Lab 06 Report: AD1Pmod and DA2Pmod in FreeRTOS

Peter Mowen
peter.mowen@temple.edu

Summary

This lab will explore digital signal processing using the Zybo board, AD1Pmod & DA2Pmod, FreeRTOS, and the EE Board. First, a piece of standalone software that passes an input to an output will be examined. Next, a naïve approach to DSP using FreeRTOS will be explored. Then, a less naïve approach to DSP using FreeRTOS will be examined. Finally, FreeRTOS will be used to take the square root of an input voltage.

Introduction

Digital signal processing is the act of taking an analog signal, digitizing it, then doing some kind of processing to the digital signal. It is an important concept to cover because computers do not operate on analog signals. To digitize a signal, a computer discretizes and quantizes the signal. Discretizing is the act of reading the signal for discrete time intervals. Quantizing a sample is the act of assigning the value of the sample a discrete value.

For example, the AD1Pmod and DA2Pmod are both 12-bit converters. This means that voltage values are assigned a number between 0 and 4095 where 0 corresponds to 0V and 4095 corresponds to 3.3V. This implies that $\frac{sampleVal}{4095} = \frac{voltageVal}{3.3}$. From this ratio, a sample value can be assigned to a voltage value by taking the ratio of the voltage and 3.3 and multiplying it by 4095. Similarly, one can find a voltage value by taking the ratio of a sample value and 4095 and multiplying it by 3.3.

The AD1Pmod and DA2Pmod peripherals use the serial peripheral interface (SPI) to communicate with the Zybo. IP blocks for both were provided by the professor. To read in a sample value from AD1Pmod, the Zybo drives a chip select (CS) signal low for 16 clock cycles of a 30MHz clock. During that time, each bit of the sample value is read into a memory location in the Zybo over a data wire. Similarly, when the Zybo wants to write-out a voltage, it drives the DA2Pmod CS signal low for 16 clock cycles of a 50MHz clock. During that time, each bit of the sample is transferred from a Zybo memory location to the DA2Pmod over a data wire.

The number of samples per second, or throughput, of a program depends on how the program is written. One can find the throughput of a program by monitoring the CS signal of the AD1Pmod and DA2Pmod. The time it takes for the AD1 CS signal to go low to high back to low is the time it takes to read in one sample. Assuming each sample is written-out before the next sample is read in, the throughput can be found by taking the reciprocal of the time it takes to read one sample.

A 100Hz sine wave with an amplitude of 1V and a DC offset of 1.5V was used to test the various DSP programs explored throughout this lab.

Discussion

Hardware Setup in Vivado HLx
Below is a screenshot of the block diagram from Vivado HLx:

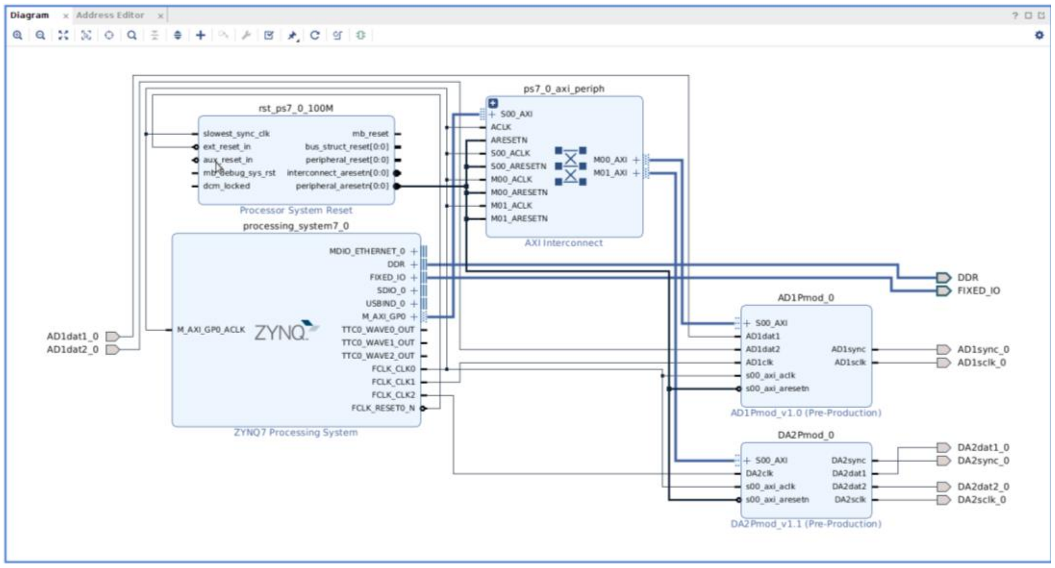


Figure 1: Vivado HLx Hardware Block Diagram

The AD1Pmod and DA2Pmod IPs were provided by Dr. Silage. Note that each has a clock line running to it from the processing system block. The clock for AD1Pmod is set 30MHz while the clock for DA2Pmod is set for 50MHz. The lines all the way to the left, AD1dat1_0 and AD1dat2_0 represent the inputs A0 and A1 on the AD1Pmod board. The lines DA2dat1_0 and DA2dat2_0 represent the channel 0 and channel 1 outputs on the DA2Pmod board. The Pmods are connected to the rest of the system via the AXI bus.

Physical Hardware Setup
Below is a picture of the physical hardware setup:

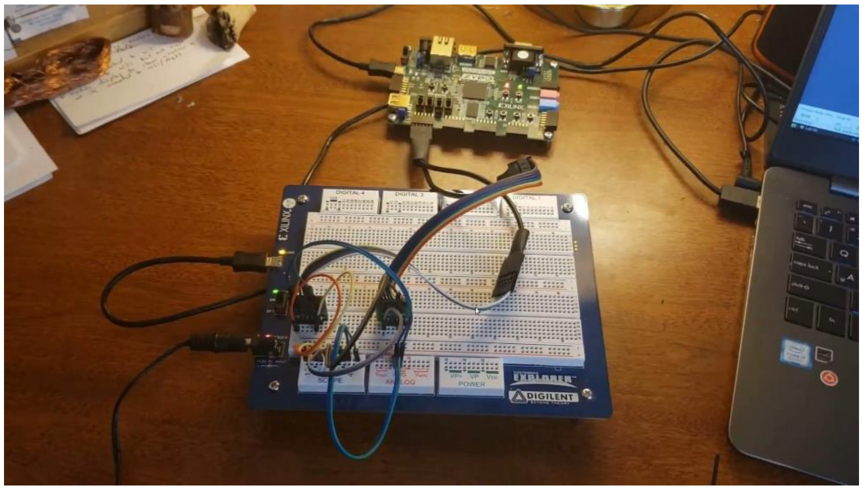
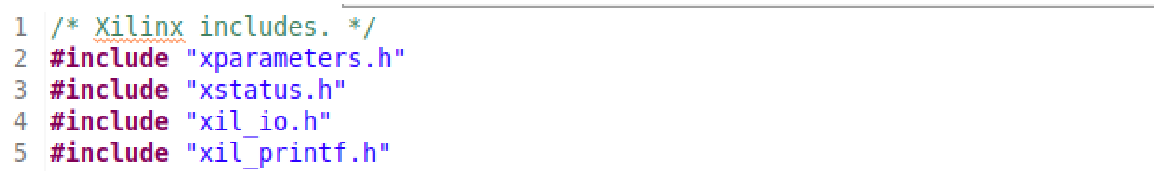


Figure 2: Physical Hardware Setup

The AD1Pmod and DA2Pmod boards are plugged-in to the breadboard. Jumper wires are used to connect the Pmods to the hydra cable, which is in turn connected to port JE of the Zybo board. Wires from the EE boards AWG are run to both oscilloscope channel 4 and input A0 on the AD1Pmod. Wires from the channel 0 output on the DA2 Pmod are connected to oscilloscope channel 3. Oscilloscope channels 1 and 2 are connected to the CS pins of AD1Pmod and DA2Pmod respectively. All of the grounds are tied together.

Task 1: Standalone Passthrough software

Below is a screenshot of the include statements for the standalone version:

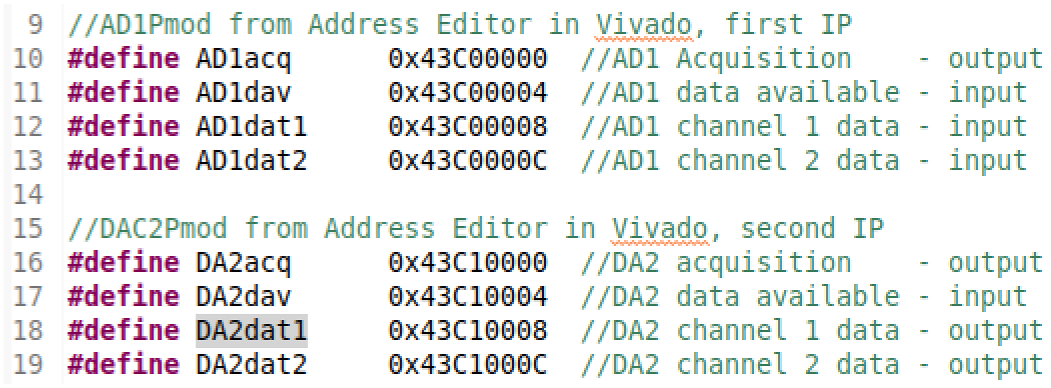


```
/* Xilinx includes. */
#include "xparameters.h"
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"
```

Figure 3: Include statements for standalone version

The only new file in that list is `xil_io.h`. This file includes the `Xil_In32()` and `Xil_Out32()` functions used later in the program. The former reads-in the 32-bit contents of a memory location. The later writes-out content to a 32-bit location.

The screenshot below shows the preprocessor define statements for various memory locations related to the AD1Pmod and DA2Pmod:

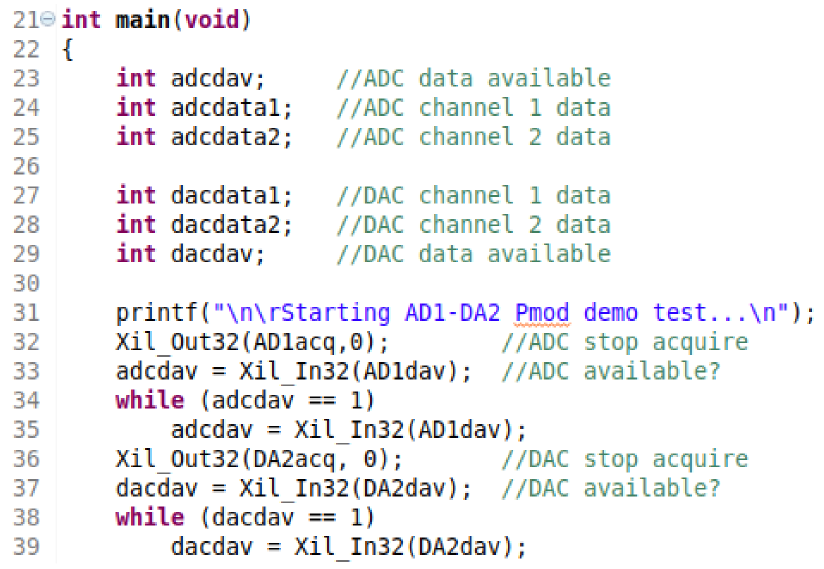


```
//AD1Pmod from Address Editor in Vivado, first IP
#define AD1acq      0x43C00000  //AD1 Acquisition    - output
#define AD1dav      0x43C00004  //AD1 data available - input
#define AD1dat1     0x43C00008  //AD1 channel 1 data - input
#define AD1dat2     0x43C0000C  //AD1 channel 2 data - input

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq      0x43C10000  //DA2 acquisition    - output
#define DA2dav      0x43C10004  //DA2 data available - input
#define DA2dat1     0x43C10008  //DA2 channel 1 data - output
#define DA2dat2     0x43C1000C  //DA2 channel 2 data - output
```

Figure 4: Preprocessor defines for AD1Pmod and DA2Pmod memory address locations

Each Pmod as four address locations: XXXacq, XXXdav, XXXdat1, and XXXdat2. The suffix “acq” stands for “data acquire.” Writing a 1 to the AD1Pacq memory location tells the AD1Pmod to acquire data, while writing a 1 to the DA2acq memory location tells the DA2Pmod to write the data in DA2dat1 and DA2dat2 to channels 0 and 1 respectively. The suffix “dav” stands for “data available.” The AD1dav memory location will be 1 when AD1Pmod has finished reading-in data. The data from channel A0 will be stored in AD1dat1 while the data from channel A1 will be stored in AD1dat2. A 1 will be sitting in the DA2dav memory location when DA2Pmod has finished writing the contents of the data memory location to the output channels.

Below is a screenshot of the setup portion of the `main` function:

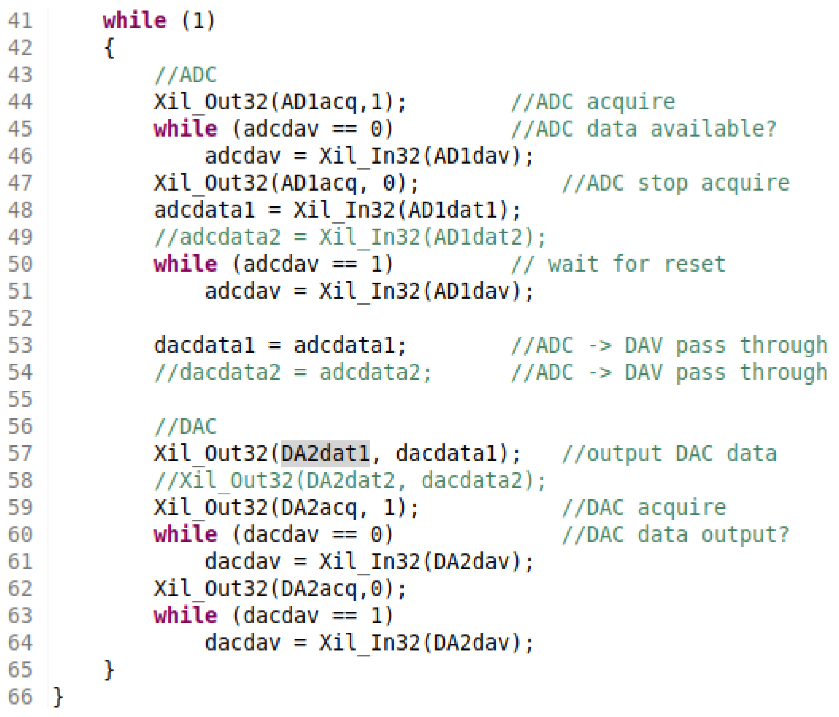


```c
21 int main(void)
22 {
23     int adcdav;      //ADC data available
24     int adcdata1;    //ADC channel 1 data
25     int adcdata2;    //ADC channel 2 data
26
27     int dacdata1;    //DAC channel 1 data
28     int dacdata2;    //DAC channel 2 data
29     int dacdav;      //DAC data available
30
31     printf("\n\rStarting AD1-DA2 Pmod demo test...\n");
32     Xil_Out32(AD1acq,0);          //ADC stop acquire
33     adcdav = Xil_In32(AD1dav);    //ADC available?
34     while (adcdav == 1)
35         adcdav = Xil_In32(AD1dav);
36     Xil_Out32(DA2acq, 0);         //DAC stop acquire
37     dacdav = Xil_In32(DA2dav);    //DAC available?
38     while (dacdav == 1)
39         dacdav = Xil_In32(DA2dav);
```

Figure 5: Setup portion of `main` function

Lines 23-25 create variables to hold the value stored at AD1dav, AD1dat1, and AD1dat 2 respectively. Lines 27-29 do the same thing for DA2dav, DA2dat1, and DA2dat2. The code from lines 32-39 ensure that the data acquire and data available signals are low before the main `while` loop starts. Line 32 writes a 0 to memory location AD1acq. Line 33 reads-in the contents at memory location. Lines 34-35 loop until the data available signal goes low. A similar process is run for the DA2Pmod.

The screenshot below shows the loop part of the main function:



```c
41     while (1)
42     {
43         //ADC
44         Xil_Out32(AD1acq,1);          //ADC acquire
45         while (adcdav == 0)           //ADC data available?
46             adcdav = Xil_In32(AD1dav);
47         Xil_Out32(AD1acq, 0);             //ADC stop acquire
48         adcdata1 = Xil_In32(AD1dat1);
49         //adcdata2 = Xil_In32(AD1dat2);
50         while (adcdav == 1)           // wait for reset
51             adcdav = Xil_In32(AD1dav);
52
53         dacdata1 = adcdata1;          //ADC -> DAV pass through
54         //dacdata2 = adcdata2;        //ADC -> DAV pass through
55
56         //DAC
57         Xil_Out32(DA2dat1, dacdata1);     //output DAC data
58         //Xil_Out32(DA2dat2, dacdata2);
59         Xil_Out32(DA2acq, 1);             //DAC acquire
60         while (dacdav == 0)               //DAC data output?
61             dacdav = Xil_In32(DA2dav);
62         Xil_Out32(DA2acq,0);
63         while (dacdav == 1)
64             dacdav = Xil_In32(DA2dav);
65     }
66 }
```

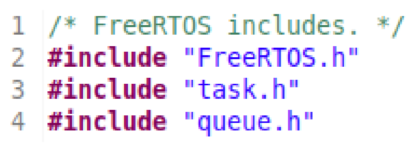
Figure 6: Loop portion of `main` function

This code reads in a sample from the ADC, the writes the sample directly to the DAC. Line 44 writes a 1 to memory location AD1acq. Lines 45-46 loop until data becomes available. Line 47 writes a 0 to AD1acq. Line 48 copies the contents of memory location AD1dat1 to a local variable. This sample represents the voltage on channel A0. Line 49 would do the same for channel A1 but is commented out since we are only using one channel in this lab. All other references to the second channel on AD1 and DA2 have been commented-out. Lines 50-51 loop until the data available signal goes low again. Line 53 passes the data in adcdata1 to dacdata1. Line 57 writes the contents of dacdata1 to memory location DA2dat1. This copies the sample read-in from the ADC memory location to the DAC memory location. Line 59 writes a 1 to memory location DA2acq. Lines 60-61 loop until dacdav does not equal 0. At this point, the sample stored in DA2dat1 has been converted back to a voltage and written to the channel 0 wire. Line 62 writes a 0 to tha DA2acq memory location. Lines 63-64 loop until the dacdav signal no longer equals 1. At this point, the program loops and reads in the next sample.

An analysis of the throughput of this program as well as a screenshot of the passed-through sine wave can be found in the Results section of this report.

Task 2: FreeRTOS Implementation of Passthrough – a Naïve Approach

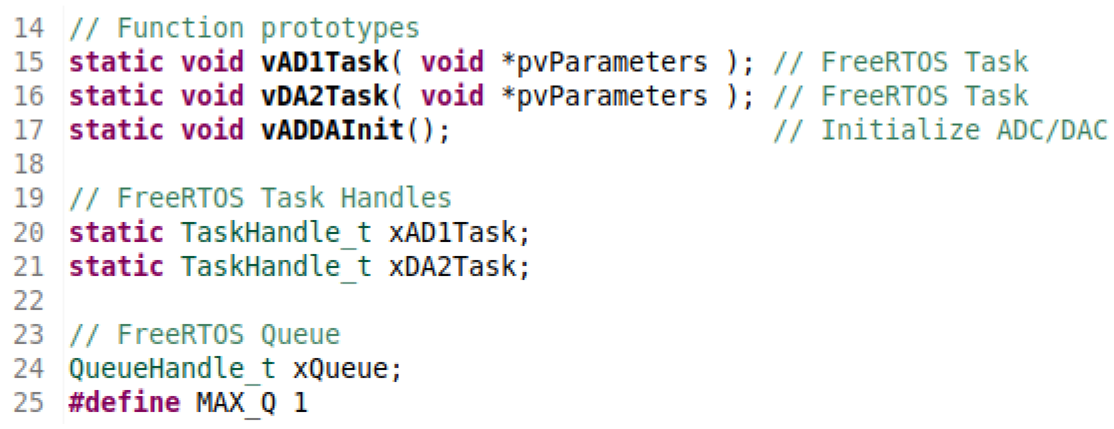
Much of the code from the standalone version is reused in the FreeRTOS version. Below is the screenshot of additional include statements needed for the FreeRTOS version:



```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
```

Figure 7: Additonal FreeRTOS include statements

The only new statement in this list is `queue.h`. This header file adds data types and functions related to dealing with queues to the code. As in previous FreeRTOS projects, function prototypes are created for the tasks, and some helper functions. Additionally, task and queue handles are created:

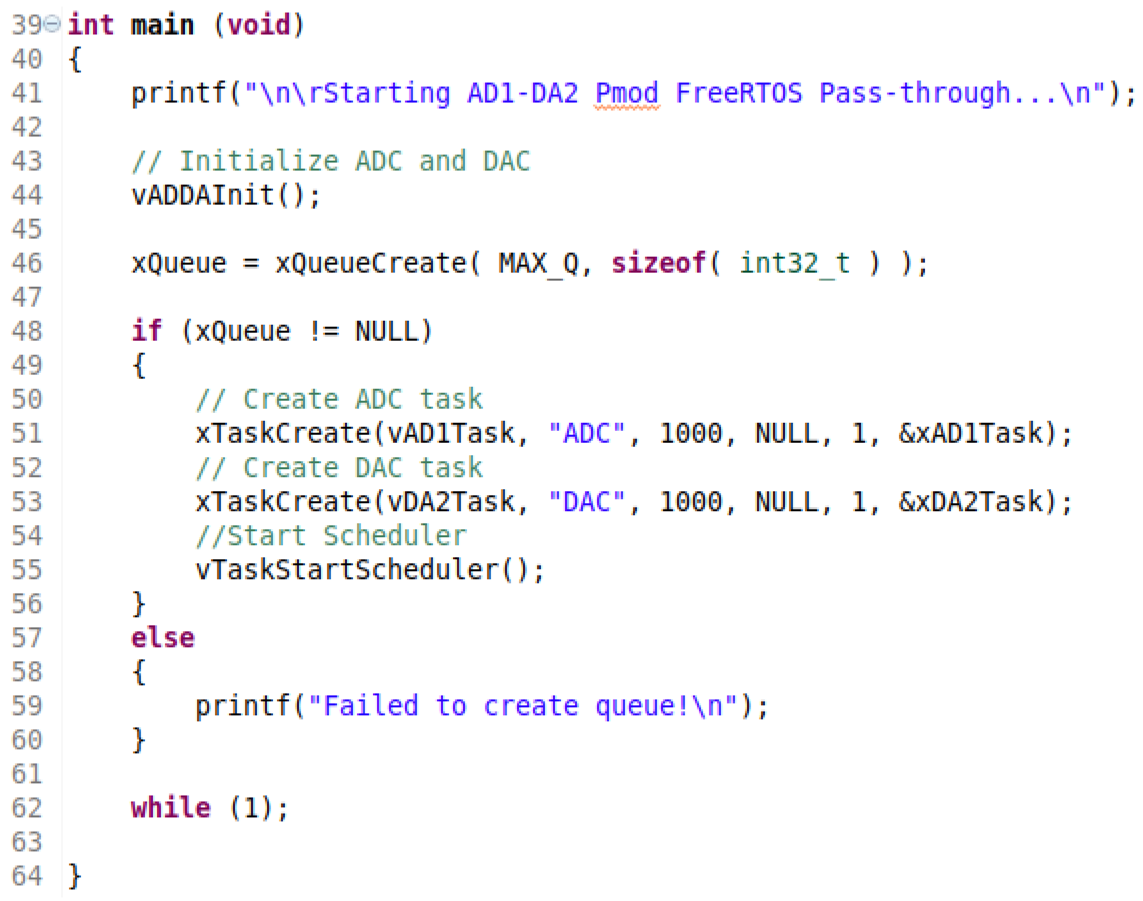


```
// Function prototypes
static void vAD1Task( void *pvParameters ); // FreeRTOS Task
static void vDA2Task( void *pvParameters ); // FreeRTOS Task
static void vADDAInit();                    // Initialize ADC/DAC

// FreeRTOS Task Handles
static TaskHandle_t xAD1Task;
static TaskHandle_t xDA2Task;

// FreeRTOS Queue
QueueHandle_t xQueue;
#define MAX_Q 1
```

Figure 8: Task prototypes and handles. Queue handle

Prototypes are created for the AD1 task, DA2 task, and function to initialize the AD and DA converters. Next, task handles are created for the tasks. Finally, a queue handle is created, and a max queue size is defined.

The main task initializes the AD and DA converters and creates the AD1 and DA2 tasks:



```
int main (void)
{
    printf("\n\rStarting AD1-DA2 Pmod FreeRTOS Pass-through...\n");

    // Initialize ADC and DAC
    vADDAInit();

    xQueue = xQueueCreate( MAX_Q, sizeof( int32_t ) );

    if (xQueue != NULL)
    {
        // Create ADC task
        xTaskCreate(vAD1Task, "ADC", 1000, NULL, 1, &xAD1Task);
        // Create DAC task
        xTaskCreate(vDA2Task, "DAC", 1000, NULL, 1, &xDA2Task);
        //Start Scheduler
        vTaskStartScheduler();
    }
    else
    {
        printf("Failed to create queue!\n");
    }

    while (1);

}
```

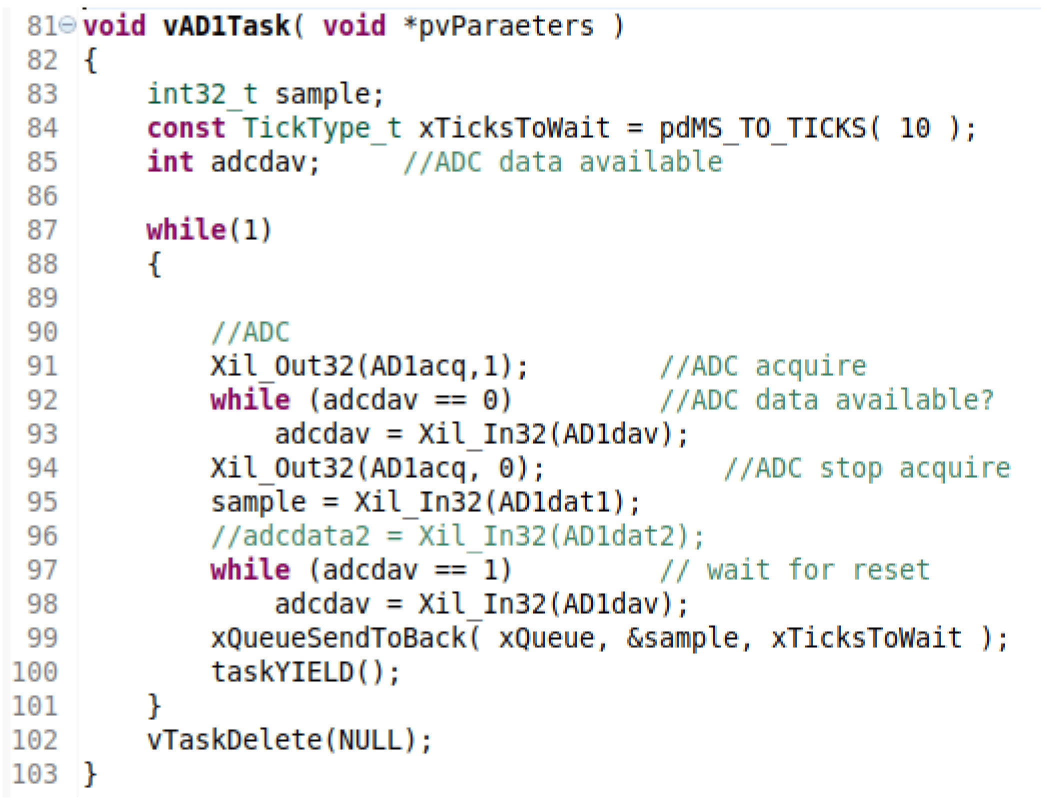
Figure 9: FreeRTOS Passthrough `main` function

The function `vADDAInit()` runs the code from lines 31-39 of the standalone code to ensure that the data acquire and data available signals are driven low before the program starts.

Next, a queue is created at attached to the `xQueue` handle. The first argument of `xQueueCreate()` sets the maximum queue size. The second argument sets the size of a single item in the queue. Since this queue is going to hold the data from 32-bit wide memory locations, the size of a single item is the size of a 32-bit integer.

If the queue was created successfully, then each task is created with a stack size of 1000, no input parameters, and priority level of 1, and the task scheduler is started. Else, a message is printed to the console and the program enters an infinite while loop.

Below is a screenshot of a naïve approach to the AD1 task:

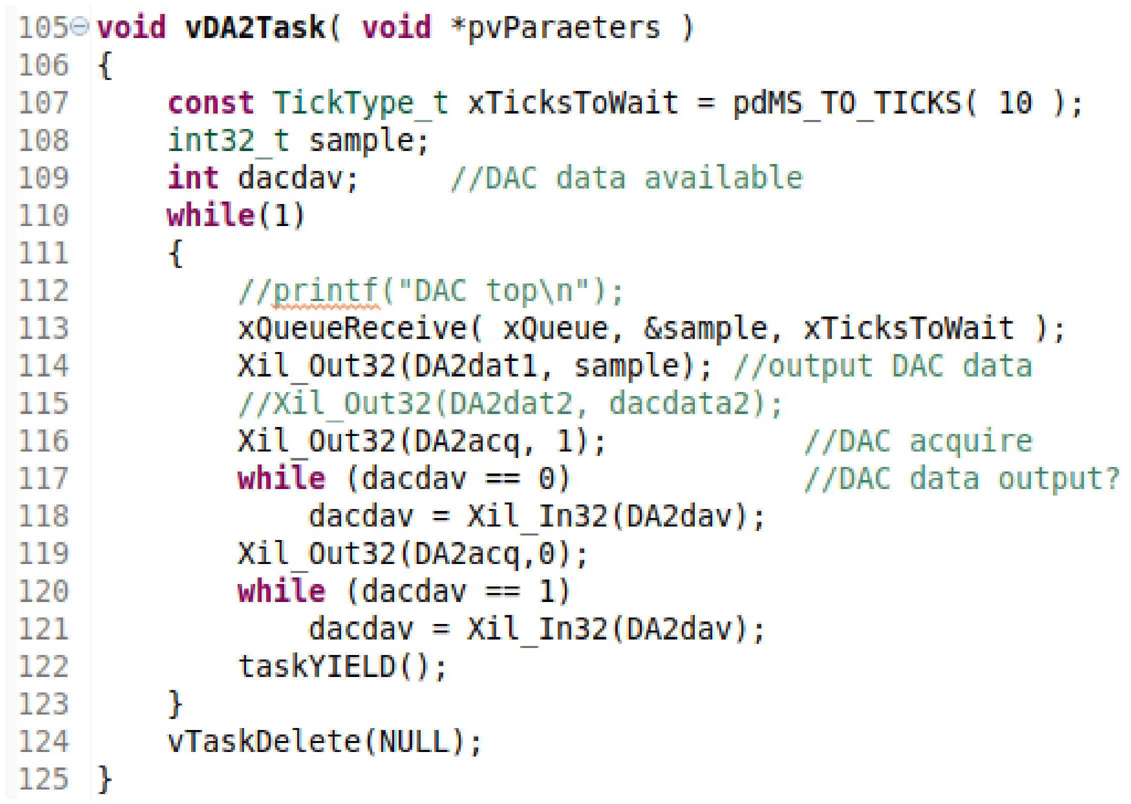


```
void vAD1Task( void *pvParaeters )
{
    int32_t sample;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
    int adcdav;     //ADC data available

    while(1)
    {

        //ADC
        Xil_Out32(AD1acq,1);        //ADC acquire
        while (adcdav == 0)         //ADC data available?
            adcdav = Xil_In32(AD1dav);
        Xil_Out32(AD1acq, 0);           //ADC stop acquire
        sample = Xil_In32(AD1dat1);
        //adcdata2 = Xil_In32(AD1dat2);
        while (adcdav == 1)         // wait for reset
            adcdav = Xil_In32(AD1dav);
        xQueueSendToBack( xQueue, &sample, xTicksToWait );
        taskYIELD();
    }
    vTaskDelete(NULL);
}
```

Figure 10: Naïve vAD1Task

The code for this task is essentially a copy-paste of the code from the standalone. The data acquire signal is driven higher, then the code waits for data to become available. Once the data is available, the data acquire signal is driven low. Next the data is read into the `sample` variable. Then the code waits for the data available signal to go low. Next, the sample is written to the queue. The first argument of `xQueueSendToBack` is the queue handle for the queue to which to write. The second argument is the address of the item to write to the queue. The data at that address is copied onto the queue. The final argument is the number of ticks to wait if the queue is full. This was set to an arbitrary number. The queue size is 1 and the DA2 task immediately reads from the queue so AD1 should never have to wait to write to the queue. Finally, `taskYIELD()` is called. It is not necessary to call `taskYIELD()`. Since the queue size is 1, when AD1 loops back around, it would find the queue to be full and start waiting for the queue to empty. However, using `taskYIELD()` makes it explicit that the next task will run.

Below is a screenshot of a naïve approach to the DA2 task:



```c
void vDA2Task( void *pvParaeters )
{
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
    int32_t sample;
    int dacdav;     //DAC data available
    while(1)
    {
        //printf("DAC top\n");
        xQueueReceive( xQueue, &sample, xTicksToWait );
        Xil_Out32(DA2dat1, sample); //output DAC data
        //Xil_Out32(DA2dat2, dacdata2);
        Xil_Out32(DA2acq, 1);           //DAC acquire
        while (dacdav == 0)             //DAC data output?
            dacdav = Xil_In32(DA2dav);
        Xil_Out32(DA2acq,0);
        while (dacdav == 1)
            dacdav = Xil_In32(DA2dav);
        taskYIELD();
    }
    vTaskDelete(NULL);
}
```

Figure 11: Naïve approach to DA2 task

Similar to the naïve approach to the AD1 task, the naïve approach to the DA2 task is essentially a copy-paste of the standalone code. It reads a sample from the queue. The `xQueueReceive` function works similarly to the `xQueueSendToBack`: the first argument is the queue handle of the queue to read from, the second argument is the address of the variable in which to store the queue item, and the third argument is the number of ticks to wait if the queue is empty. Once the sample is read-in to the task, it is written to the DA2dat1 memory location. The data acquire signal is driven high, then the function waits for the data to be written out. Next, the data available signal is driven low and the function waits for the data available signal to go low. Finally, the task yields the rest of it tick time.

An analysis of the throughput of this program as well as a screenshot of the passed-through sine wave can be found in the Results section of this report.

Task 2: FreeRTOS Implementation of Passthrough – a Less Naïve Approach

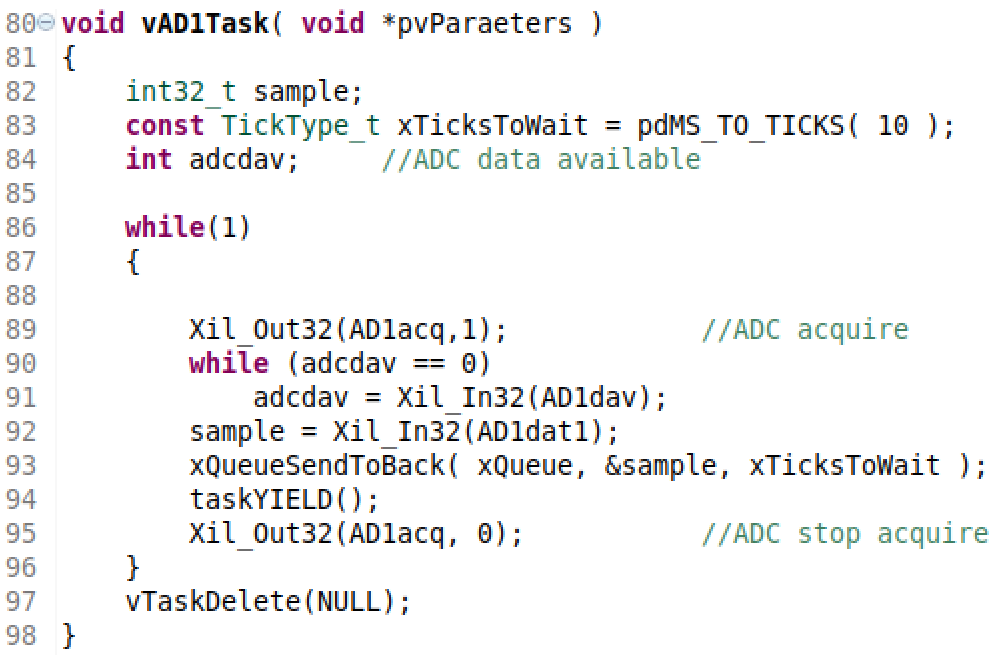
The only differences between this code and the Naïve approach is in the AD1 and DA2 tasks. All other parts are the same.

Before discussing the new tasks, it is worth mentioning how long the standalone and naïve software have to wait for the data available signal to change. Below is a snippet of output from updated code. The coded counts the number of iterations through the while loops that wait on the data available signals:

```
Starting AD1-DA2 Pmod demo test...
adcdav ON count = 1053884
adcdav OFF count = 1
dacdav ON count = 3
dacdav OFF count = 1
adcdav ON count = 5
adcdav OFF count = 1
dacdav ON count = 3
dacdav OFF count = 1
adcdav ON count = 6
adcdav OFF count = 1
dacdav ON count = 4
dacdav OFF count = 1
…
```

Note that the number of times through the while loop that waits for the data available signal to go low is always 1 and the number of times through the while loop to wait for the data available to go high is always greater than 1. This fact will be used to increase throughput.

Below is a screenshot of the less naïve approach to the AD1 task:

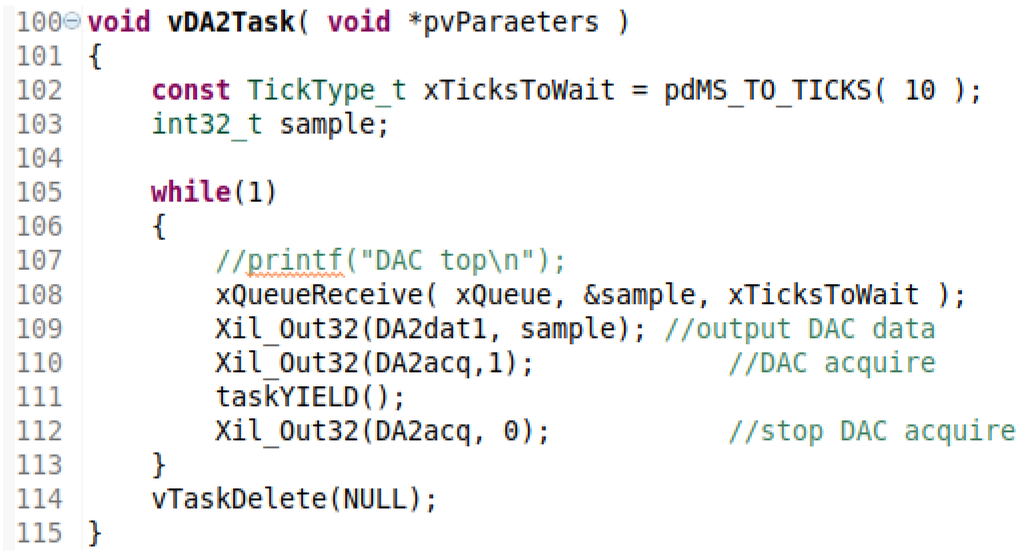


```
80 void vAD1Task( void *pvParaeters )
81 {
82     int32_t sample;
83     const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
84     int adcdav;     //ADC data available
85 
86     while(1)
87     {
88 
89         Xil_Out32(AD1acq,1);            //ADC acquire
90         while (adcdav == 0)
91             adcdav = Xil_In32(AD1dav);
92         sample = Xil_In32(AD1dat1);
93         xQueueSendToBack( xQueue, &sample, xTicksToWait );
94         taskYIELD();
95         Xil_Out32(AD1acq, 0);           //ADC stop acquire
96     }
97     vTaskDelete(NULL);
98 }
```

Figure 12: Less naïve AD1 task

The main change is the placement of `taskYIELD()` and where the data acquire signal is driven low. Once a new sample has been read-in and written to the queue, AD1 tasks immediately yields to the DA2 task. When DA2 yields back to AD1, AD1 resumes at line 95. It immediately jumps back to line 89 and drives the data acquire signal high again, then the cycle repeats. An even better version of this code might yield while waiting for the data to become available.

Below is a screenshot of less naïve version of the DA2 task:



```
100  void vDA2Task( void *pvParaeters )
101  {
102      const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
103      int32_t sample;
104
105      while(1)
106      {
107          //printf("DAC top\n");
108          xQueueReceive( xQueue, &sample, xTicksToWait );
109          Xil_Out32(DA2dat1, sample); //output DAC data
110          Xil_Out32(DA2acq,1);         //DAC acquire
111          taskYIELD();
112          Xil_Out32(DA2acq, 0);        //stop DAC acquire
113      }
114      vTaskDelete(NULL);
115  }
```

Figure 13: Code for less naïve DA2 task

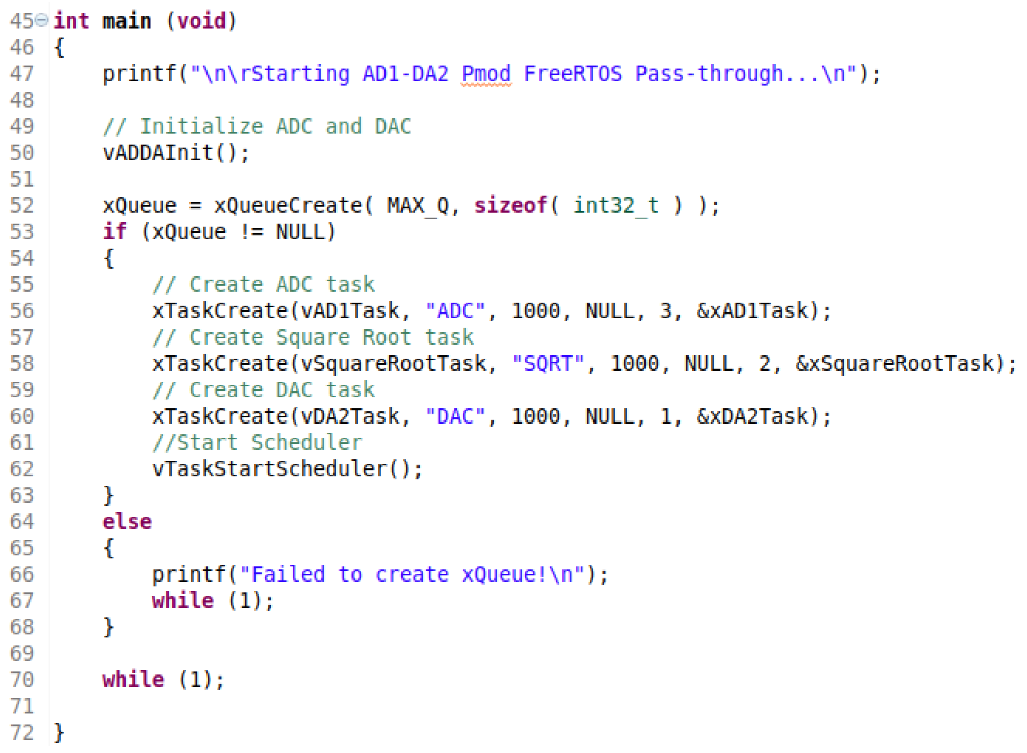
Again, the main difference is the placement of `taskYIELD()`. DA2 immediately yields once it tells the Pmod to write the data to the wire. At this point, the code jumps back to line 95 in the AD1 task. Once DA2 runs again, it starts at line 112. The data acquire signal is driven low, then the function loops and immediately drives it high.

An analysis of the throughput of this program as well as a screenshot of the passed-through sine wave can be found in the Results section of this report.

Task 3: Square Root Task

In this part of the lab, a new task will be added to the less naïve FreeRTOS program which takes the square root of the input wave, then writes that to the output. The setup to the code is almost identical to the previous FreeRTOS code. The only difference is the inclusion of the `math.h` C library. It is needed to be able to use the square root function. Two additional preprocessor definitions are made: `V_REF` and `MAX_SAMPLE_SIZE`. The `V_REF` is defined as 3.3, representing the reference voltage for the Pmods: 3.3V. The `MAX_SAMPLE_SIZE` is set to 4096 because the ADC is a 12-bit ADC and 2^{12} = 4096. Though the max sample number is 4095, making it 4096 has negligible effect on the output for this program.

Below is the main function of this new program:

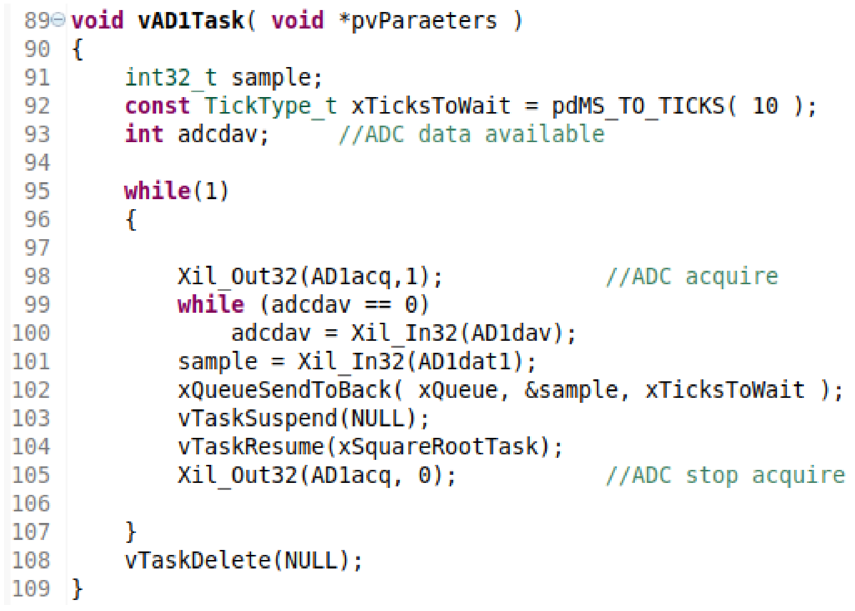


```
int main (void)
{
    printf("\n\rStarting AD1-DA2 Pmod FreeRTOS Pass-through...\n");

    // Initialize ADC and DAC
    vADDAInit();

    xQueue = xQueueCreate( MAX_Q, sizeof( int32_t ) );
    if (xQueue != NULL)
    {
        // Create ADC task
        xTaskCreate(vAD1Task, "ADC", 1000, NULL, 3, &xAD1Task);
        // Create Square Root task
        xTaskCreate(vSquareRootTask, "SQRT", 1000, NULL, 2, &xSquareRootTask);
        // Create DAC task
        xTaskCreate(vDA2Task, "DAC", 1000, NULL, 1, &xDA2Task);
        //Start Scheduler
        vTaskStartScheduler();
    }
    else
    {
        printf("Failed to create xQueue!\n");
        while (1);
    }

    while (1);

}
```

Figure 14: main function of Square Root DSP program

The main difference between this main function and the previous one is in the task creation section. Three tasks are created: the AD1 task is created with priority level 3; the Square Root task is created with priority level 2; and the DA2 task is created with priority level 1. The tasks were given different priority levels to ensure they run in the correct order.

Below is a screenshot of the AD1 task:

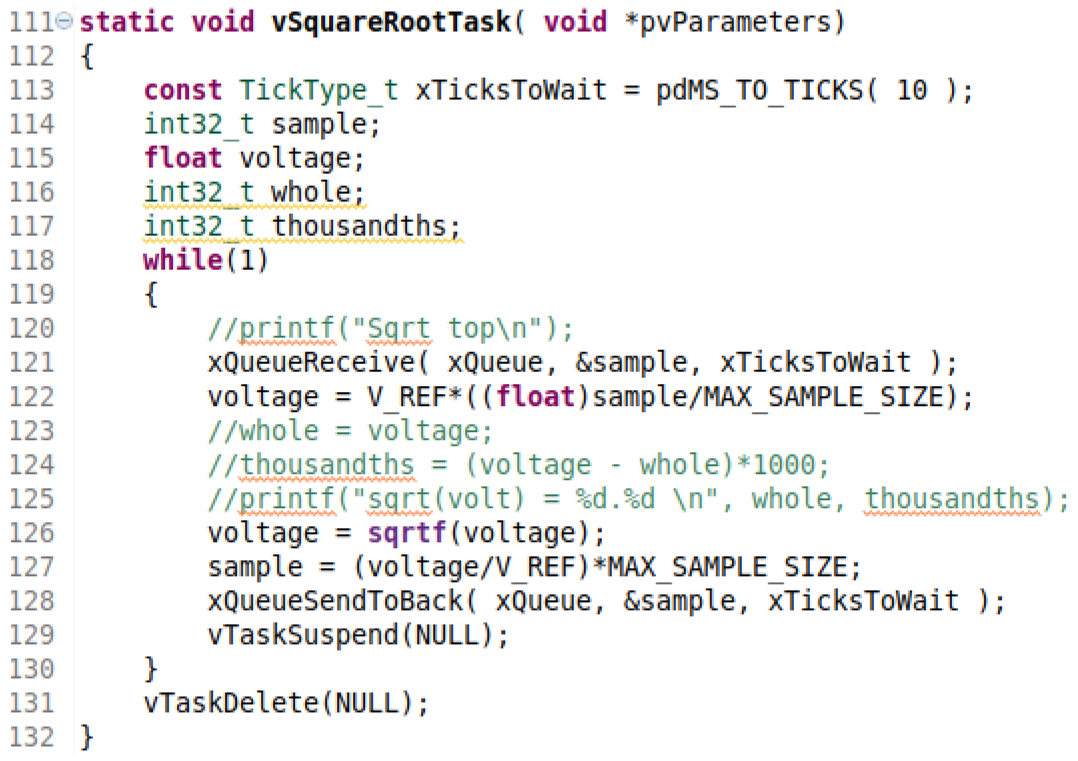


```
void vAD1Task( void *pvParaeters )
{
    int32_t sample;
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
    int adcdav;     //ADC data available

    while(1)
    {

        Xil_Out32(AD1acq,1);            //ADC acquire
        while (adcdav == 0)
            adcdav = Xil_In32(AD1dav);
        sample = Xil_In32(AD1dat1);
        xQueueSendToBack( xQueue, &sample, xTicksToWait );
        vTaskSuspend(NULL);
        vTaskResume(xSquareRootTask);
        Xil_Out32(AD1acq, 0);           //ADC stop acquire

    }
    vTaskDelete(NULL);
}
```

Figure 15: AD1 task in Square Root program

The main difference between this version of the AD1 task and the less naïve version is the use of `vTaskSuspend` instead of `taskYIELD()`. Once the task writes the sample to the queue, it suspends itself. At this point, the Square Root task starts running, since it has the next highest priority. As will be seen shortly, the DA2 task resumes this task. Once resumed, this task resumes the Square Root task, then loops as in the less naïve version of the passthrough software.

Below is a screenshot of the Square Root task:

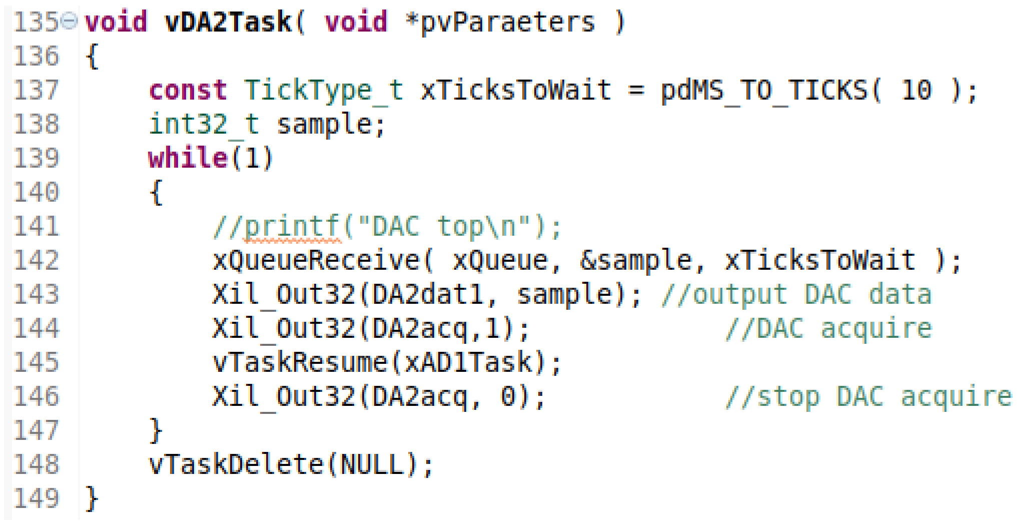


```
static void vSquareRootTask( void *pvParameters)
{
    const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
    int32_t sample;
    float voltage;
    int32_t whole;
    int32_t thousandths;
    while(1)
    {
        //printf("Sqrt top\n");
        xQueueReceive( xQueue, &sample, xTicksToWait );
        voltage = V_REF*((float)sample/MAX_SAMPLE_SIZE);
        //whole = voltage;
        //thousandths = (voltage - whole)*1000;
        //printf("sqrt(volt) = %d.%d \n", whole, thousandths);
        voltage = sqrtf(voltage);
        sample = (voltage/V_REF)*MAX_SAMPLE_SIZE;
        xQueueSendToBack( xQueue, &sample, xTicksToWait );
        vTaskSuspend(NULL);
    }
    vTaskDelete(NULL);
}
```

Figure 16: Code for Square Root task

First, the sample is read from the top of the queue. Next, the sample is converted from a sample number back into a voltage value. This is done by taking the ratio of the sample number to the max sample number and multiplying it by the reference voltage. The sample is cast as a float so that floating-point division is performed. Next, the square root of the voltage is taken using sqrtf. This is the float version of the square root function. After that, the voltage is converted back into a sample by taking the ratio of the voltage to the reference voltage and multiplying it by the max sample number. This new sample is then written to the queue, and the square root task suspends itself. Since the AD1 task is also suspended, the DA2 task gets to run.

Below is a screenshot of the DA2 task:



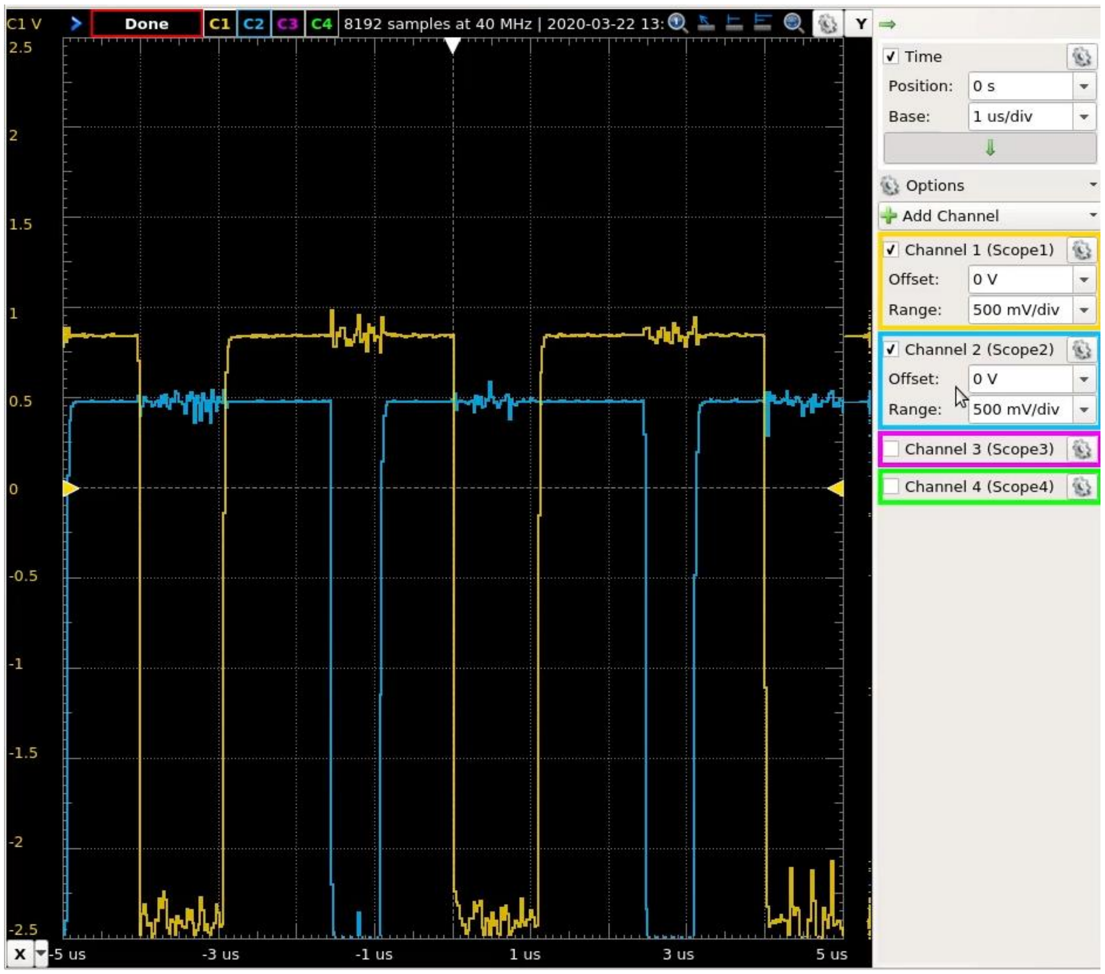
```
135 void vDA2Task( void *pvParaeters )
136 {
137     const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
138     int32_t sample;
139     while(1)
140     {
141         //printf("DAC top\n");
142         xQueueReceive( xQueue, &sample, xTicksToWait );
143         Xil_Out32(DA2dat1, sample); //output DAC data
144         Xil_Out32(DA2acq,1);            //DAC acquire
145         vTaskResume(xAD1Task);
146         Xil_Out32(DA2acq, 0);           //stop DAC acquire
147     }
148     vTaskDelete(NULL);
149 }
```

Figure 17: DA2 task in Square Root program

The main difference between this task and the less naïve version of this task is the use of `vTaskResume` instead of `taskYIELD()`. Since the DA2 task has a priority level of 1, it is preempted by the AD1 task as soon as it is resumed. The DA2 tasks picks-up on line 146 the next time the square root task suspends itself. It then loops as is the less naïve version of this task.

An analysis of the throughput of this program as well as a screenshot of the passed-through sine wave can be found in the Results section of this report.

<u>Results</u>
Below is a screenshot of the chip select signals from Task 1: Standalone Passthrough software:



Time
Position: 0 s
Base: 1 us/div

Channel 1 (Scope1)
Offset: 0 V
Range: 500 mV/div

Channel 2 (Scope2)
Offset: 0 V
Range: 500 mV/div

Figure 18: Chip select signals from AD1Pmod and DA2Pmod from standalone code

The yellow line is the CS signal from AD1Pmod, and the blue line is the CS signal from DA2Pmod. As can be seen from the oscilloscope, the period of the AD CS signal is about 4µs/sample. This implies the throughput in samples/sec is 250ksamples/s. The time spent reading data in from the ADC is about 1µs while the time spend writing data to the DAC is about 0.8µs. This means the overhead is 2.2µs, which is 55% of the total period.

Below is a screenshot of the waveforms when running the standalone software:

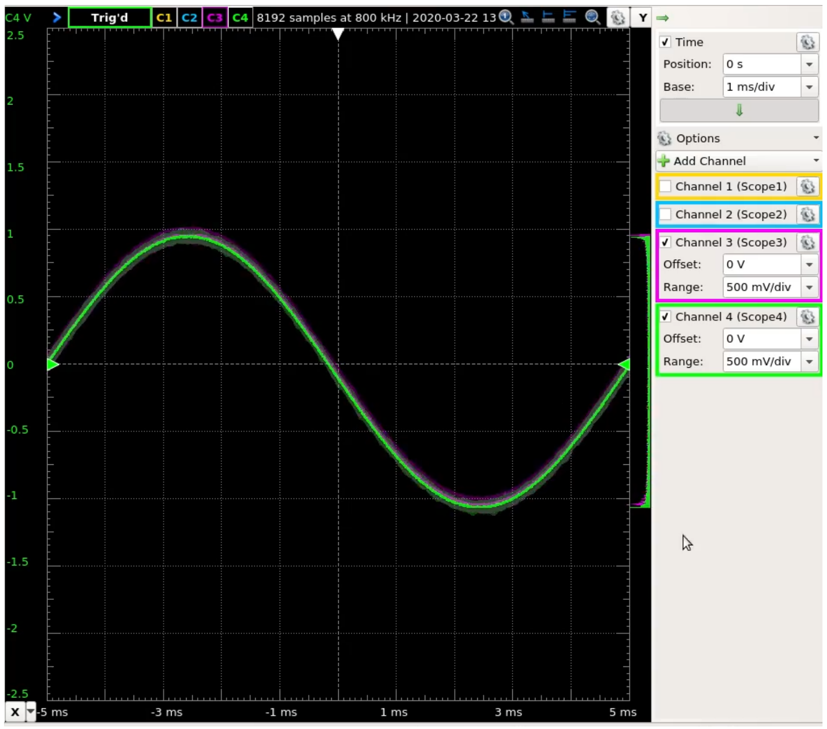


Figure 19: Standalone sine wave results

The magenta line represents the data written by the DAC, while the green line is the output of the waveform generator. Note that they overlap. This indicates that the passthrough function is working as expected.

Below is a screenshot of the chip select signals from Task 2: FreeRTOS Implementation of Passthrough – a Naïve Approach:

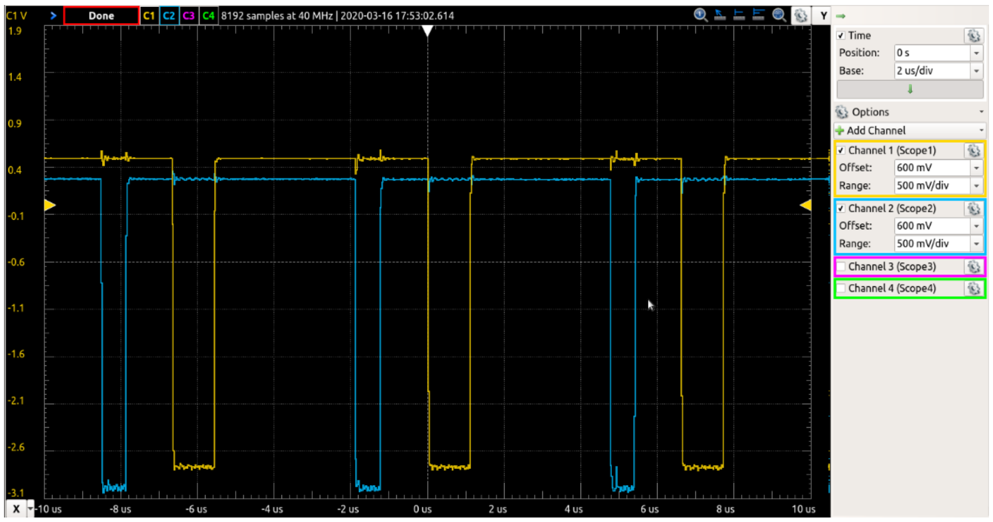
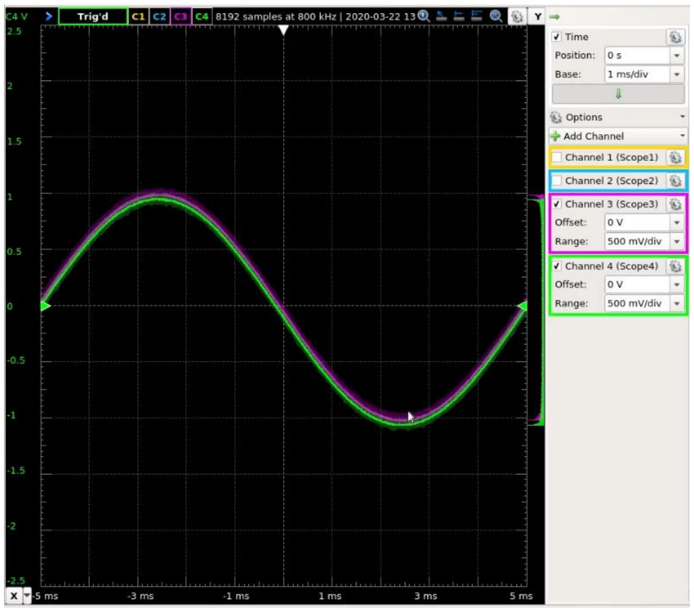


Figure 20: Chip select signals from AD1Pmod and DA2Pmod from naïve FreeRTOS passthrough

The yellow line is the CS signal from AD1Pmod, and the blue line is the CS signal from DA2Pmod. As can be seen from the oscilloscope, the period of the AD CS signal is about 6.7µs/sample. This implies the throughput in samples/sec is 189ksamples/s. The time spent reading data in from the ADC is about 1µs while the time spend writing data to the DAC is about 0.8µs. This means the overhead is 4.9µs, which is 73% of the total period.

Using FreeRTOS is this naïve way produced more overhead. The time for task management was added to the time waiting for various signals to go high or low.

Below is a screenshot of the waveforms when running the naïve FreeRTOS passthrough software:

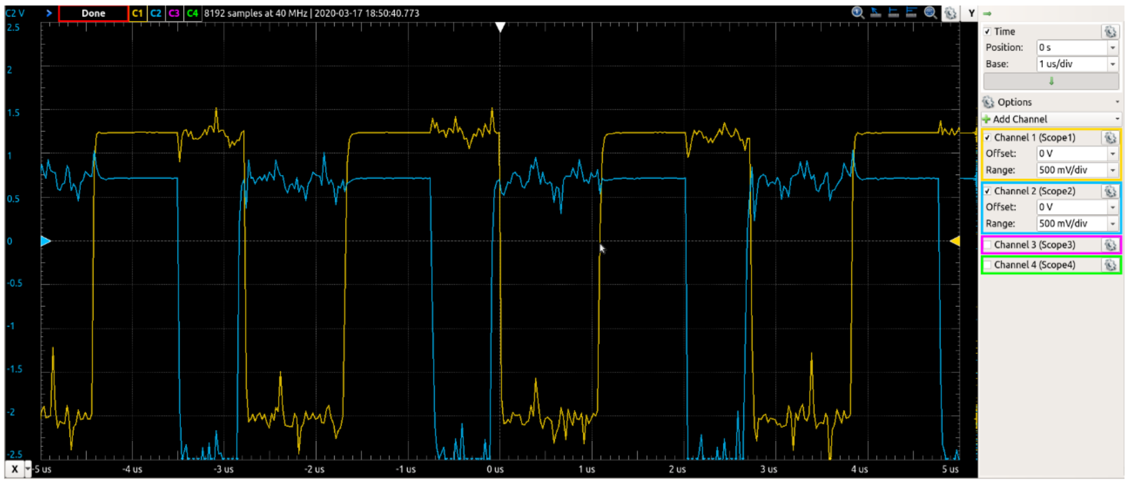


Time
Position: 0 s
Base: 1 ms/div
Channel 3 (Scope3)
Offset: 0 V
Range: 500 mV/div
Channel 4 (Scope4)
Offset: 0 V
Range: 500 mV/div

Figure 21: FreeRTOS Naïve Passthrough wave form

The magenta line represents the data written by the DAC, while the green line is the output of the waveform generator. Note that they overlap. This indicates that the passthrough function is working as expected.

Below is a screenshot of the chip select signals from Task 2: FreeRTOS Implementation of Passthrough – a Less Naïve Approach:



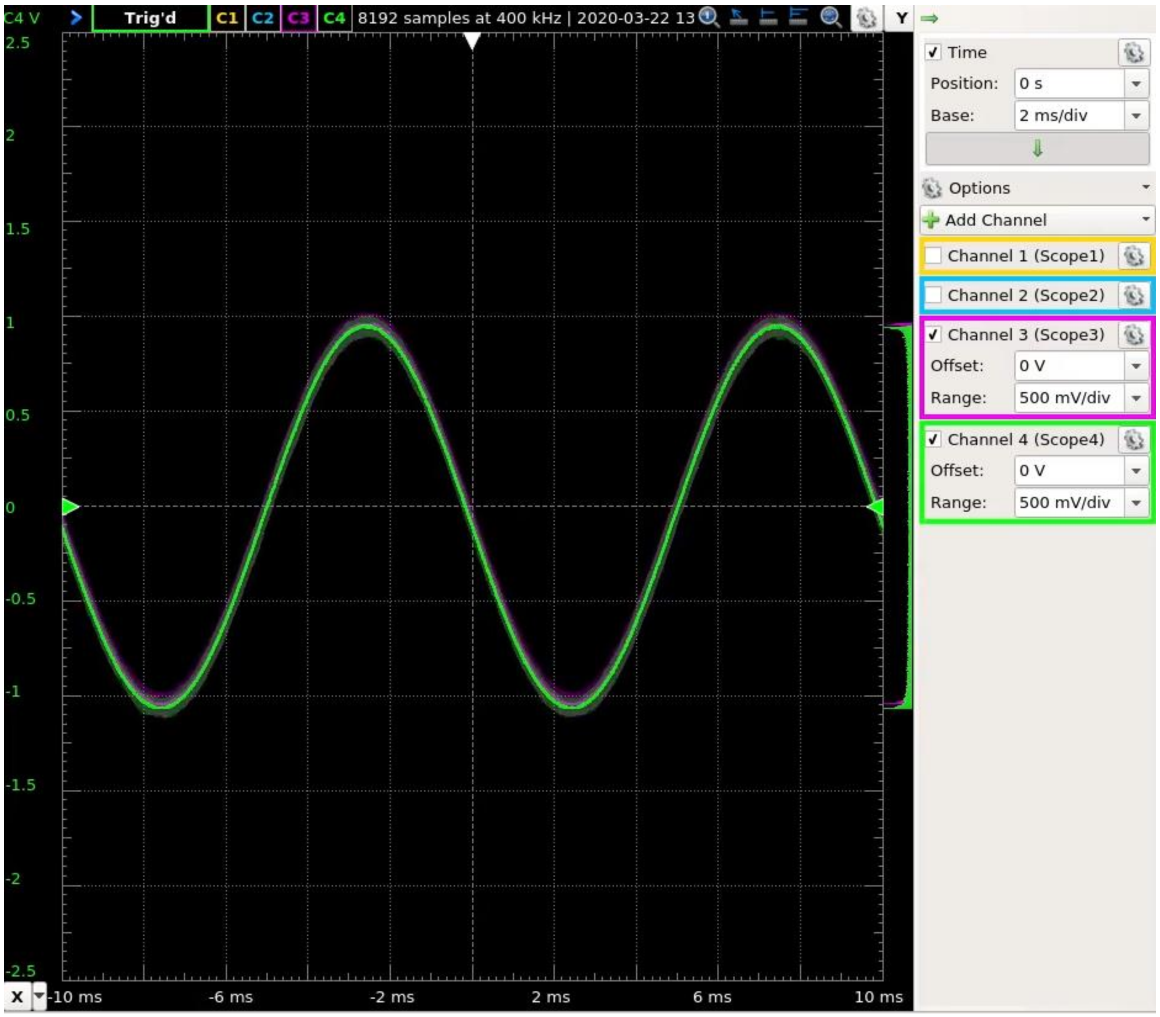
Time
Position: 0 s
Base: 1 us/div
Channel 1 (Scope1)
Offset: 0 V
Range: 500 mV/div
Channel 2 (Scope2)
Offset: 0 V
Range: 500 mV/div

Figure 22: Chip select signals from AD1Pmod and DA2Pmod from less naïve FreeRTOS passthrough

The yellow line is the CS signal from AD1Pmod, and the blue line is the CS signal from DA2Pmod. As can be seen from the oscilloscope, the period of the AD CS signal is about 2.8μs/sample. This implies the throughput in samples/sec is 357ksamples/s. The time spent reading data in from the ADC is about 1μs while the time spend writing data to the DAC is about 0.8μs. This means the overhead is 4.9μs, which is 36% of the total period.

By cutting-out some of the waiting time, the throughput of this code was 43% greater than the standalone version and 89% than the naïve FreeRTOS version.

Below is a screenshot of the waveforms when running the naïve FreeRTOS passthrough software:



✓ Time
Position: 0 s
Base: 2 ms/div

Channel 1 (Scope1)
Channel 2 (Scope2)
✓ Channel 3 (Scope3)
Offset: 0 V
Range: 500 mV/div
✓ Channel 4 (Scope4)
Offset: 0 V
Range: 500 mV/div

Figure 23: FreeRTOS Less Naïve Passthrough wave form

The magenta line represents the data written by the DAC, while the green line is the output of the waveform generator. Note that they overlap. This indicates that the passthrough function is working as expected.

Below is a screenshot of the chip select signals from Task 3: Square Root Task:

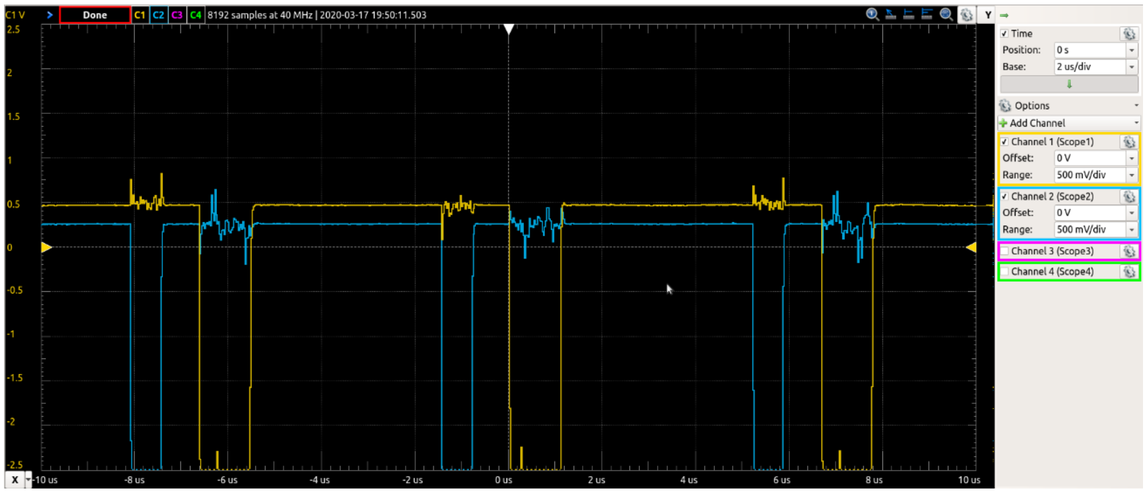


Figure 24: Chip select signals from AD1Pmod and DA2Pmod from FreeRTOS Square Root

The yellow line is the CS signal from AD1Pmod, and the blue line is the CS signal from DA2Pmod. As can be seen from the oscilloscope, the period of the AD CS signal is about 6.7µs/sample. This implies the throughput in samples/sec is 149ksamples/s. The time spent reading data in from the ADC is about 1µs while the time spend writing data to the DAC is about 0.8µs. This means the overhead is 4.9µs, which is 73% of the total period.

Though the throughput for this program was the least of all the programs run, that is to be expected; this program was the only one doing any kind of processing. Something interesting to note though is that even though its throughput was less, its overhead was still better than the naïve version of the FreeRTOS passthrough program.

Below is a screenshot of the waveforms when running the square root FreeRTOS software:

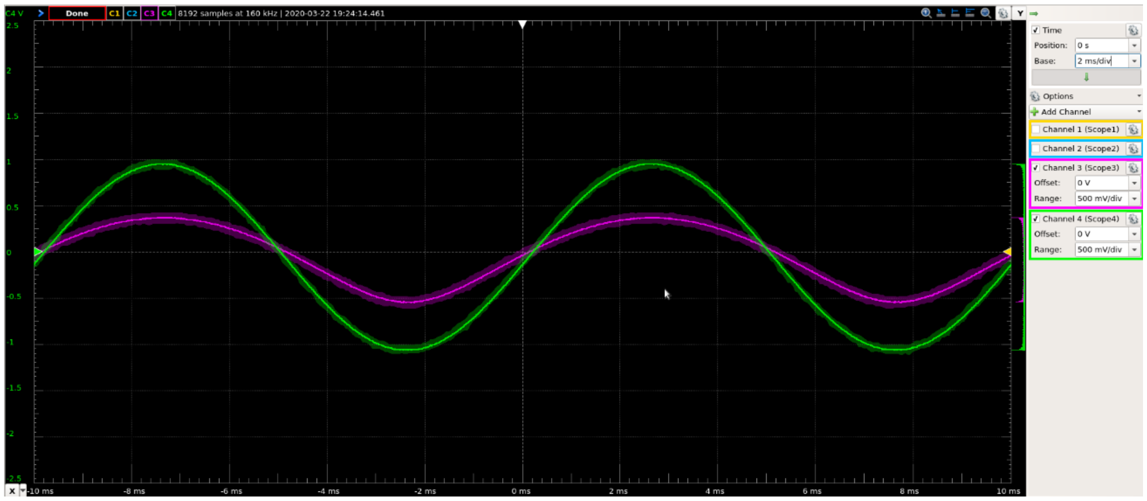


Figure 25: FreeRTOS Sqaure Root wave form

Since the input was a sine wave with an amplitude of one and a DC offset of 1.5V, the output wave is always some positive number greater than zero. The output of the oscilloscope is similar to to the graphs of $v(t) = (1)sint + 1.5$ and $v_{\sqrt{}}(t) = \sqrt{v(t)}$.

Video Link
A video walking through the hardware setup as well as demonstrating each piece of software and discussing the throughput results can be found here:

https://youtu.be/ubPU-v4lx5U

Conclusions

Overall, this lab was a success. The known-how of how to use AD1Pmod and DA2Pmod was successfully conveyed. Furthermore, the importance of multitasking in DSP was shown by demonstrating that throughput could be increased by clever use of a multitasking operating system. Finally, the concept of FreeRTOS queues was successfully introduced and used in digital signal processing.

Appendix

Code Code – Standalone

```
/* Xilinx includes. */
#include "xparameters.h"
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"

#define printf xil_printf                                   // smaller, optimized printf

//AD1Pmod from Address Editor in Vivado, first IP
#define AD1acq          0x43C00000      //AD1 Acquisition         - output
#define AD1dav          0x43C00004      //AD1 data available - input
#define AD1dat1                 0x43C00008      //AD1 channel 1 data - input
#define AD1dat2                 0x43C0000C      //AD1 channel 2 data - input

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq          0x43C10000      //DA2 acquisition         - output
#define DA2dav          0x43C10004      //DA2 data available - input
#define DA2dat1                 0x43C10008      //DA2 channel 1 data - output
#define DA2dat2                 0x43C1000C      //DA2 channel 2 data - output

int main(void)
{
        int adcdav;                     //ADC data available
        int adcdata1;   //ADC channel 1 data
        int adcdata2;   //ADC channel 2 data

        int dacdata1;   //DAC channel 1 data
        int dacdata2;   //DAC channel 2 data
        int dacdav;                     //DAC data available

        printf("\n\rStarting AD1-DA2 Pmod demo test...\n");
        Xil_Out32(AD1acq,0);                    //ADC stop acquire
        adcdav = Xil_In32(AD1dav);              //ADC available?
        while (adcdav == 1)
                adcdav = Xil_In32(AD1dav);
        Xil_Out32(DA2acq, 0);                   //DAC stop acquire
        dacdav = Xil_In32(DA2dav);              //DAC available?
```

```
	while (dacdav == 1)
		dacdav = Xil_In32(DA2dav);

	while (1)
	{
		//ADC
		Xil_Out32(AD1acq,1);			//ADC acquire
		while (adcdav == 0)				//ADC data available?
			adcdav = Xil_In32(AD1dav);
		Xil_Out32(AD1acq, 0);				//ADC stop acquire
		adcdata1 = Xil_In32(AD1dat1);
		//adcdata2 = Xil_In32(AD1dat2);
		while (adcdav == 1)				// wait for reset
			adcdav = Xil_In32(AD1dav);

		dacdata1 = adcdata1;			//ADC -> DAV pass through
		//dacdata2 = adcdata2;			//ADC -> DAV pass through

		//DAC
		Xil_Out32(DA2dat1, dacdata1);	//output DAC data
		//Xil_Out32(DA2dat2, dacdata2);
		Xil_Out32(DA2acq, 1);				//DAC acquire
		while (dacdav == 0)				//DAC data output?
			dacdav = Xil_In32(DA2dav);
		Xil_Out32(DA2acq,0);
		while (dacdav == 1)
			dacdav = Xil_In32(DA2dav);
	}
}
```

C Code – FreeRTOS Naïve Passthrough

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Xilinx includes. */
#include "xparameters.h"
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"

#define printf xil_printf					// smaller, optimized printf

// Function prototypes
static void vAD1Task( void *pvParameters );	// FreeRTOS Task
static void vDA2Task( void *pvParameters );	// FreeRTOS Task
static void vADDAInit();						// Initialize ADC/DAC

// FreeRTOS Task Handles
static TaskHandle_t xAD1Task;
static TaskHandle_t xDA2Task;

// FreeRTOS Queue
QueueHandle_t xQueue;
```

```
#define MAX_Q 1

//AD1Pmod from Address Editor in Vivado, first IP
#define AD1acq          0x43C00000      //AD1 Acquisition         - output
#define AD1dav          0x43C00004      //AD1 data available - input
#define AD1dat1                 0x43C00008      //AD1 channel 1 data - input
#define AD1dat2                 0x43C0000C      //AD1 channel 2 data - input

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq          0x43C10000      //DA2 acquisition         - output
#define DA2dav          0x43C10004      //DA2 data available - input
#define DA2dat1                 0x43C10008      //DA2 channel 1 data - output
#define DA2dat2                 0x43C1000C      //DA2 channel 2 data - output

int main (void)
{
        printf("\n\rStarting AD1-DA2 Pmod FreeRTOS Pass-through...\n");

        // Initialize ADC and DAC
        vADDAInit();

        xQueue = xQueueCreate( MAX_Q, sizeof( int32_t ) );

        if (xQueue != NULL)
        {
                // Create ADC task
                xTaskCreate(vAD1Task, "ADC", 1000, NULL, 1, &xAD1Task);
                // Create DAC task
                xTaskCreate(vDA2Task, "DAC", 1000, NULL, 1, &xDA2Task);
                //Start Scheduler
                vTaskStartScheduler();
        }
        else
        {
                printf("Failed to create queue!\n");
        }

        while (1);

}

void vADDAInit()
{
        int adcdav;                     //ADC data available
        int dacdav;                     //DAC data available

        Xil_Out32(AD1acq,0);                    //ADC stop acquire
        adcdav = Xil_In32(AD1dav);      //ADC available?
        while (adcdav == 1)
                adcdav = Xil_In32(AD1dav);
        Xil_Out32(DA2acq, 0);                   //DAC stop acquire
        dacdav = Xil_In32(DA2dav);      //DAC available?
        while (dacdav == 1)
                dacdav = Xil_In32(DA2dav);
}
```

```
void vAD1Task( void *pvParaeters )
{
	int32_t sample;
	const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
	int adcdav;			//ADC data available

	while(1)
	{

		//ADC
		Xil_Out32(AD1acq,1);			//ADC acquire
		while (adcdav == 0)				//ADC data available?
			adcdav = Xil_In32(AD1dav);
		Xil_Out32(AD1acq, 0);				//ADC stop acquire
		sample = Xil_In32(AD1dat1);
		//adcdata2 = Xil_In32(AD1dat2);
		while (adcdav == 1)				// wait for reset
			adcdav = Xil_In32(AD1dav);
		xQueueSendToBack( xQueue, &sample, xTicksToWait );
		taskYIELD();
	}
	vTaskDelete(NULL);
}

void vDA2Task( void *pvParaeters )
{
	const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
	int32_t sample;
	int dacdav;			//DAC data available
	while(1)
	{
		//printf("DAC top\n");
		xQueueReceive( xQueue, &sample, xTicksToWait );
		Xil_Out32(DA2dat1, sample);	//output DAC data
		//Xil_Out32(DA2dat2, dacdata2);
		Xil_Out32(DA2acq, 1);				//DAC acquire
		while (dacdav == 0)					//DAC data output?
			dacdav = Xil_In32(DA2dav);
		Xil_Out32(DA2acq,0);
		while (dacdav == 1)
			dacdav = Xil_In32(DA2dav);
		taskYIELD();
	}
	vTaskDelete(NULL);
}
```

C Code – FreeRTOS Less Naïve Passthrough

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Xilinx includes. */
#include "xparameters.h"
```

```
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"

#define printf xil_printf                                   // smaller, optimized printf

// Function prototypes
static void vAD1Task( void *pvParameters );     // FreeRTOS Task
static void vDA2Task( void *pvParameters );     // FreeRTOS Task
static void vADDAInit();                                                // Initializ ADC/DAC
// FreeRTOS Task Handles
static TaskHandle_t xAD1Task;
static TaskHandle_t xDA2Task;

// FreeRTOS Queue
QueueHandle_t xQueue;
#define MAX_Q 1

//AD1Pmod from Address Editor in Vivado, first IP
#define AD1acq          0x43C00000      //AD1 Acquisition           - output
#define AD1dav          0x43C00004      //AD1 data available - input
#define AD1dat1                 0x43C00008      //AD1 channel 1 data - input
#define AD1dat2                 0x43C0000C      //AD1 channel 2 data - input

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq          0x43C10000      //DA2 acquisition           - output
#define DA2dav          0x43C10004      //DA2 data available - input
#define DA2dat1                 0x43C10008      //DA2 channel 1 data - output
#define DA2dat2                 0x43C1000C      //DA2 channel 2 data - output

int main (void)
{
        printf("\n\rStarting AD1-DA2 Pmod FreeRTOS Pass-through...\n");

        // Initialize ADC and DAC
        vADDAInit();

        xQueue = xQueueCreate( MAX_Q, sizeof( int32_t ) );

        if (xQueue != NULL)
        {
                // Create ADC task
                xTaskCreate(vAD1Task, "ADC", 1000, NULL, 1, &xAD1Task);
                // Create DAC task
                xTaskCreate(vDA2Task, "DAC", 1000, NULL, 1, &xDA2Task);
                //Start Scheduler
                vTaskStartScheduler();
        }
        else
        {
                printf("Failed to create queue!\n");
        }

        while (1);
}

void vADDAInit()
{
        int adcdav;                 //ADC data available
        int dacdav;                 //DAC data available

        Xil_Out32(AD1acq,0);                //ADC stop acquire
        adcdav = Xil_In32(AD1dav);          //ADC available?
        while (adcdav == 1)
                adcdav = Xil_In32(AD1dav);
        Xil_Out32(DA2acq, 0);               //DAC stop acquire
        dacdav = Xil_In32(DA2dav);          //DAC available?
        while (dacdav == 1)
                dacdav = Xil_In32(DA2dav);
}

void vAD1Task( void *pvParaeters )
{
        int32_t sample;
        const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
        int adcdav;                 //ADC data available

        while(1)
        {

                Xil_Out32(AD1acq,1);                        //ADC acquire
                while (adcdav == 0)
                        adcdav = Xil_In32(AD1dav);
                sample = Xil_In32(AD1dat1);
                xQueueSendToBack( xQueue, &sample, xTicksToWait );
                taskYIELD();
                Xil_Out32(AD1acq, 0);                       //ADC stop acquire
        }
        vTaskDelete(NULL);
}

void vDA2Task( void *pvParaeters )
{
        const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
        int32_t sample;

        while(1)
        {
                //printf("DAC top\n");
                xQueueReceive( xQueue, &sample, xTicksToWait );
                Xil_Out32(DA2dat1, sample);         //output DAC data
                Xil_Out32(DA2acq,1);                        //DAC acquire
                taskYIELD();
                Xil_Out32(DA2acq, 0);                       //stop DAC acquire
        }
        vTaskDelete(NULL);
}
```

C Code – FreeRTOS Square Root

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Xilinx includes. */
#include "xparameters.h"
#include "xstatus.h"
#include "xil_io.h"
#include "xil_printf.h"
#define printf xil_printf                                   // smaller, optimized printf

#include <math.h>

// Function prototypes
static void vAD1Task( void *pvParameters );       // FreeRTOS Task
static void vDA2Task( void *pvParameters );       // FreeRTOS Task
static void vSquareRootTask( void *pvParameters);
static void vADDAInit();                                          // Initialize ADC/DAC

// FreeRTOS Task Handles
static TaskHandle_t xAD1Task;
static TaskHandle_t xDA2Task;
static TaskHandle_t xSquareRootTask;

// FreeRTOS Queue
QueueHandle_t xQueue;
#define MAX_Q 1

//AD1Pmod from Address Editor in Vivado, first IP
#define AD1acq          0x43C00000    //AD1 Acquisition          - output
#define AD1dav          0x43C00004    //AD1 data available - input
#define AD1dat1                 0x43C00008    //AD1 channel 1 data - input
#define AD1dat2                 0x43C0000C    //AD1 channel 2 data - input

//DAC2Pmod from Address Editor in Vivado, second IP
#define DA2acq          0x43C10000    //DA2 acquisition          - output
#define DA2dav          0x43C10004    //DA2 data available - input
#define DA2dat1                 0x43C10008    //DA2 channel 1 data - output
#define DA2dat2                 0x43C1000C    //DA2 channel 2 data - output

#define V_REF 3.3
#define MAX_SAMPLE_SIZE         4096

int main (void)
{
        printf("\n\rStarting AD1-DA2 Pmod FreeRTOS Pass-through...\n");

        // Initialize ADC and DAC
        vADDAInit();

        xQueue = xQueueCreate( MAX_Q, sizeof( int32_t ) );
        if (xQueue != NULL)
        {
```

```
			// Create ADC task
			xTaskCreate(vAD1Task, "ADC", 1000, NULL, 3, &xAD1Task);
			// Create Square Root task
			xTaskCreate(vSquareRootTask, "SQRT", 1000, NULL, 2, &xSquareRootTask);
			// Create DAC task
			xTaskCreate(vDA2Task, "DAC", 1000, NULL, 1, &xDA2Task);
			//Start Scheduler
			vTaskStartScheduler();
		}
		else
		{
			printf("Failed to create xQueue!\n");
			while (1);
		}

		while (1);

}

void vADDAInit()
{
		int adcdav;			//ADC data available
		int dacdav;			//DAC data available

		Xil_Out32(AD1acq,0);		//ADC stop acquire
		adcdav = Xil_In32(AD1dav);	//ADC available?
		while (adcdav == 1)
			adcdav = Xil_In32(AD1dav);
		Xil_Out32(DA2acq, 0);		//DAC stop acquire
		dacdav = Xil_In32(DA2dav);	//DAC available?
		while (dacdav == 1)
			dacdav = Xil_In32(DA2dav);
}

void vAD1Task( void *pvParaeters )
{
		int32_t sample;
		const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
		int adcdav;			//ADC data available

		while(1)
		{

			Xil_Out32(AD1acq,1);			//ADC acquire
			while (adcdav == 0)
				adcdav = Xil_In32(AD1dav);
			sample = Xil_In32(AD1dat1);
			xQueueSendToBack( xQueue, &sample, xTicksToWait );
			vTaskSuspend(NULL);
			vTaskResume(xSquareRootTask);
			Xil_Out32(AD1acq, 0);			//ADC stop acquire

		}
		vTaskDelete(NULL);
}
```

```
static void vSquareRootTask( void *pvParameters)
{
	const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
	int32_t sample;
	float voltage;
	int32_t whole;
	int32_t thousandths;
	while(1)
	{
		//printf("Sqrt top\n");
		xQueueReceive( xQueue, &sample, xTicksToWait );
		voltage = V_REF*((float)sample/MAX_SAMPLE_SIZE);
		//whole = voltage;
		//thousandths = (voltage - whole)*1000;
		//printf("sqrt(volt) = %d.%d \n", whole, thousandths);
		voltage = sqrtf(voltage);
		sample = (voltage/V_REF)*MAX_SAMPLE_SIZE;
		xQueueSendToBack( xQueue, &sample, xTicksToWait );
		vTaskSuspend(NULL);
	}
	vTaskDelete(NULL);
}


void vDA2Task( void *pvParaeters )
{
	const TickType_t xTicksToWait = pdMS_TO_TICKS( 10 );
	int32_t sample;
	while(1)
	{
		//printf("DAC top\n");
		xQueueReceive( xQueue, &sample, xTicksToWait );
		Xil_Out32(DA2dat1, sample);	//output DAC data
		Xil_Out32(DA2acq,1);			//DAC acquire
		vTaskResume(xAD1Task);
		Xil_Out32(DA2acq, 0);			//stop DAC acquire
	}
	vTaskDelete(NULL);
}
```