Uplink, Downlink, and How NOT to Vent a High-Altitude Balloon

James Flaten[1]
MN Space Grant / University of Minnesota – Twin Cities, Minneapolis, MN, 55455

Seth Frick[2] and Alex Ngure[3] and Hannah Weiher[4] and John Jackson[5] and Monique Hladun[6]
University of Minnesota – Twin Cities, Minneapolis, MN, 55455

We uplink commands to high-altitude balloon missions using DTMF encoding on ham radio frequencies, interpreted by an Arduino microcontroller with a DTMF decoder shield. The commands are then circulated between multiple payloads using an XBee radio network. We echo uplink transmissions back to the ground, as confirmation of receipt, and downlink other housekeeping data using a StratoSAT ZigBee module monitoring Arduino output pins. We have also developed a radio-controllable vent mechanism that attaches directly to the neck of a weather balloon to vent helium in flight, in an attempt to prolong time spent at altitude. This last project, controlled from the ground through the DTMF uplink and XBee network (with on-board autonomous back-up capabilities as well) has been problematic, but we will report on progress and results to date.

I. Introduction

AS a step beyond "basic ballooning" in which autonomous systems log sensor data and images on-board, and radios send gps information to the ground for tracking, we have been developing radio systems with which we can send commands to balloon payloads while in flight (AKA "uplink"), transmit arbitrary sensor data and/or housekeeping data (such as echos of uplink commands, to confirm receipt) to the ground (AKA "downlink"), and pass messages between payloads (AKA "network").

One particular payload that will benefit from these additional capabilities is our "Float" payload which can vent helium from the balloon and/or cut-down payloads from the balloon to terminate a flight. We would ultimately like to be able to control those functions from the ground, making decisions based on real-time tracking data, with autonomous algorithms serving as the back-up rather than the primary decision-makers.

Numerous U of MN students have worked on various aspects of this general project over the past several years involving multiple flights, both on MN-Space-Grant-sponsored summer-time ballooning teams and also members of one technical elective class for upper division students. The author list includes students who made some of the most useful contributions to various parts of this project to date.

II. DTMF Uplink

We selected DTMF (dual-tone multi-frequency) for encoding uplink commands, to be transmitted on ham radio frequencies. DTMF tones are familiar to us all – they are the tones you hear when using a touch-tone phone. Figure 1 shows a DTMF decoder shield for an Arduino microcontroller.[1] The audio output from a Yaesu VX-3 hand-held ham radio was plugged into the DTMF shield, which in turn was mated to an Arduino Uno or Mega. The shield decodes the DTMF signals it receives with an MT8870 DTMF receiver chip, displays them on its on-board 7-segment display, and passes them along to the Arduino. The Arduino also had a SD-card shield, to log information

[1] Associate Director of the MN Space Grant Consortium, Aerospace Engineering and Mechanics Department, 107 Akerman Hall, 110 Union Street SE, University of Minnesota, Minneapolis MN 55455.
[2] Graduate Student, Aerospace Engineering and Mechanics Department, University of Minnesota.
[3] Undergraduate student, Mechanical Engineering Department, University of Minnesota.
[4] Undergraduate student, Aerospace Engineering and Mechanics Department, University of Minnesota.
[5] Undergraduate student, Aerospace Engineering and Mechanics Department, University of Minnesota.
[6] Graduate Student, Aerospace Engineering and Mechanics Department, University of Minnesota.

such as DTMF tones received. A three-digit coding scheme was invented (see Appendix A) with codes appropriately separated (see Downlink section) to provide functionality for the Float payload (see Float section).

Figure 1. A DTMF decoder-only shield. DTMF tones fed in through the audio jack (upper right) are displayed on the 7-segment display and available to the Arduino microcontroller (below the shield). Our implementation also used a SD-card shield to log incoming DTMF commands plus an XBee radio shield to relay commands on the XBee network. Photo from Ref. 1.

III. SAT ZigBee Module Downlink

In addition to uplink, we wanted to establish a downlink radio connection (a) to echo uplink commands, as evidence that they were received properly, and (b) to send down status updates from on-board hardware, such as whether the vent was open or closed. Such a downlink channel could also be used to send experiment data to the ground in real time. We would have liked to implement downlink using DTMF as well, with the Arduino keying the handheld ham radio for transmissions, but our DTMF shield only does DTMF decoding, not tone generation (i.e. not DTMF encoding).

Hence we elected to implement downlink using ZigBee radio modules with the StratoSAT system we already owned. Since an Arduino cannot generate true analog output, we used pulse-width modulation (PWM) on Arduino digital pins and used a low-pass filter to approximate analog values between 0 and 5 volts which were then monitored by analog inputs on a ZigBee module and relayed to the ground through the StratoSAT command pod 900 MHz radio. This worked fairly well, though settling time issues and low resolution required us to keep legitimate 3-digit DTMF codes relatively far apart numerically so they could still be correctly distinguished upon echo through the downlink channels.

Each ZigBee module as two banks of 5 analog inputs and we flew one ZigBee in the DTMF payload and one in the Float payload, allowing us to monitor up to 10 analog voltages in each payload (though we never used that many). One ZigBee channel was used to echo 3-digit DTMF tones, converted into 0 to 5 volt analog values, to ensure they were received and interpreted properly. Other ZigBee channels were used to monitor the mechanical status of the vent itself with push-button switches (fully open, closed, intermediate), the status of the two nichrome cut-downs (ready, fired), the mechanical status of payload separation from the balloon, as evidenced by pull-pins (still attached, separated), and to report the cumulative open time for the vent. For more details about downlink channels, see Appendix A.

IV. XBee Network

We established a wireless radio network using XBee radios mounted on shields so that Arduino microcontrollers in different payloads can communicate with one another. This allows uplink commands sent by DTMF to be received by the DTMF payload then relayed on to their true destination – the Float payload, for example. It also allows payloads without ZigBee modules for StratoSAT downlink to send housekeeping data over

the XBee network to another payload that can downlink it. There are many on-line and print references[2] describing how to set up and program XBee networks with Arduino microcontrollers.

V. Float: Vent/Cut-down Payload

There are numerous challenges associated with venting helium (or hydrogen) from a balloon with the goal of achieving "float" (i.e. neutral buoyancy). The venting mechanism must reside in the neck of the balloon and provide a gas-tight seal when closed, but as much flow capacity as possible when open, so as to be as responsive as possible. Our design was for Kaymont 1200-gram and 1500-gram balloons with their relatively-narrow necks, at least compared to Howee balloons. The mechanism needs to be controllable, presumably using an actuator typically controlled by a microcontroller (using a timer, sensor input, and/or radio input). If one vents enough helium to achieve float, the mission no longer is automatically self-terminating so a termination device must also be included. In our case we used a nichrome burner "cut-down" to sever the line(s) between the balloon and the stack. However one of the hardest things to recover is the vent mechanism itself, since that is lodged in the neck of the balloon but, by definition, if you cut away from the balloon you don't get the neck back. (Other termination mechanisms are conceivable, such as trying to pop the balloon, dropping ballast to regain an adequate ascent rate to reach altitude and burst, or just venting enough helium to come down (albeit most slowly than by parachute), but these were not attempted.) Weight is another challenge, not only overall weight but also the fact that the vent mechanism is, by definition, located *above* the parachute in the stack and it might interfere with the parachute if it falls on it.

Several vent-mechanism-release schemes were designed, built, then tether-tested or flight-tested. We ultimately settled on inserting a piece of thin-wall pvc into the neck of the balloon then topping the vent mechanism with another tube which slid inside the tube in the balloon neck, using o-rings for the gas seal. The cut-down mechanism then had to sever string(s) being used to hold the inner tube inside the outer tube (which itself was inside the balloon neck). One design we tried was based on rigging – a stiff (curved) pin punctured through both tubes, holding them together. A string ran from the pin to the payloads below so when the cut-down dropped the payloads below they would fall several feet then yank out the pin, allowing the vent-mechanism to drop out of the balloon neck as well. The problem with this solution was that rotation of the stack with respect to the balloon tightened the string prematurely and ultimately pulled out the pin early during the ascent, venting the balloon and gently bringing the stack back to the ground with the balloon unpopped and partially deflated.

Figures 2 and 3 below show a more-recent incarnation of the Float payload, which attaches directly to the neck of the balloon but once released falls on tether lines to the bottom of the stack, so as not to remain above the parachute during the descent. The tube extending from the top of the Float payload goes inside (Fig. 2) or outside (Fig. 3) the tube in the neck of the balloon (which itself is not recovered). A custom 3-D printed frame forms the core of the mechanism. An Arduino controls a servo to open and close a cover at the base of the tube, venting helium into the false-bottom of the payload and then out into the atmosphere. The Arduino was controlled through the DTMF/XBee radio link but could also act autonomously if contact with the ground was lost. Ground testing revealed that several pounds of excess lift could be vented in under 5 minutes, consistent with our inadvertent experiences with low-altitude venting of balloons. However on our last test flight this vent was left open for over 15 minutes at altitude with no perceptible change in ascent velocity, suggesting that venting efficiency may depend strongly on ambient pressure (and possibly meaning that our vent mechanism solution is totally on the wrong track).

The full list of legitimate commands the DTMF can understand, most of which are relayed on to the Float payload, appears in Appendix A. Notice that some commands are contingent on sensor data such as "822: fire the cut-down only if the pull-pin has not been pulled (i.e. separation has not occurred)" whereas others are more direct like "888: fire cut-down no matter what (i.e. regardless of the pin status)." The Float payload has many autonomous features as well, including "fail-safe" timers which fire the cut-downs to terminate the mission if too much time has elapsed. These timers are to avoid the possibility of losing radio contact with the stack after it has reached a float condition so that it is no longer possible to command a termination from the ground. The Float code also has target float altitudes and target ascent rates in an "autopilot" routine with which the stack can attempt to achieve on its own float condition if not commanded from the ground. The autopilot can be enabled and disabled, and all its parameters can be modified in flight through the DTMF uplink, as conditions evolve.

Early versions of the Float code also included self-termination routines if no contact commands were received from the ground at least once every 30 minutes, though this proved to be problematic on one flight when the Arduinos were started before flight but then we had a launch delay and forgot to send commands so the Arduino initiated termination routines even before leaving the ground! As the title says, that is an example of "How NOT to vent a balloon!"

Figure 2. (a) CAD of vent mechanism within a payload shell, controlled by an Arduino (in blue, lower right). (b) The actual vent payload, with a downlink ZigBee radio module (blue, lower right), and a serious battery pack (on the left) to power the nichrome burner cut-downs. (c) Ground testing of the Float payload attached to the neck of a tethered balloon indoors.

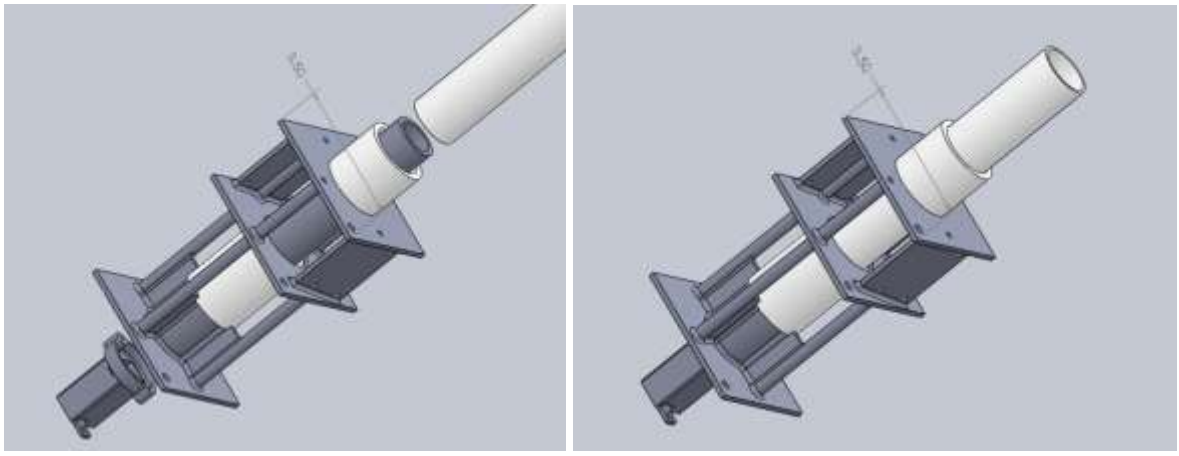

Figure 3. CAD model of the current 3-D printed Float core (a) in the "open" position (servo is on the lower left), balloon neck tubing is on the upper right, and (b) in the "closed" position. This version required the balloon neck tubing to extend well outside the balloon neck tube so it could be inserted into the frame and supported both on the inside and on the outside, for added strength. This change was made after breaking off the neck of the vent mechanism, which is particularly thin at the o-ring grooves, on two different occasions.

VI. Future Directions

We are currently rebuilding our DTMF payload using a different shield from Dossant[2] which has both DTMF decoding and encoding capabilities, so that we can do downlink by DTMF rather than using the parallel StratoSAT ZigBee radio module system (which was always overkill). The original DTMF payload seemed to hang up upon occasion, so our new DTMF payload is also equipped with an Arduino Nano which monitors the main Arduino and attempts to reset it if it stops operating. We call this our DTMF/Shepherd configuration. It has flown just once so far, successfully, so next we will add data-logging and XBee functionality to it so that it can be used to relay commands to other payloads such as the Float payload. We also plan to increase the autonomy of the Float payload so that if either the DTMF link or the XBee relay go down it can still function independently.

We remain puzzled by one recent flight experience in which a balloon continued directly to burst despite the vent being open for a long time, so we need to reconsider our assumptions about venting, consult with other balloonists with more experience venting balloons, and possibly do flight tests in which we vent periodically over the entire ascent to characterize venting efficiency as a function of altitude (i.e. as a function of ambient pressure). Another option might be to try to measure the flow rate through the vent directly at various altitudes. If venting this

way becomes intolerably inefficient at altitude we may need to come up with another hardware solution or else change our operating procedure dramatically, possibly starting to vent at much lower altitudes and/or using a second "excess-lift-only" balloon which can be cut away at altitude, reducing the amount of venting that needs to be done to achieve float. We are also in a good position to incorporate a ballasting mechanism which, coupled with a venting mechanism, would allow us to modify our excess lift either up or down at altitude.

VII. Conclusion

Although we are behind some other ballooning groups on implementing robust uplink, downlink, and venting capabilities, we are learning lots by trying to do these things ourselves rather than simply purchasing off-the-shelf solutions. We believe we have all the main pieces to a solution in place, although our vent mechanism, plus the control DTMF payload, are relatively heavy which limits the weight of experiments they might assist in getting extended time at float. Despite lots of work, this remains an interesting problem for us and will continue to occupy our attention for the foreseeable future.

Appendix

An excerpt (all comments, not executable code) from the current version of code used in the Float payload which includes a list of commands that can be passed along from the DTMF payload through the XBee network, illustrating the flexibility of the system. This also describes what data is on downlink channels from the DTMF payload and from the Float payload.

```
/*
DTMF Software v7b
Written by Seth Frick
8/21/2013
Last modified
10/3/2013

For DTMF system hardware configuration 5:
  -Arduino Mega 2560
  -Wireless SD shield with XBee module modified for Arduino Mega
  -Adafruit data logging shield with SD card, RTC, and u-blox MAX 6 GPS breakout board from HAB Supplies in prototyping area (switched to Adafruit Ultimate GPS breakout 6/17/13)
  -DTMF decoder shield

Unlike previous versions of the DTMF system, this version is designed to take binary-coded decimal values as input from the DTMF decoder shield. The BCD output nibble should be connected to pins 30-33, with pin 30 being LSB and 33 being MSB.

Upgrades new to version 6:
  -Zigbee downlink values logged directly for easier cross reference post-flight
  -Temperature sensor added near GPS module (logged every 5 seconds)
  -Battery voltage logged to SD card every 5 seconds
  -Second Zigbee downlink channel used to indicate which digit of the three-digit code was received most recently (hundreds place, tens place, or ones place)
  -DTMF code entry timeout feature added - if more than ten seconds passes between entry of successive digits, the DTMF code value is reset to zero and an additional tone or tones sent will initiate the beginning of a new code
  -GPS timestamps, number of satellites, latitude, and longitude are all logged
  -Logged altitudes are now in meters and ascent rates in m/s
  -Two ascent rates used - 5 sec average and 1 min average. 1 minute average used in evaluating fail-safe conditions; both rates logged.
  -GPS enable pin used to turn off GPS during setup(). Fixed issues with hanging code due to early GPS lock.

Upgrades new to version 7:
  -Internal temperature is downlinked on third Zigbee channel. Format is 10 x temp in Celsius.
```

-Float and cut-down "autopilot" routines are now set by default. In version 7a, the autopilot will open the float valve at a specified altitude, keep it open until a target ascent rate is achieved, stay at "float" for a specified amount of time, and then fire the cut-down. A cut-down altitude is also specified which will cause the flight to be terminated even if a float has not yet been achieved. Venting start altitude, target ascent rate, time at float, and cut-down altitude are all user-configurable in flight by sending the appropriate commands. In version 7b, the autopilot does not attempt to achieve a float, but instead runs a nominal testing routine of the float mechanism to produce data which will highlight the affect the float has on the ascent rate at various altitudes. At various altitude intervals, the float valve will be opened for a specified amount of time, and the cut-down will be autonomously fired at a specified altitude. The altitude interval, float valve open time, and cut-down altitude are all user-configurable in flight by sending appropriate commands. Autopilot can be disabled in preprocessor defines.

-Three additional Zigbee downlink channels added. The six downlink channels are used for the following:

1. Last DTMF code received
2. DTMF digits received (indicates partial codes) and code entry timeout monitor
3. Payload internal temperature (10 x temp in Celsius)
4. Altitude at which autopilot venting will begin (altitude in feet / 100) OR Altitude interval for float testing (altitude in feet / 100)
5. Targeted ascent rate for autopilot venting (feet per minute) OR Float valve open duration during each float test (seconds)
6. Altitude at which autopilot will fire cut-downs (altitude in feet / 100)

-External indicator LEDs added. One LED blinks every time through the loop, the other is solid on when in interrupt.

-Siren will automatically turn on above 10,000 feet on the way up if it is not already on.

-Autopilot functions (in both versions) can be disengaged and reengaged on command. When autopilot is disengaged, autopilot configuration parameters will still be accessible and can be changed by sending the appropriate commands. These values will be downlinked as usual. However, any action involving the float mechanism or cut-down burners will not be initiated by the autopilot while it is disengaged. Fail-safes will continue to operate.

DTMF command codes for Version 7:

-2XX: open float valve for XX seconds, then close automatically.
-302: close float valve indefinitely.
-382: open float valve indefinitely.
-4XX: reserved for competition payload.
-555: LED flash on Float payload.
-627: add one hour to fail-safe counters (both on DTMF payload and Float payload, assuming XBee link is up).
-678: hibernate (DTMF payload fail-safes inactive). Any new code received leaves hibernation.
-767: turn siren off.
-787: turn siren on.
-822: fire cut-down with pull-pin check.
-888: fire cut-down with no pull-pin check.
-940: subtract 5,000 feet from autopilot cut-down altitude.
-949: add 5,000 feet to autopilot cut-down altitude.
-990: reengage autopilot.
-999: disengage autopilot.

Autopilot configuration commands for Version 7a:

-900: subtract 5,000 feet from altitude at which autopilot opens float valve.
-909: add 5,000 feet to altitude at which autopilot opens float valve.
-920: subtract 100 ft/min from autopilot target ascent rate.
-929: add 100 ft/min to autopilot target ascent rate.
-960: subtract 5 minutes from autopilot float time.
-969: add 5 minutes to autopilot float time.

Autopilot configuration commands for Version 7b:

-900: set float valve open time for float tests to 30 seconds.
-909: set float valve open time for float tests to 1 minute.

-920: set float valve open time for float tests to 2 minutes.
-929: set float valve open time for float tests to 5 minutes.
-960: subtract 2,000 feet from altitude interval for float tests.
-969: add 2,000 feet to altitude interval for float tests.
*/

----Float Payload Downlink Info----

- Channel 1: cumulative time open for float mechanism. Output is 10 * minutes open, i.e. 350 = 3.5 minutes open.
- Channel 2: cut-down success (based on pull-pin status). Hundreds place is first burner, tens place is second burner, ones place is unused. Each burner digit will be 0-4, with 0 being no success, and 1-4 indicating success (and which "try" was successful), i.e. 200 = first burner caused pull-pin to be pulled on second try; 30 = first burner unsuccessful, second burner worked on third try.
- Channel 3: float push button status. 130 = "open" switch pressed, "closed" switch not pressed; 350 = "closed" switch pressed, "open" switch not pressed; 330 = neither switch pressed (float partially open); 150 = both switches pressed (indicating some error). NOTE: discrete time open commands (2xx) do not seem to produce correct downlink on this channel. Channels 1 and 4 can still be used for these commands. Channel 3 is useful for redundancy for indefinite open/close commands (382/302).
- Channel 4: servo position given directly from servo potentiometer wiper (bypasses microcontroller entirely). Large value (~500-700) indicates float valve is open; small value (~20-50) indicates float valve is closed.

Acknowledgments

I would like to thank the MN Space Grant for their longstanding support of my high-altitude ballooning program and for multiple sets of summer-time ballooning teams, as well as the students in my spring 2013 AEM 4495 “Space Vehicle Systems (with ballooning project)” class, for their work on this project. I would particularly like to thank Seth Frick for his long-time leadership on this project and also Lucas Kramer for his more-recent work on the DTMF/Shepherd payload.

References

[1]Schultz, Collin, “Too Many Hobbies” (blog), custom DTMF shield (decoder only) http://w2mh.wordpress.com/2010/06/03/dtmf-shield/ [cited 5 June 2014].

[2]Faludi, Robert, *Building Wireless Sensor Networks with ZigBee, XBee, Arduino, and Processing*, O’Reilly Media, 2011.

[3]Dossant, custom DTMF shield (decoder/encoder), http://dossant.com/projects/product/dtmf-shield-for-arduino/ [cited 5 June 2014].