Bubbly Technical Manual

Daniel Andrade
Revision 1.0

Last modified on September 3, 2015

Contents

List of Figures

List of Tables

1 Preface

1.1 License and Warranty

The entirety of this product (including the hardware design) is released under the open-source MIT License, which is reproduced below.

The author makes no promise of support for the system beyond this manual and the user manual.

1.2 Basic Description

Bubbly is a digital data acquisition device that samples a specific signal (noisy square wave) coming from an optical probe, processes it internally, and outputs the low average, high average, and duty cycle to a character display. Bubbly was specifically made to be used in bubble measurement, although it works as a general square wave duty cycle detector.

1.3 Input Signal Description

Bubbly's intented input signal comes from a bubble detection device. The bubble detection device consists of an optical probe and light detection hardware (essentially a 50/50 coupler and a phototransistor). Due to the different refraction indices of water and air, this system can be used to determine whether the end of the probe is in water or inside a bubble. A sketch of the system is shown below:

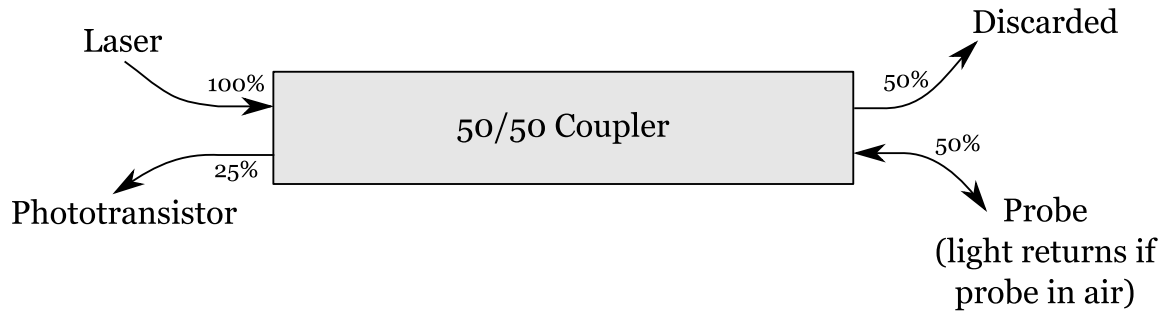


Figure 1: Diagram of the bubble measuring device

The above device generates a high output voltage whenever the probe is in air, because light hits the phototransistor, and a low output voltage when the probe is in water, because no light returns from the probe. In practice, some light always returns and so the voltage is not zero. An example output waveform with exaggerated nonidealities is shown below.

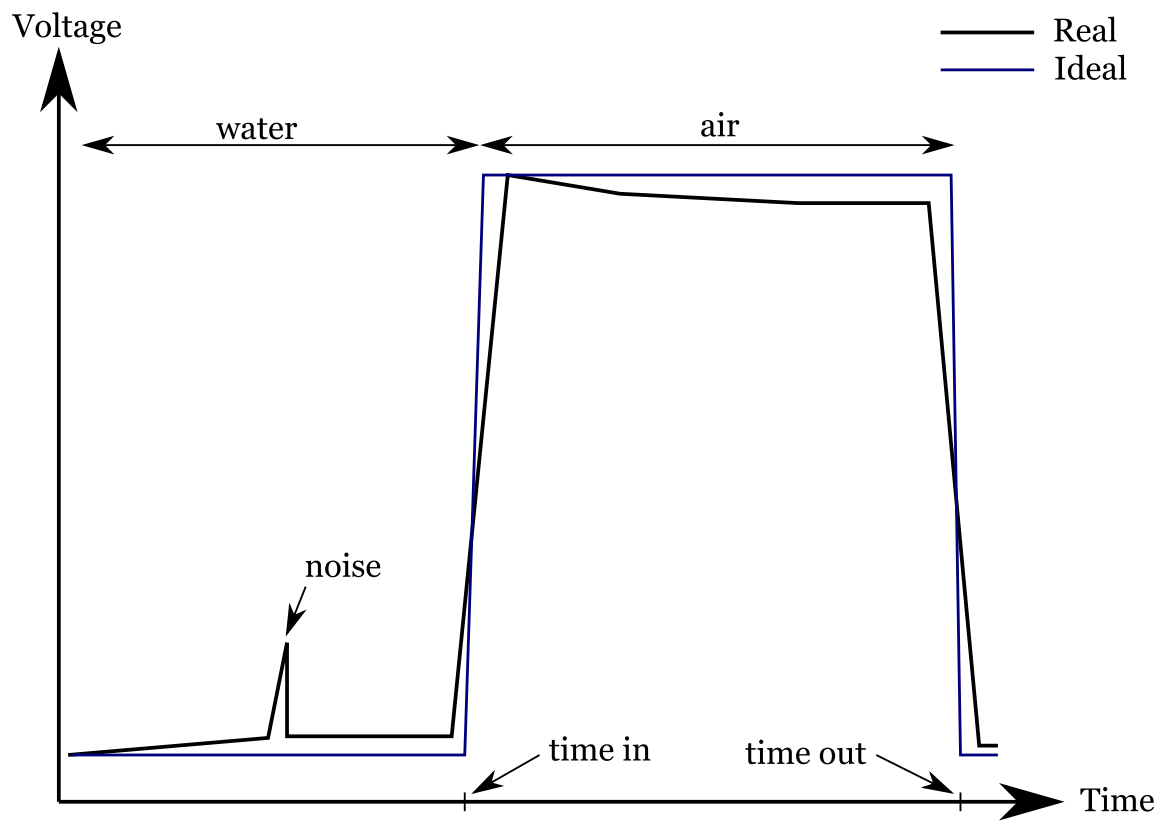


Figure 2: Real and ideal waveforms

Bubbly determines the "time in" and "time out" edges, thus turning the black input signal in Figure 2 into the shown blue output signal (a simple square wave). The edges are determined by comparing to a threshold (variable or fixed, depending on device setup). The device uses this representation to calculate the void fraction of the liquid (same as the duty cycle of the signal) as well as the average low and high values of the signal for the past thousand samples and the threshold, if a variable threshold is chosen.

2 Hardware Overview

2.1 Block Diagram

This section illustrates the high-level interconnectivity of the hardware components on Bubbly. The color coding is used to distinguish data signals (in green) from control signals (in red) and from other signals (in black). Please see Appendix A for a larger version of this diagram.

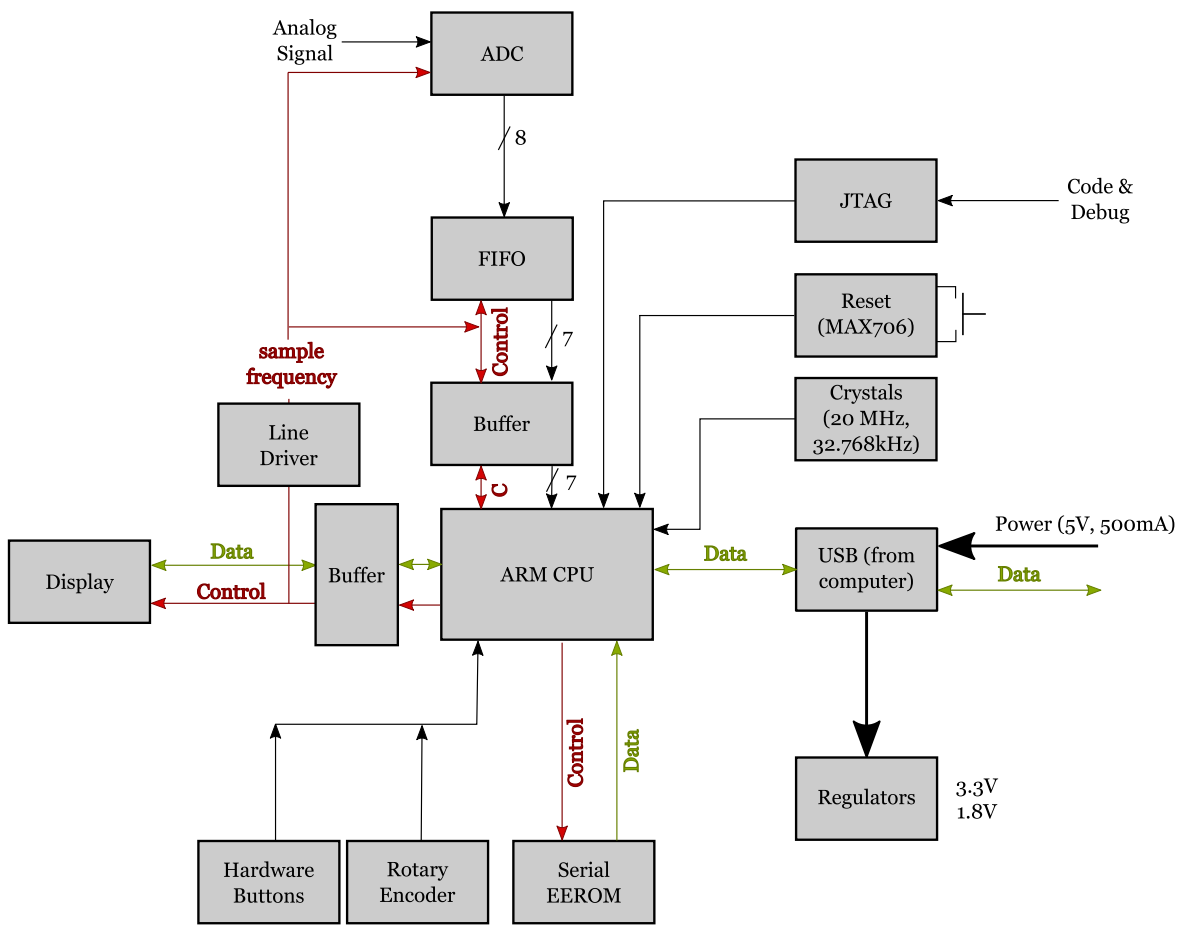


Figure 3: Hardware block diagram

2.2 System Overview

2.2.1 General Hardware

Atmel ARM Processor The system uses an Atmel AT91RM9200 processor as its central chip. This chip has many on-chip modules which make it capable of interfacing with a lot of hardware. The following on-chip modules were utilized for this project:

- 16 KB of on-chip SRAM for storing code and variables
- UDP (USB Device Port) interface to talk to the computer
- TWI (Two-Wire [serial] Interface) interface to get initial data from serial EEPROM
- PIO (Parallel IO) ports for interfacing with switches, display, and ADC
- AIC (Advanced Interrupt Controller) for dealing with interrupts

ADC After some major modifications (see the ADC subsection of the *Detailed Hardware Description* for more information), the final design for the system uses an 8 bit, 20MHz maximum sampling rate ADC to sample the incoming bubble signal data.

FIFO A 50MHz, 4 KB, 9 bit FIFO was placed between the ADC and the processor to buffer the incoming data. This allows the processor to process the data at a later time and drastically reduces the number of incoming interrupts.

Serial EEPROM The serial EEPROM holds the code and initial variable values. If the processor detects the EEPROM it will load the code and run it upon power-up.

JTAG Interface The JTAG interface is used to program the processor from a computer with a serial port through a Wiggler board.

Buffers The buffers buffer every pin used on the processor other than the button and encoder signals, which are not buffered.

Power & Regulators The system is powered directly from the USB interface (from a computer or wall power supply). The incoming 5 Volts are regulated down to 3.3 V and 1.8 V for interfacing with chips and powering the processor.

Crystals The processor requires both a slow clock, required to be 32.768 kHz, and a faster clock, which could be chosen from a set of values. This system uses a 20 MHz clock as the other, faster clock.

Reset Circuitry The reset chip is used to reset the processor when the reset button is pressed.

2.2.2 Human Interface Hardware

See *User Input/Output* and *Functional Specification* for more details

Character Display The system uses a 16×2 HD44780-compatible character display to show system status, current thresholds, and any potential errors. The system supports the display backlight as an optional feature.

Pushbuttons The system has two pushbuttons. One of these is red and is used as the reset button to the system. The other is black and functions as the debug button.

Rotary Encoder The system also has a rotary encoder switch. The switch has its own pushbutton which serves to toggle what is shown on the character display. The encoder itself is used to select settings, such as the sampling frequency.

2.3 Approximate Board Signal Layout

The following diagram gives the approximate layout of the signals on the board. This should be used as an introductory guide only in order to be aware of what section of the board performs what function. Please see the actual PCB layout for details on the trace locations on the board.

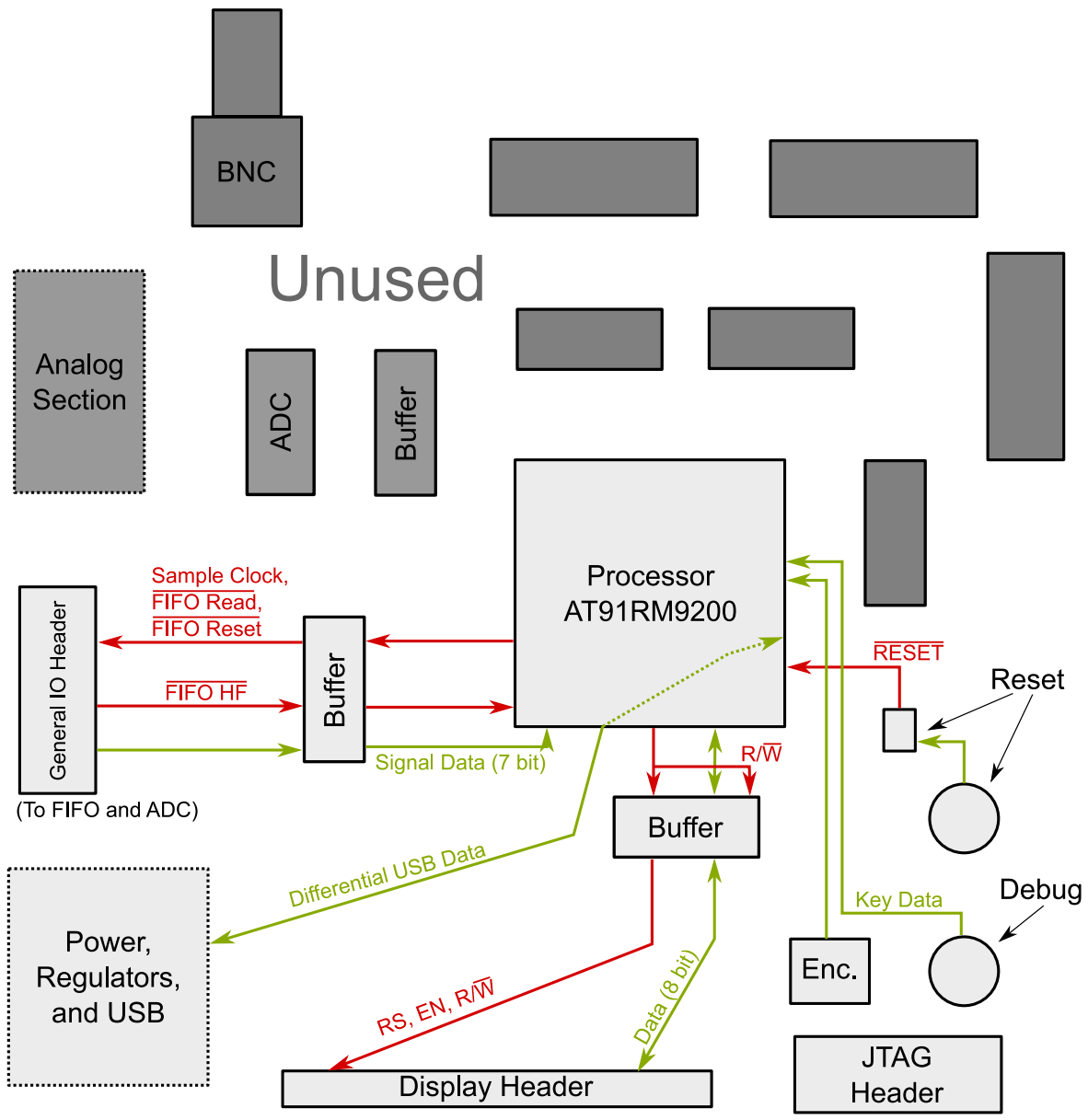


Figure 4: Approximate board layout

2.4 Memory Map

Disclaimer: This memory map was taken directly from the AT91RM9200 datasheet, page 17. I do not claim to have made this image, and it's included in this manual as a reference only.

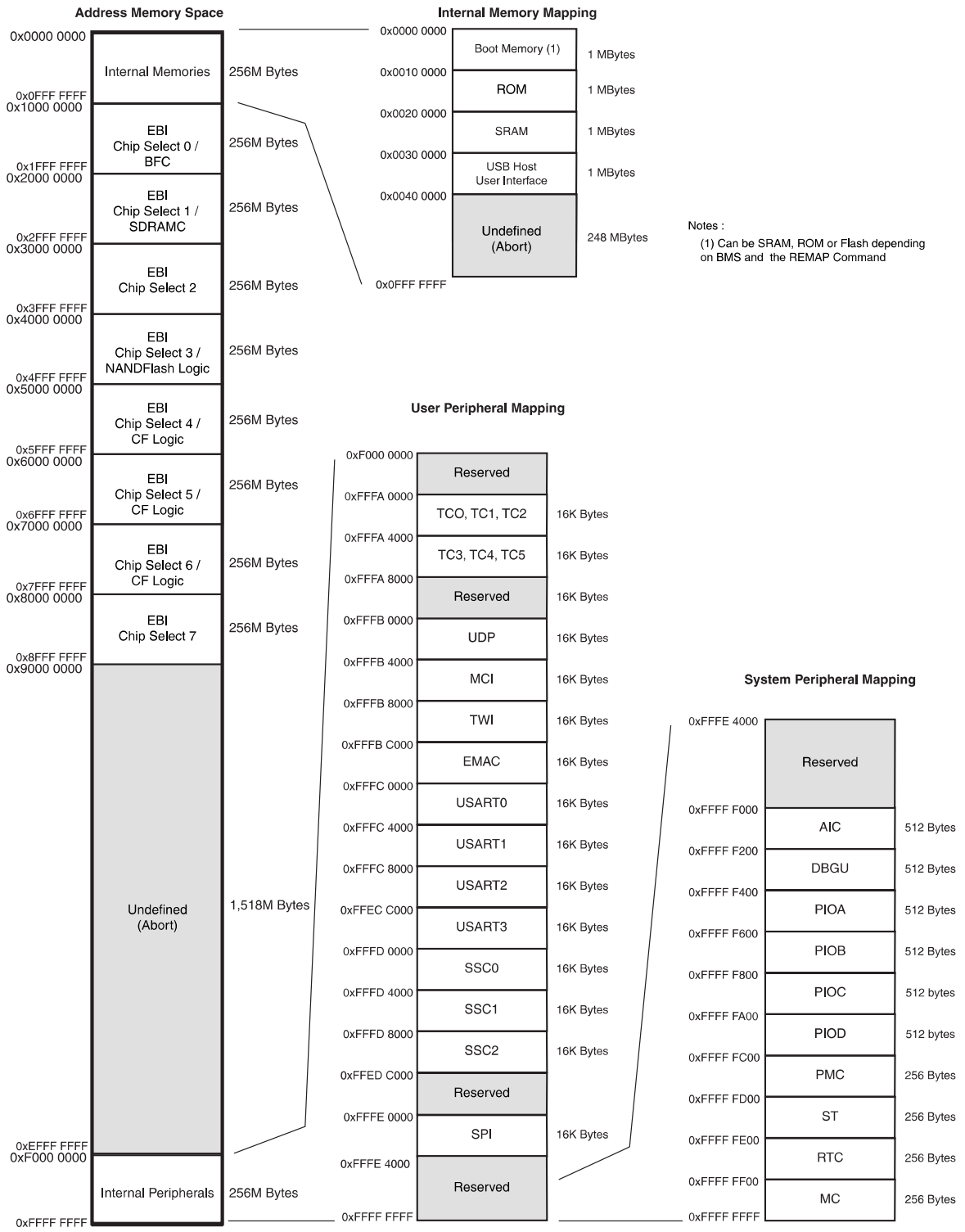


Figure 5: AT91RM9200 memory map

3 Detailed Hardware Description

3.1 Processor

The system uses the "standard" configuration for the processor, in that no special configuration was necessary to interface with it.

3.1.1 PIOs

The processor has three general PIO banks which are multiplexed with peripheral interfaces. These banks were used for the following purposes:

	PIO A	PIO B	PIO C
I/O	Eight general inputs	Display, buttons, and encoder	ADC input
Peripherals	Sample clock output, interrupts	None	None

Table 1: Signals separated by PIO Bank and purpose

The following three tables describe how each PIO pin is connected in the system. Pins that are left out were unused. All inputs/outputs are with respect to the processor.

PIO A Pin	Sch. Name	I/O	Description
PA0/**PCK3**	—	O	Clock interrupt line
PA1/**PCK0**	—	O	Sample clock
PA3	—	O	FIFO Reset Line (active low)
PA5	—	O	FIFO Read Line (active low)
PA8	—	O	FIFO Half-Full Line (active low)
PA9 - PA15	—	I	FIFO Output Lines
PA16	DirIO1	O	Buffer direction (1) (low - input)
PA17	DirIO2	O	Buffer direction (2) (high - output)

Table 2: PIO A pins used in system

PIO B Pin	Sch. Name	I/O	Description
PB0	R/W	O	Read/write line to display
PB1	EN	O	Enable line to display
PB2	RS	O	RS line to display
PB3	SMPL_CLK	O	Unused (cut, sample clock re-routed)
PB4 - PB11	D0 - D7	I/O	Display data lines
PB12	InvR/W	O	Used for display buffer direction
PB25	keyDebugB	I	Debug pushbutton input
PB26	keyToggleB	I	Toggle pushbutton input
PB27	keyBB	I	Quadrature (B) input from encoder
PB28	keyAB	I	Quadrature (A) input from encoder

Table 3: PIO B pins used in system

PIO C Pin	Sch. Name	I/O	Description
PC19	OTRB	I	Out of range indicator of ADC (unused)
PC20 - PC31	SIGxB	I	x^{th} data line of ADC (unused)

Table 4: PIO C pins used in system

Note that none of the PIO C pins were used in the final system because the ADC input was re-routed through PIO A.

3.1.2 UDP

The USB Device Port consists of two (differential) lines, labelled DDM and DDP on the schematic. For USB device configuration, please see *Appendix E.*

3.1.3 TWI

The TWI interface needs no special configuration, just two pull-down resistors on the two data lines (TWD and TWCK). These resistors are not on the schematic because of a design error; thus, two $10k\Omega$ pull-downs had to be soldered to the chip and board.

3.1.4 PLL Filter

The system uses a filter on the PLLB ports of the AT91RM9200 processor to achieve the 48MHz clock necessary for TWI and UDP. The PLL is set up to generate a 96 MHz clock, which internally gets divided down by a factor of two to 48MHz. The three necessary components to the filter are a 768 Ω resistor, a 1 nF capacitor, and a 10 nF capacitor.

3.2 ADC

The system was designed to use a ADS807 ADC, but something undetermined about the design was awry. Although the ADC did seem to work well, the processor often died (system reset) when the ADC input was connected to the PIO ports, which made the ADC design unfeasible. To remedy this design problem, a TLC5510 ADC breakout board was attached to the general PIO bank A, and the ADC outputs were rerouted through there. This "quick fix" has several important disadvantages, most importantly that the resolution decreased drastically from 12 bits to 8 bits and that the maximum sampling rate decreased from 60 MHz to 20 MHz. However, only eight of the PIOA inputs could be used as inputs, because the other 8 broken-out pins were already configured as outputs (the sample clock output, unfortunately, must come from the PIO A bank because only the PIO A bank provides programmable clock outputs). Since a higher than 8 bit resolution was very difficult to achieve given this setup, the TLC5510 chip was a solid choice, especially because the system designer was familiar with it, so the probability of committing another design error was significantly lower. This section will outline the designs for both ADCs, as the ADC breakout board still has both designs on it.

3.2.1 ADS807 Design

The ADS807 is a fairly expensive 12 bit differential TI ADC. The design includes a DC-coupled, three operational amplifier front-end that converts the signal from a single-ended signal to a differential signal (as well doubling it). The design for this signal converter was taken directly from the ADS807 datasheet. The ADS807 can also be set to use internal or external references and to have a two or three volt peak-to-peak input range. The references were permanently set to be internal for the system, but the input range selection was implemented as a jumper header to leave the decision up to the end user. The design for this is too large to fit into this page, but can be found in the *Original Schematics* appendix. Note that the only differece between the configuration found there and the one in the analog breakout board is the addition of a couple of bypass capacitors which were missing in the original design.

3.2.2 TLC5510 Design

The TLC5510 is another, more economical TI ADC. It has an eight bit resolution and a maximum sampling rate of 20 MSPS. It takes in a single-ended input, so it is much simpler to use. The TLC5510 was chosen over the TLC5510A chip to avoid having to get a regulated 4 volt source for the reference and because the extra input range was not important to the project. The schematic for the TLC5510 design on the breakout board is shown below.

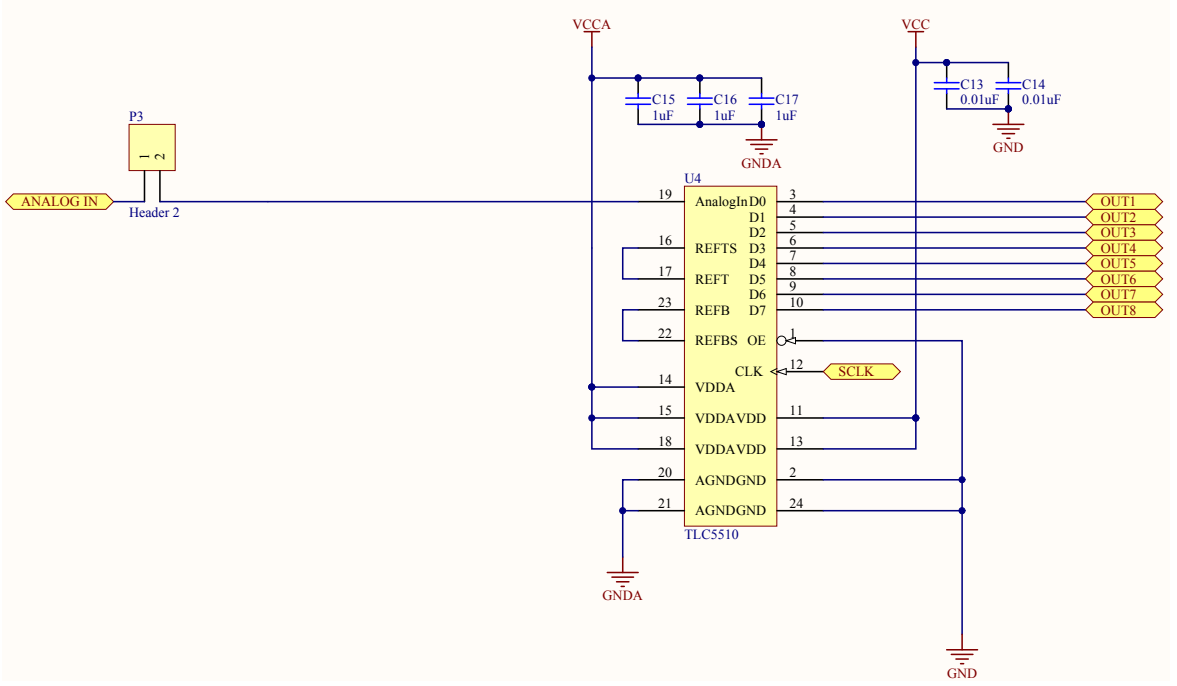


Figure 6: TLC5510 ADC design

3.3 FIFO Memory

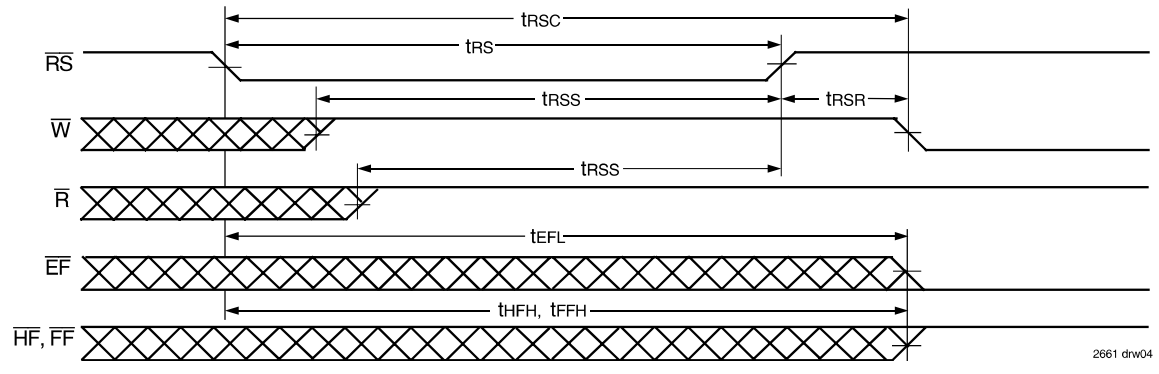
The FIFO is a Integrated Device Technology, IDT2703-08 dual-port 4 KB x 9 memory chip with a read/write speed of 50MHz. The FIFO is used as a hardware memory buffer which the processor can poll at its own leisure. This allows the sampling rate to be higher because the processor no longer needs to respond to an interrupt every time the ADC outputs a sample.

Polling The processor constantly polls the half-full flag output of the FIFO to determine when to start reading it. Once this half-full flag goes low to indicate the FIFO memory is at least half-full (it has at least 2,048 samples stored), the processor will consecutively ready 1,000 samples. For more information see the *Detailed Software Description*.

3.3.1 Timing

There are a lot of timing requirements for the chip because it has several different modes of operation and features. Because the system uses the FIFO as a simple memory buffer, there are only a couple of timing parameters that directly affect this application.

Reset Timing Before the FIFO can be used in the system, it must be reset by pulsing the RESET pin. It is important that the read and write lines be held low during this process. The reset pin must be held low for at least 12ns (t_{RS}), and the read and write lines must be low for an additional 8 ns ($t_{RSR} = t_{RSC} - t_{RS}$) after RESET goes high. A diagram of this (pulled directly from the FIFO datasheet) is reproduced below.



NOTE:
1. $\overline{W}$ and $\overline{R}$ = V_{IH} around the rising edge of $\overline{RS}$.

Figure 7: FIFO memory reset timing diagram

Read Timing A read cycle consists of a pulse of the read signal from high to low and then back to high. There is a maximum 12ns (t_A) delay until the output data is valid, so the processor must wait at least this long before reading the PIO A inputs that correspond to the output data of the FIFO. There is a further requirement that the full read cycle be at least 20ns (t_{RC}) long, giving a maximum read speed of $\frac{1}{20\,ns} = 50$ MHz if samples are consecutively read from the FIFO memory. A diagram of this (pulled directly from the FIFO datasheet) is reproduced below.

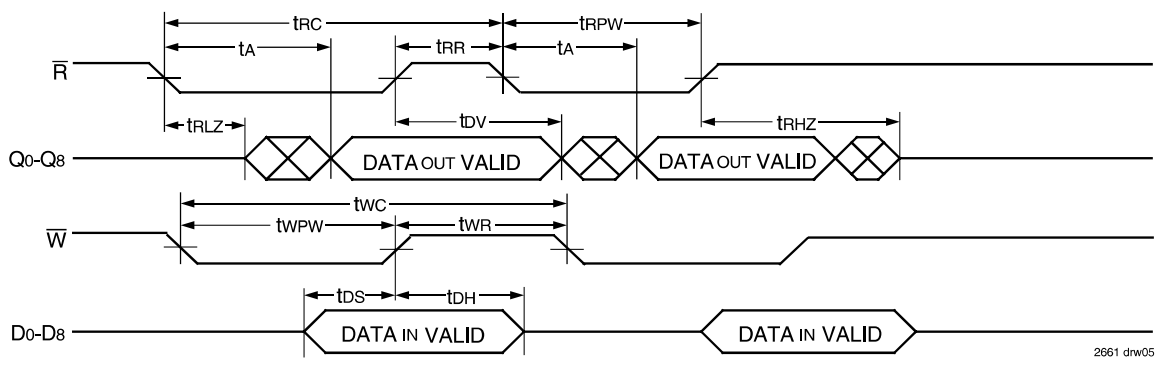


Figure 8: FIFO memory read timing diagram

The above timing parameters are reproduced below.

Timing parameter	Limit	Max/Min	Description
t_{RS}	12 ns	Min	Reset Pulse Width
t_{RSC}	20 ns	Min	Reset Cycle Time
t_{RSR}	8 ns	Min	Reset Setup Time
t_A	12 ns	Min	Access Time
t_{RC}	20 ns	Min	Read Cycle Time

Table 5: FIFO Memory Timing Parameters

With the processor clock currently chosen (20 MHz), it is impossible to not meet these timing requirements, as they are very lenient. However, a constant is used in the code in the function `init_fifo` for ensuring that the reset timing is met, even for extremely fast processor configurations (up to the maximum of 200 MHz).

The read timing should never be an issue, regardless of processor speed, because the code does a lot of calculations between reads. However, this may not necessarily hold if the base sampling hardware code is changed.

3.4 Serial EEPROM

The system uses a Microchip 24AA32 32 kilobit (4,000 bytes) I^2C serial EEPROM. This chip is used to hold the code for the processor. If the processor detects the serial EEPROM on the TWI when it boots up, then it loads the code from there; otherwise it has to be programmed via the JTAG interface. The configuration for the EEPROM is as follows:

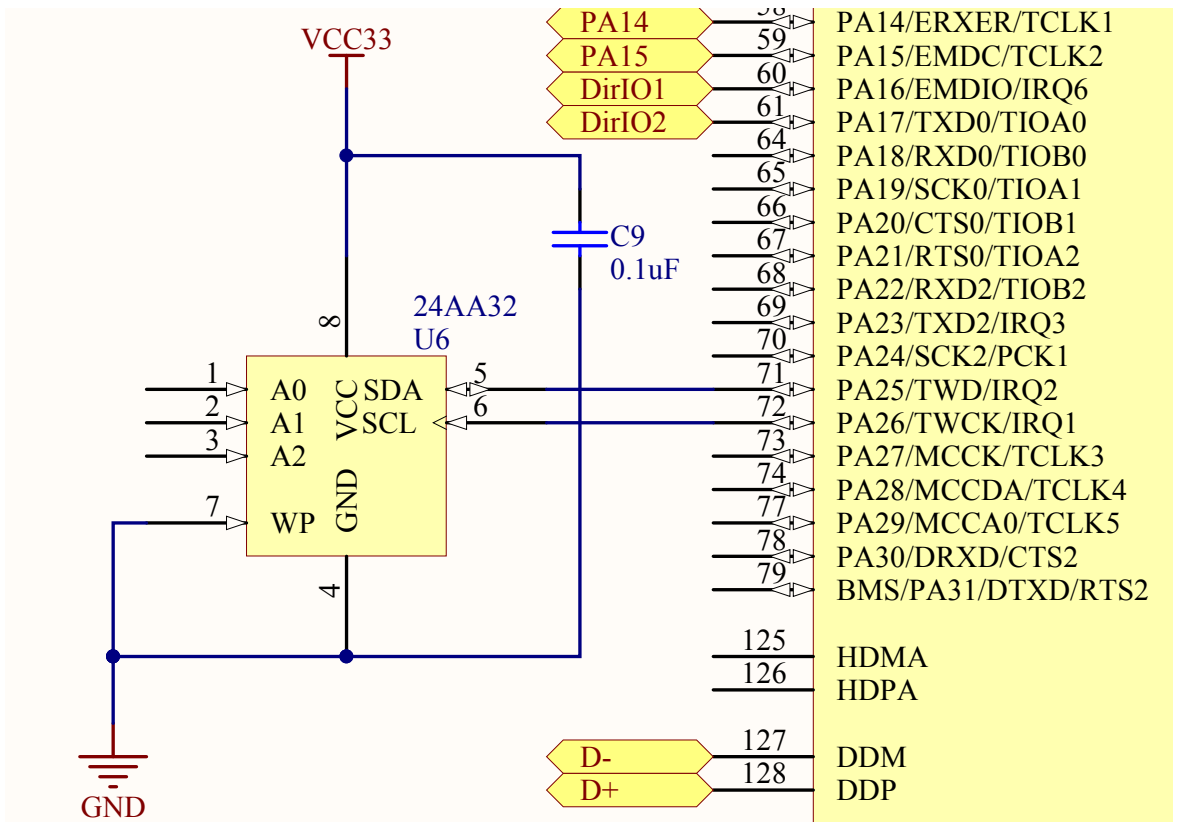


Figure 9: Serial EEPROM configuration

3.5 JTAG Interface

The JTAG interface is a 10×2 header that connects to a Wiggler board. The Wiggler board converts these lines to a serial connection that can be connected to a computer for programming the processor through a program like *OCD Commander*, *OpenOCD*, or others (none of these programs are particularly endorsed, just mentioned here for reference). The serial connection can also be

converted to a USB interface using specialized hardware. The header is configured as follows:

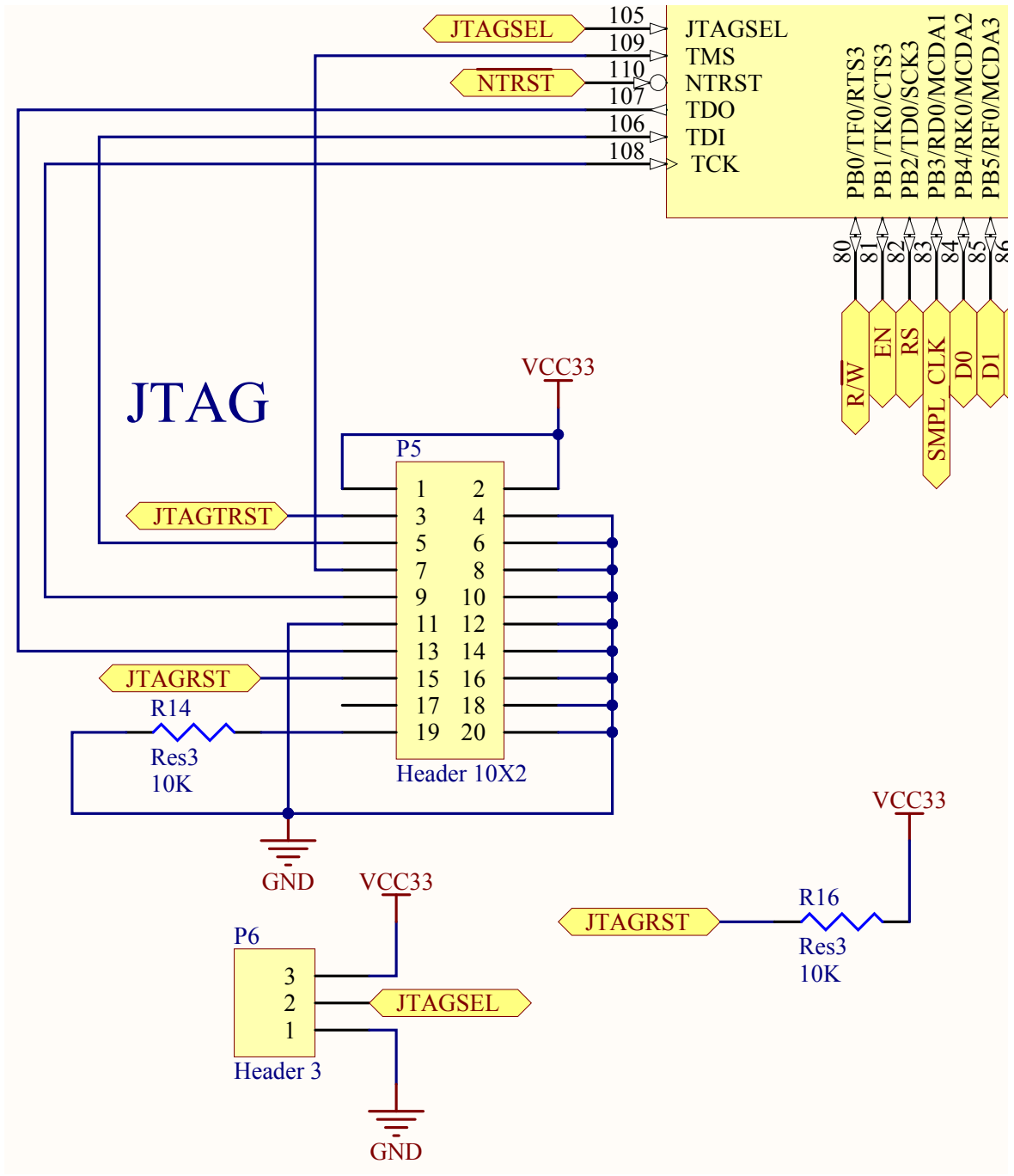


Figure 10: JTAG header configuration

3.6 Buffers

The system uses six Texas Instruments 74LVT162245 (pin compatible with the 74LVTH162245) buffers in total, labelled **U1**, **U2**, **U4**, **U5**, **U10**, and **U14**. A couple of these buffers are on the design in case extra I/O pins were necessary (or access to the address bus and data bus). Since the built system does not need to access these signals, these buffers can be left out in future designs, although this manual discusses their functions regardless. Note that in this design the output enable pins are always held low to continuously enable the output.

3.6.1 Address Bus Buffer [U1]

The address bus buffer buffers the first sixteen address lines of the processor. The direction pins of the buffer are set high because the address bus can only be an output of the processor. *This buffer was not used in the final design.*

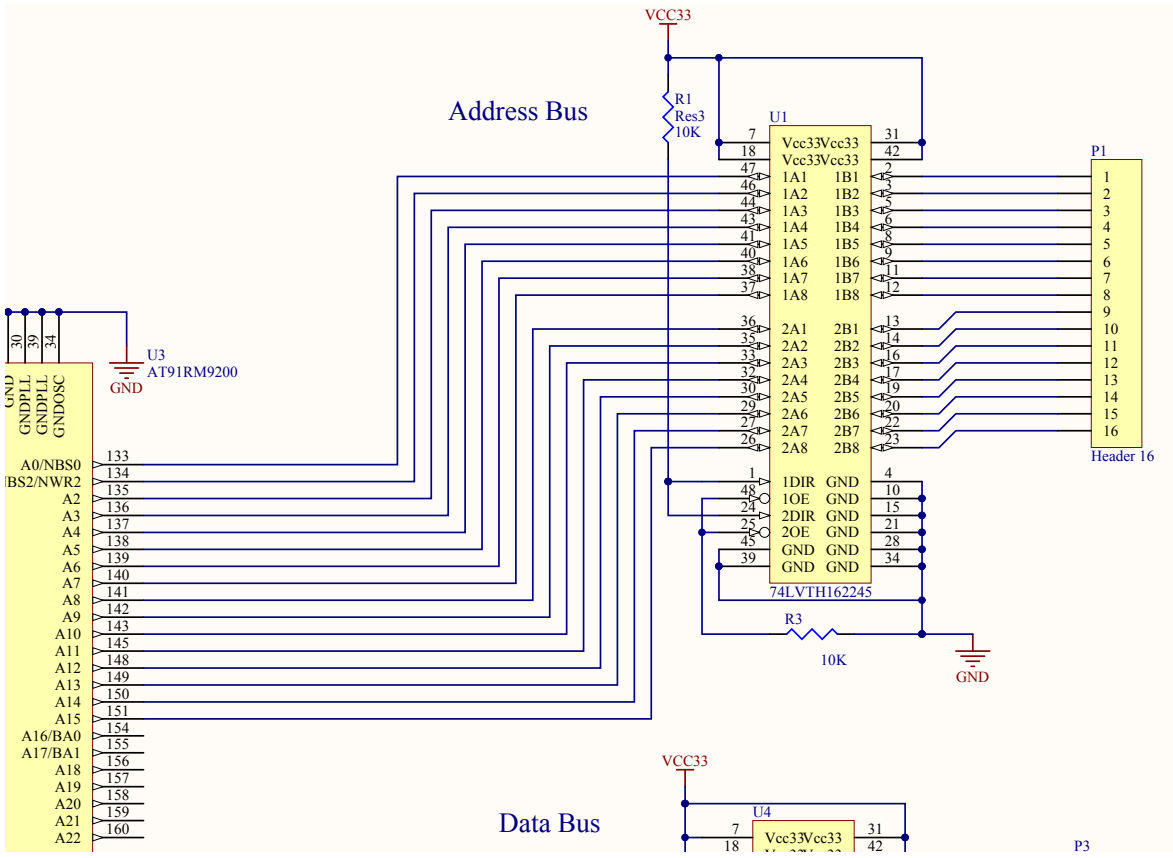


Figure 11: Address bus buffer configuration

3.6.2 "Other Output" Buffer [U2]

This buffer is for important outputs that may have been necessary to break out (in particular, chip select lines and read/write lines from the processor). *This buffer was not used in the final design.*

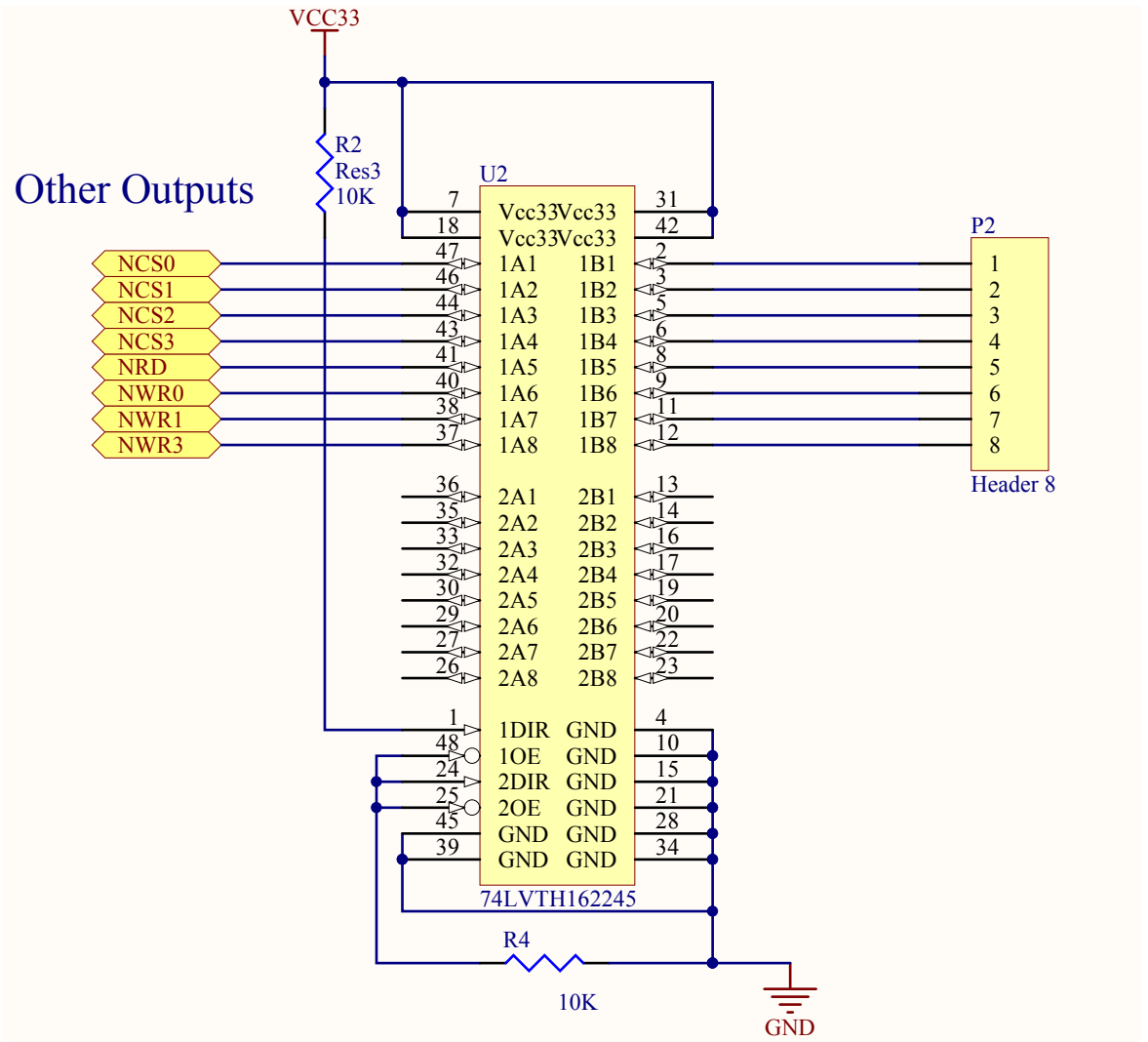


Figure 12: "Other output" buffer configuration

3.6.3 Data Bus Buffer [U4]

This buffer is for the first sixteen data lines of the data bus. As the data bus is generally bidirectional, the direction pins are controlled by the first R/W line of the processor (called NRD on the schematic). *This buffer was not used in the final design.*

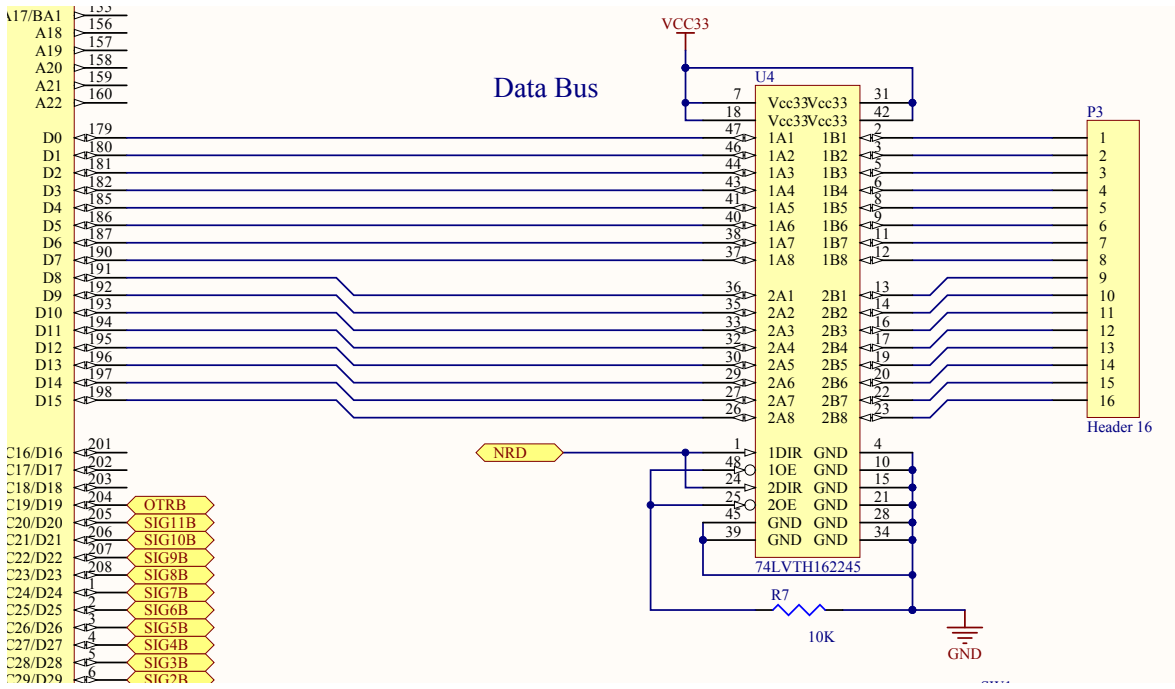


Figure 13: Data bus buffer configuration

3.6.4 General IO Buffer [U5]

This buffer is for the PIOA lines which were not originally used in the design. However, some of these lines became important to the operation of the system when it was built because of overlooks in the design, or because of problems that came up during the assembly of the project. The PIO A description under the *Processor* section of the *Detailed Hardware Description* details what the pins were used for in the final design.

The direction of the lines depends on the PA16 and PA17 pins in PIO A bank. The first eight pins are set to output (PA16 driven high) and the other eight pins are set to input (PA17 driven low).

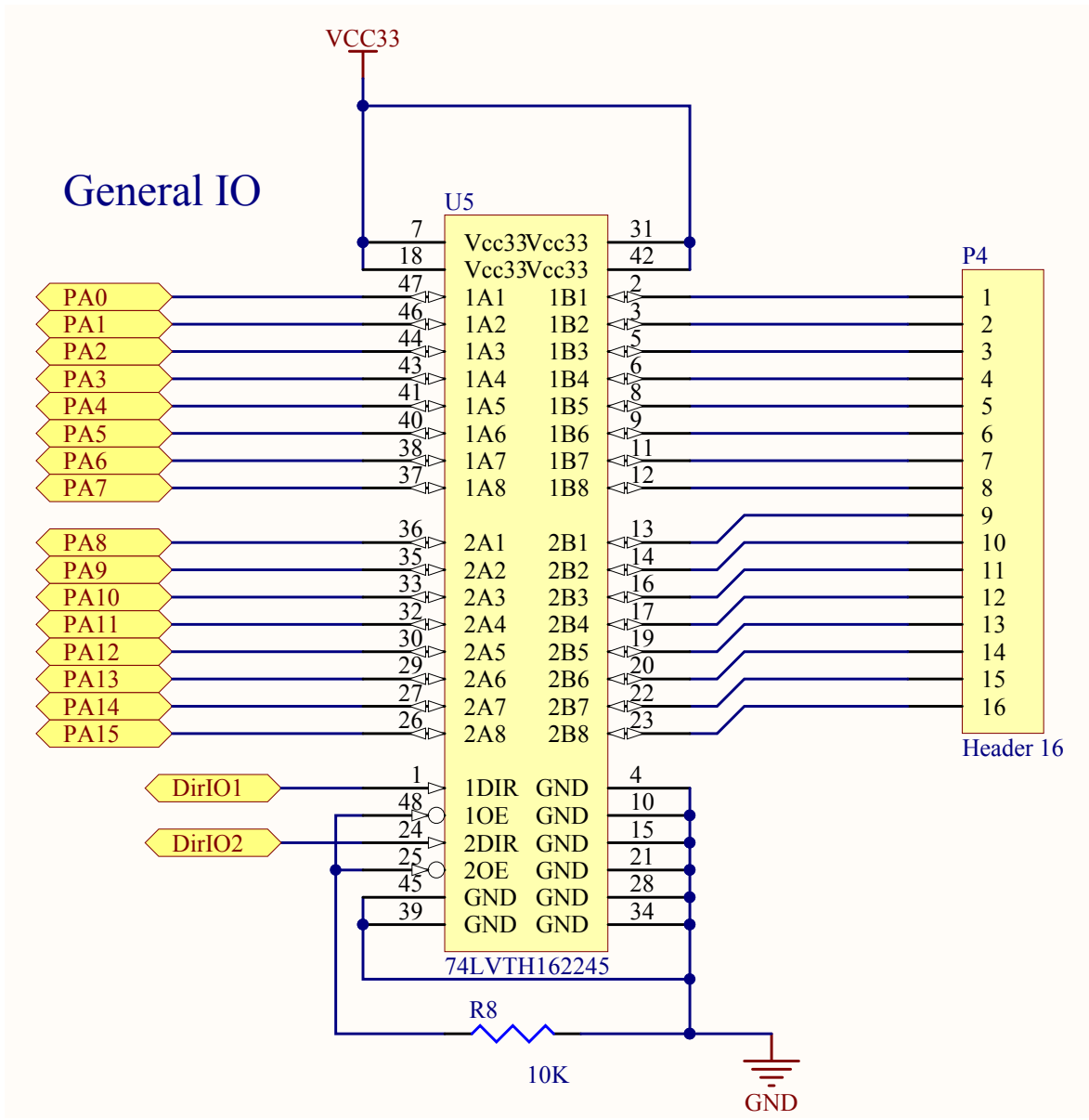


Figure 14: General IO buffer configuration

3.6.5 ADC Buffer [U10]

The ADC buffer was used in the old system to buffer the outputs of the ADS807 ADC. However, since this ADC configuration did not work (see the *Fixes, Notes, and Recommendations* section or the ADC subsection in the *Detailed Hardware Description* section for more details) this buffer is not used in the final design.

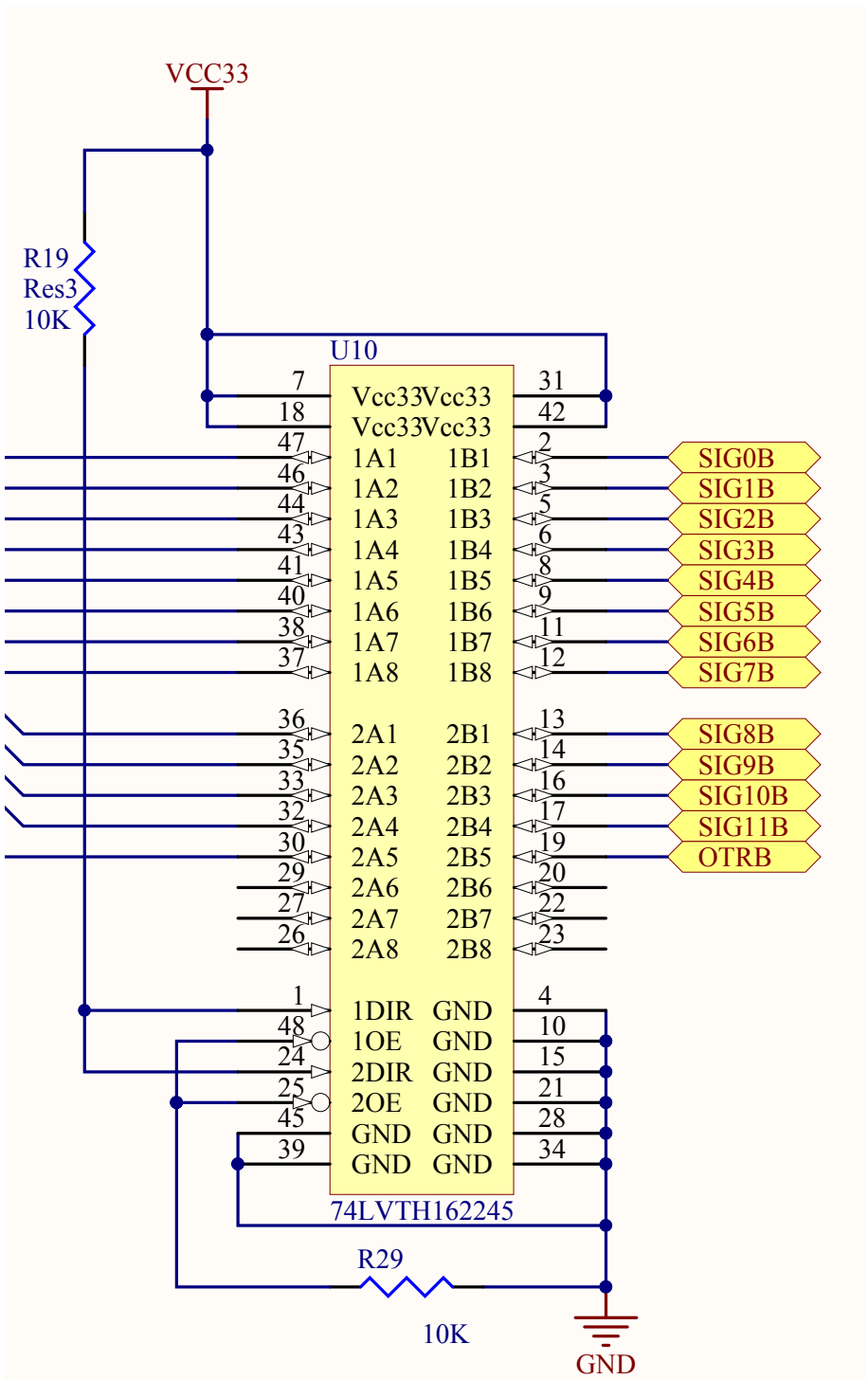


Figure 15: Original ADC buffer configuration

3.6.6 Display Buffer [U14]

This buffer is used for the character display lines. The control lines are always an output of the processor, so the direction pin is driven high. The data lines can be either inputs or outputs (although they are very rarely used as inputs to the processor), so the direction pin is set to the inverse of the R/W line sent to the display to guarantee that the direction is correct. The inverse *must be done in software*, it is not guaranteed by the hardware (i.e. the line is completely independent from the R/W line). The schematic shows a 'SMPL_CLK' line being used in this buffer, but this line was cut in the actual implementation.

The sample clock was re-routed using the General IO Buffer [**U5**] (see the previous *General IO Buffer [U5]* section for more information, in particular line PA1/**PCK0**).

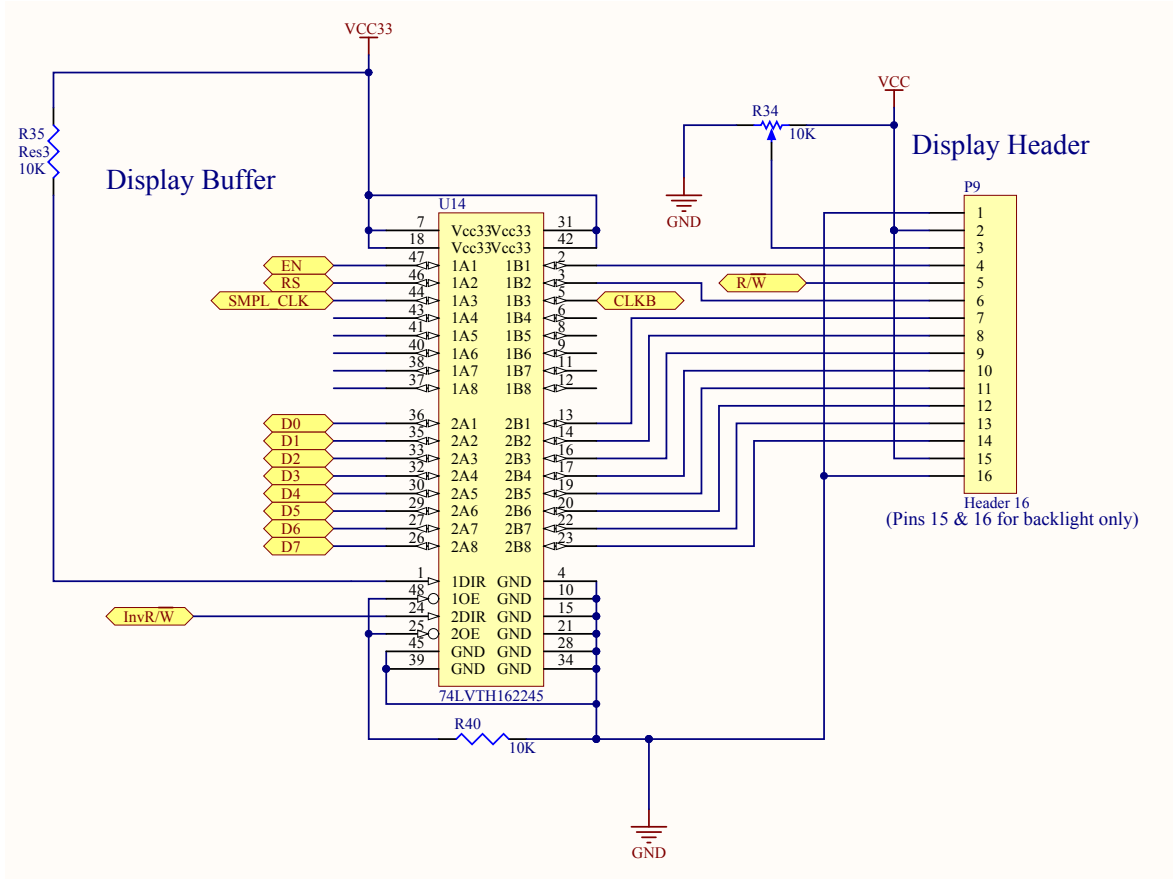


Figure 16: Display buffer and header

3.7 Power & Regulators

The system is fully powered from the USB port. The system, therefore, cannot expect more than 2.5 Watts of power, since a USB port on a computer is only guaranteed to deliver a maximum of 500mA at 5 V using the maximum power configuration. Note that when in suspend mode, the system receives only 100μA of power, which is not enough for it to function correctly. When the device is connected to a "sleeping" computer, therefore, it appears to be off.

The incoming 5 V (labeled **VCC** on the schematics) is regulated down to 3.3V (labeled **VCC33**) for powering 3.3V chips and 1.8V (labeled **VCC18**) for powering the processor. The system uses Texas Instruments LM1086 Adjustable Linear Drop-Out (LDO) regulators, where the output voltage depends on the values of resistors used according to the formula:

$$V_{out} = 1.25V\left(1 + \frac{R_x}{R_y}\right)$$

and R_y must be approximately 100 Ohms. The configuration for these is shown below:

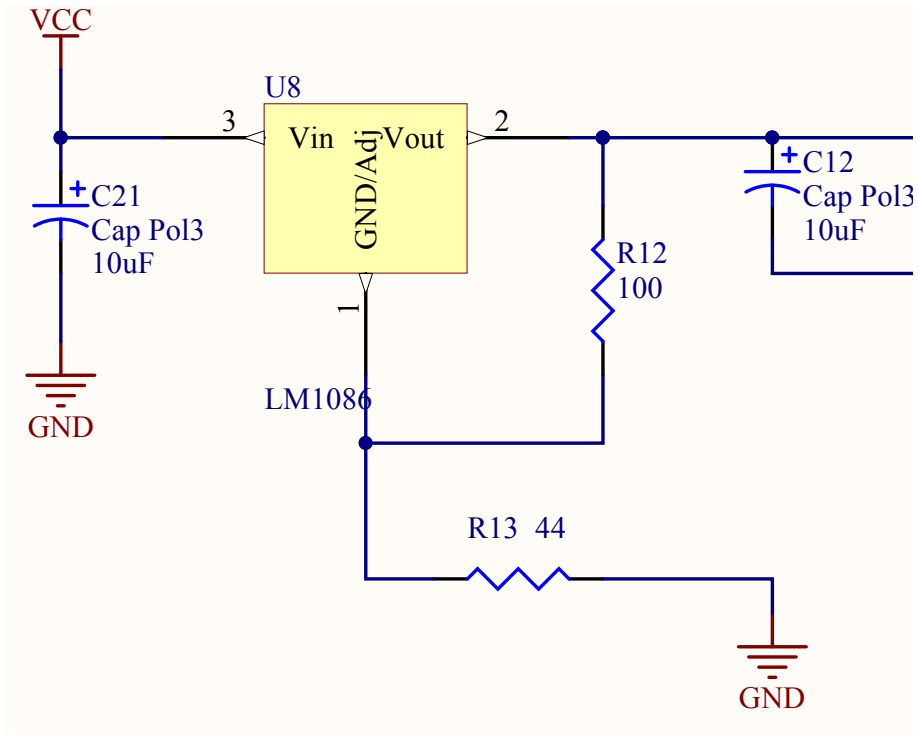


Figure 17: 1.8 volt regulator configuration

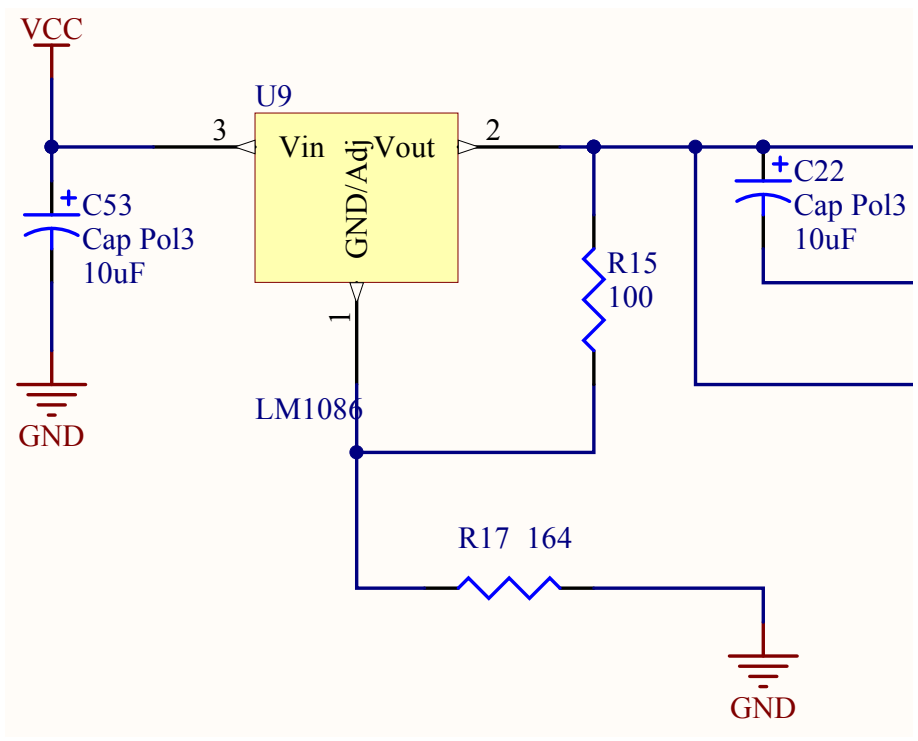


Figure 18: 3.3 volt regulator configuration

3.8 Crystals

The system uses two crystals to control the frequencies of the processor clocks. One of these is used for the *Slow Clock* which operates at 32.768kHz. The other ones is a 20MHz crystal used for the *Main Clock*. The crystals are stabilized using 20pF capacitors to ground as shown below.

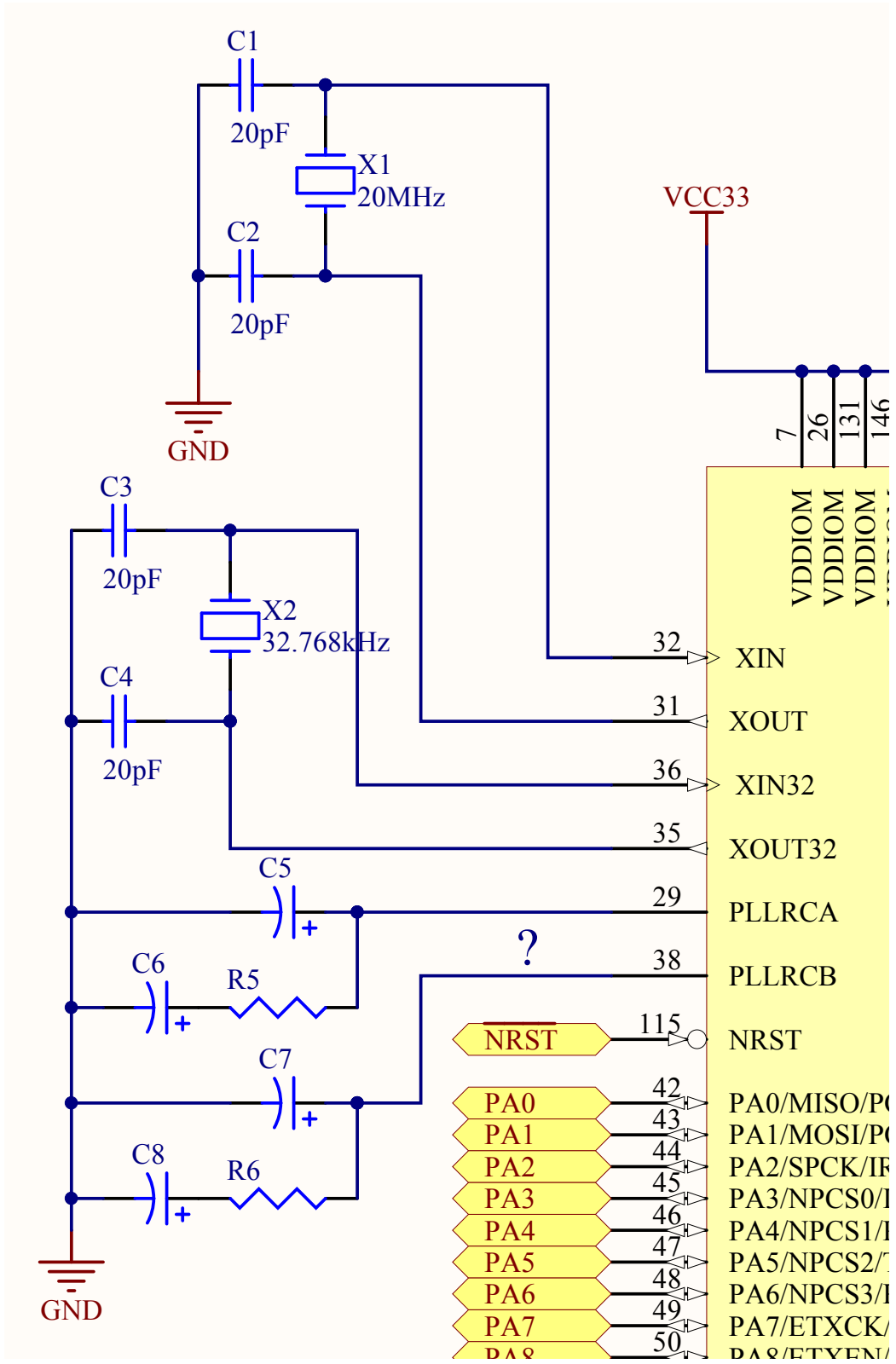


Figure 19: Crystal and PLL configuration

This diagram also shows the connections that need to be made for the PLL configuration discussed in the *Processor* section.

3.9 Reset Circuitry

The reset circuitry consists of a MAX706 chip [**U7**] connected to a red pushbutton switch, which, when pressed, sets the lines NRST and NTRST low to reset

the system. The powerfail and watchdog features on the chip were disabled for simplicity. The configuration is as follows:

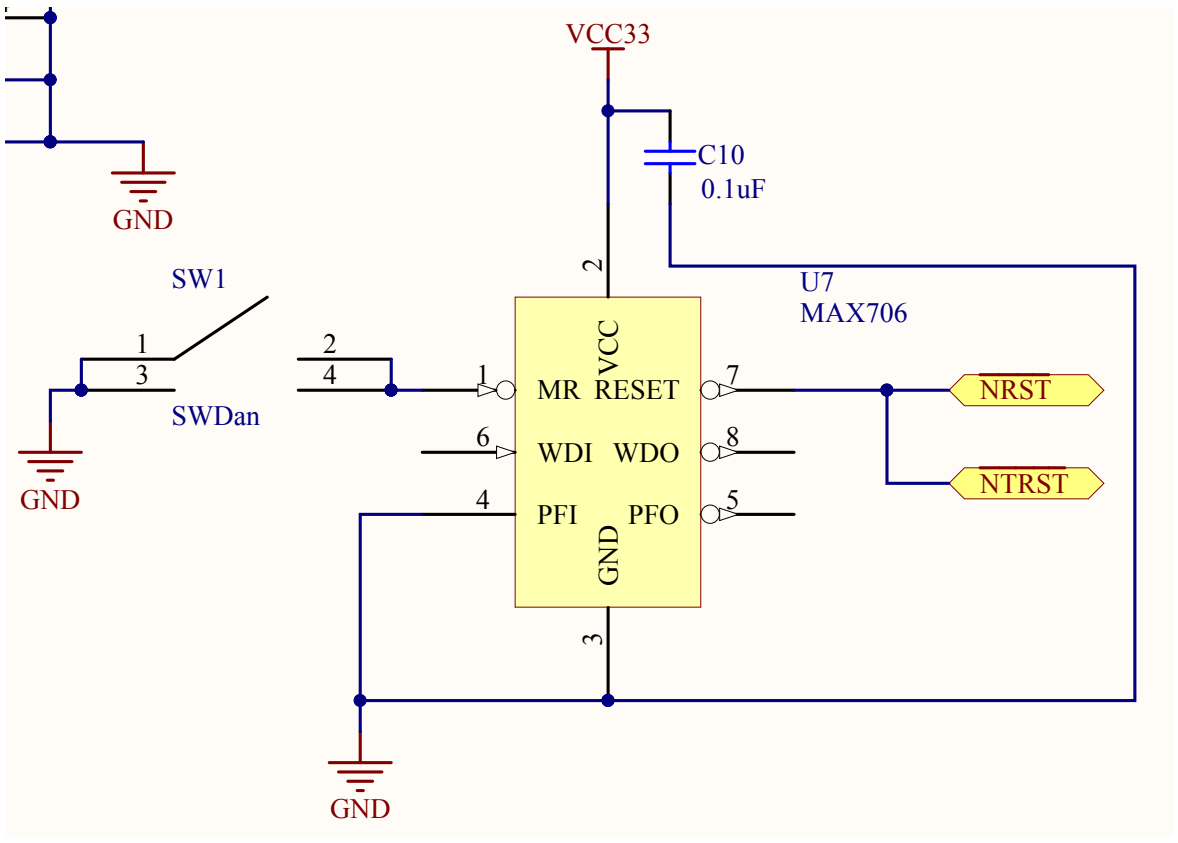


Figure 20: Reset circuitry configuration

3.10 Character Display

The character display is a 16 × 2 HD44780-compatible 5 volt display. The contrast is set by a 10kΩ potentiometer. The display controller is fairly easy to operate. The chip differentiates between "commands" and "data" using the RS line - when RS is high, the data driven on the data lines corresponds to a certain command, and when the RS is low, the data driven on the data lines corresponds to an ASCII character to be displayed. The enable line (EN) is used to send a command or character - at the beginning of a cycle, EN is driven high, then the data lines are driven according to what must be sent to the display, and, finally, EN is driven low again to actually send the data. This is all done in software, see the relevant section in the *Detailed Software Description* for more information. The display header (16×1) schematic, along with its corresponding buffer, were shown in the *Display Buffer [U14] section*, but are reproduced below for convenience.

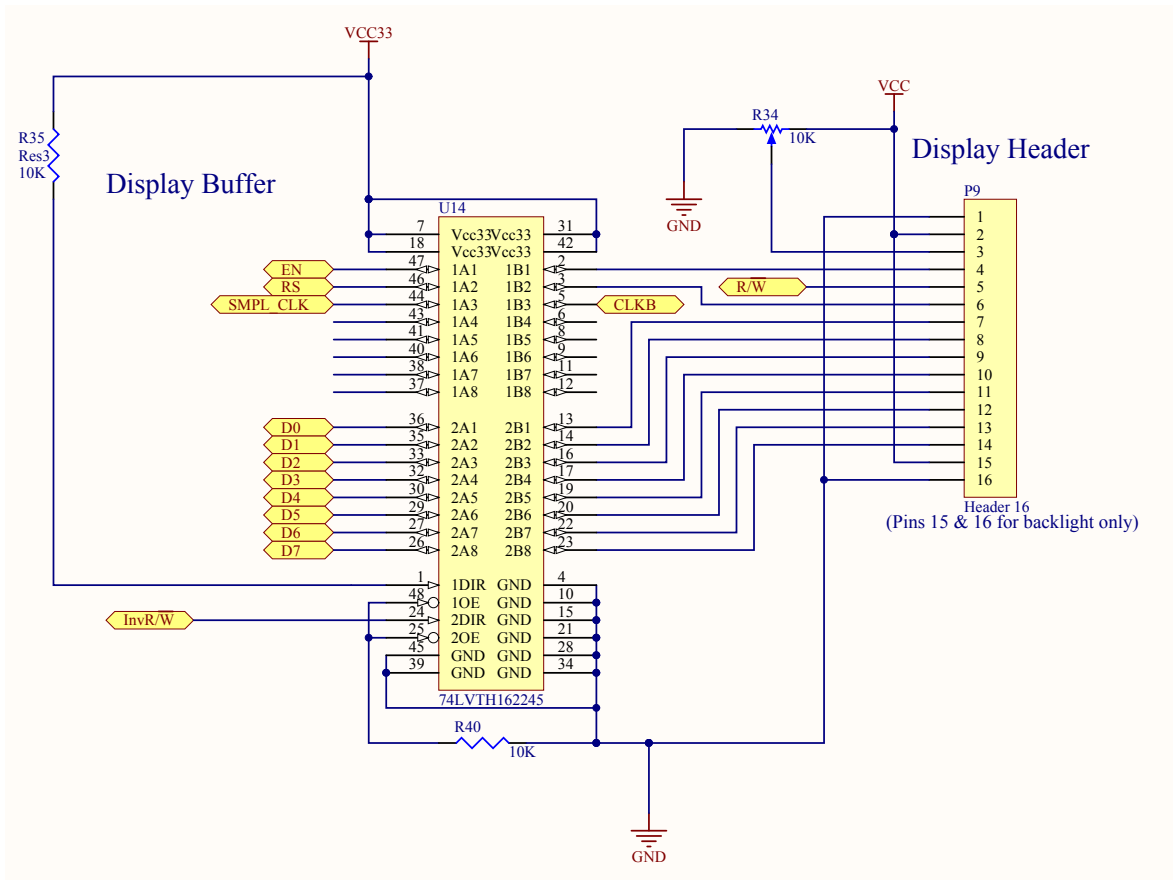


Figure 21: Display buffer and header

For a full description of how the lines are connected to the PIO pins, please see the PIO section in the *Processor* subsection of the *Detailed Hardware Description*.

3.11 Pushbuttons

The system has two pushbuttons, labelled [**SW1**] and [**SW3**]. The reset switch [**SW1**] was discussed in the *Reset Circuitry* section and warrants no further discussion. The debug switch [**SW3**] is configured as a simple pushbutton switch with a 1kΩ pull-up. It is debounced in software, please see the *Detailed Software Description* section for more information. The schematic for the switch is shown below (note that the line is not buffered before going into the processor).

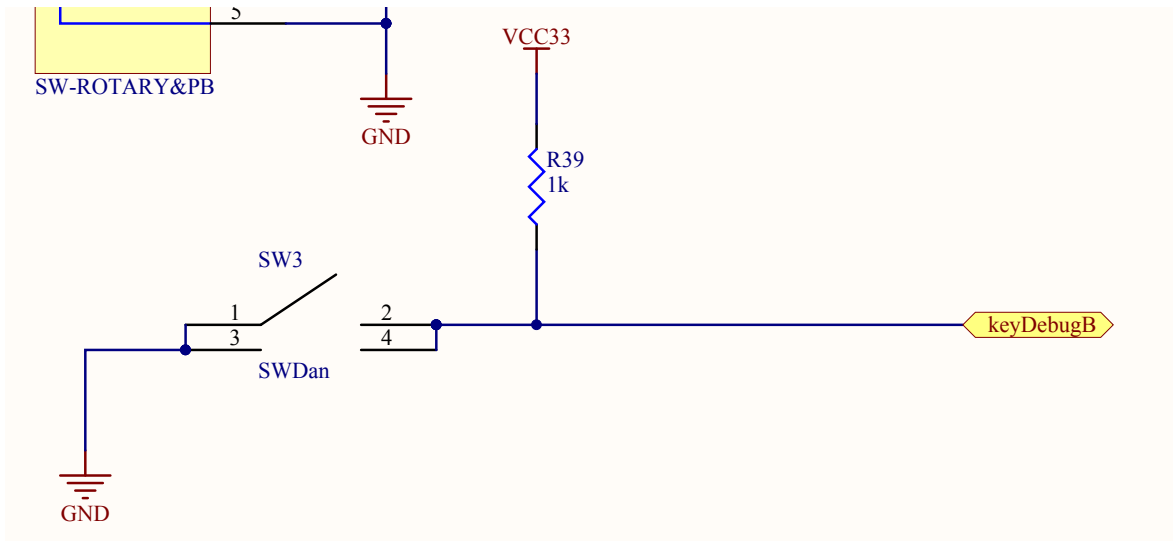


Figure 22: Debug switch configuration

3.12 Rotary Encoder

The rotary encoder has simple quadrature output in lines A and B, as well as a pushbutton which is used to toggle what is shown on the display. The quadrature output decoding, as well as the debouncing of the pushbutton, are done in software, please see the *Detailed Software Description* section for more information. None of these lines are buffered. The schematic is shown below.

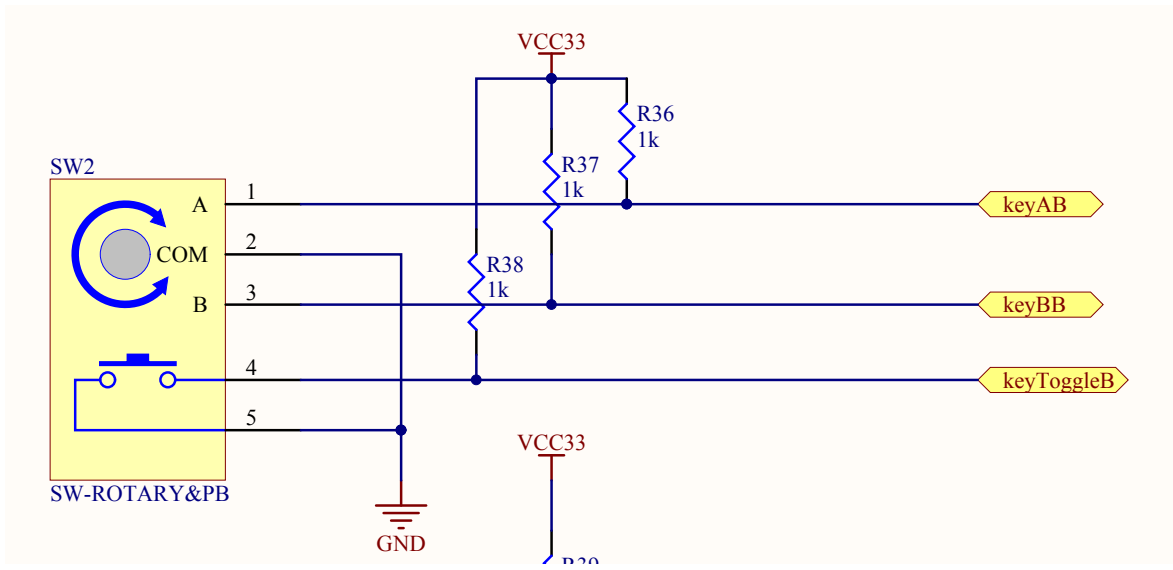


Figure 23: Rotary encoder and toggle switch configuration

4 Software Overview

4.1 Main Program Flow

Entry Point After being loaded (from JTAG or from the serial EEPROM), the code starts running at address $0x00200020$ (internal SRAM with 8 words for the IVT). The low-level initialization of the processor takes place here (more on this in the *Detailed Software Description*) and then the `main` function is called, which contains the main loop.

Main Loop The main loop loops forever and does the following each time it loops around:

1. Update the info on the display if enough time has passed since the last update.
2. Check whether any keys have been pressed, and handle them if this has been the case.
3. Check if the FIFO is at least half-full, and, if so, read the data from the FIFO and update the pertinent variables.

4.2 Interrupts

There is only one interrupt in the system. This interrupt corresponds to the first pin of PIO A, which is a programmable clock (PCK3). This programmable clock is set to the lowest possible frequency, the slow clock divided by 64, or 512 Hz. Since the PIO A interrupts fire whenever there is a change in the input, this corresponds to interrupts which happen at 1024 Hz, or approximately once every millisecond.

This interrupt is used as a timer interrupt for the entire system. It is responsible for the following:

1. Debouncing keys (checking to see if input is low for a sufficient amount of time)
2. Multiplexing the display (if currently sending data to it)
3. Keeping track of a crude sense of time to tell when to update the values shown in the display

If a better sense of time is necessary (more resolution), it may be more useful to count the times that the second pin of PIO A changes (PCK0), or the sample clock. One has to be careful with this because it's a variable clock, however, and its frequency depends on what the user has selected using the user interface. However, this is probably the best way to keep track of in and out times of bubbles, since the required resolution is in fact dependent on the sample rate,

because one would naturally set the sample clock to a faster frequency if one expects smaller bubbles.

USB Interrupt Note that the USB code does use an interrupt to call the processor's attention, but that this code has not been included in the main body of code yet.

5 Detailed Software Description

5.1 Processor intialization

This section describes the code used to set up the more general functions of the processor, including the clocks, the interrupt unit, the power management unit, and the PIOs. The code can be found in the following files.

Source files: `init.s`, `crt0.s`, `handlers.s`
Include files: `PMC.inc`, `PIO.inc`, `aic.inc`

5.1.1 Low-level initialization

The code found in `crt0.s` is responsible for all the low-level initialization necessary for the system. This includes setting up the IRQ table for low-level interrupts, the stack, the ARM processor running mode, and calling the other functions described in this section.

5.1.2 Clock and power setup

The function `init_power` is responsible for setting up the Power Management Controller (PMC), which controls the main clock and powers the peripherals. This function configures PCK0 to act as the sample clock and PCK3 to change at 512 Hz which gives the clock interrupt. The following peripherals are explicitly powered in the code (some peripherals, such as the TWI and UDP, are automatically powered at boot time).

- PIO A (power needed for interrupts, reading FIFO data)
- PIO B (power needed for button data, reading display data)

5.1.3 PIO initialization

The PIOs are set up by writing to processor registers. This initialization follows directly from the purpose of each PIO pin, which can be referenced in the *Detailed Hardware Description* section if necessary. This setup occurs in the functions `init_pio_a`, `init_pio_b`, and `init_pio_c` and is straightforward enough that no further explanation should be necessary to understand it well.

5.1.4 Interrupt initialization

The Advanced Interrupt Controller (AIC) is set up in the function `init_interrupts`. This sets up the interrupt types and vectors, and it's important to note that interrupts have to enabled their relevant sections as well (so, for example, PIO interrupts have to be enabled in the PIO code as well). The only interrupt currently enabled is the PIO A interrupt, which is handled by the function `clock_interrupt` in the file `handlers.s`. This interrupt serves as a timer interrupt to update the display and check for (and debounce) key presses.

5.1.5 Other initialization

The functions `init_fifo` and `ui_init` are called to set up their respective variables and hardware. This is described in more detail in the relevant sections (*User Interface* and *Analog Code*) below.

5.2 Character Display

This section describes the code responsible for interfacing with the character display. The code can be found in the file `display.s` and the include file is called `display.inc`.

Important Constants The following constants are used throughout the display code and their nature warrants further explanation than that given in the comments of the code.

Constant	Description
`firstData`	First command sent to display to initialize it. See *Command Descriptions* for more details.
`secondData`	Second command sent to display to initialize it. See *Command Descriptions* for more details.
`thirdData`	Third command sent to display to initialize it. See *Command Descriptions* for more details.
`RES_MEM_COMM`	Command sent to display to move cursor back to the initial position (after writing).
`DISP_CNT_LONG`	Number of calls to `display()` to wait before sending a command (stricter timing requirements than characters).
`DISP_CNT_SHORT`	Number of calls to `display()` to wait before sending a character.

Table 6: Important display constants

Command Descriptions The following commands are sent to display to initialize it correctly for this particular system and to reset the cursor when the display is fully written.

Command (Byte)	Constant Name	Description
0011 0000	`firstData`	Sets display to 1 line, 8 bit data, and 5x7 characters
0000 1100	`secondData`	Turns display ON and enables the cursor
0000 0110	`thirdData`	Sets entry mode, tells cursor to automatically increment when a new character is sent, and disables the display shift
0000 0010	`RES_MEM_COMM`	Sets cursor position back to zero, used when the full contents of the display buffer have been displayed.

Table 7: Commands sent to character display, in order sent

Variable Descriptions The following table describes the shared variables used in the display code.

Variable	Description
`comm_count`	Keeps track of which command in the command array needs to be sent next.
`data_counter`	Keeps track of which character in the character array needs to be sent next.
`enable_state`	Whether enable is currently `LOW` or `HIGH`.
`sent_data`	Whether data has been sent this cycle (`TRUE`) or not (`FALSE`).
`disp_count`	Counts calls to `display()` for timing purposes.
`comm_arr`	Array of commands to be sent to display.
`disp_buffer`	Array of characters to be sent to display.

Table 8: Display variable descriptions

Code Description The code for interfacing with the character display is separated into three different functions. The first of these, `set_display(char* s)`, takes a pointer to an ASCII string and updates the local buffer that stores what is shown on the display. If the passed string is longer than sixteen characters, it is truncated to sixteen characters. As it is a fairly simple function, it should need no further explanation here.

The second function, `display()`, is called by the previously discussed `clock_interrupt()` function at a frequency of 512 Hz. This function is responsible for making the code meet the timing requirements of the display, as well as

actually updating the display in a timely fashion. The function is essentially a small state machine that does the following:

1. If it's not okay to send the next command or character (breaks timing), then stop for now (this is done by comparing the variable `disp_count` to DISP_CNT_LONG/SHORT)

2. Otherwise, if enable is low, set it high and stop for now

3. Otherwise, if enable is high and we've set the data pins, set enable low and stop for now (this actually sends the data to the display at this point)

4. Otherwise, set the data pins by calling the helper function `set_display_pios()`

Doing these things one step at time ensures that the timing requirements of the character display are met, although it is not denied that doing it this way might be excessively cautious. This function is also responsible for determining whether the next data command sent should be a command or a character. This is determined by checking the value of the variable `comm_count` which keeps track of which command to send next. If the value of this variable is larger than `COMM_LEN`, the function sends the next character in the buffer. Otherwise, it sends the command stored at `comm_arr[comm_count]`.

The final function, `set_display_pios(rs_val, data, enable_val, only_enable)`, is a helper function that will set the PIO pins according to the arguments passed in. As it can be helpful to only set the enable pin (in particular when enable is set low to send the data that was loaded in the previous cycle), if the argument `only_enable` is set to `TRUE`, the code will ignore the other pins and only set the enable pin to the value passed in through `enable_val`.

5.3 String Manipulation

This section describes the functions used to manipulate strings to show variables in the user interface. This code can be found in the file `converts.s`.

Important Constants The following constants are used throughout this code. They can be found in the file `converts.inc`.

Constant	Description
`ASCII_ZERO`	ASCII representation of the number zero (0)
`ASCII_NULL`	ASCII representation of the null termination character
`MAX_DIGIT_NUM`	Largest power of ten that fits in a 32 bit register
`RIGHT_JUST_NUM`	Amount to shift a string address by in order to right-justify a ten digit number on a line of the 16 character display.

Table 9: Important string manipulation constants

Variables This section has no variables.

Code overview This part of the code consists of the function `dec2string(value, str_pointer)` and its helper function `divu(dividend, divisor)`. The function `dec2string` is responsible for determining the ASCII representation of each digit of a (unsigned, base ten) number and storing the digits in memory (presumably in a buffer of characters, but there is no check for this). In order to do this, the function has to do repeated divisions by ten. Since the ARM instruction set does not include a divide instruction, the function `divu` is responsible for taking two number and performing binary long division, returning the quotient and remainder.
Note: The function `divu` is not C-callable because the arguments are in `r0` and `r2`, which is not standard. C probably allows direct division, though, so no problem.

5.4 Keypad

This section describes the code for interfacing with the keys and rotary encoder.

Important Constants The following constants are used throughout the keypad code and their nature warrants further explanation than that given in the comments of the code. The enumerated type `KEY` (that is, `KEY_NONE`, `KEY_TOGGLE`, `KEY_DEBUG`, `KEY_LEFT`, and `KEY_RIGHT`), serves to map key names to integer values so that the keys can be used in tables and so forth.

Constant	Description
`DB_LEN`	How many interrupts (`clock_interrupts`) to wait before a key is considered debounced.
`CP_LEN`	How many interrupts (`clock_interrupts`) to wait before a key is considered pressed again if held down.

Table 10: Important keypad constants

Variable Descriptions The following table describes the shared variables used in the keypad code.

Variable	Description
`key_counter`	Counts interrupts to check if a key is debounced or held down.
`last_key`	Keeps track of the last key pressed to check whether it's the same key being pressed or a new key.
`cur_key`	Currently pressed key (used for returning which key has been pressed in `get_key()`, if any).
`key_pressed`	Boolean to check if any key has been pressed since last call to `get_key()`.
`prevA`	Previous value of the rotary encoder A quadrature output, needed to decode the output.
`prevB`	Previous value of the rotary encoder B quadrature output, needed to decode the output.

Table 11: Keypad variable descriptions

Keypad code overview The keypad code consists of five functions. Of these, two (`get_key` and `key_available`) are high-level functions for interfacing with the rest of the code, one (`keys`) is part of the `clock_interrupt` handler, and the other two (`encoder_key` and `key_press`) are helper functions for `keys`.

Helper functions The function `key_press()` is responsible for checking the PIO B corresponding to the toggle key and debug key and determening whether

a key is pressed (the line is low). The function `encoder_key()` is responsible for determining whether the encoder was rotated, and in which direction the user rotated the encoder. This is done by decoding the quadrature output on lines A and B; the logic for this was blatantly (yet with permission) taken from Daniel Andrade's open-source EE/CS 52 *DS-94 Digital Oscilloscope* project and translated into code. Further explanation is omitted as this is a fairly standard operation.

Please note: the order in which the keys are checked is important, as the system does not support multiple key presses. The keys are prioritized in descending order (with the exception of `KEY_NONE`). Therefore, the priority of the keys, in descending (highest priority first) order, is:

1. Toggle key
2. Debug key
3. Left
4. Right (...one would hope this is mutually exclusive with *Left*... but just in case...)

Handler function The function `keys()` gets called by the `clock_interrupt` interrupt handler. This is the entry-point of the detection part of the keypad code. The first thing the function does is to call the helper function `key_press()` to check whether the PIO line for a "normal" key (not an encoder rotation) is being pressed. If so, then the key is compared against a debounce counter to see if it is debounced. If not, then the encoder is checked by calling `encoder_key()` (the encoder does not need to be debounced).

High-level interface code The function `key_available()` should be polled by the main code to check see if a key has been pressed since the last one was handled (in particular, since `get_key()` was last called). The function `get_key()` returns which key it was that was pressed. Please note that `get_key()` is not meant to be called unless `key_available()` returns `TRUE` and that, once `key_available` returns `TRUE`, `get_key()` should be called as quickly as possible to avoid missing a key press.

5.5 User Interface

This section describes the user interface code. The purpose of this code is to call the display and keypad functions appropriately so that the system behaves as expected when the user interacts with it. The code can be found in the file `ui.s`.

Important constants The only user interface constants (besides obvious ones which hardly need to be explained) used are the ones in the file `adc.inc` which describe mappings from the frequencies displayed to the values that need to be written to the programmable clock registers.

Variable descriptions The following table describes the shared variables used in the keypad code.

Variable	Description
`KeyFuncArray`	Maps key values to the function used to handle that key.
`freqArray`	Constant array used to store which sampling frequencies the system can support.
`debug_state`	Boolean that indicates whether the system is in debug state or not.
`toggle_state`	Integer which indicates which screen is currently being displayed.
`rate_cntr`	Pointer to the frequency array used to determine which sampling frequency to display.
`freqPCK0Arr`	Table for holding the values to write to the programmable clock register to set the sample rate.
`disp_str_x`	Constant string to be displayed (some are altered by `dec2string` to update displayed numbers).
`void_rate`	Current measure of the void rate (for the last `NUM_AVG_SAMPLES`)
`vf_counter`	Counter of the 512 Hz clock used to keep track of time for auto-updating the display.

Table 12: UI variable descriptions

Code overview The function `ui_init` sets up the string variables for the display so that the system doesn't have to wait for the periodic updates to show something on the screen.

The main loop waits for a key to be pressed by continuously calling `key_available()` - once this happens, the code enters the UI section through the function `ui_do_key()`. This function uses the function table `KeyFuncArray` to figure out which function to call to handle the key pressed by the user. At the risk of being excessively obvious, the functions are:

Function Name	Description
`do_toggle()`	Handles the user pressing the toggle key
`do_debug()`	Handles the user pressing the debug key
`do_left()`	Handles the user rotating the encoder to the left
`do_right()`	Handles the user rotating the encoder to the right

Table 13: Brief description of the function table

The functions are well defined in the code and do what the *Functional Specification* specifies when the keys are pressed, so please refer to that for a more thorough description.

Variable control functions There are also getter and setter functions that provide access to the system debug state so that it can be accessed from other parts of the code. The function names are:

- `get_debug_state`
- `do_debug` (can be used to toggle the debug state)

5.6 Sampling Hardware

This section describes the code to interface with the incoming analog data. The code is written in the file `analog.s`.

Important constants The constants for the analog interface code can be found in the files `adc.inc` and `fifo.inc`. Most of the constants are used in the user interface code to map desired sample rates to the values that need to be written to the programmable clock registers to get a clock of this frequency. The important constants are presented in the table below.

Constant	Description
`ADC_DATA_MASK`	Mask to get incoming data from the FIFO from the PIO A pin values.
`ADC_DATA_SHIFT`	How much to shift the incoming data by to make it an actual 7 bit value.
`CLOCKS_FIFO_WAIT`	How many clocks to wait when holding RESET low (to meet timing requirements for FIFO reset).
`NUM_AVG_SAMPLES`	Number of samples to read from the FIFO at one time. Right now the void rate is also calculated from this number, but that is not strictly necessary (could be kept in memory and then use a larger number).
`ADC_TEST_VAL`	Starting threshold value and value used for threshold when in debug mode.

Table 14: Important analog interface code constants

Variable descriptions All of the variables in the analog code are used to calculate the averages and threshold. These variables are:

Variable	Description
`high_val_accumulator`	Accumulator value in memory used to add all the high samples together to later average
`low_val_accumulator`	Accumulator value in memory used to add all the low samples together to later average
`num_high`	Current number of accumulated high samples
`num_low`	Current number of accumulated low samples
`high_average`	Latest average of the high samples
`low_average`	Latest average of the low samples
`threshold`	Current threshold value

Table 15: Analog code variable descriptions

Code overview The function `init_fifo` will reset the FIFO memory by pulsing the RESET line. This is called on system boot-up only, and the timing is described in the *Detailed Hardware Description* section.

The rest of the analog code consists of six functions. One of these, `get_data_available()`, is a getter function which the main loop constantly polls to determine if it should start reading the FIFO. This function checks the value of the half-full flag output of the FIFO. This function returns `TRUE` when this pin is low.

When the main loop receives `TRUE` from the above function, it immediately calls `adc_process_fifo()`, which is probably the most strangely-named function ever written. This function is responsible for reading `NUM_AVG_SAMPLES` from the FIFO consecutively and returning how many of these samples were higher than the threshold. This is a measure of the void fraction of the fluid (or, equivalently, the duty cycle of the square wave). To do this the function has to meet the timing requirements of the FIFO memory, which were detailed in the *Detailed Hardware Description* section.

The function `get_adc_data()` is a helper function that reads the values of the appropriate PIO A pins and returns the 7 bit value that the FIFO is currently outputting. It is called by `adc_process_fifo()`.

The simple getter functions `get_threshold()` and `get_averages()` return the appropriate variables in memory.

Finally, the helper function `update_averages(threshold, sample)` updates the value of the accumulators and (occasionally) the averages and threshold. The function is set up to accumulate the first 1024 (a factor of two so that the division can be performed by a right shift) samples into the appropriate accumulator (high or low). Once 1024 samples have been calculated, the average of these samples is computed. Finally, when both averages have been computed (more than or equal to 1024 samples in each accumulator), the new threshold is computed by taking the difference of the two averages, halving it, and subtracting it from the high average, thus choosing the midpoint of the averages as the threshold. After this is done, the accumulators and sample counters are set back to zero to begin the process anew.
Note: The function `update_averages` is not C-callable because it takes arguments in `r0` and `r5`, which is nonstandard. This allowed for easier integration with `adc_process_fifo`, but would have to be changed if it is desired to call this function within C code.

6 Functional Specification

6.1 Global Variables

There are no global variables (although there are global functions). Most functions are global functions, unless they are helper functions intended to only be called within that file. Global functions are prefaced with the `.global` directive in the code.

6.2 Inputs

The analog waveform is as described in the overview. The signal comes out of a box which takes a probe as an input and outputs through a BNC connector. Thus the input to the embedded system comes through a BNC connector as well. The input can be scaled arbitrarily, so the input range of the ADC is a nonissue, but the input may have to be calibrated every so often in order to ensure that it stays within the range of the ADC.

6.3 Outputs

Status information and the results of calculations are sent to a 16x2 character display. Status information is updated on the display as soon as the update happens (as an example, when the debug button is pushed, a variable in memory changes and `set_display` is immediately called if the state of the debug variable is currently being shown to the user). The results of calculations are updated at a constant rate that depends on the constant `VF_COUNTER_MAX`, which is the number of PCK3 clocks the processor should count between each automatic display update. As of this moment the value of this counter is 2048, meaning that the display updates automatically every two seconds. The strings updated correspond to:

- Dynamic threshold value (*even if in debug mode*)
- Void rate
- Signal high and low averages

The following only update when there is a change in the user interface code, so the update display function can be called directly afterwads:

- Sampling frequency
- Debug variable

6.4 User Interface

The system is controlled with a rotary encoder and two buttons. The rotary encoder is used to change the sampling frequency or other variables on screen, the first button is a simple toggle button that toggles what is shown on the

display (one numbered option below is shown at a time and the button toggles between them), and the second button toggles debug mode. The user is able to see status information on the 16x2 character display (see description in *outputs* above).

Screens The device currently supports the following user interface screens. These screens are shown in this order to the user, and the user can switch between them using the toggle button. The screens wrap around, so that when the toggle button is pressed and the last screen is shown, the device switches to the first screen again. The screens are, in order:

1. Low average screen
2. High average screen
3. Threshold screen
4. Sample rate screen
5. Void rate screen
6. Debug screen

The table below details what exactly is displayed in each screen, as well as what the effect of the rotary encoder is on each screen:

Screen	Displayed	Variable	Encoder Action
Low average	`AV_LO:0000000xxx`	signal low average	none
High average	`AV_HI:0000000xxx`	signal high average	none
Threshold	`THR: 0000000xxx`	threshold	none
Sample Rate	`RATE: 00xxxxxxxx`	sample rate (in Hz)	change sample rate
Void Fraction	`VOID: 0000000xxx`	void fraction	none
Debug	`DEBUG ON` *or* `DEBUG OFF`	debug variable	toggle debug variable

Table 16: Screens shown on the 16x2 character display

There is also a welcome screen that displays the string "`Bubbly Measure!`" on device startup until the toggle key is pressed.

Display Output Interpretation The average values and threshold are expressed as seven bit values, with zero (0) being the lowest possible value and one hundred and twenty-seven (127) the highest possible value for these variables. The void rate is shown as a number from zero (0) to one thousand (1000). This number can be easily turned into a percentage by dividing by ten, e.g. a void rate of 556 is equivalent to a 55.6% high duty cycle.

6.5 Error Handling

There is currently no error handling in the system at all. Reference the USB code on page 169 and the FIFO fifo full indicator in section 7.3.5 for ideas.

6.6 Algorithms/Data Structures

There are no algorithms that are so complex that they should be especially mentioned here; the only algorithms present are relatively simple. The algorithm for converting a number to a decimal ASCII string is discussed in the *String Functions* section of the *Detailed Software Description* (5.3) and the algorithm for calculating the different analog values is discussed in the *Analog Code* section of the *Detailed Software Description* (5.6).

6.7 Limitations

- Constant integration time, see section 7.3.6
- Sample clock currently limited by breadboard, could otherwise be larger
- Current setup limits the speed of the processor to 20MHz, consider using a PLLA filter to raise this speed if high-speed sampling is desired.

6.8 Known Bugs

None.

7 Fixes, Notes, and Recommendations

This section outlines any changes made to the design, explains how any errors were fixed, and gives some direction for future improvement of the project. Some parts of this section are written in the first person to better express the views and opinions of the author.

7.1 Fixes

This subsection outlines any fixes made to the original design. These corrections can be seen by comparing the original schematics to the updated schematics in Appendices B and C. Most of the things fixed were quite trivial but still important enough to document. Below is a list of problems which were addressed.

- Minor corrections
- Two of the pins on the display contrast potentiometer were flipped.
- The pull-down resistors on the TWI were absent.
- Voltage high level on ADC sample clock line was too low.

Minor Corrections The title overlay text "Projecto de Pablo" should read "Proyecto de Pablo." Future version could also read "Bubbly Measurement Device" or just "Bubbly," perhaps with a version number, for example "Bubbly v1.2."

Potentiometer Fix Because the potentiometer pins were flipped, it was impossible to set the contrast on the display. This was fixed by cutting the traces and manually wiring the pins to the correct locations.

Pull-down Resistor Fix Two 10kΩ resistors were added to the proper TWI lines to remedy this problem.

Sample Clock Fix The TLC5510 specifies a minimum of 4 volts for a high input, so the 3.3 volt output of the AT91RM9200/74LVT162245 buffer was not high enough. A line driver (noninverting buffer chip, SN74HCT125) was used to drive this voltage to close to 5 volts.

7.2 Notes

7.2.1 The ADC Problem

This section would be incomplete without a thorough description of the most important problem which was the fact that something about the interface between the ADS807 chip and the processor often reset the CPU, despite the fact that the ADS807 outputs went through a buffer first. Unfortunately, a very thorough explanation of the problem cannot be provided because the cause of it is still unknown. However, this subsection provides an outline of what is known about this problem in the hopes that it may help to fix it in the future.

What I Know Firstly, the issue is not a power issue, because it only occurs when the output of the ADC is connected to the PIO pins. Otherwise, even if the ADC is outputting data, there is no issue with the processor. The issue is not (as far as I can tell) a software issue, either, because this issue still occurred even on the most elementary of revisions of the code. I became convinced it wasn't a software issue after running the most trivial version of the code, which sets up only the processor clock and sample clock (no interrupts) and still encountered the issue. This leaves us with a hardware issue. There seemed to be some large spikes on the data lines of the ADC, but it's curious that this should cause a problem after going through the buffer. The issue seemed to occur less frequently when fewer output lines were connected (the lines were connected one by one and tested). When only six or seven of the lines were connected, for instance, the processor only reset every ten minutes or so, as opposed to (almost) instantaneously when all twelve lines were connected. I would tend to believe that this issue, then, is an infrequently occurring issue with the data lines that, therefore, becomes increasingly worse the more data lines are connected. This is as far as I can stipulate without further testing, and I'm already far outside my field of knowledge. The only thing I can try recommending is adding small series resistors to the data lines to minimize the voltage spikes, but I am not at all convinced that this will fix the issue.

The Breakout Board Workaround A breakout board was made to work around the ADS807 issue and possibly also to help debug it. This board breaks out the outputs of the ADS807 ADC, TLC5510 ADC, and a 12 bit counter, and connects to the general, broken-out PIO A bank pins of the processor. Unfortunately, since only eight general inputs were available to the processor (see the ADC section in the *Detailed Hardware Description* for more information), it didn't really make sense to try to get the ADS807 ADC working on that board, as it would be very difficult to connect four of the outputs of the ADS807 to the processor. This is why the TLC5510 (an eight bit ADC) part of the board was built first and, when that worked, the other parts of the breakout board were left alone. It is possible to connect the aforementioned four remaining outputs to the original ADC input in PIO bank C; however, this involves (permanently) soldering the wires to TSSOP pins, which seemed like a very fragile and tem-

porary solution, and one that could introduce further problems (such as shorts) at that.

7.2.2 Makefile

Surprise! There is no makefile for the project because each assembly file was assembled individually every time the code was changed and then linked/located together at the end. This was done by calling the GNU assembler and locator (`as` and `ld`, respectively) with the file names as arguments. A couple of batch scripts were written to do this, and are presented below, but keep in mind that the paths might need to be changed if they are run on a different machine:

Listing 1: Assembler Script

```
@echo off
set /p name="Enter filename: "

"C:\Program Files (x86)\CodeSourcery\bin\arm-none-eabi-as.exe" -mcpu=arm920t -
    ↪ mlittle-endian -gdwarf-2 -o obj\%name%.o %name%.S

pause
```

Running this batch file will prompt the user for a filename and then assembly that file into the `objs\` folder (this surprisingly works even if the extension of the file is `.s` instead of `.S`). It may not work if there is no `objs\` folder, in which case the user can create their own folder before running this script.

Listing 2: Linker Script

```
set /p name="Enter output filename: "
"C:\Program Files (x86)\CodeSourcery\bin\arm-none-eabi-ld.exe" -T ldscript -o %
    ↪ name%.elf obj/*.o

pause
```

This file will create an executable `.elf` file in the directory from which it is run. The file's name will be the string entered by the user, and the file is an executable of all the `.o` files in the `objs\` folder.

All that being said, I would highly recommend making a Makefile. Compiling each file individually is not only tedious, but can also lead to errors when one file is accidentally left uncompiled. It would also help with the transition to C code if that's in the future scope of the project.

7.2.3 EEPROM File Generation

The `.elf` file generated by the linker/locator script above can be programmed into the processor directly, but cannot be loaded into the EEPROM. In order to generate a file which can be loaded into the EEPROM, it is necessary to call the `objcopy` program in the CodeSourcery suite. The following batch file does this.

Listing 3: EEPROM File Script

```
@echo off
set /p name="Enter filename: "
"C:\Program Files\CodeSourcery\Sourcery_CodeBench_Lite_for_ARM_EABI\bin\arm-
    ↪ none-eabi-objcopy.exe" -O ihex %name%.elf %name%
```

The input to this script is the name of the `.elf` file. This script will make a file that can be loaded into the EEPROM. The output file will have the same name as the `.elf` file.

7.2.4 Linker Script

The linker script is used by the aforementioned `ld` program to determine the address of the different sections of the code. It specifies the starting location of the code and that the entire code should be loaded into internal SRAM. It also specifies the size of this internal SRAM and also the first file that should be run (`crt0`). It is reproduced below.

Listing 4: Linker Script

```
/*****************************************************************************/
/*                                 ldscript                                  */
/*                  LINKER SCRIPT FOR GNU ARM C COMPILER                     */
/*                                                                           */
/* Revision History:                                                         */
/*    4/23/07   David Lin       Initial revision                             */
/*    1/24/12   Glen George     Added comments                               */
/*    5/18/15   Daniel Andrade  Minor modifications                          */
/*                                                                           */
/*****************************************************************************/

OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

/* set up the memory map for the system */

MEMORY
{
    /* internal memory locations */
    int_ram (RWX)   : ORIGIN = 0x200000, LENGTH = 16K
}

C_STACK_SIZE = 512;

SECTIONS
{
    /* locate the .text (code) section */
    .text :
    {
            /* make sure the startup code is the first thing in the located */
            /*    file since it contains the vector table                   */
            obj/crt0.o (.text)

            /* now can add the rest of the code */
```

```
        *(.text)
        __ecode = .;            /* set label for end of code segment */

        /* initialized read only data should be with the code */
        __rom_data_start = .;
        *(.rodata)
        *(.rdata)
        __rom_data_end = .;

} > int_ram

/* locate the .data section, used for variable storage when running */
.data :
{
         __data_start = .;
         *(.data)
         __edata = .;      /* keep track of the end of the data area */
} > int_ram

/* locate the .bss section, used for uninitialized variables */
.bss :
{
        __start_bss = .;
        *(.bss)
        *(COMMON)
        __end_bss = .;    /* keep track of the end of the bss section */
} > int_ram
}
```

This would likely have to be modified if adding extra memory to the system. It is probably best, for instance, to load/store variables (`.data` section) in external SRAM instead of using the internal memory, and to have the code (`.code` section) loaded into a ROM chip.

7.3 Recommendations

This subsection outlines any recommendations for future improvements on the board. The recommendations, of course, depends on the future direction of the project. Here I list a few concerns that occurred to me and that hopefully become useful if this is to be taken any further.

7.3.1 Future ADC Considerations

I don't recommend that my design for the ADS807 ADC be used as it is right now. I am quite certain that there is something wrong with my design, although I believe that I followed the directions on the datasheet. I would consult with an expert on the issue or do some further testing with that design before spinning a new board with it. If this advice is going to be ignored, then at the very least consider the PCB stack more carefully, as I gave it almost no thought, add sufficient ground layers, and also add series resistors on the output of the ADC before spinning the new board. I'm sure the ADC is good (especially considering the price), but don't just blindly go with what I have here since it is not working.

I also don't consider the TLC5510 to be a good substitute for the ADS807 in the final design. There are several reasons why it wasn't the ADC I chose from the beginning, namely that the resolution is smaller than requested and that it is also slower than desired. Since there is no reason to limit the output to eight bits when spinning a new board, I would consider other ADCs with ten or twelve bit resolution. Keep in mind that it should be relatively low-power, since the entire system is limited to at most 500mA, and that, if it is faster than 20 MHz, some effort will have to be put to get the PLLs on the AT91RM9200 working to output frequencies that large and to have a faster processor clock. I remember considering the THS1031 as possible candidate; that might be a good starting point.

7.3.2 Future Memory Considerations

Consider adding an SRAM chip and a EEPROM chip for added flexibility. The relief I got from not having to debug these memory chips was probably not worth the anxiety of having to face the possibility of running out of memory. Having more memory would also easy the transition into having mostly C code instead of assembly, since then there would be less worry with including libraries, for instance.

7.3.3 USB Alternatives

It could be worthwhile to consider alternatives to USB since that is not currently functional. One such alternative is Bluetooth, for instance (Wi-Fi is another). However, I do believe that USB is a good interface because of its flexibility, and it also seems proper right now because that is how the system is receiving

power. I would not suggest switching to another kind of interface unless USB becomes impossible to implement.

7.3.4 USB Hardware Upgrade

The current USB hardware setup should work, but I would recommend updating it to the one recommended in the AT91RM9200 datasheet if spinning a new board, as this adds a couple of useful features. See figure 24 on page 159. The only reason that is not the hardware setup currently being used is because I was not aware of it until after I had the board made.

7.3.5 FIFO Full Indicator

It would be useful to add an LED indicator on the FIFO full line in the future. This would be indicative of too fast of a sample clock for the processor to handle, showing that either the processor should be sped up by use of a PLL or that the code should be optimized in some way. This is especially useful if higher sampling frequencies are desired, i.e., with a different, faster ADC.

7.3.6 Suggested Software Improvements

Variable Integration Time Right now the integration time is constant, so that `NUM_AVG_SAMPLES` are always processed at once. A low value of this constant limits how much the signal can be oversampled, but the value should not be too high, or else updates would be very infrequent at low sample rates. I suggest that a variable integration time be added to address this issue, along with a menu screen and encoder capability so that the user can freely choose this value.

Period Detection An alternative to the above would be to update the variables only when a certain number of periods have been detected, thus ensuring that the signal can be freely oversampled. This might give slightly more accurate results, but would also make the code a bit more complicated, as the void rate would have to be explicitly calculated, since the number of samples will not be known *a priori*. If this option is taken, it would still be beneficial to allow the user to choose how many periods they would like to integrate over.

Timed Approach The most difficult approach to the issue, but possibly the most forward-thinking one, is to keep track of time and determine the void rate by the simple calculation

$$void\ rate = \frac{time\ high}{time\ low} * 100\%$$

for a given amount of time, which would be specified by the user or could be a function of the current sample rate. This approach is difficult because keeping track of time involves counting sample clocks and multiplying by a variable value depending on the sample rate. In addition, it can be a bit painful to do

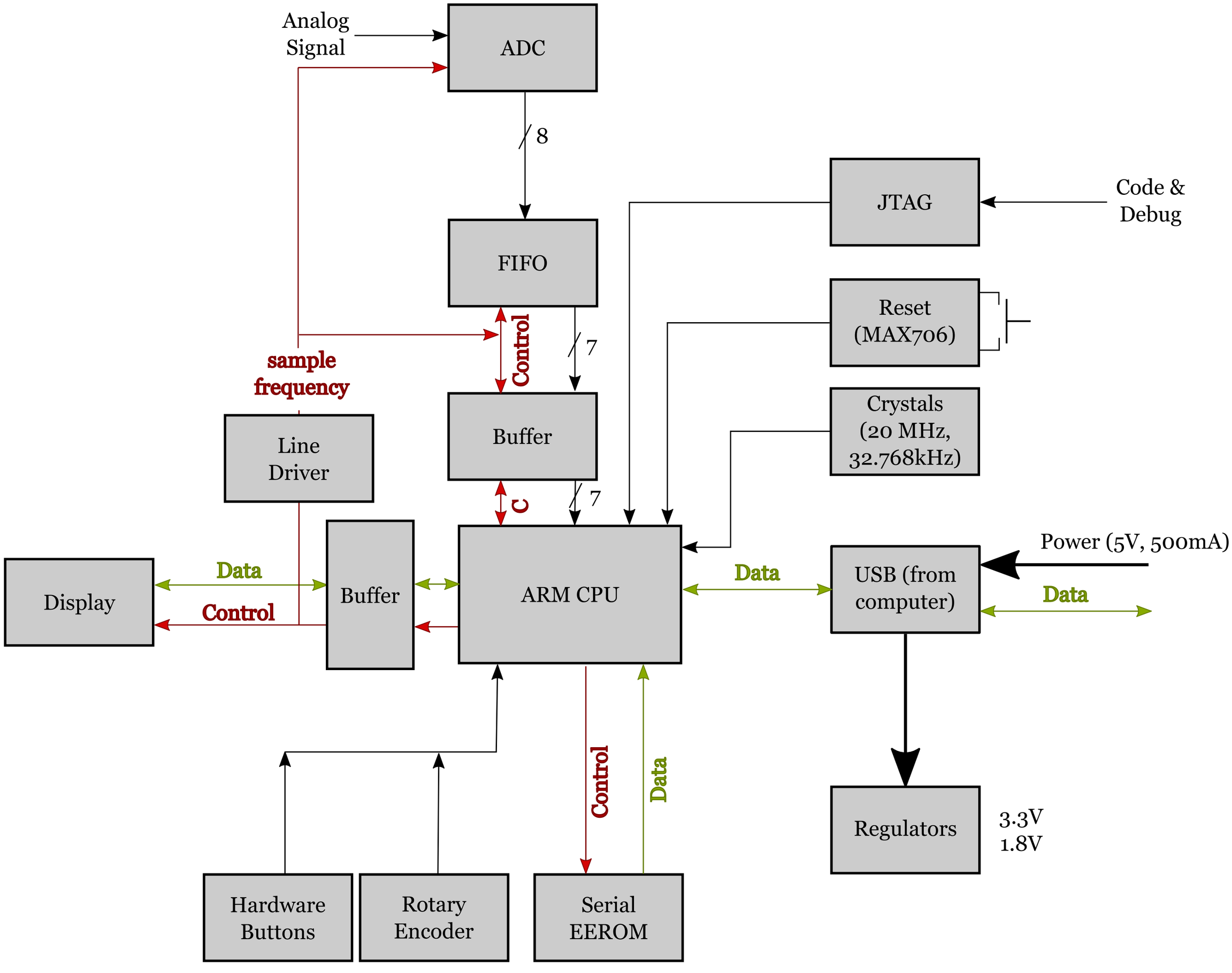
complex calculations in assembly. This approach is forward-thinking because keeping track of time will be necessary once USB is fully implemented, so that the computer can tell at what time each of the edges was detected.

Appendices

A Block Diagrams

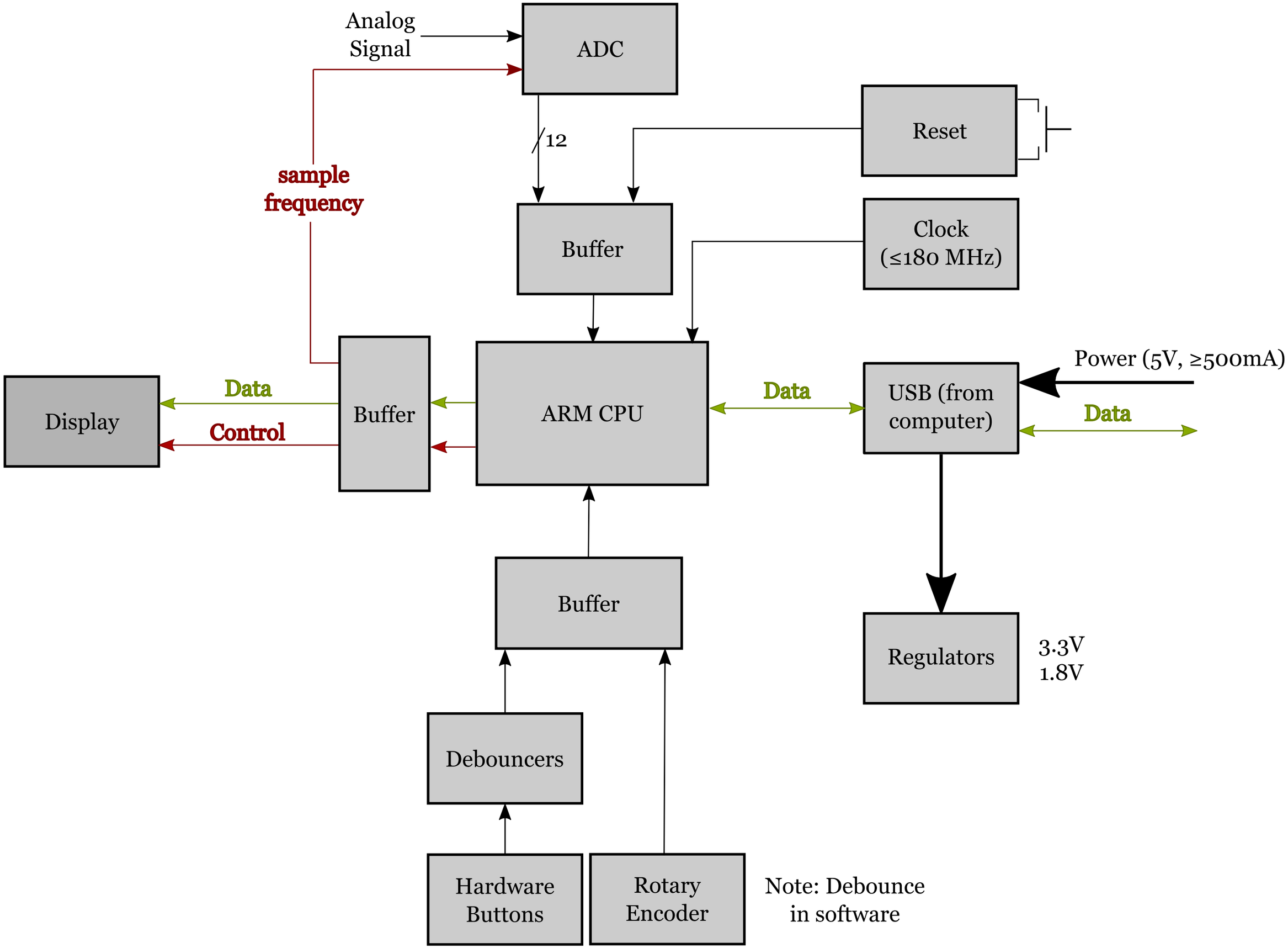
A.1 Updated Block Diagram (Large Version)

The updated block diagram shows an up-to-date, high-level view of the system. It is useful for showing signal flow without going into the quantity of detail presented in the schematics.


Analog Signal
ADC
8
FIFO
Control
7
sample frequency
Line Driver
Buffer
7
C
JTAG
Code & Debug
Reset (MAX706)
Crystals (20 MHz, 32.768kHz)
Power (5V, 500mA)
Display
Data
Control
Buffer
ARM CPU
Data
USB (from computer)
Data
Regulators
3.3V
1.8V
Control
Data
Hardware Buttons
Rotary Encoder
Serial EEROM

A.2 Original Block Diagram

The original block diagram is the first design document for the project. It represents the first view into what the project was, and it is extremely out of date. It is included here for the satisfaction of the curiosity of the readers only.


Analog Signal
ADC
12
Reset
sample frequency
Buffer
Clock (≤180 MHz)
Display
Data
Control
Buffer
ARM CPU
Data
USB (from computer)
Power (5V, ≥500mA)
Data
Buffer
Regulators
3.3V
1.8V
Debouncers
Hardware Buttons
Rotary Encoder
Note: Debounce in software

B Original Board Design Documents

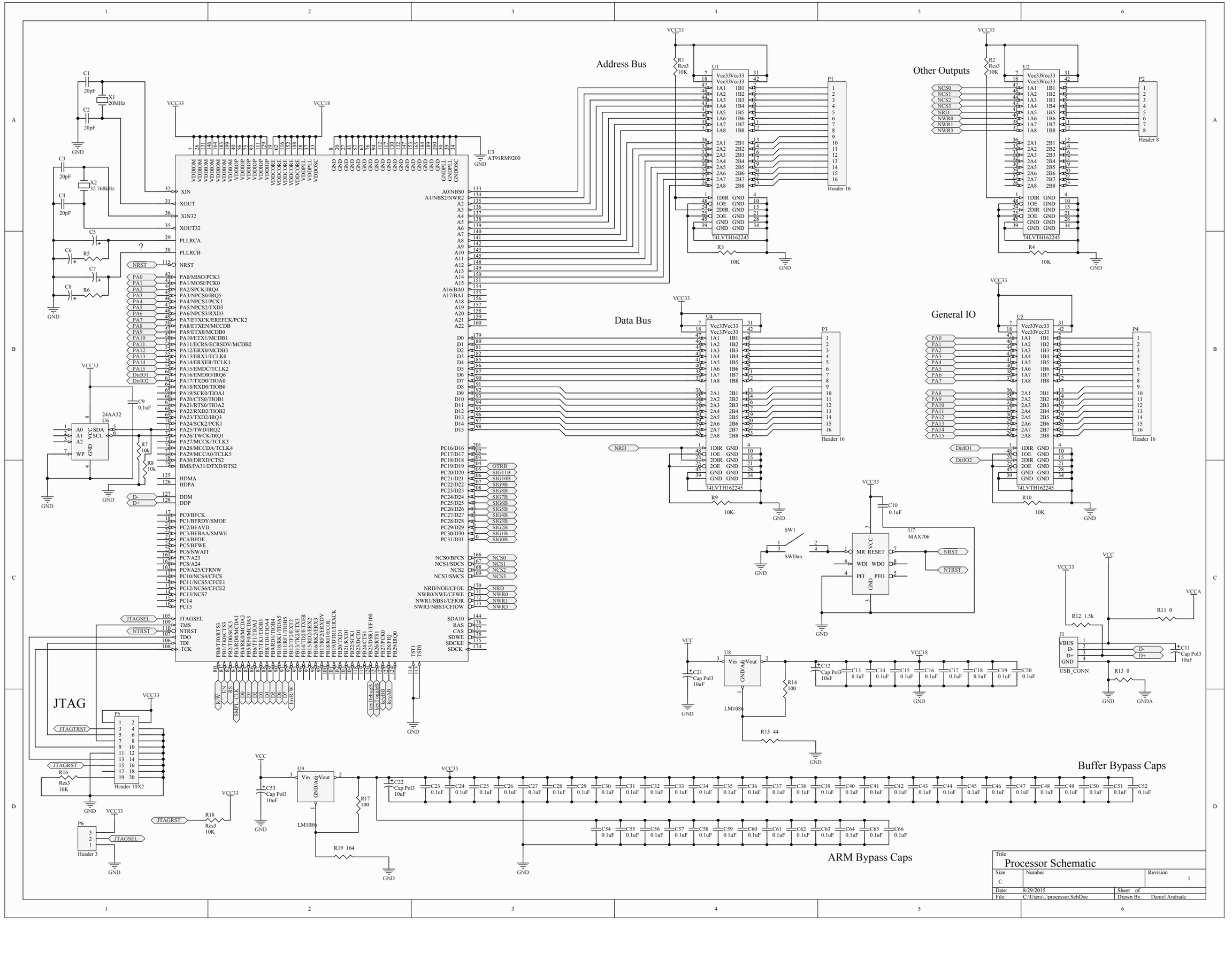
B.1 Updated Schematics

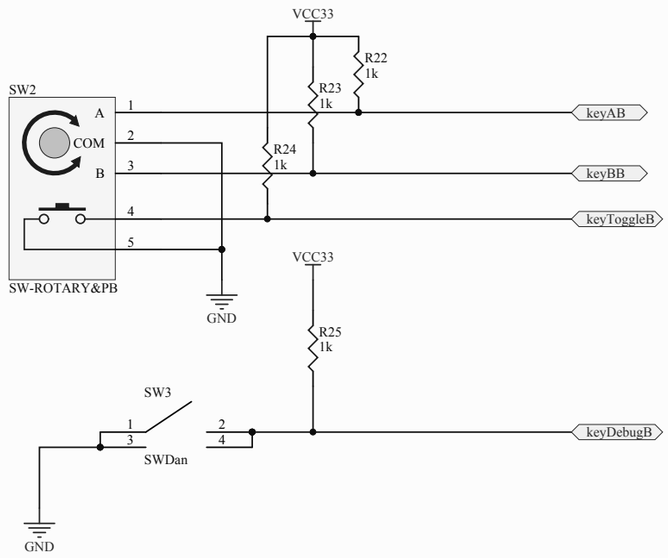
Presented in this section are the schematics of the original board revised with the changes described in the *Fixes* subsection of the *Fixes, Notes, and Recommendations* section. This is not the full schematics, in that it does not include the FIFO and ADC used (see *Appendix C* for those). There are two main reasons why the schematic documents were not combined:

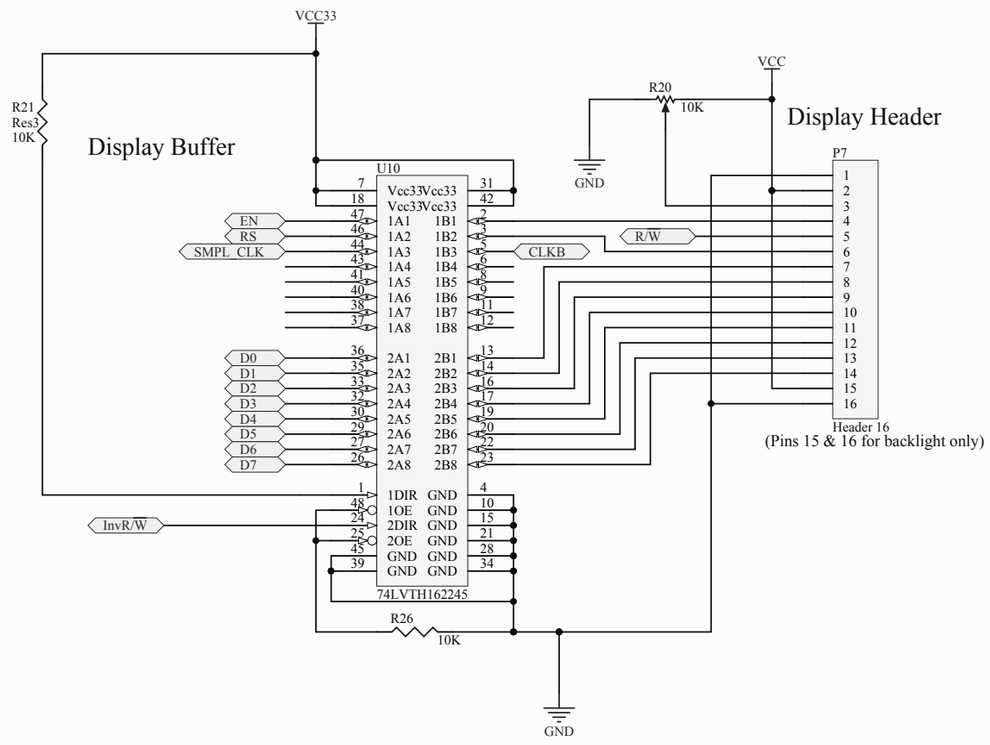
1. This view presents the current state of the project better, since right now there really are three separate boards.

2. The ADC used is likely not the one that should be used in the final design, as stated in the *Recommendations* subsection. If an ADC with a higher resolution is chosen, the FIFO would need to be changed as well. Thus, having one full document does not have the customary advantage of being ready to spin, and would present a state of finality which the project does not have at this moment.

The first ADC design (ADS807) was removed from the documents in this updated board section since it did not work. It is included, however, in the *Original Schematics* subsection of this appendix.

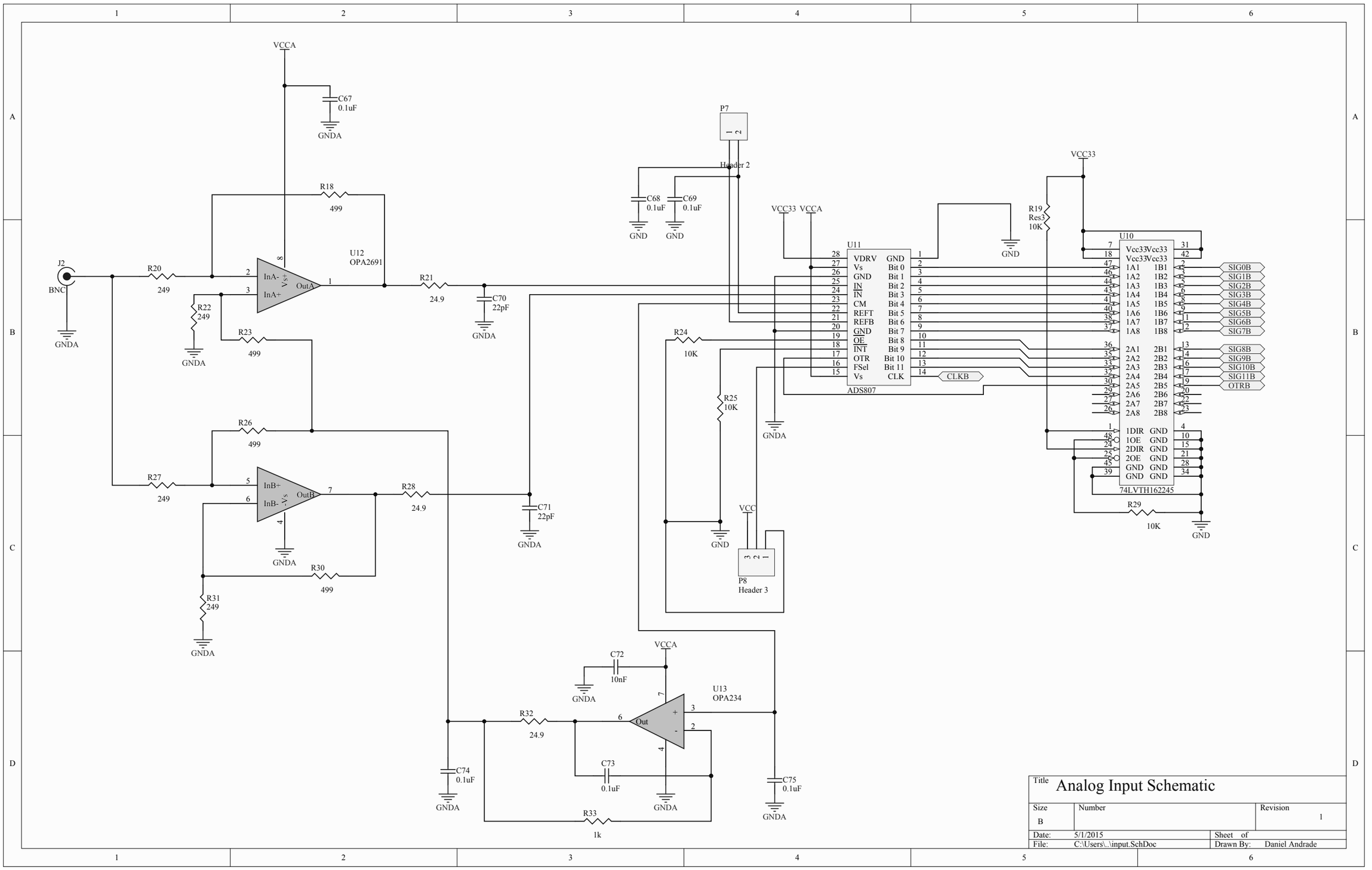

Address Bus
Data Bus
Other Outputs
General IO
JTAG
Buffer Bypass Caps
ARM Bypass Caps
Title
Processor Schematic
Size
C
Number
Revision
1
Date: 8/29/2015
Sheet of
File: C:\Users\...\processor.SchDoc
Drawn By: Daniel Andrade



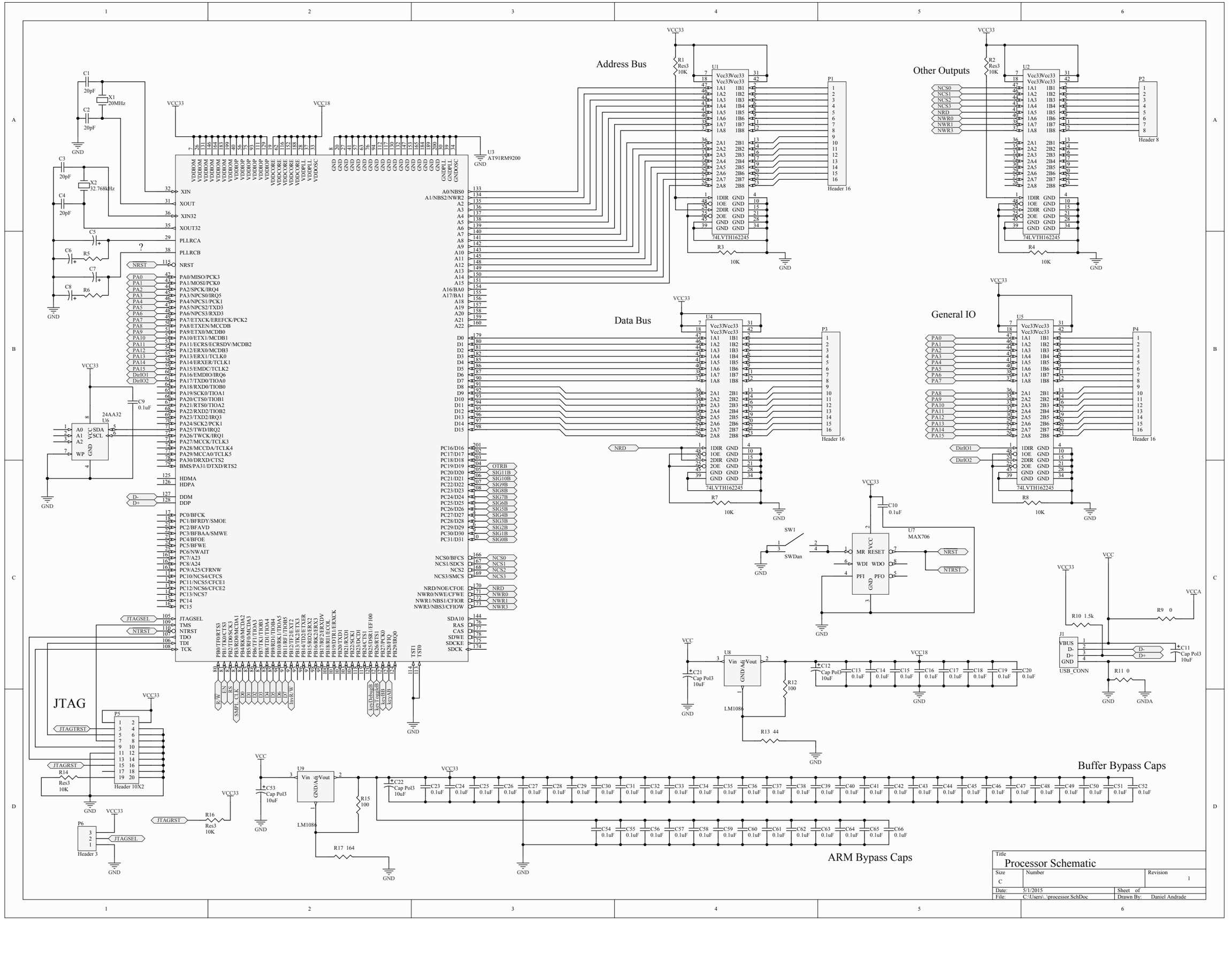


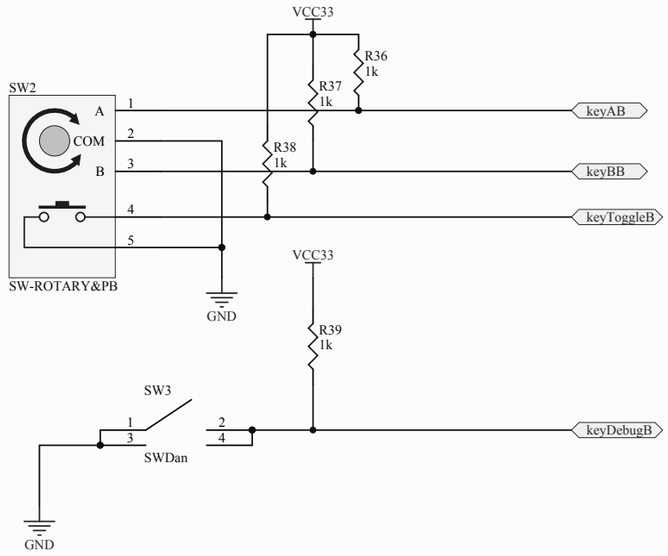
Title	User Input		
Size: B	Number:		Revision:
Date: 8/29/2015		Sheet of	
File: C:\Users\..\UI.SchDoc		Drawn By: Daniel Andrade	

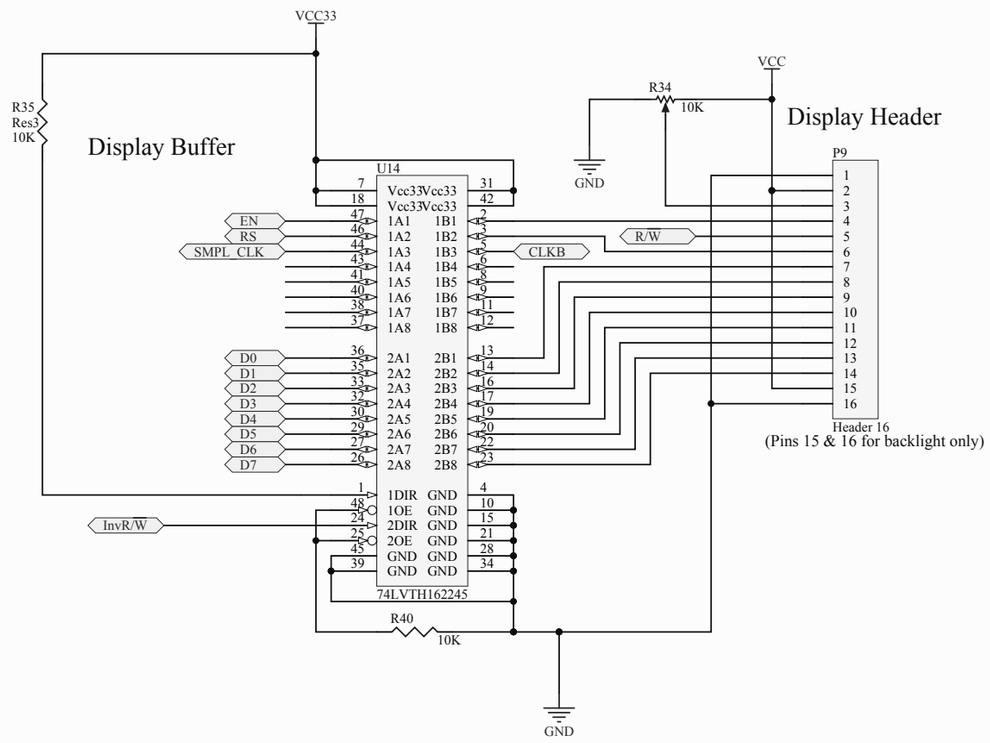
B.2 Original Schematics



Title	Analog Input Schematic		
Size: B	Number:		Revision: 1
Date: 5/1/2015		Sheet of	
File: C:\Users\..\input.SchDoc		Drawn By: Daniel Andrade	


Address Bus
Data Bus
Other Outputs
General IO
JTAG
Buffer Bypass Caps
ARM Bypass Caps
Title
Processor Schematic
Size
C
Number
Revision
1
Date: 5/1/2015
File: C:\Users\..\processor.SchDoc
Sheet of
Drawn By: Daniel Andrade

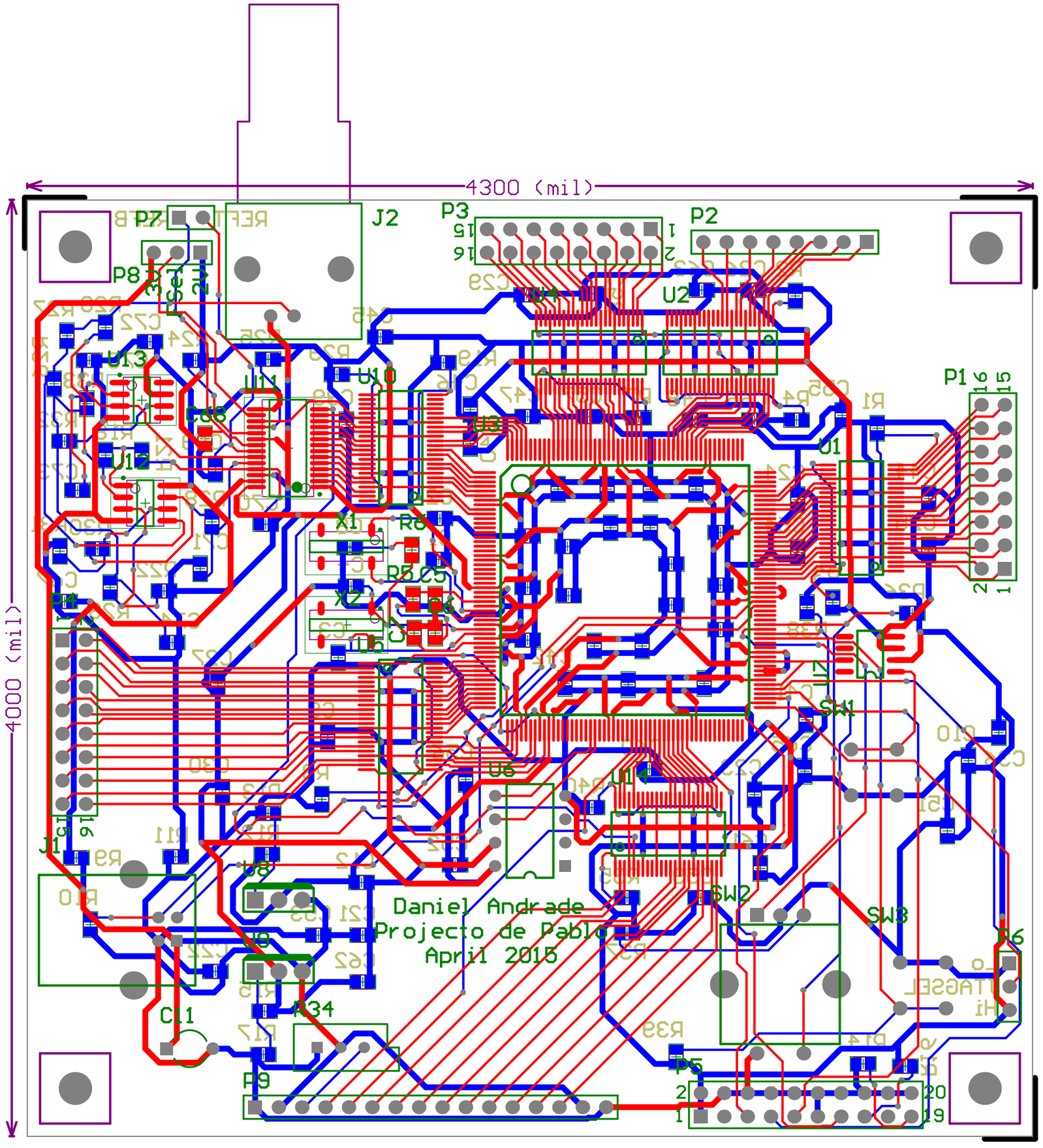




Title	User Input		
Size: B	Number:	Revision:	
Date: 5/1/2015		Sheet of	
File: C:\Users\..\UI.SchDoc		Drawn By: Daniel Andrade	

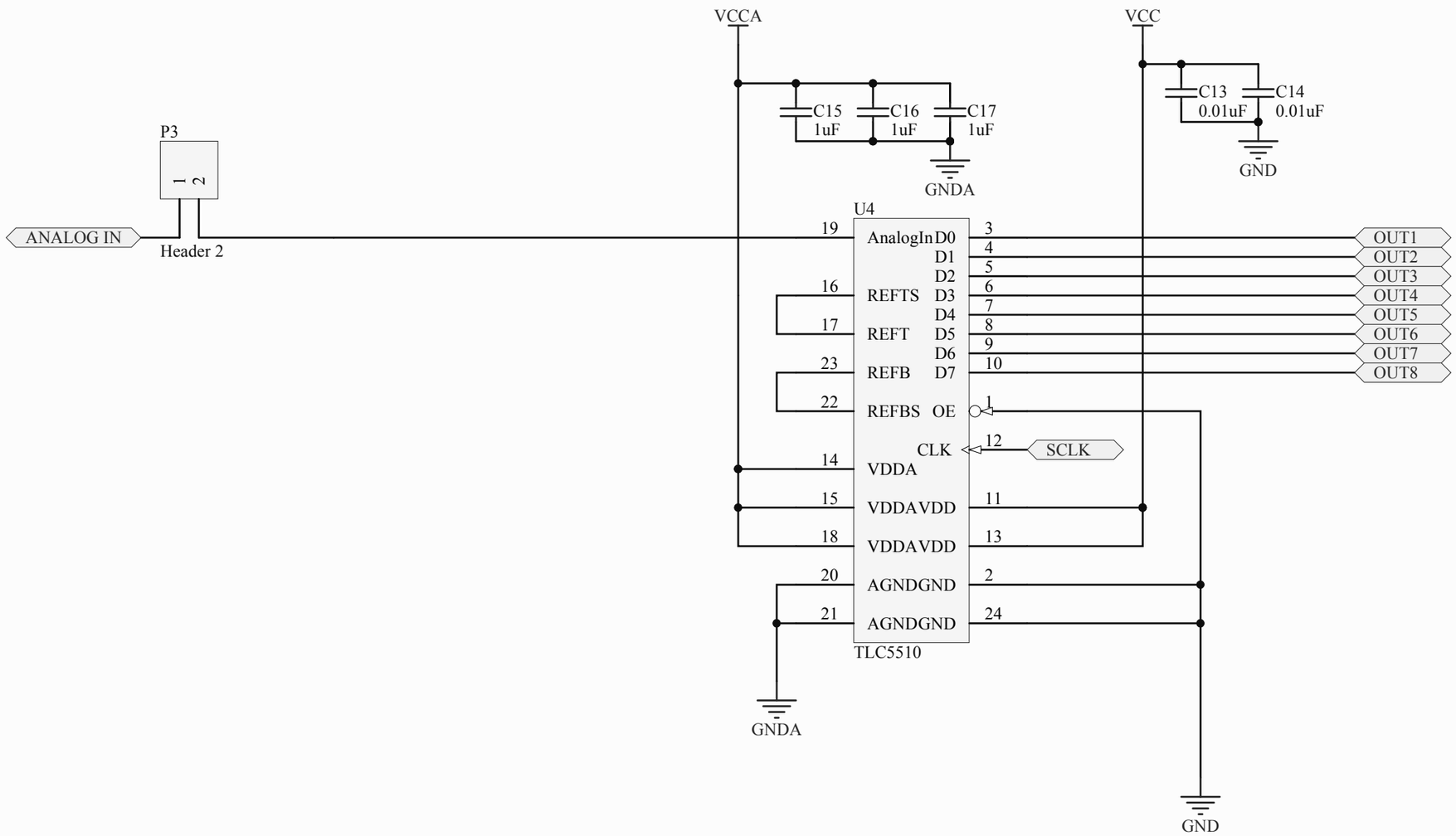
B.3 PCB Layout

The colors are the standard Altium colors, with red traces being on top, blue traces being on bottom, and the silkscreen in a green/yellow color. Mechancial features are shown in purple, and three of the board corners are marked with thick black traces for orientation.

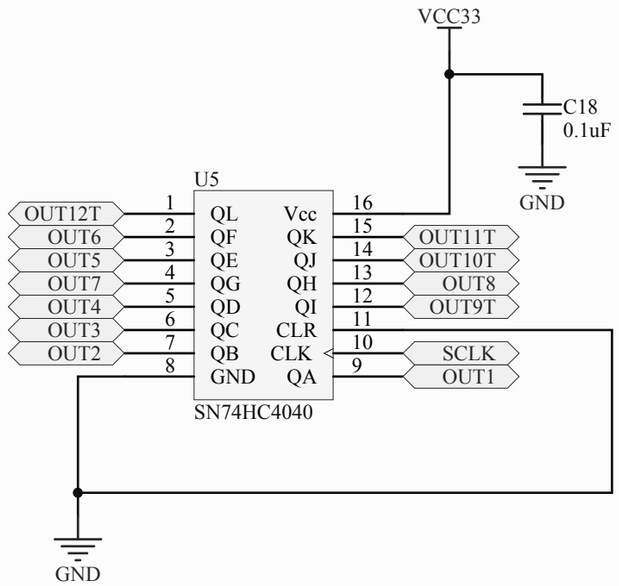

4300 (mil)
4000 (mil)
J2
P7
P8
P3
P2
P1
P4
P5
P6
P9
J1
U1
U2
U3
U4
U5
U6
U7
U8
U9
U10
U11
U12
U13
X1
X2
R5
C5
C6
C11
C38
SW1
SW2
SW3
Daniel Andrade
Projecto de Pablo
April 2015

C Breakout Boards Design Documents

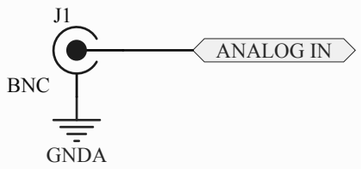
C.1 ADC Breakout Board Schematics

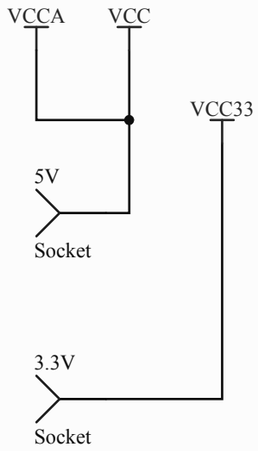


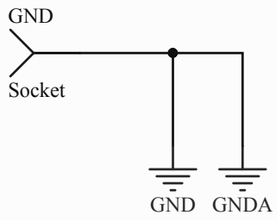
Title	8 Bit ADC	
Size: A	Number:	Revision:
Date: 7/15/2015		Sheet of
File: C:\Users\..\8bitADC.SchDoc		Drawn By:

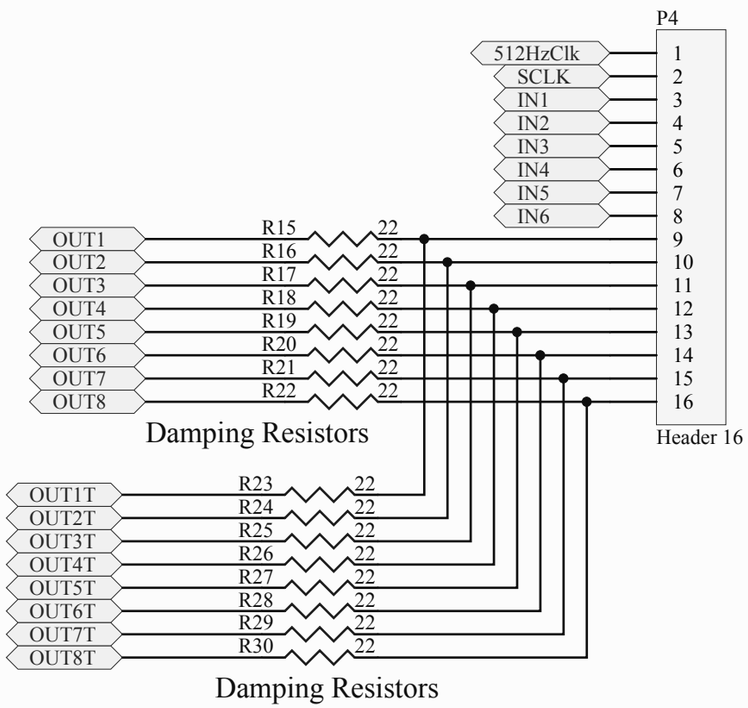


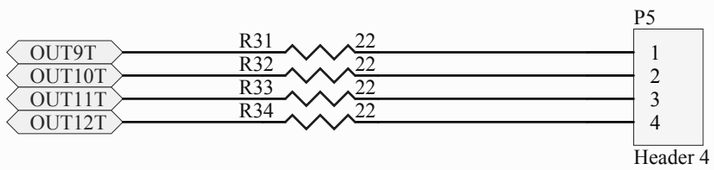
Title	Counter	
Size: A	Number:	Revision:
Date: 7/15/2015		Sheet of
File: C:\Users\..\Counter.SchDoc		Drawn By: Daniel Andrade



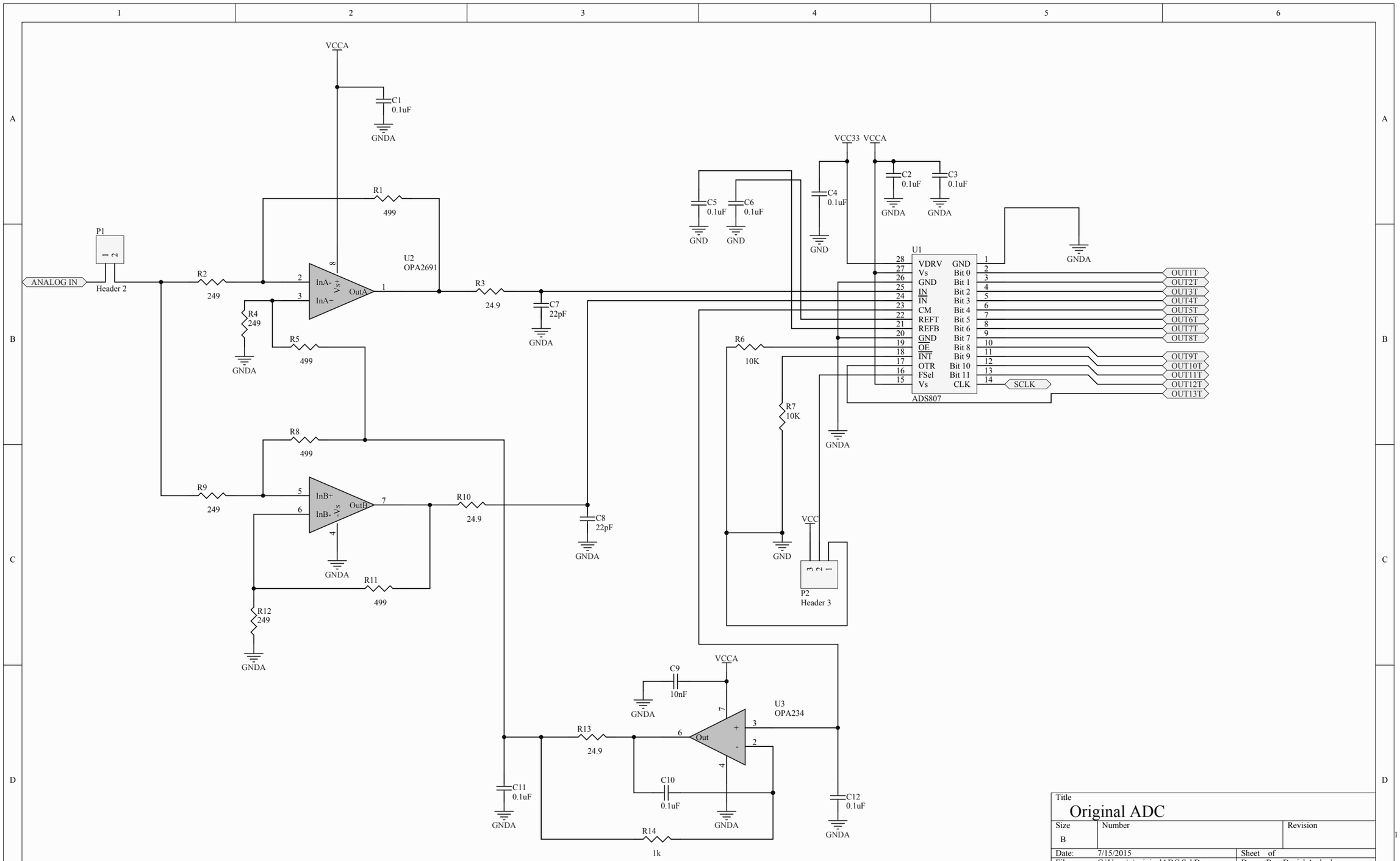






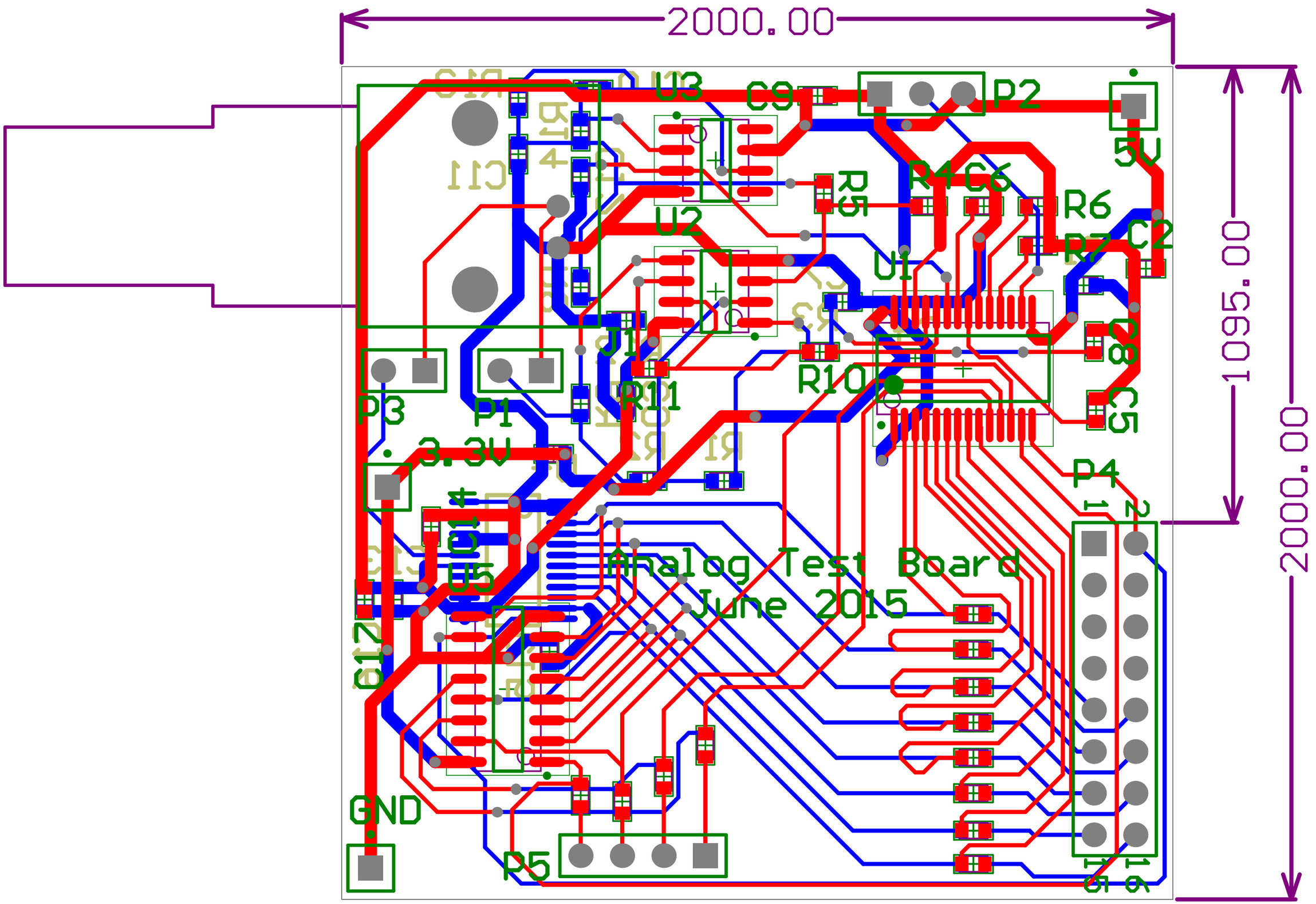


Title: External Connections and Power		
Size: A	Number:	Revision:
Date: 7/15/2015		Sheet of
File: C:\Users\..\IO.SchDoc		Drawn By: Daniel Andrade



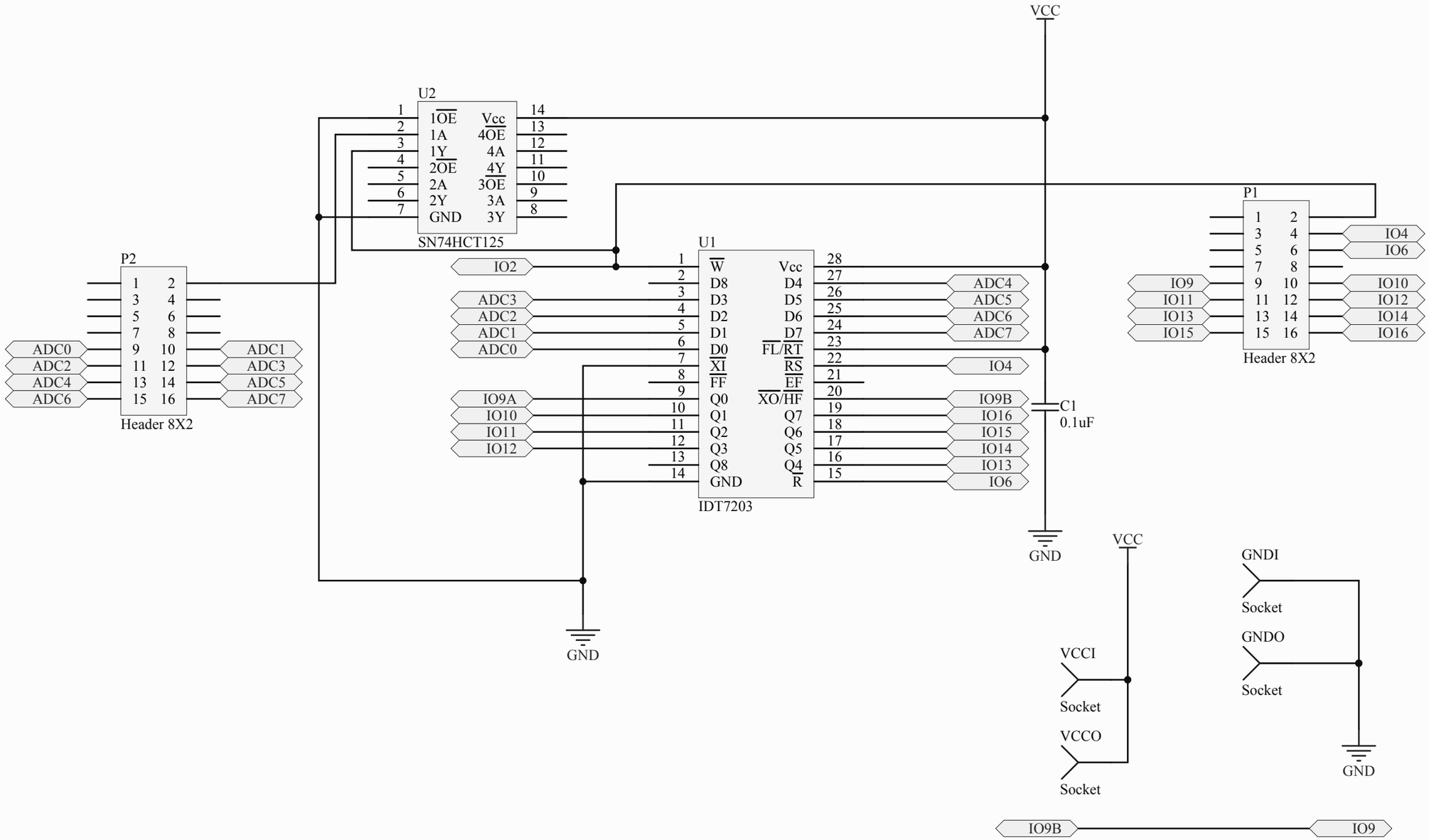
Title	Original ADC		
Size: B	Number:	Revision:	
Date: 7/15/2015		Sheet of	
File: C:\Users\..\originalADC.SchDoc		Drawn By: Daniel Andrade	

C.2 ADC Breakout Board Layout


2000.00
1095.00
2000.00
U3
C9
P2
5V
R4
C6
R5
R6
U2
U1
C2
J1
R10
R11
P3
P1
3.3V
P4
C5
J4
U5
Analog Test Board
June 2015
GND
P5

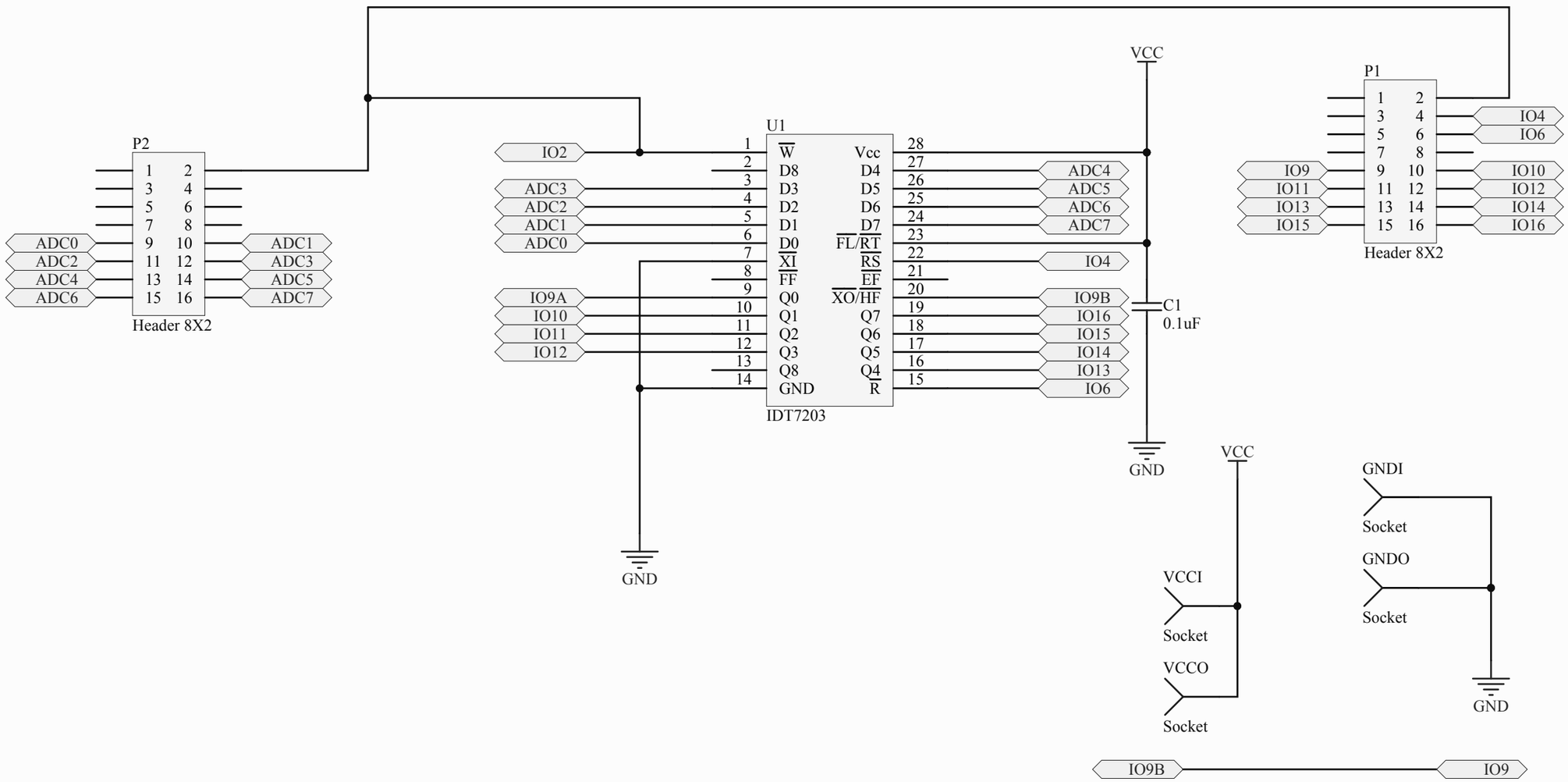
C.3 Updated FIFO Board Schematics

The required line driver described in the *Fixes* subsection of the *Fixes, Notes, and Recommendations* section to drive the sample clock at 5 volts instead of 3.3 volts is included in these schematics, but was absent from the original design, and is not present in the board layout.



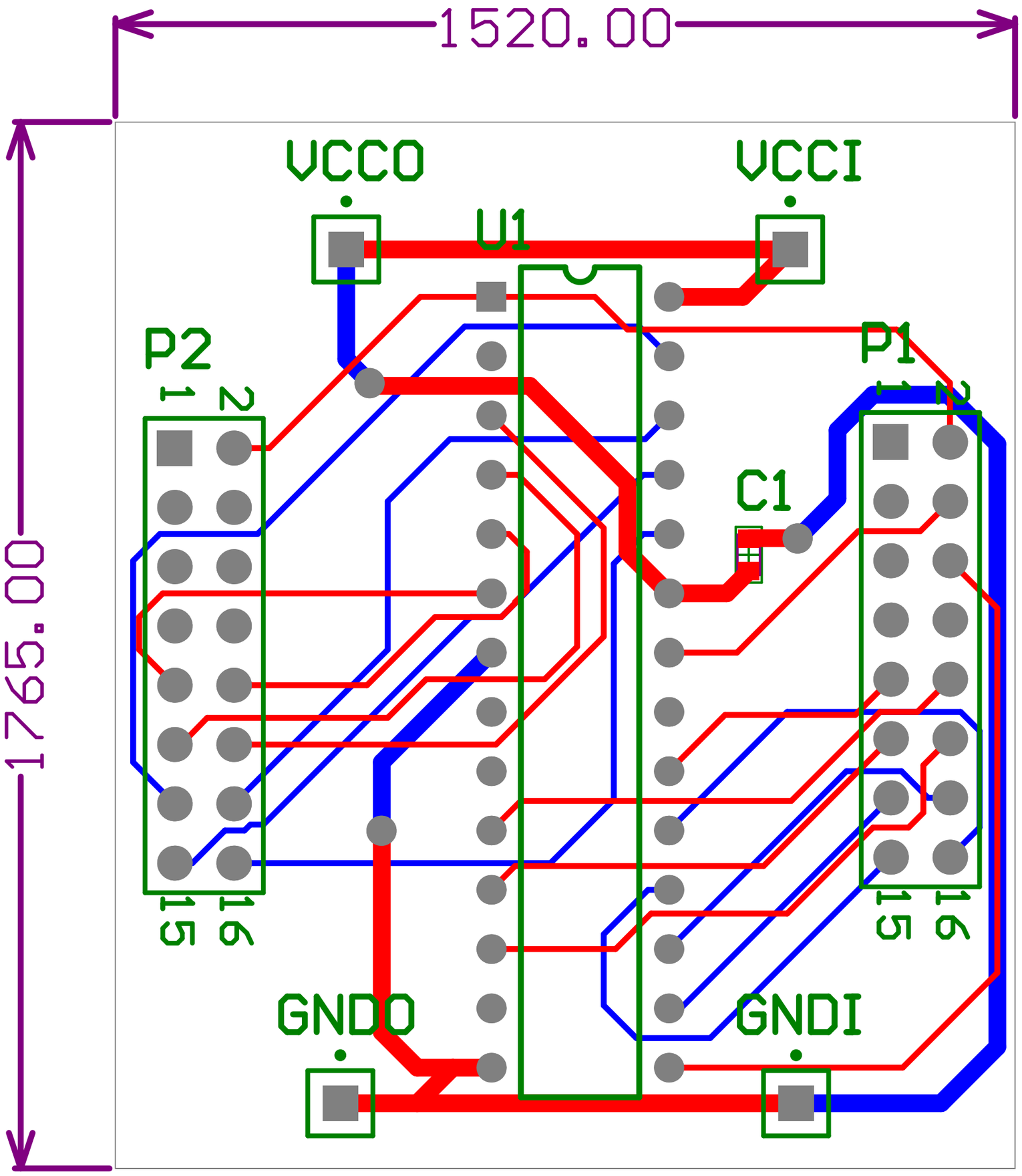
Title	FIFO Schematic		
Size: A	Number:	Revision:	
Date: 8/29/2015		Sheet of	
File: C:\Users\..\FIFO.SchDoc		Drawn By: Daniel Andrade	

C.4 Original FIFO Board Schematics



Title	FIFO Schematic	
Size: A	Number	Revision
Date: 8/14/2015		Sheet of
File: C:\Users\..\FIFO.SchDoc		Drawn By: Daniel Andrade

C.5 FIFO Board Layout


1520.00
1765.00
VCCO
VCCI
U1
P2
1
2
15
16
P1
1
2
15
16
C1
GNDO
GNDI

D Full System Code

The actual code can be found in a Bitbucket Git repository, do not take the code from this document, as there are page numbers and headers present, which would get in the way.

The following link should take you to the Bitbucket repository, which is up to date and should remain at this web address for the foreseeable future. You can also find the decently thorough commit history, log, comments, and branches there.

https://bitbucket.org/DSeabra/ee53code

The most recent code is in the `usb/` directory.

D.1 General

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                               @
@                                  General.inc                                  @
@                                     EE 53                                     @
@                                                                               @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ General constants
.equ    TRUE,   0x1
.equ    FALSE,  0x0

.equ    HIGH,   0x1
.equ    LOW,    0x0

.equ    NUM_BITS_WORD, 32
```

D.2 Main Loop

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                  Mainloop                                    @
@                               Main Loop Files                                @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  The main function and other related & helper functions.
@
@  Table of Contents: main - The main function. Does not return 0.
@

.extern key_available
.extern get_key
.extern set_display
.extern dec2string
.extern update_void_fraction
.extern auto_display_update

.include "keypad.inc"
.include "general.inc"


.section .text
.align 2
.arm


@  main
@
@  Description: Main loop.
@
@  Operation:   Loops forever doing the following things:
@                   1. Counting and updating the display if it's time to do so
@                   2. Checking for and handling key presses
@                   3. Updating analog code variables (averages, threshold,
@                       void rate) when FIFO data is present.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
```

```
@
@  Revision History:
@       05/16/2015    Daniel Andrade    initial revision
@       06/02/2015    Daniel Andrade    added polling code for keypad
@       08/03/2015    Daniel Andrade    revision for polling analog code
@                                         (instead of interrupting)
@       08/14/2015    Daniel Andrade    revision for auto-updating values
@                                         on the display (e.g. for threshold)

.global main
.type main STT_FUNC

main:
mainloop:
bl      auto_display_update @ Update the display if it's time to do so

bl      key_available       @ If there was a key pressed, then handle
cmp     r0, #TRUE           @  it here.
bne     main_check_data
bl      get_key
bl      ui_do_key

main_check_data:
bl      get_data_available  @ If there's data in the FIFO, then read
cmp     r0, #TRUE           @  it, process it, and update the value
bne     mainloop            @  of the void fraction (see UI code)
bl      adc_process_fifo
bl      update_void_fraction

b       mainloop

bx      lr                  @ Should not be returning, but if we do then
                            @  we'll end up in crt0 again

.end
```

D.3 Processor Initialization

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@ at91rm9200.inc                                                               @
@                                                                              @
@ General control register address definitions for the Atmel AT91RM9200       @
@                                                                              @
@ Revision History:                                                            @
@                                                                              @
@   2008/04/23  Arthur Chang    Initial Revision                               @
@   2010/02/01  Joseph Schmitz  Modified file I received from Arthur Chang     @
@                                  to distribute to students                   @
@   2011/02/13  Glen George     Cleaned up commenting, removed definitions     @
@                                  not related to the AT91RM9200 chip.         @
@   2014/05/16  Daniel Andrade  Got rid of a lot of stuff I already had        @
@                                  elsewhere.                                  @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@ System Mode Definitions

    .equu   ARM_MODE_USR,     0x10                  @ User Mode
    .equ    ARM_MODE_FIQ,     0x11                  @ FIQ Mode
    .equ    ARM_MODE_IRQ,     0x12                  @ IRQ Mode
    .equ    ARM_MODE_SVC,     0x13                  @ Supervisor Mode
    .equ    ARM_MODE_ABT,     0x17                  @ Abort Mode
    .equ    ARM_MODE_UND,     0x1B                  @ Undefined Mode
    .equ    ARM_MODE_SYS,     0x1F                  @ System Mode
    .equ    I_BIT,            0x80                  @ Interrupts disabled
    .equ    F_BIT,            0x40                  @ Fast Interrupts disabled


@ Static Memory Controller Definitions

    .equ     SMCBase,         0xFFFFFF70            @ base address

    .equ     SMC_CSR0,        SMCBase + 0x0         @ Chip Select 0 Register
    .equ     SMC_CSR1,        SMCBase + 0x4         @ Chip Select 1 Register
    .equ     SMC_CSR2,        SMCBase + 0x8         @ Chip Select 2 Register
    .equ     SMC_CSR3,        SMCBase + 0xC         @ Chip Select 3 Register
    .equ     SMC_CSR4,        SMCBase + 0x10        @ Chip Select 4 Register
    .equ     SMC_CSR5,        SMCBase + 0x14        @ Chip Select 5 Register
    .equ     SMC_CSR6,        SMCBase + 0x18        @ Chip Select 6 Register
    .equ     SMC_CSR7,        SMCBase + 0x1C        @ Chip Select 7 Register


@ Advanced Interrupt Ccontroller Definitions

    .equ     AICBase,         0xFFFFF000            @ base address
    .equ     AIC_SMR0,        AICBase + 0x0
    .equ     AIC_SVR0,        AICBase + 0x80
    .equ     AIC_SMR1,        AICBase + 0x4
    .equ     AIC_SVR1,        AICBase + 0x84
    .equ     AIC_SMR2,        AICBase + 0x8         @ PIOA (General PIO A Bank)
    .equ     AIC_SVR2,        AICBase + 0x88        @   Source Vector 2
    .equ     AIC_SMR11,       AICBase + 0x2C        @ UDP (USB Device Port)
    .equ     AIC_SVR11,       AICBase + 0xAC        @   Source Vector 11
    .equ     AIC_SMR14,       AICBase + 0x38        @ SSC0: Source Mode 14
    .equ     AIC_SVR14,       AICBase + 0xB8        @ SSC0: Source Vector 14
    .equ     AIC_SMR17,       AICBase + 0x44        @ TC0: Source Mode 17
    .equ     AIC_SVR17,       AICBase + 0xC4        @ TC0: Source Vector 17
    .equ     AIC_SMR18,       AICBase + 0x44        @ TC1: Source Mode 18
    .equ     AIC_SVR18,       AICBase + 0xC4        @ TC1: Source Vector 18
    .equ     AIC_SMR24,       AICBase + 0x60        @ EMAC: Source Mode 24
    .equ     AIC_SVR24,       AICBase + 0xE0        @ EMAC: Source Vector 24
```

```
    .equ    AIC_SMR25,      AICBase + 0x64      @ IRQ0: Source Mode 25
    .equ    AIC_SVR25,      AICBase + 0xE4      @ IRQ0: Source Vector 25
    .equ    AIC_SMR26,      AICBase + 0x68      @ IRQ1: Source Mode 26
    .equ    AIC_SVR26,      AICBase + 0xE8      @ IRQ1: Source Vector 26
    .equ    AIC_IVR,        AICBase + 0x100     @ Interrupt Vector Register
    .equ    AIC_ISR,        AICBase + 0x108     @ Interrupt Status
    .equ    AIC_IMR,        AICBase + 0x110     @ Interrupt Mask
    .equ    AIC_IPR,        AICBase + 0x10C     @ Interrupt Pending
    .equ    AIC_IECR,       AICBase + 0x120     @ Interrupt Enable Command
    .equ    AIC_IDCR,       AICBase + 0x124     @ Interrupt Disable Command
    .equ    AIC_ICCR,       AICBase + 0x128     @ Interrupt Clear Command
    .equ    AIC_EOICR,      AICBase + 0x130     @ End of Interrupt Command
    .equ    AIC_SPU,        AICBase + 0x134     @ For spurious interrupts


@ Serial Synchronous Controller 0 Definitions

    .equ    SSC0Base,       0xFFFD0000          @ base address
    .equ    SSC0_CR,        SSC0Base + 0x0      @ Control
    .equ    SSC0_CMR,       SSC0Base + 0x4      @ Clock Mode
    .equ    SSC0_RCMR,      SSC0Base + 0x10     @ Receive Clock Mode
    .equ    SSC0_RFMR,      SSC0Base + 0x14     @ Receive Frame Mode
    .equ    SSC0_TCMR,      SSC0Base + 0x18     @ Transmit Clock Mode
    .equ    SSC0_TFMR,      SSC0Base + 0x1C     @ Transmit Frame Mode
    .equ    SSC0_RHR,       SSC0Base + 0x20     @ Receive Holding
    .equ    SSC0_THR,       SSC0Base + 0x24     @ Transmit Holding
    .equ    SSC0_SR,        SSC0Base + 0x40     @ Status
    .equ    SSC0_IER,       SSC0Base + 0x44     @ Interrupt Enable
    .equ    SSC0_IDR,       SSC0Base + 0x48     @ Interrupt Disable


@ General Definitions

    .equ    WORD_SIZE,      0x4                 @ size of a word in bytes
    .equ    HALFWORD_SIZE,  0x2                 @ size of a halfword in bytes
    .equ    BYTE_SIZE,      0x1                 @ size of a byte in bytes
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                                @
@ system.inc                                                                     @
@                                                                                @
@ Include file for the system hardware definitions for the EE/CS 52 VoIP        @
@ phone project.  This file should contain all of the hardware specific          @
@ definitions (chip select register values, control words, etc).                 @
@                                                                                @
@ Revision History:                                                              @
@                                                                                @
@   2012/01/29  Glen George     Initial revision.                                @
@   2015/05/20  Daniel Andrade  Updated TOP_STACK and some dummy values          @
@                                 for the other constants.                       @
@   2015/06/02  Daniel Andrade  Huh. Turns out these constants actually          @
@                                 make a big difference in whether code          @
@                                 works or not. Who knew? Changed                @
@                                 IRQ_STACK_SIZE and SVC_STACK_SIZE to           @
@                                 something more reasonable.                     @
@                                                                                @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@ Stack definitions

@ top of the stack, usually the end of the external SRAM
    .equ    TOP_STACK, 0x0020399C

@ interrupt stack size (see chapter 13 of the manual)
    .equ    IRQ_STACK_SIZE, 256

@ supervisor mode stack size (see chapter 13 of the manual)
    .equ    SVC_STACK_SIZE, 256
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                  PMC.inc                                     @
@                    ARM Processor Initialization Routines                     @
@                                   EE 53                                      @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for initializing the power management control registers

@ Base addresses
.equ    PMCAddr,    0xFFFFFC00          @ Base address

@ Offsets for PMC
.equ    PMC_PLLBR,  0x2C
.equ    PMC_SCER,   0x00
.equ    PMC_SCDR,   0x04
.equ    PMC_MOR,    0x20
.equ    PMC_MCKR,   0x30
.equ    PMC_SR,     0x68
.equ    PMC_PCK0,   0x40
.equ    PMC_PCK3,   0x4C
.equ    PMC_PCER,   0x10

@ Values
.equ    SCERVal,    0x00000903          @ Enable main clock, PCK0, and PCK3
.equ    MCKRVal,    0x00000001          @ Enable main clock
.equ    MORVal,     0x0000FF01          @ Max wait time
.equ    PCK0Val,    0x00000018          @ PCK0 set to 512 Hz to begin with
.equ    PCERVal,    0x0000001C          @ Enables the following peripherals:
                                        @   PIOA, PIOB, PIOC
.equ    PCK3Val,    0x00000018          @ Set up PCK3

@ Misc
.equ    MCKReadyMsk, 0b00000000000000000000000000001000     @ To check if main
                                                            @   clock is ready
.equ    MORReadyMsk, 0b00000000000000000000000000000001
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                   PIO.inc                                    @
@                    ARM Processor Initialization Routines                     @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for initializing the PIO controller registers

@ Base addresses
.equ    PIOAAddr,   0xFFFFF400
.equ    PIOBAddr,   0xFFFFF600
.equ    PIOCAddr,   0xFFFFF800


@ Offsets for PIO
.equ    PIO_PER,    0x00
.equ    PIO_PDR,    0x04
.equ    PIO_OER,    0x10
.equ    PIO_ODR,    0x14
.equ    PIO_IFER,   0x20
.equ    PIO_IFDR,   0x24
.equ    PIO_SODR,   0x30
.equ    PIO_CODR,   0x34
.equ    PIO_ODSR,   0x38
.equ    PIO_PDSR,   0x3C
.equ    PIO_IER,    0x40
.equ    PIO_IDR,    0x44
.equ    PIO_ISR,    0x4C
.equ    PIO_MDER,   0x50
.equ    PIO_MDDR,   0x54
.equ    PIO_PUDR,   0x60
.equ    PIO_PUER,   0x64
.equ    PIO_ASR,    0x70
.equ    PIO_BSR,    0x74
.equ    PIO_OWER,   0xA0
.equ    PIO_OWDR,   0xA4


@ Values to write to each register (PIO A)
.equ    perValA,    0x0003FFFE
.equ    pdrValA,    0x00000001
.equ    sodrValA,   0x00010003              @ Sets the direction bits
.equ    codrValA,   0x00020000
.equ    bsrValA,    0x00000001
.equ    ierValA,    0x00000001              @ Enables interrupts on PCK3
.equ    oerValA,    0x000300FF
.equ    odrValA,    0x0000FF00

@ Values to update PIOA for interrupts
.equ    pdrValA2,   0x00000003
.equ    bsrValA2,   0x00000003

@ Values to write to each register (PIO B)
.equ    perValB,     0b11111111111111111111111111111111
.equ    pdrValB,     0b00000000000000000000000000000000
.equ    oerValB,     0b00000000000000000001111111110111
.equ    odrValB,     0b00011110000000000000000000001000
.equ    iferValB,    0b00011110000000000000000000000000
.equ    ifdrValB,    0b00000000000000000001111111111111
.equ    sodrValB,    0b00000000000000000001000000000000
.equ    codrValB,    0b00011110000000000001111111111111
.equ    ierValB,     0b00000000000000000000000000000000
.equ    idrValB,     0b11111111111111111111111111111111
```

```
.equ    mderValB,     0b00000000000000000000000000000000
.equ    mddrValB,     0b00011110000000000001111111111111
.equ    pudrValB,     0b00011110000000000001111111111111
.equ    puerValB,     0b00000000000000000000000000000000
.equ    owerValB,     0b00000000000000000001111111111111
.equ    owdrValB,     0b00000000000000000000000000000000

@ Values to write to each register (PIO C)
.equ    perValC,      0b11111111111111111111111111111111
.equ    oerValC,      0b00000000000000000000000000000000
.equ    odrValC,      0b11111111111111111111111111111111
.equ    iferValC,     0b11111111111111111111111111111111
.equ    idrValC,      0b11111111111111111111111111111111
.equ    mddrValC,     0b11111111111111111111111111111111
.equ    pudrValC,     0b11111111111111111111111111111111
.equ    owdrValC,     0b11111111111111111111111111111111


@ Misc.
.equ    numBulkRegB,    16      @ Number of registers written to in
                                @  initialization loop for PIO B
.equ    numBulkRegC,    8       @ Number of register to write in initialization
                                @  loop for PIO C

.equ    PCK3IntMsk,  0x00000001  @ Value in PIO_ISR when there is a PCK3 interrupt
.equ    PCK0IntMsk,  0x00000002  @ Value in PIO_ISR when there is a PCK0 interrupt
.equ    FIFOHFMsk,   0x00000100  @ Value for a fifo half-full interrupt


@ PIO A pin names
.equ    FIFOWrite,  0b00000000000000000000000000000010  @ (also sample clock)
.equ    FIFORead,   0b00000000000000000000000000100000
.equ    FIFOReset,  0b00000000000000000000000000001000
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                   AIC.inc                                    @
@                     ARM Processor Initialization Routines                    @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for initializing the interrupt controller registers

.equ    SMR2Val,     0x000000F0
.equ    SMR11Val,    0x000000F0
.equ    IECRVal,     0x00000004      @ Enables PIOA and UDP interrupts
.equ    IDCRVal,     0xFFFFFFFB

.equ    CLEAR_PCK3_INT, 0x00000004
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@ crt0.s                                                                       @
@                                                                              @
@ Initialization file for EE52 ARM VoIP phone project.  It sets up the IRQ     @
@ vector table, initializes the stacks for both the IRQ and System modes,      @
@ sets up the Master Clock, all of the chip selects for external memories,     @
@ and will eventually call all of the intialization functions for each         @
@ hardware block.  Finally it invokes the main user interface code.            @
@                                                                              @
@                                                                              @
@ Revision History:                                                            @
@                                                                              @
@   2008/02/02  Joseph Schmitz  Modified code from Arthur Chang to make it     @
@                               available to the students.                     @
@   2011/01/27  Joseph Schmitz  Split into crt0.s and boot.s.                  @
@   2011/01/31  Joseph Schmitz  Removed unused comments.                       @
@   2012/01/24  Glen George     Updated comments, modified included files,     @
@                               and cleaned up the code a little.              @
@   2012/01/29  Glen George     Added comments explaining initialization.      @
@   2015/05/16  Daniel Andrade  Added own initialization function calls        @
@   2015/08/29  Daniel Andrade  Finalized init functions, comments             @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

.include     "at91rm9200.inc"
.include     "system.inc"

.text
.arm

@@@ Exception Vectors @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@
@ Start of the IRQ vector table.  This defines the interrupt hander for each
@ type of interrupt.  Must be at the memory address 0x0 (remapped at boot).
@
@ Exception                 Description
@
@ Reset                     Occurs when the processor reset pin is asserted.
@                           This exception is only expected to occur for
@                           signalling power-up, or for resetting as if the
@                           processor has just powered up. A soft reset can be
@                           done by branching to the reset vector (0x0000).
@
@ Undefined Instruction     Occurs if neither the processor, or any attached
@                           coprocessor, recognizes the currently executing
@                           instruction.  Software Interrupt (SWI) This is a
@                           user-defined synchronous interrupt instruction. It
@                           allows a program running in User mode, for example,
@                           to request privileged operations that run in
@                           Supervisor mode, such as an RTOS function.
@
@ Sofwarte Interrupt        Occurs when the processor generates a software
@                           interrupt.
@
@ Prefetch Abort            Occurs when the processor attempts to execute an
@                           instruction that has prefetched from an illegal
@                           address.
@
@ Data Abort                Occurs when a data transfer instruction attempts to
@                           load or store data at an illegal address.
@
@ Vector 6                  Used to specify the number of bytes to download from
@                           and external boot memory into internal SRAM on
@                           reset.
```

```
@
@ IRQ                       Occurs when the processor external interrupt request
@                           pin is asserted (LOW) and the I bit in the CPSR is
@                           clear.
@
@ FIQ                       Occurs when the processor external fast interrupt
@                           request pin is asserted (LOW) and the F bit in the
@                           CPSR is clear.
@
@
@ The format is as follows:
@
@   reset:    Reset vector
@   undef:    Undefine instruction
@   swi:      Software interrupt
@   abort:    Prefect abort
@   data:     Data abort
@   btldr:    Defines how much data to download from the boot memory
@   irq:      Normal interrupt (low priority).  Actual ISR address loaded from AIC
@   fiq:      Fast interrupt (high priority)

IRQTable:
.org 0x0

    reset:
        B   _start
    undef:
        B   undef
    swi:
        B   swi
    prefetch:
        B   prefetch
    data:
        B   data
    btldr:
        .word 0x0000000B
    irq:
        LDR PC, [PC, #-0xF20]
    fiq:
        B   fiq

.org 0x20
.global _start
_start:

.global low_level_init
low_level_init:

@ Stack and IRQ Initialization

    LDR     r0, =TOP_STACK      @ get the location of the top of the stack

    @ put the CPU in interrupt mode and set the stack pointer for this mode
    MSR     cpsr_c, #ARM_MODE_IRQ | I_BIT | F_BIT
    MOV     sp, r0

    @ adjust the starting stack address for the interrupt stack just setup
    SUB     r0, r0, #IRQ_STACK_SIZE

    @ put the CPU in supervisor mode and set the stack pointer for this mode
    MSR     cpsr_c, #ARM_MODE_SVC | I_BIT | F_BIT
    MOV     sp, r0

    @ adjust the starting stack address for the supervisor mode stack just setup
    SUB     r0, r0, #SVC_STACK_SIZE
```

```
    @ finally, put the CPU in user mode and set the stack pointer for this mode
    MSR     cpsr_c, #ARM_MODE_USR | F_BIT
    MOV     sp, r0

                                        @ Gets addresses in order to set
    @LDR     r1,=usb_interrupt_handler  @  breakpoints in OCD Commander. This
    LDR     r2,=update_void_fraction    @  is debugging code, it can be removed.
    LDR     r3,=disp_str    @ Load up the start screen, and set the display
    BL      set_display
    BL      init_power      @ Initializes clocks and power to peripherals
    BL      init_pio_b      @ Initializes PIO B for the display and keys
    BL      init_pio_a      @ Initializes PIO A for sample clock, FIFO, and ADC
    BL      init_fifo       @ Resets the FIFO
    BL      ui_init         @ Initializes user interface
    BL      init_interrupts @ Initialize interrupts on the processor

    BL      main            @ run the main function (no arguments)

    B       low_level_init      @ if main returns (shouldn't)
                                @   reinitialize everything and start
                                @   over

.section .data

disp_str:
.asciz  "Bubbly Measure!"

.end
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                    Init                                      @
@                    ARM Processor Initialization Routines                     @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  ARM initialization functions (does not include
@                       initialization for peripherals like display)
@
@  Table of Contents:
@   init_power      - Initializes the PMC (power management controller)
@   init_pio_a      - Initializes PIO A bank
@   init_pio_b      - Initializes PIO B bank
@   init_pio_c      - Initializes PIO C bank
@   init_interrupts - Initializes interrupts on the AIC
@   init_fifo       - Initializes the FIFO hardware
@
@  Revision History:
@   5/13/2015   Daniel Andrade  Initial revision
@   5/14/2015   Daniel Andrade  Added timers
@   5/16/2015   Daniel Andrade  Cleaned up and added PCK0 to test clock setup
@   5/24/2015   Daniel Andrade  Added AIC intialization
@   5/31/2015   Daniel Andrade  Got rid of timer code completely, changed
@                                 over to PCK3 interrupts instead.
@   6/03/2015   Daniel Andrade  Updated documentation
@   7/31/2015   Daniel Andrade  Wrote init_fifo implementation
@   8/29/2015   Daniel Andrade  Updated table of contents and some comments

.include "PIO.inc"
.include "PMC.inc"
.include "aic.inc"
.include "fifo.inc"
.include "at91rm9200.inc"

.section .text
.align 2
.arm


@  init_power
@
@  Description: Initializes the power management controller on the ARM
@               processor.
@
@  Operation:   Writes the proper constants to the power management controller
@               registers.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
```

```
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      05/13/2015    Daniel Andrade    initial revision
@
@
@
.global init_power
.type init_power STT_FUNC

init_power:
stmdb   sp!,{r0-r2}

load_address:
ldr     r0,=PMCAddr          @ Set up the address of the power management
                             @   controller.

set_up_MOR:                  @ Set up the main oscillator first
add     r1, r0, #PMC_MOR
ldr     r2,=MORVal
str     r2, [r1]

add     r1, r0, #PMC_SR

check_MOR:
mov     r2, #MORReadyMsk     @ To check that the main oscillator is stable
ldr     r1, [r1]             @ Read in the value in PMC_SR
and     r2, r1, r2
cmp     r2, #0               @ If this is zero, then bit is not set (not ready)
beq     check_MOR            @ Loop until ready bit is set

set_up_main:
add     r1, r0, #PMC_MCKR    @ Add offset for MCKR register
ldr     r2,=MCKRVal          @ Load the value to write
str     r2, [r1]

set_up_clk_enable:
add     r1, r0, #PMC_SCER    @ Add offset for SCER register
ldr     r2,=SCERVal          @ Load the value to write
str     r2, [r1]

set_up_PCK3:
mov     r1, r0               @ PCK3 will be used for interrupts. Set up as
add     r1, r1, #PMC_PCK3    @  slowest possible clock ( = SLW_CLK / 64 ).
ldr     r2,=PCK3Val
str     r2, [r1]

set_up_periphs:              @ Set up the peripherals that we want running
add     r1, r0, #PMC_PCER    @  (see PMC include file for details)
ldr     r2,=PCERVal
str     r2, [r1]

tear_down_power:
ldmia   sp!,{r0-r2}
```

```
bx      lr


@  init_pio_a
@
@  Description: Initializes the PIO A controller registers on the ARM
@               processor.
@
@  Operation:   Sets all of the PIOA lines as peripheal Bs, except the
@               buffer direction pins, which are set to high (output). Enable
@               interrupts on PCK3.
@
@  Arguments:           None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:
@
@  Revision History:
@      05/16/2015    Daniel Andrade    initial revision
@      05/31/2015    Daniel Andrade    enable interrupts on PCK3

.global init_pio_a
.type init_pio_a STT_FUNC

init_pio_a:
stmdb   sp!,{r0-r2}

load_pioa_address:
ldr     r0,=PIOAAddr          @ Load the address of the PIO A controller

PIOA_disableIO:
add     r1, r0, #PIO_PDR      @ Disable all the IO pins
ldr     r2,=pdrValA
str     r2, [r1]

PIOA_enableIO:
add     r1, r0, #PIO_PER      @ Enable the direction pins
ldr     r2,=perValA
str     r2, [r1]

PIOA_set_pins:
add     r1, r0, #PIO_SODR   @ Set the direction pins to output
ldr     r2,=sodrValA
str     r2, [r1]

PIOA_clear_pins:             @ Set the direction pins
add     r1, r0, #PIO_CODR
```

```
ldr     r2,=codrValA
str     r2, [r1]

PIOA_set_outputs:
add     r1, r0, #PIO_OER
ldr     r2,=oerValA
str     r2, [r1]

PIOA_set_input:
add     r1, r0, #PIO_ODR
ldr     r2,=odrValA
str     r2, [r1]

PIOA_enable_per_B:
add     r1, r0, #PIO_BSR     @ Set the other pins to peripheral B
ldr     r2,=bsrValA
str     r2, [r1]

PIOA_enable_ints:
add     r1, r0, #PIO_IER     @ Enable interrupts on PCK3
ldr     r2,=ierValA
str     r2, [r1]

tear_down_PIOA:
ldmia   sp!,{r0-r2}

bx      lr


@  init_pio_b
@
@  Description: Initializes the PIO controller registers on the ARM
@               processor.
@
@  Operation:   Writes the proper constants to the PIO controller
@               registers by looping through constant arrays defined
@               in the data section.
@
@  Arguments:           None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:
@
@  Revision History:
@      05/13/2015    Daniel Andrade    initial revision

.global init_pio_b
```

```
.type init_pio_b STT_FUNC

init_pio_b:
stmdb   sp!,{r0-r5}

load_piob_address:
ldr     r0,=PIOBAddr        @ Load the address of the PIO B controller

PIOB_loop_init:
mov     r3, #0              @ Initialize the loop counter
ldr     r1,=AddrArray       @ Pointers to the beginning of the arrays
ldr     r2,=ValArray

set_up_PIOB_reg_loop:
ldr     r4, [r1, r3, lsl #2]    @ Load address offset
add     r4, r0, r4              @ Add the base address in
ldr     r5, [r2, r3, lsl #2]    @ Load value to store into register
str     r5, [r4]                @   and store it in the right address

add     r3, r3, #1
cmp     r3, #numBulkRegB - 1    @ End of loop check
bls     set_up_PIOB_reg_loop

tear_down_PIOB:
ldmia   sp!,{r0-r5}

bx      lr

@  init_pio_c
@
@  Description: Initializes the PIO controller registers on the ARM
@               processor.
@
@  Operation:   Writes the proper constants to the PIO controller
@               registers by looping through constant arrays defined
@               in the data section.
@
@  Arguments:           None.
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:
@
@  Revision History:
@      06/03/2015    Daniel Andrade    initial revision

.global init_pio_c
.type init_pio_c STT_FUNC
```

```
init_pio_c:
stmdb   sp!,{r0-r5}

load_pioc_address:
ldr     r0,=PIOCAddr            @ Load the address of the PIO C controller

PIOC_loop_init:
mov     r3, #0                  @ Initialize the loop counter
ldr     r1,=AddrArrayC          @ Pointers to the beginning of the arrays
ldr     r2,=ValArrayC

set_up_PIOC_reg_loop:
ldr     r4, [r1, r3, lsl #2]    @ Load address offset
add     r4, r0, r4              @ Add the base address in
ldr     r5, [r2, r3, lsl #2]    @ Load value to store into register
str     r5, [r4]                @   and store it in the right address

add     r3, r3, #1
cmp     r3, #numBulkRegC - 1    @ End of loop check
bls     set_up_PIOC_reg_loop

tear_down_PIOC:
ldmia   sp!,{r0-r5}

bx      lr


@  init_interrupts
@
@  Description: Sets up the AIC to handle the interrupts.
@
@  Operation:   Writes the proper constants to the AIC registers.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:
@
@  Revision History:
@      05/24/2015    Daniel Andrade    initial revision
@
.global init_interrupts
.type init_interrupts STT_FUNC

init_interrupts:
stmdb   sp!,{r0-r2}
```

```
ldr     r0,=AIC_SPU     @ The following interrupts should not occur, so, if
str     r0, [r0]        @  they do, let's put them in an infinite loop so
                        @  that we at least have some kind of idea of what
ldr     r0,=AIC_SVR0    @  might have happened (which interrupt threw us
str     r0, [r0]        @  off).

ldr     r0,=AIC_SVR1
str     r0, [r0]

ldr     r0,=AIC_SMR2    @ Set up the PIO A interrupt, which is caused by
ldr     r1,=SMR2Val     @  the 512 Hz clock changing
str     r1, [r0]

ldr     r0,=AIC_SVR2
ldr     r1,=clock_interrupt
str     r1, [r0]

@ldr     r0,=AIC_SMR11   @ Set up the USB interrupt
@str     r1, [r0]

@ldr     r0,=AIC_SVR11
@ldr     r1,=usb_interrupt_handler

ldr     r0,=AIC_IDCR    @ Disable all the interrupts that we're not interested
ldr     r1,=IDCRVal     @  in.
str     r1, [r0]

ldr     r0,=AIC_IECR    @ And enable the ones that we are interested in.
ldr     r1,=IECRVal
str     r1, [r0]

it_tear_down:
ldmia   sp!,{r0-r2}
bx      lr


@  init_fifo
@
@  Description: Resets the FIFO so that it is ready.
@
@  Operation:   Sets the read and write lines of the FIFO to high, then
@                 sets the reset line low to reset it. It then waits a little
@                 bit to ensure that timing requirements are met, then sets
@                 reset high again.
@               NOTE: There is a timing requirement on when the read and write
@                 lines can go low again after reset. This function does not
@                 ensure this, but it should hold because those lines won't
@                 change until their respective parts are initialized later.
@
@  Arguments:           None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
```

```
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:
@
@  Revision History:
@      07/31/2015    Daniel Andrade    initial revision
@
.global init_fifo
.type init_fifo STT_FUNC

init_fifo:
stmdb   sp!, {r0-r2}
ldr     r0, =PIOAAddr
add     r1, r0, #PIO_CODR
mov     r2, #FIFOReset
str     r2, [r1]

mov     r1, #CLOCKS_FIFO_WAIT

if_wait:
subs    r1, r1, #1          @ NOTE: Overestimates the wait time, but
bne     if_wait             @  that's okay because this is during
                            @  initialization.
add     r1, r0, #PIO_SODR
str     r2, [r1]

ldmia   sp!, {r0-r2}
bx      lr


.section .data
.align 2

@ Arrays for cleaner (sequential) loading of PIO register values
ValArray:
.word   perValB, pdrValB, oerValB, odrValB, iferValB,  ifdrValB, sodrValB, codrValB, i
erValB, idrValB, mderValB, mddrValB, pudrValB, puerValB, owerValB, owdrValB

AddrArray:
.word   PIO_PER, PIO_PDR, PIO_OER, PIO_ODR, PIO_IFER, PIO_IFDR, PIO_SODR, PIO_CODR, PI
O_IER, PIO_IDR, PIO_MDER, PIO_MDDR, PIO_PUDR, PIO_PUER, PIO_OWER, PIO_OWDR

AddrArrayC:
.word   PIO_PER, PIO_OER, PIO_ODR, PIO_IFER, PIO_IDR, PIO_MDDR, PIO_PUDR, PIO_OWDR

ValArrayC:
.word   perValC, oerValC, odrValC, iferValC, idrValC, mddrValC, pudrValC, owdrValC

.end
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                   Handlers                                   @
@                              Interrupt Routines                              @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  Interrupt handlers
@
@  Table of Contents: clock_interrupt - Deals with interrupts from PIOA, in
@                       particular PCK interrupts, since those are the only
@                       ones allowed to interrupt the processor on that PIO
@                       bank.
@
@  Revision History:
@   06/03/2015   Daniel Andrade  Initial revision (moved clock_interrupt from
@                                    display.s)
@   07/11/2015   Daniel Andrade  Updated comments a little.
@   07/18/2015   Daniel Andrade  Changed up clock_interrupt to handle PCK0
@                                    interrupts (from the sample clock) as
@                                    well as the constant-time PCK3 interrupts

.include "aic.inc"
.include "pio.inc"
.include "at91rm9200.inc"

.extern keys
.extern display
.extern adc_handler

@ clock_interrupt
@
@ Description: This function gets called on a PCK3 interrupt (actually, PIOA
@              interrupt, but that is the only source of interrupts there).
@
@  Operation:  This function calls display and keys and clears the interrupt.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
```

```
@
@  Revision History:
@      05/31/2015    Daniel Andrade    initial revision
@

.global clock_interrupt
.type clock_interrupt STT_FUNC

clock_interrupt:
sub     lr, lr, #4          @ construct the return address
stmfd   sp!, {lr}           @ and push the adjusted lr_IRQ

stmdb   sp!, {r0-r1}

ldr     r0,=PIOAAddr        @ Read PIOA_ISR to clear the interrupt and
add     r0, r0, #PIO_ISR    @  determine which kind of interrupt we're
ldr     r0, [r0]            @  dealing with.

bl      display             @ Update display
bl      keys                @ Check for a key press
bl      update_time_var     @ Update the time variable we're keeping track of

ci_tear_down:
ldr     r0,=AIC_ICCR        @ Clear the interrupt in the interrupt controller
ldr     r1,=CLEAR_PCK3_INT
str     r1, [r0]

ldr     r0,=AIC_EOICR       @ This is the end of an interrupt, so write to EOI
mov     r1, #CLEAR_PCK3_INT @ Any value will do, just placeholder
str     r1, [r0]            @ Write to the EOI register to indicate we're done

ldmia   sp!, {r0-r1}
ldmfd   sp!, {pc}^          @ return from IRQ.

.end
```

D.4 Character Display

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                 Display.inc                                  @
@                               Display Routines                               @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for display

@ General
.equ    RS_COMM,    0x0     @ Value of RS for sending commands to display
.equ    RS_DATA,    0x1     @ Value of RS for sending data to display

@ Initialization
.equ    firstData,  0b00110000
.equ    secondData, 0b00001100
.equ    thirdData,  0b00000110

@ Other
.equ    DISP_SIZE,       0x10        @ Number of characters on a line of display
.equ    RES_MEM_COMM,    0b00000010  @ Command to set VRAM address back to zero
                                     @  on the display
.equ    COMM_LEN,        0x3         @ Length of command array
.equ    BUFF_LEN,        16          @ Length of display buffer (in chars)
.equ    TERM_CHAR,       0x0         @ Null termination character
.equ    CHAR_SPACE,      0b00100000  @ Space character
.equ    ENABLE_POS,      2           @ PIOB corresponding to enable
.equ    RS_POS,          1           @ PIOB corresponding to RS
.equ    DATA_POS,        4           @ PIOB corresponding to DATA_POS

.equ    DISP_CNT_LONG,   5           @ To slow down the system (there's a better
                                     @   way of doing this I'm sure...)
.equ    DISP_CNT_SHORT, 2


@ Masks
.equ    DATA_MSK,        0xFFFFF00D
.equ    EN_MSK,          0xFFFFFFFB
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                                @
@                                   Display                                      @
@                               Display Routines                                 @
@                                    EE 53                                       @
@                                                                                @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  Display functions
@
@  Table of Contents: set_display - Sets the display buffer to a string
@                     display     - Updates the display, gets called on a
@                                   TC0 interrupt.
@                     set_display_pios - Sets the PIOB pins to the appropriate
@                                         values (helper function for display)
@
@  Revision History:
@   5/14/2015   Daniel Andrade  Initial revision
@   5/16/2015   Daniel Andrade  Added actual content
@   5/17/2015   Daniel Andrade  Major revisions of display function
@   5/19/2015   Daniel Andrade  Added set_display_pios
@   5/20/2015   Daniel Andrade  Started debugging
@   5/24/2015   Daniel Andrade  Code compiles, but crashes. Timer interrupts
@                                 seem to be displeased with how I treated
@                                 them.
@   5/27/2015   Daniel Andrade  Fixed the crash, but timers are horribly
@                                 inconsistent? Definitely interrupting
@                                 faster than I told them to.
@   5/29/2015   Daniel Andrade  Display is working, but it's kind of cheating
@                                  because it's counting to really large
@                                  values.
@   5/31/2015   Daniel Andrade  Gave up on using the timers, used a
@                                   programmable clock and now it's much
@                                   better. Moved interrupt handler into its
@                                   own function so display only does display
@                                   things.
@   8/22/2015   Daniel Andrade  Stopped unnecessarily multiplexing the display.

.include "PIO.inc"
.include "aic.inc"
.include "display.inc"
.include "general.inc"
.include "at91rm9200.inc"

.section .text
.align 2
.arm


@ set_display
@
@ Description:          Takes a string pointer and initializes the display
@                       buffer to the contents of that string. If the string
@                       is longer than 16 characters, only the first 16
@                       characters will be stored. The string should be null
@                       terminated otherwise.
@
@  Operation:           Writes the correct values to disp_buffer. Also fills
@                       disp_buffer with spaces if null-termination occurs.
@
```

```
@  Arguments:          r3 - pointer to byte array with string contents.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   disp_buffer - String to write to display (w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        Expects null termination.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      05/17/2015    Daniel Andrade    initial revision
@

.global set_display
.type display STT_FUNC

set_display:
stmdb   sp!,{r0-r2}

sd_start:
mov     r0, #0          @ Initialize counter
ldr     r2,=disp_buffer @ Load address of display buffer

sd_loop:
ldrb    r1, [r3, r0]
cmp     r1, #TERM_CHAR  @ Is this the null-termination character?
beq     sd_fill_spaces  @ If so, fill the rest of the array with spaces

strb    r1, [r2, r0]    @ Store this character in the buffer

sd_update_counter:
add     r0, r0, #1      @ Increment counter
cmp     r0, #BUFF_LEN - 1
bls     sd_loop
b       sd_tear_down

sd_fill_spaces:
mov     r1, #CHAR_SPACE
strb    r1, [r2, r0]    @ Store a space instead
add     r0, r0, #1
cmp     r0, #BUFF_LEN - 1
bls     sd_fill_spaces

sd_tear_down:
ldr     r1, =display_changed    @ Just updated the display buffer, so indicate
mov     r0, #TRUE               @   that there's a new value there.
strb    r0, [r1]

ldmia   sp!,{r0-r2}
bx      lr
```

```
@ display
@
@ Description: This function gets called on clock interrupt so that the
@               display is always updating. The display demands that we do the
@               following:
@                   1. Set enable bit high and wait
@                   2. Set RS and data pins accordingly and wait
@                   3. Set enable bit low and wait
@
@               Therefore, it takes 3 timer interrupts to send a character or
@               command to the display. The display should also be initialized
@               at the beginning.
@
@  Operation:  The function is like a small state machine. In order to meet
@               the requirements above, we do the following. First, check
@               if enable is low. In this case, we set it high and wait. If
@               enable is high and we haven't sent any data or command yet,
@               then we send a data and command and wait. Finally, if
@               enable is high and we've sent data this cycle, then set enable
@               low and wait.
@              In addition to this, the function also determines what kind
@               of data to send by looking at comm_count. If this is higher
@               than COMM_LEN, then we've sent all commands, so we're sending
@               data. Otherwise we send a command from the command array. When
@               we reach the last character in the display, we have to send
@               a command to tell the display cursor to go back to zero, this
@               is done by changing comm_count.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   comm_count   - Current command to send. If greater than
@                       COMM_LEN, then not sending a command. (r/w)
@                      comm_arr     - Array of commands for simplicity of code (r)
@                      data_counter - Counter for display buffer (r/w)
@                      enable_state - Current state of enable line (r/w)
@                      sent_data    - Whether we've sent data this cycle or not (r/w)
@                      disp_buffer  - Display buffer containing the characters
@                       we're currently displaying. (r)
@                      display_changed - Whether the display buffer has changed
@                        since the last update (r/w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             Display pins on PIOB.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      05/16/2015    Daniel Andrade    initial revision
@      05/17/2015    Daniel Andrade    major revision for timing compliance
@      08/22/2015    Daniel Andrade    stopped multiplexing the display (no need)
```

```
@       08/23/2015    Daniel Andrade    debugged and finished the code for the
@                                        above.

.global display
.type display STT_FUNC

display:
stmdb   sp!,{r0-r3}

check_counter:
ldr     r0,=disp_count      @ Load the counter, decrement it, and compare
ldrb    r2, [r0]            @  it to the zero that we want. If it's not the
sub     r2, r2, #1          @  same, just move on for now.
strb    r2, [r0]
cmp     r2, #0
bne     tear_down_display
ldr     r1,=comm_count      @ Otherwise, let's do something with the display
ldrb    r1, [r1]            @  and set the counter to the correct value again.
cmp     r1, #COMM_LEN       @  (depends on whether we're sending a command
ldrhi   r2,=DISP_CNT_SHORT  @  or not)
ldrle   r2,=DISP_CNT_LONG
strb    r2, [r0]

check_changed:
ldr     r0,=display_changed
ldrb    r0, [r0]
cmp     r0, #TRUE
bne     tear_down_display

check_enable:
ldr     r2,=enable_state    @ If enable is cleared, then need to set it and
ldrb    r2, [r2]            @   can't send any data now
cmp     r2, #FALSE
beq     set_enable

check_s_data:
ldr     r2,=sent_data       @ Have we sent data in this cycle yet?
ldrb    r2, [r2]
cmp     r2, #TRUE           @ If so, then we end the cycle and clear enable
beq     clear_enable

check_data_type:            @ At this point, we know we're sending something,
ldr     r2,=comm_count      @   but are we sending a command or a character?
ldrb    r2, [r2]
cmp     r2, #COMM_LEN
bhi     send_char

send_comm:                  @ In this case we're sending a command to the
mov     r0, #RS_COMM        @   display, so set RS appropriately
ldr     r1,=comm_arr        @ Load up the correct command to send
ldrb    r1, [r1, r2]        @   from the command array
mov     r2, #TRUE           @ Enable always high when sending data and we
mov     r3, #TRUE           @   actually care about setting the bits
push    {lr}
bl      set_display_pios
pop     {lr}

ldr     r1,=comm_count
ldrb    r0, [r1]
add     r0, r0, #1          @ Increment the counter variable and store it
strb    r0, [r1]            @   in its proper place

cmp     r0, #COMM_LEN
bls     send_comm_set_data
ldr     r0, =data_counter
```

```
ldrb    r0, [r0]
cmp     r0, #DISP_SIZE
blo     send_comm_set_data

ldr     r2,=display_changed @ Since we're at the end of the cycle, set this
mov     r1, #FALSE          @  variable to FALSE to avoid multiplexing
strb    r1, [r2]

send_comm_set_data:
ldr     r1,=sent_data
mov     r2, #TRUE
strb    r2, [r1]
b       tear_down_display

send_char:
mov     r0, #RS_DATA        @ Sending a character in this case
ldr     r2,=data_counter
ldrb    r2, [r2]
ldr     r1,=disp_buffer     @ Load up the display buffer and get the right
ldrb    r1, [r1, r2]        @  character from there.
mov     r2, #TRUE
mov     r3, #TRUE
push    {lr}
bl      set_display_pios    @ Set the PIOs accordingly
pop     {lr}

ldr     r1,=sent_data
mov     r2, #TRUE
strb    r2, [r1]

update_counter:             @ Will need to update the data counter in this case
ldr     r0, =data_counter
ldrb    r2, [r0]
add     r2, r2, #1
cmp     r2, #DISP_SIZE - 1
bls     update_counter_nr   @ No need to reset in this case

update_counter_r:
mov     r2, #0              @ Otherwise, set the counter back to zero
strb    r2, [r0]
ldr     r2,=comm_count      @ Also have to reset the display memory counter
mov     r1, #COMM_LEN       @   so we're sending the last command in the
strb    r1, [r2]            @   array.
b       tear_down_display   @ And we're done for now

update_counter_nr:
strb    r2, [r0]            @ Update the counter variable in memory
b       tear_down_display   @  and we're done

set_enable:
ldr     r2,=enable_state
mov     r3, #TRUE
strb    r3, [r2]
mov     r2, #TRUE           @ Just set the enable line. Here we're setting up
mov     r3, #FALSE          @   the arguments to set_display_pios
push    {lr}
bl      set_display_pios
pop     {lr}
b       tear_down_display

clear_enable:
ldr     r2,=enable_state
mov     r3, #FALSE
strb    r3, [r2]
ldr     r2,=sent_data       @ Done with cycle, so reset sent_data as well
```

```
mov     r0, #FALSE
strb    r0, [r2]            @  (indicates new cycle begins)
mov     r2, #FALSE
mov     r3, #FALSE
push    {lr}
bl      set_display_pios
pop     {lr}

tear_down_display:
ldmia   sp!, {r0-r3}        @ restore registers
bx      lr



@ set_display_pios
@
@ Description: Sets the PIOB registers to the arguments given.
@
@  Operation:  If r3 is set, then we are currently sending data or a command
@              to the display. Thus, the data and RS pins are set/reset
@              appropriately and the enable pin is not touched. Otherwise,
@              we are just changing the value of enable, so only the enable
@              pin is set/reset appropriately.
@
@  Arguments:          r0 - Value of RS to send
@                      r1 - Data pins to send
@                      r2 - Value of enable to send
@                      r3 - If this is reset, only enable is changed (useful)
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             PIOB pins for display control.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      05/19/2015    Daniel Andrade    initial revision
@
.type set_display_pios STT_FUNC

set_display_pios:
stmdb   sp!,{r4-r6}
ldr     r4,=PIOBAddr
add     r4, r4, #PIO_ODSR   @ Get the status of all the PIOs
ldr     r4, [r4]

sdp_chk_only_enable:
cmp     r3, #FALSE
beq     sdp_only_enable
```

```
sdp_all:
ldr     r6,=DATA_MSK
and     r4, r4, r6
orr     r5, r4, r0, lsl #RS_POS
orr     r5, r5, r1, lsl #DATA_POS
ldr     r6,=PIOBAddr
add     r6, r6, #PIO_ODSR   @ Update everything except enable pin
str     r5, [r6]
b       sdp_tear_down

sdp_only_enable:
ldr     r6,=EN_MSK
and     r4, r4, r6
orr     r5, r4, r2, lsl #ENABLE_POS
ldr     r6,=PIOBAddr
add     r6, r6, #PIO_ODSR   @ Updates the IO pin with the value of enable
str     r5, [r6]

sdp_tear_down:
ldmia   sp!,{r4-r6}
bx      lr

.section .data
.align 2

comm_count:
.byte   0

@ Which character we're currently trying to send to the display
data_counter:
.byte   0

@ Whether enable bit is currently high or low
enable_state:
.byte   LOW

@ Whether we have sent data yet this time around
sent_data:
.byte   FALSE

@ Boolean to see if we should update the contents of the display
display_changed:
.byte   TRUE

@ For getting the timing correct
disp_count:
.byte   DISP_CNT_LONG

@ Command array for sending commands
comm_arr:
.byte  firstData, secondData, thirdData, RES_MEM_COMM

@ Display buffer
disp_buffer:
.asciz "Testing the keys"
```

D.5 String Manipulation

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                 Converts.inc                                 @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for convert functions (to ASCII representation)
.equ    ASCII_ZERO,     0x30
.equ    ASCII_NULL,     0x0

.equ    MAX_DIGIT_NUM,  1000000000  @ Largest power of ten divisor that fits
                                    @  in 32 bits.

.equ    RIGHT_JUST_NUM, 6       @ Number to add to the beginning of a 16
                                @  character string so that the number
                                @  appears right-justified
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                  Converts                                    @
@                              Display Routines                                @
@                                   EE 53                                      @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  Conversion functions (to ASCII).
@
@  Table of Contents:
@   dec2string - Gives the unsigned, decimal representation of a number
@       as a zero-padded, null-terminated ASCII string.
@   divu       - Does unsigned division, as the ARM instruction set has
@                  no support for this.
@
@  Revision History:
@   6/1/2015   Daniel Andrade  Initial revision

.include "general.inc"
.include "converts.inc"

.section .text
.align 2
.arm

@ dec2string
@
@  Description:        Converts an unsigned binary number into its decimal
@                       representation as a zero-padded string in ASCII. The
@                       string is always null-terminated.
@
@  Operation:          This was translated to ARM assembly from the asm86
@                       Dec2StringUnsigned that I wrote in EE/CS 51.
@
@  Arguments:          r0 - 32-bit unsigned value to convert
@                      r1 - Memory address pointer at which to store string.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        For unsigned numbers only. Does no check for length
@                       of buffer, could potentially overflow. Be careful.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
```

```
@  Revision History:
@      06/1/2015    Daniel Andrade    initial revision
@      06/2/2015    Daniel Andrade    commenting and touching up a few things

.global dec2string
.type dec2string STT_FUNC

dec2string:
stmdb   sp!,{r0-r4}
ldr     r2,=MAX_DIGIT_NUM       @ Load the divisor

dec2loop:
cmp     r2, #0                  @ Done if divisor is equal to zero. Should
bls     d2s_tear_down           @  not be less, but might as well check.

push    {lr}
bl      divu                    @ Perform the division
pop     {lr}

add     r0, r0, #ASCII_ZERO     @ Convert the quotient into ASCII
strb    r0, [r1]                @  and store it in the buffer
add     r1, r1, #1              @ The devisor becomes the dividend now,
mov     r0, r2                  @  we want to divide it by ten because
mov     r2, #10                 @  of the decimal representation.
mov     r4, r3                  @ Store the remainder so as not to lose it

push    {lr}
bl      divu                    @ Perform the second division
pop     {lr}

mov     r2, r0                  @ Shuffle things around for next iteration
mov     r0, r4
b       dec2loop

d2s_tear_down:
mov     r0, #ASCII_NULL         @ Finally terminate the string
strb    r0, [r1]

ldmia   sp!,{r0-r4}
bx      lr



@ divu
@
@  Description:         Performs 32-bit unsigned division.
@
@  Operation:           Long division (in binary) implemented with shifts.
@
@  Arguments:           r0 - Dividend
@                       r2 - Divisor
@
@  Return Value:        r0 - Quotient
@                       r3 - Remainder
@
@  Local Variables:     r4 - Temporary storage for quotient.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
```

```
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  r3 (to return the remainder)
@
@  Revision History:
@      06/1/2015    Daniel Andrade    initial revision
@      06/2/2015    Daniel Andrade    comments
@
@  Pseudocode (from Wikipedia) (this is just long division):
@     Q := 0                  -- initialize quotient and remainder to zero
@     R := 0
@     for i = n-1...0 do      -- where n is number of bits in N
@         R := R << 1         -- left-shift R by 1 bit
@         R(0) := N(i)        -- set the least-significant bit of R equal to
@          if R >= D then     --   bit i of the numerator
@              R := R - D
@              Q(i) := 1
@     end
@  end

.global divu
.type divu STT_FUNC

divu:
stmdb   sp!,{r4-r6}
mov     r3, #0                  @ Initialize the quotient and remainder
mov     r4, #0
mov     r5, #NUM_BITS_WORD - 1  @ Initialize loop counter

divu_loop:
cmp     r5, #0
blt     divu_tear_down          @ Done if counter is less than zero
mov     r3, r3, lsl #1          @ Otherwise, shift remainder left once
mov     r6, #1                  @   and set the least-significant bit
mov     r6, r6, lsl r5          @   of R, but only if the i(th) bit of N was
ands    r6, r6, r0              @   set
movne   r6, #1
orr     r3, r3, r6
cmp     r3, r2
blo     divu_update_counter

sub     r3, r3, r2              @ If R >= D, then subtract D from R and
mov     r6, #1                  @  set the i(th) bit of the quotient
orr     r4, r4, r6, lsl r5

divu_update_counter:
sub     r5, r5, #1
b       divu_loop

divu_tear_down:
mov     r0, r4
ldmia   sp!,{r4-r6}
bx      lr
```

D.6 Keypad

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                 Keypad.inc                                   @
@                               Keypad Routines                                @
@                                   EE 53                                      @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for the keypad

@ Constants representing each different key
.equ    KEY_NONE,       0
.equ    KEY_TOGGLE,     1
.equ    KEY_DEBUG,      2
.equ    KEY_LEFT,       3
.equ    KEY_RIGHT,      4

@ Debouncing time constants
.equ    DEB_LEN,        50          @ Number of interrupts to get before
                                    @  key is considered debounced
.equ    CP_LEN,         255         @ Number of interrupts to get before
                                    @  key is considered pressed again (if held
                                    @  down)

@ PIO constants for keys
.equ    TOGGLE_PIO_MASK,    0x04000000      @ To check if toggle is pressed
.equ    DEBUG_PIO_MASK,     0x02000000      @ To check if debug is pressed
.equ    B_PIO_MASK,         0x08000000      @ To check if A signal is high
.equ    A_PIO_MASK,         0x10000000      @ To check if B signal is high

@ Mapping
@   TOGGLE - PIOB, pin 26
@   DEBUG  - PIOB, pin 25
@   B      - PIOB, pin 27
@   A      - PIOB, pin 28
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                   Keypad                                     @
@                                Key Routines                                  @
@                                   EE 53                                      @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  Keypad functions (for keys and rotary encoder)
@
@  Table of Contents: encoder_key   - Decodes rotary encoder
@                     key_press     - Returns which key is pressed (if any)
@                     keys          - Debounces keys (entry point)
@                     key_available - Returns whether a key has been pressed
@                                       since get_key() was last called.
@                     get_key       - Returns the last key pressed.
@
@  Revision History:
@   5/28/2015   Daniel Andrade  Initial revision
@   5/30/2015   Daniel Andrade  Debugging, combined handler with display, and
@                                 added main body of code
@   6/01/2015   Daniel Andrade  Added code for rotary encoder
@   6/03/2015   Daniel Andrade  Updated documentation


.include "PIO.inc"
.include "keypad.inc"
.include "general.inc"
.include "at91rm9200.inc"
.include "timers.inc"

.section .text
.align 2
.arm

@ encoder_key
@
@  Description:        Decodes the rotary encoder and returns whether
@                       it turned and in which direction.
@
@  Operation:          See attached diagram. Just decodes the quadrature
@                       output of the decoder. This was modified into code
@                       from my digital logic from EE/CS 52.
@
@  Arguments:          None.
@
@  Return Value:       KEY_NONE if the encoder did not move, KEY_LEFT if the
@                       encoder rotated left, and KEY_RIGHT if it rotated
@                       right, in r0.
@
@  Local Variables:    None.
@
@  Shared Variables:   prevA & prevB - Previous values of A and B (r/w)
@
@  Global Variables:   None.
@
@  Input:              Reads PIO values in PIOB (in particular PIO_PDSR). See
@                       the mask constants to see where A and B are mapped, or
@                       look at the keypad include file.
@
@  Output:             None.
```

```
@
@  Error Handling:     None.
@
@  Limitations:        Does no debouncing beyond checking for a detent
@                       (should not need to do any, anyway).
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/1/2015    Daniel Andrade    initial revision

.type encoder_key STT_FUNC

encoder_key:
stmdb   sp!, {r1-r5}

ldr     r1,=PIOBAddr
add     r1, r1, #PIO_PDSR   @ Get the status of all the PIO pins
ldr     r1, [r1]

ek_get_current:
ldr     r2,=A_PIO_MASK      @ Get the current values of A and B from
ands    r2, r2, r1          @  the PIO pins
movne   r2, #HIGH           @ If the masked value is not zero, then A is high
ldr     r3,=B_PIO_MASK
ands    r3, r3, r1
movne   r3, #HIGH

ek_get_prev:
ldr     r4,=prevA           @ Get the previous values of A and B (stored
ldrb    r0, [r4]            @  in memory) and update them
strb    r2, [r4]

ldr     r4,=prevB
ldrb    r1, [r4]
strb    r3, [r4]

ek_check_detent:
ands    r4, r0, r1          @ Check that we were previously at a detent
moveq   r0, #KEY_NONE       @  position. If not, nothing to do here.
beq     ek_tear_down

ek_check_change:
eor     r4, r3, r1          @ Check that there was a change in either A
eor     r5, r0, r2          @  or B since last time. If not, then nothing
orrs    r4, r5, r4          @  to do here, either.
moveq   r0, #KEY_NONE
beq     ek_tear_down

ek_get_key:
eors    r4, r0, r3          @ If we got here, then something happened, so
moveq   r0, #KEY_LEFT       @  figure out which key was pressed to return it.
movne   r0, #KEY_RIGHT

ek_tear_down:
ldmia   sp!, {r1-r5}
bx      lr

@ key_press
@
@  Description:        Checks the PIOs for any pressed keys and returns the
@                       highest priority pressed key in r0.
```

```
@
@  Operation:           Returns whether any of the PIO ports corresponding
@                        to keys are low.
@
@  Arguments:           None.
@
@  Return Value:        The highest priority key that is pressed (in r0).
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               Reads PIO values in PIOB (in particular PIO_PDSR). See
@                        the mask constants to see where A and B are mapped, or
@                        look at the keypad include file.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      05/30/2015    Daniel Andrade    initial revision

.type key_press STT_FUNC

key_press:
stmdb   sp!, {r1-r2}

ldr     r1,=PIOBAddr
add     r1, r1, #PIO_PDSR   @ Get the status of all the PIO pins
ldr     r1, [r1]

key_check_toggle:
mov     r0, #KEY_TOGGLE
ldr     r2,=TOGGLE_PIO_MASK
ands    r2, r1, r2
beq     tear_down_kp

key_check_debug:
mov     r0, #KEY_DEBUG
ldr     r2,=DEBUG_PIO_MASK
ands    r2, r1, r2
beq     tear_down_kp

key_encoders:
mov     r0, #KEY_NONE

tear_down_kp:
ldmia   sp!, {r1-r2}
bx      lr


@ keys
@
@  Description:         Interrupt handler for keypad. Checks to see if
@                        any keys have been pressed. Also works as a debouncer.
```

```
@
@  Operation:           Calls key_press() to get which key is pressed. If
@                        the key has been pressed for long enough, it is
@                        considered debounced.
@                       If key_press() returns that no key is pressed, then
@                        the encoder is checked by calling encoder_key().
@
@  Arguments:           None.
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    key_counter - Debounce counter (r/w)
@                       last_key    - Last key to be pressed (r/w)
@                       cur_key     - Currently pressed key (w)
@                       key_pressed - Flag variable to indicate that a key
@                                     has been pressed (w)
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         Can only deal with one key press at a time. There is
@                        a priority for the keys which corresponds to the
@                        key numbers themselves. Lower key number corresponds
@                        to higher priority (except for #KEY_NONE)
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      05/29/2015    Daniel Andrade     initial revision
@      06/02/2015    Daniel Andrade     updated documentation and comments

.global keys
.type keys STT_FUNC

keys:
stmdb   sp!, {r0-r2, lr}

key_scan:
bl      key_press           @ Returns the key pressed (if any)

check_press:
cmp     r0, #KEY_NONE       @ If no key was pressed, check for an encoder
beq     no_key_press        @  rotation.

ldr     r2,=last_key        @ We want to compare this to the key pressed
ldrb    r1, [r2]            @   last time.

cmp     r0, r1              @ Is this the same key as last time?
beq     debounce_key_press

strb    r0, [r2]            @ Now this is the last key that was pressed
ldr     r2,=key_counter
mov     r0, #DEB_LEN        @ Reset the debounce counter since a new key
strb    r0, [r2]            @   was pressed.
b       tear_down_keys
```

```
debounce_key_press:
ldr     r2, =key_counter
ldrb    r1, [r2]            @ Get the value of the debounce counter
cmp     r1, #0              @ Is the key debounced?
beq     key_is_debounced

sub     r1, r1, #1          @ If not, decrease the counter and end for now
strb    r1, [r2]
b       tear_down_keys

key_is_debounced:
ldr     r2,=key_counter
mov     r1, #CP_LEN         @ Set the counter for continuous key press
strb    r1, [r2]

ldr     r2,=cur_key
strb    r0, [r2]            @ This is the currently pressed key
ldr     r2,=key_pressed     @ Set the key was pressed flag
mov     r1, #TRUE
strb    r1, [r2]
b       tear_down_keys

no_key_press:
bl      encoder_key         @ Check for encoder rotation
cmp     r0, #KEY_NONE
beq     no_key_at_all
ldr     r2,=key_pressed     @ If it changed, update key_pressed and cur_key
mov     r1, #TRUE
strb    r1, [r2]
ldr     r2,=cur_key
strb    r0, [r2]
b       tear_down_keys

no_key_at_all:
ldr     r2,=last_key        @ Indicate that no key was pressed
mov     r0, #KEY_NONE
strb    r0, [r2]

ldr     r2,=key_counter
mov     r0, #DEB_LEN        @ Reset the debounce counter since no key was
strb    r0, [r2]            @   pressed.

tear_down_keys:
ldmia   sp!, {r0-r2, lr}
bx      lr


@ key_available
@
@  Description:        Returns whether a key has been pressed since the last
@                       call to get_key().
@
@  Operation:          Returns the status flag (key_pressed)
@
@  Arguments:          None.
@
@  Return Value:       Whether a key has been pressed (in r0)
@
@  Local Variables:    None.
@
@  Shared Variables:   key_pressed - Flag variable to indicate that a key
@                                    has been pressed (r)
@
@  Global Variables:   None.
```

```
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         Assumes get_key() will be called directly afterwards
@                        if it returns #TRUE. Does not update key_pressed flag
@                        variable itself.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      05/30/2015    Daniel Andrade    initial revision

.global key_available
.type key_available STT_FUNC

key_available:
ldr     r0,=key_pressed
ldrb    r0, [r0]

bx      lr

@ get_key
@
@  Description:         Gets the currently pressed key. Should only be called
@                        if key_available() returned #TRUE.
@
@  Operation:           Returns the key that is pressed (cur_key). Sets the
@                        key_pressed status flag to #FALSE.
@
@  Arguments:           None.
@
@  Return Value:        The key that is pressed (in r0).
@
@  Local Variables:     None.
@
@  Shared Variables:    cur_key     - The currently pressed key (r)
@                       key_pressed - Flag variable to indicate that a key
@                                     has been pressed (w)
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      05/30/2015    Daniel Andrade    initial revision

.global get_key
```

```
.type get_key STT_FUNC

get_key:
stmdb   sp!, {r1-r2}
ldr     r0,=cur_key
ldrb    r0, [r0]

ldr     r1,=key_pressed
mov     r2, #FALSE
strb    r2, [r1]

ldmia   sp!, {r1-r2}
bx      lr

.section .data

@ Debouncing and continuous key press counter
key_counter:
.byte   DEB_LEN

@ Last key to be pressed
last_key:
.byte   KEY_NONE

@ Current key pressed (read from get_key)
cur_key:
.byte   KEY_NONE

@ Flag indicating whether a key was pressed
key_pressed:
.byte   FALSE


@ Previous values of A and B (the rotary encoder outputs)
prevA:
.byte   TRUE

prevB:
.byte   TRUE
```

D.7 User Interface

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                     UI                                       @
@                              Main Loop Files                                 @
@                                   EE 53                                      @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ UI constants
.equ    MAX_TOGGLE_STATE,   6     @ Number of display screens to have

.equ    DEBUG_TOGGLE_STATE, 0
.equ    AV_LOW_STATE,       1
.equ    AV_HIGH_STATE,      2
.equ    TH_STATE,           3
.equ    SMPL_RATE_STATE,    4
.equ    VOID_STATE,         5

.equ    VF_COUNTER_MAX,     2048  @ I think this is two seconds
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                      UI                                      @
@                               Main Loop Files                                @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description: Deals with the user interface for the project. Calls
@                      relevant functions for dealing with key presses, etc.
@                      Also has a function responsible for periodically updating
@                      the display.
@
@  Table of Contents:
@   ui_init             - Initializes display strings
@   ui_do_key           - Calls correct helper function for a pressed key.
@   do_toggle           - Does the action corresponding to the toggle key.
@   do_debug            - Does the action corresponding to the debug key.
@   do_left             - Does the action corresponding to the left key.
@   do_right            - Does the action corresponding to the right key.
@   get_debug_state     - Gets the debug state of the system.
@   update_time_var     - Increments the time variable.
@   auto_display_update - Updates the display strings and the display if
@                           one of those strings was being shown.
@
@  Revision History:
@   6/11/2015   Daniel Andrade  Initial revision
@   7/01/2015   Daniel Andrade  Added 'getter' functions to give other parts
@                                 of the code access to the variables.
@   7/11/2015   Daniel Andrade  Updated comments, filled in 'TODOs', etc.
@   7/26/2015   Daniel Andrade  Added some ADC stuff to the UI
@   8/14/2015   Daniel Andrade  Changed the code so the display would auto
@                                update every so often.
@   8/22/2015   Daniel Andrade  Added more auto-updating variables (threshold
@                                and averages). Removed the notion of a high
@                                and low threshold from the UI code, that's
@                                all in the analog part, now. As it should be.

.include "keypad.inc"
.include "general.inc"
.include "ui.inc"
.include "adc.inc"
.include "pmc.inc"
.include "pio.inc"
.include "converts.inc"

@  ui_init
@
@  Description:        Initializes the UI display and strings to the right
@                       values.
@
@  Operation:          Initializes the strings by calling dec2string to fill
@                       in the strings with the correct numbers.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   disp_str_5 - String corresponding to sampling frequency (w)
```

```
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@       06/13/2015    Daniel Andrade    initial revision
@       07/11/2015    Daniel Andrade    updated comments
@       08/22/2015    Daniel Andrade    took out the threshold string
@                                          initialization because of new
@                                          threshold system

.global ui_init
.type ui_init STT_FUNC

ui_init:
stmdb   sp!, {r0-r2, lr}

ldr     r0, =freqArray
ldr     r0, [r0]
ldr     r1, =disp_str_5
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string

ui_init_pios:
ldr     r0, =PIOAAddr
add     r1, r0, #PIO_PDR
mov     r2, #pdrValA2
str     r2, [r1]

add     r1, r0, #PIO_BSR
mov     r2, #bsrValA2
str     r2, [r1]

set_up_PCK0:
ldr     r1, =PMCAddr
add     r1, r1, #PMC_PCK0
ldr     r2,=PCK0Val
str     r2, [r1]

ui_init_tear_down:
ldmia   sp!, {r0-r2, lr}
bx      lr


@  ui_do_key
@
@  Description:        Calls the right function to deal with a key press.
@
@  Operation:          Takes a key and performs the correct action corresponding
@                       to that key by calling its corresponding function. Uses
@                       the function table stored in KeyFuncArray.
@
@  Arguments:          r0 - Pressed key
```

```
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/11/2015    Daniel Andrade    initial revision
@      07/11/2015    Daniel Andrade    updated documentation

.global ui_do_key
.type ui_do_key STT_FUNC

ui_do_key:
stmdb   sp!,{r1, lr}

ui_check_key_type:
ldr     r1,=KeyFuncArray
sub     r0, r0, #1
ldr     r0, [r1, r0, lsl #2]    @ Load the function to call from memory array
mov     lr, pc                  @ Store return address so that we can get back
bx      r0                      @  and call the right function

uidk_tear_down:
ldmia   sp!,{r1, lr}
bx      lr

@ do_debug
@
@  Description:        Sets the debug variable (called when DEBUG key is
@                       pressed). Updates the display if necessary.
@
@  Operation:          Toggles the debug variable in memory. If the debug
@                       state is currently being displayed, then update what's
@                       on the display.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   debug_state - Whether the system is in debug mode
@                       or not. (r/w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
```

```
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      06/11/2015    Daniel Andrade    initial revision
@      07/11/2015    Daniel Andrade    updated documentation

.type do_debug STT_FUNC

do_debug:
stmdb   sp!,{r0-r1, r3, lr}
ldr     r0,=debug_state     @ Load state of debug variable
ldrb    r1, [r0]
cmp     r1, #TRUE           @ Toggle the state of the debug variable
moveq   r1, #FALSE          @  in memory.
movne   r1, #TRUE
strb    r1, [r0]            @ And store toggled version.

dd_update_disp:
ldreq   r3,=disp_str_2      @ Update the display if we're currently outputting
ldrne   r3,=disp_str_1      @   the state of the debug variable
ldr     r1,=toggle_state
ldrb    r1, [r1]
cmp     r1, #DEBUG_TOGGLE_STATE
bne     do_debug_teardown
bl      set_display

do_debug_teardown:
ldmia   sp!,{r0-r1, r3, lr}
bx      lr


@ do_toggle
@
@  Description:        Toggles what's shown on the display.
@
@  Operation:          Determines which string to display based on the value
@                       of toggle_state and calls set_display to display the
@                       correct string.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   toggle_state - What's currently being displayed (r/w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
```

```
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/11/2015    Daniel Andrade    initial revision

.type do_toggle STT_FUNC

do_toggle:
stmdb   sp!,{r0-r4, lr}

dp_set_toggle:
ldr     r1,=toggle_state        @ Load the toggle state
ldrb    r0, [r1]

add     r0, r0, #1              @ Increment it
cmp     r0, #MAX_TOGGLE_STATE   @ Compare to its maximum and wrap
moveq   r0, #0                  @  it around if necessary.
strb    r0, [r1]                @ Store it back into memory

dt_check_debug:
cmp     r0, #DEBUG_TOGGLE_STATE @ Display the right debug message
bne     dt_check_th_l           @  if we're currently on the debug
ldr     r1,=debug_state         @  screen.
ldrb    r2, [r1]
cmp     r2, #TRUE
ldreq   r3,=disp_str_1
ldrne   r3,=disp_str_2
b       dt_change_display

dt_check_th_l:
cmp     r0, #AV_LOW_STATE
bne     dt_check_tl_r
ldr     r3,=disp_str_3          @ If we're on the 'low average' message,
b       dt_change_display       @   then display that.

dt_check_tl_r:
cmp     r0, #AV_HIGH_STATE
bne     dt_check_rate
ldr     r3,=disp_str_4          @ If we're on the 'high average' message,
b       dt_change_display       @   then display that.

dt_check_rate:
cmp     r0, #SMPL_RATE_STATE    @ If we are showing the current sample rate,
bne     dt_check_void           @   then display it here.
ldr     r3,=disp_str_5
b       dt_change_display

dt_check_void:
cmp     r0, #VOID_STATE         @ If we're currently showing the void rate
bne     dt_check_th             @   then display it here.
ldr     r3,=disp_str_6
b       dt_change_display

dt_check_th:
cmp     r0, #TH_STATE           @ If we're currently showing the threshold,
bne     dt_tear_down            @   then display it here.
ldr     r3,=disp_str_7

dt_change_display:
bl      set_display
```

```
dt_tear_down:
ldmia   sp!,{r0-r4, lr}
bx      lr


@ do_left
@
@  Description:         Does the left key press action, called when the
@                        left key is pressed by the function ui_do_key.
@
@  Operation:           Checks toggle_state to see what is currently being
@                       shown on the display. If the debug state is shown,
@                       then the debug variable is toggled. Else if the
@                       sample rate is shown, then the sample rate is decreased
@                       unless it is at a minimum, in which case there is no
@                       change. If there is a change in the sampling rate,
@                       then the display is updated. If something else is
@                       displayed, the function does nothing.
@
@  Arguments:           None.
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    toggle_state - What's currently being displayed (r/w)
@                       rate_cntr    - Counter to keep track of current sample
@                                      rate (r/w)
@                       freqArray    - Array of possible sampling frequencies (r)
@                       disp_str_5   - String which shows current sampling rate (w)
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/11/2015    Daniel Andrade    initial revision
@      07/11/2015    Daniel Andrade    updated documentation

.type do_left STT_FUNC

do_left:
stmdb   sp!, {r0-r1, r3, lr}

dl_get_state:
ldr     r0, =toggle_state
ldrb    r0, [r0]

cmp     r0, #DEBUG_TOGGLE_STATE
beq     dl_debug
cmp     r0, #SMPL_RATE_STATE
beq     dl_set_smpl_rate
b       dl_tear_down
```

```
dl_set_smpl_rate:
ldr     r1, =rate_cntr
ldrb    r0, [r1]

subs    r0, r0, #1              @ Decrement counter and set back to zero if
movls   r0, #0                  @  necessary
strb    r0, [r1]

ldr     r1, =freqArray
ldr     r0, [r1, r0, lsl #2]    @ Load the number to display from the array
ldr     r1, =disp_str_5
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string              @ Change the string in memory and display
ldr     r3, =disp_str_5         @  it.
bl      set_display

ldr     r0, =rate_cntr
ldrb    r0, [r1, r0]
ldr     r1, =freqPCK0Arr        @ Sample clock intialized here
ldrb    r0, [r1, r0]
ldr     r3, =PMCAddr
add     r3, r3, #PMC_PCK0
str     r0, [r3]
b       dl_tear_down

dl_debug:
bl      do_debug

dl_tear_down:
ldmia   sp!, {r0-r1, r3, lr}
bx      lr

@ do_right
@
@  Description:         Does the right key press action, called by ui_do_key
@                        when the right key is pressed.
@
@  Operation:           Checks toggle_state to see what is currently being
@                        shown on the display. If the debug state is shown,
@                        then the debug variable is toggled. Else if the
@                        sample rate is shown, then the sample rate is increased
@                        unless it is at a maximum, in which case there is no
@                        change. If there is a change in the sampling rate,
@                        then the display is updated. If something else is
@                        displayed, the function does nothing.
@
@  Arguments:           None.
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    toggle_state - What's currently being displayed (r/w)
@                       rate_cntr    - Counter to keep track of current sample
@                                      rate (r/w)
@                       freqArray    - Array of possible sampling frequencies (r)
@                       disp_str_5   - String which shows current sampling rate (w)
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
```

```
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/11/2015    Daniel Andrade    initial revision
@      07/11/2015    Daniel Andrade    updated documentation

.type do_right STT_FUNC

do_right:
stmdb   sp!, {r0-r1, r3, lr}

dr_get_state:
ldr     r0, =toggle_state
ldrb    r0, [r0]

cmp     r0, #DEBUG_TOGGLE_STATE
beq     dr_debug
cmp     r0, #SMPL_RATE_STATE
beq     dr_set_smpl_rate
b       dr_tear_down

dr_set_smpl_rate:
ldr     r1, =rate_cntr
ldrb    r0, [r1]

add     r0, r0, #1
cmp     r0, #numFreqs
moveq   r0, #numFreqs - 1
strb    r0, [r1]

ldr     r1, =freqArray
ldr     r0, [r1, r0, lsl #2]    @ Load the number to display from the array
ldr     r1, =disp_str_5
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string
ldr     r3, =disp_str_5
bl      set_display

ldr     r0, =rate_cntr
ldrb    r0, [r0]
ldr     r1, =freqPCK0Arr
ldrb    r0, [r1, r0]
ldr     r3, =PMCAddr
add     r3, r3, #PMC_PCK0
str     r0, [r3]
b       dr_tear_down

dr_debug:
bl      do_debug

dr_tear_down:
ldmia   sp!, {r0-r1, r3, lr}
bx      lr

@ get_debug_state
@
@  Description:        Gets whether the device is in a debug state or not.
@
```

```
@  Operation:          Returns the requested variable
@
@  Arguments:          None.
@
@  Return Value:       r0 - Value of the debug state
@
@  Local Variables:    None.
@
@  Shared Variables:   debug_state - The debug state (r)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      07/01/2015    Daniel Andrade    initial revision

.global get_debug_state
.type get_debug_state STT_FUNC

get_debug_state:
ldr     r0, =debug_state
ldrb    r0, [r0]

bx      lr

@ update_void_fraction
@
@  Description:        Updates the void fraction value to the passed-in
@                       void fraction value.
@
@  Operation:          Stores the value in memory.
@
@  Arguments:          r0 - Number of high points
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   void_frac - Void fraction value (w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
```

```
@  Registers Changed:  None.
@
@  Revision History:
@      07/25/2015    Daniel Andrade    initial revision
@      08/12/2915    Daniel Andrade    moved display updates to
@                                        update_void_disp instead

.global update_void_fraction
.type update_void_fraction STT_FUNC

update_void_fraction:
stmdb   sp!, {r1}

ldr     r1, =void_frac
str     r0, [r1]

uvf_tear_down:
ldmia   sp!, {r1}
bx      lr
```

```
@ update_time_var
@
@  Description:        Updates the time variable in memory.
@
@  Operation:          Called by clock_interrupt, merely increments a counter
@           in memory.
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      08/12/2015    Daniel Andrade    initial revision
@      08/14/2015    Daniel Andrade    reworked the logic to make it more
@                   polling-based, most of the code was
@                       moved down to auto_display_update

.global update_time_var
.type update_time_var STT_FUNC

update_time_var:
stmdb   sp!, {r0-r1}

ldr     r0, =vf_counter         @ Update the counter to properly keep track
ldr     r1, [r0]                @  of time.
```

```
add     r1, r1, #1              @ Don't need to do anything other than increment
str     r1, [r0]                @  it here.

utv_tear_down:
ldmia   sp!, {r0-r1}
bx      lr


@ auto_display_update
@
@  Description:       Updates the strings and the value shown on the display.
@
@  Operation:         Updates the variable strings and calls set_display to
@                      update the display to show the new string, if one
@                      of the changed strings was being shown.
@                     Only updates the display if enough time has passed
@                      since the last update.
@
@  Arguments:         None.
@
@  Return Value:      None.
@
@  Local Variables:   None.
@
@  Shared Variables:  disp_str_3/4/6/7 - Display strings for low average,
@                      high average, void rate, and threshold (resp.) (w)
@                     vf_counter       - 512 Hz time counter (r/w)
@                     toggle_state     - Counter indicating which screen is
@                                          being shown (r)
@
@  Global Variables:  None.
@
@  Input:             None.
@
@  Output:            None.
@
@  Error Handling:    None.
@
@  Limitations:       None.
@
@  Algorithms:        None.
@  Data Structures:   None.
@
@  Registers Changed: None.
@
@  Revision History:
@      08/14/2015    Daniel Andrade    initial revision
@      08/21/2015    Daniel Anrade     updated to update averages as well
@                                        as void rate
@      08/22/2015    Daniel Andrade    ditto above, but with the threshold, too.

.global auto_display_update
.type auto_display_update STT_FUNC

auto_display_update:
stmdb   sp!, {r0-r1, r3, lr}

ldr     r0, =vf_counter         @ Check the value of the counter to see
ldr     r1, [r0]                @  if we should update yet.
cmp     r1, #VF_COUNTER_MAX
blo     adu_tear_down

mov     r1, #0                  @ If we should update, then first reset
str     r1, [r0]                @  the counter.
```

```
@ NOTE: May not want to reset here if we are actually interested in keeping
@ track of time for longer than the update period.

adu_update_strings:
ldr     r0, =void_frac          @ Then get the value we should write to the
ldr     r0, [r0]                @  display and write it to the string
ldr     r1, =disp_str_6         @  disp_str_6.
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string

bl      get_averages            @ Also update the strings for the averages
mov     r3, r1                  @  while we're at it.
ldr     r1, =disp_str_3
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string

mov     r0, r3
ldr     r1, =disp_str_4
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string

bl      get_threshold           @ And update the threshold here
ldr     r1, =disp_str_7
add     r1, r1, #RIGHT_JUST_NUM
bl      dec2string

adu_update_disp:
ldr     r1, =toggle_state       @ Check to see if we're currently displaying
ldrb    r1, [r1]                @  the void rate.
cmp     r1, #VOID_STATE
bne     aduud_check_avl         @ If we are, then update the display to show
ldr     r3, =disp_str_6         @  the new void rate.
b       aduud_set_display

aduud_check_avl:
cmp     r1, #AV_LOW_STATE       @ Also update the display for other string
bne     aduud_check_avh         @  that may have changed above, but only
ldr     r3, =disp_str_3         @  if we're currently displaying that screen.
b       aduud_set_display

aduud_check_avh:
cmp     r1, #AV_HIGH_STATE
bne     aduud_check_th
ldr     r3, =disp_str_4
b       aduud_set_display

aduud_check_th:
cmp     r1, #TH_STATE
bne     adu_tear_down
ldr     r3, =disp_str_7

aduud_set_display:
bl      set_display

adu_tear_down:
ldmia   sp!, {r0-r1, r3, lr}
bx      lr

.section .data
.align 2

@ Function table that maps keys pressed on the board to functions that should
@  be called to perform the correct action.
KeyFuncArray:
.word   do_toggle, do_debug, do_left, do_right
```

```
@ Array of possible sampling frequencies (based on dividers on programmable
@  clocks).
freqArray:
.word   512, 1024, 2048, 4096, 8192, 16384, 32768, 312000, 625000, 1250000, 2500000, 5
000000

@ Current value of void fraction
void_frac:
.word   0

@ Counter to determine if we should update the displayed version of the
@  void fraction or not
vf_counter:
.word   0

@ Whether the system is in debug state or not
debug_state:
.byte   TRUE

@ Variable to keep track of what is currently being shown on the display
toggle_state:
.byte   0

@ Variable to keep track of what the current sampling rate is (counter to
@  freqArray above).
rate_cntr:
.byte   0

@ Array of values to write to the PCK0 register so to set the PCK0 line to
@  the right frequency.
freqPCK0Arr:
.byte   F512, F1024, F2048, F4096, F8K, F16K, F32K, F312K, F625K, F125M, F250M, F5M, F
10M, F20M


@ The following strings dictate what will be shown on the display in each
@  different toggle_state.
disp_str_1:
.asciz "DEBUG ON"

disp_str_2:
.asciz "DEBUG OFF"

disp_str_3:
.asciz "AV_LO:          "

disp_str_4:
.asciz "AV_HI:          "

disp_str_5:
.asciz "RATE:           "

disp_str_6:
.asciz "VOID:           "

disp_str_7:
.asciz "THR:            "
```

D.8 Sampling Hardware

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                                @
@                                    FIFO.inc                                    @
@                                 FIFO Routines                                  @
@                                     EE 53                                      @
@                                                                                @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ FIFO constants

.equ    CLOCKS_FIFO_WAIT,   10      @ Should be at least 50 nanoseconds, then
                                    @  clocks at 200 MHz.

.equ    NUM_AVG_SAMPLES, 1000       @ Warning: Should be less than 2048!!
                                    @  Number of samples to read from FIFO at
                                    @  one time.

.equ    FIFO_HF_PIN,     0b00000000000000000000000100000000 @ Mask for HF pin
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                 ADC.inc                                      @
@                  ARM Processor Initialization Routines                       @
@                                  EE 53                                       @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ Constants for initializing the PIO controller registers
.equ    F512,       0b11000
.equ    F1024,      0b10100
.equ    F2048,      0b10000
.equ    F4096,      0b01100
.equ    F8K,        0b01000
.equ    F16K,       0b00100
.equ    F32K,       0b00000

.equ    F312K,      0b11001
.equ    F625K,      0b10101
.equ    F125M,      0b10001
.equ    F250M,      0b01101
.equ    F5M,        0b01001
.equ    F10M,       0b00101
.equ    F20M,       0b00001

.equ    FMASK,      0b00000
.equ    numFreqs,   12

.equ    ADC_DATA_MASK,  0b00000000000000001111111000000000
.equ    ADC_DATA_SHIFT, 9           @ Shift ADC data to the right by this amount

.equ    ADC_TEST_VAL,    0x3F       @ Debug (and starting) threshold value
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                     ADC                                      @
@                            Analog Data Routines                              @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  Functions for processing data from the ADC.
@
@  Table of Contents:
@   get_adc_data       - Gets the outputs of the FIFO from the PIO A bank.
@   adc_process_fifo   - Processes the first NUM_AVG_SAMPLES from the FIFO
@                           and returns the void rate [duty cycle] of the
@                           square wave.
@   get_data_available - Returns whether the FIFO is half-full or not.
@   update_averages    - Updates the accumulators every time that a sample
@                           is received. If enough of one kind of sample has
@                           been received, the proper average is updated. If
@                           both averages were recently updated, the new
@                           dynamic threshold is calculated and stored.
@   get_threshold      - Returns the value of the threshold.
@   get_averages       - Gets the high and low averages.
@
@  Revision History:
@   06/03/2015  Daniel Andrade  initial revision
@   Early July  Daniel Andrade  first revisions of functions for dealing with
@                                 the ADC and thresholds from user interface
@   Late July   Daniel Andrade  complete rewrite of ADC functions so that
@                                 it works with a single threshold and with
@                                 the eight bit ADC.
@   Early Aug.  Daniel Andrade reworked to work with the FIFO instead of the
@                                 ADC directly.
@   08/15/2015  Daniel Andrade  moved threshold to analog sections instead of
@                                 leaving it in the user interface files.
@   08/20/2015  Daniel Andrade  added functions for getting and setting the
@                                 the threshold value, and finally updated this
@                                 documentation.
@   08/29/2015  Daniel Andrade  updated table of contents


.include "PIO.inc"
.include "adc.inc"
.include "fifo.inc"
.include "general.inc"

.section .text
.align 2
.arm


@ get_adc_data
@
@ Description:        Gets the data from the PIO A bank that corresponds to
@                       the output of the ADC. Gets called by the PCK0
@                       interrupt handler to get the data.
@
@ Operation:          Reads the value of PIO_PDSR, masks it accordingly, and
@                       returns the data in r0.
@
```

```
@ Arguments:          None.
@
@ Return Value:       r0 - Data from ADC.
@
@ Local Variables:    None.
@
@ Shared Variables:   None.
@
@ Global Variables:   None.
@
@ Input:              Checks PIO A bank for the output of the FIFO.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        None.
@
@ Algorithms:         None.
@ Data Structures:    None.
@
@ Registers Changed:  None.
@
@ Revision History:
@    06/03/2015    Daniel Andrade    initial revision
@    07/01/2015    Daniel Andrade    updated to 8 bit ADC
@    08/07/2015    Daniel Andrade    updated for 7 bit resolution

.global get_adc_data
.type get_adc_data STT_FUNC

get_adc_data:
stmdb   sp!, {r1}
ldr     r0,=PIOAAddr        @ Get the address of the PIOA register
add     r0, r0, #PIO_PDSR   @ In particular we want PDSR to get the data pins

ldr     r0, [r0]            @ Load the data
ldr     r1,=ADC_DATA_MASK   @ And apply the mask to get the data only
and     r0, r0, r1
mov     r0, r0, lsr #ADC_DATA_SHIFT  @ Shift it back to one byte

ldmia   sp!, {r1}
bx      lr

@ adc_process_fifo
@
@ Description:        Processes NUM_AVG_SAMPLES number of samples in the FIFO.
@                       This function is called by the main loop once
@                       data_available is set to TRUE.
@
@ Operation:          Reads the samples one at a time by pulsing the read clock.
@                       The samples are compared to a threshold to determine
@                       if they should count as 'high' samples or 'low' samples,
@                       and if they are high, a counter is incremented. Finally
@                       the counter is returned.
@
@ Arguments:          None.
@
@ Return Value:       r0 - The current void fraction.
@
@ Local Variables:    None.
@
@ Shared Variables:   None.
@
```

```
@ Global Variables:   None.
@
@ Input:              None.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        FIFO read speed is limited to 50 MHz, but we won't
@                       be able to reach this speed with this function, since
@                       there's some stuff going on between reads. Need to
@                       use a buffer if that's desired.
@
@ Algorithms:         None.
@ Data Structures:    None.
@
@ Registers Changed:  None.
@
@ Revision History:
@    07/18/2015    Daniel Andrade    initial revision
@    07/24/2015    Daniel Andrade    major revision to move display code
@                                       elsewhere
@    08/01/2015    Daniel Andrade    major revision to use FIFO instead
@    08/03/2015    Daniel Andrade    major revision to work with polling
@                                       functions instead of inside the
@                                       interrupt handler. Rename from
@                                       adc_handler, which is now a simpler
@                                       function.
@
@  NOTE: Assumes that we can read the FIFO faster than data is coming in
@        to it.

.global adc_process_fifo
.type adc_process_fifo STT_FUNC

adc_process_fifo:
stmdb   sp!, {r1-r6, lr}

mov     r2, #0                  @ Initialize the number of high samples and
ldr     r1, =NUM_AVG_SAMPLES    @   loop counter.

ldr     r3, =PIOAAddr
add     r4, r3, #PIO_CODR
add     r3, r3, #PIO_SODR

bl      get_debug_state         @ Get the debug state to determine what we
mov     r6, r0                  @  should be comparing against

apf_get_data:
mov     r0, #FIFORead           @ Clear the FIFO read line to start a read
str     r0, [r4]

bl      get_adc_data            @ Read the incoming ADC data from the FIFO
mov     r5, r0                  @ Store returned data into a temporary variable

cmp     r6, #TRUE               @ If we're in debug, then compare the value
moveq   r0, #ADC_TEST_VAL       @  against a constant.
beq     apf_order_sample

ldr     r0, =threshold          @ Otherwise, get the threshold value and check
ldrb    r0, [r0]                @  to see if the value we got was larger.

apf_order_sample:
cmp     r5, r0
addhs   r2, r2, #1              @ Update the counter if larger.
```

```
bl      update_averages

mov     r0, #FIFORead           @ Set the read line of the FIFO high to end
str     r0, [r3]                @  the read.

subs    r1, r1, #1
bne     apf_get_data

apf_td:
mov     r0, r2                  @ Return the current void fraction

ldmia   sp!, {r1-r6, lr}
bx      lr

@ update_averages
@
@ Description:        Updates the high and low sample accumulators and the
@                       high and low averages. Also updates the threshold if
@                       it's we're capable of doing so.
@
@ Operation:          Adds the sample to the appropriate accumulator (high if
@                       the sample was determined to be high and low otherwise).
@                       Updates the counter on that accumulator. If the counter
@                       happens to be 1024 (power of two so we can shift), then
@                       average the samples by shifting the accumulated value
@                       right by 10 (dividing by 1024). If both counters have
@                       at least 1024 samples, then calculate the threshold
@                       and reset the accumulators and counters.
@
@                     Sorry in advance for the magic numbers, but I couldn't
@                       come up with good constant names that wouldn't sound
@                       just as magic.
@
@ Arguments:          r0 - Current threshold
@                     r5 - Sample value
@
@ Return Value:       None.
@
@ Local Variables:    high_val_accumulator - Accumulator of samples classified
@                       as high. (r/w)
@                     low_val_accumulator  - Accumulator of samples classified
@                       as low. (r/w)
@                     num_high - Counter of samples in high_val_accumulator (r/w)
@                     num_low  - Counter of samples in low_val_accumulator (r/w)
@
@ Shared Variables:   high_average - Current average of latest high samples (r/w)
@                     low_average  - Current average of latest low samples (r/w)
@                     threshold    - System threshold value (w)
@
@ Global Variables:   None.
@
@ Input:              None.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        This function is not C-callable right now because of
@                       the strange registers used for the arguments. It makes
@                       it easier to call from adc_process_fifo, but that
@                       will have to be changed if calling from C is desirable.
@                     Averages and threshold can take a long time to compute.
@                       This is especially true for the threshold, since the
@                       code waits for both accumulators to have sufficient
```

```
@                       values. It might be benefitial to make the 1024 magic
@                       number a variable that the user can specify if this
@                       becomes a problem.
@
@ Algorithms:         Nothing with a concrete name. See Operation for more
@                       details on how things are computed.
@
@ Data Structures:    None.
@
@ Registers Changed:  r0 is trashed, not an issue from where it's called, but
@                       something to keep in mind. Might want to push it.
@
@ Revision History:
@    08/21/2015    Daniel Andrade    initial revision
@    08/22/2015    Daniel Andrade    updated function header with operation,
@                                       limitations, and shared & local variables.

.type update_averages STT_FUNC
update_averages:
stmdb   sp!, {r1-r6}

cmp     r5, r0          @ Figure out which type of point this is and load the
ldrhs   r1, =high_val_accumulator   @ registers accordingly.
ldrlo   r1, =low_val_accumulator
ldrhs   r3, =num_high
ldrlo   r3, =num_low
ldrhs   r6, =high_average
ldrlo   r6, =low_average

ldr     r2, [r1]        @ Update the accumulator with this sample so that
add     r2, r2, r5      @  we can average it later.
str     r2, [r1]

ldr     r4, [r3]        @ Update the number of this type of points we have had
add     r4, r4, #1      @  so far.
str     r4, [r3]

cmp     r4, #1024       @ Doing shift operations here, hard to not have magic
bne     ua_calc_tresh   @  numbers. Essentially we're calculating the average.
mov     r0, #10
mov     r2, r2, lsr r0
strb    r2, [r6]        @ And storing it in memory for future usage.

ua_calc_tresh:
ldr     r1, =num_low
ldr     r2, [r1]
ldr     r3, =num_high
ldr     r4, [r3]

cmp     r2, #1024       @ Only update the threshold if we have sufficient
blo     ua_tear_down    @  samples for a good high average and a good low
cmp     r4, #1024       @  average.
blo     ua_tear_down

ldr     r2, =low_average
ldrb    r2, [r2]
ldr     r4, =high_average
ldrb    r4, [r4]
sub     r0, r4, r2      @ New threshold is half the difference of the high
mov     r6, #1          @  average and the low average subtracted from
mov     r0, r0, lsr r6  @  the high average.
sub     r0, r4, r0

ldr     r2, =threshold  @ Store the new threshold in memory
```

```
strb    r0, [r2]

eor     r0, r0          @ Zero out all the variables, we're ready to start
ldr     r1, =num_high   @  over.
str     r0, [r1]
ldr     r1, =num_low
str     r0, [r1]
ldr     r1, =high_val_accumulator
str     r0, [r1]
ldr     r1, =low_val_accumulator
str     r0, [r1]

ua_tear_down:
ldmia   sp!, {r1-r6}
bx      lr

@ get_data_available
@
@ Description:        Gets whether the FIFO is half full or not.
@
@ Operation:          Checks the HF output on the FIFO.
@
@ Arguments:          None.
@
@ Return Value:       r0 - Whether the FIFO is half full or not.
@
@ Local Variables:    None.
@
@ Shared Variables:   None.
@
@ Global Variables:   None.
@
@ Input:              None.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        None.
@
@ Algorithms:         None.
@ Data Structures:    None.
@
@ Registers Changed:  None.
@
@ Revision History:
@    08/03/2015    Daniel Andrade    initial revision
@    08/05/2015    Daniel Andrade    changed to work completely with polling
@                                       functions

.global get_data_available
.type get_data_available STT_FUNC

get_data_available:
ldr     r0, =PIOAAddr
add     r0, r0, #PIO_PDSR
ldr     r0, [r0]

ands    r0, r0, #FIFO_HF_PIN
moveq   r0, #TRUE
movne   r0, #FALSE

bx      lr
```

```
@ get_threshold
@
@ Description:        Gets the value of the threshold.
@
@ Operation:          Returns the threshold value in memory.
@
@ Arguments:          None.
@
@ Return Value:       r0 - Current threshold value.
@
@ Local Variables:    None.
@
@ Shared Variables:   None.
@
@ Global Variables:   None.
@
@ Input:              None.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        None.
@
@ Algorithms:         None.
@ Data Structures:    None.
@
@ Registers Changed:  None.
@
@ Revision History:
@    08/20/2015     Daniel Andrade      initial revision

.global get_threshold
.type get_threshold STT_FUNC

get_threshold:
ldr     r0, =threshold
ldrb    r0, [r0]

bx      lr

@ get_averages
@
@ Description:        Gets the value of both of the averages.
@
@ Operation:          Returns the average values.
@
@ Arguments:          None.
@
@ Return Value:       r0 - Average low value
@                     r1 - Average high value
@
@ Local Variables:    None.
@
@ Shared Variables:   None.
@
@ Global Variables:   None.
@
@ Input:              None.
@
@ Output:             None.
@
@ Error Handling:     None.
@
@ Limitations:        None.
```

```
@
@ Algorithms:          None.
@ Data Structures:     None.
@
@ Registers Changed:   r0 and r1 (return values)
@
@ Revision History:
@    08/21/2015     Daniel Andrade      initial revision

.global get_averages
.type get_averages STT_FUNC

get_averages:
ldr     r0, =low_average
ldrb    r0, [r0]
ldr     r1, =high_average
ldrb    r1, [r1]

bx      lr

.section .data
.align 2

@ Accumulator for high samples, used to calculate average
high_val_accumulator:
.word   0

@ Accumulator for low samples, used to calculate average
low_val_accumulator:
.word   0

@ Number of samples currently summed in the high_val_accumulator
num_high:
.word   0

@ Number of samples currently summed in the low_val_accumulator
num_low:
.word   0

@ Average value for past 1024 high samples
high_average:
.byte   0

@ Average value for past 1024 low samples
low_average:
.byte   0

@ Samples below this threshold are considered low where as samples
@  above this threshold are considered high (no hysteresis)
threshold:
.byte   ADC_TEST_VAL
```

E USB Setup and Code

E.1 USB Overview and Basic Setup

Disclaimer Please note that this code is nonfunctional right now, which is why it is at the end in the appendices section. Nevertheless, it can probably serve as a good starting point for a USB codebase, provided that it's not wildly off from what it should be.

Sources The website/eBook found at

http://www.beyondlogic.org/usbnutshell/usb1.shtml (*USB in a Nutshell*)

has been particularly helpful. Other sources were referenced, but, since they were all consistent with the above and tended to less helpful (in that details were often skipped), there's probably little value in including them here. Since the code does not yet work, please reference the above source (or any other source) in addition to this document when implementing the USB interface to make sure there are no errors in the design.

USB Overview USB is quite complicated, so an in-depth overview is necessarily long. This is because USB is extremely generalizable and thus there is a lot of configuration that goes on behind the scenes.

USB Hardware (See Chapter 2 of *USB in a Nutshell*) The hardware setup is very simple, as it only involves the connection itself and a 1.5kΩ pull-up resistor to indicate that the device is a full-speed (12 Mbps) device, which corresponds to the fastest configuration of the AT91RM9200 UDP port. A slightly more complicated hardware setup that informs the processor of suspends can be done as well; this solution is presented the AT91RM9200 datasheet, and shown below for reference. Keep in mind that the current device does not use this configuration, but rather the simpler one that involves only the pull-up resistor.

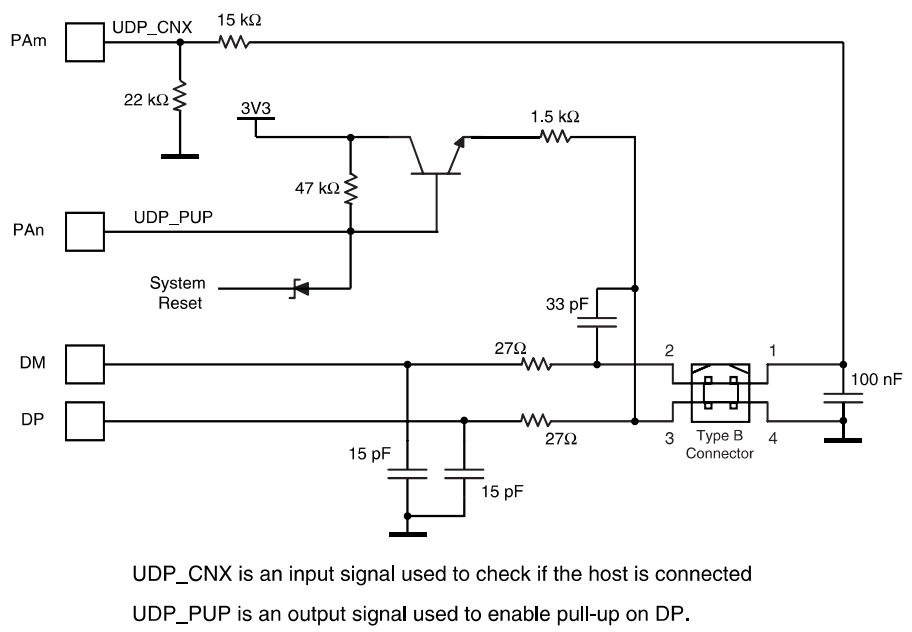


Figure 24: Suggested USB hardware configuration

USB Structure Before going into the details of the software implementation, it is necessary to understand how USB configuration is structured in the first place. At the top level, a USB interaction takes place between a host (in this case the computer commanding the device) and a number (maximum 128) of devices; Bubbly would be one of these. The host communicates to the device via an interface through an endpoint. A device is capable of having many configurations, each one of which can have several interfaces, which in turn can have many endpoints, thus making a "tree-like," recursive structure. This ensures that the USB interface is extremely general because almost all kinds of devices can be placed into such a structure.

Bubbly itself does not need to have a complicated configuration - there only needs to be one device configuration and one interface, since Bubbly only has one possible state it can be in (it only functions as a data acquisition device). The configuration is stored in the device, interface, and endpoint descriptors.

Endpoint Types (See Chapter 4 of *USB in a Nutshell*) Endpoint zero always serves as the CONTROL endpoint. This endpoint receives configuration data from and sends data to the host. This endpoint is used by the host to determine what kind of device it is talking to (receiving the configuration) as well as for setting and getting device states (device address, data directions, setting different device configurations, and so forth). The other endpoints are optional and can be either BULK IN, BULK OUT, INTERRUPT IN, INTERRUPT OUT, ISOCHRONOUS IN, or ISOCHRONOUS OUT endpoints. BULK is used for sending general data (with error-checking, handshaking, and insurance that the data arrives in the proper order), INTERRUPT is used for calling attention to the device (just as interrupts in microcontrollers work), and ISOCHRONOUS is used for sending fast data where the insurance of proper order is not so

important (e.g. video frames). Bubbly needs a BULK IN endpoint for sending data to the host (the "IN" is with respect to the host) and possibly a DATA OUT endpoint if it expects to receive commands from some kind of computer-based UI running on the host (this is completely unimplemented). Right now the code (at least tries to) implement a CONTROL endpoint on endpoint zero and a DATA IN endpoint on endpoint one.

Transactions (See Chapters 4 and 6 of *USB in a Nutshell*) Transactions are the interactions that happen between the host and device(s) over a USB hub. There are four different kinds of stages of transactions that can happen, but two of these, SETUP and STATUS, are only possible on CONTROL endpoints. The four stages are SETUP, STATUS, DATA IN, and DATA OUT.

CONTROL endpoints follow the convention of having a starting SETUP stage, possibly followed by DATA IN/OUT stages, and finally ending with a STATUS stage. In a SETUP stage, the host sends a request to the device. The type of request and other information is encoded in the first 8 bytes, which is called the setup packet. The setup packet is quite intricate and holds a lot of information; it is critical that it be parsed correctly for the device to work properly. Reference Chapter 6 of *USB in a Nutshell* for details on the structure of the setup packet. **Note that there are maximum timing restraints on the SETUP transaction**, so it's not always possible to step through the code if there is a problem somewhere.

After the SETUP packet is sent/processed, there can be further DATA IN or DATA OUT transactions which hold relevant information pertaining to the request (either the device responds with relevant data or the host sends additional data). Finally, there is either a STATUS IN or STATUS OUT transaction (depending on the direction of data flow that happens in the DATA stage), which consists of sending or receiving an empty message acknowledging the end of the entire control cycle.

The other endpoints are not very remarkable, in that they behave completely as expected. A BULK IN endpoint, for instance, is well-suited to having chains of DATA IN transactions for sending data to the host. Note that all of this must be done *after* the device is configured.

AT91RM9200 UDP Interface In addition to understanding the general USB structure, it is necessary to have working knowledge of how this particular microprocessor handles USB transactions before the code can be examined. The AT91RM9200 UDP interface has six configurable endpoints (including endpoint zero, the control endpoint). The types of these endpoints can be specified by writing to the correct bits in the control register corresponding to that endpoint. In this register there are also flags which indicate when each kind of transaction is received, and the UDP interface will put the relevant data in a piece of FIFO memory, which can be accessed by sequentially reading a register.

The UDP interface is responsible for handling the PID and ACK messages between the host and device. The code, in turn, is responsible for determining the content of the messages and responding to the host in an appropriate and timely fashion. The UDP interface allows the code to write bytes to a named FIFO register corresponding to an endpoint; these bytes are sent when flags are set in the control register.

Each of the six endpoints on the AT91RM9200 can be configured to interrupt the processor when it receives a transaction. The interrupt handler must then look at the control register to determine what kind of transaction stage it should handle. This process can also be achieved through polling the control registers directly, but keep in mind that it would be harder to guarantee the timing constraints if the code is rewritten in this way (although the timing restraints are quite lenient (on the order of tens of milliseconds), so it would probably not be a problem).

The UDP interface functions as a state machine and has several states it can be in, depending on messages and data it has received on the CONTROL endpoint from the host. The device will transition from state to state when different bits are set in the global state register of the UDP interface, assuming certain prerequisites have been met (e.g., need to have received an address before being in the address state. A diagram of the state machine, taken directly from the AT91RM9200 datasheet, is reproduced below.

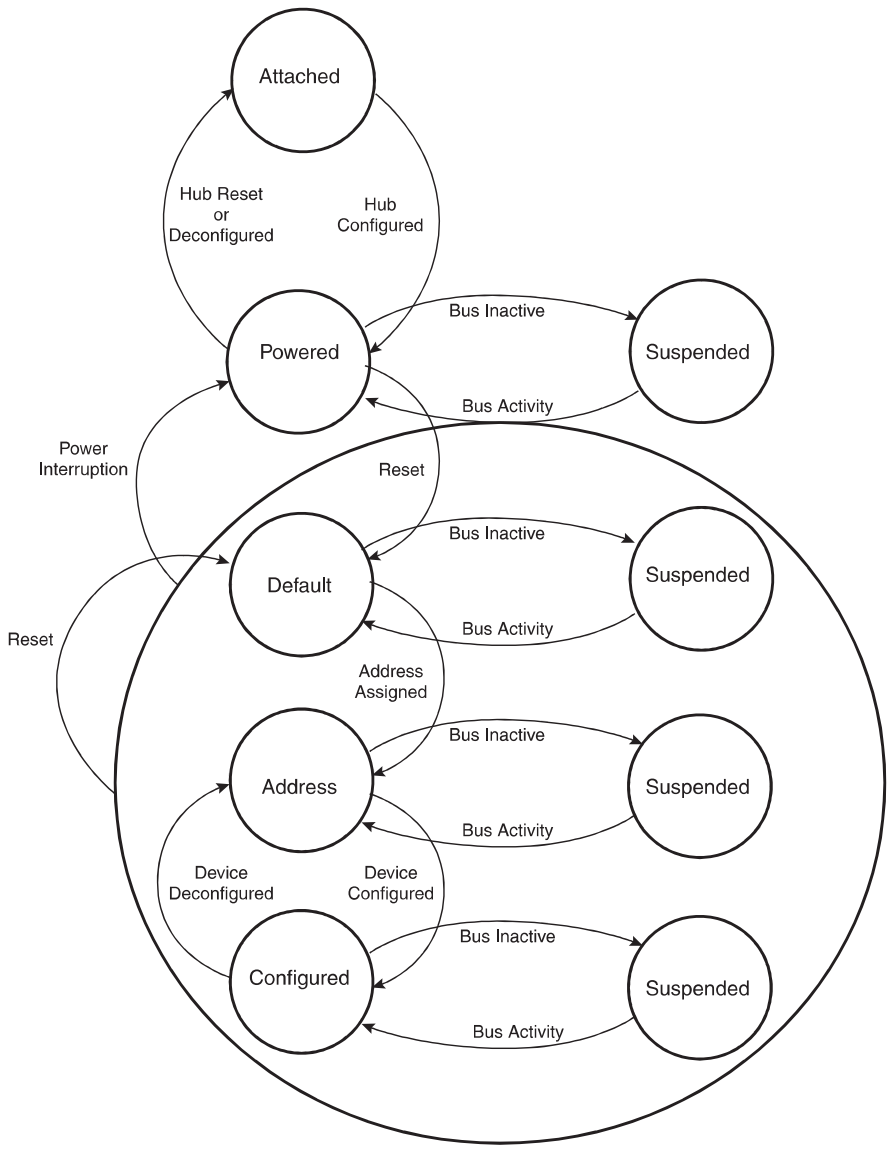


Figure 25: UDP interface state machine

The following lists detail what the UDP interface does and what the code should handle when each different stage of a USB transaction takes place. This is also detailed in the AT91RM9200 datasheet.

Setup Stage

1. The UDP interface will acknowledge the packet automatically and set the RXSETUP flag in the control register for that endpoint. The device will interrupt at this point if that feature is enabled.

2. The setup packet must be read from the FIFO pertaining to that endpoint, and processed appropriately. If data needs to be sent to the host, then it should be loaded into the FIFO.

3. RXSETUP must be cleared to acknowledge.

Data In Stage

1. Before loading data into the FIFO, the code should poll the TXPKTRDY flag. Data should only be sent if that flag is clear.

2. After this, data can be loaded into the FIFO by writing to the FIFO register one byte at a time.

3. The TXPKTRDY flag should be set to indicate that the data is loaded.

4. The host will acknowledge by setting the TXCOMP flag, which will interrupt the processor if that feature is set.

Data Out Stage

1. The RX_DATA_BK0 flag in that endpoint's control register is set to indicate that there is received data in the FIFO. This will interrupt the processor if that feature is set.

2. The processor should read the field RXBYTECNT to determine how many bytes were received.

3. The processor should read the data in the FIFO by sequentially reading bytes from the FIFO register pertaining to that endpoint.

4. Finally the code should clear the RX_DATA_BK0 flag.

Status Stages STATUS IN and STATUS OUT stages should be treated as empty DATA IN and DATA OUT stages. So a STATUS IN would be the same as a DATA IN but with no data written to the FIFO, and a STATUS OUT would be the same as a DATA OUT but with the value of RXBYTECNT would read as zero.

Please note that the UDP interface needs a working 48 MHz clock. This is not a problem since this clock was achieved through a PLL, and there is confirmation that it works because it is also needed for the EEPROM TWI interface, which has been tested.

Please reference the AT91RM9200 data sheet for more information. Pages 547 to 565 fully describe the UDP interface. Shown below is a quick reference guide which was drawn up while writing the code. All the information there should be present above, but it might be helpful to have it in a more concise form during development.

Listing 5: UDP Interface Quick Reference

```
AT91RM9200 USB Notes v1.0
Daniel Seabra de Andrade

Initialization:
    - Clear the TXVDIS bit in UPD_TXVC
    - Need to set up the following:
        - One control endpoint for talking to host (should be endpoint 0)
        - One DATA IN endpoint for sending data to host


Setup transaction:      (CANNOT use ping-pong endpoint)
    - USB device acknowledges packet automatically
    - RXSETUP set in UDP_CSRx register
    - Interrupt generated on endpoint until RXSETUP is cleared (if interrupts
      enabled on this endpoint)
    - Inside the interrupt handler, we have to do the following things:
        - Read the setup packet in FIFO
        - Based on this data, set the DIR flag appropriately
        - Clear RXSETUP (must do this AFTER reading FIFO and AFTER dir flag)

DATA IN transaction:    (without ping-pong endpoints)
    - Poll TXPKTRDY flag in UDP_CSRx register (should be cleared).
    - Write data in the endpoint's FIFO (UDP_FDRx)
    - Set the TXPKTRDY flag to indicate data has been stored and should be sent
    - TXCOMP is set when data is acknowledged. Interrupt pending until it
      is cleared.

DATA OUT transaction:   (without ping-pong endpoints)
    - RX_DATA_BK0 is set when data is written to FIFO, can interrupt on it.
    - RXBYTECNT in UDP_CSRx gives number of bytes available
    - Microcontroller should read UDP_FDRx to get this data and transfer it to
       physical memory somewhere (buffer?)
    - Clear RX_DATA_BK0 to indicate that data has been transferred

Status transactions:     (CANNOT use ping-pong endpoint)
    - IN:
        - This is basically a DATA IN transaction but without any data. So
          just do everything like it's done there, but without writing any
          data to the FIFO.
        - Steps:
            - Wait for TXPKTRDY flag to be cleared
                "At this step, TXPKTRDY must be cleared because the previous
                transaction was a setup transaction or a Data OUT transaction"
            - Set TXPKTRDY flag without writing to FIFO
            - TXPKTRDY (typo, should be TXCOMP?) flag set by UDP device to
                to acknowledge
    - OUT:
        - Zero-length DATA OUT instruction
        - RX_DATA_BK0 is set, so it'll interrupt
        - But in this case RXBYTECNT will be zero, since it's zero-length
        - Microcontroller should clear RX_DATA_BK0 to acknowledge
```

E.2 Detailed USB Code Description

The USB code can be found in the file `usb.s`, with constants in the file `usb.inc`.

E.2.1 Constants

There are many masks used throughout the code to determine what exactly is going on. This is especially true for specific bits of the control registers which need to be checked (e.g. the flags TXCOMP, RX_SETUP_BK0, RXSETUP, etc.). These constants are fairly self-explanatory and the comments should help, so further discussion on these types of constants will be omitted.

Setup packet constants There are also constants for processing the setup packets. They are presented in the table below.

Constant	Description
`SETUP_PACKET_SIZE`	Size of the setup packet
`DEV_bReq_MAX_VAL`	Maximum bRequest value for device requests
`INT_bReq_MAX_VAL`	Maximum bRequest value for interface requests
`ENDP_bReq_MAX_VAL`	Maximum bRequest value for endpoint requests
`XXXX_GET_STATUS_...`	Responses to `GET_STATUS` requests from the host for device, interface, and endpoint requests

Table 17: USB Setup Packet constants

Descriptor Constants There are four different types of descriptor constants in the code: device, interface, endpoint, and configuration descriptor. The device descriptor describes the purpose of the device as a whole, where as the interface and endpoint descriptors talk about their relevant portions as described in the *USB Overview and Basic Setup* section. The configuration descriptor is meant to be used if the device has several different configurations and also describes meta information, that is, number of interfaces and endpoints available on the device, as well as the power configuration for the device.

Note that many of the choices I made here for the variables are questionable (this is especially true if there is a *Who knows?* comment on that same line). I was not able to find too many good resources explaining what some of these fields mean. My guess is that the ones that are hard to explain are probably not too important, but take special note of this in case my assumption proves to be problematic.

Device Descriptor The device is configured to use USB v1.1, same as the AT91RM9200 processor. It has a vendor-specified class code, because that sounded the most general to me, and the product id is 53 for EE53. It is also configured to have no string descriptors. The maximum packet size is limited to 32 bytes (it's possible to choose 8, 16, 32, or 64 byte packets). The fields

subclass, protocol number, and vendor id are set to zero because I had no idea what else to set them to.

Configuration Descriptor The configuration descriptor is set up to have one interface and specifies that the device should be powered by the host (bus-powered device). It also asks for maximum power from the host (500 mA).

Interface Descriptor The interface descriptor specifies that there will be two endpoints and again has a vendor-specified class code, because I thought that sounded the most general. The subclass and protocol number were set to zero because I did not know what else choose for those values.

Endpoint Descriptor There is no need to set up the control endpoint, so the endpoint descriptor in the code is for the BULK IN endpoint. It specifies that there is a maximum packet size of 32 bytes and the direction (in to the host).

E.2.2 Variables

In addition to two bookkeeping status variables (`address_set` and `did_setup`), variables are used for storing jump tables and the arrays for the device, configuration, interface, and endpoint descriptors defined in the *Constants* section above. This last category is straightforward enough that it should need no further explanation - the descriptors are stored into an array for convenience, so that they can be sequentially loaded into the FIFO for output to the host. Note that the configuration descriptor is merely the other descriptors concatenated together.

Bookkeeping Variables The first of these, `address_set`, is a flag variable that is set to `TRUE` when the host first gives the host its address. It is necessary because the device can't actually set the UDP device in addressed state (by setting a bit in the global status register) until the complete cycle is done. This means that although the address is received in the setup packet, it can only be set in the STATUS message that acknowledges it. This variable is set in the SETUP stage and checked in every STATUS message.

The second, `did_setup`, is a flag variable that indicates whether the device is in a SETUP cycle. I do not remember the exact purpose of it right now, it seems a bit unnecessary.

Jump Tables The use of the jump tables is described in the subsection *Code Overview* below. The following table describes the jump tables.

Constant	Description
`setup_dev_jump_table`	Jump table for processing device requests
`setup_int_jump_table`	Jump table for processing interface requests
`setup_endp_jump_table`	Jump table for processing endpoint requests

Table 18: USB Jump Tables

E.2.3 Code Overview

The code consists of a initialization function, an interrupt handler, helper functions which handle each kind transaction stage, and a lot of data space for the device, interface, and endpoint descriptors, and very large (and admittedly quite ugly) jump tables.

The function `usb_setup` is responsible for initializing the UDP interface, enabling the interrupts, and resetting the USB FIFO memories.

The function `usb_interrupt_handler` is the interrupt handler for the UDP interface. It determines which endpoint is interrupting, figures out which kind of transaction stage is causing the interrupt, and calls the appropriate helper function for dealing with this transaction stage. Finally the function clears the interrupt. Note that there is no function for STATUS IN or STATUS OUT transactions, because the UDP interface does not distinguish these from empty DATA IN or DATA OUT transactions.

The below helper functions all take two arguments. The first of these, `csr_reg_val`, is the value of the control register at the moment the interrupt happened. The second, `endp_num`, is the endpoint number that interrupted.

The function `do_setup_transaction(csr_reg_val, endp_num)` is (possibly) the worst piece of code I have ever written and also the only reason I regret not writing the USB code in C instead of assembly, as it would look a lot better with `switch` statements rather than jump tables everywhere. This function is responsible for dealing with SETUP transactions (and in particular SETUP packets) from the host in a CONTROL endpoint. The code does the following:

1. Reads the setup packet into several registers by repeatedly reading from the FIFO register in the UDP interface. Some concatenation is done to turn 2 byte values into single word values.

2. The code branches out based on the value of `bRequest`, one of the bytes of the setup packet which specifies the request type. The code separates into different device, interface, or endpoint requests based on this value.

3. The code branches out yet again based on the value of `bRequest`, this time to determine what kind of request it should handle. This is done via three different jump tables, one each for device, interface, or endpoint. This is done to avoid having a lot of blank spaces in one large `bRequest` jump table.

4. Finally, the code deals with the myriad things it must do in each different kind of situation. This all can be found in Chapter 6 of *USB in a Nutshell*, and can involve reading data, setting the address of the device, setting the direction of an endpoint, and sending data (such as descriptors) to the host. Some features were left unimplemented because I believed them to be unnecessary but I'm not very sure about that, so I marked these spots with "`TODO`"s in the code. There is also some structure for error handling, but it is currently empty, so that needs to be filled in.

The function `do_data_out_transaction(csr_reg_val, endp_num)` is responsible for dealing with DATA OUT and STATUS OUT transactions. The code right now does not expect any data from the host other that the setup packet, so the only thing this function does is clears the flag so that the UDP interface is no longer interrupting. This should take care of the STATUS OUT transaction automatically, but this is untested.

The function `do_data_in_transaction(csr_reg_val, endp_num)` is responsible for dealing with DATA IN and STATUS IN transactions. This function is called on an acknowledge from the host that it received the data the device has sent, so there's nothing much to do other than clear the flag so that the UDP interface will no longer interrupt.

E.3 Full USB Code

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                               @
@                                    USB.inc                                    @
@                                 USB Constants                                 @
@                                     EE 53                                     @
@                                                                               @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@ USB Addresses
.equ    UDPBaseAddress,     0xFFFB0000
.equ    numUSBInitRegs,     7             @ Number of register to initialize

@ Offsets
@ NOTE: There are 6 endpoints total, right now I'm just using 2 of them.
.equ    UDP_FRM_NUM,        0x00
.equ    UDP_GLB_STAT,       0x04
.equ    UDP_FADDR,          0x08
.equ    UDP_IER,            0x10
.equ    UDP_IDR,            0x14
.equ    UDP_IMR,            0x18
.equ    UDP_ISR,            0x1C
.equ    UDP_ICR,            0x20
.equ    UDP_REST_EP,        0x28
.equ    UDP_CSR0,           0x30
.equ    UDP_CSR1,           0x34
.equ    UDP_FDR0,           0x50
.equ    UDP_FDR1,           0x54
.equ    UDP_TXVC,           0x74

@ Clears TXVDIS bit to enable UDP
.equ    TXVCVal,        0x0

@ Enable interrupts for endpoints 0 and 1
.equ    UDPIERVal,      0b00000000000000000000000000000011

@ Mask for clearing the UDP interrupt in the AIC
.equ    CLEAR_UDP_INT,  0b00000000000000000000100000000000

@ The following are used to detect which endpoint is interrupting
.equ    ENDP0_INT,      0b00000000000000000000000000000001
.equ    ENDP1_INT,      0b00000000000000000000000000000010

@ Disable all other interrupts (will ignore the fancy features of the
@  UDP controller like Wakeup, Start of Frame, etc. for now, as well
@  as the other endpoints)
.equ    UDPIDRVal,      0b00000000000000000010111100111100

@ The following are used to reset the FIFOs on startup to ensure everything
@  is clean at the beginning
.equ    REST_EP_ON,     0b00000000000000000000000000011111
.equ    REST_EP_OFF,    0b00000000000000000000000000000000

@ Enable endpoint zero and identify it as a control endpoint
.equ    UDPCSR0InitVal, 0b00000000000000001000000000000000

@ Enable endpoint one and identify it as a BULK IN endpoint (this one will
@   be used to send data to the host)
.equ    UDPCSR1InitVal, 0b00000000000000001000011000000000

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Setup Packet Constants (used for configuration)
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Offsets for each field
.equ    bmReqTypeOffset,    0
.equ    bRequestOffset,     1
```

```
.equ    wValueOffset,       2
.equ    wIndexOffset,       4
.equ    wLengthOffset,      6

@ What kind of information is the packet requesting/sending? (in bmReqType)
.equ    bmReqDevice,        0b00
.equ    bmReqInterface,     0b01
.equ    bmReqEndpoint,      0b10
.equ    bmReqMask,          0b00000011

@ What is the packet asking for or giving us?
@  (constants for setup packets only)
.equ    bReq_GET_STATUS,    0x0     @ Get status of device, interface, or endpoint
.equ    bReq_CLEAR_FEATURE, 0x1     @ Clears a feature
.equ    bReq_SET_FEATURE,   0x3     @ Sets a feature
.equ    bReq_SET_ADDRESS,   0x5     @ Sets the address of the device
.equ    bReq_GET_DESCR,     0x6     @ Gets the descriptor of the device
.equ    bReq_SET_DESCR,     0x7     @ Sets the descriptor of the device
.equ    bReq_GET_CONF,      0x8     @ Gets configuration of device
.equ    bReq_SET_CONF,      0x9     @ Sets configuration of device
.equ    bReq_GET_INTERF,    0xA     @ Gets interface data
.equ    bReq_SET_INTERF,    0x11    @ Sets interface data
.equ    bReq_SYNCH_FRAME,   0x12    @ To report endpoint synchronization frames
                                    @   (for isynchronous endpoints)

@ Masks
.equ    dirBitMask,         0b10000000  @ Mask for the direction bit
.equ    RXSETUP_CLR_MASK,   0x1111111B  @ Mask for clearing RXSETUP
.equ    DES_TYPE_MASK,      0x000000F0  @ Gets top bit of wValue for getting
                                        @  the descriptor type in a GET_DESCR
                                        @  request
.equ    DIR_BIT_MASK_CSR,   0xFFFFFF7F  @ To clear the dir bit in CSR
.equ    DIR_BIT_MASK_CSR2,  0x00000080  @ To get value of direction flag
.equ    BKO_BIT_MASK_CSR,   0xFFFFFFFD  @ To clear the RX_DATA_BK0 bit in CSR

.equ    BK_DATA_MSK,        0x2         @ For getting value of RX_DATA_BK0
.equ    RXSETUP_MSK,        0x4         @ For getting value of RXSETUP
.equ    TXCOMP_MSK,         0x1         @ Mask for getting value/clearing TXCOMP
.equ    TXPKTRDY_MSK,       0x10        @ Mask for getting value of TXPKTRDY

.equ    UDP_CONFG_MASK,     0x2         @ For getting whether device is configured
.equ    UDP_FADDEN_MASK,    0x1         @ For getting whether device has address

.equ    UDP_DIR_IN,         0x0         @ For setting the direction bit in
.equ    UDP_DIR_OUT,        0x80        @   CSR correctly


@ Size of setup packets (in bytes)
.equ    SETUP_PACKET_SIZE,      32

@ Maximum values for bRequest for each type of request
.equ    DEV_bReq_MAX_VAL,       0x9
.equ    INT_bReq_MAX_VAL,       0x11
.equ    ENDP_bReq_MAX_VAL,      0x12

@ Bytes for responding to setup requests
.equ    DEV_GET_STATUS_BOT_BYTE,    0   @ Bus powered with no remote wakeup
.equ    DEV_GET_STATUS_TOP_BYTE,    0   @ Reserved.

.equ    INT_GET_STATUS_BOT_BYTE,    0   @ Must respond with two zeros. Pretty
.equ    INT_GET_STATUS_TOP_BYTE,    0   @  lame, if you ask me.

.equ    ENDP_GET_STATUS_BOT_BYTE,   0   @ No endpoints are halted. Ever.
.equ    ENDP_GET_STATUS_TOP_BYTE,   0   @ Reserved.
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Device Descriptor
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.equ    dev_bLength,        18          @ Size of descriptor in bytes
.equ    dev_desType,        0x1         @ Device descriptor corresponds to 1
.equ    dev_bcdUSB1,        0x10        @ Corresponds to the USB version 1.1
.equ    dev_bcdUSB2,        0x01
.equ    dev_bDevClass,      0xFF        @ Vendor specified class code
.equ    dev_bDevSubClass,   0x0         @ Subclass (who knows?)
.equ    dev_bDevProt,       0x0         @ Protocol number (who knows?)
.equ    dev_bMaxPackSize,   32          @ Packets limited to 32 bytes
.equ    dev_idVendor1,      0x0         @ Vendor id (who knows?)
.equ    dev_idVendor2,      0x0
.equ    dev_idProduct,      0x53        @ Product id (53 for EE 53)
.equ    dev_idProduct2,     0x00
.equ    dev_bcdDevice1,     0x00        @ Version 0 (release 0)
.equ    dev_bcdDevice2,     0x00
.equ    dev_iMan,           0x0         @ No string descriptions for the entire
.equ    dev_iProduct,       0x0         @   project. No one has time for that.
.equ    dev_iSerial,        0x0
.equ    dev_bNumConf,       1           @ Device has only one configuration.


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Configuration Descriptor
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.equ    conf_bLength,       9           @ Size of descriptor in bytes
.equ    conf_desType,       0x2         @ This is a configuraiton descriptor

@ Total length of configuration, includes all descriptors
.equ    conf_wTotLen1,      conf_bLength + int_bLength + endp_bLength
.equ    conf_wTotLen2,      0
.equ    conf_bNumInts,      1           @ Total number of interfaces
.equ    conf_bConfVal,      0           @ This is configuration zero (the only one)
.equ    conf_iConf,         0x0         @ No string descriptor
.equ    conf_bmAttr,        0b10000000  @ Not self powered, no remote wakeup
.equ    conf_bMaxPower,     0xFF        @ Maximum power request (500 mA)


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Interface Descriptor
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.equ    int_bLength,        9           @ Size of descriptor in bytes
.equ    int_desType,        0x4         @ Interface descriptor corresponds to 4
.equ    int_bIntNum,        0           @ Only one interface, so index of zero
.equ    int_bAltSetting,    0           @ Interface is not alternative
.equ    int_bNumEndpoints,  1           @ One endpoint (other than control)
.equ    int_bIntClass,      0xFF        @ Class code (vendor specific)
.equ    int_bIntSubclass,   0x0         @ Subclass code (who knows?)
.equ    int_bIntProt,       0x0         @ Protocol number (who knows?)
.equ    int_iInterface,     0x0         @ No string descriptor

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Endpoint Descriptor
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.equ    endp_bLength,       7           @ Size of descriptor in bytes
.equ    endp_desType,       0x5         @ Endpoint descriptor corresponds to 5
.equ    endp_addr,          0b10000001  @ Endpoint one with data going IN to host
.equ    endp_bmAttr,        0b00000010  @ Bulk endpoint. Other bits reserved.
.equ    endp_maxPackSize1,  32          @ Supports packets of 32 bytes
.equ    endp_maxPackSize2,  0
.equ    endp_bInterval,     0           @ Ignored for bulk endpoints

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Other constants
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.equ    CONTROL_BUFFER_SIZE,    32      @ Size of control buffer (in bytes)
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@                                                                              @
@                                     USB                                      @
@                                 USB Routines                                 @
@                                    EE 53                                     @
@                                                                              @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


@  Daniel Andrade
@  EE 53
@  TA: Andy Zhou
@
@  File Description:  USB functions (initialization, interrupt handler, helper
@                       functions, and relevant data).
@
@  Table of Contents: TODO
@
@  Revision History:
@   6/15/2015   Daniel Andrade  Initial revision


.section .text
.align 2
.arm

.include "usb.inc"
.include "general.inc"
.include "at91rm9200.inc"


@ usb_setup
@
@  Description:        Sets up the ARM registers for communicating to the host
@                       via the UDP interface.
@
@  Operation:          TODO
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   None.
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/15/2015    Daniel Andrade    initial revision
@
.global usb_setup
```

```
.type usb_setup STT_FUNC

usb_setup:
stmdb   sp!, {r0-r5}
ldr     r0,=UDPBaseAddress      @ Load the base address of the UDP controller

usbs_loop_init:
mov     r3, #0                  @ Initialize the loop counter and the pointers
ldr     r1,=UDPInitOffsetArray  @  to the beginning of the arrays
ldr     r2,=UDPInitVals

usbs_loop:
ldrb    r4, [r1, r3]            @ Load address offset and add it to the
add     r4, r0, r4              @  base address
ldr     r5, [r2, r3, lsl #2]    @ Load the value to store into the register
str     r5, [r4]                @  and store it

add     r3, r3, #1
cmp     r3, #numUSBInitRegs     @ End of loop check
blo     usbs_loop

usbs_tear_down:
ldmia   sp!, {r0-r5}
bx      lr


@  usb_interrupt_handler
@
@  Description:         Interrupt handler for USB interrupts.
@
@  Operation:           Looks at UDP_ISR to determine which endpoint is
@                        causing the interrupt. Then reads the value
@                        of the appropriate control register (CSR) to determine
@                        what kind of transaction is causing the interrupt.
@                        The appropriate function is called to handle the
@                        transaction.
@
@  Arguments:           None.
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      06/15/2015    Daniel Andrade    initial revision
@      08/08/2015    Daniel Andrade    updated constants

.global usb_interrupt_handler
```

```
.type usb_interrupt_handler STT_FUNC

usb_interrupt_handler:
sub     lr, lr, #4          @ construct the return address
stmfd   sp!, {lr}           @ and push the adjusted lr_IRQ

stmdb   sp!, {r0-r2, lr}

usbih_get_source:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_ISR    @ Determine which endpoint is causing the interrupt
ldr     r0, [r0]

ands    r0, r0, #ENDP0_INT  @ Go to the endpoint zero label if that's the endpoint
bne     usbih_endp0int  @  that's causing us to interrupt (this one has higher
                        @  priority)
ands    r0, r0, #ENDP1_INT  @ Otherwise, if endpoint one is interrupting, go to
bne     usbih_endp1int  @  that label
beq     usbih_tear_down @ If these two didn't interrupt, something strange is
                        @  happening. Ignore it for now, but probably shouldn't.
                        @  TODO: This.

usbih_endp0int:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0   @ Load the CSR0 register to get the type of transaction
ldr     r0, [r0]
mov     r1, #0          @ Local variable to indicate it's an endpoint 0 interrupt
b       usbih_get_transaction

usbih_endp1int:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR1
ldr     r0, [r0]
mov     r1, #1
@b       usbih_get_transaction

usbih_get_transaction:
ands    r2, r0, #RXSETUP_MSK    @ Call the right function for each bit that
blne    do_setup_transaction    @  could have caused us to interrupt
ands    r2, r0, #BK_DATA_MSK
blne    do_data_out_transaction @ NOTE: It is possible that more than one
ands    r2, r0, #TXCOMP_MSK     @  of these functions gets called. I think.
blne    do_data_in_transaction

usbih_tear_down:
ldr     r0,=AIC_ICCR        @ Clear the interrupt in the interrupt controller
ldr     r1,=CLEAR_UDP_INT
str     r1, [r0]

ldr     r0,=AIC_EOICR       @ This is the end of an interrupt, so write to EOI
mov     r1, #CLEAR_UDP_INT  @ Any value will do, just placeholder
str     r1, [r0]            @ Write to the EOI register to indicate we're done

ldmia   sp!, {r0-r2, lr}
ldmfd   sp!, {pc}^


@  do_setup_transaction
@
@  Description:         Processes a setup transaction (which starts a control
@                        cycle)
@
@  Operation:           Reads the setup packet (must be 8 bytes) in the FIFO
@                        and, based on what was sent, branches off to several
@                        different labels to process the request.
```

```
@
@  Arguments:          None.
@
@  Return Value:       None.
@
@  Local Variables:    None.
@
@  Shared Variables:   did_setup   - Indicates that we're in a setup cycle (w)
@                      address_set - Flag indicating whether address was set
@                                    in setup (w)
@
@  Global Variables:   None.
@
@  Input:              None.
@
@  Output:             None.
@
@  Error Handling:     None.
@
@  Limitations:        None.
@
@  Algorithms:         None.
@  Data Structures:    None.
@
@  Registers Changed:  None.
@
@  Revision History:
@      06/15/2015    Daniel Andrade    initial revision
@      06/16/2015    Daniel Andrade    continued working...
@      06/18/2015    Daniel Andrade    finished up the code, not debugged yet.
@                                        the descriptors are a pain, not entirely
@                                        sure how most of this works...
@      07/20/2015    Daniel Andrade    updated code to handle the queues better
@      08/08/2015    Daniel Andrade    another queue update. If I understand
@                                        this correctly, we can probably get
@                                        away just with using the built-in queues
@                                        and not have to implement our own data
@                                        structure
@     08/09/2015     Daniel Andrade    forgot how my addressing system works
@                                        so I had to go back to update it all

.type do_setup_transaction STT_FUNC

do_setup_transaction:
@@@@ TODO: Ignore request if already in setup transaction @@@@@@@@@@@@@@@@@@@@@@@@
stmdb   sp!, {r0-r6, lr}
ldr     r0, =did_setup      @ Indicate that we're on a setup cycle
mov     r1, #TRUE
strb    r1, [r0]

dst_get_packet:             @ Setup transaction must come with an 8 byte packet
@ TODO: Check packet size for errors? @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
ldr     r5, =UDPBaseAddress
add     r5, r5, #UDP_FDR0
ldr     r0, [r5]            @ r0 now contains the bmRequestType
ldr     r1, [r5]            @ r1 now contains bRequest

dst_get_dir:
ands    r2, r0, #dirBitMask @ Direction is top bit of bmRequestType
movne   r3, #UDP_DIR_OUT    @ If high, then this is going out of device;
moveq   r3, #UDP_DIR_IN     @   otherwise, this is going into device.
ldr     r4, =UDPBaseAddress
add     r4, r4, #UDP_CSR0   @ We want to store the direction flag in the
ldr     r5, [r4]            @   CSR register to tell it which direction it'll have
ldr     r6, =DIR_BIT_MASK_CSR
```

```
and     r5, r5, r6          @ Clear the dir flag and then set it appropriately
orr     r5, r5, r3
str     r5, [r4]

dst_get_wValue:
ldr     r2, [r5]            @ r2 contains first byte of wValue
ldr     r3, [r5]            @ load the second byte of wValue
mov     r3, r3, lsl #8      @ Shift it by the size of a byte to OR into r2
orr     r2, r2, r3          @ Now full value of wValue is in r2

dst_get_wIndex:
ldr     r3, [r5]            @ r3 contains first byte of wIndex
ldr     r4, [r5]            @ load the second byte of wIndex
mov     r4, r4, lsl #8      @ Shift it by the size of a byte to OR into r3
orr     r3, r3, r4          @ Now full value of wIndex is in r3

dst_get_wLength:
ldr     r4, [r5]            @ r4 contains first byte of wLength
ldr     r5, [r5]            @ load the second byte of wLength
mov     r5, r5, lsl #8      @ Shift it by the size of a byte to OR into r4
orr     r4, r4, r5          @ Now full value of wLength is in r4


dst_jump_type:
and     r0, r0, #bmReqMask  @ Let's branch out depending on bRequest
cmp     r0, #bmReqDevice
beq     dst_do_req_device
cmp     r0, #bmReqInterface
beq     dst_do_req_interface

dst_do_req_endpoint:
cmp     r1, #ENDP_bReq_MAX_VAL  @ If larger than this value, then there must
bhi     dst_endp_err            @   have been an error
ldr     r5, =setup_endp_jump_table
ldr     r5, [r5, r1, lsl #2]    @ Jump to the right label to deal with the
bx      r5                      @   message.

dst_do_req_interface:
cmp     r1, #INT_bReq_MAX_VAL   @ If larger than this value, then there must
bhi     dst_int_err             @   have been an error
ldr     r5, =setup_int_jump_table
ldr     r5, [r5, r1, lsl #2]    @ Jump to the right label to deal with the
bx      r5                      @   message.

dst_do_req_device:
cmp     r1, #DEV_bReq_MAX_VAL   @ If larger than this value, then there must
bhi     dst_dev_err             @   have been an error
ldr     r5, =setup_dev_jump_table
ldr     r5, [r5, r1, lsl #2]    @ Jump to the right label to deal with the
bx      r5                      @   message.

dst_dev_get_status:             @ GET_STATUS on device
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_FDR0       @ Send off the status of the device, which is
mov     r1, #DEV_GET_STATUS_BOT_BYTE
str     r1, [r0]                @  (possibly counterintuitively) a constant.
mov     r1, #DEV_GET_STATUS_TOP_BYTE
str     r1, [r0]

ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0
ldr     r1, [r0]
orr     r1, r1, #TXPKTRDY_MSK   @ Set TXPKTRDY to indicate we have data to send
str     r1, [r0]
b       dst_tear_down
```

```
dst_dev_set_des:                @ Not needed. Hosts don't really set descriptors.
dst_dev_set_feat:               @ Unimplemented for now. Should not be
dst_dev_clr_feat:               @   necessary. Probably.
b       dst_tear_down

dst_dev_set_addr:
@ TODO: Check for maximum address of 128, reject, throw error @@@@@@@@@@@@@@@@@@
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_FADDR      @ Load the FADDR register to tell the UDP
ldr     r0, [r0]                @   device the address we just got.
orr     r0, r0, r2

ldr     r0, =address_set        @ Set this flag in memory to indicate we have
mov     r1, #TRUE               @   to update the state of the UDP device once
strb    r1, [r0]                @   the status packet gets sent.

dst_dev_get_des:                @ Get descriptor request, tells the host who we are.
and     r2, r2, #DES_TYPE_MASK  @ Get the descriptor type that the host wants
mov     r2, r2, lsr #8          @ Shift it by a byte to right to get the actual
cmp     r2, #dev_desType        @  value.
ldreq   r0, =devDescriptor      @ Figure out which kind of descriptor the host
moveq   r1, #dev_bLength        @  wants so that we can give it to it.

cmp     r2, #int_desType
ldreq   r0, =intDescriptor
moveq   r1, #int_bLength

cmp     r2, #endp_desType
ldreq   r0, =endpDescriptor
moveq   r1, #endp_bLength

cmp     r2, #conf_desType
ldreq   r0, =confDescriptor
moveq   r1, #conf_wTotLen1

@ TODO: What if it's none of these? Error? Send a STALL to tell the host?

ldr     r3, =UDPBaseAddress
add     r3, r3, #UDP_FDR0       @ Load the queue address so that we can start
                                @  adding elements
dst_dev_get_des_loop:           @ Loop until the descriptor is fully in queue
ldrb    r2, [r0]
strb    r2, [r3]
add     r0, r0, #1              @ Go on to next byte of descriptor
subs    r1, r1, #1              @ And check to see if we've sent the entire
bne     dst_dev_get_des_loop    @  thing. If not, then go back to loop.

ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0
ldr     r1, [r0]
orr     r1, r1, #TXPKTRDY_MSK   @ Set TXPKTRDY to indicate we have data to send
str     r1, [r0]                @ NOTE: The queue could theoretically overflow
                                @  here, but it SHOULDN'T, because the packet
b       dst_tear_down           @  size of 32 bytes fits every descriptor.

dst_dev_get_conf:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_GLB_STAT   @ Need to get whether the device is configured
ldr     r0, [r0]                @  right now so we can send this information to
ands    r0, r0, #UDP_CONFG_MASK @  the host.
movne   r0, #TRUE
moveq   r0, #FALSE
ldr     r1, =UDPBaseAddress
```

```
add     r1, r1, #UDP_FDR0       @ Store this flag into the control queue
str     r0, [r1]                @  to send it off to the host

ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0
ldr     r1, [r0]
orr     r1, r1, #TXPKTRDY_MSK   @ Set TXPKTRDY to indicate we have data to send
str     r1, [r0]
b       dst_tear_down

dst_dev_set_conf:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_GLB_STAT   @ First check that we're in the address state,
ldr     r2, [r0]                @  otherwise something went wrong and we need
ands    r1, r0, #UDP_FADDEN_MASK    @ to throw an error.
beq     dst_dev_err
orr     r2, r2, #UDP_CONFG_MASK @ If in address state, we can safely set the
str     r2, [r0]                @  the config flag. Since this device has only
b       dst_tear_down           @  one configuration, we can end here.

dst_int_get_status:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_FDR0       @ Send the status to the host by putting it
mov     r1, #INT_GET_STATUS_BOT_BYTE
str     r1, [r0]                @  on the queue.
mov     r1, #INT_GET_STATUS_TOP_BYTE
str     r1, [r0]

ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0
ldr     r1, [r0]
orr     r1, r1, #TXPKTRDY_MSK   @ Set TXPKTRDY to indicate we have data to send
str     r1, [r0]
b       dst_tear_down

dst_int_clr_feat:               @ No features specified for interfaces
dst_int_set_feat:               @   in USB version 2.0 (and 1.1). Nothing to
b       dst_tear_down           @   do here.


dst_int_get_int:                @ Don't really need to implement this feature
dst_int_set_int:                @   since we don't plan on using alternative
b       dst_tear_down           @   interfaces (or multiple interfaces at all)

dst_endp_get_status:
ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_FDR0       @ Send the status of the endpoint by putting
mov     r1, #ENDP_GET_STATUS_BOT_BYTE
str     r1, [r0]                @  it in the queue.
mov     r1, #ENDP_GET_STATUS_TOP_BYTE
str     r1, [r0]

ldr     r0, =UDPBaseAddress
add     r0, r0, #UDP_CSR0
ldr     r1, [r0]
orr     r1, r1, #TXPKTRDY_MSK   @ Set TXPKTRDY to indicate we have data to send
str     r1, [r0]

b       dst_tear_down

dst_endp_clr_feat:              @ Not implemented. Should not be necessary
dst_endp_set_feat:              @   for our configuration
dst_endp_synch:
b       dst_tear_down
```

```
dst_dev_err:                    @ TODO: Error correction
b       dst_dev_err

dst_int_err:                    @ TODO: Error correction
b       dst_int_err

dst_endp_err:                   @ TODO: Error correction
b       dst_endp_err

dst_tear_down:
ldr     r1, =UDPBaseAddress
add     r1, r1, #UDP_CSR0       @ Load CSR0 and clear the RXSETUP flag to
ldr     r0, [r1]                @  indicate we processed the setup packet
ldr     r4, =RXSETUP_CLR_MASK
and     r0, r0, r4
str     r0, [r1]

ldr     r4, =UDPBaseAddress
add     r4, r4, #UDP_CSR0
ldr     r4, [r4]
ands    r4, r4, #DIR_BIT_MASK_CSR   @ We can send the status here (I think?)
bleq    do_status_transaction       @  since we're not expecting any more data

ldmia   sp!, {r0-r6, lr}
bx      lr


@  do_data_out_transaction
@
@  Description:         Processes a DATA OUT transaction. This happens when
@                        the device receives some data from the host.
@
@  Operation:           Since we're not expecting any data from the host right
@                        now, we'll ignore the data itself. This just clears
@                        the flag so that the UDP Device can move on.
@
@  Arguments:           r0 - Value of CSR register corresponding to the endpoint
@                       r1 - Endpoint number
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               Data from host.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      06/18/2015    Daniel Andrade    initial revision
```

```
.type do_data_out_transaction STT_FUNC
do_data_out_transaction:
stmdb   sp!, {r1-r3}

ddo_normal:
ldr     r2, =UDPBaseAddress

cmp     r1, #0              @ Load the right CSR register to update the
addeq   r2, r2, #UDP_CSR0   @  data read flag
addne   r2, r2, #UDP_CSR1
ldr     r3, =BKO_BIT_MASK_CSR
and     r3, r3, r0          @ Clear the RX_DATA_BK0 flag
str     r3, [r2]            @ TODO: Does this take care of STATUS_OUT
                            @  automatically? I think so...
ddo_tear_down:
ldmia   sp!, {r1-r3}
bx      lr

@  do_data_in_transaction
@
@  Description:         Processes a DATA IN transaction. This happens when
@                        the hosta acknowledges the data we sent to it.
@
@  Operation:           There's nothing much to do here except for error
@                        correction, if any. All we really have to do is clear
@                        the TXCOMP flag to indicate that we're done with
@                        the transaction.
@
@  Arguments:           r0 - Value of CSR register corresponding to the endpoint
@                       r1 - Endpoint number
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               None.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      06/18/2015    Daniel Andrade    initial revision
@      08/08/2015    Daniel Andrade    wrote the actual code (replaced stub)

.type do_data_in_transaction STT_FUNC
do_data_in_transaction:
stmdb   sp!, {r1-r3}

ldr     r2, =UDPBaseAddress
cmp     r1, #0              @ Load the address of the correct control register
addeq   r2, r2, #UDP_CSR0   @  so that we can clear the TXCOMP flag.
addne   r2, r2, #UDP_CSR1
ldr     r3, =TXCOMP_MSK     @ Prepare to clear TXCOMP
```

```
mvn     r3, r3              @ Negate it to so that we can use it to clear the value
and     r3, r3, r0

str     r3, [r2]            @ And store the value of the CSR with the flag cleared

ldmia   sp!, {r1-r3}
bx      lr


@  do_setup_transaction
@
@  Description:         Processes a SETUP transaction. This happens at the end
@                        of a control transaction and is basically an empty
@                        DATA IN or DATA OUT transaction.
@
@  Operation:           TODO
@
@  Arguments:           r0 - Value of CSR register corresponding to the endpoint
@                       r1 - Endpoint number
@
@  Return Value:        None.
@
@  Local Variables:     None.
@
@  Shared Variables:    None.
@
@  Global Variables:    None.
@
@  Input:               Data from host.
@
@  Output:              None.
@
@  Error Handling:      None.
@
@  Limitations:         None.
@
@  Algorithms:          None.
@  Data Structures:     None.
@
@  Registers Changed:   None.
@
@  Revision History:
@      06/18/2015    Daniel Andrade    initial revision
@      07/24/2015    Daniel Andrade    finished implementation
do_status_transaction:
stmdb   sp!, {r0-r3}

ldr     r2, =did_setup      @ If we're here, we should have done a setup
ldrb    r3, [r2]            @  transaction at some point
cmp     r3, #FALSE
beq     dset_error
cmp     r1, #0              @ We should only get setup transactions on endpoint
bne     dset_error          @  zero, since that's the control endpoint.

mov     r3, #FALSE          @ This is the end of the control cycle, so set
strb    r3, [r2]            @  the flag to FALSE to indicate this.

ldr     r3, =address_set    @ If we got a set_address request in the setup
ldrb    r2, [r3]            @  transaction, then need to set FADDEN here.
cmp     r2, #TRUE
bne     dset_get_dir
mov     r2, #FALSE          @ Now that we set FADDEN, we can set this variable
strb    r2, [r3]            @  back to false.
```

```
ldr     r3, =UDPBaseAddress @ Set FADDEN to indicate device now has address
add     r3, r3, #UDP_GLB_STAT
ldr     r2, [r3]
orr     r2, r2, #UDP_FADDEN_MASK
str     r2, [r3]


dset_get_dir:
ands    r2, r0, #DIR_BIT_MASK_CSR2
bne     dset_dir_in         @ This is a DIRECTION IN instruction (wrt the host)
                            @  it is going out of the device.

                            @ Otherwise it's going into the DEVICE (a DATA
@b      dset_dir_out        @   OUT instruction)

dset_dir_out:
ldr     r2, =UDPBaseAddress
add     r2, r2, #UDP_CSR0   @ All we have to do to acknowledge this is to
ldr     r0, [r2]            @  clear the RX_DATA_BK0 flag.
ldr     r3, =BKO_BIT_MASK_CSR
and     r3, r3, r0          @ Clear the RX_DATA_BK0 flag here
str     r3, [r2]
b       dset_tear_down

dset_dir_in:                @ Get the value of the TXPKTRDY flag to determine
ldr     r2, =UDPBaseAddress @  if we can send the empty data packet. If we can,
add     r2, r2, #UDP_CSR0   @  then set TXPKTRDY to send it.

dset_dir_in_loop:
ldr     r0, [r2]
ands    r1, r0, #TXPKTRDY_MSK
bne     dset_dir_in_loop    @ Wait until TXPKTRDY is clear

orr     r0, r0, #TXPKTRDY_MSK   @ Set the TXPKTRDY flag to send the empty packet
str     r0, [r2]
b       dset_tear_down

dset_error:                 @ TODO: Do something smart with errors
b   dset_error

dset_tear_down:
ldmia   sp!, {r0-r3}
bx      lr

.section .data
.align 2

@ Arrays for initialization of UDP registers
UDPInitVals:
.word   TXVCVal, REST_EP_ON, REST_EP_OFF, UDPCSR0InitVal, UDPCSR1InitVal, UDPIDRVal, U
DPIERVal

UDPInitOffsetArray:
.byte   UDP_TXVC, UDP_REST_EP, UDP_REST_EP, UDP_CSR0, UDP_CSR1, UDP_IDR, UDP_IER

address_set:
.byte   FALSE

did_setup:
.byte   FALSE

.align 2
@ Jump tables for setup
setup_dev_jump_table:
```

```
.word   dst_dev_get_status, dst_dev_clr_feat, dst_dev_err, dst_dev_set_feat, dst_dev_e
rr, dst_dev_set_addr, dst_dev_get_des, dst_dev_set_des, dst_dev_get_conf, dst_dev_set_
conf

setup_int_jump_table:
.word   dst_int_get_status, dst_int_clr_feat, dst_int_err, dst_int_set_feat, dst_int_e
rr, dst_int_err, dst_int_err, dst_int_err, dst_int_err, dst_int_err, dst_int_get_int,
dst_int_err, dst_int_err, dst_int_err, dst_int_err, dst_int_err, dst_int_err, dst_int_
set_int

setup_endp_jump_table:
.word   dst_endp_get_status, dst_endp_clr_feat, dst_endp_err, dst_endp_set_feat, dst_e
ndp_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_end
p_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_endp_err, dst_endp_
err, dst_endp_err, dst_endp_synch

devDescriptor:
.byte   dev_bLength, dev_desType, dev_bcdUSB1, dev_bcdUSB2, dev_bDevClass, dev_bDevSub
Class, dev_bDevProt, dev_bMaxPackSize, dev_idVendor1, dev_idVendor2, dev_idProduct, de
v_idProduct2, dev_bcdDevice1, dev_bcdDevice2, dev_iMan, dev_iProduct, dev_iSerial, dev
_bNumConf

confDescriptor:
.byte   conf_bLength, conf_desType, conf_wTotLen1, conf_wTotLen2, conf_bNumInts, conf_
bConfVal, conf_iConf, conf_bmAttr, conf_bMaxPower, int_bLength, int_desType, int_bIntN
um, int_bAltSetting, int_bNumEndpoints, int_bIntClass, int_bIntSubclass, int_bIntProt,
 int_iInterface, endp_bLength, endp_desType, endp_addr, endp_bmAttr, endp_maxPackSize1
, endp_maxPackSize2, endp_bInterval

intDescriptor:
.byte  int_bLength, int_desType, int_bIntNum, int_bAltSetting, int_bNumEndpoints, int_
bIntClass, int_bIntProt, int_iInterface

endpDescriptor:
.byte   endp_bLength, endp_desType, endp_addr, endp_bmAttr, endp_maxPackSize1, endp_ma
xPackSize2, endp_bInterval

.end
```

F Acknowledgements

This section is dedicated to all the wonderful people and tools that helped get this project done. It is largely written in the first person to give a more personal feel. A big thank you to everyone that helped me fulfill my ambitions.

F.1 People

I would firstly like to thank Glen George for making this possible, helping to set it up, answering the many questions I asked, and helping out with many of the critical issues that came up along the way. I am additionally extremely grateful for the deadline extension for the project, which allowed me to work throughout the summer.

I also want to send a big thank you to my teaching assistant, Andy Zhou, for his help, especially with figuring out how to use the AT91RM9200 processor. I'll include here a shout-out to my friend Santiago Navonne for helping me out with some issues and for his concern, too.

I would also like to thank Pablo Carrica for his wonderful project idea and for his help during the initial brainstorming sessions and beyond. I would like to thank the rest of my family for their support as well.

F.2 Software

I would like to acknowledge the following software in helping me get the project done.

1. Altium for schematics and PCB design
2. OCDCommander for running and debugging my assembly code
3. GNU Code Sourcery for their assembler, linker, and object copy programs.
4. Vim (esp. the graphical port gVim), a wonderful text editor.
5. LaTeX and especially the PDF compiler pdflatex to help create this documentation.
6. The Advanced Circuits service FreeDFM for looking over my board and ensuring that it could be manufactured.
7. Inkscape for all the vector diagrams I drew, both in my proposal and in this documentation.
8. PDFScissors for the various PDF clippings (especially of my schematics) so that I could keep my diagrams in a nice, scalable format.

F.3 Other Acknowledgements

I would like to thank Texas Instruments and Maxim Integrated for sending me free samples of some of their chips.