H2O-Rocket-Control

Clayton G, NEW ZEALAND

clayton@isnotcrazy.co.nz

This revised document was originally written for the AVR design competition – entry AT3409
It has since been modified to include my contact details.

Introduction

This project really is rocket science! Recently I have been building water rockets (for the kids of course!). These are built from pop bottles, filled with some water, pumped up with a bike pumped, and released – and do they fly. However the deployment of the parachute, and the determination of altitude, is an ongoing problem. The usual arrangement has been a nose-cone that should just fall off at the top of the climb. Often the cute would not deploy at all, resulting in a quick and destructive (to the rocket) trip to the ground, or the cute would deploy too early, significantly reducing the height reached.

This project is designed to overcome these problems. The rocket controller includes:

- an ATmega16 AVR CPU
- an altimeter and some accelerometers (three of them to cover the X,Y and Z axis). These record the performance of the rocket.
- a deployment output to fire a parachute release mechanism after a delay.
- An IrDA interface to transfer data to/from a Palm PDA.
- A beeper to indicate the rocket status, and to aid finding a lost rocket.

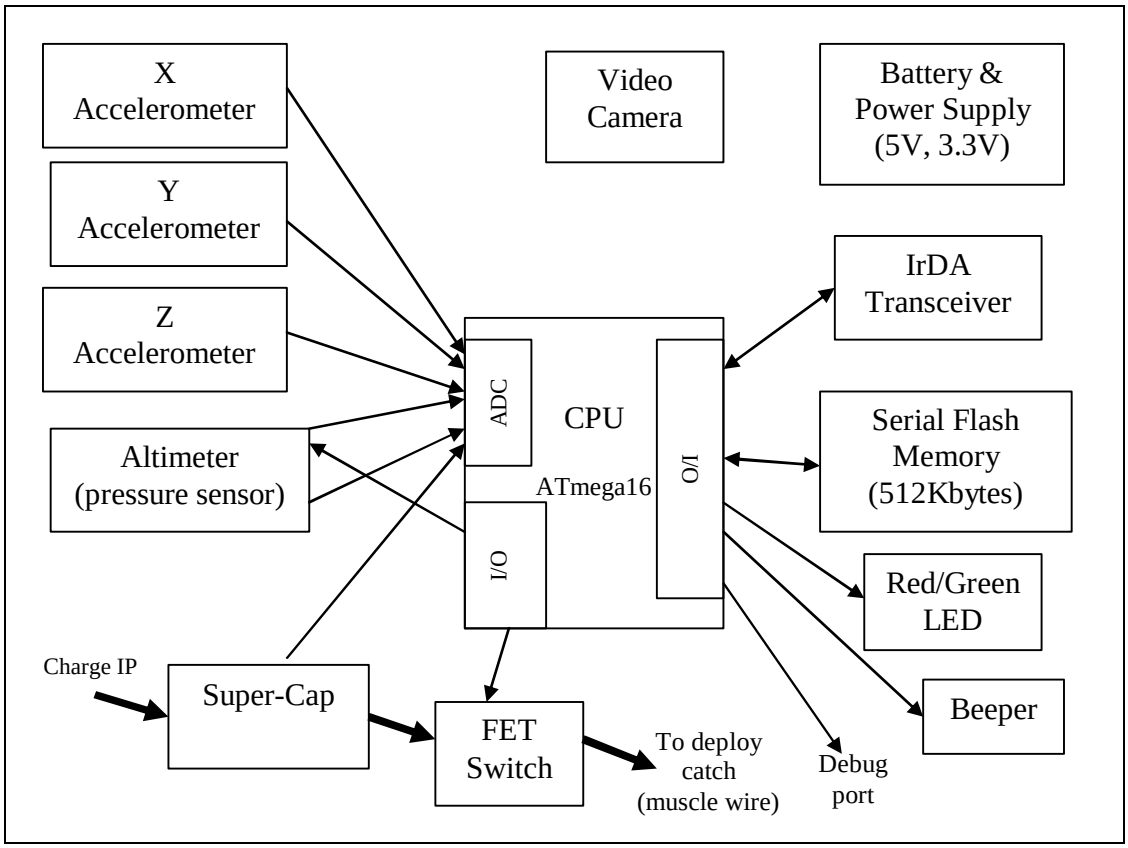
The controller fits into the rocket nose-cone. Below it is the parachute. When the cute is to be deployed, the controller releases the nose cone which then falls off and allows the cute to come out. Nose-cone and rocket body are attached to the cute, so the rocket will (hopefully) gently fall to the ground without any damage.

The rocket nose-cone also includes a small video camera to record the flight – ‘Rocket-Cam’.

The complete H2O Rocket Controller in the nose-cone, atop the parachute compartment. All ready for installation onto the rocket body.

The H2O Rocket Controller installed on a rocket, mounted on the launcher.

System Details



System Block Diagram

CPU

The CPU chosen is an ATMega16. This provides sufficient program memory, and also has all of the necessary peripherals to interface the hardware needed.

Altimeter

A Freescale (Motorola) MPX4115A altimeter is used to record the atmospheric pressure, and hence determine altitude. To obtain a reasonable resolution would require an ADC of about 14 bits. However the ATMega16 only has a 10-bit ADC. To improve its resolutions over-sampling is used (see Atmel app-note AVR121 “Enhancing ADC resolution by oversampling”). By taking many samples (256 to be exact) and by introducing noise (in the form of a triangular wave created using a PWM output) the resolution is theoretically increased by 4 bits.

The altimeter is power directly from a CPU port. This allows it to be turned on and off to save power when it isn’t needed. Two CPU port pins are used to provide enough current. The actual voltage on the altimeter VCC pin is read by an ADC input so that correct for the VCC voltage can be applied if necessary.

Accelerometers

Three Analog Devices AD22281 accelerometers are used to measure acceleration in all three dimensions. These small (5mm x 4.5mm) devices are micro-machined iMEMS accelerometers, capable of measuring up

to +- 70g. The main dimension of interest is the vertical acceleration, as this provides an indication of when take-off occurs.

Serial Flash Memory

To record the flight data an SPI flash memory device is used – an M25P40. This provides 512Kbytes of storage – enough for many flights.

IrDA Interface

A link to a PDA is provided through an IrDA interface. The only hardware needed here is the IrDA transceiver – an Agilent HSDL-3201. Usually a IrDA encoder/decoder device is also necessary to translate the transceiver signal to standard asynchronous serial data for processing through a UART. In my approach I have done away with this, and the UART. Instead translation of the transceiver signals to serial bytes is done in software, with the help of a timer and an interrupt input.

LED

A dual colour LED is used to provide an indication of the rocket status.

Power Supply

The whole system needs a light-weight power source. A 12V battery, as usually used in car remote locks, is used. This supply is regulated down to 5V for most of the electronics. There is also a 3V regulator for the flash memory device. A magnetic reed switch is used to turn the controller on and off. A N.C. switch is used, so that when a magnet is brought close to the controller it is turned off.

Beeper

A beeper is used to indicate the rocket state. It also provides a means to synchronise the recorded data with the on-board video camera. The beep state is recorded along with the other data. By listening to the beeps in the video recording, it should be possible to tie observed video with the flight data at the time.

Deployment System (Super-Cap & FET)

Parachute deployment uses a catch which will release the nose-cone from the top of the rocket. The parachute is held inside the rocket along with some foam so that when the catch is released the nose cone will be pushed off. The catch is spring loaded, with a latch which when activated, allows the catch to spring back. The activation mechanism uses ‘muscle-wire’. This wire is a special alloy which contracts when heated (with an electrical current). Muscle-wire provides a much lighter and more power-full solution than a solenoid, however the wire selected requires about 400mA to operate – too much for a small 12V battery. A super-capacitor (4.7F) is therefore used to provide sufficient charge to operate the muscle-wire. This capacitor needs to be charged from an external source before the flight. It is only a 2.5V capacitor, but this is enough voltage for the wire. A FET is used to turn on the current to the muscle-wire. The CPU monitors the voltage on the capacitor, and indicates the level of charge through the beeper.

Debug Port

The AVR UART was used as a debug port as well as software download port (using my own boot loader app). A small USB-serial device was used, with a 3-pin header to attach to the UART pins.

Video Camera

Although not directly part of the rocket controller, the nose-cone also has a small video camera. This is a USB memory stick with a low resolution camera. It stores about 5 minutes of video. The camera is easily operated through two buttons. The camera is simply activated at the start of a flight (as the rocket is assembled on the launcher) and left running as the rocket is readied and launched.

Operation

When turned on the controller inspects the flash memory, and determines the end of any data already in the memory. New data can then be added at this point.

It then sits there occasionally beeping, waiting for a sudden upwards acceleration (indicating a launch) or for an IrDA connection. It is also collecting and saving data (although saving at a reduced rate).

On acceleration the controller goes into flight mode. The beeping changes. Data is saved at the maximum rate (20 records per second). After a set time (configurable) into this state the parachute is deployed, and the controller then moves to post-flight mode.

In post-flight mode the beeps are longer (to assist in finding the rocket). The controller is also ready for an IrDA transfer, and remains in this state until a transfer attempt is made.

The parachute deployment system needs to be charged. This requires a voltage source (12V @ 100mA) to be clipped to the charging points. While charging, the beeper indicates the charge state, beeping faster and faster until the system is fully charged (a continuous tone). Charging takes about 1 minute.

IrDA transfers are used to configure the controller, request the status, erase the flash, and request upload of data. When an IrDA request is made (by beaming a memo to the controller), a response is beamed back. In the case of data upload, the response may consist of many memos.

The commands that are supported are:

status	report the controller status
upload	request upload of data from Flash memory
erase	erase the flash memory
flytime <n>	set the flight time (initial acceleration to parachute deployment) to <n> mSecs
accelthresh <n>	set the vertical acceleration threshold to <n>. If the accelerometer output reading falls below the threshold, it indicates a launch. Note that due to the accelerometer orientation, upwards acceleration (downward G) results in the reading falling.

Response memos are sent with the first line indicating the reply number (e.g. ‘reply12’). This number is stored in the CPU EEPROM, and increments with every reply memo (even over power cycles).

Uploaded data is formatted as comma-separated decimal values (CSV file format). These records contain:

reset count	A number that increments with each power cycle of the controller
time	mSec timer. Wraps around every 65536 mSecs
Acceleration-Z	Vertical acceleration reading. 512 (approx) = 0g. Lower for upwards g.
Acceleration-X	X acceleration reading. Should be close to 512 when sitting ready for launch.

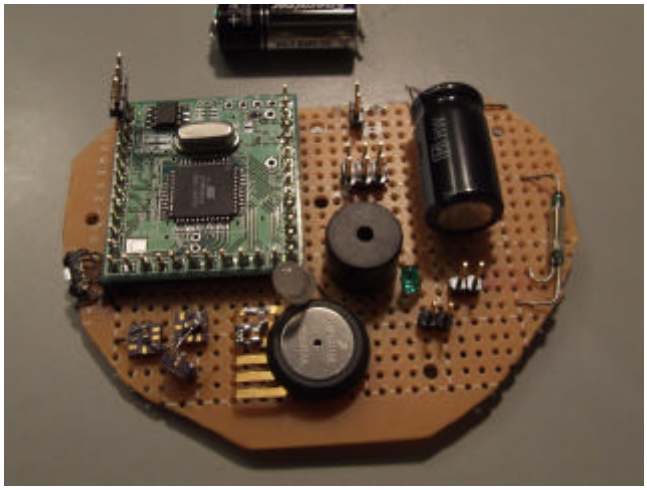
Acceleration-Y	Y acceleration reading. Should be close to 512 when sitting ready for launch.
Capacitor Charge	Super-cap voltage. 512 = fully charged.
Charge input	Voltage on charge input. 0 when disconnected. 1023 when voltage applied (assuming >5V)
Accessory channel	Not used – reads 1023
Altimeter VCC	Supply voltage to altimeter. Usually close to 1023 (5V)
Altimeter reading	18 bit reading – sum of 256 ADC readings
Controller state	0=IDLE; 1=IRDA; 2=FLIGHT; 3=POST-FLIGHT
Beeper state	1 = beeper on
Red LED state	1 = LED on
Green LED state	1 = LED on
Deployment state	1 = Deployment output active

Data Processing

The IrDA transfers are set up to support the Palm PDA 'memo' application. In this way no special Palm software is necessary. All that is needed is a few memos of commands that can be beamed to the rocket. After a memo has been beamed, a memo is beamed back in response. Unfortunately the Palm memo application has a memo size limit of only 4K bytes. This made it necessary for uploading data to consist of many memos, each with about 50 records. When the upload command is initiated, a potentially large number of memos are returned. Each of these needs to be accepted (quickly – before the next one is sent!). Once all data is in the PDA the flash can be erased (with the 'erase' command). The data memos can be transferred to a PC by synchronising the PDA and copying the data into a text file. Luckily the PDA desktop application makes this easy – multiple memos can be copied at once and pasted into a file in one step. With some simple editing a clean CSV file can be produced, ready for data analysis in Excel.

Hardware

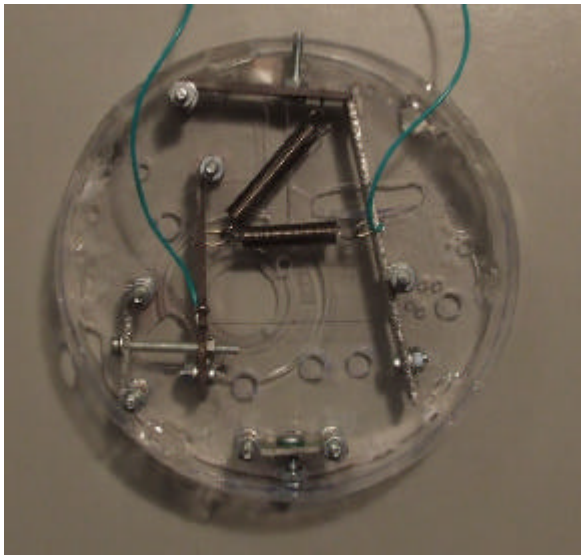
The main CPU, regulators and Flash memory were already on a PCB developed for another project. This PCB could then be placed onto a piece of prototype PCB and the rocket-controller specific components added. SMD components were soldered to the underside of the PCB, and the larger components mounted on top. Thin wire-wrap wire was used to interconnect everything.



The Complete PCB.

The accelerometers needed to be oriented into the 3 axes – X, Y & Z. This made it necessary for one of them to be mounted on its end. The IrDA transceiver is positioned at the PCB pointing out so that it is visible through the nose-cone. Once tested the components were secured with hot-melt glue, so that they would survive the rigors of rocket flight!

The whole PCB assembly is mounted on top of the parachute release mechanism. This mechanism is assembled on a plastic base. Bolts are used to hold down levers made from scraps of PCBs, with bearing made of small copper tubes – an outside tube soldered to the lever, with an inside tube secure to a bolt. Springs provide the required tension. The muscle-wire is secured between two moving levers – one of these being the main catch release lever, the other being a tension adjustment lever with a screw to set the wire tension. Some slots in the base of the mechanism allow the catch to be reset.



The Deployment Catch

Over this whole assembly is placed the top of a pop bottle. Attached to the side of this bottle, wrapped in foam rubber and secures in a sleeve, is the video camera. Some strategically placed holes allow the camera to be operated using a stick through these holes.

This whole nose cone then sits into the parachute storage compartment, held down with the catch mechanism. A spring loaded catch opposite the release catch allows the mechanism to be in lock in place with the catch engaged. The parachute compartment is in turn attached to the body of the rocket. String is used between the parachute and the rocket body and nose-cone (or a separate parachute on the body and nose-cone can be used). To ensure that the nose-cone does actually separate from the body, some foam is packed with the cute to provide pressure onto the nose-cone.

The rocket controller uses the AVR CPU resources as follows:

Timer 0	Main software timer.
Timer 1	PWM output used to add 'noise' to the altimeter signal
Timer 2	IrDA software UART timer
INT1	IrDA receive pulse input
ADC 0	Altimeter VCC voltage (used to compensate)
ADC 1	Super-cap voltage
ADC 2	Altimeter signal (with added noise)
ADC 3	Accessory input (currently unused)
ADC 4,5,6	Accelerometers
ADC 7	Charge points input
UART	Debug port
SPI	Flash memory interface
PB2,3	LED outputs
PB0,1	VCC driver to altimeter
PC2	IrDA TxD pin
PC3	IrDA shutdown pin
PD6	Deployment mechanism FET drive
PC5	Beeper output

Software

Application

The software is written in C using the WinAVR tool set. Many of my common libraries (e.g. command consoles, command interpreters, drivers) as used in other projects were reused in the controller.

The IrDA stack uses the 'Pico-IrDA' stack written by Gerd Rausch of BlauLogic (see http://blaulogic.com). This stack provides an API to transfer objects (in my case, memo files) between the application and a Palm PDA. The code dropped straight in with no modification at all! All I had to do was write the physical layer API for my hardware setup.

The bulk of the application is in a single file - 'RocketAVR.c'. This contains all state processing and support routines.

The AVRs UART was used as a debug port. Console messages allowed debug messages during development. It also provided the download system through a boot-loader application.

Timer 0 is the main system clock – the timer ISR fires every 256uSec, and just increments a counter. The 'Background' routine in the main execution loop then handles the individual timers and clocks based on this counter to provide a number of mSec resolution timers and clocks (where timers count down to 0 and stop, and clocks just increment for ever). In this way the timer system has minimal ISR overhead, and the main loop timer processing removes a lot of management of ISR and main loop interactions.

Timer 2 is used for the IrDA timing – receive and transmit (not at the same time though – IrDA is half-duplex). IrDA receive uses INT1 to capture IR pulses. The times of these pulses is recorded from Timer 2 and used to create the received byte. IrDA transmit uses the timer for bit timing, sending out a short pulse (software timed) for any 0 bit in the byte (including the start bit).

The software records all ADC channels. These are: Altimeter, Altimeter power supply (used to correct the altimeter reading), accelerometers (3 of), super-cap charge level, the super-cap charge input voltage, and accessory channel (as yet unused, but planed to be used with a light sensor to detect sky and ground and hence determine rocket orientation – based on the assumption that the sky is brighter than ground).

All recorded data (system time, ADC data and status flags) is packed into 16 bytes, and stored in the serial flash. When data is extracted, the 16 bytes are read out, unpacked, and formatted into a line of ASCII text of comma separated values. These lines are then sent as a memo message (which is simply a .txt file).

The main software loop moves between the four states (see below). A background function is used by all states to manage the timers, beeper and LED operation. Each state function then uses the timers to perform it required processing (such as sampling data or operating the deployment timer).

States

The controller has 4 states of operation:

Idle state

When sitting on the launch pad it is idle. The beeper occasional beeps. All data is being captured (at 20 samples per second), with every 4th data set being saved to flash memory. The controller is also monitoring the IrDA port, looking for a beginning-of-frame (BOF) character.

IrDA state

When an IrDA BOF is received this state is entered. The IrDA stack is initialised and set up to receive an object. This object is processed as a series of commands. A simple command interpreter recognised valid commands and either actions them immediately (as in the case of parameter configurations) or sets a flag for them to be performed once the object receive is complete.

Once the receive operation is complete, there is a small delay before the controller sends a file (or files) in response. This file consists of the response to the commands, which may include the status of the controller. Subsequent files may follow consisting of recorded data formatted as comma separated fields.

Once the data transfers are complete (or fail), the state returns to 'Idle' state.

Flight State

Once acceleration reaches a threshold (configurable through a command), the rocket goes from idle state to flight state. All captured by unsaved data is saved to flash memory, and all data capture is now stored. The deployment timer (also configurable through a command) is started. The beeper beep rate increases.

When the deployment timer expires, the deployment FET is activated and the state goes to post-flight state.

Post-Flight state

This state is when the rocket is (hopefully) sailing safely down to earth under it parachute and is waiting to be collected. The beeper gives longer beeps, as an aid to find the rocket in case has landed out of site. Data is still stored at maximum rate for a time (30 seconds), but then reduces to every 4th data set as in idle state. The IrDA port is monitored, and on a BOF character, the controller goes to IrDA state. This is in fact the only way to exit the post-flight state (other than turning the recorded off!). Post-flight state also handles the turning off of the deployment system FET, turning it off 2 seconds after the state is entered.

Charging

Although not a separate state, the beeper has special handling of the capacitor charging. When a voltage is present on the charge input, the beeper operates as a 'fuelling' level indicator. In this mode it beeps at a rate determined by the capacitor voltage. Slow beeping when it is low, to faster and faster until it is a constant tone when the charge reaches 2.5V.

Boot-Loader

Although not an essential part of the project, the boot-loader helped speed up development by allowing for quick software updates. The boot-loader is my standard loader used in a number of other applications. As well as xmodem transfers (to load in software), it supports flash status reporting, and supports system with multiple CPUs on a shared serial port bus.

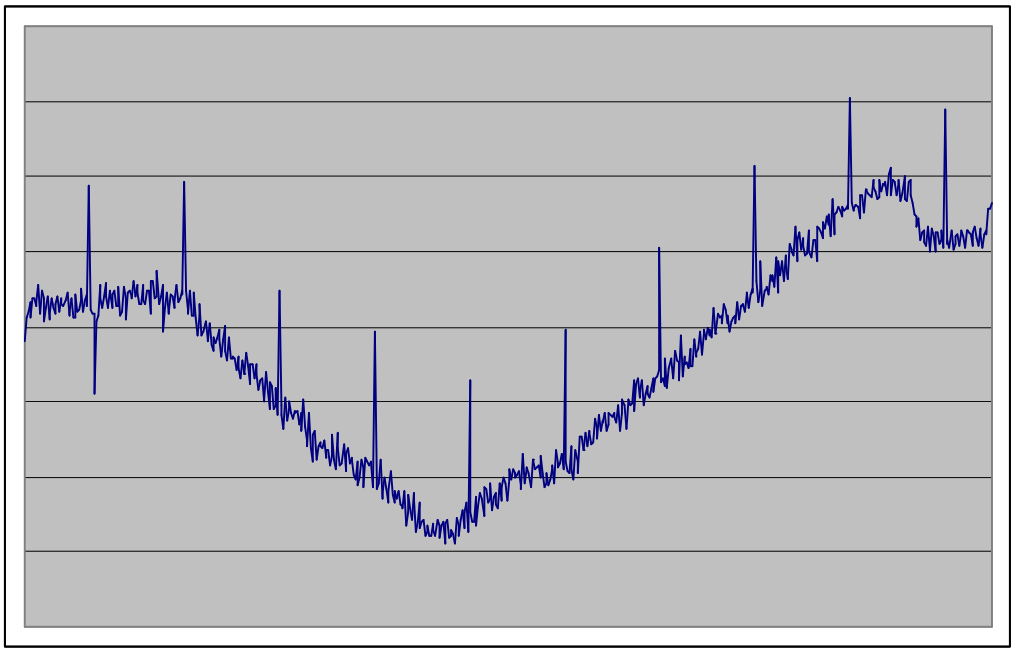
The boot-loader runs at reset. It operates for 2 seconds, waiting for a trigger character (a ';') on the UART. If received, the boot loader is entered, from which an xmodem transfer can be initiated. If not, the application is run (from address 0). The boot-loader is written in assembler (to keep it small). It fits in the top 512 bytes of the flash.

Testing and Results

Time constraints did not permit a full test on the top of a rocket, but some basic testing was performed to check operation. These tests were successful.

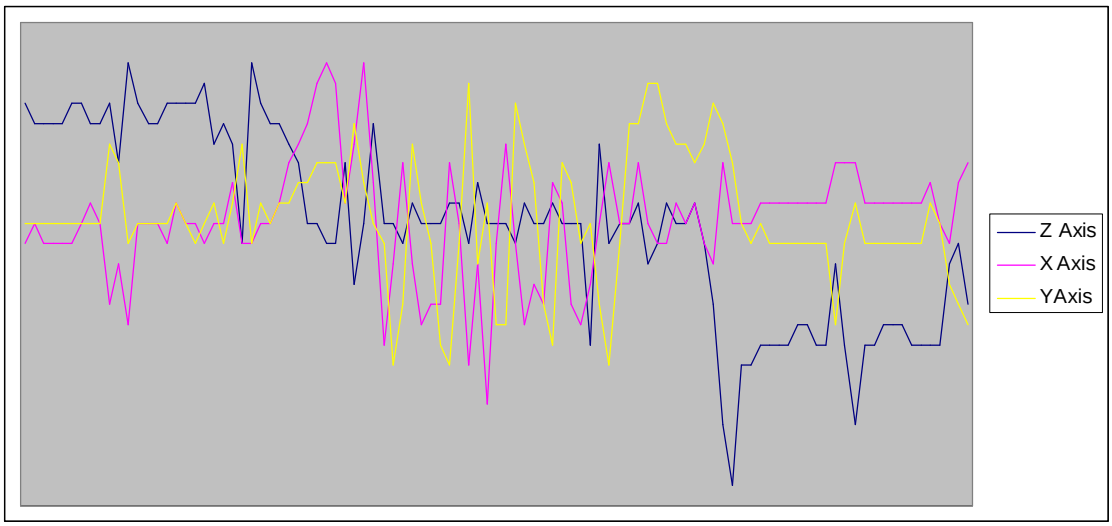
When the rocket nose-cone assembly was thrown up into the air, it entered flight state, and soon after the deployment mechanism fired and the nose-cone broke away from the cute compartment.

The altimeter and accelerometers were tested by carrying the controller up and down the stairs. After some processing of the spread-sheet data to remove invalid (wildly out) readings, the altimeter graph clearly showed the pressure dropping as I went up the stairs, and increasing as I went down (see below). This showed that the over-sampling was working. There was however a lot of noise on the signal from the beeper. This is possibly due to the beeper being directly driven through a CPU port bit – the beeper load possibly affected the altimeter VCC voltage. A small hardware change (driving the beeper and possibly the altimeter via a FET) should fix this.



Altimeter data – spikes correspond to beeper beeps.

The accelerometers clearly showed the orientation of the controller. The vertical throw up (not shown) was clearly visible.

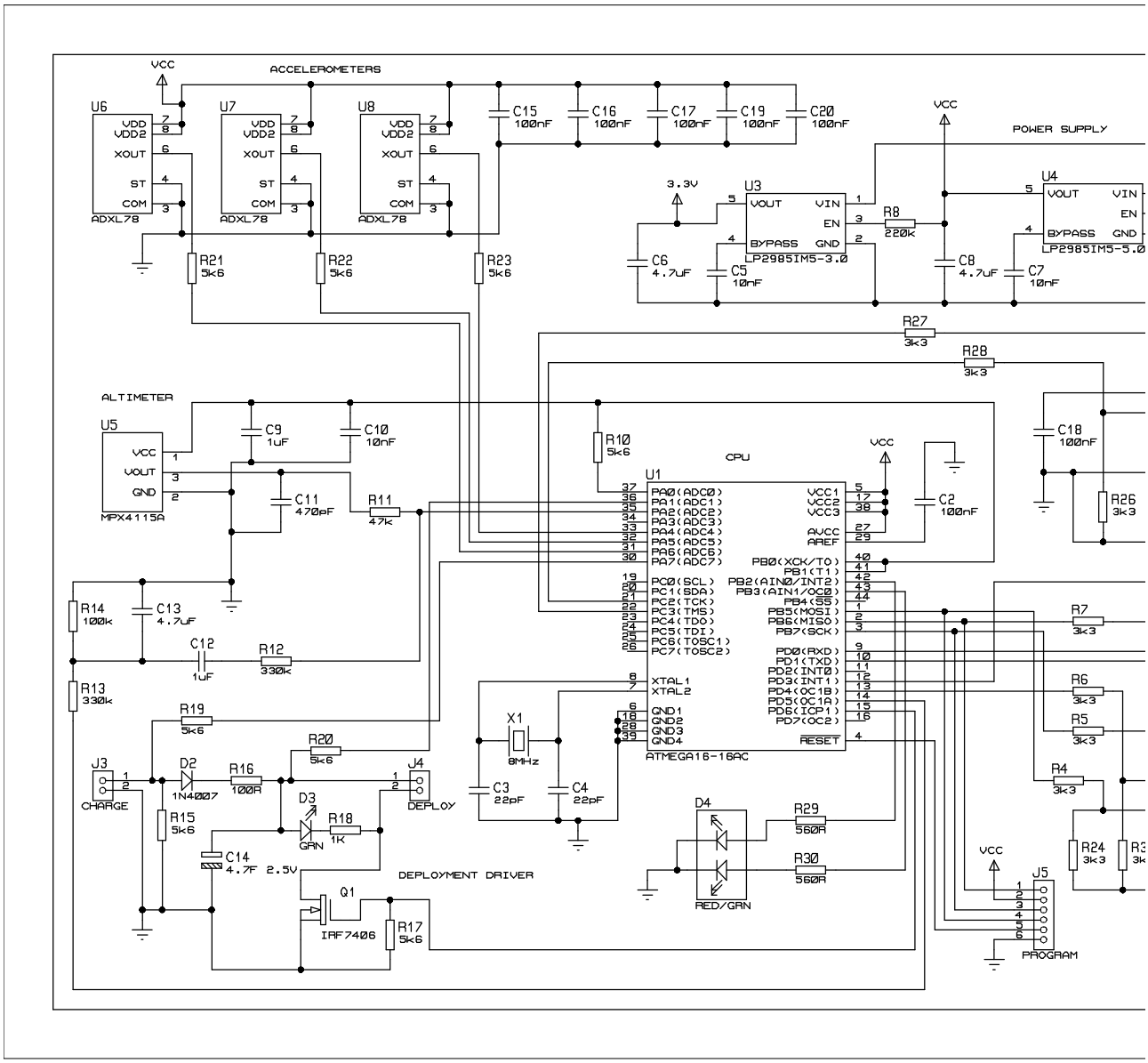


Accelerometers – inverted, then rolled on its side, then stood vertically.

The coming weeks should see this controller on board a real rocket launch. With the testing already performed, it should be a successful mission!

The next step is to use the data to control the deployment during flights, and to calculate and report the maximum altitude reached. It should be possible for deployment to be triggered by the rocket starting to fall. Maximum altitude can be easily calculated by comparing the altimeter reading before launch with the peak (after data smoothing) reached during the flight. This altitude could then be reported with the status request.

Schematic


ACCELEROMETERS
POWER SUPPLY
ALTIMETER
CPU
DEPLOYMENT DRIVER
U1
ATMEGA16-16AC
U3
LP2985IM5-3.0
U4
LP2985IM5-5.0
U5
MPX4115A
U6
U7
U8
ADXL78
D2
1N4007
D3
GRN
D4
RED/GRN
Q1
IRF7406
X1
8MHz
J3
CHARGE
J4
DEPLOY
J5
PROGRAM
VCC
R21 5k6
R22 5k6
R23 5k6
R27 3k3
R28 3k3
R10 5k6
R19 5k6
R20 5k6
R16 100R
R15 5k6
R17 5k6
R18 1K
R29 560R
R30 560R
R14 100k
R13 330k
R12 330k
R11 47k
R7 3k3
R6 3k3
R5 3k3
R4 3k3
R24 3k3
R26 3k3
R8 220k
C15 100nF
C16 100nF
C17 100nF
C19 100nF
C20 100nF
C18 100nF
C2 100nF
C6 4.7uF
C5 10nF
C8 4.7uF
C7 10nF
C9 1uF
C10 10nF
C11 470pF
C13 4.7uF
C12 1uF
C3 22pF
C4 22pF
C14 4.7F 2.5V