ET301 GPS-UAV Development Platform

Part 3: Development suggestions

ET301 GPS-UAV Development Platform

This is the third part of a three part series of manuals for the ET301 GPS-UAV. The first part covers the hardware. The second part covers flight dynamics and control. This part covers development suggestions and programming examples

Development Suggestions

Safety

Before you start you should give a great deal of thought to safety. The following are some suggestions you should consider.

- Work your way up from simple to complex control incrementally. For example you could start with a truck, move up to rudder control of a sailplane, then elevator, then rudder plus elevator. Each stage should work flawlessly before proceeding to the next stage.
- The first safety feature that you will need is manual control, so that you can take over when (not if) there is some problem with automatic control. You should get the manual control working flawlessly before proceeding to automatic control. You should be able to control the plane yourself before proceeding to UAV development.
- Never let the plane fly out of your sight.
- Use a slowly flying aircraft, such as a paraplane or a sailplane. Consider using a Gentle Lady with a power pod or a “high-start” so that even if it hits anything, there is no spinning propeller out in front.
- If you use a power pod, engage the automatic control only after the engine runs out of fuel.
- Never fly high enough to be of concern to general aviation.
- Thoroughly test the controls on the ground before allowing them in the air.

This sort of activity is recommended only for experienced RC fliers who are members of the Academy of Model Aeronautics (AMA). Fly either at a club field or in an isolated area. Thoroughly check out the firmware before you launch. Before you fly the GPS-UAV, do a "walk-around-simulation" test flight on the ground by setting the controls for automatic and walking the plane around the flying field, pointing, turning, and moving it according to the response of the rudder and elevator, to make sure that everything is working all right.

The GPS-UAV can actually improve safety. For example, a "come-home" control mode can ensure that if the receiver loses the signal, the plane will return automatically to the launch point.

Getting started

You are probably anxious to get started. Before you dive in and start writing code, there are a few things you should think about:

- Goals – Think about what you hope to accomplish. Perhaps you just want to play around and write a little software, or perhaps you want some hands-on application of control theory. Consider how you will use the GPS-UAV. Do you want to accomplish fully autonomous control, or perhaps you only want something to make the plane more easily controlled on windy days? Do you want to implement the ideas outlined in the first two parts of this manual, or do you have your own ideas? Will you be satisfied achieving gentle, level turns, or do you want to be able to perform aerobatics? What is realistic for you to achieve will depend on your motivation and skill level. You might want to start with something simple, and work your way up to more ambitious goals. Unless you have an extremely large open field, it is not feasible to achieve automated take-offs and landings.
- Skills – For ambitious goals you will need a number of skills. First and foremost, you (or someone on your team) will need to be able to fly the aircraft that you select, because sooner or later during the development of your firmware, your automatic control will fail and you will have to revert to manual control. You should be familiar with both feedback control theory and flight principles as outlined in the second part of this manual. You will need firmware design, testing, and debugging skills.
- Selecting a vehicle or aircraft – The GPS-UAV was developed on a “Gentle Lady” sailplane with a power pod. You might want to consider something similar, or perhaps a “paraplane”. Slower is better because you will find that automatic control is easier to achieve at low speeds and that feedback control tends to become unstable as speed increases beyond a critical value. The GPS-UAV has two PWM output channels with corresponding input channels, intended for control of rudder and elevator of a high-dihedral aircraft. There is a third input channel that can be used to select control modes. If your aircraft has additional control servos that you want to use, you will have to connect them directly to the radio receive and control them manually. It is recommended that you mount the GPS-UAV inside your aircraft. You may very well want to start with a very simple, ground-based platform such as an RC truck, before moving up to an aircraft. It is a lot easier to debug a single axis control on the ground than it is to debug a two axis control in the air.
- Estimating parameters – You will need rough estimates of a few key parameters which define the flight dynamics of your aircraft, in order to design a feedback control with gains that are approximately right. Basically, you will need to estimate the gains that describe how the aircraft responds to the elevator and rudder, including how much and how quickly it banks in response to a turn. Nearly all of the parameters that you need can be computed from rough estimates of time constants, radii of curvatures of motion, and velocity, which you can get by careful observation of the response of your aircraft to manual control.
- Tools, compilers, programming and debugger – You will be spending a great deal of time and effort developing firmware, so give some thought to what firmware tools you will want to use. The GPS-UAV was developed entirely with assembly language. Perhaps you would prefer using a higher level language. Free assemblers

are available, while good high level language compilers will cost several hundred dollars. You will need a hardware interface for programming and debugging. Spark Fun's ICD2 was used to develop the GPS-UAV. A laptop computer is useful, but not absolutely necessary. Most of the development of the GPS-UAV was done with a desktop computer. Near the end of the development, a laptop computer became available and was used to make rapid changes in the firmware, especially feedback gains, between flights right at the flying field.

- Design – As described in the previous parts of the manual, there are several approaches to control design. Put some thought into designing a control that will achieve your goals. For example, the GPS-UAV was developed to control a "Gentle Lady" in level flight. If you want to go beyond that, you will need to go beyond the ideas described in the previous parts of the manual. In any case, before you start writing firmware, you should prepare a complete control design that will meet your goals.
- Incremental development – It is recommended that you follow a long series of small incremental development stages to simplify the development process. The development of the GPS-UAV proceeded through about a dozen stages, starting with single axis control of a truck and culminating with a full-featured control of a sailplane. Stages included various combinations of control features. For example, there was a stage in which there was automatic control of the rudder, and manual control of the elevator. Later, there was a stage in which there was automatic control of the elevator while the rudder was controlled manually.
- Debugging – It is recommended that you do most of the debugging of your firmware before launching your aircraft into flight. There are several ways that can be done. In the early stages of the development of some particular feature, you can simply use the ICD2 as a debugger to examine memory locations, with the GPS-UAV. The author spent a lot of time using this technique at a desktop computer, with the roof-mount antenna in a window to pick up a strong signal in a home with aluminum siding. Once the implementation appears to be doing the correct calculations, the next step is to rotate the board around pitch and yaw axses to see if the servos appear to be responding appropriately. Next, you might want to mount the board in your plane and do a "walk-around", watching the rudder and elevator to see if they respond correctly, at the same time moving the plane as if it were responding to the rudder and elevator. During the wintertime, the author did a fair amount of debugging from inside a moving car in an empty parking lot.
- Simulations – It is not necessary to perform simulations, but they may be useful if your goals are ambitious. During the initial stages of the development of the GPS-UAV, it was thought that it would not be necessary to do any simulations. But the initial design (compass instead of gyros) was doomed not to work very well, and eventually it was necessary to run some simulations to understand the control issues, which led to abandoning a compass in favor of gyros. In hindsight, using gyros instead of a compass at the outset would have obviated the need to perform simulations.

Engineering units and measurements

Early in your development process will you will come to grips with engineering units, gains, and conversion factors, especially if you perform simulations. The plane has several real world parameters and variables, including turning response to the rudder and elevator and equations of motion. The GPS reports position in terms of longitude and latitude. Velocity is reported by the GPS in units of kilometers per hour as well as knots. Accelerometers and gyros have gains relating acceleration and turning rate to a voltage. The A/D converter samples voltages and converts them to binary numbers. There are multiple ways of representing angles. The mathematical convention is to measure angles counter clockwise from due east. The GPS convention is to measure angles clockwise from due north.

You will need to make some design decisions regarding internal binary representations of various variables that are used in your calculations. Firmware development and simulations will go more smoothly if you select a consistent set of units and express all gains in terms of them. The most important gains (and perhaps most confusing), are those involving inputs and outputs. You should figure out the values of the following conversion factors for your aircraft and your choice of units, and keep them available during design, simulation, and setting other gains:

- Gyro gain – This is the conversion from physical rotation rate to an internal binary representation. During the development of the GPS-UAV, this gain was equal to approximately 0.17, when the rotation rate is expressed in units of radians per second, and when the binary representation is thought of as a fraction of full scale. In other words, when the rotation rate is 1 radian per second (57.3 degrees/second), the binary value is 0.17 of full scale.
- Acceleration gain – This is the conversion from acceleration to an internal binary representation. (For small values of pitch angle, what is actually being measured is the pitch angle.) During the development of the GPS-UAV, this gain was equal to approximately 0.2, when the pitch angle is expressed in units of radians, and when the binary representation is thought of as a fraction of full scale.
- GPS angle gain – This is the conversion from GPS course direction in GPS units to an internal binary representation. During the development of the GPS-UAV, this gain was set to be equal to ½ divided by pi.
- Servo gain – This is the conversion from an internal binary representation of a servo deflection to the reciprocal of the radius of curvature of the resulting motion. This gain depends on many things, including the mechanical transfer function from the servos to the rudder and elevator, as well as the dimensions of the aircraft. For the GPS-UAV prototype installation these gains were approximately ½ for both the rudder and the elevator, when the radius of curvature is measured in meters. In other words, when the internal binary servo signal reaches its maximum value, the radius of curvature of the motion is 2 meters.

If you express all other conversions and gains in terms of the basic ones, it becomes a simple matter to keep track of units.

State machine

You will very likely want to include a state machine in your control to handle transitions between various control states. For example, in the GPS-UAV prototype, there were several control states, including startup, waiting for GPS startup, self-nulling, manual, partially automatic, and fully automatic. Transitions between pairs of states depend on various conditions, such as whether or not the radio is on, for example. Once a state is established, it implies the values of several internal control flags, such as whether or not the GPS is used for navigation, for example.

It is recommended that you develop a state machine representation for your control. Decide what states you need, what conditions will trigger specific transitions between pairs of states, and what you want to happen within each state.

It is then a simple matter to convert your diagram into code and provide for implementation. This was done in the prototype as follows:

- A timer was used to generate an interrupt about once every 2 seconds. The value of 2 seconds was selected as being long enough to recognize whether or not the GPS was providing information needed for navigation, but not so long as for the control to feel unresponsive.
- The state machine code determined conditions required to make decisions, such as whether or not the radio or the GPS were working.
- For each state, there were sections of code for activities within that state, and for transitioning to other states.

For example, a high priority interrupt from timer0 can be configured to call the state machine code once every 2 seconds. Within the state machine, tasks that need to be carried out every few seconds are executed.

If there is an outstanding request to configure the GPS, the required commands are transmitted:

```
      btfsc        GPS_config
      call         set_gxx
```

Next, the number of valid pulses arriving on the "selin" channel is examined to determine whether or not the transmitter is on. The pulse rate is 50 pulses per second, so there should be 100 pulses over a 2 second period. For convenience, the threshold is set at 16 or more pulses. The count is then reset in preparation for the check on the next execution:

```
      movf         pulsesselin , W
      andlw        0xF0
      bz           radio_is_off
      clrf         pulsesselin
```

If the radio is on, set the radio_on status flag, turn on the corresponding LED, and enable the interrupt that performs pass-through manual control:

```
radio_is_on
      bsf          radio_on
      bcf          radio_led
      bsf          INTCON, RBIE
```

If the radio is off, reset the radio_on status flag, turn off the corresponding LED, take care of some variables that would normally be set through the radio, and turn off the interrupt that performs pass-through manual control, to avoid servo chattering in response to random noise:

```
radio_is_off
      bcf          radio_on
      bsf          radio_led
      movff        strngTrim , pwrudin
      movff        elevTrim , pwelein
      movlw        0xFF
      movwf        pwselin
      clrf         pulsesselin
      bcf          INTCON, RBIE
```

Next, look at the width of the “selin” pulse to determine which state is being requested. The pulse is very short for manual, is moderately long for a request for augmented mode, and is longest for a request for the circling mode:

```
      movf        pwselin , w
      addlw       -(auto_pulse)
      bc          above_auto
      bsf         man_req
      bcf         auto_req
      bcf         circle_req
      bra         det_state
above_auto
      movf        pwselin , w
      addlw       -(circle_pulse)
      bc          above_circle
      bsf         auto_req
      bcf         man_req
      bcf         circle_req
      bra         det_state
above_circle
      bsf         circle_req
      bcf         auto_req
      bcf         man_req
```

With the preliminaries out of the way, the actual state machine is then executed. The first step is to determine the present state (or mode) and execute the corresponding code:

```
det_state
      btfsc       startM
      bra         startS
      btfsc       calibrateM ; remove gyro, accelerometer offsets
      bra         calibrateS
      btfsc       acquiringM
      bra         acquiringS
      btfsc       manualM
      bra         manualS
      btfsc       autoM
      bra         autoS
      btfsc       returnM
      bra         returnS
      btfsc       circlingM
      bra         circlingS
```

For each state, there is code to enter that state as well as code to carry out the activities for that state, and logic to execute state transitions. For example, for a transition into the manual state, the following code is executed:

```
ent_manualS
      clrf        cntrl_flags
      clrf        waggle      ; turn off the rudder waggle
      bsf         manualM     ; set manual mode flag
      clrf        CCP1CON     ; turn off computed PWM control
      clrf        CCP2CON
```

```
        bcf         pwmout1      ; turn off the outputs
        bcf         pwmout2
        bsf         mode_led     ; turn off the mode LED
        bcf         GPS_steering; turn off GPS steering
```

Within the manual state, the following activities are carried out:

```
manualS
        btfss       radio_on     ; fly back home on loss of radio
        bra         ent_returnS
;       circle_req & nav_capable -> snap circle origin and enter
circling:
        btfss       circle_req
        bra         m_check_auto
        btfss       nav_capable
        bra         m_check_auto
        bsf         enable_focus
        bra         ent_circlingS
m_check_auto
;       auto_req -> autoS
        btfss       auto_req
        return      FAST
        bra         ent_autoS
```

Interrupts

Interrupts greatly simplify implementation of the control by providing for timely processing on an as-needed basis. In particular, the following interrupts provided by the 18F2520 are particularly useful:

- Serial communications – Communications between the 18F2520 and the GPS is via a serial transmitter/receiver. You may or may not wish to use interrupts to send messages from the CPU to the GPS, but you will certainly want to use interrupts to receive messages sent from the GPS to the CPU.
- Timers – There are several timers that can be used to generate interrupts. These are particularly useful for scheduling tasks that need to be performed on a regular basis, such as executing the state diagram, performing navigation, sampling the outputs of the gyros and the accelerometers, and computing control loops, for example.
- Interrupt-on-change – This is the best way to measure pulse widths of input signals from the radio to the CPU.

The 18F2520 supports two priority levels. The two levels were used during development of the GPS-UAV to provide a high priority for the pass-through manual control, ensuring that manual control would continue to work even if for some reason the rest of the firmware stopped working, a scenario which never materialized.

You will need to write an interrupt service routine, whose job it is to decide what condition(s) caused the interrupt to occur, to then execute whatever time-critical tasks are needed, and then to turn interrupts back on. For each type of interrupt there is a priority flag, an interrupt enable flag, and an interrupt flag. The priority flag selects the priority. The interrupt enable selects whether or not that type of interrupt is being used. When an interrupt is generated, the corresponding interrupt flag is automatically set so that the interrupt handler can identify the interrupt. The handler must reset the flag, otherwise the interrupt will be generated again as soon as interrupts are turned back on.

Interrupts are configured before they are turned on. The following are examples taken from a portion of the initialization code from GPS-UAV prototype:

```
bsf         RCON, IPEN        ; enable dual priorities
bcf         IPR1, RCIP        ; USART low priority
bsf         PIE1, RCIE        ; USART enable
bcf         PIR1, RCIF        ; clear any stale interrupt
bcf         IPR1, TMR2IP      ; PWM timer 2, low priority
bsf         PIE1, TMR2IE      ; PWM timer 2, enable
bcf         PIR1, TMR2IF      ; clear any stale interrupt
bsf         INTCON2, TMR0IP   ; timer0, high priority
bsf         INTCON, TMR0IE    ; timer0 enable
bcf         INTCON, TMR0IF    ; clear any stale interrupt
bsf         INTCON, INT0IE    ; interrupt 1, enable
bsf         INTCON2, INTEDG0  ; interrupt 1 on rising edge
bsf         INTCON3, INT1IP   ; interrupt 2, high priority
bsf         INTCON3, INT1IE   ; interrupt 2, enable
bcf         INTCON2, INTEDG1  ; interrupt 2 on falling edge
bsf         INTCON2, RBIP     ; port b, high priority
bcf         INTCON, RBIF      ; clear any stale interrupt
bcf         PIE1, TMR1IE      ; timer1, disable interrupt
movlw       B'10000000'       ; 1 = tmr1, 16 bit operation
                              ; 0 = not used
                              ; 0 = 1:1 prescaler
                              ; 0 = not used
                              ; 0 = T1 osc off
                              ; 0 = not used
                              ; 0 = internal clock
                              ; 0 = timer off (for now)
movwf       T1CON
bsf         INTCON, GIEL      ; enable low interrupts
bsf         INTCON, GIEH      ; enable high interrupts
bsf         T2CON , TMR2ON    ; turn on the PWM timer2
bsf         T1CON , TMR1ON    ; turn on timer1
bsf         RCSTA , CREN      ; turn on the GPS receiver
bsf         T0CON , TMR0ON    ; turn on timer0
```

The main program, the high priority service routine, and the low priority service routine are vectored from pre-assigned memory locations:

```
STARTUP CODE
      NOP
      goto        start             ;     reboot
      ORG         0x08
      goto        serv_inter_H      ;     high priority interrupt
      ORG 0x18
      goto        serv_inter_L      ;     low priority interrupt
      NOP
```

The main job of the interrupt handlers is to determine what needs to be done and to execute the appropriate code. For example, the high priority interrupt handler used to develop the GPS-UAV is:

```
serv_inter_H

      movff       TMR1L , tmr1Lsnap ;     snapshot the time
      movff       TMR1H , tmr1Hsnap

      btfsc       INTCON, RBIF      ; port B interrupt?
      rcall       serv_PORTB
      btfsc       INTRISE           ; hardwired interrupt rise?
      rcall       serv_rise
      btfsc       INTFALL           ; hardwired interrupt fall?
      rcall       serv_fall
      btfsc       INTCON, TMR0IF    ; timer zero?
      bra         serv_TMR0
      retfie      FAST
```

The "retfie FAST" instruction executes a fast return from interrupt, re-enables the interrupt at the same time, and restores the few registers that are saved when interrupt service routine is executed. It is also possible to explicitly re-enable interrupt, and to save and restore registers. For example, the following routines are employed to save and restore registers used by multiplication firmware, using a separate stack:

```
saveMult
      movff       PRODL , PREINC2
      movff       PRODH , PREINC2
      movff       ARG1L , PREINC2
      movff       ARG1H , PREINC2
      movff       ARG2L , PREINC2
      movff       RES3  , PREINC2
      movff       RES2 , PREINC2
      movff       RES1 , PREINC2
      movff       RES0 , PREINC2
      return

restoreMult
      movff       POSTDEC2, RES0
      movff       POSTDEC2, RES1
      movff       POSTDEC2, RES2
      movff       POSTDEC2, RES3
      movff       POSTDEC2, ARG2L
      movff       POSTDEC2, ARG1H
      movff       POSTDEC2, ARG1L
      movff       POSTDEC2, PRODH
      movff       POSTDEC2, PRODL
      return
```

The actual service routines should execute time-critical code as quickly and efficiently as possible, and then turn interrupts back on. A useful technique is to call a separate routine to complete the non time-critical code after the interrupts are back on as shown in the following example for a low-priority service routine:

```
serv_inter_L
      movff       STATUS, PREINC2   ; save status
      movwf       PREINC2           ; save WREG

      btfsc       PIR1, TMR2IF      ; timer 2 interrupt?
      call        serv_PWM          ; service the PWM interrupt
      btfsc       PIR1, RCIF        ; GPS interrupt?
      call        serv_GPS          ; service the GPS interrupt

      bsf         INTCON, GIEL      ; re-enable

      btfsc       GPS_req           ; pending GPS request?
      rcall       call_cmplt_GPS
      btfsc       ELE_req           ; compute elevator?
      rcall       call_elev_cntrl
      btfsc       RUD_req           ; compute rudder?
      rcall       call_rudd_cntrl

      movf        POSTDEC2, W       ; restore WREG
      movff       POSTDEC2, STATUS  ; restore status bits

      return
```

It was found that with the 4X clocking boost of the crystal frequency multiplier producing an effective clock frequency of 16 megahertz, the 18F2520 had plenty of CPU power to complete all service routines in plenty of time. Still, it is a good idea to ensure that your firmware does not unintentionally cause multiple instances of the same routine to be running at the same time as a consequence of one of them not completing in time. For example, suppose that you decide to do extensive computations in your navigation software, which you would probably want to execute once per second as new GPS data becomes available. To make sure that you do not accidentally generate stack overflow if the routine takes more than 1 second to complete, you might want to use a flag to block the execution of one pass of the firmware until the previous pass is complete. You could do that with the following segments of code. In the routine that calls the navigation routine, set a busy flag before the call, and clear it afterwards:

```
      bsf         nav_busy    ;     block re-entrant call
      call        navigate
      bcf         nav_busy    ;     re-enable call
```

Only execute the routine that calls navigation when the flag is clear:

```
call_one_sec

      btfss       nav_busy
      call        one_sec_tasks     ; includes a call to navigate
      return
```

Manual control

You will likely want to implement a manual control option because, sooner or later, one of the versions of your firmware will produce unstable control and you will want to recover manually. During the design of the GPS-UAV it was decided to implement manual control in software rather than hardware. This can be done by simply echoing the rudder and elevator inputs to the outputs. Use the interrupt on change-of-value, and simply copy the inputs to the outputs. For example:

```
serv_PORTB
      bcf         INTCON, RBIF      ; clear the interrupt
      btfss       INTCON, RBIE
      return

      movff       PORTB , PORTB_snap; snapshot the B port

      btfss       pwmin1            ; echo the rudder
      bcf         pwmout1
      btfsc       pwmin1
      bsf         pwmout1

      btfss       pwmin2            ; echo the elevator
      bcf         pwmout2
      btfsc       pwmin2
      bsf         pwmout2

      movf        PORTB_snap, W     ; save port B
      xorwf       PORTB_old , W     ; look for changes
      movwf       PORTB_xor
      movff       PORTB_snap, PORTB_old

      btfsc       pwmin1_xor        ; rudder channel changed
      rcall       pwmin1_time
      btfsc       pwmin2_xor        ; elevator channel changed
      rcall       pwmin2_time
      return
```

The inputs can be always copied to the outputs, even during automatic control, because the way the 18F2520 is architected, turning on the PWM control will override digital outputs on the PWM pins.

Measuring pulse width

The code in the previous section also implements measurement of the widths of the rudder and elevator pulses coming from the radio. The time is recorded on every interrupt, and is used on the falling edges of the pulses to compute pulse widths. The following is the code for the rudder control pulses. The code for the elevator is similar:

```
pwmin1_time
      btfss       pwmin1
      bra         pwmin1_fall
      movff       tmr1Lsnap , tmrudinL
      movff       tmr1Hsnap , tmrudinH
      return

pwmin1_fall
      movf        tmrudinL , W
      subwf       tmr1Lsnap , W
      movwf       tmrudinL
```

```
        movf        tmrudinH , W
        subwfb      tmr1Hsnap , W
        movwf       tmrudinH
        movf        tmrudinH , W ; get the high order nibble
        andlw       0xF0
        movwf       tmrnible
        swapf       tmrnible
        movlw       0x01        ; pulse from 1 msec to 2 msec ?
        subwf       tmrnible , W
        bnz         rudinsat
        incf        pulsesrudin ; count valid pulses

        movlw       0x0F        ; divide the 2 byte value by 16
        andwf       tmrudinH , F
        swapf       WREG , W
        andwf       tmrudinL , W
        iorwf       tmrudinH , W
        swapf       WREG , W
        movwf       pwrudin     ; pulse width rudder input
        return

rudinsat
        movlw       0x00
        subwf       tmrnible , W
        bz          rudinmin
        movlw       0x02
        subwf       tmrnible , W
        bz          rudinmax
        return

rudinmin
        clrf        pwrudin
        incf        pulsesrudin
        return

rudinmax
        movlw       0xFF
        movwf       pwrudin
        incf        pulsesrudin
        return
```

The saturation calculation is performed because the range of pulse widths is close to, but not exactly equal to, the exact range of an 8 bit value. Rather than use 16 bits to sometimes pick up a 9th bit, a saturation calculation is used to map the measured pulse width to 8 bits.

Select input

The GPS-UAV has three input channels. One of them is for the rudder, one for the elevator, and one for control mode selection. The rudder and elevator inputs connect to the B port, and generate an interrupt on both rising and falling edges of the pulses. Because all other interrupt inputs on the B port are dedicated to other functions, the third input channel is connected to two interrupt pins. Because the interrupt pins generate an interrupt on either rising or falling edge, but not both, two pins are connected in parallel. This leads to a slightly different method for processing the third input. The code that was used in the prototype firmware to process the select input follows. The high priority interrupt routine calls serv_rise for a rising edge, and serv_fall for a falling edge. The variables INTRISE and INTFALL are defined to be the two interrupt input pins. The processing is very similar to that of the rudder and elevator input processing:

```
serv_rise

      bcf                         INTRISE
      movff        tmr1Lsnap , tmselinL
      movff        tmr1Hsnap , tmselinH
      return

serv_fall
      bcf                         INTFALL

      movf         tmselinL , W
      subwf        tmr1Lsnap , W
      movwf        tmselinL

      movf         tmselinH , W
      subwfb       tmr1Hsnap , W
      movwf        tmselinH

      movf         tmselinH , W ; get the high order nibble
      andlw        0xF0
      movwf        tmrnible
      swapf        tmrnible

      movlw        0x01        ; valid pulse from 1 msec to 2 msec
      subwf        tmrnible , W
      bnz          selinsat

      incf         pulsesselin ; count valid pulses

      movlw        0x0F        ; divide the 2 byte value by 16
      andwf        tmselinH , F
      swapf        WREG , W
      andwf        tmselinL , W
      iorwf        tmselinH , W
      swapf        WREG , W
      movwf        pwselin     ; select input third pulse width

      return

selinsat
      movlw        0x00
      subwf        tmrnible , W
      bz           selinmin
      movlw        0x02
      subwf        tmrnible , W
      bz           selinmax
      return
```

```
selinmin
      clrf        pwselin
      incf        pulsesselin
      return

selinmax
      movlw       0xFF
      movwf       pwselin
      incf        pulsesselin
      return
```

Tasking

There are several tasks that need to be executed on a regular basis, such as sampling the gyros and accelerometers, filtering, and updating servo pulse widths. This can be done by using a timer to generate an interrupt on a regular basis and to select a task from a list of tasks on each interrupt. A convenient place to start is the timer that is used to control pulse width modulation. It generates an interrupt approximately once every millisecond. The tasks need to be executed once approximately every 23 milliseconds, so it is convenient to create that many time slots for tasks.

When timer 2 generates an interrupt, the following code decrements PWM_count from PWM_skip to zero, continually repeating.

```
serv_PWM

      bcf         PIR1, TMR2IF      ; turn off the interrupt flag
      dcfsnz      PWM_count , F     ; decrement the pulse count
      movff       PWM_skip , PWM_count    ; reload the counter
```

PWM_count is then used to implement a computed go-to. First, the low byte of a computed address is initialized with the low byte of the start of the task table:

```
      movlw       low PWM_goto_table ; base of the table
      movwf       PWM_goto_L
```

An offset into the table is computed as two times PWM_count, because each entry in the task table takes two bytes. The offset is then added to the low byte of the computed address:

```
      movf        PWM_count, W      ;     offset into the table
      decf        WREG , W
      addwf       WREG , W
      addwf       PWM_goto_L , F
```

The high byte of the start of the task table is then loaded into the high byte of the program counter latch. This does not actually change the program counter yet:

```
      movlw       high PWM_goto_table
      movwf       PCLATH
```

The possibility that a carry was generated when the offset was added to the base address must be accounted for by adding the carry to the high byte of the computed address:

```
clrf        WREG
addwfc      PCLATH , F
```

Finally, the computed go-to is executed by loading the low byte of the computed address into the program counter, which causes the high byte stored in the program counter latch to be loaded at the same time:

```
movf        PWM_goto_L , W
movwf       PCL
```

As a result, entries from the following table of instructions are executed one at a time in reverse order, starting from task_22 backward to task_00.

```
PWM_goto_table
      bra                    task_00
      bra                    task_01
      bra                    task_02
      bra                    task_03
      bra                    task_04
      bra                    task_05
      bra                    task_06
      bra                    task_07
      bra                    task_08
      bra                    task_09
      bra                    task_10
      bra                    task_11
      bra                    task_12
      bra                    task_13
      bra                    task_14
      bra                    task_15
      bra                    task_16
      bra                    task_17
      bra                    task_18
      bra                    task_19
      bra                    task_20
      bra                    task_21
      bra                    task_22
```

The above structure makes it very convenient to assign tasks to the various slots simply by placing calls and go-to's at the task addresses. The following task list was used in the prototype to take samples, perform control calculations, and generate servo pulses:

```
task_22           ; spare
      return

task_21           ; select voltage reference
      bra         sel_vref

task_20           ; read voltage reference, select yaccel
      call        read_vref
      bra         sel_yaccel

task_19           ; read yaccel, select xaccel
      call        read_yaccel
```

```
      bra         sel_xaccel

task_18           ; read xaccel , select pitch gyro
      call        read_xaccel
      bra         sel_pitch

task_17           ; read pitch gyro, select yaw gyro
      call        read_pitch
      bra         sel_yaw

task_16           ; read yaw gyro, select yaw gyro again
      call        read_yaw1
      bra         sel_yaw

task_15           ; read yaw gyro again
      bra         read_yaw2

task_14           ; filter the mixing
      bra         filter_mix

task_13           ; spare
      return

task_12           ; spare
      return

task_11           ; compute elevator servo
      bra         compute_elevator

task_10           ; filter the rudder pulse width
      bra         filter_pwrud

task_09           ; compute rudder servo
      bra         compute_rudder

task_08           ; scale the gains according to the third input
      bra         scale_gains

task_07           ; rudder 1 msec pulse
      bra         PWM1_full_pulse

task_06           ; rudder partial (0-1 msec) pulse
      bra         PWM1_pulse

task_05           ; turn off the pulses
      bra         PWM_clear

task_04           ; spare
      return

task_03           ; elevator 1 msec pulse
      bra         PWM2_full_pulse
```

```
task_02          ; elevator partial (0-1 msec) pulse
      bra        PWM2_pulse

task_01          ; turn off the pulses
      bra        PWM_clear

task_00          ; spare
      return
```

GPS interface

You will probably want to refer to the data sheet and the NMEA reference manual for the ET-301 that are available on the Spark Fun website.

Although the GPS interface is conceptually rather simple, in the prototype firmware, the GPS interface module was the largest module. That was in part the result of a decision to use the NMEA standard interface to the ET-301 rather than its binary interface. Much of the code was required to parse the ASCII text into binary values. Even so, in retrospect, the decision to use the NMEA interface was a good one because:

- Using the NMEA interface simplified the debugging of the GPS interface. It was possible to understand what was coming out of the ET-301 by simply connecting a hyperterminal to an ET-301 evaluation kit available from SFE.
- Using the NMEA interface made most of the firmware portable. In fact, the earliest prototype of the GPS-UAV used a different GPS receiver. Porting the firmware to the ET-301 was simplified because the NMEA interface was being used.

The choice is up to you whether to use the NMEA interface to the ET-301 or to use the binary interface. The NMEA interface will be easier to debug, but the binary interface will not require as much firmware. Most of what you might want is available through the NMEA interface. The only information that you might want to use that is available exclusively through the binary interface is vertical velocity, but that was not used in the GPS-UAV prototype firmware. The information that was used was extracted from the GGA and VTG messages in the NMEA interface:

- GGA – Latitude, longitude, position fix indicator and satellites used information was used by the GPS-UAV prototype. Also available in GGA, but not used in the prototype, is altitude.
- VTG – Measured heading was used by the GPS-UAV prototype. Also available, but not used in the prototype, is speed over ground.

The ET-301 boots up with its communications baud rate set to 4800, which is what was used in the GPS-UAV prototype. 4800 was a convenient baud rate, high enough to transfer all needed information in a timely fashion, and slow enough to allow for quite a bit of processing between characters.

The default messages at boot-up are probably not the ones you want. It is recommended you explicitly turn each message type on or off, depending on whether or not you intend to use the information. There is no point in having the ET-301 interrupt the firmware with messages that are not going to be used.

The CPU will boot much sooner than the ET-301, so you will need to wait to send configuration commands to the ET-301, otherwise the ET-301 will ignore the commands. This was handled in the prototype by waiting until midway through the startup process to send configuration messages to the ET-301.

It is possible for the USART interface to the ET-301 to generate errors. Although you would think that careful design would preclude such errors, they were encountered from time to time during the development of the prototype firmware. Therefore, you should address errors. For example, framing and receiver errors must be cleared to avoid “hanging” the CPU. In particular, an uncleared overrun error will regenerate an interrupt as soon as interrupts are re-enabled, effectively locking up the CPU. You will probably want to include something like the following:

```
serv_GPS

      bcf         PIR1, RCIF  ; reset the interrupt flag
      btfsc       RCSTA , FERR; check for a framing error
      rcall       GPS_ferr
      btfsc       RCSTA , OERR; check for an overrun error
      bra         GPS_oerr
                              ; otherwise, normal reception
      .
      .
      .
GPS_ferr          ; Framing error
                  ; clear by reading the receiver register.

      movff       RCREG, GPS_char
      btfss       RCSTA , OERR
      return

GPS_oerr          ; Overrun error
                  ; clear by toggling the receiver enable.
      bcf         RCSTA , CREN
      nop
      bsf         RCSTA , CREN
      return
```

The bulk of the rest of the GPS interface is parsing the ET-301 messages and converting them into binary values. Refer to the source code for the test firmware for examples. In those examples, conversion from absolute longitude and latitude to Cartesian coordinates relative to a starting point was based on the assumption of operation in the western hemisphere at a latitude of around 42 degrees north. If you are operating in the eastern hemisphere or at a latitude much different than 42 degrees north, you will want to rewrite the conversion code.

Also, the test firmware assumed that operation would not be any further from the starting point than 5 kilometers. That should be more than enough, because you should not be flying your aircraft out of sight.

Sampling the gyros and accelerometers

The gyros and accelerometers are read through the A/D converter. As part of your design you will need to think about sampling rates, noise, bandwidth, filtering, and reference voltages. You will also need to think about controlling the A/D converter.

The GPS-UAV uses Spark Fun breakout boards for the gyros and accelerometers that include capacitors on the output pins for simple lowpass filters for bandwidth/noise control. The accelerometers have filters with a corner frequency of around 500 Hz (higher than what is actually needed) and the gyros have filters with a corner frequency of around 40 Hz.

A sampling rate of around 50 samples per second for each analog input channel is more than adequate for good transient response and noise filtering, and fits well with both the CPU processing power and the servo pulse rate. There is no need to sample any slower than that, but depending on what digital filters you incorporate into your control, you may want to sample at a higher rate and use a decimation filter to reduce noise. It is best not to pass much noise through to the servos to avoid a high battery drain. The prototype firmware worked quite smoothly with 50 samples per second.

You will undoubtedly want to do some digital filtering on the gyro and accelerometer signals as part of your control. The prototype used simple digital lowpass filters with time constants on the order of a few seconds as part of a "washout" filtering scheme. As a side effect, the filters greatly reduced noise from the gyros and accelerometers.

You will also need to think about reference voltages. The outputs of the accelerometers are "ratiometric" (meaning they are proportional to the power supply), while the outputs of the gyros are "non-ratiometric" (meaning as long as the power supply is within limits, the outputs are independent of the supply. The gyros provide a constant 2.5 volt reference. Here is the approach used in the prototype to achieve control that is independent of supply voltage:

- The A/D converter used an internal reference that is proportional to the supply voltage. Therefore, A/D converted values a proportional to the ratio of the sample voltage divided by the supply voltage.
- The 2.5 volt reference was sampled by the A/D converter. This can be used to figure out what the supply voltage is.
- Samples of the accelerometer signals do not have to be corrected for variations in supply voltage, because both the accelerometer signals and the A/D converter are "ratiometric".
- Samples of the gyro signals were corrected for variations in the supply voltage.

Taking the actual samples is rather simple using the tasking technique described in one of the previous sections. To take a sample the sample and hold circuit internal to the A/D has to be first connected to the appropriate input channel long enough to charge an internal capacitor. The actual time required is not very long, on the order of several 10s of microseconds. The approach used in the prototype was to select the channel in one task slot and then to read it in the next, which allowed approximately 1 millisecond (1000 microseconds), which is more than long enough for the sample and hold capacitor to settle out.

The most efficient way to service the A/D conversion itself is to use A/D interrupts, but that is more complicated than is worth the trouble. The following approach is suggested:

- Select an A/D channel input at the end of the processing for one of the 1 millisecond task time slots.
- At the beginning of the next task time slot, command the A/D to convert.
- The conversion will take a few microseconds, so it is efficient to perform some part of the control that will take a few microseconds before checking to see if the conversion is complete.
- Wait for the conversion to complete, and then fetch the values.

For example, the pitch gyro is selected in task 18 and read in task 17. (Remember, the tasks are executed in reverse order.)

```
task_18           ; read xaccel , select pitch gyro
      call        read_xaccel
      bra         sel_pitch

task_17           ; read pitch gyro, select yaw gyro
      call        read_pitch
      bra         sel_yaw
```

The pitch gyro is selected by the following few lines of code:

```
sel_pitch
      movlw       pitchAD ; pitch
      movwf       ADCON0
      return
```

Reading the pitch gyro proceeds as follows. First, the conversion control bit is set to start the conversion:

```
read_pitch
      bsf         ADCON0,GO
```

The conversion itself may take a few microseconds, so it would be efficient to perform a few computations. In this case, a portion of a low pass filter for the pitch signal is executed.

```
      movf        pitchH , w
      subwf       pitchL , f

      movf        pitchU , w
      subwfb      pitchH , f

      clrf        WREG
      subwfb      pitchU , f
```

Now, we just wait for the conversion to complete by polling the status bit:

```
cnvrt_pitch
      btfsc       ADCON0,GO
      bra         cnvrt_pitch
```

When the conversion is complete, continue on with the computation. Read the A/D registers and continue the filtering computation:

```
      movf        ADRESL , w
      addwf       pitchL , f

      movf        ADRESH , w
      addwfc      pitchH , f

      clrf        WREG
      addwfc      pitchU , f
```

This is a convenient place to service the request to snapshot the pitch gyro offset during the self-calibration process, to be used later on to remove the gyro offset.

```
btfss       calibpitch
return
bcf         calibpitch
movff       pitchU , pitch0U
movff       pitchH , pitch0H
return
```

Although the accelerometers are immune to drift due to temperature variation, the gyros experience some drift with temperature. A temperature signal is available if you want to perform temperature compensation in software, but it was not used in the prototype. The operational technique that was used with the prototype was to simply let the GPS-UAV adjust to the ambient temperature. Prior to each flight the GPS-UAV was reset so that the self-zeroing features would balance out the gyro offsets, which then did not change very much during a single flight.

Rudder control

The rudder control described in the previous manual in this series can be implemented in firmware as described in this section.

First, clear the control flag that was used to generate the request to compute the rudder deflection:

```
rudd_cntrl
      bcf         RUD_req
```

Retrieve the filtered, unsigned yaw rate gyro value:

```
movff       yawU , ARG1H
movff       yawH , ARG1L
```

Subtract the baseline yaw offset that was recorded during the power up self-calibration process:

```
movf        yaw0H , w
subwf       ARG1L , f
movf        yaw0U , w
subwfb      ARG1H , f
```

Subtract the measured reference voltage deviation from the gyro in order to compensate for the fact that the A/D conversion is based on the power supply and the gyro output is based on a 2.5 constant reference voltage. This approximately compensates for variations in battery voltage. The deviation itself is the measured reference voltage minus the value recorded during power up, so the actual calculation is to add the baseline and subtract the present value. Adding the baseline is accomplished by:

```
movf        ref0H , w
addwf       ARG1L , f
movf        ref0U , w
addwfc      ARG1H , f
```

Subtract the reference voltage:

```
    movf        refH , w
    subwf       ARG1L , f
    movf        refU , w
    subwfb      ARG1H , f
```

At this point, ARG1 contains the signed yaw rate. Next, multiply it by the appropriate feedback gain:

```
    movff       yawgain , ARG2L
    call        MULS2X1
```

The result is the first of several terms in the total for the signed rudder deflection. Move it into the three-byte rudder deflection accumulator:

```
    movff       RES0 , ruddL
    movff       RES1 , ruddH
    movff       RES2 , ruddU
```

If the state machine requests GPS steering, compute the GPS feedback term:

```
    btfss       GPS_steering
    bra         gps_is_off

    movf        actualDir, W  ; actual direction in W
    subwf       desiredDir, W ; subtracts actual F-W->W
    movwf       errorDir      ; save the signed difference
    smult       strngGain , errorDir ; GPS feedback gain

;   add into the rudder accumulator
    movf        PRODL , w
    addwf       ruddL , f
    movf        PRODH , w
    addwfc      ruddH , f

;   extend the sign of the 2 byte result into 3 bytes
    movlw       0x0
    btfsc       PRODH , 7
    movlw       0xFF
    addwfc      ruddU , f

    bra         gps_is_on

gps_is_off
```

Next, add the augmentation deflection to account for the tendency of the yaw gyro to cancel the manual turn commands. The theory is explained in the previous manual in this series. The term is equal to a gain multiplied by the rudder signal from the radio minus the rudder trim. This can be computed in terms of pulse widths:

```
    clrf        WREG
    movwf       ARG1H
```

```
        movff       pwrudfilH , ARG1L

        movf        strngTrim , w
        subwf       ARG1L , f
        clrf        WREG
        subwfb      ARG1H , f

        movff       auggain , ARG2L
        call        MULS2X1

        movf        RES0 , w
        addwf       ruddL , f
        movf        RES1 , w
        addwfc      ruddH , f
        movf        RES2 , w
        addwfc      ruddU , f

gps_is_on
```

Convert from signed deflection into a PWM time period offset:

```
        movlw       0x08
        addwf       ruddH , F
        movlw       0x00
        addwfc      ruddU , F
```

Note that the total PWM time period in this implementation is the sum of two portions, a fixed portion of approximately 1 millisecond, and a variable portion of from 0 to approximately 1 millisecond. At this point, we have the variable portion of the time period.

Check upper byte for overflow:

```
        movf        ruddU , w
        andlw       0xFF
        bz                      ruddU_normal
        bn                      rudd_min
        movlw       0xFF
        bra                     rudd_trm_adjst

ruddU_normal
```

Check high byte for overflow:

```
        movf        ruddH, W    ;     retrieve MSbyte of result
        andlw       0xF0        ;     check for overflow
        bz          rudd_normal ;     no "clamping" needed
        bn          rudd_min    ;     clamp to minimum for
        movlw       0xFF
        bra         rudd_trm_adjst

rudd_min
        clrf        WREG
        bra         rudd_trm_adjst
```

```
rudd_normal
```

Normal case:

```
movlw       0xF0        ;     mask for upper 4 bits
andwf       ruddL, W; move the upper 4 bits of ruddL into W
iorwf       ruddH, W; "or" the lower 4 bits of ruddH into W
```

At this point the result is in W, except the nibbles are swapped.

```
swapf       WREG ; Swap the nibbles for the 4 bit shift.
```

Here is where the manual trim from the radio comes in. Add the trim, test for overflow:

```
rudd_trm_adjst
```

First, convert rudder control from a time period back to a deflection. The astute reader may notice that some of the previous code and some of the following could be combined and simplified. What is here came out this way for historical reasons:

```
btg                         WREG , 7
```

Fetch the manually commanded rudder pulse width and convert it to a deflection:

```
movff       pwrudin , strngTrimTemp
btg         strngTrimTemp , 7
```

Add the trim deflection to the commanded deflection to get the total deflection:

```
addwf       strngTrimTemp , W
bov         ruddtrim_ov ;     overflow
```

Convert from a deflection to a time period:

```
btg         WREG , 7
```

Waggle the rudder during startup:

```
addwf       waggle , w
```

Move the final result to the register used to set the duty cycle for the rudder PWM timer:

```
movwf       PWM1_dc
return
```

Overflow handling:

```
ruddtrim_ov
      btfss       WREG , 7
      bra                         rudd_clamp_min
      bra                         rudd_clamp_max
rudd_clamp_max
      movlw       0xFF
      movwf       PWM1_dc
```

```
      return
rudd_clamp_min
      movlw       0x00
      movwf       PWM1_dc
      return
```

Elevator control

Elevator control is very similar to the rudder control with a few exceptions:

- The accelerometer is used instead of the GPS.
- The elevator does not “waggle” during power up.
- The elevator incorporates both mixing and gyro decoupling to account for effects due to banking.

The mixing and gyro decoupling described in the second part of the series of manuals can be implemented as follows:

Start with the unfiltered yaw rate gyro signal:

```
movff       yawunfH , ARG1H
movff       yawunfL , ARG1L
```

Subtract the baseline yaw rate:

```
movf        yaw0H , w
subwf       ARG1L , f
movf        yaw0U , w
subwfb      ARG1H , f
```

Add the baseline reference voltage:

```
movf        ref0H , w
addwf       ARG1L , f
movf        ref0U , w
addwfc      ARG1H , f
```

Subtract the most recent measurement of the reference voltage:

```
movf        refH , w
subwf       ARG1L , f
movf        refU , w
subwfb      ARG1H , f
```

At this point the offset has been removed from the unfiltered yaw rate, and it has been adjusted for drift in the supply voltage. Save the adjusted value:

```
movff       ARG1L , yawadjL
movff       ARG1H , yawadjH
```

Retrieve the filtered yaw rate:

```
movff       yawfilU , ARG2H
movff       yawfilH , ARG2L
```

The blending of yaw rate into pitch rate is approximately proportional to the filtered yaw signal times the unfiltered yaw signal, because the bank angle is proportional to the filtered yaw signal:

```
    call        MULS2X2
```

Filter the result once, because the actual mixing error also gets filtered:

```
    movff       yawsqU , outfilU
    movff       yawsqH , outfilH
    movff       yawsqL , outfilL
    movff       RES1 , infilL
    movff       RES2 , infilH
    movlw       .1
    movwf       taufil
    call        filter
    movff       outfilL , yawsqL
    movff       outfilH , yawsqH
    movff       outfilU , yawsqU
```

At this point we have a term that is proportional to the error. Next we must account for the gain of the error by multiplying by the gyro mix gain:

```
    movff       yawsqU , ARG1H
    movff       yawsqH , ARG1L
    movff       mixgyr , ARG2L
    call        MULS2X1
    movf        RES0 , W
    addwf       elevL , F
    movf        RES1 , W
    addwfc      elevH , F
    movf        RES2 , W
    addwfc      elevU , F
```

A similar process is used to compute mixing of rudder command into elevator command. The only difference is that the servo signals are used instead of the gyro signals:

```
    movf        PWM1_dc , W
    movff       strngTrim , strngTrimTemp
    subwf       strngTrimTemp , F
    bov         mixinover

squareRudd
    movff       strngTrimTemp , ruddunfL
    clrf        ruddunfH
    btfsc       strngTrimTemp , 7
    comf        ruddunfH , f

    smult       ruddfilH , strngTrimTemp
    movff       PRODL , ARG1L
    movff       PRODH , ARG1H
```

Multiply by mixing gain and add into the total:

```
movff       mixgain , ARG2L
call        MULS2X1
movf        RES1 , W
addwf       elevL , F
movf        RES2 , W
addwfc      elevH , F
movlw       0x00
btfsc       RES2 , 7
movlw       0xFF
addwfc      elevU , F
```

Pulse width modulation servo control

The most popular analog servos respond to pulse width modulation. Because several channels are time-division multiplexed over a single radio channel, the pulse width is narrow compared with the repeat period. Pulse width is on the order of 1 to 2 milliseconds, repeated approximately every 20 milliseconds. The servos include “pulse-stretchers” internally to convert the 1 to 2 milliseconds to something that can be used to assert control between pulses. For that reason, it will not do you any good to send pulses to the servos any faster that about once every 20 milliseconds. Any faster than that and you will cause the “H-bridge” in the servo to actually short the power supply.

The servos move to the approximate center in response to a 1.5 millisecond pulse and move to the approximate extremes of motion in either direction in response to 1 or 2 millisecond pulses. To control the servos you need to map your internal representation of servo deflection to pulses with a range of 1 to 2 milliseconds repeated approximately every 20 milliseconds.

This was accomplished in the prototype firmware described here with the aid of the task structure described in one of the previous sections. In particular, there are two tasks that were used to construct the pulses. Each task repeats approximately every 20 milliseconds to match the desired repetition rate. The first task simply raises the control line to the servo for approximately 1 millisecond. The second task generates a partial pulse from 0 to 1 milliseconds by loading the PWM control registers with appropriate values. The 18F2520 has a 10 bit PWM control register. For simplicity, only 8 bits were used in the prototype firmware, it was found that control was smooth enough at that resolution.

The following task generates a 1 millisecond portion of a pulse on channel 1:

```
PWM1_full_pulse

      movlw       0xFF
      movwf       CCPR1L
      bsf         CCP1CON,CCP1X
      bsf         CCP1CON,CCP1Y
```

The following task, when executed 1 millisecond after the previous task, generates a partial pulse, taking advantage of the fact that CCP1X and Y are already set:

```
PWM1_pulse
      movff       PWM1_dc , CCPR1L
```

After the complete pulse is generated, the PWM control must be cleared to prevent more pulses from going out:

```
PWM1_clear
      clrf         CCPR1L
      bcf          CCP1CON,CCP1X
      bcf          CCP1CON,CCP1Y
```

Complete PWM control of the first PWM channel consists of the execution of PWM1_full_pulse, pulse, and clear, on three sequential tasks. The PWM control of the second channel is similar.

The 8 bit variable PWM1_dc ("dc" means "duty cycle") controls the total pulse width. The actual pulse width in milliseconds is approximately equal to 1 plus PWM1_dc/256. Therefore, to convert a computed signed 8 bit desired servo deflection value into a "dc" value, simply toggle the most significant bit.

Navigation

Navigation in the prototype firmware is very simple, based on aiming toward a target point. The desired direction is the direction of the vector from present location to the target point. An interesting elegant side effect of this algorithm is that once the target point is reached, the ensuing trajectory is a circle around the target point, with an error between the actual and desired direction of 90 degrees. The trajectory is quite stable. The radius of the circle depends on the speed and the feedback gains.

Here is what the code looks like:

```
nav_circle
```

Check to see if the control is in the circling mode:

```
btfss        circlingM
return
```

Retrieve the present location, xyWorldLH, and store in the temporary variable XY_rectLH:

```
movff        xWorldH,X_rectH
movff        xWorldL,X_rectL
movff        yWorldH,Y_rectH
movff        yWorldL,Y_rectL
```

Subtract the target coordinate, xyFocusLH, from the present location. It would have been more convenient to do it the other way around, but this code evolved from another piece of code, so it just happened to come out this way:

```
movf xFocusL,w
subwf        X_rectL,f
movf         xFocusH,w
subwfb       X_rectH,f
movf         yFocusL,w
subwf        Y_rectL,f
movf         yFocusH,w
subwfb       Y_rectH,f
```

Convert from rectangular to polar coordinates. The angle of the vector is returned in THETA.

```
call        rect_to_polar
movf        THETA, w
```

Flip the sign of the angle, because we actually used the negative of the vector we should have:

```
addlw       0x80        ; come home angle, inward
```

The result is the direction that we should be going. Setting desiredDir to this value will cause the rudder feedback loop to use gyro yaw information and GPS heading information to seek to head towards that direction:

```
movwf       desiredDir
return
```

Math

The entire prototype firmware was written without requiring any division. All computations were performed with fixed point arithmetic.

A math library was written to perform the needed math computations. In addition to multi-byte signed and unsigned multiplication operations, the following three routines were particularly useful:

- sine_lookup, cosine_lookup – A very efficient method to compute the sine or cosine using a lookup table.
- rect_to_polar – An efficient method for converting from rectangular to polar coordinates based on a technique called Cordic arithmetic, the same technique that is used in hand calculators to perform the same conversion.

The sine and the cosine lookup are similar. The following is an implementation of the sine lookup. The angle is a signed 8 bit value in THETA. The sine is fetched from a 256 entry table of 2 bytes per entry and returned in the variables SINEL and SINEH:

```
sine_lookup
```

Load the table pointer with the address of the sine table using a previously defined macro to do that:

```
ld_tblptr   sine_table
```

Add THETA two times to the table pointer:

```
movf        THETA , W
addwf       TBLPTRL , F
movlw       0x0
addwfc      TBLPTRH , F

movf        THETA , W
addwf       TBLPTRL , F
movlw       0x0
addwfc      TBLPTRH , F
```

The table pointer now points to the desired table entry, which can now be fetched using the table read and increment instruction:

```
tblrd*+
movff       TABLAT , SINEL
tblrd*+
movff       TABLAT , SINEH
return
```

The routine rect_to_polar converts from rectangular to polar coordinates by performing a binary search one bit at a time on the 8 bit resultant angle. At each step of the search, one bit of the angle is determined based on the sign of the y coordinate and the vector in rectangular coordinates is rotated toward the x axis. When the computation is complete the y coordinate is zero, the x coordinate is the magnitude of the vector, and the polar angle is determined. Here is an implementation:

```
rect_to_polar
```

The variable theta_temp will be used accumulate the polar angle. Initialize it to zero:

```
clrf        theta_temp
```

The variable delta_theta is the amount of rotation. It starts at the equivalent of a quarter of a full circle:

```
movlw       B'01000000'
movwf       delta_theta
```

The following cordic step is repeated until the computation is complete:

```
cordic_step
```

Rotate clockwise if the sign of Y is positive, else rotate counterclockwise. Add or subtract delta_theta from the accumulated value of the polar angle accordingly:

```
movff       delta_theta , THETA
btfss       Y_rectH , 7
negf        THETA

rcall       ROTATE      ;     perform the rotation
movf        THETA , W
addwf       theta_temp , F
```

Divide the angle increment by 2 to proceed to the next bit in the binary search:

```
rrncf       delta_theta
```

Keep going until we are done:

```
btfss       delta_theta , 7
bra         cordic_step
```

Compute the least significant bit of the result:

```
btfss       Y_rectH , 7
decf        theta_temp , F
movff       theta_temp, THETA
```

We actually computed the negative of what we want, so we must negate the result:

```
negf        THETA
return
```

The routine ROTATE uses sine and cosine lookup to rotate the vector XY_rectLH by THETA:

```
ROTATE
```

Compute both the sine and the cosine of THETA:

```
rcall       TRIG_LOOKUP
```

Multiply cosine times X, load into X_temp:

```
movff       COSINEH , ARG1H
movff       COSINEL , ARG1L
movff       X_rectH , ARG2H
movff       X_rectL , ARG2L
rcall       MULS2X2ROT
movff       RES2 , X_tempH
movff       RES1 , X_tempL
```

Multiply sine times X, load into Y_temp:

```
movff       SINEH , ARG1H
movff       SINEL , ARG1L
rcall       MULS2X2ROT
movff       RES2 , Y_tempH
movff       RES1 , Y_tempL
```

Subtract sine times Y from X_temp:

```
movff       Y_rectH , ARG2H
movff       Y_rectL , ARG2L
rcall       MULS2X2ROT
movf        RES1 , W
subwf       X_tempL , F
movf        RES2 , W
subwfb      X_tempH , F
```

Add cosine times X to Y_temp:

```
movff       COSINEH , ARG1H
movff       COSINEL , ARG1L
rcall       MULS2X2ROT
movf        RES1 , W
addwf       Y_tempL , F
movf        RES2 , W
addwfc      Y_tempH , F
```

Copy the temporary back out to the vector:

```
movff       X_tempH , X_rectH
```

```
movff       X_tempL , X_rectL
movff       Y_tempH , Y_rectH
movff       Y_tempL , Y_rectL
```

return

Trim, offsets

You will want to give some thought to control surface trim and sensor offsets. In the prototype firmware, trim and offsets were handled as follows:

- The offsets of the gyros and accelerometers were measured during a self-calibration process during power up, to give the best chance of removing them later. The way this was done was by simply snap-shotting the filtered values after the filters had sufficient time to reach steady state. It was found that there was practically zero residual gyro drift, at least not enough to affect the controls. Without this self-calibration process the resting voltage output of the gyros and accelerometers are uncertain enough to result in considerable bias.
- During the power up self-calibration process, the angle of the accelerometer was recorded as a baseline for the pitch angle calculations, compensating for any slight angle in the mounting of the board.
- The rudder and elevator positions were recorded during the power up sequence, and served as baselines for any computation that required the difference between joystick positions and neutral settings, such as the computations involved in computer-augmented manual control. The joystick positions were always included as terms in the control of the rudder and elevator so that the trim could be continuously adjusted throughout the course of the flight.

Installation

During debugging of your firmware it is simplest to separate your electronics from your plane. Simply connect everything up, including some spare servos, out in the open so that you can see the LEDs and have easy access to connections and switches. You should have some idea on how you intend to mount the GPS-UAV in your plane, though, so that you can do testing and debugging with the GPS-UAV pointing in the same direction as it will be in your plane.

At some point you will want to do some debugging with your actual plane. You still may want to be able to see the LEDs, so you might want to do this with the wing off, with a partial installation. This is particularly easy to do with a sailplane such as the gentle lady.

During debugging, you may or may not decide to install the GPS backup battery. It does result in faster satellite acquisition, but if you forget and leave the battery installed for several weeks, it will become depleted. It is best to take it out when you are not using it.

There will come a time, of course, when you will want to do some flying, so you will want to final installation of the GPS-UAV into your plane. Here are some thoughts and tips:

- Be careful during installation not to twist or bend the GPS-UAV board. The author did this once, and cracked some traces. It is best to remove other components, if they are in the way.
- The author used both internal and external installations. In particular, the external installation was used in the early stages, when the author’s hand-assembled

breadboard would not fit inside his Gentle Lady, and was simply attached to the nose with rubber bands. This approach is NOT recommended. Eventually, the author's board was destroyed by the spinning propeller during an aborted takeoff. The GPS-UAV should fit inside most sailplanes, and that is the recommended place to put it.

- It is recommended to mount the GPS-UAV with the components facing up. It does not matter whether the antenna connector points toward the nose or toward the tail of the plane. Either will work just fine, though you will have to take account of which orientation you use, because it will reverse the sense of the pitch gyro and the pitch accelerometers. You will probably want to select an orientation that simplifies connections. The author mounted his GPS-UAV in his Gentle Lady in the compartment under the wing, with the connector for the antenna facing the tail, with the antenna in the same compartment.
- Foam rubber around the GPS-UAV and around the antenna is recommended.
- Be careful not to flex the antenna wire too much, particularly where it connects to the antenna. The author broke a couple of wires this way. It is recommended that you mechanically reinforce the connection such as with a small dab of epoxy.
- If you are concerned about reducing total weight, you might want to use lightweight servos and battery. In the end, the author used a regular sized battery and lightweight servos.
- Connect the battery to the radio receiver, through a power switch if you want, in the usual fashion.
- Connect the servos to the outputs of the GPS-UAV. Connect three channels from the radio to the inputs.
- You might want to use servo extension connectors between the GPS-UAV and your radio. You will probably be able to connect your servos directly to the GPS-UAV.
- The GPS-UAV servo connectors may or may not connect directly to your servos and radio if there is a tab on your servo connectors. You may wish to trim away the tabs, or you may want to unsolder the connector wires and make your own from your favorite servo connector extension cables.

Flying

Finally, some flying suggestions, assuming that you are using a sailplane and the test firmware that is available on the Spark Fun website:

- Make sure you have the rudder and elevator trims set on the transmitter for where you want them. The test firmware records the trim positions during power up. Trim can be adjusted during initialization of the GPS if you do it quickly before initialization is complete, but you might want to do it before initialization, or you might want to force a re-initialization after setting the trims to make sure they are recorded.
- You may or may not wish to use the GPS battery backup. Using it will reduce the amount of time it takes for the GPS receiver to become active.
- Before you turn on the GPS-UAV, make sure the plane is level and at the location you want to record as the "return-home" point. The test firmware records the pitch attitude during power up and uses it as an offset in the pitch measurement. Make sure the third, command channel, typically the throttle, is set for full off, with full off trim.

- During the final stages of initialization, the test firmware will wag the rudder every two seconds to signal that the GPS is active and that the initialization is completing. If the wagging does not occur, try cycling the power. If that does not work, take the wing off and make sure that the GPS led is flashing to indicate the GPS is active.
- Make sure that manual control of rudder and elevator are working normally, particularly that they move in the correct direction.
- Perform your usual radio range check.
- Place the controls in the partially augmented mode, typically by advancing throttle to mid position, and check the operation of accelerometers and gyros by pitching and yawing the plane and then see if the rudder and elevator respond appropriately, particularly if they move in the correct direction.
- Shut the transmitter off. The rudder should respond seemly at random, because the plane is at the “return-home” point. Pick up the plane and walk it some distance away. Turn around and walk back toward the “return-home” point. Manually yaw the plane and the direction that you are walking in response to the rudder position to see if the “return-home” function of the test firmware appears to working properly.
- Turn the transmitter back on and set the controls for manual. Make sure that manual control is still working. You can make small trim adjustments if you want.
- Launch your plane under manual control, and maintain manual control until you reach maximum altitude.
- If you are using the test firmware, experiment with augmented control, circling control, and return-home control. If you are using the throttle to select the control mode, manual control is throttle full off. Augmented control is selected with a mid range throttle setting with the amount of augmentation proportional to the setting. Circling control is selected by full throttle and full throttle trim.
- Under augmented control, the test firmware will use the gyros and accelerometers to stabilize the plane, yet at the same time will respond to manual elevator and rudder controls. The “feel” of the plane will be about the same as it is in manual control, except the tendency of the plane to respond to wind gusts will be greatly reduced, as well as any tendency for the plane to “porpoise” will be eliminated. In augmented mode, it should be possible to easily fly the plane close to its stall speed.
- Under circling control, the test firmware will record the longitude and latitude of the position at the time the circling control is engaged, and will result in circling control around that location. If there is any sort of lift or thermals, this will “lock” the plane into the lift. You can still adjust the elevator trim to control altitude.
- If you shut the transmitter off, the plane should respond in a few seconds by turning back toward its initialization location, and fly in a straight line towards it. After passing over it, the plane will circle that location. If you shut the transmitter off, *do not forget to turn it back on*, especially if the “return-home” function becomes unstable. Do not panic, but do remember to turn the transmitter back on so that you can resume manual control.
- Make careful observations of your plane in flight so that you can make refinements in the controls afterwards. If something does not seem to be working right, put the controls in manual mode and land the plane. Then, manually walk it around, observing rudder and elevator, to understand what is going on.