App Note: On-Board Processing

EEC 134AB

Jiapeng Zhong (Team Leidar)

Introduction:

- This app note covers the on-board processing part using a TI micro-controller unit (MCU) cc3200: including sampling, on-board FFT. Some functions are provided by TI through Code Composer Studio, such as pin mux, pin writing API, interrupts, etc. The OLED drawing part is provided by Adafruit, and the basic version of Cooley-Tukey FFT algorithm is from online resource. We wrote the SPI communication part of OLED, and improved space and latency for the FFT.

OLED SPI communication:

writeCommand()

```
void writeCommand(unsigned char c) {
    unsigned long ulDummy;

    MAP_SPICSEnable(GSPI_BASE);
    //DC low to indicate command, then enable MCU communication
    GPIOPinWrite (GPIOA0_BASE, 0x40, 0x00);
    GPIOPinWrite (GPIOA0_BASE, 0x80, 0x00);

    //Imitate spiwrite
    MAP_SPIDataPut(GSPI_BASE,c);
    MAP_SPIDataGet(GSPI_BASE,&ulDummy);

    //Disable MCU communication
    GPIOPinWrite (GPIOA0_BASE, 0x80, 0x80);
    MAP_SPICSDisable(GSPI_BASE);
}
```

- This function follows the SPI protocol to pull the target pin low and then write the data and re-pull the target high. Enable the GSPI_BASE at beginning and Disable GSPI_BASE at the end for protection.

FFT

```
//*****************************************************************************
```

```
//******** Implements the Cooley-Tukey FFT algorithm *************************
//*****************************************************************************
static
void FFT_CooleyTukey(int N, int N1, int N2) {
    int k1, k2;
    int k, n;
    /* Allocate column-wise matrix */
    signed long long** columns_real = (signed long long**)
                                    malloc(sizeof(signed long long*) * N1);
    for(k1 = 0; k1 < N1; k1++) {
        columns_real[k1] = (signed long long*)
                                    malloc(sizeof(signed long long) * N2);
    }

    /* Reshape input into N1 columns */
    for (k1 = 0; k1 < N1; k1++) {
        for(k2 = 0; k2 < N2; k2++) {
            columns_real[k1][k2] = (signed long long) sample[N1*k2 + k1];
        }
    }

    complex** columns = (complex**) malloc(sizeof(struct complex_t*) * N1);
    for(k1 = 0; k1 < N1; k1++) {
        columns[k1] = (complex*) malloc(sizeof(struct complex_t) * N2);
    }

/************** Compute N1 DFTs of length N2 using naive method **************/
    for (k1 = 0; k1 < N1; k1++) {
        //columns[k1] = DFT_naive(columns[k1], N2);
        for(k = 0; k < N2; k++) {
            columns[k1][k].re = 0;
            columns[k1][k].im = 0;
            for(n = 0; n < N2; n++) {
                columns[k1][k].re = columns[k1][k].re +
                                columns_real[k1][n] * cos_Naive[k][n];
                columns[k1][k].im = columns[k1][k].im +
                                columns_real[k1][n] * sin_Naive[k][n];
            }
        }
            free(columns_real[k1]);
    }
    free(columns_real);

/*********************Allocate row-wise matrix ********************************/
    complex ** rows = (complex**) malloc(sizeof(struct complex_t*) * N2);
    for(k2 = 0; k2 < N2; k2++) {
        rows[k2] = (complex*) malloc(sizeof(struct complex_t) * N1);
    }

/*** Multiply by the twiddle factors ( e^(-2*pi*j/N * k1*k2)) and transpose ***/
    for(k1 = 0; k1 < N1; k1++) {
        for (k2 = 0; k2 < N2; k2++) {
            rows[k2][k1].re = (columns[k1][k2].re*cos_twiddle[k1][k2] -
                                columns[k1][k2].im*sin_twiddle[k1][k2]) >> 14;
            rows[k2][k1].im = (columns[k1][k2].re*sin_twiddle[k1][k2] +
                                columns[k1][k2].im*cos_twiddle[k1][k2]) >> 14;
```

```
        }
        free(columns[k1]);
    }
    free(columns);

    complex* X_row = (complex*) malloc(sizeof(struct complex_t) * N1);
/*************** Compute N2 DFTs of length N1 using naive method **************/
    for (k2 = 0; k2 < N2; k2++) {
        //rows[k2] = DFT_naive(rows[k2], N1);
        for(k = 0; k < N1; k++) {
            X_row[k].re = 0.0;
            X_row[k].im = 0.0;
            for(n = 0; n < N1; n++) {
                X_row[k].re = X_row[k].re + ((rows[k2][n].re*cos_Naive[k][n] -
                                    rows[k2][n].im*sin_Naive[k][n]) >> 14);
                X_row[k].im = X_row[k].im + ((rows[k2][n].im*cos_Naive[k][n] +
                                    rows[k2][n].re*sin_Naive[k][n]) >> 14);
            }
        }
        for(n = 0; n < N1; n++) rows[k2][n] = X_row[n];
    }
    free(X_row);

/*********************** Flatten into single output ***************************/
    for(k1 = 0; k1 < N1; k1++) {
        for (k2 = 0; k2 < N2; k2++) {
            result[N2*k1 + k2] = mag(rows[k2][k1]);
        }
    }

    for(k2 = 0; k2 < N2; k2++) free(rows[k2]);
    free(rows);

    return;
}
```

The original code was from https://github.com/jtfell/c-fft.git. The original version used 2 double to represent the real and imaginary part of a complex number, and the sin/cos result was generated from the sin/cos function from `<math.h>`
To speed up the processing, we changed the type from **double** to **signed long long with with 14 bits after decimal point** (~0.00006). Each signed long long data type is 8 Bytes (64 bits) at least; for one FFT computation, we need to keep at least 3 arrays simultaneously even if we free the intermediate arrays. The micro-controller cc3200 has 256 kB RAM on the processor, thus, after considering the data storage and code storage, we decide to using a sample space with size of **1024 samples**. This require 8 * 1024 = 8 kB from stack for each FFT. We also changed the way to obtain sin/cos result to using **two set of tables to store the naive sin/cos result and the twiddled sin/cos result**, since we know we are using 1024 samples to perform the FFT.

Finding Frequency

```
max_val = result[1];
max_index = 1;
for (i=70;i<730;i++){
    if (result[i] > max_val){
        max_index = i;
        max_val = result[i];
    }
}
freq = max_index * 3125 / 1024;
```

After performing FFT, we get 1024 indices evenly distributed in the 0 - 2π range. We have a sample frequency of 3125 Hz (scaled using counter, maximum sample frequency is 62500 Hz). Our triangle has a frequency of 33.33 Hz, thus the maximum frequency we are expecting for 50m is

$$maximum frequency = \frac{2 \times 50m}{3 \times 10^8 m/s} \times \frac{1}{15ms} \times 3V \times 33.1MHz/V = 2206.6667$$

Thus, we are expecting a minimum frequency of 220.67 Hz.
In order to find the frequency with max power from 220.67 Hz (around index 72.3) to 220.67 Hz (around index 723), we only looking for the max_index from 70 to 730

from the frequency, we can find the distance by using:

$$Range = \frac{\Delta f \times c \times T}{4 \times \Delta V \times sensitivity}$$