**Software Timer in FreeRTOS**

Benjamin A. Blouin

*benjamin.blouin@temple.edu*

**Summary**

This lab is a rehash of the previous two labs using the ADC and DAC but doing the both using FreeRTOS. The example given in the lab includes a C-style program to be run as-is, which shows the functionality of the ADC and DAC data transfer. Then this program is implemented in FreeRTOS using a task for each component, passing the signal through. The last lab requirement implements a filtering task, the same filter designed from a previous lab. The ADC takes in a generated signal, pass the data through a queue to the filtering task, which then passes the filtered data to the DAC for output to an oscilloscope. The Digilent EEBoard is used as a waveform-generator and oscilloscope.

**Introduction**

The DAC and ADC have been explored in previous labs, as well as using FreeRTOS tasks for behavior management. This lab combines these previous lessons. The block-design is straightforward, but a constraint file must be included to map IO-pin address to signal names. The IPs are custom and need to be connected manually to each other and to external virtual ports.

The given source code can be run out of the box in the SDK, with the caveat that the Pmods and EEBoard need to be wired up correctly, as seen in figure 8. Setting the waveform generator to a unipolar sinewave leads to a sinewave being shown on the oscilloscope. The same also happens for the FreeRTOS passthrough lab task. The third task has the same workflow, except the resultant sinewave will be filtered.

**Discussion**

Like all previous labs a block diagram is made, as shown in figure 1. The given constraints file and IP blocks must be imported, which allows for a correct block design functioning as desired. Getting to creating a bitstream is the ultimate goal, which can then be exported for use by the SDK.

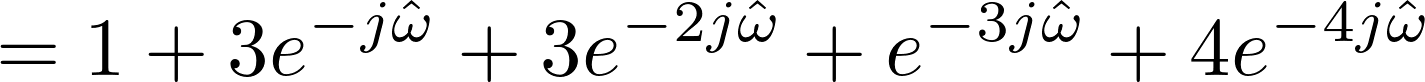
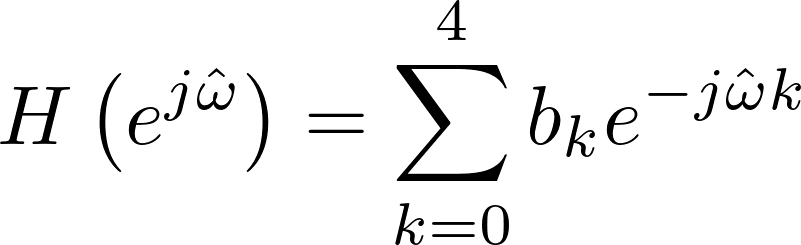
The bare-metal code given is a passthrough between ADC to DAC. No modifications are needed, and the resultant oscilloscope reading is seen in figure 2.

Recreating the bare-metal in FreeRTOS requires two tasks, one for the ADC and the other for the DAC. There is one queue, which is used to pass data. This is the basic template needed. It is not unlike the previous task-management lab in priority between tasks, and the queue is one word deep, containing an int32\_t variable that represents the data taken in from the ADC and to be passed to the DAC.

Taking this idea, a step further another task is added that takes care of filtering the signal. Only one queue is needed as long as prioritization and queue passing are done deliberately. Since the filter task is the glue between the input and output it is given the highest priority. When it is done with the receive, filter, and send, it resumes the DAC, and self-suspends. Because the DAC has the next highest priority it proceeds by taking the data off the queue and outputting it to the oscilloscope through the DAC. When the DAC is done, the next highest priority task goes, which is the ADC. The ADC dataflow is mostly the same as the DAC, except it resumes the filter task. At this point the tasks are in a controlled loop.

The filter is the same as previously devised but is more deliberate in knowing what state the filter buffer is in; this allows for the initial unfilled state to be taken care of, and then the pushdown list can be implemented. The difference in filtering characteristics due to FreeRTOS task overhead can be seen in the difference in sampling frequency in figures 5, 6, and 7.

The FreeRTOS tasking and queueing can create or alleviate overhead for sampling frequency. The previous filter lab had a sampling frequency provided so the class had a consistent basis, but this lab differs results from the different implementations of FreeRTOS and filtering algorithms that can be create by a student. Measuring the active-low frequency change on the AD1 CS-pin can give the absolute sampling frequency, as seen in figures 6 and 7. Using the same calculation algorithms from the previous filter lab, the frequency response can be shown as:



At n=1, the magnitude of H≈ 3.81, but Euler’s formula in reference to the sine wave construction is half, so we will say |H|≈7.6; At n=0 the filter’s response is hard to characterize since it has four taps, needing four previous samples. The figure shows a magnitude of H≈7.6, but this is hard to show in real life. The filtered function had to be scaled by multiplying by ≈ 1/7.6, or 0.083. This scaled the filtered signal back down to the original signal.

**Conclusions**

This lab is the culmination of previous task management and signal modification designs. It was awesome to see the controlling the management through queue and task managements. Being able to see how code and OS choice affect filter design will make me keep this in mind when designing filters in the future.

**Appendices**

**Github**

<https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/blob/master/AD1Pmod%20and%20DA2Pmod%20in%20FreeRTOS/AD1Pmod_and_DA2Pmod_in_FreeRTOS.sdk/lab8_FreeRTOS/src/main.c>

**Pictures**

**A screenshot of a cell phone

Description automatically generated**

Figure Block Design

**A picture containing electronics

Description automatically generated**

Figure C-program Passthrough

**A picture containing electronics

Description automatically generated**

Figure FreeRTOS Passthrough

**A screenshot of a computer

Description automatically generated**

Figure FreeRTOS Filtered Sine wave

**A screenshot of a computer

Description automatically generated**

Figure C-Program CS-pin Measurements

**A screenshot of a computer

Description automatically generated**

Figure FreeRTOS CS-pin Measurement

**A screenshot of a computer

Description automatically generated**

Figure FreeRTOS CS-pin Second Measurement

**A close up of a device

Description automatically generated**

Figure Pmod-EEboard Wiring

**A close up of a map

Description automatically generated**

Figure FreeRTOS Bode Plot

**A close up of a map

Description automatically generated**

Figure FreeRTOS Pole-Zero Map**A close up of a map

Description automatically generated**

Figure Bare-Metal Pole-Zero Map

**A close up of a map

Description automatically generated**

Figure Bare-Metal Bode Plot

A close up of a map

Description automatically generated

Figure 13 Theoretical Simulation

**Code**

//

//

#include <stdio.h>

// FreeRTOS includes

#include <FreeRTOS.h>

#include <task.h>

#include <queue.h>

// Xilinx includes

#include <xparameters.h>

#include <xil\_io.h>

#include <xil\_printf.h>

#include <sleep.h>

#include <stdlib.h>

//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

//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

// tasks prototypes

static void prvAD1task(void\* pvParameters);

static void prvDA2task(void\* pvParameters);

static void prvFiltertask(void\* pvParameters);

// declare tasks handles

static TaskHandle\_t xAD1Task;

static TaskHandle\_t xDA2Task;

static TaskHandle\_t xFilterTask;

// declare queues

static QueueHandle\_t xTransferQueue = NULL;

// ADC and DAC global vars

int32\_t adcdav; //ADC data available

int32\_t dacdav; //DAC data available

int main(int argc, char\*\* argv) {

xTaskCreate(prvAD1task,

"AD1Task",

configMINIMAL\_STACK\_SIZE,

NULL,

tskIDLE\_PRIORITY+1,

&xAD1Task);

xTaskCreate(prvDA2task,

"DA2Task",

configMINIMAL\_STACK\_SIZE,

NULL,

tskIDLE\_PRIORITY+2,

&xDA2Task);

xTaskCreate(prvFiltertask,

"FilterTask",

configMINIMAL\_STACK\_SIZE,

NULL,

tskIDLE\_PRIORITY+3,

&xFilterTask);

// create the Queue that will be used to pass the data

xTransferQueue = xQueueCreate(1,

sizeof(int32\_t));

// make sure that the Queue was created

//

configASSERT(xTransferQueue);

// start the task Scheduler

vTaskStartScheduler();

while(1);

return 0;

}

static void prvAD1task(void\* pvParameters) {

static int32\_t data;

while(1) {

//ADC acquire

//

Xil\_Out32(AD1acq,1);

while (adcdav == 0) {

adcdav=Xil\_In32(AD1dav);

}

//ADC stop acquire

Xil\_Out32(AD1acq, 0);

//input ADC data

data = Xil\_In32(AD1dat1);

//wait for reset

//

while (adcdav==1) {

adcdav = Xil\_In32(AD1dav);

}

// pass the data to the queue

xQueueSend(xTransferQueue,

&data,

0UL);

vTaskResume(xFilterTask);

}

}

static void prvFiltertask(void\* pvParameters){

static int32\_t filter\_data;

static int32\_t filtered\_data;

static int32\_t buffer[4096];

static int8\_t init\_flag,first\_flag,continue\_flag,n;

while(1){

if(n==4){init\_flag=1;first\_flag=1;}

xQueueReceive(xTransferQueue,

&filter\_data,

portMAX\_DELAY);

if(init\_flag==0){

buffer[n]=filter\_data;

n++;

}

if(first\_flag==1 && n==4){

filtered\_data = buffer[0]+3\*buffer[1]+3\*buffer[2]+buffer[3]+4\*buffer[4];

first\_flag = 0;

continue\_flag = 1;

}

if(continue\_flag==1&&n==4){

buffer[4]=buffer[3];

buffer[3]=buffer[2];

buffer[2]=buffer[1];

buffer[1]=buffer[0];

buffer[0]=filter\_data;

filtered\_data = 0.2621\*(buffer[0]+3\*buffer[1]+3\*buffer[2]+buffer[3]+4\*buffer[4]);

}

xQueueSend(xTransferQueue,

&filtered\_data,

0UL);

vTaskResume(xDA2Task);

vTaskSuspend(NULL);

}

}

static void prvDA2task(void\* pvParameters) {

static int32\_t incoming\_data;

while(1) {

// get the data from the Queue

xQueueReceive(xTransferQueue,

&incoming\_data,

portMAX\_DELAY);

//output DAC data

//xil\_printf("data %u\n",incoming\_data);

Xil\_Out32(DA2dat1, incoming\_data);

//DAC acquire

Xil\_Out32(DA2acq,1);

//DAC data output?

while (dacdav==0) {

dacdav=Xil\_In32(DA2dav);

}

//stop DAC acquire

Xil\_Out32(DA2acq,0);

//wait for reset

while(dacdav==1) {

dacdav=Xil\_In32(DA2dav);

}

}

}