Lab2: Energy Meter

In this lab, you'll build and program a meter that measures voltage, current, power, and energy at DC and AC.

Assigned: October 2, 2017

Due: Week of October 9, 2017

Part 1

New Code

Pull the latest code from the class Github repo using `git pull`. If you are working on a private repository you will have to run `git pull upstream master` assuming you have mapped a remote to the class repository. If you haven't set up a remote repository for the class Github you can through the command `git remote add upstream https://github.com/ndanyliw/green-electronics.git`. The lab starter files will be under 'labs/lab2/'. For more details look at our Git Tutorial.

For this lab, you should be able to do everything by editing lab2.c and lab2.h. Function prototypes are provided and all you need to do is write the code for all the necessary methods.

Part 2

Analog to Digital Conversion

To measure power delivered to the load, you need to measure the voltage across the load and current through the load nearly simultaneously. Circuitry is provided to convert the load voltage and current into voltages which are safe for the microcontroller to measure.

The STM32F303 has 4x12-bit 5 Msps ADCs connected to 39 external channels. There are many modes in which these ADCs can be operated and controlled. For those who are curious please check out the chip datasheet and user manual (located in the "datasheets" folder). The provided ADC library samples the user-specified channels sequentially and stores the results using the DMA controller. The conversion order is specified by the programmer. At the end of a set of conversions, the library will call the provided user callback and pass in the results to be processed. The reference for the ADCs is set to 3V so analog signals will have to range between 0-3V to avoid saturation.

The converters are set to operate in a single ended mode so a result of 0 corresponds to 0V and 0xfff is 3V. On the fast channels the conversions take 0.19 μs. This can be sped up to 0.16 μs if reducing the resolution to 10 bits.

The EE155 libraries provide a simplified interface to the ADCs with a small subset of their possible func-

Part 2 continued on next page. . .

tionality. To fully utilize the converters on the board you will have to expand on the libraries' functionality using STM's abstraction libraries (check out stm32f30x_adc.h/c).

For a good overview on the ADCs check out the STM32F3_ADC PDF in the Github repository.

ADC Initialization

Before you can use the ADCs, you will have to initialize the library. The basic ADC initialization is handled in the `ge_init()` method.

To set the sampling frequency use `adc_set_fs(float fs)`. This will set up TIM2 to trigger the ADC at the appropriate intervals. Specify the frequency in Hz.

Setting up Channels

To enable specific ADC channels, you must specify the channels and pass them to the `adc_enable_channels()` method. This function expects an array with the desired channels in the order to convert. The library will then handle enabling the necessary pins and scheduling the conversions. The final order of conversion is determined by the order specified by the user and the ADC used. Each ADC runs simultaneously and the order it does its conversions is dictated by the user's desired order. For example if the user specified {ADC1_1, ADC1_3, ADC2_1, ADC1_2} for their conversions, where ADCx_y corresponds to channel y on ADCx, the actual conversion order would be as follows:

ADC1_1 — ADC1_3 — ADC1_2

ADC2_1

Assuming that ADC1 and ADC2 are triggered on the same event (which they are in the library) the conversions for ADC1_1 and ADC2_1 occur simultaneously.

The names for the available ADC channels can be found in ge_adc.h in the enumerated ADC_CHAN_Type structure. The channels follow the naming scheme ADCx_y (channel y of ADCx). To see what pins are mapped to the ADC channel, refer to the adc_pin_map array in ge_adc.c. The available channels and mapped pins are also outlined in Table 1.

Starting Conversions

After configuring the ADC library, to actually start conversions use `adc_start()` which will launch the timer and begin conversions. To stop the converter use `adc_stop()`.

Processing Results

After all the conversions complete, the library will call the user provided callback function. The function is given a pointer to the array containing all the most recent conversion results. The data is formatted as 12-bit unsigned integers and stored in `uint16_t` variables. The order of the array corresponds to the original channel ordering given by the user to `adc_enable_channels()`.

Part 2 continued on next page...

ADC1 Channel	Pin	ADC2 Channel	Pin	ADC3 Channel	Pin	ADC4 Channel	Pin
ADC1_1	PA0	ADC2_1	PA4	ADC3_1	PB1	ADC4_1	PE14
ADC1_2	PA1	ADC2_2	PA5	ADC3_2	PE9	ADC4_2	PE15
ADC1_3	PA2	ADC2_3	PA6	ADC3_3	PE13	ADC4_3	PB12
ADC1_4	PA3	ADC2_4	PA7	ADC3_5	PB13	ADC4_4	PB14
ADC1_5	PF4	ADC2_5	PC4	ADC3_12	PB0	ADC4_5	PB15
ADC12_6	PC0	ADC2_11	PC5	ADC3_13	PE7	ADC4_12	PD8
ADC12_7	PC1	ADC2_12	PB2	ADC3_14	PE10	ADC4_13	PD9
ADC12_8	PC2			ADC3_15	PE11		
ADC12_9	PC3			ADC3_16	PE12		
ADC12_10	PF2			ADC34_6	PE8		
				ADC34_7	PD10		
				ADC34_8	PD11		
				ADC34_9	PD12		
				ADC34_10	PD13		
				ADC34_11	PD14		

Table 1: ADC channels and corresponding pins.

It is important that any code run in the callback finishes execution before the next sampling interval. Otherwise the data may be overwritten with the most recent conversions.

The provided callback function must follow the following prototype:

`void (*adc_reg_callback)(uint16_t *)`

The callback is registered with the function `void adc_callback(void (*callback)(uint16_t *))`.

ADC Channels on the Breakout Board

The Green Electronics breakout board has 4 readily accessible analog channels through the defined macros: `GE_Ax` where "x" is between 1 and 4. Two of the channels (`GE_A3` and `GE_A4`) are connected directly to ADC pins on the Discovery board. The other two (`GE_A1` and `GE_A2`) pass through instrumentation amplifiers first which allow them to accept a differential signal and reject common-mode noise, making them ideal for sensitive measurements. **The instrument amps have an internal gain of 5 so you may have to condition the input appropriately and add attenuation if necessary. Additionally to handle differential signals, the output is referenced to 1.5V (so a differential input of 0V corresponds to 1.5V).** This means that you will have to subtract a fixed offset from the conversion result to get the actual measured voltage. To see what ADC channels the external pins correspond to, take a look at the definitions in ge_pins.h.

ADC example

Listing 1: ADC example code

```
/**
 * Calculate power using the ADCs
 */
```

```c
#include "ge_system.h"
#include "ge_adc.h"

//volts per division
#define V_PER_DIV .0025
//amps per division
#define A_PER_DIV .001

//ADC callbacks
void calculate_power(uint16_t *data);

float power_result;

//initialize ADCs
ge_init();

//set sampling frequency at 50kHz
adc_set_fs(50000);

//add callback to channel 1
adc_callback(&calculate_power);

//enable ADC1_1 (PA0) and ADC1_2 (PA1)
adc_enable_channels([ADC1_1, ADC1_2], 2);
adc_initialize_channels();

//start ADCs
adc_start();

//wait for interrupts
while (1) {};

void calculate_power(uint16_t *data) {
  uint16_t chan1 = data[0];
  uint16_t chan2 = data[1];

  power_result = (float) chan1 * V_PER_DIV * (float) chan2 * A_PER_DIV;
}
```

Part 3

Hardware

Your energy meter should be able to measure 200V and 10A, for a total instantaneous power of 2kW. It should work on both DC and AC.

Our power and energy meter connects between a power source and a load. The meter senses the load voltage VL by measuring the voltage between the positive and negative load terminals. The meter senses the load current by measuring the current that flows from LOAD- to SOURCE-.

Part 3 continued on next page. . .

The schematic for the power and energy meter is shown in Figure 1. The exact component values are left for you to calculate as a lab exercise. The energy meter measures the current using a current sense resistor and the on-board instrumentation amplifier. The voltage reading is sensed through a voltage divider with an offset set by R3 and R4 to allow the measurement of negative voltages.

The current sense resistor RS is on the "component board", which has terminals for using various passive high-power parts. All of the circuit except RS, R1, R2, and C1 should be built on a breadboard and powered from the control module. There are two sets of terminals for RS: the large terminals connect to the source and load, while the small header terminals connect to the energy meter circuit.

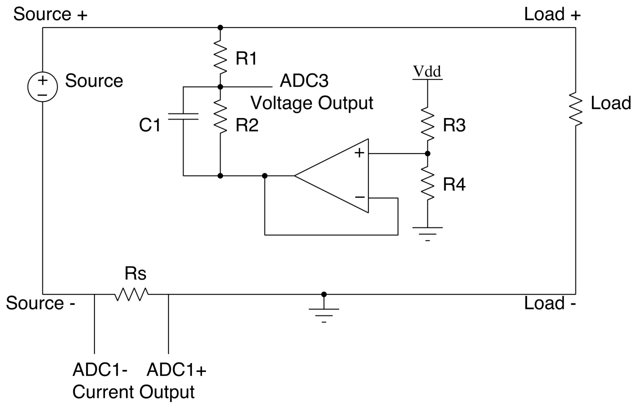

Figure 1: The energy meter sits between a source and a load. It connects between the source and the load on the negative side.

Two analog inputs of the processor module, ADC3 and ADC1, are used to sense the load voltage and current respectively. A voltage divider composed of resistors R1 and R2 scales the load voltage VL to the 0-3V input range of the processor's ADC. The voltage divider of R3 and R3 should set the zero-input voltage on ADC3 to 1.5V (the center of the ADC range) to allow bipolar (positive and negative) measurement.

The RS = 10mΩ current sense resistor connected between NL and NS converts the 10A load current into a 100mV voltage. We connect LOAD- to processor ground so that our voltage measurement is the voltage across the load, which differs from the voltage across the source by the amount dropped across the current sense resistor. We use one of the provided instrumentation amplifiers (INA827) on the breakout board to amplify the ±100mV voltage across the sense resistor to fit the 3V input range of the ADC. We use a differential amplifier here, rather than a single-ended inverting or non-inverting amplifier, to cancel any noise that exists between LOAD- and the actual processor ground. The gain of the amplifier is 5 by default but can be set by populating R3 (on the breakout board) and picking the resistor according to the formula: gain $= 5 + (\frac{80\text{k}\Omega}{\text{R3}})$.

Since R1 has a high voltage connected to it, the connections on R1 and R2 must be solid and reliable. This voltage divider should be soldered together. R1 should be at least 100kΩ.

Part 4

Filtering

A common and simple approach to smoothing signals is to use a one-pole IIR filter: $y_n = \alpha y_{n-1} + \beta x$ where the DC gain is $\frac{y}{x} = \frac{\beta}{(1-\alpha)}$.

We want α to be slightly less than one to provide a low-pass response. To keep the implementation fast and simple, we'll choose $\alpha = 1 - \frac{1}{2^{shift}}$ and $\beta = 1$, where *shift* is some integer > 0. This can be written as: $y_n = y_{n-1} - \frac{y_{n-1}}{2^{shift}} + x$

The DC gain is $\frac{y}{x} = 2^{shift}$, so to produce unity gain the output needs to be divided by 2^{shift}. These divisions can be done with bit shifts if working with integers, so they are fast.

You will need to write filtering functions for your power meter to avoid excess noise on the readings.

Part 5

Calibration

The ADC measurements are not perfect, and the resistor values used in the voltage divider and current-sense circuit are not exact. The result is that 0V and 0A will not typically produce an ADC value of 2047, and other voltages will not produce exactly what you expected when you calculated the resistor values. To correct for this, you need to measure at least two points to find the conversion from ADC counts to voltage or current.

Since any change to the circuitry could change the offset or scale, the calibration procedure needs to be simple and automated. To do this, add a special mode to your program to measure 0V, 0A, a known voltage, and a known current. Make the known values small enough to be safely handled when testing but large enough to avoid excessive rounding error (a single ADC count is not a good choice). Set `CAL_VOLTS` and `CAL_CURR` in lab2.h to the values you choose. The defaults are 10V and 3A.

The main loop provided to you uses pushbutton 1 to select among four screens: live measurements, calibrating offset, and calibrating voltage and current scale factors. Pressing the pushbutton 2 in a calibration screen causes one of the `calibrate_*()` functions to be called, which is where you should calculate and store the new calibration data.

Calibration data is stored in EEPROM so you don't have to recalibrate every time you turn on the meter. `meter_init()` reads this calibration data from EEPROM and the `calibrate_*()` functions write it back to EEPROM when it's updated.

Part 6

Testing

Use a heating element as the load. This device is almost purely resistive and can dissipate lots of power (enough to burn you, so don't touch it after you start delivering significant power to it). For a DC supply, use one of the 100V 10A power supplies. Demonstrate that your energy meter correctly measures the power and energy delivered to the load for three different operating points. Apply a known amount of power for a known amount of time and verify that the total energy delivered is as predicted. How accurate is your meter? How much error does it show under heavy load (1kW) and under no load? Use the infrared thermometer to measure the temperature of the load after delivering 1kJ.

Signoffs

1. Show DC power and energy measurements for at least three different operating points.

Show correct four-quadrant measurements:

1. Exchange the source and the load to invert the current.
2. Exchange the source terminals to invert the voltage.