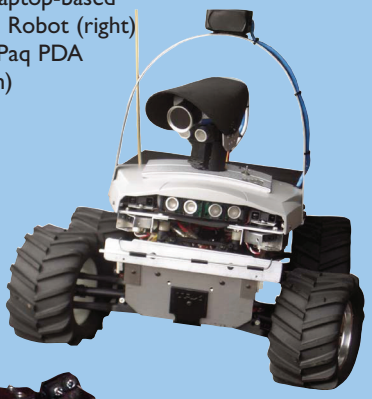
[Part 3]

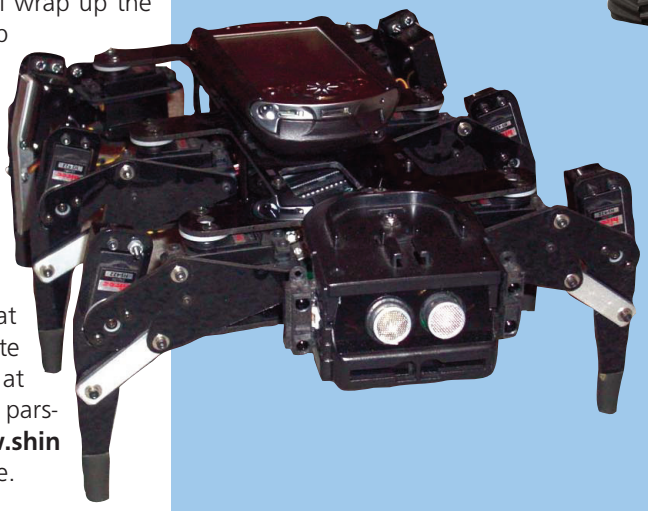
INTERMEDIATE ROBOTS

Building a Laptop- or PDA-Based Robot

MEET THE 'BOTS

- HelmBot — iPaq PDA Robot (left)
- Seeker — Laptop-based Robo-Magellan Robot (right)
- BugBot — iPaq PDA Robot (bottom)





In the first two parts of this series, I described the hardware design and core PC software for a laptop-based robot. In this final article of the series, I will finish up on the laptop software with a discussion of GPS and color tracking. Then, I will describe the PIC microcontroller software, and explain how the laptop and PIC communicate. Finally, I'll wrap up the series with some information on how to replace the laptop with a PocketPC-based PDA, such as an iPAQ.

Reading GPS

The laptop communicates with a GPS receiver via serial over USB, RS-232, or Bluetooth, using a protocol known as NEMA 0183. This standard was developed by the National Marine Electronics Association (NMEA). I had started writing an NMEA parser state machine when I discovered a great version written by Monte Variakojis of VisualGPS, LLC. Monte has an excellent white paper describing the parsing of NMEA at **www.visualgps.net** With his permission, I have integrated his parser code into the Seeker project and made it available at **www.shinsel.com/robots** See the sidebar on GPS parsing to learn more.

BY DAVE SHINSEL

After parsing the GPS information, the Seeker software function GPSDegreesToRWInches() converts the reported latitude and longitude to the robot map coordinates, in inches from a fixed origin. The internal robot map may be "anchored" to the real world by associating any single GPS coordinate with its corresponding X,Y robot map coordinate. This then allows all other data such as path waypoints (described last month), obstacles, etc. to be entered as either GPS coordinates or X,Y coordinates relative to the anchor point.

The robot can then use the GPS receiver to track its location and navigate to any location on the map. Note, however, that GPS is generally not accurate enough for small robots. The position error can be 20 feet or more, so the GPS is usually only used to augment the compass and odometer in tracking the robot position. In addition, most GPS receivers do not work well (if at all) indoors, so GPS is mostly usable for outdoor robots.

Object Color Tracking

For the SRS Robo Magellan contest, the robot must be able to find and touch an orange traffic cone that has been placed at the end of the course. In addition, bonus time is awarded for touching other cones scattered along the way. The most common way to do this is with "color blob" spotting – find a blob with lots of orange, and it is probably a cone!

I considered using the popular CMU Cam for this, but was not really satisfied with the performance for spotting a cone outdoors at long range. In addition, I really wanted to do some work on the laptop with vision, with the idea that at some point I could add advanced vision processing, such as object avoidance and path navigation.

My first laptop robot — Mr. Roboto.

Mr. Roboto reading *Nuts & Volts*.

Mr. Roboto roaming.

I had looked at several vision libraries when I discovered Robin Hewitt's Mavis project. This turned out to be a good fit for what I needed, but Robin had not yet added color blob tracking. Well, I went ahead and integrated (okay, hacked) Robin's library into my robot code, and then added a new ColorBlob object to the ObjRec (Object Recognition) class.

To implement the ColorBlob object, I needed a way to reliably detect a color, and reject all others. I had heard that the RGB (red, green, blue) values that most cameras provide are very difficult to use for color spotting, as any change to brightness causes all three values to change, and not uniformly! I had also heard that working in "YUV" color space works a lot better, but did not know how to go about converting from RGB to YUV.

Fortunately, a friend of mine, Matt Curfman, pointed me to the "FourCC" website at **www.fourcc.org** This website has a ton of information about common video formats and color space conversion. I was able to find a simple conversion formula to get to the pure red and blue color components, with all luminance (brightness) removed. The code boils down to the following, with Y representing luminance:

```
Y = 0.299*R + 0.587*G + 0.114*B;
// Clamp Y at valid values
if( Y < 16 ) Y = 16;
if( Y > 235 ) Y = 235;
Cr = 0.713*(R - Y);
if( Cr < 0 ) Cr = 0;
if( Cr > 255 ) Cr = 255;
Cb = 0.564*(B - Y);
if( Cb < 0 ) Cb = 0;
if( Cb > 255 ) Cb = 255;
```

Color purists may argue about the correctness of this. In fact, there are several conflicting versions posted at FourCC, but this works pretty well for this application. Note that the result is just two values: Cr (Component Red) and Cb (Component Blue). The green component can be calculated from the red and blue, so is not needed.

The actual color tracking code is quite simple. The camera is pointed at a sample of the color you wish to track, and the Cr and Cb values are stored as the target color. In operation, each video frame is quickly converted from RGB to Cr/Cb values, and then the frame is searched for values where both the Cr and Cb are within a pre-defined threshold of the target color. If enough pixels are found that "match" the target color, an object is considered found. The average X and Y of all the matching pixels is computed during the search, and returned as the center of the color blob.

Once the color blob X,Y coordinate is known, the location is sent from Mavis to the Robot Camera Control Module. The distance of X,Y from the center of the frame is used to calculate the amount of servo travel needed to center the color blob in the video frame.

A BRIEF INTRODUCTION TO GPS DATA FORMAT

Here are a few "NMEA Sentences" (as they are called) from a typical GPS receiver:

```
$GPGSA,A,1,31,03,25,29,,,,,,,,,11.9,6.1,10.2*30
$GPGSV,3,1,10,29,70,130,37,28,53,328,,15,53,026,,21,47,101,*73
$GPGSV,3,2,10,08,28,121,,25,27,014,39,14,25,253,,31,23,292,41*7D
$GPGSV,3,3,10,09,09,143,,03,07,334,36*7B
$GPRMC,054105.998,V,4250.5461,S,14718.4860,E,0.26,185.02,211200,,*0A
```

You got all that? Well, it's actually not so bad. The last sentence shown is one of the most common sentences: RMC, or "Recommended Minimum Navigation Information." RMC is formatted as follows. Each comma represents one "field" as follows:

Field:	1	2	3	4	5	6	7	8	9	10	11	12
$GPRMC	054105.998	V	4250.5461	S	14718.4860	E	0.26	185.02	211200			*0A

For RMC, the fields are defined as follows:

1. Time (Universal Time Code)
2. Status (V = Navigation receiver warning)
3. Latitude
4. N or S
5. Longitude
6. E or W
7. Speed over ground, knots
8. Track made good, degrees true
9. Date, ddmmyy
10. Magnetic Variation, degrees
11. E or W
(* instead of comma)
12. Checksum

Generally, the robot mostly cares about the latitude, N/S, and longitude, E/W. This can be used to map the robot location anywhere on the planet, but there are many other NMEA Sentences that provide information such as direction quality of fix and number of satellites tracked.

For more information about NMEA Sentences, see
www.gpsinformation.org/dale/nmea.htm

*Other names and brands may be claimed as the property of others.

This information is sent to the camera tilt and pan servos, causing the camera to track the object nicely. The Robot Navigation Module uses the camera pan servo position to drive the robot to the object.

There is plenty of room for improvement to the current implementation. First on my list is to add a region search to make sure enough of the target pixels are clumped together to assure a valid object. Second is to check the object outline shape, to make sure that the robot is not tracking somebody's jacket! However, despite its limitations, the current implementation works surprisingly well. Take a look at Figure 1 and notice the red cross hairs on the cone. The cross hairs indicate the center of the tracked color blob.

If you are interested in learning more about vision processing, I highly recommend that you read the "Getting Started with Vision" series of articles by Robin Hewitt, starting with the July 2005 issue of *SERVO Magazine*.

PIC Microcontroller Software

Remember from the last article that the PIC acts as a "slave" to the laptop software. All higher-level decisions are made on the laptop, while hardware control, sensor input, and timing-critical tasks are performed by the PIC. All control actions are in response to a command from the laptop.

PIC Main Loop

The PIC software is designed around two loops that run continuously. You are probably already familiar with the concept of a "Main" loop. Most small robot controllers follow this basic design:

- Initialize software variables.
- Initialize hardware.
- Start of Loop
  - Read sensors.
  - Decide what to do.
  - Issue motor commands.
- Go to Start of Loop.

For a laptop-based robot, the PIC Main Loop is modified to handle serial communication with the laptop:

- Initialize software variables.
- Initialize hardware.
- Start of Loop
  - Check for Serial Command received from the Laptop. If a command is ready, process the command. If the command is Get_Status, send the current status to laptop.
  - Every 20 ms, read A2D sensors (IR rangers, battery level, etc.), and keep a running average.
  - Every 40 ms, read Compass and keep a running average.
- Go to Start of Loop.

Note that the decision of "what to do" is now handled by the laptop. Also, note that sensor readings only take place as often as each sensor is able to update. Even so, most sensors may be read many times faster than the laptop requests the sensor data. The program takes advantage of this to average the sensor readings on the PIC to improve sensor accuracy.

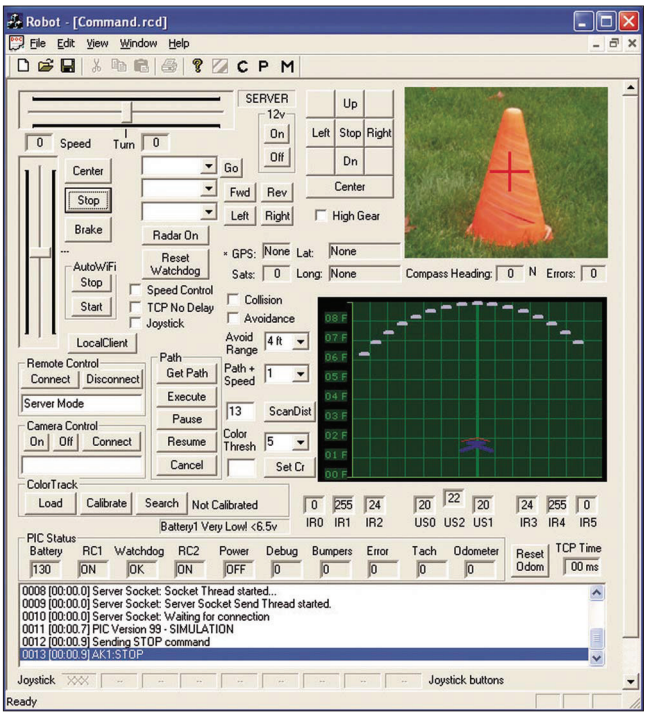
PIC Timer Loop

For tasks that require precise timing — such as servo control — a hardware timer (Timer0) is used to form a loop made up of 1 ms "ticks." At each timer interval, the Main Loop is interrupted, a single "tick" in the Timer Loop is executed, and then control is returned to the Main Loop.

The Timer Loop syncs around a 20 ms cycle (convenient for servo control) as follows:

- Base Code (Executed each time):
  - Increment/decrement timers.
  - Check wheel odometer for black/white transition.
  - Check ultrasonic capture register for echo received.
- T0: 400 µS duration
  - Copy servo values to counters.
  - Transition high for all enabled servos.
  - Start ultrasonic sensor pulse.
- T1: 3 ms duration
  - Go low when appropriate for each servo. This allows for full servo travel, with fine granularity.
  - During this period, execute Base Code every 1 ms to keep timings correct.
- T2: 600 µS duration
  - Pad to get back to an even 1 ms cycle.
- T3: 1 ms
  - Check bumper switches.
- T4: 1 ms
  - Check "dead man" RC control.
- T5: 1 ms
  - Every 200 ms, calculate tachometer and odometer.
  - Calculate speed control feedback and adjust motor speed.

FIGURE 1. Seeker Command GUI.



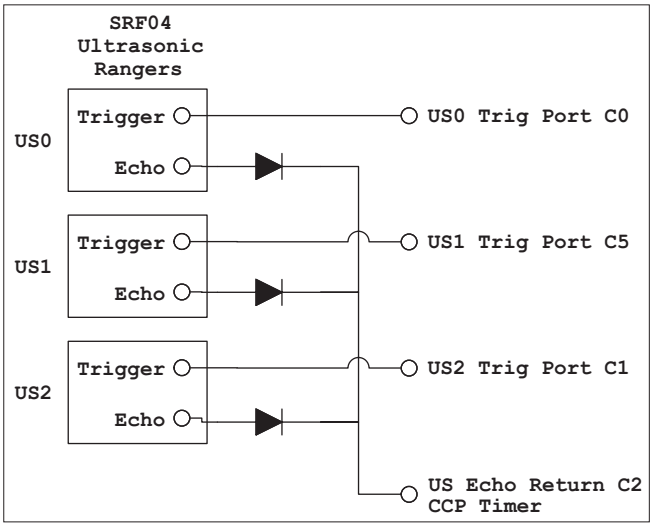


FIGURE 2. Ultrasonic Interface.

- T6: 1 ms
  - Handle slow servo movements (automatically increments servo position).
- T7: 1 ms
  - Handle motor "brake" control.
- T8, T9: 1 ms each
  - Not used.
- T10: 1 ms:
  - Set flag to allow reading of compass and A2D sensors in main control loop. This operation takes several ms so a good time to do this is while the Timer Loop is not busy.
- T11-T18: 1 ms each
  - Not used.

Note that there are only 19 "ticks," due to an extra long T1 time, but the total adds to 20 ms.

Ultrasonic Range Timing

The SRF04 is used for all ultrasonic ranging. Since sound travels at approximately 1.125 feet per millisecond, this module requires a fast timer to get accurate measurements. The PIC has a pair of CCP (Capture/Compare/PWM) registers that can be associated with one of the timers to measure a pulse width (among other things). I use one of these CCP registers for the ultrasonic sensors as follows:

1. Raise the SRF04 trigger line high for at least 10 µS. This starts the Ultrasonic Burst.

2. Wait for the echo line to go high, indicating the Burst is done.

3. Start the CCP counter to time the echo response time.

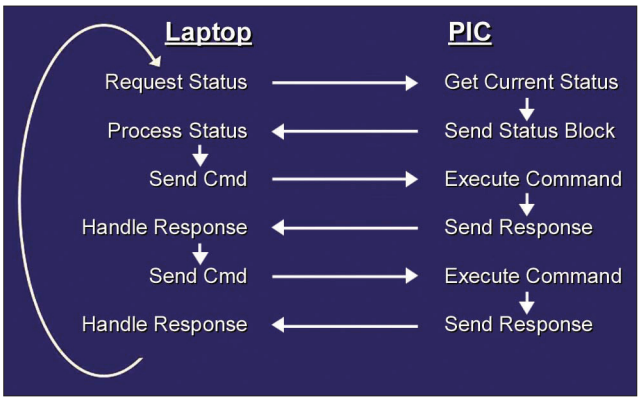
4. Once each ms (in the PIC Timer Loop), check to see if the CCP register is set to non-zero, indicating that an echo was received, and the time of flight is stored in the CCP register. If so, copy the range value to the Status block to be sent to the laptop.

The conversion from timer count to distance (in inches) is done on the laptop, along with all other conversions, such as IR range conversion.

The preceding method works great for one sensor. But how do you get multiple SRF04s to share a single CCP?

The solution turned out to be very easy. As you might recall from Part 1 of this series, each SRF04 has a separate Trigger line that is toggled to start a reading, but the Echo Return lines from all the SRF04s are "wired OR'd" together, using a diode on each sensor (see Figure 2). The software reads each SRF04 in "round robin" sequence by toggling a sensor's Trigger line, waiting for a response on the shared Echo Return line, and then proceeding to the next sensor. It is important during this process that at least 36 ms elapse between each sensor trigger event. (I give it 50 ms just to be sure.) Otherwise, when the next sensor is read, it might inadvertently receive an echo from the previous sensor's ping.

FIGURE 3. Laptop/PIC Communication.



Serial Communication

Refer to Figure 3. The Laptop and PIC communicate via RS-232 Serial. Communication between the Laptop and PIC is always initiated by the Laptop. The Laptop begins by sending the GET_STATUS Command (it does this every 100 ms, or 10 times per second). The PIC responds to this command by building a PIC_STATUS block and sending it back to the laptop.

As explained in last month's article, on the laptop the PIC_STATUS is sent to the Behavioral Modules which, in turn, issue a series of commands. For each command, the PIC may optionally send a response. This response is mostly used for debugging that the command was received and processed correctly.

A lot of data is being sent back and forth between the laptop and PIC. When you start pushing the limits of serial bandwidth as this design does, there are some issues that arise:

Issue 1: The laptop will buffer up commands, so one command can run right into another command. If a byte is dropped, the PIC won't know

where the start of each command is, and will misread the next command. This can be disastrous if the command was supposed to be stop, but was read as "go real fast!" The way I solved this issue was to provide sync characters and termination characters in the command.

The resulting command is structured as follows. (I had planned to replace the Termination Character with a checksum, but never got around to it, and have not found it necessary so far.)

Byte Command
1. Sync0 (I use 0xE5)
2. Sync1 (I use 0x5F)
3. Command Byte
4. Parameter 1
5. Parameter 2
6. Parameter 3
7. Parameter 4
8. Termination Character (I use 0xC4)

If one of the framing characters (Sync0, Sync1, or Term) are not received, the command is discarded, and the serial bit stream is scanned until the next valid Sync0 is found.

Issue 2: The PIC has a very small receive buffer (typically three bytes), which will overflow if not handled quickly enough. This limitation is overcome by using the Serial Read Data Interrupt, INT_RDA. As soon as incoming serial bytes are received, they are parsed and placed into the Incoming Command structure by the read data Interrupt Service Routine (ISR), CheckForSerialData().

However, there is one "gotcha." The PIC does not handle "nested interrupts." For example, if the PIC Timer Loop is handling the 1 ms interrupt, and a serial byte comes in, the INT_RDA will never trigger. Therefore, I call the CheckForSerialData() function explicitly at the end of each Timer0 interrupt, assuring the receive buffer is checked at least once every millisecond.

Issue 3: No matter what you do, sometimes commands are dropped. My robot used to get stuck at times because the PIC never got the next move command. To resolve this, I tried a number of command retry schemes, (some quite convoluted) and finally found one that works quite well.

The key was in realizing that commands to the PIC have a limited life. A command to turn 10 degrees might be replaced half a second later with a command to turn 15 degrees, so there's no sense retrying the first command if it has been dropped. On the other hand, an urgent Stop command really does need to get through!

The solution to this is that each Status returned from the PIC contains the current state for critical subsystems, such as motor speed and turn. The laptop keeps track of the current desired state, and compares it to the one reported by the PIC. If they do

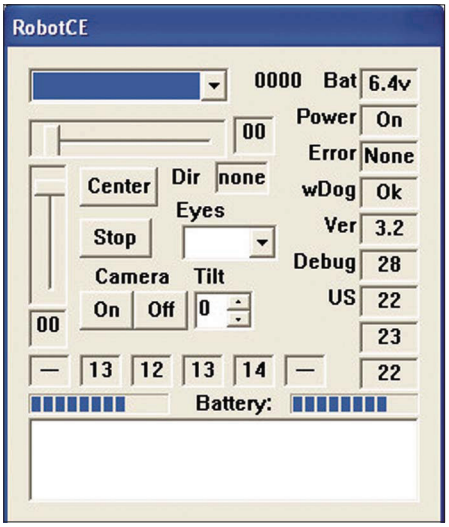


FIGURE 4. HelmetBot iPAQ GUI.

not agree, the laptop re-issues the command. This happens up to 10 times per second, until the PIC confirms that it is in the state that the laptop expects.

Using a PocketPC PDA

There are two main types of Personal Data Assistants (PDAs) typically used for robots: Palm PDAs and PocketPC PDAs. Since I was already using Microsoft Visual C++ for my laptop software, I chose to use iPAQ PocketPC PDAs for my smaller robots, HelmetBot and BugBot. PocketPC PDAs run Microsoft WindowsCE (usually referred to as "WinCE"). The development environment I use for WinCE is Microsoft eMbedded Visual C++.

One really cool feature of this environment (besides the fact that it is just like C++ for Windows) is the remote debugger. If your iPAQ has a wireless card, you can compile a new program, download it to the iPAQ, and step through the code as it executes — even if the robot itself is sitting in the next room!

To get started developing on a PocketPC, I suggest the following: First, buy a used PocketPC on eBay. I have seen the iPaq H5550, which has wireless built in, sell for under $200. If you don't need wireless, older iPAQs, such as the 3765 may sell for under $100, but to add wireless to one of these, you will need a PCMCIA or Compact Flash 802.11 Wireless card, and an iPaq Jacket to plug it into. Make sure the iPaq has a good battery, as it is usually the first thing to go bad (but if it does, you can buy replacements on the Internet).

Next, I highly recommend buying the book *WindowsCE 3.0 Application Programming* by Grattan and Brain (ISBN: 0130255920). This excellent book has a ton of information you need to know for programming WinCE, and Microsoft eMbedded Visual C++ is included on CD-ROM!

You will find that 90% of your robot code is sharable between a laptop robot and PocketPC robot. The main difference is the Graphical User Interface (GUI). See Figure 4 for HelmetBot's GUI. To go from a laptop robot based on Win32 to a WinCE based robot, I usually split the code into a Win32 Client project, which stays on the laptop or PC, and a WinCE Server project, which is used for robot control.

For the robot WinCE Server program, create a new WinCE project, design the GUI, and then start copying the robot engine program files into the new project. If you plan to implement common code for both WinCE and Win32, it's best to get the code working on WinCE first, since some functions available for WinCE are subsets of their Win32 cousins.

For example, if you look at my code, you will notice most strings are surrounded by the "_T()" macro. This macro forces strings to be in Unicode, which is required for all WinCE strings. If you forget this, the code will compile for Win32 just fine, but will not compile for WinCE.

Programming Languages

As mentioned previously in this series, I don't currently use Microsoft .NET programming for either the laptop or WinCE robots. However, that does not mean you can't! I think most of the source code I have posted should port fairly easily to the .Net environment, but I just have not gotten around to it. Also, I know people that are using Java, Visual Basic, etc. Pick a language that works for you!

On the PIC Microcontroller, I use the CCS C compiler (available at **www.ccsinfo.com/pic**). I chose this because I like using C, and it's a fairly inexpensive compiler. The command line compiler works fine, and integrates easily into Microsoft Visual Studio. There are also various PIC Basic and other compilers available, many of them cheap or even free.

Conclusion

The subject of laptop robotics could easily fill a book. In this article, I provided a brief overview of several subjects in which I have seen the most interest. I hope that there is enough information provided to get you started building your own laptop- or PDA-based robot.

As stated previously, both a demo of the Seeker application and the hardware schematic are available on the *SERVO* website (**www.servomagazine.com**). Further, all the source code mentioned is available at my website at **www.shinsel.com/robots** Now, go build that 'bot! SV

ABOUT THE AUTHOR

Dave Shinsel built his first robot in 1980 (yes, before the IBM PC was released) using a 1 MHz 6502 processor and 9K of RAM. He has been a hardware and software engineer for a number of companies including Hughes Aircraft, Epson Printers, Mentor Graphics, and for the last 12 years, Intel Corporation. At Intel, Dave manages a software engineering team for the Consumer Electronics Group in Portland, OR. His degrees in Electrical Engineering and Computer Science are from Cal State Long Beach, CA.