[Part 2]

INTERMEDIATE ROBOTS

Building a Laptop- or PDA-Based Robot

MEET THE 'BOTS

- HelmBot — iPaq PDA Robot (left)
- Seeker — Laptop-based Robo-Magellan Robot (below)

In last month's article, I described the hardware design of a laptop-based robot. Now, let's get into the software. The first thing you should consider is "what do I want to be able to do with this software?"

Software Requirements

The requirements for my design were:

- Must communicate reliably with an embedded microcontroller.
- Must have extensible behavior programming capabilities.
- Must have an easily customizable User Interface, to aid in troubleshooting (see Figure 1).
- Must have wireless remote operation and the ability to debug from another computer.
- Should have a Path entry for following a complex course.
- Should have Graphical mapping for navigation debugging.
- Should have a simulator, to allow testing some of the algorithms without having to actually run the robot.

Notice that the last three items are "should haves." I could still have built a successful Robo-Magellan robot without these, but they made developing a lot easier!

BY DAVE SHINSEL

High Level Design

Refer to Figure 2. The design is built around a robot behavior and control engine that performs the following:

1. Accepts user commands from either a local or a remote User Interface.

2. Processes sensory data passed to it from the PIC microcontroller or other inputs (such as a video camera).

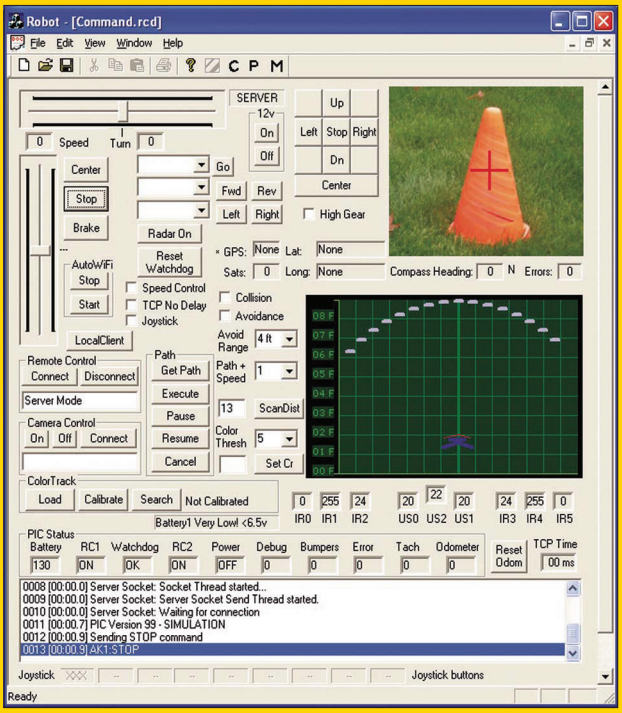
3. Based upon the requested behavior and the sensory data, determines the appropriate action, and issues commands to send to the PIC which are, in turn, sent to the hardware.

4. Sends status and debug information back to the User Interface, to aid in navigation (in the case of remote control) and debug.

A QUICK WORD ABOUT THREADS

Win32 (Windows NT and XP) provides a threading model based upon Preemptive Multitasking. This allows several tasks to all appear to run at the same time. The processor is actually letting each run for a short time-slice, and then switches to the next task. Inside a single program, these tasks are called threads. Threads are handy because they can block while waiting for some event to happen, and not cause the whole system to stop responding. Threads require almost no attention from the CPU when blocked, so performance stays high. Threads make writing complex, multitasking programs much easier.

FIGURE 1. Seeker Command GUI



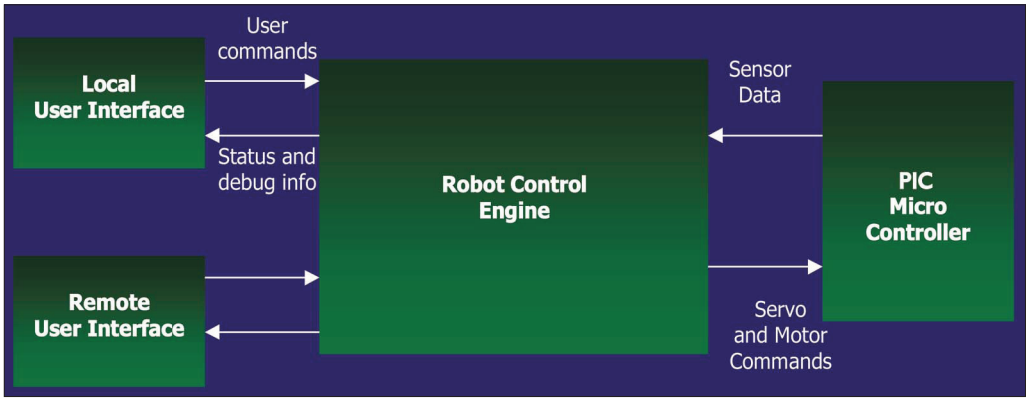


FIGURE 2. Robot Control High-level Overview

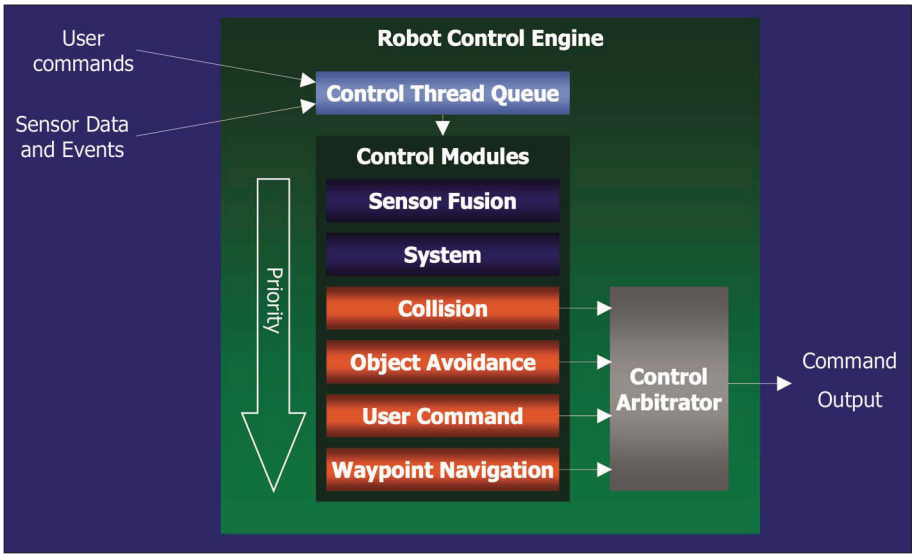
Control Engine

Refer to Figure 3. All user commands and sensory data are fed into the control engine. This engine uses a modified subsumption architecture, first proposed by Rodney Brooks at MIT. This behavioral programming model is frequently used in robots. The basic idea is this: Sensory and command data is sent to all modules and each module makes its own decisions on what the output to the motors or other devices should be. Modules are assigned relative priority, and an arbitrator blocks commands from lower-priority modules.

Let's try an example. Say the Object Avoidance Module sends a command to "turn left" while the Navigation Module sends a command to "turn right." Since the Object Avoidance Module has priority over the Navigation Module, the arbitrator will allow only the turn left command to get through.

There are times, however, when it might be desirable for a lower priority module to suppress commands from a higher priority module. For example, in the Robo-Magellan contest, robots must actually touch cones on the course to get extra points. With no way to suppress the behavior, the Object Avoidance Module would always cause the robot to swerve away from the cone! So, when the Navigation Module has locked on to a cone and is heading towards it, it will suppress the Object Avoidance and Collision modules until the cone is reached.

FIGURE 3. Robot Control Engine



Control Modules

The control engine features a pluggable architecture. New modules may be fairly easily inserted into the control engine, and the control thread will send commands and sensor data to the new module. I found the pluggable architecture so useful that I even used it for two processing modules that take care of non-behavioral tasks; the Sensor Fusion and System modules.

The highest priority module is the **Sensor Fusion Module**. This module pre-processes data that is then fed to all the other modules. It scales sensor data from raw values to a standard unit (inches), combines inputs to calculate interesting data (such as the nearest threat), and updates the robot's location on its internal map.

The **System Module** mostly tracks the health of the PIC microcontroller. It posts error messages from the PIC, reports the PIC version info, and sends "snapshots" of sensory data to the User Interface for debug purposes.

The **Collision Module** is the highest priority behavior module. The Collision Module monitors sensor data for range values that fall below specific thresholds. Examples of this are IR sensors reporting an object less than four inches away, or bumper switches being pressed. Upon triggering, the Collision Module's internal state machine takes over control of the motors and steering. It will retain control until it has completed a sequence of steps which attempt to clear the robot from the obstruction. Once the behavior has completed, control is returned to other modules.

The next highest priority is given to the **Object Avoidance Module**. It is interesting to note that this module receives exactly the same sensor data as the Collision Module, but acts differently upon the data it receives. The Object Avoidance Module keeps moving the robot forward, but will attempt to steer the robot around objects in its path.

The **User Command Module** processes all commands from the User Interface. This module allows the user to manually drive the robot, pan the camera, etc. Note that a user command to drive into a tree would be overridden by either the Object Avoidance or Collision modules. This is handy when remote controlling the robot over the Internet.

Robot **Waypoint Navigation** is the lowest priority module. When no other modules have asserted control, the Waypoint Navigation Module is free to drive the robot to its destination. If one of the other modules forces the robot off course (for example, to avoid an object), the navigation module will recalculate its route and resume heading to the next waypoint.

Internal Map

The robot software utilizes an internal map based upon a grid that is just over one mile square. All coordinates are expressed in inches. Since a 16-bit word can hold a number up to 65,535, a single 32-bit value can hold the X and Y coordinates for any location within a 1.03 square mile area.

The map Origin (coordinate 0,0) is in the South West corner of the map. The map may be anchored to the real world by associating any one point on the map with a GPS coordinate. The Origin is automatically calculated as an offset from that point. It is interesting to note, however, that the map is only required to be anchored to real world coordinates if GPS is being used. For the SRS Robo-Magellan contest in Seattle, I did not use GPS at all. Instead, I chose an arbitrary start location set to (1000, 1000). All real world features — such as cone placements and obstacles — are calculated automatically as offsets from that point. The only critical requirement is that all interesting information must stay within the one square mile boundary of (0,0) to (65,535, 65,535).

Navigation

There are many types of navigation that might be interesting to a robot builder. For the purposes of this article, I will focus on pre-planned navigation. In pre-planned navigation, the robot is provided with a starting location, final destination, and *Waypoints* along the way.

Using an internal map, the robot must be able to find its way from one Waypoint to the next, avoiding obstacles along the way. I use the term *Segment* to refer to the connecting lines between Waypoints, and the term Path to represent the full collection of Waypoints connected by Segments (see Figure 4).

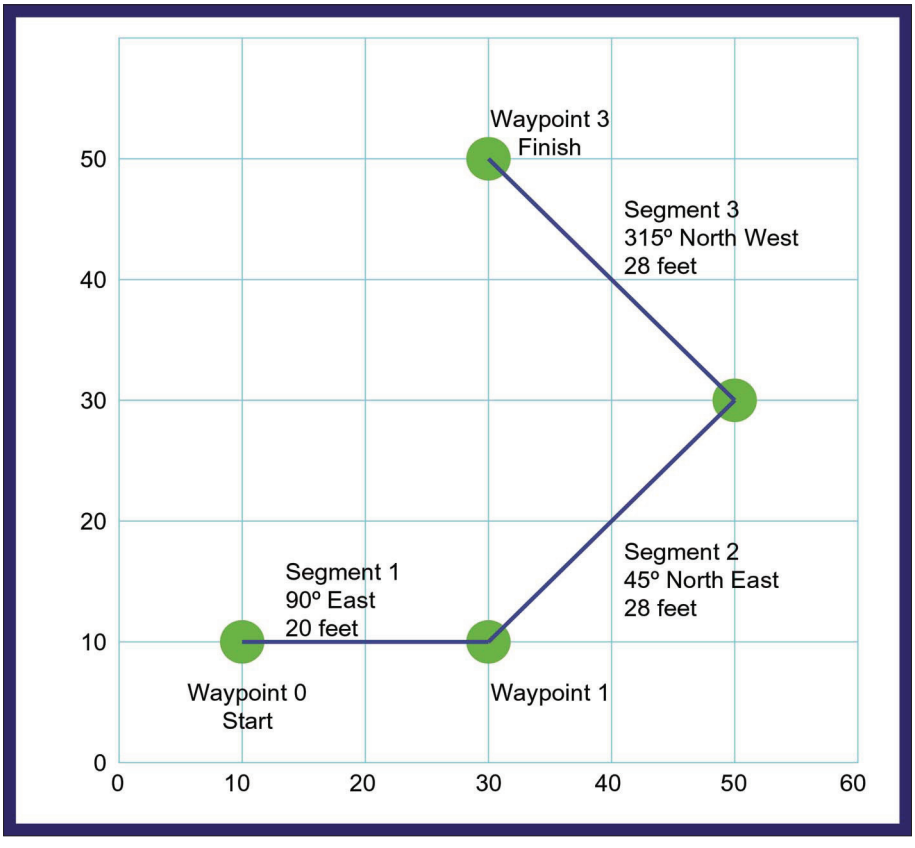
The Path Entry Dialog (see Figure 5) is used to enter the various Segments the robot is expected to follow. First, the Start Waypoint is entered in absolute coordinates. By default, this is (1000, 1000). Next, absolute direction (in degrees from North) and distance (in feet and inches) is entered for each segment. Once all of the segments are entered, pressing the "ReCalculate All" button will calculate the absolute location of all the other Waypoints.

Waypoints are associated with an (X,Y) location coordinate. In addition, a Waypoint may optionally have a

MORE ABOUT SUBSUMPTION ARCHITECTURE

Subsumption Architecture is much like your nervous system. Your high-level thought process may instruct your hand to pick up a pan from the stove. However, your reflex behavior upon touching the hot pan will override the initial command, because the reflex to not get burned has higher priority. If you decide you really want to pick up the pan anyway, you can suppress your natural reaction and pick up the pan. Reflex behaviors are high priority; they get processed even when your mind is on something else. High-level thinking takes longer to process stimulus data and decide what to do. But, once a decision has been made, the lower priority behavior can temporarily suppress reflex behaviors. A good example of this is when you consciously decide to hold your breath.

FIGURE 4. Internal Map — Segments and Waypoints



ATTRIBUTES

Waypoint Attributes:
- Waypoint location (X,Y)
- Landmark Type
  - None
  - Cone
  - Drop Off
  - Pole
  - Tree
  - Wall
- Landmark Range (in inches)
- Landmark Direction (in degrees)

Segment Attributes:
- From and To Waypoints
- Distance (length of segment)
- Direction
- Speed
- Behavior
  - Compass
  - Compass+GPS
  - Follow Wall
  - Follow Dropoff
  - Follow Hall
  - Find Doorway
- Optional: Follow Wall parameters
  - Left/Right
  - Distance

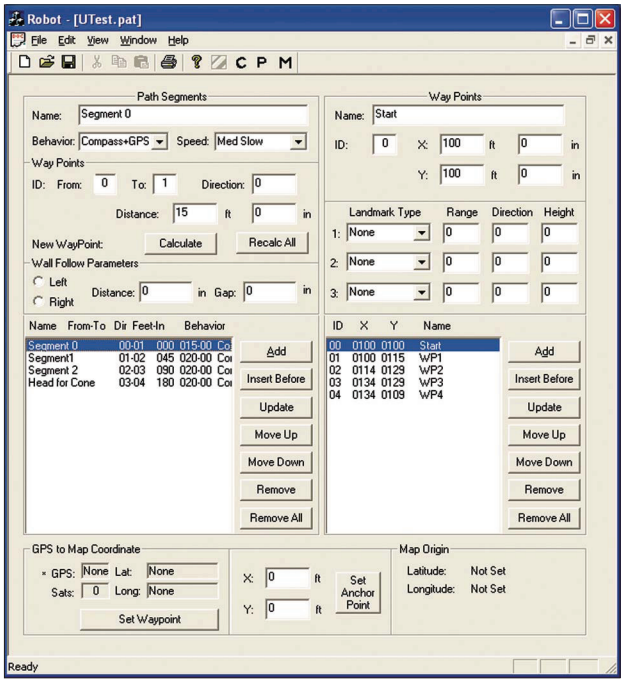


FIGURE 5. Path Entry Dialog

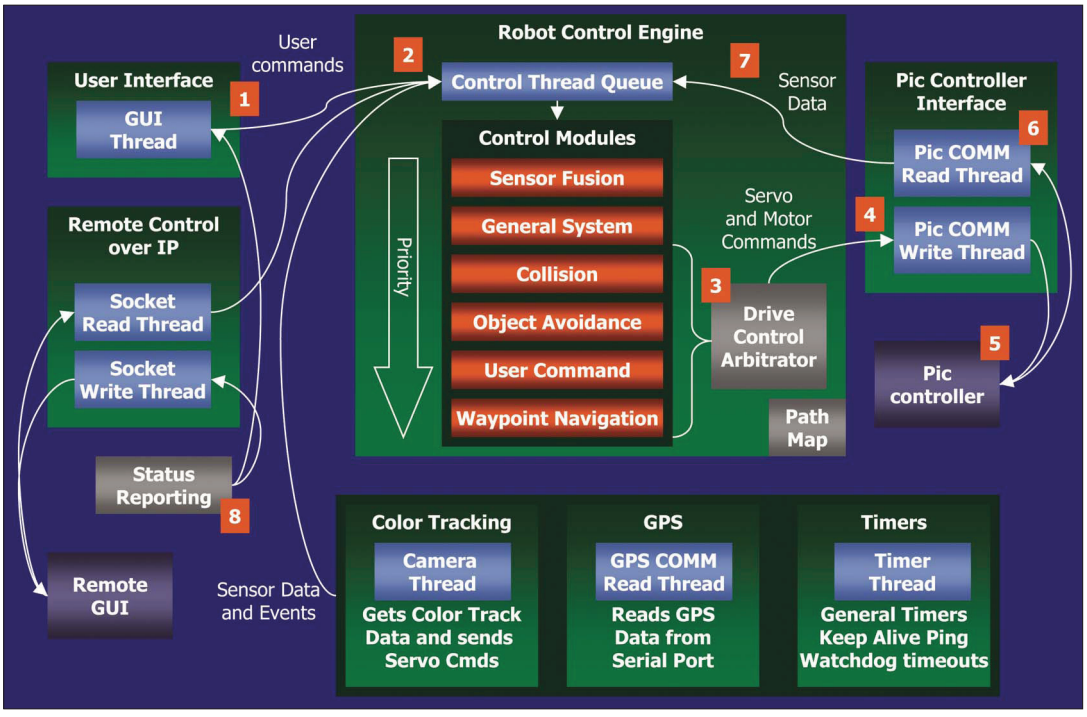
Landmark Type associated with it. If the Landmark Type is set to "Cone," the robot camera will actively search for a cone while heading for the landmark. If the Landmark Type is "Pole" or "Tree," the robot will search for a narrow object in the right location, and head for it. When the robot reaches the specified distance from the landmark, it will consider the Waypoint reached, and head for the next Waypoint. Landmarks are very useful for correcting errors that build up as the robot navigates along the path.

Segments can have attributes as well, which help the robot successfully navigate specific portions of the course. For example, if the Segment Behavior is "Follow Wall," the robot will drive a path parallel to the wall. Additional parameters are provided to indicate left or right side and distance to maintain from the wall. A behavior of "Follow Hall" or "Enter Doorway" will cause the robot to search for an open area between two objects, and head for that middle area between them.

Simulator

The Robot Control Software has a built-in simulator. The program is available for download from the *SERVO* website (**www.servomagazine.com**). This will give you a good feel for the robot's capabilities prior to jumping into the source code. Once you download the program, simply create a new Path and enter some data. Next, create a new Map. If you tell the robot to move, the robot icon on the map will move in response to your commands. If you tell the robot to follow the path, it will to the best of its ability. Try it!

FIGURE 6. Control Software Data Flow



Connecting it All Together

Refer to Figure 6.

This detailed block diagram of the full system shows how all the parts of the system interact with each other. First, notice the nine threads indicated by blue boxes. Each of these threads block while waiting for some event to happen. Most are tied to a "command queue." When one thread wants to send data to another thread, it puts the data in the other thread's command queue.

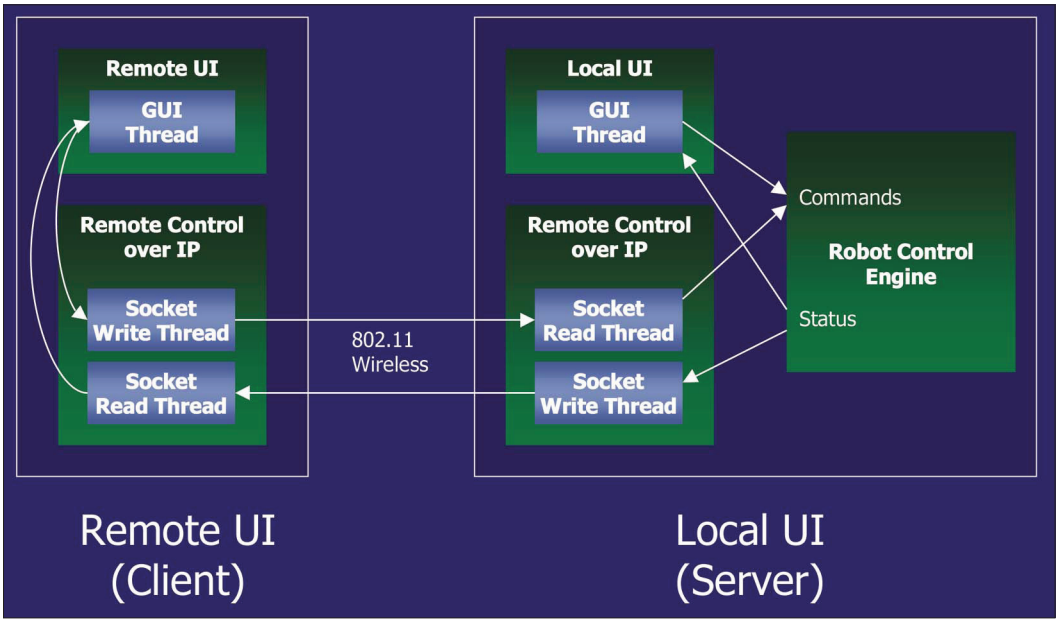


FIGURE 7. Remote User Interface

Let's walk through the model. Assume the user presses a button to increase the robot speed. The User Interface GUI thread (1) will get the User Event (button pressed), and place a command into the Control Thread Queue (2). The Control Thread will pull the "set speed" command from the queue, and pass it to each of the Control Modules. The User Command Module will act on the command and issue a change speed command to the Drive Control Arbitrator (3). Assuming no other module has issued a higher priority command, the Arbitrator will send the command to the serial queue in the PIC Comm Write Thread. The PIC Comm Write Thread (4) will send the command to the PIC controller via RS-232. The PIC controller (5) will act on the command, and adjust the motor speed.

Now, let's assume an object has been detected ahead by one of the sensors. The PIC controller (5) will send a status packet to the host computer. The PIC Comm Read Thread (6) will read the status packet from the serial port, and post it to the Control Thread (7). The status is sent to each of the Control modules in turn. If the object is close enough, the Collision or Object Avoidance module may send a new command (perhaps Stop or Turn), to the Drive Control Arbitrator, and the cycle repeats.

There is a global Status Reporting module (8) that is accessible to all threads. Debug messages, sensor status, and other useful information are sent to the GUI thread by the Status Reporting module.

In addition to communication from the PIC microcontroller, there are Camera, GPS, and Timer modules that also send their data to the Control Engine. The interfaces to these will be discussed in Part 3 of this series.

Remote Control and Monitoring

One nice feature of this software is that the robot can be controlled and monitored from a remote location, such as over the Internet or from a local PC using 802.11 wireless. When the Robot Control Software is compiled, there is a "build switch" that allows either the Server or Client version to be built. Both the client and

server versions are built from the same source, eliminating the need to maintain two separate projects.

Figure 7 shows how the Client (remote PC) and the Server (the laptop on the robot) communicate. Standard Windows WinSock is used to create sockets on both machines, and threads are created to read and write to the opposite machine's socket. When a remote (client) is connected to the robot, all the status and debug information is sent to the client, and all commands from the Client UI are sent to the robot.

For remote control and "remote presence," audio and video can also be sent. Currently, Microsoft Netmeeting is used to handle the audio and video, but other mechanisms can be used, as well.

For detailed debugging (stepping through code), I often use the Microsoft Remote Desktop. This is a great feature of Windows XP that allows you to take control of a computer from a remote location. However, it's not very good during actual robot operation, because the overhead can slow the robot responses down too much. During robot operation, I rely more on the remote log information that I receive on the client to understand what the robot is doing, and why.

Conclusion

Hopefully, this article has provided you with a good overview of the design approach I have taken for my robots. As mentioned earlier, a binary version of the Robot Control Software is available for download from the *SERVO* website, so you can get a good feel for the robot's capabilities.

The source code to all of the software discussed in this article is available from my website at **www.shinsel.com/robots**

To compile the software without modification, you will need Microsoft Visual C 6.0 (MSVC6). I believe you can also use Microsoft .Net, but I have not tried it yet. And, of course, since it is source code, you can convert to Linux, or whatever you want!

Next Month

In Part 3 of this series, I will go into detail about the PIC microcontroller software, how the Laptop and PIC communicate, GPS, and color tracking with a USB camera. Until then, the thought for this month is: "Real programmers don't use comments. If the code was hard to write, it should be hard to understand." SV

ABOUT THE AUTHOR

Dave Shinsel 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.