "withdvb.rbf")

- "opencaster" – Fork of the open source software OpenCaster
    - "extras"
    - "libs" – Contains source code for dependencies
    - "tools" – Contains source code for all the utilities
        - mpe2sec – Contains source code for the mpe2sec program
        - sec2mpe – Contains source code for the sec2mpe program
        - sec2ts – Contains source code for the sec2ts program
        - ts2sec – Contains source code for the ts2sec program

  Alternatively, pull from https://github.com/mattzgto/opencaster

- “Paper Images” – Contains the draw.io diagram files and images used in this paper

- “Software Implementation” – Contains all the files required for the software implementation
    - “dvbs2_tx.grc” – GNU Radio block diagram file
    - “dvbs2_tx.py” – Python script generated by GNU Radio to execute the DVB-S2 transmitter
    - “hostedx40.rbf” – bladeRF I/Q symbol pass-through FPGA image

- “ts_mux” – Custom transport stream multiplexer (“ts_mux.c”)

  Alternatively, pull from https://github.com/mattzgto/ts_mux

Appendix C – Software Used

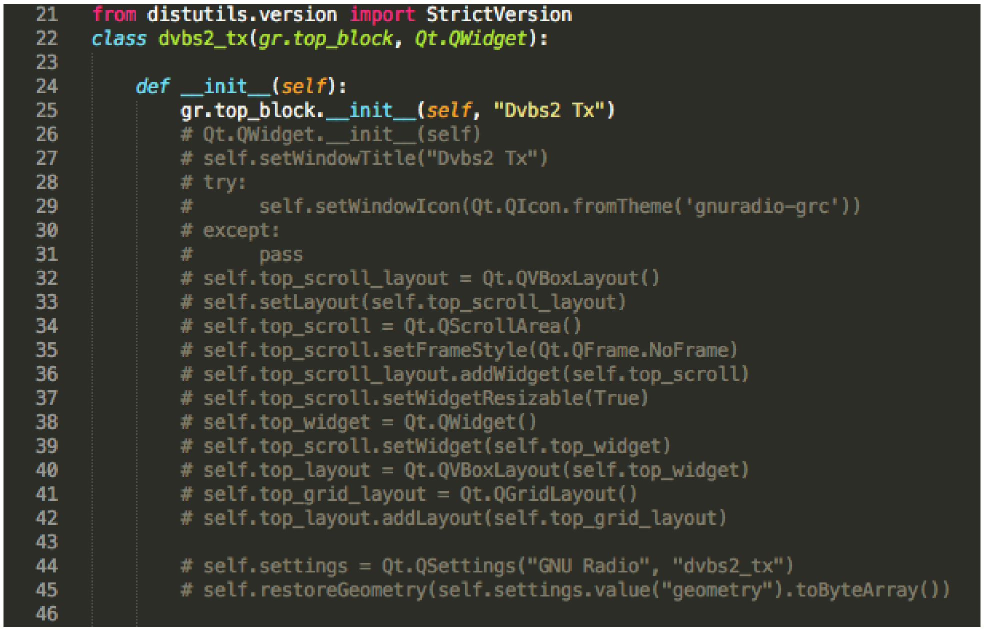
This appendix lists the software used for the development of this project.

- Quartus Prime version 17.0 running on Ubuntu 14.04 (for compiling the bladeRF and DVB-S2 projects)
- ModelSim-Altera version 17.0 running on Ubuntu 14.04 (for simulating the DVB-S2 project)
- http://draw.io (for generating diagrams used in this paper)
- Rasbian Jessie version 8.0 operating system installed on Raspberry Pi 3
- GNU Radio version 3.7.5 running on Raspberry Pi 3
- GR-OsmoSDR version 0.1.3-2 installed to the Raspberry Pi 3 using apt-get (used to allow GNU Radio to transmit via the bladeRF)
- libbladeRF-dev version 0.2014.09~rc2-5 installed on the Raspberry Pi 3 using apt-get (includes the bladeRF-cli command line utility)
- FFMPEG version number N-85767-gdec2fa8 installed to the Raspberry Pi 3

Appendix D – Modifying the GNU Radio Python Script

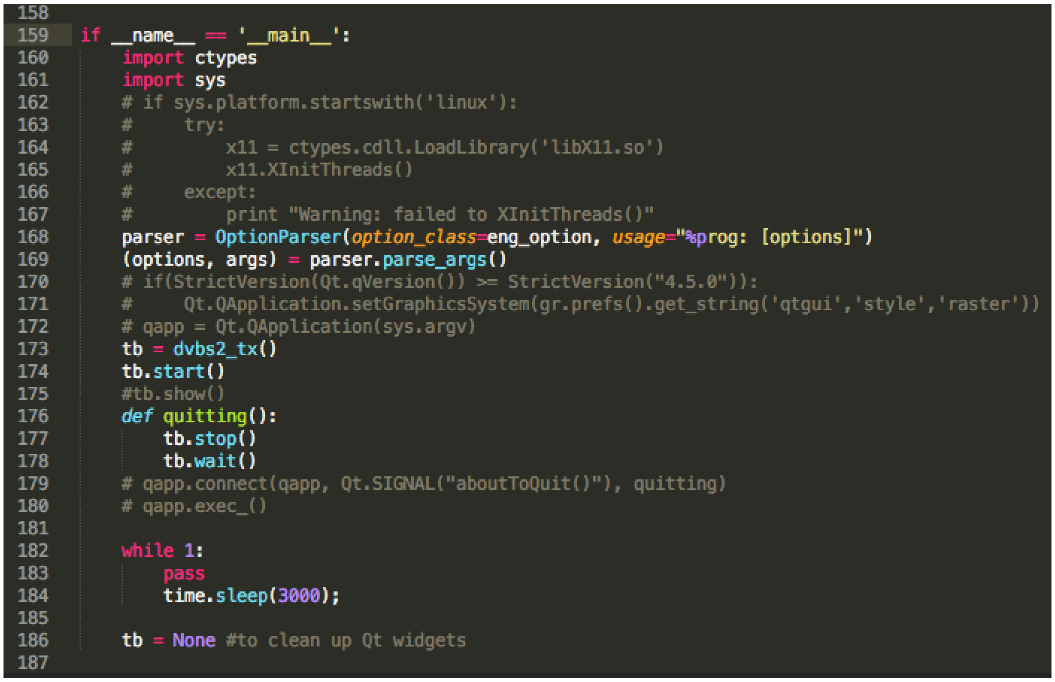
The following two screenshots show how the Python script generated by GNU Radio was modified to prevent the GUI from opening. By commenting out the lines as shown, and adding the while loop, the Python script can run directly without having to start the GNU Radio GUI. Changes to the block can be made to program in the Python script, although it is more difficult than using the GUI. If the Python is regenerated by the GUI, the changes shown below will have to be redone. To run the python script, simply execute:

```
sudo python dvbs2_tx.py
```



```
21  from distutils.version import StrictVersion
22  class dvbs2_tx(gr.top_block, Qt.QWidget):
23
24      def __init__(self):
25          gr.top_block.__init__(self, "Dvbs2 Tx")
26          # Qt.QWidget.__init__(self)
27          # self.setWindowTitle("Dvbs2 Tx")
28          # try:
29          #     self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
30          # except:
31          #     pass
32          # self.top_scroll_layout = Qt.QVBoxLayout()
33          # self.setLayout(self.top_scroll_layout)
34          # self.top_scroll = Qt.QScrollArea()
35          # self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
36          # self.top_scroll_layout.addWidget(self.top_scroll)
37          # self.top_scroll.setWidgetResizable(True)
38          # self.top_widget = Qt.QWidget()
39          # self.top_scroll.setWidget(self.top_widget)
40          # self.top_layout = Qt.QVBoxLayout(self.top_widget)
41          # self.top_grid_layout = Qt.QGridLayout()
42          # self.top_layout.addLayout(self.top_grid_layout)
43
44          # self.settings = Qt.QSettings("GNU Radio", "dvbs2_tx")
45          # self.restoreGeometry(self.settings.value("geometry").toByteArray())
46
```

Lines 26-45 commented out



```
158
159 if __name__ == '__main__':
160     import ctypes
161     import sys
162     # if sys.platform.startswith('linux'):
163     #     try:
164     #         x11 = ctypes.cdll.LoadLibrary('libX11.so')
165     #         x11.XInitThreads()
166     #     except:
167     #         print "Warning: failed to XInitThreads()"
168     parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
169     (options, args) = parser.parse_args()
170     # if(StrictVersion(Qt.qVersion()) >= StrictVersion("4.5.0")):
171     #     Qt.QApplication.setGraphicsSystem(gr.prefs().get_string('qtgui','style','raster'))
172     # qapp = Qt.QApplication(sys.argv)
173     tb = dvbs2_tx()
174     tb.start()
175     #tb.show()
176     def quitting():
177         tb.stop()
178         tb.wait()
179     # qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
180     # qapp.exec_()
181
182     while 1:
183         pass
184         time.sleep(3000);
185
186     tb = None #to clean up Qt widgets
187
```

Lines 162-167, 170-172, 179-180 commented out
‘while’ loop added to prevent the script from exiting