EEC 134A/B Application Tutorial Note
On Board Signal Processing
Team: DiodeHard3
Vinay Vidyasagar

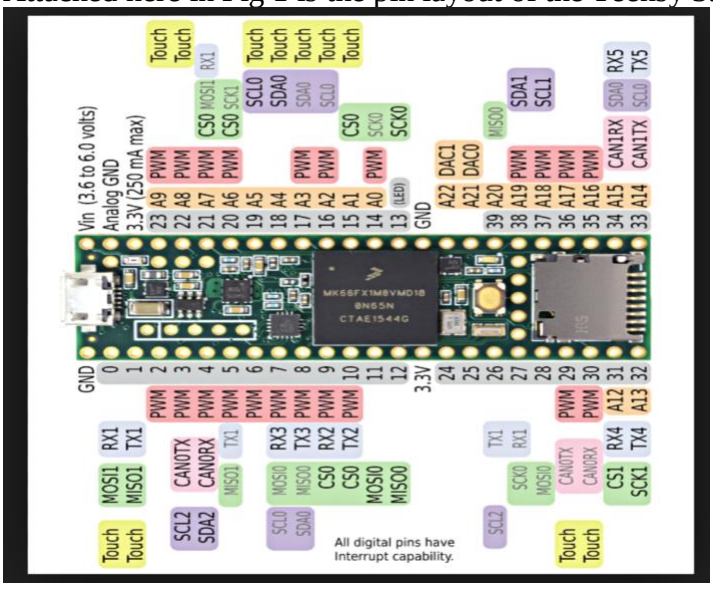
Embedded Signal Processing on the Teensy 3.6

Introduction

The development phase of the radar system involves not only the Analog Circuitry required for the Baseband and RF boards but also processing the signal that is sent from the TX antenna. For the signal processing portion of the radar I had attempted on to perform DSP operations on a embedded system. When deciding which microcontroller to work with for this task, you have to keep in mind the hardware specifications of the MCU. It's recommended that one goes for MCUs with more memory and faster clock rates but also with additional features such as ADCs, SD Card slots and overall support for the MCU. Popular choices would be either Arduino based MCUs or Raspberry Pi boards.

The microcontroller of choice was a Teensy 3.6, by PJRC, which runs on the Arduino environment just as the Teensy 3.2 that was used to produce the Triangle wave input on our Baseband PCB. The reason why we chose to proceed with the Teensy 3.6 was of it hardware properties as well as the availability of libraries that would help us to sample the data that was required.

Attached here in Fig 1 is the pin layout of the Teensy 3.6



[1] Fig . 1 – Teensy 3.6 Pin Layout

The Teensy 3.6 has 12 bit ADCs which met our specification of having an ADC of higher resolution. The higher resolution would allow us to be precise in our data conversion and in our

[1] https://www.pjrc.com/teensy/pinout.html

overall result. Further, the built in SD card allowed us to store data of size that was more than the Flash memory size of (1MB). Therefore, due to the hardware features available on this MCU we decided to use it in our radar design.

Flowchart of Embedded DSP.

The goal for the Embedded Signal Processing aspect of the radar design was to understand completely how the system level integration would work and to do that one should need a module level understanding of the different steps to achieve the task. It is recommended to make a flowchart of the steps
Attached below, Fig 2, is an initial flowchart that describes at a high level of how one can go about the embedded DSP portion of the radar system design.

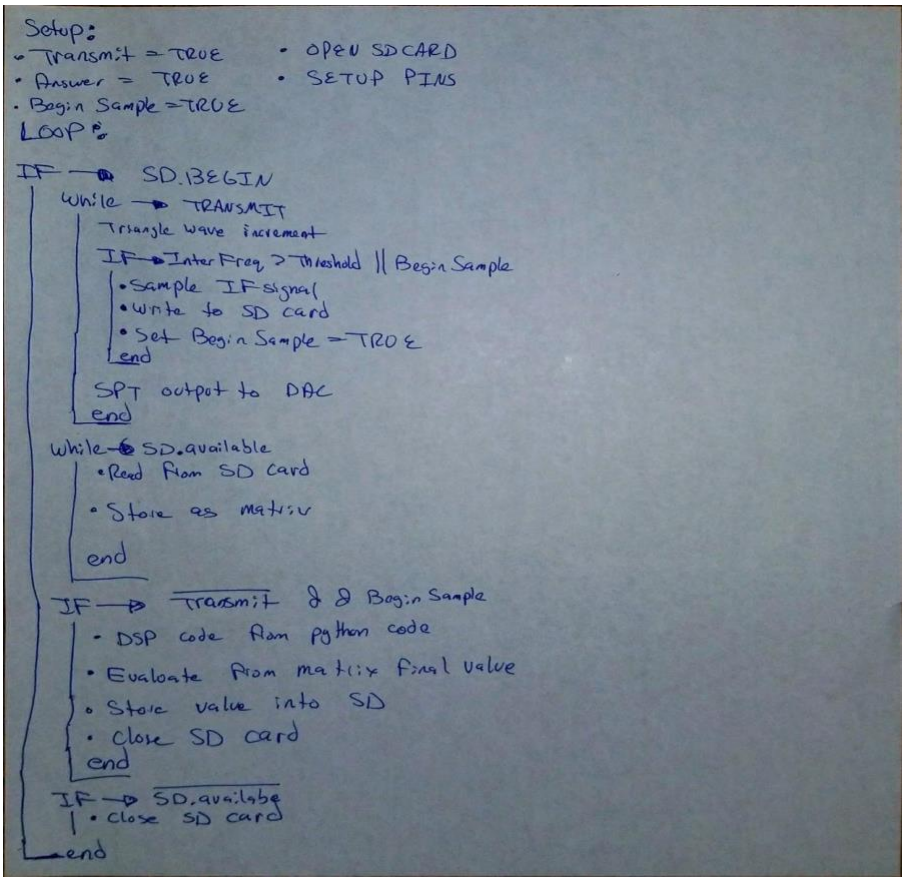


Fig 2. Flowchart of Embedded DSP

For our setup we used two MCUs as using only one MCU for the transmitting and receiving would result in the loss of crucial data that would give us in an inaccurate distance as would have to stop transmitting and start sampling. To solve this issue, we used the Teensy 3.2 to transmit and the 3.6 to process the received signal.

Sampling Data from the ADCs and storing the data.

When attaching any form of voltage to the GPIO pins on the Teensy 3.6 please make sure that the voltage does not exceed 3.3V as suggested by the specifications on PJRC's website. In order to sample the data using the Teensy 3.6 the ADCs have to be used as the incoming data (IF signal) is analog in nature and must be converted to a digital format so that the MCU can process it. In our test case we attached a signal to Pin 16 or Pin A2 of the Teensy 3.6. To perform the sampling of data and storing it on an SD card the following header files must be included in your code, these are ADC.h and SD.h. The ADC header files allows access to the "adc" object that is used to initiate data sampling and setting sampling properties such as speed and conversion; on the other hand, the SD.h header file allows one to write or read from an SD card which was used extensively throughout our procedure.

Attached here, Fig 3, is a sample code snippet that describe the sampling process using the ADC and writing data to a text file on board the ADC.

```
#include<ADC_Module.h>
#include<Audio.h>
#include<SD.h>
#include<SPI.h>


ADC *adc = new ADC();// Creating ADC object
File sync_file;//
File data_file;
File readFile;// Two file variables
const int chipSelect = BUILTIN_SDCARD;

void setup() {

  Serial.begin(9600);// Serial baud rate set to 9600
  // put your setup code here, to run once:
  int Pin= A2;// Pin where data is coming in
  int SyncPin = 27;// Place holder for the sync pin
  int start_pin = 24;// This is the pin that we're using for controlling when we can sample the data.


 //*****************************ADC properties are being set here *************************
  adc->setConversionSpeed(ADC_CONVERSION_SPEED::HIGH_SPEED,ADC_0);// High conversion speed setting
  adc->setResolution(12);// Using 12 bit resolution
  adc->setSamplingSpeed(ADC_SAMPLING_SPEED::HIGH_SPEED,ADC_0);// High sampling speed setting


  int transmit_flag;
  pinMode(Pin,INPUT);
  pinMode(SyncPin, INPUT);
  pinMode(start_pin,INPUT_PULLUP);// Using the pullup mode to perform a digital read for our transmit flag setting
  Serial.println("Initializing SD Card");
  // Checking to see if SD card is accessable.
  //The main issue with on board signal processing is the memory part. Using an SD card to store for a Teensy is the only option however it's quite complex to work with
  // As you have to deal with Files and treat those files as matrices.
  if (!SD.begin(chipSelect)){

    Serial.println("initialization failed!");
    return;
  }
  else{
    Serial.println("initialization done.");
  }

  Serial.println("Opening text file");


 // ****************REMOVING OLD TEXT FILE COPIES**************************
  if (SD.exists("sync.txt")){
    SD.remove("sync.txt");
  }
  if (SD.exists("data.txt")){
      SD.remove("data.txt");
  }

 //***************CREATING THE NECESSARY TXT FILES TO STORE THE DATA*********
  sync_file = SD.open("sync.txt",FILE_WRITE);// Opening the file to write
  data_file = SD.open("data.txt",FILE_WRITE);
  if (sync_file){
     Serial.println("Able to open the SD card for sync_file");
   }
   else{
        Serial.println("Could not open sync file");
   }
```

Fig 3 – Initialization of parameters

In this code snippet the ADC object was called through "new ADC()" and associated to a variable along with the necessary variable for a file based operation. In our case we associated every important variable to a file of the same name as that variable. For example, the sync signal coming from the read by the Teensy 3.6 at Pin 27 was stored in a file called "sync.txt". Similarly, the sampled data from the ADC was stored in "data.txt". To initialize properties of the ADC functions native to the ADC library such as setConversionSpeed, setResolution and setSamplingSpeed have to be used. The setConversionSpeed and setSamplingSpeed functions control the speed of data conversion and sampling , which has three modes: low, medium, high and very high, it's best to use very high if you want to work with more data points. The setResolution function sets the resolution property of the ADC and in our case it was set to 12, the ADC's output bit limit, in order to get the best resolution of our final output value.

To initialize the SD card you have to use the function in the SD.h header file called SD.begin(). The function takes in a parameter that corresponds to the SD card location on the MCU. For the Teensy 3.6 the SD Card is built in and to initialize it you would need to assign "BUILTIN_SDCARD" to a variable; for our purpose we assigned "BUILTIN_SDCARD" to a variable called chipSelect that was passed in as an argument to the SD.begin function. Once the SD card was initialized we used functions such as SD.open and SD.remove to create or remove files on the SD card ,both functions require the file name as a parameter.

After these settings are initiated the sampling and storing operations can be performed. Attached below is a code snippet, Fig 4, that allowed us to sample and store the data on an SD card.

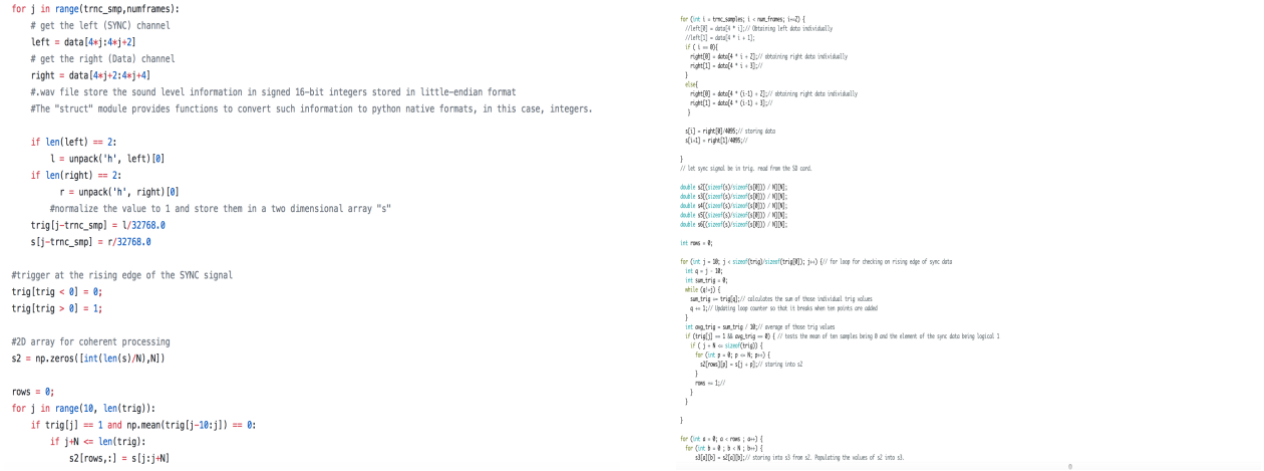
```
    //*****************THIS LOOP CHECKS TO SEE IF THE CONDITION NECESSARY FOR SAMPLING IS MET. ONCE WE HAVE TRANSMITTED WE CAN SAMPLE THE SIGNAL.**************
    while(transmit_flag != 0){
     transmit_flag = digitalRead(start_pin);
     Serial.println(digitalRead(start_pin));
     delay(200);
    }
    Serial.println("Ready to Sample");


    while (transmit_flag != 1){
       Serial.println("In while");
       bool adc_status =  adc->startContinuous(Pin,ADC_0); // Obtaining status of ADC

       if( adc_status == 1){
         int SyncValue = digitalRead(SyncPin);
         int pinValue = adc->analogReadContinuous(Pin);// ADC Value being read
         sync_file.println(SyncValue);
         data_file.println(pinValue);// The sync and the ADC output are written to the text file


       }
       else{
           continue;
         }
       Serial.print("At the end of while");
       transmit_flag = digitalRead(start_pin);
   }

   sync_file.close();
   data_file.close();// Store the data and sync file in 2 different files
   Serial.print("The text files have been completed.");


}
```

Fig 4 – Sampling and Storing data

The sync signal from the Teensy 3.2 was connected at Pin 27 of the 3.6 as this data is required for the processing aspect of the embedded DSP task. A function within the ADC library called analogReadContinuous performs the reads data from the ADC. The input that this takes is the pin, in our case A2, of the ADC where the IF signal is attached to. Once that was done, we read (digitalRead) the data at Pin 27 this where the Sync signal was attached and stored it in the SD card under the file "sync.txt". Similarly, the ADC value was read and placed into the file on the SD card called "data.txt". Controlling these actions was a transmit flag that would stop sampling when the condition failed. The condition was controlled using an external switch, when the switch was turned on the sampling, reading and storing of data would take place. The use of a sampling flag to indicate when to sample and perform other operations is necessary if you are using two MCUs for the embedded DSP part.

Processing of Sampled Data

The last phase of the task requires processing in the form of FFTs and IFFTs of the sampled data that was stored on the SD card. Originally, a Python script called “range_wav.py” was provided to us during Quarter 1 but since we had to port it to the Arduino environment for the Teensy 3.6 we had to convert the Python code syntax to the syntax of C.
Attached below is a side by side comparison of a code snippet , Fig 5, of a section of the “range_wav.py” along with the converted C syntax form. It can also be noted that there’s no need to utilize the FFT function in the Audio library as the Python script already performs Fourier Transform on the data set without calling any external math functions. The IFFT on the other hand needs to be invoked and that depends upon the MCU of choice. If you’re working with the Teensy MCU then ARM provides its own set of mathematical functions within its library that may aid you in your development process.



```
for j in range(trnc_smp,numframes):
    # get the left (SYNC) channel
    left = data[4*j:4*j+2]
    # get the right (Data) channel
    right = data[4*j+2:4*j+4]
    #.wav file store the sound level information in signed 16-bit integers stored in little-endian format
    #The "struct" module provides functions to convert such information to python native formats, in this case, integers.

    if len(left) == 2:
        l = unpack('h', left)[0]
    if len(right) == 2:
          r = unpack('h', right)[0]
        #normalize the value to 1 and store them in a two dimensional array "s"
    trig[j-trnc_smp] = l/32768.0
    s[j-trnc_smp] = r/32768.0

#trigger at the rising edge of the SYNC signal
trig[trig < 0] = 0;
trig[trig > 0] = 1;

#2D array for coherent processing
s2 = np.zeros([int(len(s)/N),N])

rows = 0;
for j in range(10, len(trig)):
    if trig[j] == 1 and np.mean(trig[j-10:j]) == 0:
        if j+N <= len(trig):
            s2[rows,:] = s[j:j+N]
```

```
for (int i = trnc_samples; i < num_frames; i+=2) {
  //left[0] = data[4 * i];// Obtaining left data individually
  //left[1] = data[4 * i + 1];
  if ( i == 0){
    right[0] = data[4 * i + 2];// obtaining right data individually
    right[1] = data[4 * i + 3];//
  }
  else{
    right[0] = data[4 * (i-1) + 2];// obtaining right data individually
    right[1] = data[4 * (i-1) + 3];//
    }

  s[i] = right[0]/4095;// storing data
  s[i+1] = right[1]/4095;//

}
// let sync signal be in trig, read from the SD card.

double s1[(sizeof(s)/sizeof(s[0][0])) / N][N];
double s2[(sizeof(s)/sizeof(s[0][0])) / N][N];
double s4[(sizeof(s)/sizeof(s[0][0])) / N][N];
double s5[(sizeof(s)/sizeof(s[0][0])) / N][N];
double s6[(sizeof(s)/sizeof(s[0][0])) / N][N];

int rows = 0;

for (int j = 10; j < sizeof(trig)/sizeof(trig[0]); j++) {// for loop for checking on rising edge of sync data
  int q = j - 10;
  int sum_trig = 0;
  while (q<j) {
    sum_trig += trig[q];// calculates the sum of those individual trig values
    q += 1;// Updating loop counter so that it breaks when ten points are added
  }
  int avg_trig = sum_trig / 10;// average of those trig values
  if (trig[j] == 1 && avg_trig == 0) { // tests the mean of ten samples being 0 and the element of the sync data being logical 1
    if ( j + N <= sizeof(trig)) {
      for (int p = 0; p <= N; p++) {
        s2[rows][p] = s[j + p];// storing into s2
      }
      rows += 1;//
    }
  }

}

for (int a = 0; a < rows ; a++) {
  for (int b = 0 ; b < N ; b++) {
    s1[a][b] = s2[a][b];// storing into s1 from s2. Populating the values of s2 into s1.
```

Fig 5 – Matrix operations on data read from SD card

The use of matrices to process the data meant that immense amount of data would need to be operated on. Since we used a high sampling rate the storage for the number of samples exceeded the 1MB flash memory limit on the Teensy 3.6. To tackle the memory shortage problem, we used the SD card to act as main memory and this meant that every matrix operation done in C would need to be done by treating the text files in the SD card as large matrices. Therefore, the code was changed so that we could be able to read and write into a text files while treating them as matrices. The shortage of memory on the Teensy is what makes the task of Embedded DSP complex.

Attached below is a modified version of the code, Fig 6, that allowed us to work with text files on board the SD card. The challenge that we encountered was working with File pointers and file operations, this had to be done as there are certain parts that require data extraction of matrix’s column, in our case a file’s column, which is not easily done in C as file operations read data through rows rather than columns.

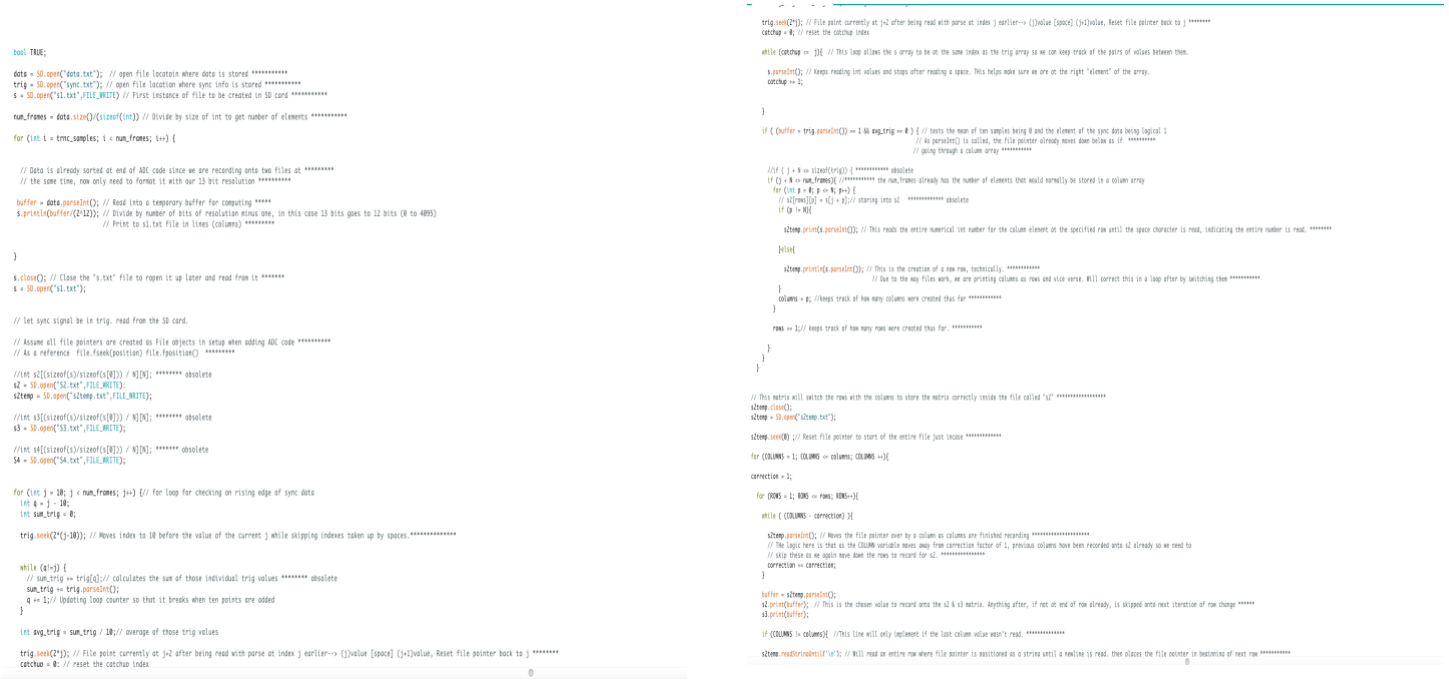


Fig 6 – Corrected version of the processing section

Using the SD card's memory as main memory in this case was the only solution that we thought so that we can perform all actions necessary for signal processing on board the Teensy MCU. Perhaps, there are other elegant solutions out there but this is what we found given the time limit. Please make sure to tackle the memory issue from the beginning as that is the main issue when working on the Embedded DSP part. An alternative solution could be to use a Raspberry Pi that has 1GB of flash memory but requires an external ADC to convert the data.

Conclusion

When designing the Embedded DSP part of the radar design please make sure that you work on areas you are strong with. Please keep in mind that this portion would involve both hardware as well as software understanding to get a functional system. Utilize the benefit of libraries to ensure that there is a modular approach to your design. Overall, this task will help you understand the wonderful world of embedded systems and how to program them, not only would this give you an exposure but also help you to think of solutions to your radar processing using circuitry and software.