Below is the source code, followed by the memory map screenshots.

/\*\*

\* Author: Peter A. Dranishnikov

\* Lab #: 6

\* Course: EEL4685C

\* Due date: March 19th, Spring 2019

\*/

// Below DOC is for boilerplate template attribution

// Original Author: David Foster

// Last modified: 3-14-2018

// Purpose: Learn to use C in the uVision program for ARM Cortex series by implementing a simple filter.

//

#include <stm32f2xx.h>

uint32\_t coeff1[] = {1, 1, 1, 1}; // 4-point simple moving window

uint32\_t coeff2[] = {10, 8, 6, 4, 2, 1}; // 6-point weighted

uint32\_t coeff3[] = {1, 2, 4, 2, 1}; // 5-point weighted

uint32\_t x1\_n[] = {4, 6, 8, 8, 24, 17, 32, 34, 33, 32, 40, 4, 40, 44}; // first set of data samples

uint32\_t x2\_n[] = {1000, 1012, 1040, 2000, 2004, 2080, 0, 2092, 2000, 2003, 1999}; // second set of data samples

// form constants for array sizes

uint32\_t SIZEFILTER1 = sizeof(coeff1)/sizeof(uint32\_t);

uint32\_t SIZEFILTER2 = sizeof(coeff2)/sizeof(uint32\_t);

uint32\_t SIZEFILTER3 = sizeof(coeff3)/sizeof(uint32\_t);

const uint32\_t SIZEX1N = sizeof(x1\_n)/sizeof(uint32\_t);

const uint32\_t SIZEX2N = sizeof(x2\_n)/sizeof(uint32\_t);

// create space for answers

uint32\_t x1\_n\_coeff1[SIZEX1N];

// LAB 6 - create arrays for the other 5 combinations for coefficients and input sequences

uint32\_t x1\_n\_coeff2[SIZEX1N];

uint32\_t x1\_n\_coeff3[SIZEX1N];

// ""

uint32\_t x2\_n\_coeff1[SIZEX2N];

uint32\_t x2\_n\_coeff2[SIZEX2N];

uint32\_t x2\_n\_coeff3[SIZEX2N];

// C function prototype - declares the function inputs and output type so that it may be called below.

uint32\_t filter(uint32\_t\*, uint32\_t, uint32\_t\*, uint32\_t, uint32\_t\*);

// C programs MUST contain this function, and this is where execution begins.

int main(void)

{

volatile uint32\_t errorcode;

errorcode = filter(coeff1, SIZEFILTER1, x1\_n, SIZEX1N, x1\_n\_coeff1);

// LAB 6 - add calls for the other 5 combinations.

errorcode = filter(coeff2, SIZEFILTER2, x1\_n, SIZEX1N, x1\_n\_coeff2);

errorcode = filter(coeff3, SIZEFILTER3, x1\_n, SIZEX1N, x1\_n\_coeff3);

errorcode = filter(coeff1, SIZEFILTER1, x2\_n, SIZEX2N, x2\_n\_coeff1);

errorcode = filter(coeff2, SIZEFILTER2, x2\_n, SIZEX2N, x2\_n\_coeff2);

errorcode = filter(coeff3, SIZEFILTER3, x2\_n, SIZEX2N, x2\_n\_coeff3);

while(1){} // endless loop to keep micro from crashing

}

// Implement an LTI difference equation (FIR filter) in which each output[n] is a weighted average of the coeff[i]\*samples[n-i] values

// roughly: (coeff[0]\*samples[n] + coeff[1]\*samples[n-1]+...+coeff[M]\*samples[n-M]) / sum(coeff[i]'s)

// inputs: coeff is the array of constant coefficients.

// Note: If there is not yet sufficient input data for the filter,

// samples[] should be 0. For example, if the filter needs samples[1], samples[0], and samples[-1] for calculating output[1] for a 3-point filter, then 0 should be used for samples[-1].

// numCoeffs is the number of constant coefficients

// samples is the array of data with samples[0] being the first sample (oldest)

// numSamples is the number of data samples and the number of output values.

// output is the array to store the filtered values to, with the same number of values as samples.

uint32\_t filter(uint32\_t\* coeff, uint32\_t numCoeffs, uint32\_t\* samples, uint32\_t numSamples, uint32\_t\* output)

{

uint32\_t count;

uint32\_t window\_sum = 0; //not a hard coded value

uint32\_t coeff\_sum = 0;

//LAB 6 - add code to solve the difference equation passed to the function. Use the passed lengths and do not hard-code values.

for(uint8\_t i = 0; i < numCoeffs; i++)

{

coeff\_sum += coeff[i];

}

//The differing weights in a weighted average prevent significant improvement from a nested loop

for (count = 0; count < numSamples; count++)

{

window\_sum = 0;

for (uint32\_t i = numCoeffs; i > 0; i--)

{

uint32\_t index\_sam = count - numCoeffs + i;

if(index\_sam <= count)

{

/\*\*

91 92 93 94 95 96 97 98 99 100 101 (sample stream)

6 5 4 3 2 1 (window)

\*/

window\_sum += coeff[numCoeffs - i] \* samples[index\_sam];

}

else

break; //terminate inner loop early if window is expended

/\*\*

0 1 2 3 4 5 6 7 (sample stream)

6 5 4 3 2 1 (window)

\*/

}

output[count] = window\_sum / coeff\_sum; //TODO determine if integer division rounding needs to be handled

//output[count] = (count<<24)+(count<<16)+(count<<8)+count+1; // 32-bit dummy output to help find array in the memory window.

}

return 0; // no specific error codes to return yet, so 0 will indicate success (really that errors is false)

}



