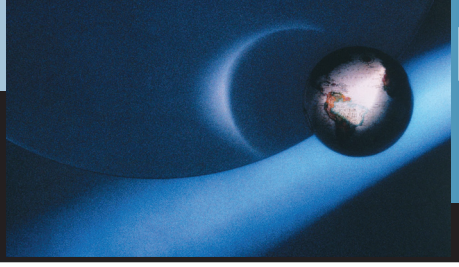

NEAR SPACE

APPROACHING THE FINAL FRONTIER

■ BY L. PAUL VERHAGE

PROGRAMMING THE ULTRALIGHT NEAR SPACE FLIGHT COMPUTER

The UltraLight has a split personality. One side — its tracker side — reformats GPS sentences into position reports for transmission over amateur radio. Because of it, chase crews can follow the near spacecraft and recover it at the end of its mission. Its other side — the PICAXE side — operates experiments, as well as records their results. It's the reason we launch near spacecraft in the first place. The previous column described how to program the tracker side, so this time we'll cover the PICAXE side.

MEMORY

The UltraLight takes advantage of the memory and I/O features found in the PICAXE-28X1. First, let's look at memory. There are actually three types of memory in many of the PICAXE microcontrollers. The type most *Nuts & Volts* readers are familiar with is the PICAXE-28X1's 4K of EEPROM memory. This is where the UltraLight's flight program is stored. Data can also be stored in this EEPROM along with the flight code, but with a 32 Kb memory bank onboard the UltraLight, there's no need to store data in the PICAXE. Now, a limit of 4,096 bytes for flight code may sound small. However, a program large enough to operate four analog sensors, a Geiger counter, two cameras, a servo, and GPS receiver requires less than 1/5th of that memory.

The next type of memory is the random access memory or RAM. RAM is actually broken into two types. There's the 28 bytes of RAM (addressable as B0 to B27 or W0 to W13) for the logical and mathematical manipulation of data. This is the RAM used by commands like FOR/NEXT and LET. Then there's an additional 95 bytes of temporary storage that can't be accessed directly by name. Instead, the POKE command writes variables into this memory and the PEEK command reads it back out. Both the POKE and PEEK commands reference specific bytes by their numerical address. So, for example, the command **PEEK B0,100,** stores the value currently located in B0 to a byte located at position 100. This second type of RAM memory is useful when a separate, protected copy of the variable values is required. I like to think of it as keeping a second accounting book. Remember that data stored in RAM memory is lost when power is removed from the UltraLight. Therefore, anything important must be written into EEPROM.

In addition to the general-purpose and storage memory described above, there's a third type of memory consisting of an additional 128 bytes called scratch pad memory. Like the storage variables above, data in scratch pad memory can't be referenced directly by name either; it's addressed at the byte level by its address. However, instead of POKE and PEEK, memory in scratch pad memory is accessed using the READ and WRITE commands. You'll see that the UltraLight makes extensive use of scratch pad memory because of the GPS receiver.

To use the scratch pad memory for GPS sentences, the PICAXE-28X1's hardware serial port must first be configured. The following command sets it up for the GPS receiver's 4800 baud, inverted sentences:

```
hsersetup b4800_4,%00
```

Use the following command to receive 128

characters of GPS data in scratch pad memory. The command waits for the letter "W" before recording GPS sentences:

```
hserin [4000,Bad_GPS],0,1,("W")
```

Depending on the GPS receiver configuration, you may need several HSERIN commands to isolate the specific GPS sentence desired.

Now that a GPS sentence (we hope the correct one) is loaded into scratch pad memory, the PICAXE parses the data to locate the information of interest. One way to determine if the GPS sentence in scratch pad is the GPGGA sentence (the one I use the most often) is to look for the meters symbol (M) some 40 characters into the sentence. If the GPS does not have a lock on satellites, then the time field in the GPGGA sentence remains blank which moves M closer to the beginning of the sentence. I've started using the following code to determine if the GPS has a lock:

```
Scan_GPS:
  for b2 = 40 to 55
    get b2,b0
    if b0 = "M" then Confirmed_Good_GPS
  next
  goto GPS
```

The range in which to search for the M character can be tightened up if you know the format of your GPS receiver's GPGGA sentence. Alternatively, one could write code that checks to see that the first six characters after GPGGA are digits (in other words, the time field is filled). Since GPS sentences are made up of ASCII characters, a way to convert an ASCII digit into binary form is needed if the UltraLight is to use fields like the altitude. The following example demonstrates one way to convert the ASCII altitude into a word sized variable. A similar conversion could be used to convert most of the ASCII time into a word variable:

```
Get_Altitude:
  get 42,b0,b1,b2,b3,b4,b5
  b0 = b0 - 48
  Reading = b0
  if b1 = 46 then Altitude_Done
  b1 = b1 - 48
  Reading = Reading * 10
  Reading = Reading + b1
  if b2 = 46 then Altitude_Done
  b2 = b2 - 48
  Reading = Reading * 10
  Reading = Reading + b2
  if b3 = 46 then Altitude_Done
  b3 = b3 - 48
  Reading = Reading * 10
  Reading = Reading + b3
  if b4 = 46 then Altitude_Done
  b4 = b4 - 48
  Reading = Reading * 10
  Reading = Reading + b4
```

The code reads six ASCII characters out of scratch pad memory and then converts each into its binary representation by subtracting 48 from its ASCII value. Each digit is added into the final altitude variable by first multiplying the altitude by 10. This is essentially shifting the altitude variable in base 10 before adding the next digit to the altitude.

THE ULTRALIGHT MEMORY BANK FOR DATA

Before discussing the input and output of the PICAXE-28X1, let's look at the EEPROM memory for flight data. The UltraLight's 24LC256 EEPROM memory uses fast (400 kHz) I^2C communications and double byte (one word) addressing. The command that configures communication between the 24LC256 and the PICAXE-28X1 must be executed once before the memory can be accessed:

```
i2cslave %10100000,i2cfast,i2cword
```

Since the EEPROM is addressed by two bytes, data must be stored one word — or two bytes — at a time. The following code writes the values of bytes B6 and B7 into EEPROM. The memory address of the write is tracked with the word W0:

```
Store_Data:
  writei2c W0,(B6,B7)
  pause 10
  W0 = W0 + 2
  return
```

After recovery, the data stored in EEPROM is read out by the following code:

```
Download:
  sertxd ("Start,")
  for W0 = 0 to 4091 step 2
    readi2c W0,(B2,B3)
    sertxd (#W1, ",")
  next
```

Since the EEPROM is addressed by the word, the FOR-NEXT loop steps through the address by twos. The subroutine also assumes there are 4,091 bytes of data recorded in memory; you'll need to change this based on the flight. Mission data is downloaded over the programming port with the PICAXE Editor's built-in terminal program. You'll find the terminal program under PICAXE - Terminal. Be sure to set the Terminal for a download speed of 4800 baud.

INPUT/OUTPUT PORTS

Now, let's look at the inputs and outputs to the PICAXE-28X1.

Analog Port

For the collection of sensor data, the UltraLight has four channels (ADC0 to ADC3) in the Analog port. Each channel digitizes (that is, converts analog voltages into their digital values) sensor voltages in the range of zero to five volts. The commands READADC and READADC10 permit the PICAXE to digitize voltages with resolutions of either eight bits (19.5 mV per bin), or 10 bits (4.8 mV per bin, or four times the resolution of the READADC command), respectively. Each channel of the Analog port provides five volts and ground to each of its experiments. This means that in most cases, experiments will not require a separate power supply. You literally plug and play the experiment.

The following code digitizes sensor voltages into 1,024 bins (10 bits of resolution). Each reading is then stored into EEPROM in one word records. More data could be packed into EEPROM if two eight-bit analog readings are stuffed into a word and then stored, but that would leave out the third sensor voltage. Perhaps it could be stuffed into a word along with a digital sensor reading:

```
Analog:
  for B2 = 0 to 2
  readadc10 B2,W0
    gosub Store_Data
  next
  return
```

Digital Port

Not every experiment produces an analog voltage. Therefore, the UltraLight also has a Digital port for sensors like Geiger counters. In the case of the Geiger counter, the signal is a digital pulse at the detection of each cosmic ray. For this example, the PICAXE counts the number of "clicks" over a fixed time interval. The Aware Electronics RM-60 Geiger counters I typically use produce a 20 microsecond long pulse at each detection. The PICAXE counts the number of pulses for 10 seconds, and I later multiply the result by six in a spreadsheet. Here's an example of counting cosmic rays with the Digital port:

```
Cosmic_Ray:
  count 0,10000,b2
  gosub Store_Data
  return
```

As with the Analog port, every channel in the Digital port provides experiments with five volts and a ground, relieving the designer of creating a power supply for the experiment. The channels in the Digital port are input 0 to input 2.

Servo Port

There are times when devices must be opened, closed, or oriented during a near space mission. For those cases, the UltraLight also has a Servo port with two channels. There is a small risk that servos could drain batteries during a mission. There's also a tiny risk that the operation of a servo will create voltage jitters that interfere with the rest of the UltraLight. For those reasons, the Servo port uses a battery pack separate from the main battery pack. The servo channels are output 0 and output 1.

The following code rotates a servo on channel 0 to a position of 110 (close to the midpoint for most servos):

```
servo 0,110
```

The range of a servo's rotation depends on the servo, but you can usually expect it to be from 75 to 225. If you are going to drive a servo close to its limits, first test it by gradually increasing the position value in the servo command.

Camera Port

Each of the UltraLight's two-channel Camera ports have two ways to operate cameras. The first is for cameras with modified shutters. In most cameras today, the shutter switch is not a mechanical system, but instead is a switch that signals the microcontroller inside the camera to record an image. We can bypass the shutter switch in this kind of camera with two wires. The wires are connected to the Camera port so that the relay on board the UltraLight replaces the function of the shutter switch.

The same signal that operates a relay can also operate a Canon camera running CHDK and the remote USB shutter script. I haven't tested it yet; however, I suspect the UltraLight can operate both the relay and USB connections simultaneously. If so, the UltraLight can operate four cameras. (That is, if you don't mind them acting together in pairs.)

To take a picture, the PICAXE applies five volts to the relay or USB port for about one second to trigger the shutter. After waiting one second, the shutter is released so the camera is ready to take the next picture. The code to do just that is shown below:

```
Camera:
  high 4
  pause 1000
  low 4
  return
```

Antenna Port

At the bottom of the UltraLight is an SMA connector. This is the output for the two meter transmitter used to send position reports to chase crews on the ground. In a later article, I'll describe an antenna (and a camera rotator)

that works well with the UltraLight.

Commit Pin

The last input to the UltraLight is the Commit pin. The Commit pin was built into the UltraLight because GPS receivers require time to get their satellite lock. So, rather than have the UltraLight record data while the near spacecraft is on the ground waiting for the GPS to catch up, the Commit pin inhibits the UltraLight from operating in flight mode. This doesn't mean the UltraLight can't operate experiments on the ground, however. It just means the UltraLight only performs actions on the ground that you desire. If you're like me, you don't want the camera taking lots of pictures of the launch crew filling the weather balloon — you want to save the SD card for pictures in flight.

The UltraLight Commit pin can be used in at least two ways. To start with, it can signal the UltraLight to download mission data when it is first powered up. Doing so permits recovery crews to download flight data in the field without having to reprogram the UltraLight. If the Commit pin is missing when the UltraLight starts up, then the PICAXE assumes the mission has been completed and the data is ready to be downloaded. Use this code to signal the UltraLight to download data:

```
if pin6 = 1 then Download
```

To use the Commit pin to inhibit data collection until closer to launch, use this code:

```
Wait_to_Commit:
  if pin6 = 0 then
Wait_to_Commit
' wait for commit pin to be
pulled
```

If this code occurs *after* the download detection, then the Commit pin does double-duty of downloading data and starting data logging.

THAT'S A WRAP

That completes the UltraLight flight computer. You now have all the information needed to construct and program your own. As I've mentioned before, *Nuts & Volts* readers are allowed to make their own UltraLight computer. I just ask that no one make copies to sell. Let me know how your project goes and I'll be happy to help readers get their own flight computer up and running.

For those who are interested in near space exploration but can't make their own printed circuit boards, NearSys LLC (**nearsys.com/catalog**) produces a complete UltraLight kit.

Onwards and Upwards,
Your near space guide **NV**