Project Lab III

Final Report

Alexandre Soares da Silva

ECE 3334 - Project Lab III, 2018 Summer

Texas Tech University

QAM_land

Alex Soares, Nikolik Zachary, Derek Tagert

Abstract

This paper describes what QAM_land accomplished throughout the 2018 summer while attempting to build a transmitter/receiver RF system that uses Quadrature Amplitude Modulation to transmit a signal over 100 m of distance. The parts of the system that work and the challenges faced by the team are described, with a focus on programming the system.

Contents

List of Figures

List of Tables

1. Introduction

Quadrature Amplitude Modulation maps bits to two carrier waves in superposition, separated by an offset of 90°, with their phases and amplitudes modulated. Because of the modulations and two carrier waves together, it is possible to represent more than one bit with one QAM symbol.

This report describes QAM_land's efforts to implement the system described in the Abstract section. It focuses on the programming on both the transmitter and receiver sides. QAM_land's system encodes the preamble of the U.S. Constitution in 16-QAM symbol and mixes it with a 28.8 MHz carrier signal. The receiver side, decoding the preamble, has not been implemented fully due to unforeseen difficulties on the decoding.

2. TX-RX QAM-system design

2.1. Block Diagram

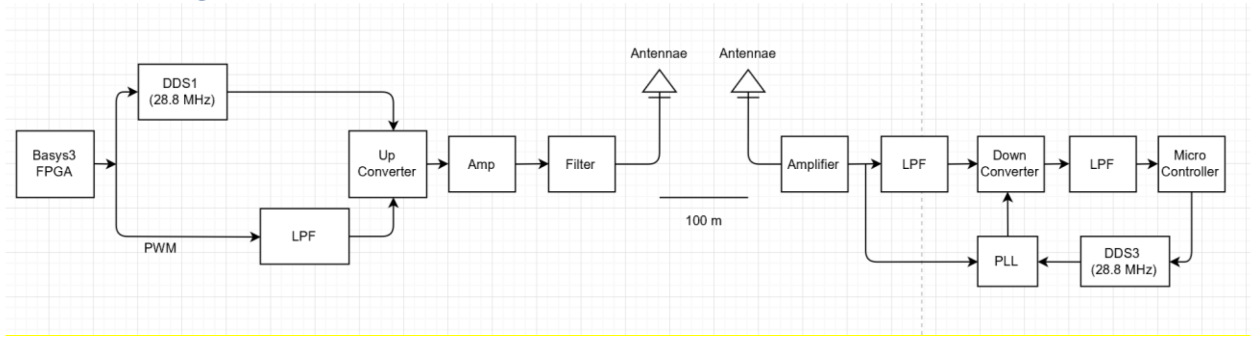


Figure 1: System Block Diagram

2.2. System components

TRANSMITTER	
3	High Signal Level Active Mixer
1	Basys3 FPGA
1	Single to Differential Amp
1	Differential to Single Amp
1	Sa612 Mixer
RECEIVER	
3	High Signal Level Active Downconverter
2	Phase Locked Loop
1	Ti C2000 F28379d
BOTH	
2	Nooelec AD9850 Signal Generator
3	RF Npn Transistor
3	Uxcell Telescopic Radio Antenna
2	Ti Msp430g2553
2	Hiletgo 120pcs/3x40pcs Jumper Wires

Table 1: parts used on each end (receiver/transmitter) and their quantities (leftmost column)

Transmitter

To encode the 16-QAM symbols, the transmitter side uses the Digilent Basys3 FPGA, which generates PWM signal representing the QAM carrier wave (1 kHz signal). A passive RC low pass filter smoothens the PWM signal from FPGA with cutoff frequency of approximately 1 kHz. The RC LP filter uses a 10 k Ω resistor in series with the signal (just after it) and a 15nF capacitor in parallel with the filtered signal's output. The smoothed signal is read from after the 10 k Ω resistor.

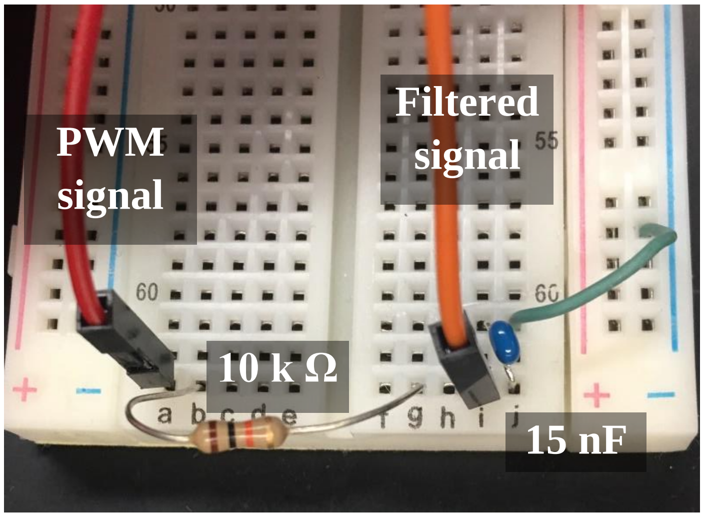


Figure 2: first order passive LP filter used to smoothen the PWM signal from the Basys3 FPGA

The high frequency (28.8 MHz) carrier signal is generated by an Analog Devices AD9850 Direct Digital Synthesis IC. The AD9850 is programmed and powered by a TI MSP430G2553.

Both the signals are mixed with an NXP Sa612a mixer, with the DDS used as the external oscillator and the RF input is the filtered 16-QAM, 1kHz signal. The mixed carrier signal would be then amplified with the amplifier designed by Zach:

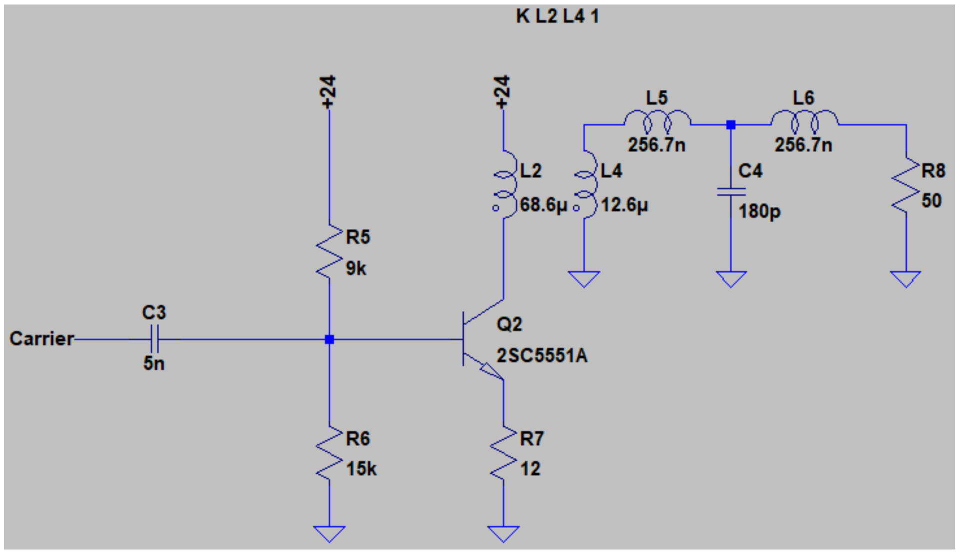


Figure 3: transmitter signal amplifier, designed by Zach Nikolic

The amplified signal would then be fed to a Uxcell telescopic radio antenna for transmission.

Receiver

The high frequency carrier would then be received by another Uxcell telescopic radio antenna on the receiver’s end. A phase-locked loop would synchronize the received signal with a 28.8 MHz sine wave produced by an AD9850 DDS, run by a second MPS430G2553, facilitating the downcoversion of the signal to the expected 16-QAM 1 kHz. The signal would be sampled and decoded by the TI C2000 F28379D microcontroller.

2.3. Timing as a crucial component

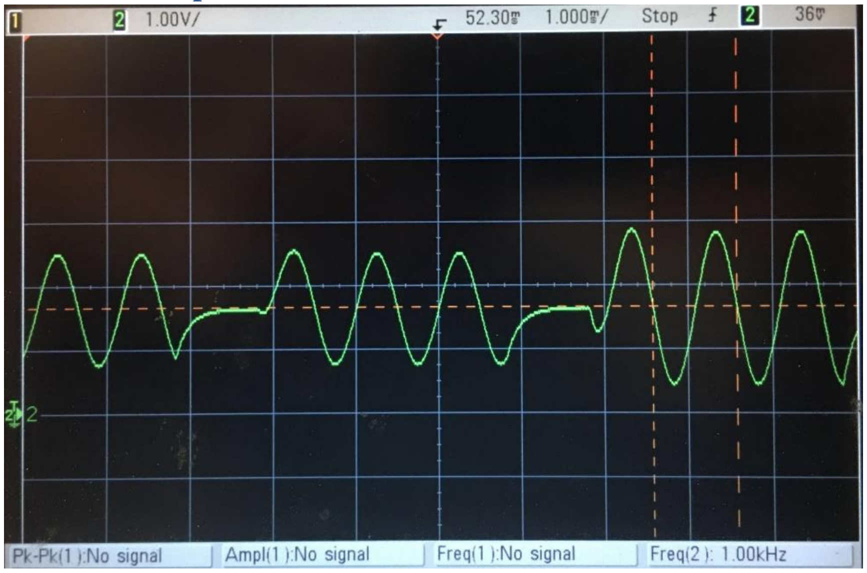


Figure 4: Oscilloscope reading from the filtered 16-QAM signal produced by the FPGA. Notice the start of each symbol marked by a sinusoid zero-level and the frequency, 1 kHz

Generating the QAM sine wave with the Digilent Basys3 FPGA requires calculating many different constants. There should be enough clock cycles for a PWM with an appropriate resolution to differentiate modulated amplitudes on the receiver side, and enough samples in each sine wave period for the detection of the phase shifts.

Because of previous difficulties related to generating a consistent PWM signal at a frequency beyond 1 MHz –affected by RC constant effects within the FPGA circuit – the Basys3, previously run at 400 MHz, now runs at 100 MHz. To generate the 1 kHz carrier signal (also different from the previous 25 kHz), 800 clock cycles are used for one PWM period; hence a PWM frequency of 125 kHz (100 MHz / 800 clock cycles).

If each PWM period has 800 clock cycles available and a carrier signal period lasts 1 ms, then each QAM sine wave period is made up of 125 samples, which creates a smooth sine wave for this project's purposes and for differentiating phase shifts. Four periods of the QAM signal have been chosen to carry one QAM symbol, yielding a symbol rate of 250 symbol/s or a 1 kbps bitrate with 16-QAM. Testing the system by transmitting the preamble of the U.S. Constitution with a 64 bit start signal takes (336 chars * 8 bits /char) / 1 kbps = 2.688 seconds of transmission for the full preamble.

On the receiver's side, the 1 kHz signal is recovered from 16 samples per period. The sampling theorem states that the sampling frequency should be at least equal to twice the frequency sampled. To recover the sine wave on the receiver side, that means at 2 k samples per second; in practice, this sampling rate should be at least 4 – 8 times the original frequency for more accurate phase recovery. Thus, 16 kSps is an appropriate rate for recovering the carrier signal with accuracy on the receiver side, using the TI F28379D with a timer interrupt that runs every 62.5 µs.

As demonstrated, a tight control over timing is necessary for both generating the QAM signal and for decoding it on the receiver's side.

2.4. Implementation of 16-QAM

To calculate the bit rate per symbol of a Quadrature Amplitude Modulation M-QAM, being M the constellation used, we feed M to $\log_2(M)$. In the case of 16 -QAM, that answer is 4, which means every 16-QAM symbols carries 4 bits:

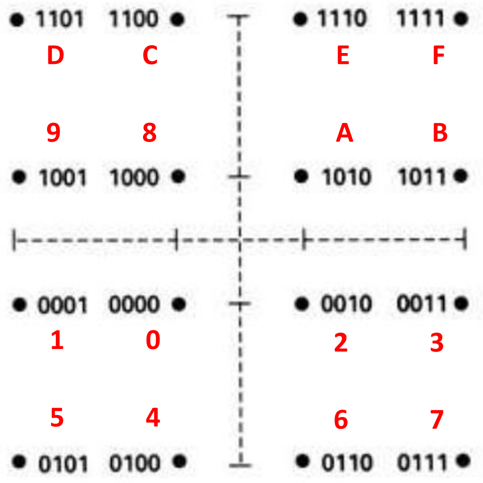


Figure 5: 16-QAM constellation and the corresponding 4 bits / hexadecimal numbers

4-bit encoded value	Amplitude A	Phase θ	4-bit encoded value	Amplitude A	Phase θ
0	0.64	-135	8	0.64	135
1	1.14	-161.57	9	1.14	161.57
2	0.64	-45	A	0.64	45
3	1.14	-18.43	B	1.14	18.43
4	1.14	-112.5	C	1.14	112.5
5	1.5	-135	D	1.5	135
6	1.14	-71.57	E	1.14	71.57
7	1.5	-45	F	1.5	45

Table 2: Phases and amplitudes used to modulate the QAM sine wave $x(t) = A\,sin(2\pi f_c t + \theta)$

In the QAM_land's project, 4 periods of the 1 kHz sine wave carry one 16-QAM symbol. The first period is on a base voltage (the sinusoid's zero), followed by 3 phase-shifted and amplitude modulated periods. The Verilog simulation below shows the PWM duty cycles outputted as an analog wave:

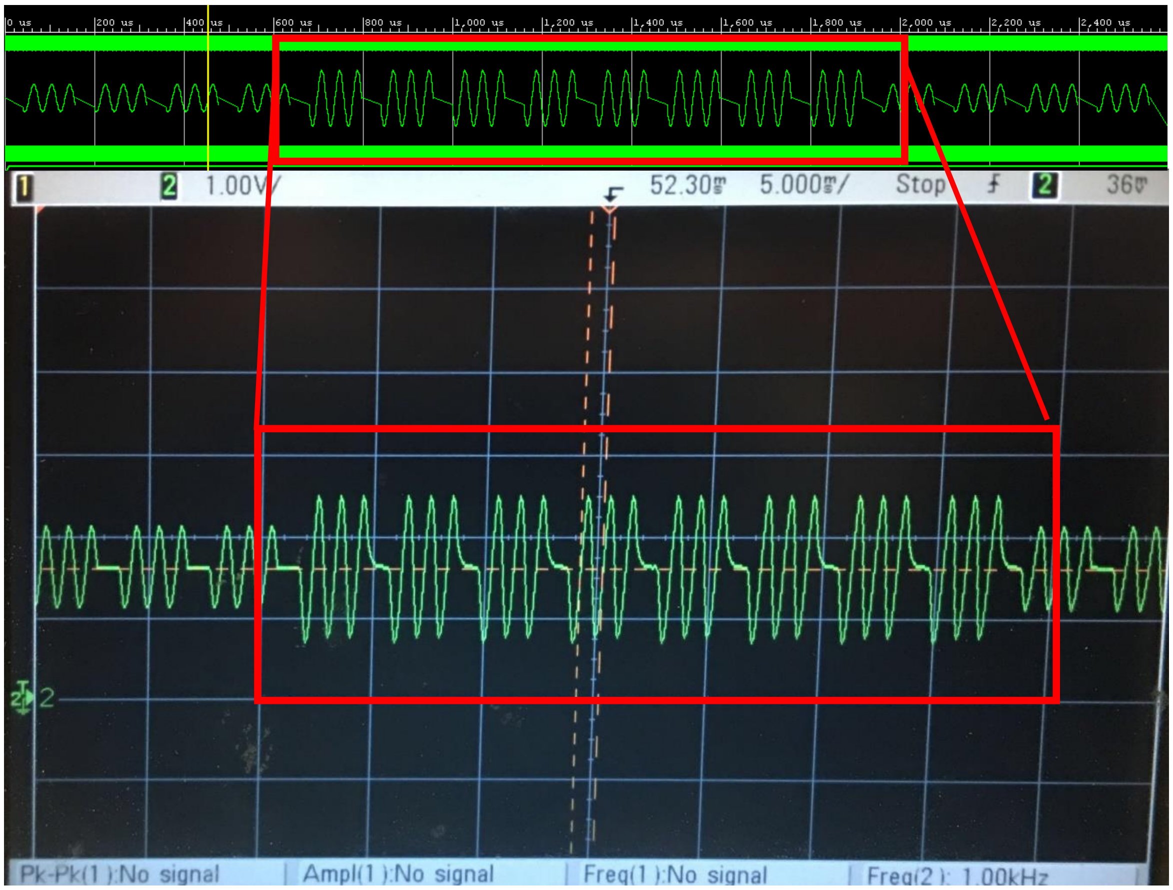


Figure 6: The start signal, both seen on an oscilloscope (PWM signal through passive LP filter) on the bottom picture, and above it, on a simulation

Previously, there was an issue with the beginning of each QAM symbol not having the sine wave zero-level; instead, all periods were fulfilled by the PWM sine wave. This was fixed in the Verilog code.

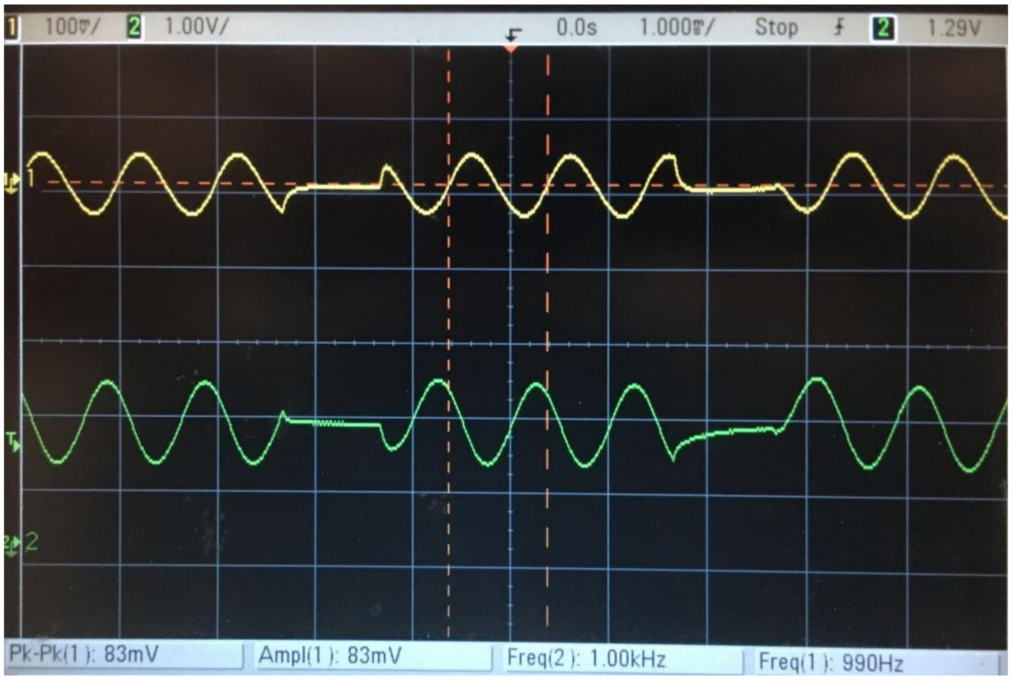


Figure 7: 28.8 MHz and 16-QAM mixed signal on the bottom, filtered FPGA 16-QAM signal on the top.

When the 28.8 MHZ carrier and the 16-QAM signal are mixed, there is a 180° phase shift caused by the capacitors in the mixer’s circuit. Moreover, the 28.8 MHz carrier wave needs a larger amplitude to become and effective carrier, as it can be seen on the oscilloscope readings above (the high frequency carrier can hardly be seen). The fact that the DDS signal might have needed amplification was not anticipated by QAM_land’s team.

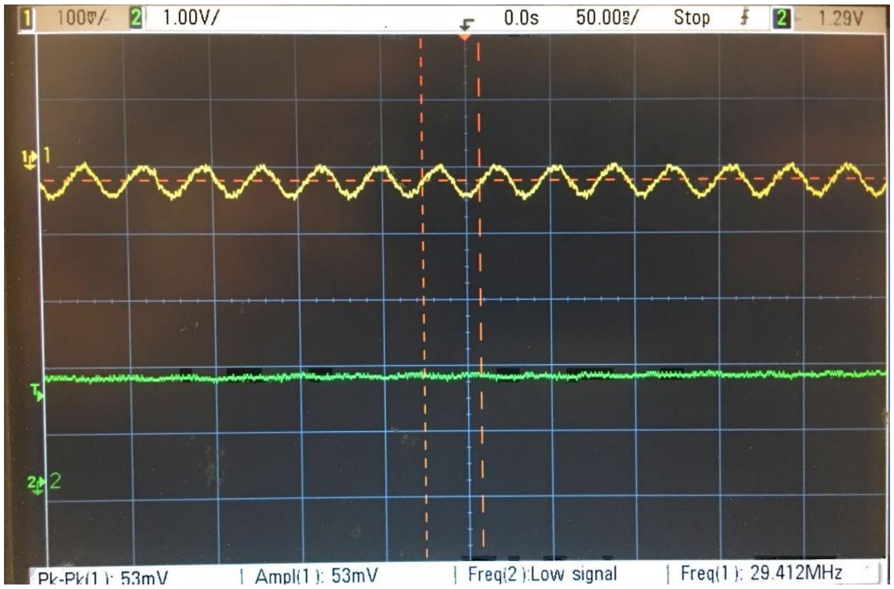


Figure 8: 28.8 MHz and 16-QAM mixed signal on the top, filtered FPGA 16-QAM signal on the bottom (50 ns window)

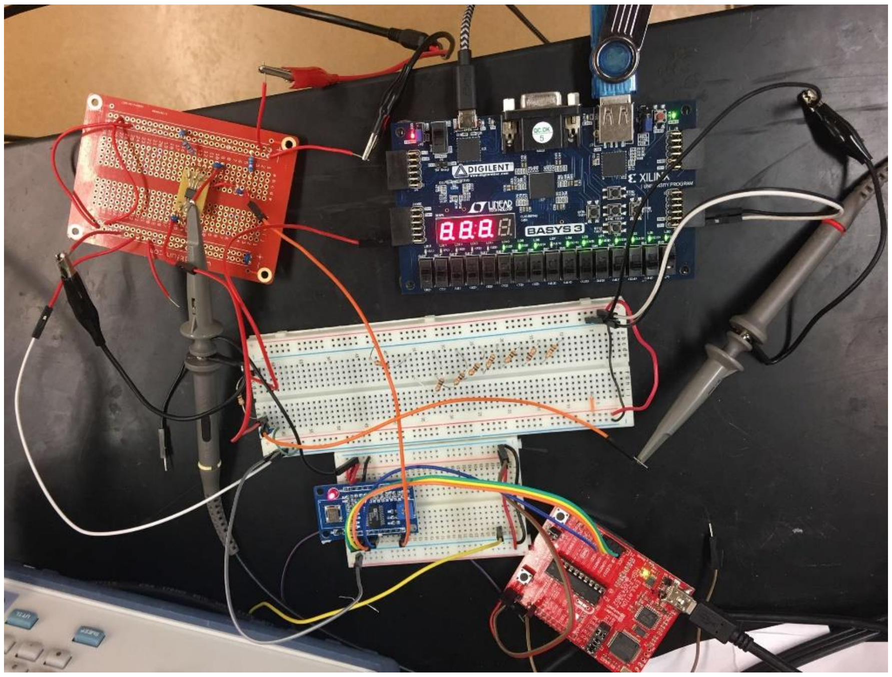


Figure 9: clockwise direction – mixer on the top left, followed by Basys3 FPGA, followed by MSP43G2553 microcontroller driving the AD9850 DDS (mounted on the small breadboard)

On the receiver side, the algorithm for demodulating the 16-QAM signal follows straightforward steps:

1. Detect a symbol;
2. Sample the symbol for 3 periods;
3. Use the last 2 periods to prevent artifacts from steep voltage shifts, present in the first period (where the sine wave starts after the zero-cross voltage);
4. Apply dot products between the retrieved signal and normalized sine and cosine arrays, to get the signal's projection onto the x and y axes:
    a. x = dot (cos (2*pi*2T), signal sampled);
    b. y = dot (sin(2*pi*2T), signal sampled);
5. Apply thresholds to recover the hex value corresponding to the 16-QAM symbol;
6. Get two hex values in sequence to recover the 4 high and 4 low bits of an ASCII value;
7. Use the logic above to detect the sequence of symbols that make up the start signal of the transmission and decode the message.

Detecting the flat region that precedes the amplitude modulated/phase carrier signal is trivial. To detect the start of a symbol (sine wave zero voltage), one can look for values between a window and, if they are recorded for 6 times in a row and are consecutive, the start of a signal is detected. To make the algorithm more accurate, differences between consecutive samples are taken into consideration.

Removing that "flat" region from the samples and only storing useful carrier values is, however, not trivial. The values corresponding to this one symbol's start signal vary more than expected, and sometimes look very much like the beginning of a small amplitude sine wave:

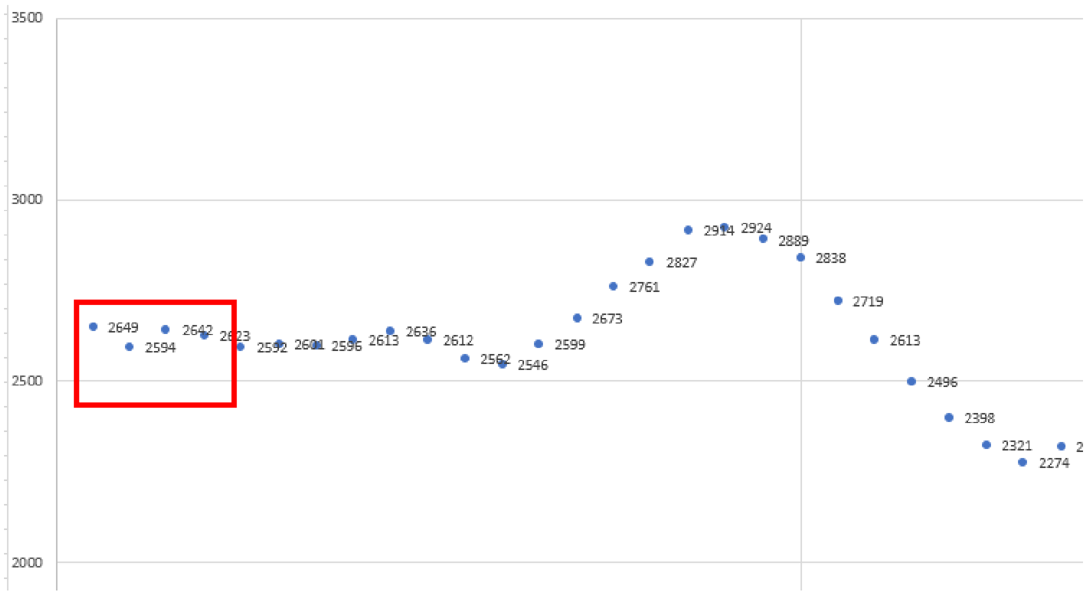


Figure 10: the beginning of a 16-QAM symbol, with a start signal followed by the carrier sine wave. Notice the roughness of the "flat" region and how some of the consecutive values have larger differences between them than when the actual phase-shifted sine wave starts

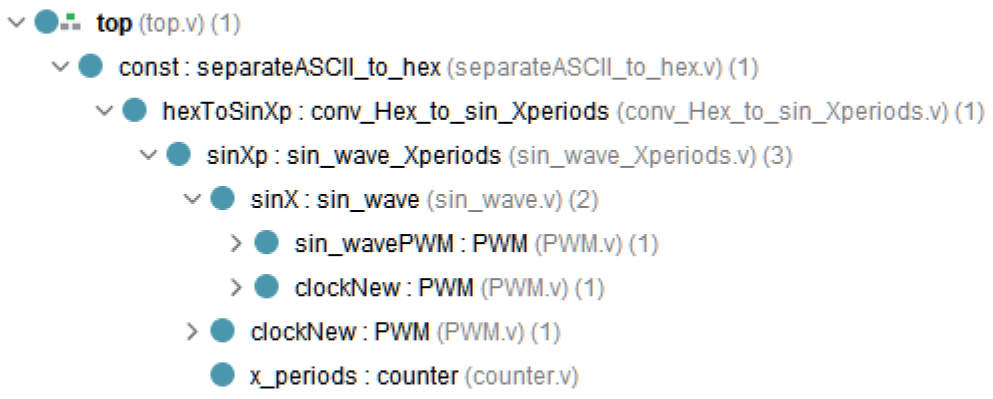
Calculating difference between consecutive values, which was aimed at preventing false positives – e.g. slicing before the flat region was over – does not help that much, either. Some changes are larger in the flat region than they are at the beginning of the smallest amplitude-modulated signal, as shown in the picture above.

The QAM_land team anticipated that decoding the signal would be the most challenging part of this project. Unforeseen, however, were the irregularities within the start of a symbol sampled by the ADC – the flat region was created to make symbol detection easier, not more difficult. The rest of the algorithm would have been somewhat simple to implement; but without the reliable slicing of useful parts of the symbol, the demodulation remains not functional.

3. Code

3.1. Transmitter

All the Verilog modules are driven by a 100 MHz clock:



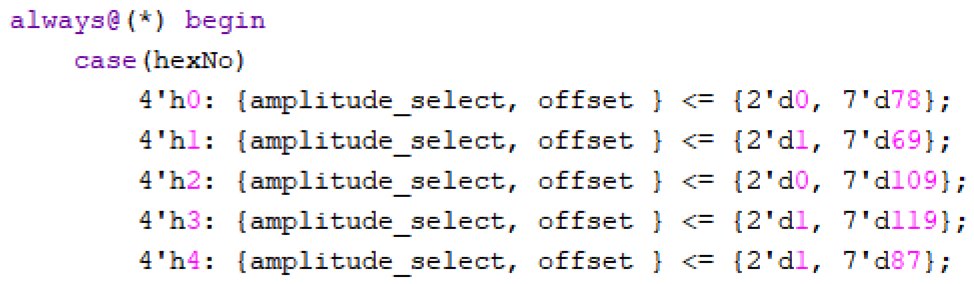
```
top (top.v) (1)
  const : separateASCII_to_hex (separateASCII_to_hex.v) (1)
    hexToSinXp : conv_Hex_to_sin_Xperiods (conv_Hex_to_sin_Xperiods.v) (1)
      sinXp : sin_wave_Xperiods (sin_wave_Xperiods.v) (3)
        sinX : sin_wave (sin_wave.v) (2)
          sin_wavePWM : PWM (PWM.v) (1)
          clockNew : PWM (PWM.v) (1)
        clockNew : PWM (PWM.v) (1)
        x_periods : counter (counter.v)
```

Figure 11: Verilog modules that create the 16-QAM PWM carrier

These are the Verilog modules used, from top to bottom:

- **top.v:** outputs the signals to the Basys3 ports.
- **separateASCII_to_hex:** stores the preamble of the U.S. constitution with 64'h0000_1111_1111_0000 as start signal within a 2688 bit-wide register. Indexing in Verilog required the preamble to be written backwards from last to first letter within this module.
  Every time the module separateASCII_to_hex counts twice as the amount of ASCII characters in the preamble, it restarts encoding and sending the text, as each ASCII is 8 bits wide and each 16-QAM symbol, 4 bits wide.

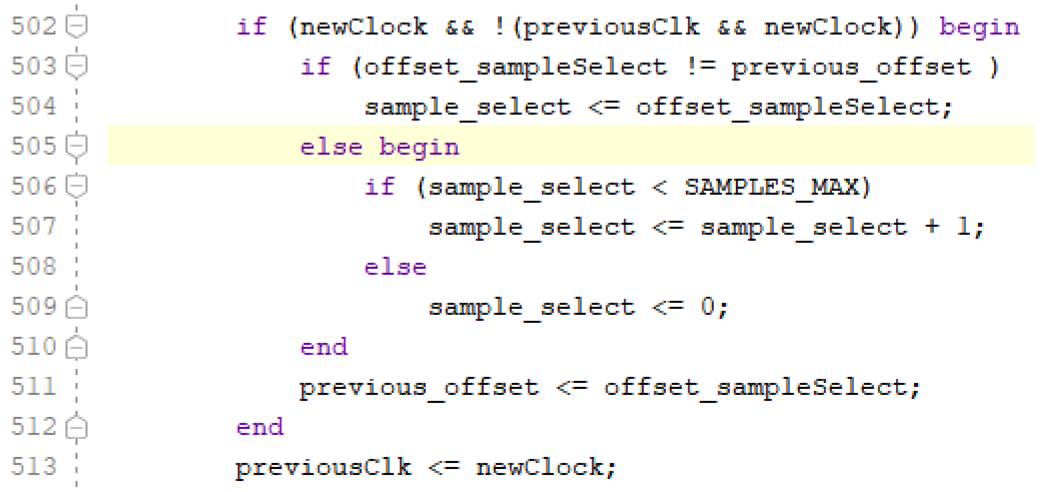
- **conv_Hex_to_sin_Xperiods:** this module converts hexadecimal values (4 bit-wide numbers) to the appropriate 16-QAM symbol. It assigns an amplitude and the offset to a lower module that generates the 4 periods of the sine wave that correspond to one 16-QAM symbol. The offsets select a starting PWM duty cycle out of 125 available within one amplitude case in the lower sin_wave module



```
always@(*) begin
    case(hexNo)
        4'h0: {amplitude_select, offset } <= {2'd0, 7'd78};
        4'h1: {amplitude_select, offset } <= {2'd1, 7'd69};
        4'h2: {amplitude_select, offset } <= {2'd0, 7'd109};
        4'h3: {amplitude_select, offset } <= {2'd1, 7'd119};
        4'h4: {amplitude_select, offset } <= {2'd1, 7'd87};
```

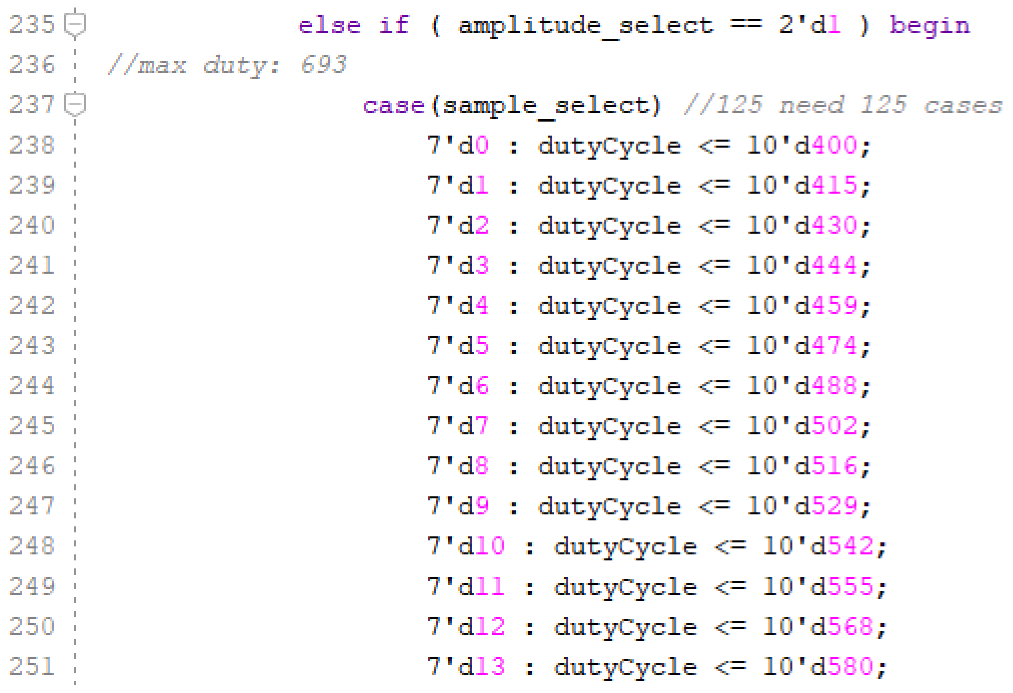
Figure 12: Examples of cases relating 16-QAM sine wave amplitudes and offsets to the represented hexadecimal symbol.

- **sin_wave_Xperiods:** instantiated within conv_Hex_to_sin_Xperiods, the sin_wave_Xperiods module executes the lower module sin_wave for a parametrized amount of times during a parametrized period. The amount of execution times is programmed through a left shift parameter. The first sine wave period produced, however, is always a flat line (base voltage) when observed in a test bench simulation.
- **sin_wave:** holds 3 look up tables, corresponding to the duty cycles (within 800 clock cycles) needed to produce the 3 different amplitudes present in the 16-QAM symbols. Each case statement has 125 steps for the sine wave generation, or 64 PWM samples/ sine wave period.
  Within the sin_wave module, there are two PWM modules instantiated: one outputs the PWM signal, while the other is a pseudo clock ("newClock") for controlling the time it takes to move from one duty cycle to the next:



```
        if (newClock && !(previousClk && newClock)) begin
            if (offset_sampleSelect != previous_offset )
                sample_select <= offset_sampleSelect;
            else begin
                if (sample_select < SAMPLES_MAX)
                    sample_select <= sample_select + 1;
                else
                    sample_select <= 0;
            end
            previous_offset <= offset_sampleSelect;
        end
        previousClk <= newClock;
```

Figure 13: pseudo clock (newClock variable) decides when to move to the next sample, and when to change to a new offset (meaning start the sine wave from a sample other than 0)



```
            else if ( amplitude_select == 2'd1 ) begin
//max duty: 693
                case(sample_select) //125 need 125 cases
                    7'd0 : dutyCycle <= 10'd400;
                    7'd1 : dutyCycle <= 10'd415;
                    7'd2 : dutyCycle <= 10'd430;
                    7'd3 : dutyCycle <= 10'd444;
                    7'd4 : dutyCycle <= 10'd459;
                    7'd5 : dutyCycle <= 10'd474;
                    7'd6 : dutyCycle <= 10'd488;
                    7'd7 : dutyCycle <= 10'd502;
                    7'd8 : dutyCycle <= 10'd516;
                    7'd9 : dutyCycle <= 10'd529;
                    7'd10 : dutyCycle <= 10'd542;
                    7'd11 : dutyCycle <= 10'd555;
                    7'd12 : dutyCycle <= 10'd568;
                    7'd13 : dutyCycle <= 10'd580;
```

Figure 14: example of duty cycle look up table within sin_wave.v

The duty cycles on the lookup table within the sin_wave module were calculated with the help of a Matlab script. The script accounted for the QAM signal frequency (1 kHz), the 100 MHz clock running all the modules, and the number of samples desired for one sine wave period (125). After the number of clock cycles per PWM period had been calculated (800 in this case), the three arrays corresponding to the three amplitudes were shifted vertically by 1.5 (so the minimum voltage would be 0 V). Then, they were divided by the maximum value found among the three, and the normalized elements multiplied by 800. Thus, an 800-duty cycle means a 3 V output over an 8 μs period (10 ns clock cycle period).

The standard period for the PWM module, 800 clock cycles, yields a 125 kHz MHz PWM frequency, as mentioned before.

- **Counter:** this module has many uses; most importantly, when instantiated in the PWM module, it defines for low long an output will receive a logical high or low. The counter module also controls timing of both one sine wave period (1 ms) and of how long the sequence of periods that make up a QAM symbol will be. The counter module can be used either in up counting or in down counting mode. This project uses the up counting mode.

3.2. Receiver

The receiver side would use a TI C2000 F28379D dual core microcontroller to decode the 16-QAM carrier signal, running at 200 MHz (main clock). Due to its high frequency of operation and precision, timer interrupts are precise and can used to retrieve the 12-bit samples from the ADC at the expected times:

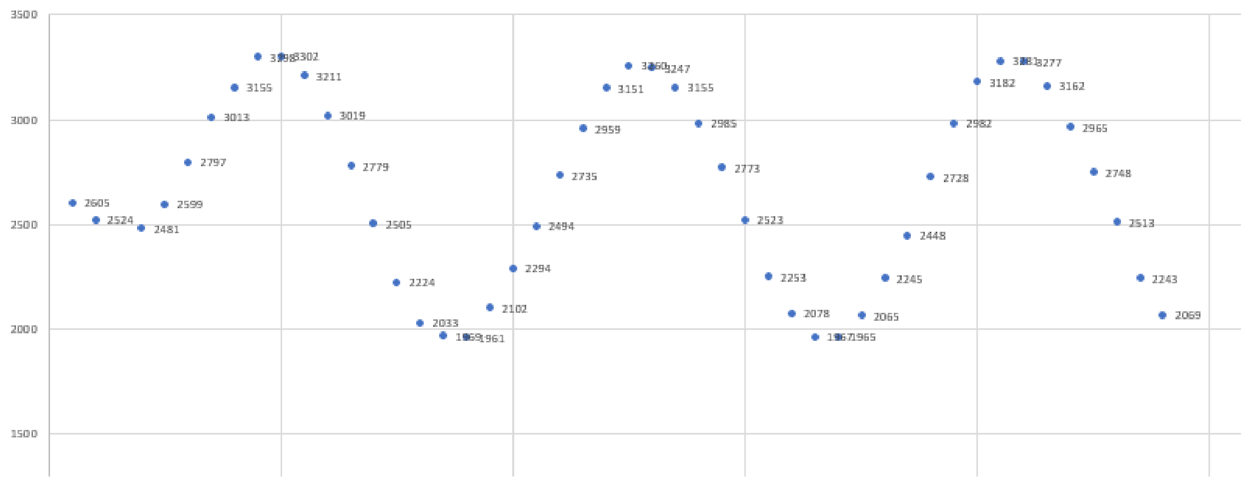
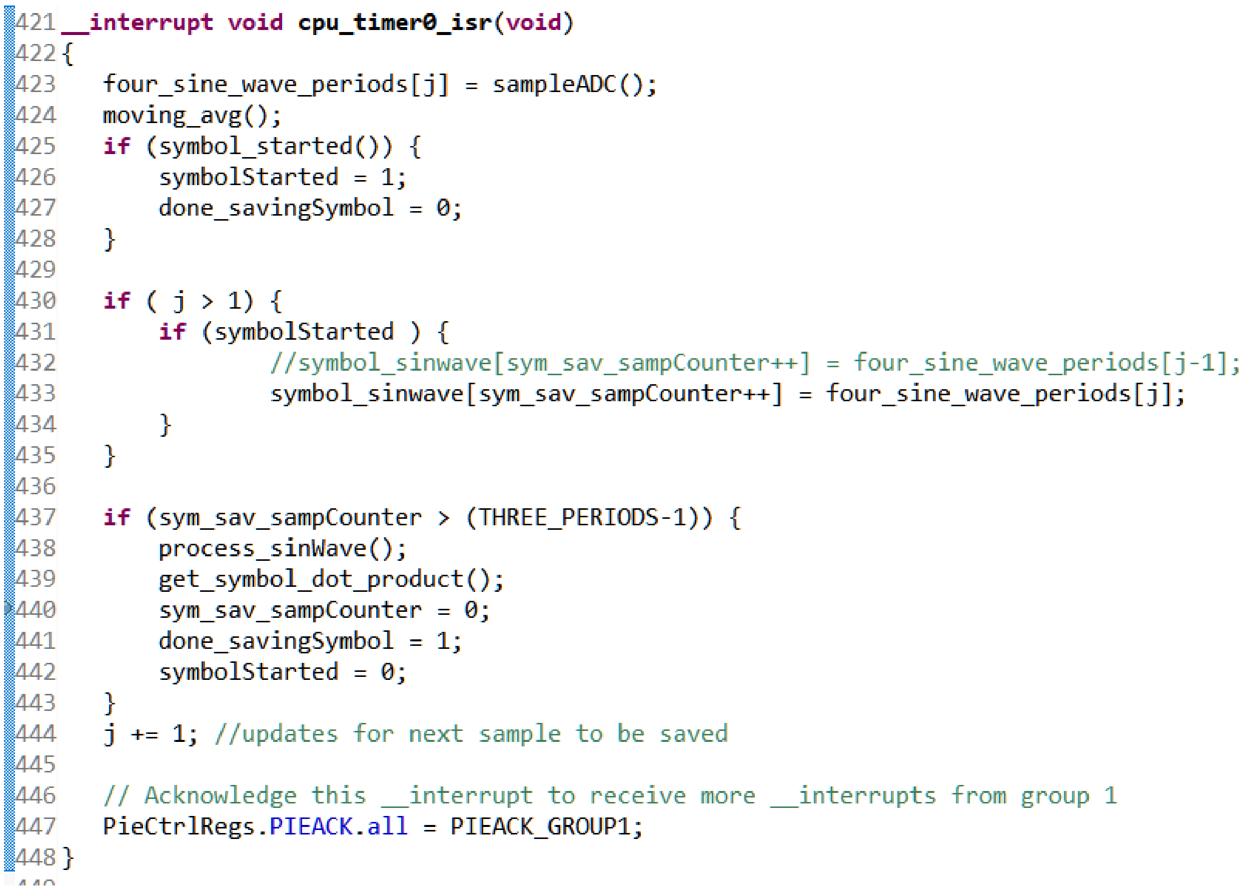


Figure 15: reading from the F28379D ADC; exactly 16 samples can be seen per period

The interrupt function cpu_timerA0_isr controls sampling times. Window acquisition times can be controlled by setting bits on the ADCSOC0CTL special register. The minimum window for a

F28379D’s ADC to convert an analog signal to a 12-bit sample is 75 ns. In this project, the window acquisition time is not an issue, as the 16 kSps sampling rate is rather low, when compared to the microcontroller’s fastest ADC sample rates (roughly 3 MSps in 12 bit).

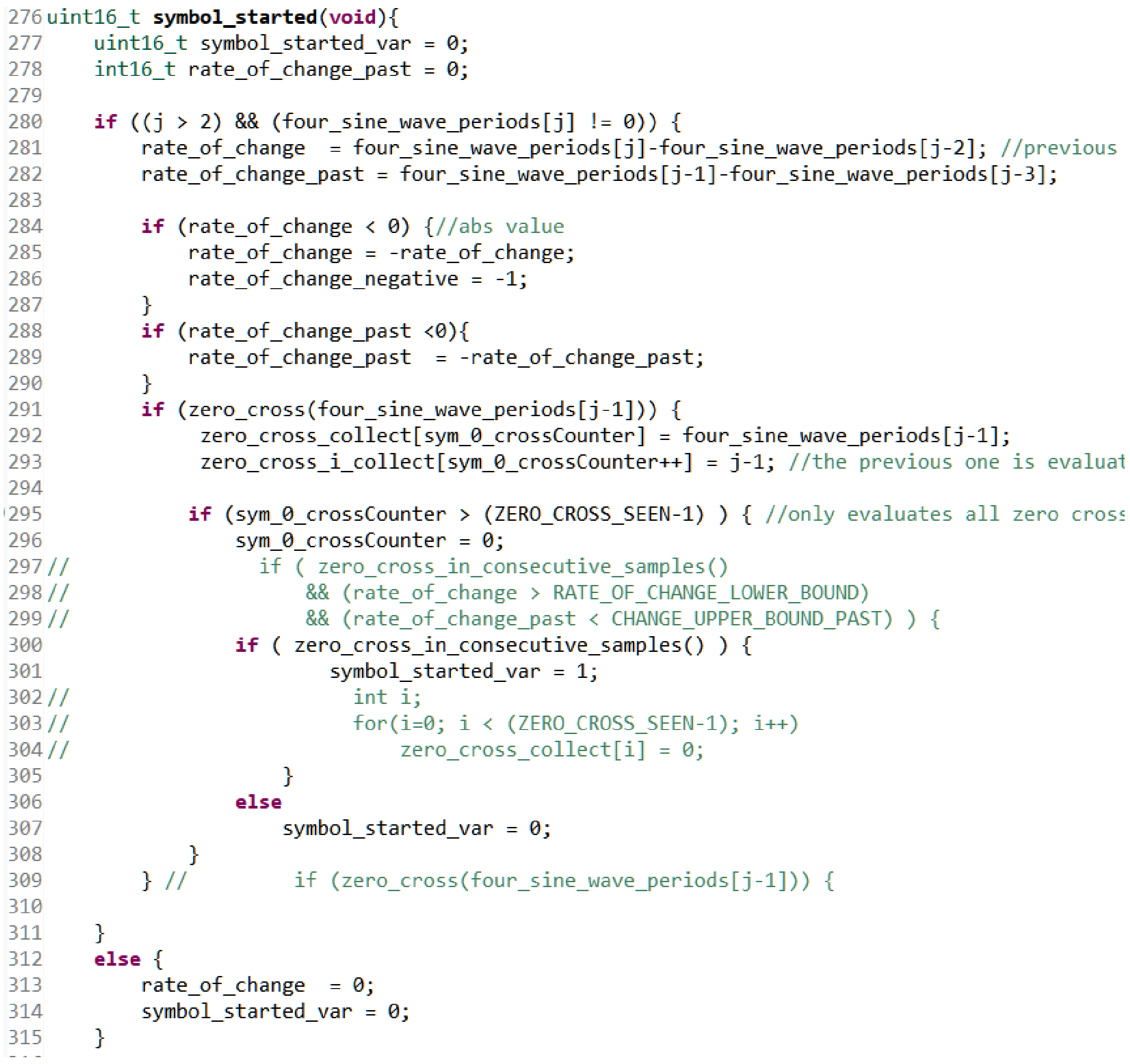


```
421 __interrupt void cpu_timer0_isr(void)
422 {
423    four_sine_wave_periods[j] = sampleADC();
424    moving_avg();
425    if (symbol_started()) {
426        symbolStarted = 1;
427        done_savingSymbol = 0;
428    }
429
430    if ( j > 1) {
431        if (symbolStarted ) {
432                //symbol_sinwave[sym_sav_sampCounter++] = four_sine_wave_periods[j-1];
433                symbol_sinwave[sym_sav_sampCounter++] = four_sine_wave_periods[j];
434        }
435    }
436
437    if (sym_sav_sampCounter > (THREE_PERIODS-1)) {
438        process_sinWave();
439        get_symbol_dot_product();
440        sym_sav_sampCounter = 0;
441        done_savingSymbol = 1;
442        symbolStarted = 0;
443    }
444    j += 1; //updates for next sample to be saved
445
446    // Acknowledge this __interrupt to receive more __interrupts from group 1
447    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
448 }
```

Figure 16: the ISR cpu_timerA0_isr samples the ADC, calls helper functions that check whether a symbol started , and keeps track of the saved symbol’s indeces (lines 431 and 437)

The other core function of the receiving side is **symbol_started()**, a function called within the timer interrupt routine. **symbol_started()** returns a 1 for true (symbol started) or a 0 for false (not yet the start of a symbol). This function calls helper functions for checking if a zero-cross was found (line 291), if that zero-cross was found a certain amount of times (line 295), and if those were consecutive (line 300).

Beyond the consecutive zero-crossings found, **symbol_started()** was also tested using the difference between two samples, one-sample apart from each other, as one of the criteria for finding the beginning of the QAM sine wave. Nevertheless, reliability did not improve much over the zero-crossings, for the reasons stated before: the differences recorded within the flat region are too similar to the differences recorded at the beginning of the carrier modulated with the smallest amplitude.



```c
uint16_t symbol_started(void){
    uint16_t symbol_started_var = 0;
    int16_t rate_of_change_past = 0;

    if ((j > 2) && (four_sine_wave_periods[j] != 0)) {
        rate_of_change  = four_sine_wave_periods[j]-four_sine_wave_periods[j-2]; //previous
        rate_of_change_past = four_sine_wave_periods[j-1]-four_sine_wave_periods[j-3];

        if (rate_of_change < 0) {//abs value
            rate_of_change = -rate_of_change;
            rate_of_change_negative = -1;
        }
        if (rate_of_change_past <0){
            rate_of_change_past  = -rate_of_change_past;
        }
        if (zero_cross(four_sine_wave_periods[j-1])) {
             zero_cross_collect[sym_0_crossCounter] = four_sine_wave_periods[j-1];
             zero_cross_i_collect[sym_0_crossCounter++] = j-1; //the previous one is evaluat

            if (sym_0_crossCounter > (ZERO_CROSS_SEEN-1) ) { //only evaluates all zero cros
                sym_0_crossCounter = 0;
//                if ( zero_cross_in_consecutive_samples()
//                    && (rate_of_change > RATE_OF_CHANGE_LOWER_BOUND)
//                    && (rate_of_change_past < CHANGE_UPPER_BOUND_PAST) ) {
                if ( zero_cross_in_consecutive_samples() ) {
                        symbol_started_var = 1;
//                        int i;
//                        for(i=0; i < (ZERO_CROSS_SEEN-1); i++)
//                            zero_cross_collect[i] = 0;
                    }
                else
                    symbol_started_var = 0;
            }
        } //         if (zero_cross(four_sine_wave_periods[j-1])) {

    }
    else {
        rate_of_change  = 0;
        symbol_started_var = 0;
    }
```

Figure 17: the **symbol_started()** function

Conclusion

QAM_land spent easily more than one hundred and thirty hours implementing the project. From my part, writing the Verilog code for the encoding on the FPGA was a challenge on itself – and possibly, the FPGA was a mistake as platform choice, if completing the project was the main goal for this class.

Because I had started writing the modules for the FPGA during the mini project, I thought I had enough Verilog code to implement the whole encoding part quickly – the "quickly" proved to be false. In comparison, within one afternoon spent with a MSP430, I already had results: a DDS sending the encoded preamble of the U.S. constitution. That DDS encoding had its own issues with the flat region between symbols, but I infer they could be corrected with a couple of more hours of experimentation. Thus, developing the encoding on an FPGA was, I feel, not the right choice, even though I was proud to see the exact same results in both the MATLAB simulations and in the oscilloscope readings of the filtered PWM signal.

Another mistake was an attempt to implement the DDS driver on the FPGA. It worked sparsely, which is why the Verilog modules related to the DDS were not included in this final report. It took almost 13 hours to get the modules to barely work. I infer the problem was that I had not implemented the reset and the power off routines, but after a day and a half of work, I considered that it had surpassed the diminishing returns threshold. It would have been nice to have only one device driving both the DDS (28.8 MHz) and producing the QAM carrier, but a MSP430 took 45 min of work with a library found online for achieving results (the same library used by other lab 3 groups, slightly modified to not use any multiply/division). I must say I was delighted by the challenge of implementing QAM on the FPGA - the absolute control over the signal that the FPGA provides is awesome – but it took precious hours that could have been better spent implementing the decoding side instead.

Finally, beyond the time spent with the FPGA, I also think that the team suffered from a lack of defined/official leadership. I expected that everyone would just spend the time needed to get their parts done without external pressure; nevertheless, that was not the case. I am guilty, thus, of not being responsible for the leadership in this case. In the future, I will never assume that people are automatically committed to a project's completion without some external pressure.