

## ENHANCED SMALL MOBILE PLATFORMS CONTROLLED BY HAND-HELD COMPUTERS

Jorge Blanch and Sabri Tosunoglu (346-68)

Florida International University  
Department of Mechanical Engineering  
10555 West Flagler Street  
Miami, Florida 33174

**Abstract** – *Advances in computer technology make it possible to have mini-computers on smaller mobile platforms than previously available. Since hand-held PCs are stand-alone products, they incorporate faster processors relatively large quantities of memory, and a touch screen, which serves as both, input and output device. These and other features can be used to creatively enhance mobile platform capabilities. Initially pioneered at Carnegie Mellon University, this paper looks at the advantages of incorporating such devices as the controller of a robotic platform.*

**Keywords** – Robotic platform, mobile platform, PDA, handheld-PC, micro-controller.

### INTRODUCTION

A great deal of research as well as courses in mobile robotics at universities has been carried out in small, affordable platforms [1, 6]. Popular controllers for such platforms were Basic stamps, Handy-boards, Mini-boards and other microcontrollers based on Motorola's HC11 [2]. Because of size and power limitations, these robots had no access to more powerful microprocessor-based controllers (i.e., laptops or computers.) Recent advances in consumer electronics have led to the development of the so-called PDAs and hand-held computers. These new computing devices have quasi-PC capabilities while they are small enough that they fit on small mobile platforms. The combination of a more powerful processor and larger amounts of memory allows for far more complex controlling programs as well as the use of higher-level languages like C++ or Java. The built-in touch-screen display provides easy and friendly user interface.

Handheld devices also incorporate some of the latest advances in computer technology, which can be used to add functionality to the controlled platforms.

*“New capabilities are driving demand for PDAs and taking them from the consumer market into the more demanding business market. [...] With this expansion will come demand for higher levels of functionality and interoperability [...]”* [3]

As handheld devices become more and more popular, companies develop better and faster handhelds that incorporate cutting edge technology, both in hardware and software. For example, handhelds are marketed as computer companions, so that connection interfaces and protocols have been resolved; but, because wireless communication is more comfortable for users, companies are incorporating wireless capabilities to their PDA designs. Software enhancements in current handhelds include media playback, but in the near future, this will probably evolve into media streaming (through the built-in wireless communication, of course).



Figure 1. Compaq's iPAQ™, one of the more popular handheld-PCs, runs windows OS with media playback and wireless communication capability.

## ENHANCED MOBILE PLATFORMS

Handheld-controlled platforms can be classified into two types, those who move up in the computational power scale, and those who move down in the cost scale:

- Some systems may see their portable PC replaced by a handheld-PC for a fraction of the cost. Replacing the portable PC alone is a significant reduction in cost. Also, if starting from the design stages, the smaller platform required for the handheld-PC will probably be a cheaper construction. Of course, this set-up assumes that the handheld will be powerful enough for the computing requirements of the robotic platform.
- Other platform designs will be subject to an upgrade from a microcontroller to a microprocessor-controller [2]. Replacing an old microcontroller by a handheld device reflects a considerable increment in computing resources. These will be the so-called enhanced platforms that will adopt the advantages of handheld PC design in a mobile robotic platform.

Mobile platforms that will be significantly rewarded by the addition of a handheld-PC are primarily those that have frequent user interaction [5]. Systems that interact with users for data input or output will be able to profit from the output screens that handhelds have, which allow to display more information and with better resolution than many previous LCD displays within the same size category. Since the display is also a touch screen, it can be used as an input device at runtime, without the need of additional hardware.

Another reason for using a handheld PC as a platform controller would be to take advantage of its wireless communications packages. The built in IR available on many units is directional and does not have a strong signal, but it can still be used to send/receive signals across small distances [5]. More promising wireless communication capabilities come in the form of wireless LAN for computer network communications, and, soon enough, bluetooth for peer to peer communications. Once bluetooth becomes available to handhelds and other equipment, a platform will be able to communicate much more

easily with elements in its environment (and not just computers or other robots.)

All handhelds on the market have anywhere between 2 MB and 64 MB of RAM, which is a considerable upgrade from most existing microcontrollers. Such amount of memory is more than enough to develop comprehensive control algorithms in a higher-level language. Before now, C++ and Java programming could only be implemented on RAM-abundant laptop computers. Besides the built-in memory, most handheld devices have memory expansion cards for “unlimited” storage.

Handheld devices offer therefore, enough power and autonomy to implement sophisticated control algorithms. At the same time, a communications package allows for any off-platform computation that might be required.

## HARDWARE SETUP

One of the few shortcomings of using a handheld computing platform over a microcontroller board is that the handheld cannot directly control any servos or read from the sensors, a separate servo controller board is needed [2]. Some of the servo controller boards available in the market are Pontech’s SV203 or Lynxmotion’s SSC and SSC-12 [2]. An alternative to replacing the current microcontroller would be to add the handheld device without taking out the microcontroller.

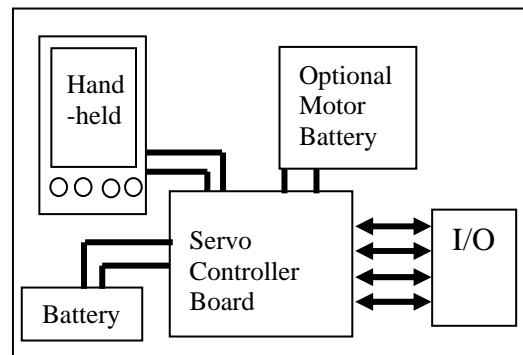


Figure 2. Handheld-controlled platform schematic.

For existing platforms, it may be faster and easier just to reprogram the old microcontroller to transmit data to and from the handheld. In other words, have the handheld use microcontroller board as the interface board. The advantage of

using this set up is that most microcontrollers have an excellent array of servo outputs and sensor inputs that the handheld could use.

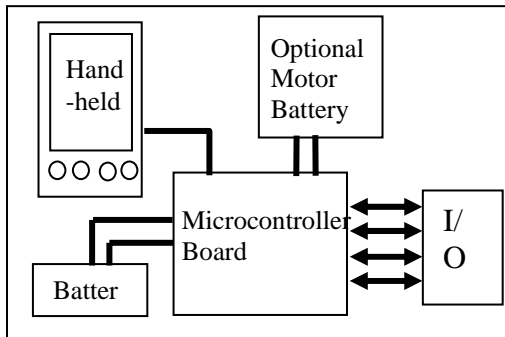


Figure 3. Microcontroller-based platform controlled by a handheld.

## SOFTWARE SETUP

The flexibility of the platform/software combinations is excellent, and can be chosen in function of past or future requirements. Several operating systems can be chosen from: Palm-Os, Windows-CE and Linux among others; and all operating systems support one or another higher-level programming language (C++, Java.) Since the use of higher-level languages allows for object-oriented programming, it would be foolish not to take advantage of the use of it. As a matter of fact, the use of classes modularizes the software architecture, which, if done properly means that some of the parts can be easily portable to other mobile platforms. To prove this, we intend to port the software created by Greg Reshko and Joshua M. Mouch to control the Palm Pilot Robot Kit at Carnegie Mellon University [1]. While keeping the original concept intact, we plan modify the classes as little as possible so that they can control a rover platform (the original PPRK has a holonomic design.) Fortunately, we share several parts (servos, sensor, and interface board) in the design and thus the defining classes for these components will need little, if any, modification. The classes that require the most significant modifications are the “driver” classes and the “movement” classes.

There are two driver classes: the Servo Driver and the Sensor Driver. The Servo Driver class initializes all the servos that belong to the robot, while the Sensor Driver does the same for the

sensors. Initializing a servo requires assigning an interface board and an output channel, a sensor, on the other hand, requires an input channel (if the platform has many servos, it may need several interface boards.) The driver class also needs to know the position of the elements (servos and sensors) within the robot and their orientation so it can relate each element with the physical world. The positions need to be known for coordinating individual servo velocities to obtain the desired platform movement or for calculating the direction and distance of detected objects.

Servo Driver has a list of servos, with their board, their port, and their coordinates with respect to the “center” of the platform. The constructor initializes all of them:

```
servoDriver::ServoDriver() {
    for( int r = 0; r < NumServos; r++ ) {
        servo[ r ] = new Servo( ServoBoard[ r ],
                               ServoPort [ r ] );}
}
```

Rovers are driven by actuating both their wheels at the same time, and steered by the difference in velocities between wheels:

```
Void servoDriver::RoverDrive(double R, double L)
{
    theRobot.updateLocation();
    servo[ 0 ]->Drive( servo[ 0 ]->RevToVal( R ));
    servo[ 1 ]->Drive( servo[ 1 ]->RevToVal( L ));}
}
```

Here, the Servo Driver Class is calling twice a function of the Servo Class to “Drive( )” each servo at the value corresponding with the desired velocity (R or L). The Planner Class is in charge of updating the location of the robot, if there is no Planner Class, updating is not necessary.

Initialization of the sensors is similar to the initialization of the servos:

```
sensorDriver::SensorDriver() {
    eventState = 1;
    for( int r = 0; r < NumItr; r++ )
        ptrRobotItr[ r ] = new SensorItr( IrBoard[ r ],
                                           IrPort[ r ], IrLocX[ r ], IrLocY[ r ],
                                           IrLocTheta[ r ]);
    ptrRobotBp[ 0 ] = new SensorBp( BpBoard,
                                    BpPort ) }
}
```

Each class of sensor has its own constructor. Because all bumpers are connected in a resistor ladder, they all initialize as one, actual obstacle position has to be determined elsewhere within the

bumper class. Because sensors have to be continuously “listening” to the environment, they need an overhead function that reads new sensor values and stores them. The overhead function in Sensor Driver Class needs to call the individual overhead functions from all the sensors:

```
Void sensorDriver::Overhead() {
    switch(eventState++) {
        case 1: ptrRobotlr[ 0 ]->Overhead( );
            break;
        case 2: ptrRobotlr[ 1 ]->Overhead( );
            break;
        case 3: ptrRobotlr[ 2 ]->Overhead( );
            break;
        case 4: ptrRobotBp[ 0 ]->Overhead( );
            break;
        default: eventState = 1; } }
```

For each infrared sensor, the overhead function takes a reading, transforms the value into a distance and, through the position and orientation of the sensor (declared at initialization), calculates the relative position of the obstacle. If the robot keeps track of its current position and orientation, then the global position of the obstacle is known. Finally, overhead stores the position into an array; if the array becomes too large, overhead has to accommodate by either pushing new values and deleting older ones or enlarging the array.

The third main class that will need to be modified is the “Holonomic Movement” class. This class is in charge of driving the platform, it knows the platform’s velocity and orientation, and can move accordingly. The now renamed “Rover Movement” class should also include basic movement commands for easy control, for example:

```
Void roverMovement::roverLPivot( double speed ){
    speed = base * speed / wheelRad;
    //change  $\theta_{ROBOT}$  to  $\theta_{WHEEL}$ 
    theRobot.servoDriver.RoverDrive( speed, 0 ); }
```

```
Void roverMovement::roverRTurn(double speed ) {
    speed = base * speed / wheelRad;
    //change  $\theta_{ROBOT}$  to  $\theta_{WHEEL}$ 
    theRobot.servoDriver.RoverDrive(-speed,
                                     speed)}
```

```
Void roverMovement::stop( ) {
    theRobot.servoDriver.RoverDrive( 0, 0 ); }
```

As can be seen, servo motion is ultimately delegated to the Servo Driver Class, but first, the Movement Class must calculate the individual servo velocities, depending on the desired motion. Changing from a rover platform to any other platform should need similar modifications: drivers need to be updated for number, position and type of servos and sensors, and the “movement” class has to be changed according to the platform design. By breaking the programming down into classes, some pieces can be directly exchanged, between robots, specially the “lower classes”; the higher the class, the more complex it is and the more likely it is it will have to be modified.

## PLATFORM DESIGN

Since our platform was initially based on the PPRK, we adopted many of the same components that they used, but instead of a holonomic platform, a simpler differential-drive rover design was implemented.

At any point, the position of the robot  $P_0$  is given by  $[\mathbf{R} \ \phi]^T$  or  $[x \ y \ \phi]^T$  and the velocity is given by  $[\mathbf{v} \ \dot{\phi}]^T$  or  $[v_x \ v_y \ \dot{\phi}]^T$ . The angular velocity of a wheel, in a no-slip condition, generates a linear velocity depending on its radius,  $v = r \cdot \omega$ . The velocities of the wheels relate to the velocity of the platform through the physical properties of the platform:

$$\begin{bmatrix} v \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1/b & -1/b \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$\text{But } \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix} \cdot \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}$$

$$\text{Thus, } \begin{bmatrix} v \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} r & r \\ r/b & -r/b \end{bmatrix} \cdot \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix} \text{ or, inversely}$$

$$\begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix} = \begin{bmatrix} 1/r & b/r \\ 1/r & -b/r \end{bmatrix} \cdot \begin{bmatrix} v \\ \dot{\phi} \end{bmatrix}$$

This relationship between the wheel's angular velocity and the platform's velocity is used in the Class "Rover Movement" to adjust the wheels' velocity according to the desired platform velocity.

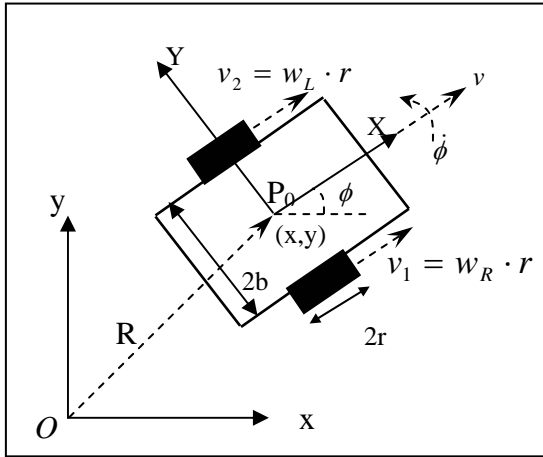


Figure 4. Position/Velocity Rover model.

To be able to use the range finding sensors for relevant data compilation, the sensors have to be well defined within the robotic frame. The sensor reads a signal corresponding to a distance 'd' to the obstacle. The relative position of the obstacle with respect to the platform is  $\vec{r} = \vec{t} + \vec{d}$  and the global position is  $\vec{T} = \vec{R} + \vec{r}$ . Or, by components:

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} d \cdot \cos \theta \\ d \cdot \sin \theta \end{bmatrix}$$

$$\begin{bmatrix} T_x \\ T_y \end{bmatrix} = \begin{bmatrix} R_x \\ R_y \end{bmatrix} + \begin{bmatrix} r_x \\ r_y \end{bmatrix}$$

But  $\mathbf{r}$  has to be transformed to global coordinates first:

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = \begin{bmatrix} (t_x + d \cdot \cos \theta) \cdot \cos \phi \\ (t_y + d \cdot \sin \theta) \cdot \sin \phi \end{bmatrix}$$

Finally,

$$\begin{bmatrix} T_x \\ T_y \end{bmatrix} = \begin{bmatrix} R_x + (t_x + d \cdot \cos \theta) \cdot \cos \phi \\ R_y + (t_y + d \cdot \sin \theta) \cdot \sin \phi \end{bmatrix}$$

The Sensor Class uses this transformation to calculate and store the position of obstacles detected by the range sensor.

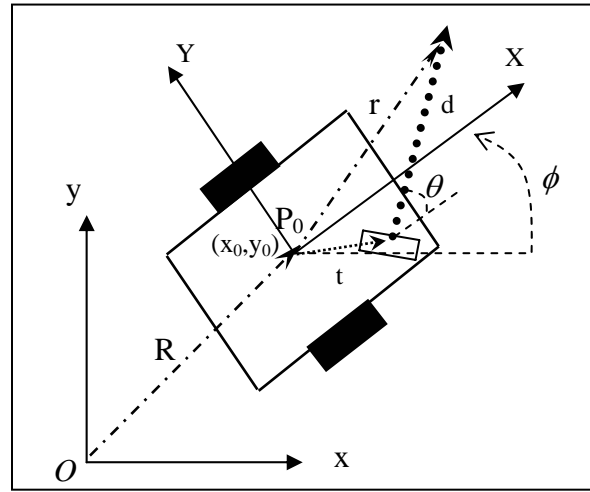


Figure 5. Sensor-Reading model.

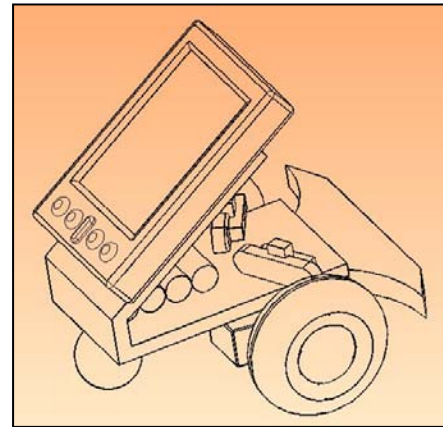


Figure 6. Rover Design.

For our rover platform, a Palm IIIx uses Pontech's® SV203 board to control two Futaba-type modified servos (to rotate continuously). Two Sharp GP2D12 analog infrared rangars and three reset switches complete the sensor package. This intentionally simple design will be used to test the different components and to acquire general knowledge in systems integration. Some interesting ideas were implemented with the sensor configuration. The bumper (front end of the rover) is suspended on rubber bands, which make it return to the "neutral" after the bumper is released. Depending on the point of contact,

different reset switches are triggered, which the robot interprets as left, right or center through the software. It has been observed that only two and not three switches are needed for accurate bumper sensing. The infrared sensors are also being used to test a novel position arrangement. Usually, sensors are placed in the outermost part of the robot, probably to avoid any interference. In our case, the infrared sensors were placed away from the edge of the platform. To maximize the distance to the edge, they were crossed so that the sensor on the right is aiming to the left and vice-versa. Consequently, their infrared signal paths intersect, which will have to be examined for possible interference in the sensor readings. The reason behind driving back the position of the sensors is that most range-finding sensors have trouble with short distances (hence the typical use of bumper sensors). With this “crossed positioning” the edge is as far as 6 cm from the sensor, which means that the “dead space” in front of a range finder is reduced from nine to three centimeters. With just over an inch of dead sensing space, navigation with the sensors can be much more reliable than previously possible.

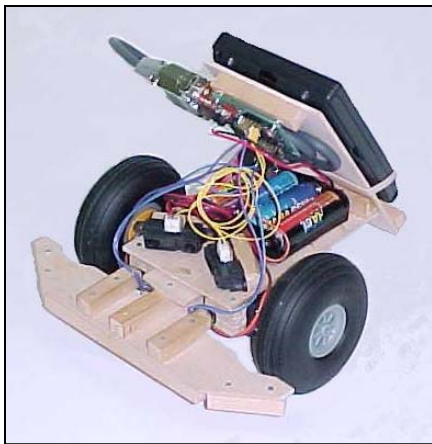


Figure 7. Rover.

Future additions to the platform can include a rear-mounted bumper switch, a forward-looking ultrasound range finder (for comparison purposes) and digital compass. Maybe the most interesting sensor will be the digital compass, which will allow for more precise navigation and sensor orientation.

## CONCLUSION

In this paper we have addressed the use of modern handheld computing devices as controllers of mobile robot platforms. It is predicted that the advances in computer technology have driven and will keep pushing the boundaries of computer miniaturization for some time to come. To test the utility and versatility of using handhelds as controllers, we have assembled a simple rover platform that utilizes a Palm Pilot as a controller. With a working platform recently completed, research will next focus on proper software structuring and testing.

## REFERENCES

- [1] Main site for Carnegie Mellon's PPRK: <http://www.cs.cmu.edu/~pprk/index.html>.
- [2] J. Blanch & S. Tosunoglu (2001): Hand-Held Computers as Mobile Platform controllers. *Florida Conference on Recent Advances in Robotics*.
- [3] Toshiba Press Release. Jul. 16, 2001: [http://www.toshiba.co.jp/about/press/2001\\_07/pr1601.htm](http://www.toshiba.co.jp/about/press/2001_07/pr1601.htm).
- [4] T. Fukao, H. Nakagawa, N. Adachi (1999): Adaptive Tracking Control of a Nonholonomic Mobile Robot. *IEEE Transactions on Robotics and Automation*. Volume 16, Number 5, pp. 609-614.
- [5] N. Tschichold-Gurman, S.J. Vestli, G. Schweitzer (2001): The service robot MOPS: First Operating Experiences. *Robotics and Autonomous Systems*. Volume 34 (pp. 165-173.) Elsevier Science B.V. <http://www.elsevier.nl/locate/robot>.
- [6] University of Florida. Intelligent machines course: <http://www.mil.ufl.edu/imdl>
- [7] MIT 6.270 robot-building course: <http://web.mit.edu/6.270/www/about>