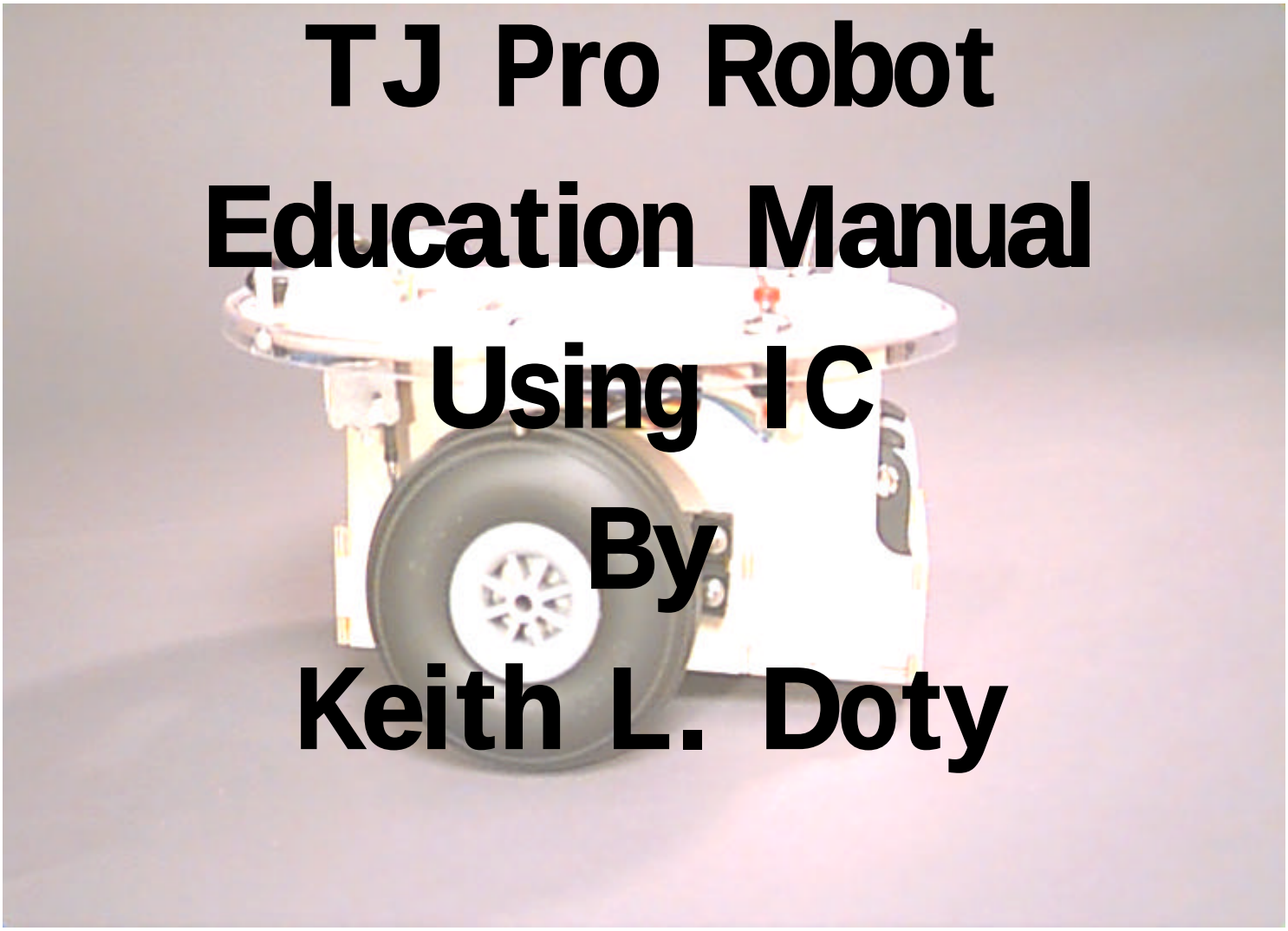
TJ Pro Robot Education Manual Using IC



By

Keith L. Doty

Gainesville, Florida Phone 407-672-6780

AGREEMENT

This is a legal agreement between you, the end user, and Mekatronix™. If you do not agree to the terms of this Agreement, please promptly return the purchased product for a full refund.

1. **Copy Restrictions. N**o part of any Mekatronix™ document may be reproduced in any form without written permission of Mekatronix™. For example, Mekatronix™ does not grant the right to make derivative works based on these documents without written consent.

2. **Software License.** Mekatronix™ software is licensed and not sold. Software documentation is licensed to you by Mekatronix™, the licensor and a corporation under the laws of Florida. Mekatronix™ does not assume and shall have no obligation or liability to you under this license agreement. You own the diskettes on which the software is recorded but Mekatronix™ retains title to its own software. You may not rent, lease, loan, sell, distribute Mekatronix™ software, or create derivative works for rent, lease, loan, sell, or distribution without a contractual agreement with Mekatronix™.

3. **Limited Warranty**. Mekatronix™ strives to make high quality products that function as described. However, Mekatronix™ does not warrant, explicitly or implied, nor assume liability for, any use or applications of its products. In particular, Mekatronix™ products are not qualified to assume critical roles where human or animal life may be involved. For unassembled kits, you accept all responsibility for the proper functioning of the kit. Mekatronix™ is not liable for, or anything resulting from, improper assembly of its products, acts of God, abuse, misuses, improper or abnormal usage, faulty installation, improper maintenance, lightning or other incidence of excess voltage, or exposure to the elements. Mekatronix™ is not responsible, or liable for, indirect, special, or consequential damages arising out of, or in connection with, the use or performances of its product or other damages with respect to loss of property, loss of revenues or profit or costs of removal, installation or re-installations. You agree and certify that you accept all liability and responsibility that the products, both hardware and software and any other technical information you obtain has been obtained legally according to the laws of Florida, the United States and your country. Your acceptance of the products purchased from Mekatronix™ will be construed as agreeing to these terms.

Gainesville, Florida Phone 407-672-6780

MANIFESTO

Mekatronix™ espouses the view that the personal autonomous agent will usher in a whole new industry, much like the personal computer industry before it, if modeled on the same beginning principles:

- Low cost,
- Wide availability,
- Open architecture,
- An open, enthusiastic, dynamic community of users sharing information.

Our corporate goal is to help create this new, exciting industry!

WEB SITE: http://www.mekatronix.com
Address technical questions to tech@mekatronix.com
Address purchases and ordering information to an authorized Mekatronix Distributor http://www.mekatronix.com**/distributors**

TABLE OF CONTENTS

1 SCOPE 6
2 INTRODUCTION....6
2.1 Exactly What is Interactive C?....7
3 TJ PRO™ EXPERIMENTAL SETUP....8
3.1 About Batteries and Bench Testing Programs....8
3.2 Installing TJPED01 Directories....9
3.3 Executing IC for the First Time....9
4 INTERACTING WITH IC....10
4.1 Calculating with IC....10
4.2 Listing Files....11
4.3 Loading and Unloading Files....11
4.4 System Time Variables....12
5 TOUR DE ROBOT WITH IC....13
5.1 Motor Control....13
5.2 Bumper Sensor....14
5.3 Infrared Proximity Sensors....15
6 SOME POSSIBLE BEHAVIORSS....16
7 PROGRAMMING BEHAVIOURS....16
8 ADVICE ON DEVELOPING BEHAVIORS....16
8.1 Vulcan Mind Meld....17
8.2 Relative calibration of sensors of the same type....17
8.3 Adjusting to Ambient Conditions....17
8.4 Create simple behaviors....18
8.5 Build on simple behaviors....18
8.6 Integrating Behaviors....18
9 RECOMMENDED STRUCTURE FOR YOUR C CODE....18
10 TJ PRO™ EXPERIMENTS....21
10.1 Robot Connections During Program Development....22
10.2 Motor Experiments....22
10.2.1 Calibrating the Servos....22
10.2.2 Motor Angular Speed Characteristics....23
10.2.3 Writing an IC Program....24
10.2.4 Robot Translation....29
10.2.5 Robot Spin....33
10.3 Bumper Experiments....34
10.4 Terminal Output without IC....36
10.5 Infrared Experiments....38
11 APPLICATIONS....40
11.1 Programming a Behavior from a Specification....40
11.2 Application Program Descriptions....41
12 MULTITASKING BEHAVIORS....49
12.1 IC Multitasking in Operation....50
12.2 Multitasking in Robotics....54
13 FURTHER EXPLORATION....56

LIST OF FIGURES

Figure 1 A template standard for lexical structuring of your TJ PRO™ C-Code. The correctness of your code does not depend upon adhering to this standard. Rather, it makes it easier for you and other people to be able to read and understand the code. 20
Figure 2. A C-function written according to this standard makes the code more readable and maintainable. In the description block outlined by the asterisks, the possible input and output parameter sources are listed. Most functions use only a small subset. Those not used can be deleted, but some programmers prefer to keep them there with a *None* specifier (I must confess that I am not consistent). Many programmers find the asterisks boxes a nuisance, so feel free to drop them!..... 21
Figure 3 This program turns the both motors on 100 percent for 10 seconds and then stops them. 26
Figure 4. Adding a persistent variable to the program in *Figure 3*. 28
Figure 5. This code urges the robot to move forward. 45
Figure 6. This code adjusts the IR threshold when bumper contact occurs..... 46
Figure 7. Structure of an IC program to multitask five processes. The code for each process is shown in the next figure. 51
Figure 8. Each of these four multitasked processes take different amounts of time to output two strings of two characters each. The `msleep()` command forces the processes to consume more than the allotted five ticks. 52

1 SCOPE

This manual is intended for teachers, educators, university and advanced high school students, hobbyists, researchers and anyone else interested in advancing the infrastructure of robotics and advanced technology in our society. The goal of this manual is to enable teachers and professors, from middle school through the university level, to develop exciting, stimulating, and entertaining instructional materials for hands-on laboratories using embodied, intelligent, autonomous mobile robots.

This manual provides exercises and applications of Interactive C (IC) to the TJ PRO™ autonomous mobile robot. A commercial, Windows version of IC can be purchased from a Mekatronix distributor (http://www.mekatronix.com/distributors) for a nominal fee. Only the Windows version will be discussed in this manual. An older, free DOS version can be downloaded from the EEL5666 class web site at the University of Florida: http://www.mil.ufl.edu/imdlf.html. While there, also download the Interactive C Manual, Chapter 6 of the MIT 6.270 notes. A quick read of the IC Manual will enable you to better understand this manual. As you read this manual, you can conveniently refer to the relevant sections of the IC Manual for further assistance.

Caution!

Do all robot moving experiments on a tile, wooden, or otherwise smooth flat surface!

2 INTRODUCTION

An autonomous mobile robot, similar to its distant carbon-based life forms, exhibits three principal features, 1) machine actuation, 2) machine perception, and 3) machine cognition. In anthropomorphic terms, these translate to acting, sensing, and thinking. The philosophical and scientific issues as to whether a machine can actually think, perceive or act with intent is beyond the scope of this manual.

Programming behaviors is what autonomous mobile robots is all about, or, at least a substantial part of what it is all about! Without being technical, a behavior is whatever the robot does. The emphasis is on action. From the engineering viewpoint, you want to program behaviors that produce useful results. Make a robot vacuum cleaner, or a robot valet. With an artistic eye, you want to program behaviors that esthetically please or excite. Why not make TJ PRO™ dance? Have it perform a ballet on wheels.

From the scientific perspective you can inquire about the scope of machine intelligence and test your theories on a real robot. Out of intellectual curiosity and the creation urge, you might want to develop physically embodied *animats*, or artificial animals. Develop your own ecology with predator robots that drain the prey robots' batteries and prey robots that hide and avoid predator robots and seek battery recharging stations as food sources. Or, you can tailor a robot to enter many of the robot contests around the world.

Many of these contests require manipulation and sensors not supplied with TJ PRO™. But, with one or more Mekatronix MSCC11 single chip computers controlling the additional sensors and servo driven manipulation devices, a TJ PRO™ can often be expanded to meet contest requirements.

This manual assumes the reader has read the IC Manual and has it at hand. The text also presumes some capability in C programming and does not attempt to teach that skill. You may want to refer to the latest edition of *The C Programming Language* by B.W. Kernighan and D.M. Ritchie to brush up on your C programming skills, or even to learn C. Since some of the exercises illustrate elementary programming concepts, the skilled C programmer can skip the text and go straight to the code and begin playing. The extra programming development makes the manual easier to read and understand by individuals with little knowledge of C or Interactive C. By copying and modifying the programs here, those unfamiliar with C may actually compose useful IC programs, but programming independent solutions based only on what you read and learn here may be difficult.

2.1 Exactly What is Interactive C?

To quote from the IC manual,

> "Interactive C (IC for short) is a C language consisting of a compiler (with interactive command-line compilation and debugging) and a run-time machine language module. IC implements a subset of C… IC works by compiling into pseudo-code for a custom stack machine, rather than compiling directly into native code for a particular processor. This pseudo-code (or *p-code*) is then interpreted by the run-time machine language program."[1]

This virtual p-code machine allows interpreted execution with run-timer error checking. The resulting compiled p-code consumes considerably less storage than assembly code that performs the same functions. The run-time module and the stacked-based virtual p-code machine also enables multi-tasking for sophisticated behavior based programming. Examples will illustrate all these advantages. The typical disadvantage of this approach, loss of processing speed, cause little to no performance degradation in many autonomous robot behavior programs.

Caution

Interactive C is not a Standard ANSI C, so be careful of language limitations using IC.

[1] IC Manual, pp1 (pp119 of the 6.270 notes).

3 TJ PRO™ EXPERIMENTAL SETUP

You will need a PC running Windows, IC for Windows, a Mekatronix MB2325 serial communications board, a serial cable, a 6-wire serial cable, and a small box to hold TJ PRO™'s wheels off a desktop surface. Refer to the *TJ PRO™ Users Manual*[2] for instructions on how to configure these components into a working system. This manual assumes you have

1) Installed IC and the TJ PRO™ distribution software,
2) Read and understood the *TJ PRO™ Users Manual* and have it available as a handy reference,
3) The IC manual readily available, and
4) Connected the robot through the six wire serial cable, the MB2325 Communications board and a serial cable to your PC.

The TJ PRO™ robot does not move well on rugs or rough surfaces. You will also get longer battery life if you restrict the robot motion experiments to smooth flat surfaces.

Caution!

Do all robot moving experiments on a tile, wooden, or otherwise smooth flat surface!

3.1 About Batteries and Bench Testing Programs

Only use six AA nickel-cadmium rechargeable batteries to power the robot. Always keep the robot's batteries charged. While bench testing, keep the wheels off the bench and the charger plugged into the robot. This will insure many hours of testing and debugging without having to wait six hours for the batteries to charge or the hassle and expense of changing out the batteries. When leaving the bench for extended periods of time, turn the robot power off with the charger plug into the robot. Leave the DOWNLOAD/RUN switch in RUN mode. The charger will keep the memory indefinitely, so if IC is in memory, it will be there when you power up again. This saves the hassle, minimal to be sure, of repeating a first-time load of IC into memory every time you begin experimenting anew (refer to Section 3.3).

If the power light suddenly disappears anytime during experimentation with the robot, the most likely problem is that rapid, jerky motion of the robot has loosened the batteries in the battery- pack. Simply reseat the batteries firmly into the pack and power will usually come up. If reseating loose batteries does not fix the problem, check for a loose power connection to the computer board or for broken battery wires. If power problems persist, check the battery-pack voltage and the computer-board voltage to make sure power is actually available. You should get a nominal 6 to 7 volts from the battery-pack, depending upon level of charge, and the regulated computer voltage should be about 5 volts. A

[2] You can download all Mekatronix manuals free from the web site: http://www.mekatronix.com

battery-pack reading of 5.4volts indicates the batteries are discharged and require recharging or exchanged out with fresh batteries.

When the robot looses power, for whatever reason, you will have to perform a first time load of IC again, a slightly tricky process (refer to Section 3.3).

3.2 Installing TJPED01 Directories

The distribution disk, `proiced01,` which comes with this manual, contains four directories. Throughout the manual, I will refer to these directories as sources of programs, which are either solutions to problems or examples.

Execute the batch file *`install_proiced01.bat`* to transfer the four directories

```
TJPRO_APPLICATIONS
TJPRO_Experiments
TJPRO_Libsrc
TJPRO_Utilities
```

into the IC directory. Specify a path to the IC directory as an argument to the batch file. The default path is *`c:\ICtjp`*. Even more simply, just drag and drop the directories with a mouse.

3.3 Executing IC for the First Time

IC was designed originally to execute on the MIT Handy Board, which has a rather obscure hack on the serial input. IC takes advantage of that hack, but it makes for strange initial loading of the system on TJ Pro.

Turn on the robot, place the *DOWNLOAD/RUN* toggle switch into *DOWNLOAD*. From Windows execute IC. IC will give an error message that the board is not responding. Not to worry! The board is not responding because you have not had a chance to load the *p-coder* and relevant libraries into the robot. Select the *Yes* button in the IC error Window to configure the board. The system is configured to enable COM1 for serial communication. If you wish to use another COM port select it on the screen that now appears. To make your selection the default, you will need to change the *`Port = 1`* command in the *`ic.ini`* file to *`Port = n,`* where n is the COM port number you picked. Next, select the *Download Pcode* button in the opened IC Window. IC will now open the *`Libs`* directory. Select the file *`TJ-pro.icd`* file to open. Press the red *RESET* button on the robot and select the *OK* button in the window. You will observe a flurry of loading activity with the green LEDs on the MB2325 board rapidly flashing, and then another error message☺! IC just realizes there is no p-coder program to communicate with. Place the robot *DOWNLOAD/RUN* toggle switch into *RUN*, press the red *RESET* button and select the *Yes* button in the IC errorWindow. IC will then load the remaining code onto the robot and you are ready to go.

As long as you keep power to the TJ PRO™ memory, the *p-coder* will not have to be reloaded and you can turn the power switch on-and-off at will. If the batteries drain or you otherwise overwrite memory you will have to repeat the *first time load procedure* described above.

4 INTERACTING WITH IC

Before writing any robot programs lets look at some of the features of an interactive environment. You now have IC up and running in Windows. Open this document in Windows95 at the same time and tile the two applications horizontally. As you read this manual, you can jump back and forth between applications and do the exercises as you read them. All IC command lines are terminated and then executed when you type the *Enter* key. I will not remind you of this in the exercises, but it will soon become second nature to you and should not cause any confusion.

4.1 Calculating with IC

You can perform complex mathematical computations using IC by typing the expressions in the command window at the bottom.

For example, in the command window of IC type

```
1+2
```

IC responds with

```
IC> 1+2
Downloaded 7 bytes (addresses C200-C206)
<int> 3
```

The answer equals the integer three (`<int> 3`), which may come as no surprise. But, how about

```
sin(45.0*3.14159/180.)
```

which yields

```
IC> sin(45.0*3.14159/180.)
Downloaded 18 bytes (addresses C200-C211)
<float> 0.707106
```

Those of you mathematically inclined recognize the sin of 45 degrees. Of interest is that all numbers must be cast as floating point numbers, hence, the decimal point appearing in all the numbers.

As a final example, compute the sigmoid function

```
1./(1.+exp(-0.2))
```

to get

```
IC> 1./(1.+exp(-0.2))
Downloaded 19 bytes (addresses C200-C212)
<float> 0.549834
```

Using IC as a sophisticated calculator has nothing to do with TJ PRO™ or robotics, but it does illustrate how you can perform calculations as needed when playing with your robot!

4.2 Listing Files

You can see what library files you have loaded onto TJ PRO™ by typing

```
list files
```

in the IC command window. IC responds with

```
IC> list files
C:\ICTJP\libs\lib_rw11.c
C:\ICTJP\libs\twoservo.c
C:\ICTJP\libs\motorp.c
twoservo.icb
#done
```

These are exactly the files you loaded earlier when setting up IC using the file *TJ-pro.icd*.

4.3 Loading and Unloading Files

The command

```
load C:\ICtjp\tjpcode\tjpro1.c
```

will cause the C program `tjpro1.c` to be loaded onto the robot. Type `list files` to see the addition of the file `tjpro1.c` to the list of library files. To unload this file type

```
unload tjpro1.c
```

Now, type `list files` and observe that the file `tjpro1.c` is no longer in the robot's memory.

We will execute this file later. If you cannot wait, disconnect the robot, take the robot from your desk work area to the floor and press the red *RESET* button. This program will

run forever, so to continue with this manual, turn the robot off, place it on a stand so the wheels do not touch any surface, connect the serial cable, turn the robot on and unload the program. The robot will stop executing the program and be ready for more functions or programs.

You can also load a program from the IC window by selecting *File* and then *Download File*. I prefer this method since I do not have to type paths. No `unload` window command exist. One way to unload a function or program without actually typing out the command is to execute a `list file` command to get the right name. Copy the name (select and copy with the mouse) from the IC console window and paste it into the command line. Judicious maneuvering with the *up-arrow* and *down-arrow* line editing keys also speeds up command line typing.

4.4 System Time Variables

To see how many seconds have elapsed since you started your IC session, execute the function

```
seconds()
```

Reset the system clock with the function

```
reset_system_time()
```

and then execute `seconds()` again. The number will now be quite low because the clock started ticking from zero when you reset it. By the way the clock will never overflow in your lifetime. The time it takes to count to 10^{38} clock ticks at a millisecond rate far, far (and I mean FAR) exceeds the known age of the Universe!

The millisecond clock is another system clock that counts, you guessed it, in milliseconds instead of seconds. Execute the function `mseconds()`. Clearly, it too will never overflow during any session! Execute several in rapid succession to see how it rapidly changes.

The `sleep(<float_seconds>)` function puts the robot to sleep for `<float_seconds>` seconds. If you execute this function, say `sleep(10.0)`, the robot will not respond to any commands from IC for 10 seconds. Try doing a calculation of 1+2 immediately after typing `sleep(10.0)`. What happened?[3]

[3] IC aborts the execution.

5 TOUR DE ROBOT WITH IC

The following exercises allow you to interact with your TJ PRO™ robot and learn how to control the motors and observe the sensory inputs. As you expand your TJ PRO™ with more sensors and actuators you will be able to do the similar tests with your own functions and drivers. An understanding of these basic functions will enable you to implement robot behaviors!

5.1 Motor Control

Implementation choices enable you to control TJ PRO™'s motors using the *twoservo.c* library program. You do not have to understand this program, but I thought you might be interested in what library program drives the motors. The actual motor control function is

```
motorp(<motor number>, <percent speed>)
```

To enable motor operations type (*Be sure the wheels of the robot do not touch anything before typing the next command)*

```
servo_on()
```

The servos will turn-on and one or both might move slightly. Technically, if they were exactly calibrated, they would not move at all. We will illustrate calibration later. Beware! The electronics drift some, so even calibrated servos will still probably rotate slowly after a period of time. To still the motors so you can think, type

```
motorp(0,0)
```

and then

```
motorp(1,0)
```

Both motors should come to a complete stop, even if not calibrated, because the software disables the motors when it detects zero percent. This provides the programmer with a reliable *stop-motor* command.

To move motor zero, type

```
motorp(0,100)
```

Which wheel turned and in what direction?[4] (Read answer in footnote). You may wish to stop the motor before you read on. Use the *up-arrow* key to select the previous motor command that stopped the motor.

Execute the command

```
motorp(0,-100)
```

What does the wheel do now?[5]

Try different percentages for the motor speed.

Caution: make sure that the percent speed of the motor is always between –100 and 100, otherwise you will get unpredictable results.

Repeat the above for motor one.

5.2 Bumper Sensor

The TJ PRO™ bump sensors uses a slick trick. Each of the three bumper switch closures adds more current through a voltage divider circuit. This allows a program to actually determine where on the bumper TJ PRO™ has made contact with an object. The bumper voltage is measured by the function `analog(0)`. You will explore the resulting bumper values next.

Do not touch the bumper. Type

```
analog(0)
```

IC should return the integer 0. When you typed the above command, the *p-code* version of the analog function was transmitted to the robot and interpreted by the *p-code* interpreter or virtual machine.

Press and hold the middle front bumper switch and execute `analog(0)` again (Use the *up-arrow* edit key for fast action☺). You should get something around 46. This number may vary somewhat from robot to robot. Execute `analog(0)`while pressing and holding each of the four bumper switches in turn. You should get readings close to the values on the right.

Switch	Value
Front	46
Front right	24
Rear	129
Front left	68

[4] Left wheel in the forward direction. If the right wheel moved instead, reverse the servo connectors on the microcontroller board. Be sure to remember which connectors and the orientation of the plugs.
[5] The motor reverses direction and turns backward.

Now, press and hold down combination of bumper switches and execute *analog(0).* For example, hold the front and back switches down and execute *analog(0).* The value returned equals about 140. Hold and press the front and front-left bumper switches to get about 61. A value near 61, therefore, indicates that the bumper struck an object on the right side between the front and right-front bumper switches.

5.3 Infrared Proximity Sensors

The two IR LED on top of TJ PRO™'s plate emit 40KHz modulated infrared light when enabled. The MC68HC11 processor uses memory mapped input and outputs, so the programmer controls all output devices by writing to memory. IC provides a *poke* command to enable the programmer to control output devices.

```
poke(<address>, <byte data>)
```

The *poke* command loads the memory byte at <address> with <byte data>. For the TJ PRO™ the IR emitter can be controlled by storing data at *<IR_address> = 0x7000* (the prefix '0x' means that the address is at 7000 base 16 or hexadecimal). Each bit of location 0x7000 can control a digital output. For the TJ Pro, bits 0, 1 and 2, reading from right-to-left on the <data byte>, control the two front and the one rear IR emitters. The command

```
poke(0x7000,0x07)
```

turns on all three IR emitters.

Turn on the three emitters. What happens? Well, nothing that you can see. IR light is invisible to us. To determine if the IR emitters are actually working we will have to read the IR sensors or detectors. These detectors are in the shiny cans underneath the top plate and are read by *analog(2)* and *analog(3)*. At this time, in case you might be wondering, *analog(1)* has not been assigned is available for user sensor expansion.

Keep all objects a foot or more away from the front of your robot. Read the sensor associated with *analog(2)*. The return value should be the minimum value the IR sensor outputs to the microcontroller, around 85. Hold your hand on the front-left side of the robot and take an *analog(2)* reading. Be careful not to obstruct the right side. Repeat for the front-right side. What were your two readings? Which side gave the larger response? The value you get will depend upon how close you hold your hand to the robot sensor. Hold your hand about two inches away from the edge of the robot for each experiment. As long as you are careful to hold your hand to one side or the other, one value should equal about 85 and the other about 123. When you do this experiment

successfully, you will have identified which IR sensor is associated with `analog(2)`, the side with the larger reading. So, which sensor does `analog(2)` return?[6]

If you came to the correct conclusion, you can now predict that `analog(3)` returns the value of the left IR sensor. Verify this claim by repeating the above experiments with `analog(3)`. For accurate turbo typing, use the line editor to simply change 2 to 3 in your previous command. The ranges of the two sensor, while close, may not be exactly equal, an important fact to remember when programming behaviors.

We have completed the tour of TJ PRO™'s basic capabilities. You will certainly be amazed (I was!) when you discover the sophisticated robot behaviors you can program with just these simple capabilities.

6 SOME POSSIBLE BEHAVIORS

In the previous section you saw how IC library programs provide the basic hardware interrupt and device driver routines for the robot. These allow the user to access the sensor readings and drive the motors. With these routines, the user can program an unlimited number of behaviors. A representative set, but, by no means, an exhaustive set, of primitive set of behaviors, from which more complex ones can be developed, are listed in Table 1. Some of these behaviors, like line following, require installation of auxiliary sensor kits.

Table 1 Possible Primitive Behaviors

Collision avoidance	Collision detection	Line following	Light following
IR light avoidance	Pushing	Collision detection	Shy behavior
Aggressive behavior	Exploring behavior	Wall following	IR beacon tracking
Attraction to motion	Floor drawing	Fixed motion patterns	Dance motions

7 PROGRAMMING BEHAVIOURS

The exercises in this manual will help you write simple TJ PRO™ behavior programs to become more familiar with the robot and its features while, at the same time, building your confidence in TJ PRO™ program development. Understanding how to program the various robot features and simple behaviors will then permit you to piece them together to create complex behaviors. You will certainly be surprised along the way, especially when you think you have programmed the robot to do one thing and its actual behavior is quite different. Many times the robot's behaviors can only be understood after post-experimental analysis.

8 ADVICE ON DEVELOPING BEHAVIORS

The following advice is based on several years experience teaching engineering students to program autonomous robot behaviors.

[6] Analog(2) returns the value of the right IR sensor.

8.1 Vulcan Mind Meld

To effectively program a behavior for TJ PRO™, or, quite possibly, any autonomous robot, and to gain insight into the problems encountered by your robot, you should play Vulcan to the robot and imagine performing a *Vulcan Mind Meld* with it. All you Trekkie fans know what this means. But, to be specific, try to perceive the universe as the robot does with its limited sense capabilities. This is harder to do then you might think. Imagine yourself one with the robot. Play out different sensations and responses. Help yourself by actually recording robot sense data and examine typical responses, or responses to special environmental conditions of interest in the behavior you are developing. The mind meld will help prevent the common error of asking the robot to respond to environmental conditions it cannot detect with its sensors! While this statement is so totally obvious, it is also a difficult self-discipline to psychologically enforce. Why? Humans typically interact with each other, or intelligent animals, expecting and perceiving sophisticated behavior and sensory performance. These expectations seem to subconsciously creep into our agenda when working with autonomous machines, often with disappointing results! Autonomous robots have nowhere near the sensory and behavioral capabilities of an insect, let alone higher animals.

8.2 Relative calibration of sensors of the same type

Manufacturing tolerances, circuit tolerances, and mounting variations make it possible for two instances of the same type of sensor to respond differently to the same stimulus. Behaviors, therefore, should not be programmed to depend upon two sensors of the same type producing identical responses to the same stimulus. Instead, write programs to calibrate sensors of the same type in some fixed environment. For example, place a cardboard box in front of, and parallel to, the wheel axis of a TJ PRO™. Measure the response of the two front IR detectors using `analog(2)` and `analog(3)`. Note the differences in the readings. If there are none, that's great! In general, however, they will differ somewhat due to electrical noise or manufacturing tolerances.

Another approach for making your robot behave more reliably is to program robot behaviors that respond to relative sense stimuli, not absolute sense measurements. This will make the robot behave more organically and robustly to uncertain, dynamic environments.

8.3 Adjusting to Ambient Conditions

A programmed behavior will often be *brittle*, i.e., not flexible or adaptive, if that behavior depends upon specific magnitudes of robot sensor readings. Brittle behaviors fail when the environment changes from the environment in which the behavior was developed. The smaller the change that causes the failure, the more brittle that behavior is. For example, the IR detectors on TJ PRO™ will detect white objects at larger distances than dark objects. Suppose a collision avoidance algorithm sets a threshold value of the IR as an indication of an impending collision. If this threshold is determined experimentally with light colored obstacles, then dark colored obstacles will not be detected and the robot will

bump into them. On the other hand, if the threshold is set for dark colored obstacles, the robot will end up spinning in circles in a light colored environment because it detects threats everywhere. The solution is not to pick an average color threshold, but rather, program the robot to adjust its threshold downward if it has not detected a collision for some specified time, or, to adjust the threshold upward if it is colliding too frequently. The difficulty, of course, is determining exactly what the "specified time" between collision should be or what "colliding too frequently" means! The easy, but difficult to implement, answer is to let the robot learn these parameters based upon some performance criteria.

Robot behaviors and sensors, therefore, should adjust to ambient conditions. Biological organisms perform this function fantastically well. The human eye adjusts to bright sunlight or a darkened cathedral with dimly lit candles. This procedure is easier to state then execute, but serves as a general principle.

8.4 Create simple behaviors

The beginning robot practitioner usually formulates behaviors too complicated to implement directly. With experience, the virtue of simple, direct behaviors becomes apparent. Complex behaviors should be broken down into sequences of simple, primitive behaviors. If this can be done, the chances of successful implementation are high. If not, there is little value in trying to implement such behaviors directly.

8.5 Build on simple behaviors

As the user accumulates a repertoire of primitive behaviors, complex behaviors open up. Perhaps the easiest way to generate complex behaviors is simply to sequence a collection of primitive behaviors. For example, wall following might be decomposed as follows: 1) detect a "large" object, 2) approach the object until "near", 3) turn until the robot front-to-rear axis aligns "parallel" with the "surface" of the obstacle, 4) move "parallel" to the obstacle surface. At each instant of time a particular behavior in the sequence is invoked based on the current state of the robot and its sensory inputs. Of course, the programmer will have to establish to the robot's perception the meaning of such terms as "large", "near", "surface", and "parallel". Remember to Vulcan Mind Meld!

8.6 Integrating Behaviors

More complex behaviors may require the combination of primitive behaviors in a way not well understood. Artificial neural network activation, opinion guided reaction, non-linear dynamics, and fuzzy logic all offer techniques for integrating behaviors. Each technique offers specific advantages and specific difficulties. Discussion of such issues is beyond the scope of this manual. The reader's attention is brought to this matter to encourage investigation into these possibilities.

9 RECOMMENDED STRUCTURE FOR YOUR C CODE

Figure 1 and Figure 2 illustrate suggest a coding format standard for C programs. Elements that go in the *Includes, Constants,* and *Prototypes* blocks can be grouped

according to function, and alphabetical within the group. For IC, the *Includes* libraries are loaded from the terminal and not placed in the program as with a compiled version of C. We comment the *Includes* block with the relevant library functions used by the program. Variables that are defined in the *Globals* block should be arranged by context (All sensor variables together, all motor variables together, etc.). Obscure names should be avoided. A small description should accompany each function prototype:

```
/* magnitude(x,y) returns the magnitude of vector [x,y] */
  int magnitude(int x, int y);

/* init_SCI() initializes the serial communications port */
   void init_SCI(void);
```

Comments on the other items (*Globals, Constants, #defines and #includes*) will increase readability. Here are some recommended guidelines for coding in general. These guidelines are not laws of nature and esthetic tastes will vary among individuals. So do not force them over reasonable alternatives (some programmers object to * boxes because they waste so much time to generate). The underlying philosophy behind these guidelines is to make your code easier to read, debug and understand, not to make life painful.

1. Indent each new syntactic level at least 2 spaces. This can be called the *neatness rule*. Example:

```
  while (i > 0)
  {
    printf("I've been assimilated\n");
    if (x == 1)
      i--;
    else
      i++;
    x = i + 10;
  }
```

2. When possible, comment on what a block of code will do, instead of describing the purpose of each line in a block. Some exceptions to this are when writing assembly programs or when writing "tricky" code that would otherwise be unreadable. Single line comments can make reading easier, but do not go overboard. When in doubt the general rule is: " Does this comment make the program easier to understand?"

3. Put the main program before all the function definitions. There is great esthetic dispute on this, so if you want to put `main()` last, feel free!

4. Describe all functions clearly.

In the main body of your program, namely, in *`main()`*, you will typically need some initialization functions, particularly,

```
                    MEKATRONIX CODING STANDARD

/*********************************************************************
 *                MEKATRONIX Copyright 1998                          *
 * Title                                                             *
 * Programmer                                                        *
 * Date                                                              *
 * Version                                                           *
 *                                                                   *
 * Description                                                       *
 * Version History:                                                  *
 *                                                                   *
 *********************************************************************/


/************************** Includes *********************************
#include <tjbase.h> /* All TJ code requires this #include */
/************************ End of Includes ****************************/


/************************** Constants ********************************/
#define FULL_SPEED 100

/************************ End of Constants ***************************/


/************************** Prototypes *******************************/


/************************ End of Prototypes **************************/


/************************** Globals **********************************/


/************************ End of Globals *****************************/
main()
/******************************* Main ********************************/
{

}
/***************************** End of Main ***************************/
```

Figure 1 A template standard for lexical structuring of your TJ PRO™ C-Code. The correctness of your code does not depend upon adhering to this standard. Rather, it makes it easier for you and other people to be able to read and understand the code.

Figure 2. A C-function written according to this standard makes the code more readable and maintainable. In the description block outlined by the asterisks, the possible input and output parameter sources are listed. Most functions use only a small subset. Those not used can be deleted, but some programmers prefer to keep them there with a *None* specifier (I must confess that I am not consistent). Many programmers find the asterisks boxes a nuisance, so feel free to drop them!

```
function()
/*********************************************************************
 * Function description:                                             *
 * Returns:                                                          *
 *                                                                   *
 * Inputs                                                            *
 *   Parameters: None                                                *
 *   Globals:    None                                                *
 *   Registers:  None                                                *
 * Outputs                                                           *
 *   Parameters: None                                                *
 *   Globals:    None                                                *
 *   Registers:  None                                                *
 * Functions called: None                                            *
 * Notes: None                                                       *
 *********************************************************************/
{

}
/***********************End function ****************************/
```

```
servo_on();     /* Enables motor control. */
init_serial();  /* Provides primitive serial communication. */
```

Many programming problems arise whenever one of these initialization functions has not been executed at the beginning of *main()*.

10 TJ PRO™ EXPERIMENTS

The numbered items below suggest a sequence of ever more complex programs and experiments you can perform to familiarize yourself with TJ PRO™'s capabilities. These experiments will open up to you the rich variety of behaviors you can program. After each successful experiment, save your program and do not change it. Use copies of it to begin other programs, but do not write over and destroy your only copy of a successful program. You will never know when you might want to use it again, either as is, or as a basis for another program.

Program solutions to the following experiments are on the education diskette *proiced01*. Since a computer language provides a rich structure for developing procedural solutions to problems, you should realize that the **Solutions** given here represent only one of large number of possible ways of solving the problems stated. Also, ambiguity of language will

lead to different interpretations of what the problem states and, thus, give rise to other solutions. In fact, one of the great difficulties in the discipline of computer programming is to translate an imprecise natural language specification of a problem into a precise computer language specification and then determine if the program does what you specified. In fact, once the program performs to the end-user's satisfaction, the program *becomes* the precise statement of the solution and *defines* what problem it solves. Often, the resulting program solution does not exactly match the original, ambiguous specification, but provides what the end-user finds acceptable!

10.1 Robot Connections During Program Development

Do not forget the recommended operating procedure for robot program development.

1) Keep the robot on its charger during program development. In this way the robot will almost always have fresh batteries to perform floor experiments when you reach that point in your design.
2) Maintain serial connection between the robot and your PC.
3) For convenience, keep the TJ PRO™ mounted on a stand with the wheels suspended in the air. This way you can perform most experiments and tests during early program development without placing the robot on the floor, which requires connecting and disconnecting the serial cable and the battery charger each time.

10.2 Motor Experiments

These experiments acquaint you with the motor functions and how to get TJ PRO™ to move the way you want it to move.

10.2.1 Calibrating the Servos

To calibrate the servos requires finding the modulating pulse width that stops the wheels from turning. Because of some instabilities in the control electronics designed by the servo vendors, you will find it difficult to exactly pulse the motors so precisely that they do not move. The pulse width for zero motion of the wheels is nominally 3000 processor cycles, about 1.5 milliseconds. The persistent IC variables that controls the servo pulse widths `servo_pulse_wavetime1` and `servo_pulse_wavetime2` are `Left_Zero` and `Right_Zero,`

```
servo_pulse_wavetime1=Left_Zero;
servo_pulse_wavetime2=Right_Zero;
```

The servo calibration utility routine, `calsvtjp.c` in the directory `TJPRO_Utilities`, defines four functions

```
void Rs(int delta) {Right_Zero += delta;}
void Ls(int delta) {Left_Zero  += delta;}
void Ras(int num) {Right_Zero = num;}
void Las(int num) {Left_Zero  = num;}
```

which allow you to change the values of `Left_Zero` and `Right_Zero` incrementally or absolutely. Good experimental values that I found for almost stopping both servos on my TJ PRO robot where `Right_Zero = Left_Zero = 3072`.

Servo Calibrate Procedure

1) Download `calsvtjp.c`
2) Press reset,
3) Type the IC command Ras(3072),
4) Type the IC command Las(3072).

Did both motors stop or at least move very slowly, say, less than ½ revolution per minute? Use `Rs()` and `Ls()` to incrementally change the settings and observe how the wheel rotate. Of course, without the above data, you can start at the initial settings and explore on your own how to find the right values that zero the servos.

Once you have good values zero values for both servos, open `motorp()` in the `Libtjp` directory and change the two `#defines`,

```
#define Right_Zero   3000
#define Left_Zero    3000
```

to the calibrated values you experimentally determined.

Caution: Even calibrated servos drift and so try to program behaviors that do not require precision motor control.

10.2.2 Motor Angular Speed Characteristics

In this section, you will measure the wheel motor angular speed, the number of wheel revolutions/second, at various speed settings.

Angular Speed Experiment 1

1) Put a narrow strip of tape on the wheel to make it easy for you to identify a complete rotation.
2) Execute `motorp(0,100)` from the IC command line.
3) Measure the speed of the turning wheel. Use a stopwatch and measure how long it takes the wheel to turn 10 times.
4) *Compute the angular rotation rate of the wheel in revolutions/second.*
5) Change the speed of the motor to 75%, 50%, 25%, 15%, 10%, 5%, successively, and measure the angular rotation at each speed.
6) Plot angular rotation of the wheel vs. the speed.

What conclusions can you draw from the curve?[7]

Angular Speed Experiment 2

Do the previous experiment using `motorp(1,?).`

Do the left and right motors have the same response to percentage specifications? What does this tell you about precise control of the robot motion?[8]

Angular Speed Experiment 3

Repeat **Angular Speed Experiment 1** using negative percentages.

Compare the graphed results of this experiment with the graph plotted in **Angular Speed Experiment 2.** *Do the graphs appear to be close?*

Angular Speed Experiment 4

Repeat **Angular Speed Experiment 2** using negative percentages.

Compare the graphed results of this experiment with the graph plotted in **Angular Speed Experiment 1.** *Do the graphs appear to be close?*

The idea behind the last two experiments is to illustrate that the motors probably perform about the same when turning in the same relative direction, but have significant differences in performance when their relative angular velocities are oppositely directed. Consequently, rotations using a negative speed percentage on one wheel and a positive speed percentage on the other will, in general, be more precise than motion resulting from same-sign speed percentages on both wheels.

Angular Speed Exercise

1) Measure the diameter of each wheel. Are they the same? How much error do you estimate in your measurement?
2) From the measurement of the right wheel diameter convert one of the two graphs generated for the right motor angular speed to linear speed of the wheel contact on the floor versus the various percentages. Recall that the wheel contact linear velocity magnitude equals the radius of the wheel times the angular velocity.

10.2.3 Writing an IC Program

In this section you will write a simple program and then learn to use *persistent* variables for changing parameters in your program from the IC command line.

---

[7] *One Possible Conclusion: The curve is non-linear, meaning that a % change in the motor software speed parameter does not give you the same % change in speed of the actual motor.*

[8] *The two motors appear to have quite different responses. This asymmetry may result from the mechanical structure of the brush mechanism, which performs differently when the servo motor rotates in one direction versus the other direction. Each motor is actually turning in a direction opposite to the other when the robot goes forward or backward , hence, the different motion characteristics for the same speed % . You need to keep this in mind when programming motion control.*

Program Objective

Code a simple program to turn on both of TJ PRO™'s motors for 10 seconds.

Specification

Turn on both motors 100 percent in the forward direction for 10 seconds and stop.

A Solution

The program *xm1atjp.c* in Figure 3 does the job.

The *servo_on* function actually sets up an interrupt service routine that periodically executes and controls the motors using pulse-width-modulation (PWM). As long as the processor is powered, this interrupt service routine will drive the motors at the last speed specified by *motorp*, even after the instruction has completed execution.

Since I can never remember the index numbers for the right and left motors, I use #define to name such constants:

```
#define LEFT_MOTOR      0
#define RIGHT_MOTOR     1
#define MAXSPEED      100
#define ZEROSPEED       0
```

These #defines changes the meaningless index numbers into useful information. As a general rule constants should be defined to give them context. The number 0 means the left motor in the context of *motorp(0, 100)* , but in other contexts it may mean something totally different. For instance, in *motorp(LEFT_MOTOR, ZEROSPEED)* both arguments have the numerical value of zero, but the zeros mean totally different things. The #define allows you to differentiate contexts simply by naming the constants. A given constant can be assigned many names to allow differentiation in usage.

Most of the text in the program is documentation and has nothing to do with the executable functions and statements. This is typical. If you save the standard program documentation structure in a file, you can save time by inserting that structure each time you develop a new program. Of course, if you are modifying a program, most of the structure will be in place, provide, of course, you used the structure in the first place!

Experiment with *xm1atjp.c*

Download the program *xm1atjp.c* in Figure 3 into your TJ PRO™ using the IC *Download File* menu selection. Execute this program by pressing the reset button on the robot. *Remember, robot wheels not touching, otherwise you will have to move fast!*

Verify the motors turn for 10 seconds. You can use the Windows95 console clock to measure seconds or you can estimate by counting.

```
/*********************************************************************
 * Title          xm1atjp.c                                          *
 * Programmer     Keith L. Doty                                      *
 * Date           August 21, 1998                                    *
 * Version  1                                                        *
 *                                                                   *
 * Description                                                       *
 *   Spin both wheel 100% in the forward direction for 10 seconds.   *
 *                                                                   *
 *********************************************************************/

/*************************** Includes *********************************/

/*
  Before loading this program, be sure you load the library files
      twoservo.icb
      twoservo.c
      motorp.c
*/

/************************ End of includes *****************************/

/************************** Constants *********************************/

#define LEFT_MOTOR      0
#define RIGHT_MOTOR     1
#define MAXSPEED      100
#define ZEROSPEED       0

/************************ End of Constants ****************************/

void main()
/****************************** Main **********************************/
{
  float sleeptime;

/*Initialization */
  servo_on();
  sleeptime=10.0;

/* Start the motors */
  motorp(RIGHT_MOTOR, MAXSPEED);
  motorp(LEFT_MOTOR, MAXSPEED);

  sleep(sleeptime);

/* Stop the motors after sleeptime seconds */
  motorp(RIGHT_MOTOR, ZEROSPEED);
  motorp(LEFT_MOTOR, ZEROSPEED);

}
```

Figure 3 This program turns the both motors on 100 percent for 10 seconds and then stops them.

This program is rather boring! You cannot even change the `sleeptime` variable. Try to change it from the IC command line with *`sleeptime = 1.0`*. IC will tell you

```
IC> sleeptime = 1.0
Undeclared symbol sleeptime
```

The problem here is that the variable *`sleeptime`* declared in the function *`main()`* is local to *`main().`* That is to say that *`sleeptime`* no longer exists, or persists, after execution of *`main().`* You cannot run this program with different values of *`sleeptime`* without rewriting the statement that initializes that variable. There is a way out. IC provides global and *persistent* global variables, which permit you to input, from the IC console, different values of a program variable prior to execution.

To make a variable global it has to be defined outside of any function. This means any function has access to the variable. Declaring such variables *persistent* makes them available with their values, even after a reset or termination of the program. You can read persistent variables during a session as well as change them. Other global variables will initialize on reset.

The program in Figure 4 differs from the program in Figure 3 only in the definition of *`sleeptime`* as a persistent global variable.

Experiment with *`xm1btjp.c`*

Download the program *`xm1btjp.c`* from the directory *`TJPRO_EXPERIMENTS`* into your TJ PRO™ using the IC *Download File* menu selection. Execute this program by pressing the reset button on the robot, or by typing *`main()`* in the IC command line. The typing of *`main()`* as an alternative method of starting a program may be useful. I prefer pressing the reset button, normally, but when loading and downloading files remotely using radio or IR communications, the typing alternative is quite handy!

The program does not do any thing. Why? The variable `sleeptime` was not assigned a value by the program, so IC automatically assigns the value 0 to it. Type `sleeptime` into the IC command line to see its current value. It should be zero. A zero wait time between starting and stopping the motors gives no time for the motors to move before being stopped.

Execute `sleeptime=3.0` and press reset on the robot. What happens now? You should see the wheels move forward for 3 seconds. Any number of resets from now on will give the same response, since the value of `sleeptime` persists. Make `sleeptime=10.0` while the program is executing. Although the variable changes to 10, the program only uses sleeptime once, so it has no effect on this particular execution of the program. However, when you start the program again, it will run for 10 seconds.

```
/*****************************************************************
 * Title          xm1btjp.c                                      *
 * Programmer     Keith L. Doty                                  *
 * Date           August 21, 1998                                *
 * Version  1                                                    *
 *                                                               *
 * Description                                                   *
 *   Spin both wheel 100% in the forward direction for sleeptime *
 *   seconds.                                                    *
 *****************************************************************/
 /*************************** Includes ***********************************/
/*
  Before loading this program, be sure you load the library files
      twoservo.icb
      twoservo.c
      motorp.c
*/
/************************ End of includes ******************************/

/************************** Constants ***********************************/
#define LEFT_MOTOR      0
#define RIGHT_MOTOR     1
#define MAXSPEED      100
#define ZEROSPEED       0
/************************ End of Constants *****************************/

/*************************** Globals ************************************/
  persistent float sleeptime;
/************************** End of Globals *****************************/

void main()
/***************************** Main *************************************/
{
/*Initialization */
  servo_on();

/* Start the motors */
  motorp(RIGHT_MOTOR, MAXSPEED);
  motorp(LEFT_MOTOR, MAXSPEED);

/* Note: sleeptime initialized zero automatically. Use IC command
   line to change the values of this persistent global variable and
   then execute this program with the new value of sleeptime by
   pressing the reset button on the robot.
*/
  sleep(sleeptime);

/* Stop the motors after sleeptime seconds */
  motorp(RIGHT_MOTOR, ZEROSPEED);
  motorp(LEFT_MOTOR, ZEROSPEED);
}
```

Figure 4. Adding a persistent variable to the program in ***Figure 3***.

You can also generate dynamic effects with persistent global variables. For example, if you place the statement

```
while(sleeptime <2.5);
```

before the statement

```
servo_on();
```

the revised program (`xm1ctjp.c` in `TJPRO_Experiments` directory) will idle on the *while* statement until you change `sleeptime` to a number larger than 2.5 seconds using the IC command line. This capability provides some serious external program control that you might find useful in complex applications.

Exercise using Persistent Variables

Write a program that controls both motors. Define `speedl` and `speedr` as persistent integer global variables that control the left and right motor speeds, respectively. Bracket the two motor control statements in *`main()`* with an endless `while` statement:

```
…
servo_on();
 while(1)
   {
    program statements
   }
…
```

The while statement forces endless execution of the program. As the program executes change `speedl` and `speedr` to different values and observe the wheel speed changing as a result of your console commands.

A solution to this exercise is `xm1dtjp.c` in `TJPRO_Experiments` directory.

10.2.4 Robot Translation

The motion of a rigid body breaks down into two essential motions, translation and rotation. Translation means the body moves in a direction without changing its orientation. Translation does not necessarily mean that the body moves in a straight line. However, since TJ PRO™ must change its orientation to change its direction, TJ PRO™ can only translate in a straight line (well, “straight line” exaggerates reality a tad!). In this section you examine how to control the motors to translate TJ PRO.

Programming A General Motion Control Program

Objective

Measure the robot's deviation from straight-line motion when it is supposed to be going straight.

Specification

Turn on both motors at 100% forward for 10 seconds and stop the robot. Use the back bumper switch to start the motion.

A Solution

You can generate a simple solution by modifying `xm1atjp.c` to start executing only after the rear bumper is pressed.

A solution to this exercise is `xm1etjp.c` in `TJPRO_Experiments` directory.

For a more general solution, modify the above program by making `sleeptime`, `speedr` and `speedl` persistent global variables so you can perform a variety of experiments. Also, put the relevant code in an endless while loop so you do not have to press reset between experiments, only the rear bumper.

A solution to this exercise is `xm1ftjp.c` in `TJPRO_Experiments` directory.

Perform this experiment in a wide-open space at least 4 feet wide and 12 feet long. Disconnect the robot and place on the floor. Press reset to start the program and tap the rear bumper to close the bumper switch and start the robot moving. The robot will move forward for 10 seconds.

Questions

a. *After 10 seconds, how many inches has the robot moved forward along its initial direction?*
b. *How many inches has the robot veered to the left or right from the line of its initial direction?*

To enable answering these two questions, you can perform the experiment on a tile floor. Line up the left wheel on a long, straight tile-line and determine how far the wheel has deviated from the axis of that tile-line after the robot stops.

With *`xm1ftjp.c` in `TJPRO_Experiments` directory* you can easily perform a wide variety robot motion experiments.

Experiment 1 with *`xm1ftjp.c`*

Objective

Move the robot in forward and determine if it goes straight.

Specification

Turn on both motors in the forward direction at the same percentage.

Persistent Control Variables

```
speedl          Speed of left motor in ±%
speedr          Speed of right motor in ±%
sleeptime       Duration of the motion in seconds
```

Download *xm1ftjp.c* into your TJ PRO. Input *speedl=100* and *speedr=100* for the left and right motors speeds, respectively. Set `sleeptime = 10.0` seconds. Unplug the serial and charger cables, carefully place TJ PRO™ on the floor, press reset to start the program and tap the rear bumper switch gently to start the robot moving.

Questions

a. Does the robot go straight? Which way does it prefer to turn, to the left or to the right?

b. Does this make sense with respect to you previous motor graphs? Explain.

Experiment 2 with *xm1ftjp.c*

Objective

Move the robot in reverse and determine if it goes straight backward.

Specification

Turn on both motors in the reverse direction at the same percentage.

If you are continuing from the previous experiment, you can

1) wait for the experiment to stop, or catch the robot and turn off the power switch),
2) bring the robot to its desk stand,
3) connect the serial cable (and the charger, if you think you will take more than a few minutes),
4) (power up the robot, if you had turned it off in step 1!)
5) enter the reverse speeds, -100 and –100 for both motors on the IC command line and, then,
6) execute the code.

Otherwise, download `xm1ftjp.c` into your TJ PRO, set `sleeptime =10.0` and the motor speeds to –100 and –100 percent and execute the program by pressing the reset button. Place robot on the floor again and tap the back bumper to set the robot in motion.

Questions

a. Does the robot go straight backwards? Which way does it prefer to turn, to the left or to the right?

b. Which motor, left or right, appears to be turning faster for the 100% reverse command? Are your results consistent with the previous experiments? Explain.

Further Experiments:

1. Try to compensate for the robot veering from forward straight-line motion by slowing down the faster motor (use program *xm1ftjp.c in TJPRO_Experiments directory*).

2. Always keep the slow motor at 100%. You could use a systematic, binary search to find the best percentage for the fast motor. First, try 50% for the fast motor. If 50% slows the fast motor too much and the robot veers in the direction of the fast motor, try 75%, half way between 50 % and 100%. If 50% does not slow the fast motor enough, i.e., the robot continues to veer in the direction of the slow motor, then try 25%, half way between 0% and 50% (this is not likely!). Continue dividing the resulting range in half for each experiment, increasing your estimate for the fast motor if the robot veers in the direction of the fast motor and decreasing your estimate for the fast motor if the robot veers in the direction of the slow motor. Repeat this process until you cannot divide the percentage range further. This *binary search process* will require, at most, 7 experiments (*Why?*) before you obtain the best *open-loop* straight-line motion possible. *Open-loop* means no feedback (no sensory measurement) is used to provide control information that would help the robot compensate for error in its motion.

   Your program should drive the slow motor at 100% forward and the fast motor at X% for 10 seconds and stop the robot. X% will be assigned a specific value by you for each experiment. A good initial guess at X% will save you time by reducing the number of searches.

Questions

a. What percentage did you get for the fast motor that yielded the best straight-line motion?

b. *Does the robot still veer? If so, which way, towards the fast motor or the slow motor? Does the robot veer consistently to the same side over a set of 5 to 10 experiments? Explain your results and discuss any puzzling features you discovered.*

Experiment 3 with `xm1ftjp.c`

Repeat **Experiment 2 with `xm1ftjp.c`** except, now, move the robot in reverse.

Questions

a. Compare the results moving the robot backward in a straight line to moving the robot forward in a straight line. Did you find your results puzzling (Hint: I did)?

b. Can you provide an explanation for what you observed, an explanation that can be tested? (This question does not have an easy answer. I would like to hear from anyone who has a verifiable hypothesis for the observed behavior and has tested and verified that hypothesis. Email your explanations to doty@mekatronix.com.)

Translate and Back Motion

Write a new program starting from `xm1ftjp.c`. The new program makes the robot go forward for *n* seconds, stops the robot for a full second, and then reverse for *n* seconds, after the back bumper switch is pressed. Download, execute and test.

Questions

a. *Does the robot return exactly to the same spot it left?*
b. *Repeat the experiment 5-10 times and measure the x and y position errors from a fixed reference point.*

A solution to this exercise is `xm1gtjp.c` *in* `TJPRO_Experiments` *directory.*

10.2.5 Robot Spin

In this set of experiments you observe and measure the characteristics of robot spin, the second type of rigid body motion.

Spin Experiments with `xm1ftjp.c`

1. Make the robot spin clockwise about the right wheel at maximum angular speed. Make `sleeptime` a large number in order to give yourself time to watch the robot behavior.

   a. *Does the right wheel stay in one place or does it drift away from its initial spot?*
   b. *Measure the angular rotation rate in revolutions/second. Suggestion: pick a feature on the TJ PRO™'s top plate and use a stopwatch to measure how long it takes that feature to rotate 10 times.*
   c. *Measure the drift displacement. (Hint: Mark a water erasable footprint of a wheel on a tile floor. Place the wheel on the footprint. Enter a* `sleeptime` *equal the approximate time for the robot to make 10 turns. Run the robot for that time and mark a new footprint. Measure the direction and distance from a point on the first footprint to the corresponding point of the second.)*
   d. *If you double or triple the time for spinning, does the displacement distance double or triple?*

2. Make the robot spin counterclockwise about the right wheel at maximum angular speed.
   Apply the questions in the previous experiment to this experiment.

3. Make the robot spin clockwise about the left wheel at maximum angular speed.
   Apply the questions in the first experiment to this experiment.

4. Make the robot spin counterclockwise about the left wheel at maximum angular speed.
   Apply the questions in the first experiment to this experiment.

5. Make the robot spin clockwise at maximum angular speed about its center axis by setting the left and right motor speed equal in magnitude, but opposite in sign.
   Apply the questions in the first experiment to this experiment.

6. Make the robot spin counterclockwise at maximum angular speed about its center axis by setting the left and right motor speed equal in magnitude, but opposite in sign.
   Apply the questions in the first experiment to this experiment.

Data Analysis

a. Compare the drift displacements for Experiments 3 and 4.
b. Compare the drift displacements for Experiments 5 and 6.
c. Compare the angular rotation rates of Experiments 3 and 4.
d. Compare the angular rotation rates of Experiments 5 and 6.
e. If r_i is the angular rotation rate in Experiment-i, then compute the ratios r_1/r_5, r_2/r_6, r_3/r_5, r_4/r_6. Do you see a relationship between the angular rates? Explain or justify your claims.

10.3 Bumper Experiments

To this point you have investigated the actuation capabilities of the TJ PRO™ robot and you have seen how to access sensory information. In this section, and the next, you will explore the interaction between TJ PRO™'s perceptual and motor capabilities to realize robot behaviors.

TJ PRO™'s bumpers provide the robot with a sense of touch. TJ PRO™ can differentiate a bump contact at six different points: at each of the four bumper switches, between the front-middle and front-right, and between the front-middle and front-left switches. Three bumper switches in front and one in back make TJ PRO™ more sensitive to frontal contact than back contact. The design presumption is that TJ PRO™ mostly moves forward and needs to be more touch-capable in the forward direction.

Caution!
In the bump experiments TJ PRO™ will bump into objects in order to respond to them, so take care that TJ PRO™ does not bump into anything that will harm TJ PRO™ or the object. The small size of TJ PRO™ does not usually invoke cause of concern, but the world is complex and a word of caution will remind you to evaluate TJ PRO™'s environmental situation before setting it free.

Program Bump_Around Behavior

TJ PRO™ moves forward until it bumps into something, at which point it backs up for T seconds, turns a random angle and goes forward again. If, while backing up, the back bumper switch contacts an object, TJ PRO™ will go forward T/2.0 seconds, turn a random angle and continue forward once again. Use the back bumper to start the program after each reset for ease of handling. (For one "ease", back bumper switch control lets you stop the critter on the run by pushing the red reset button!)

Assignment

Design a TJ PRO™ IC program to implement the Bump_Around behavior.
Specifications

Variables:

`persistent float T=0.350` Times the motions
`float marktime` Saves current value of seconds()

Motor speeds:

Forward: 100% both motors
Backward: –100% both motors
Turn Right: 100% left motor, -100% right motor
Turn Left: 100% right motor, -100% left motor

Functions:

`float seconds(void)` System function used to time actions.
`void turn(void)`

A user defined function that turns the robot a "random" angle. IC does not support `random()`, so what I did is use the last digit of

```
rand = (int)mseconds();
```

to select a "random" direction of spin

`if (rand & 0x0001)` spin left **else** spin right

then, after setting the appropriate motor speeds, I computed the "random" duration of the spin in milliseconds with the code sequence

```
rand = (rand & 0x00ff)<<2 ;
msleep((long)rand);
```

Bumper Values:

Assume a back bumper hit if `BUMPER (= analog(0))>120`. Ignore the fact that if both a back and any front bumper switch is pressed simultaneously `BUMPER > 120`. Similarly, assume that

```
if((BUMPER>10)&&(BUMPER<110))
```

one or more of the front bumpers have been switched on.

Important! *Note the requirement that* `BUMPER>10`. *Ten is a "fuzzy zero" to protect against bumper noise inadvertently triggering a false detect. In theory,* `BUMPER>0` *should work, and it might sometimes, but it is not reliable. I have detected noise as high as 8 on the bumpers*.

A solution to this exercise is `xbmp1tjp.c` *in* `TJPRO_Experiments` *directory.*

Turning_Away Behavior Based on Bumper Contact

Program TJ PRO™ to turn away from a bumper contact as described in the table below. As with the *Bump_Around* behavior, define `persistent float T=0.35` as the backup time and T/2.0 as the forward time on a back bumper switch closure. You can change the value of T experimentally to see what other values do. I find `T=0.35` gives good performance, but you may find another value that suits your purposes better.

Switch Closure(s)	Range	Action
Front-Middle	40-50	Back up 350 ms[1]. Turn right 500 ms.
Front-Right	20-30	Back up 350 ms. Turn left 250 ms.
Front-Left	67-70	Back up 350 ms. Turn right 250 ms.
Front-Middle:Front-Right	60-63	Back up 350 ms. Turn left 125 ms.
Front-Middle:Front-Left	90-100	Back up 350 ms. Turn right 125 ms.
Back	120-135	Forward 175 ms. Turn left 500 ms.

[1]ms = milliseconds = 0.001 seconds

You will need the IC millisecond timer function called `msleep(<Long_integer>)` to time out the turns. Redefine the *`turn()`* function,

```
void turn(int bump)
```

where *`bump`* is the current reading of the bumper switches, *`analog(0),`* and *`turn(bump)`* computes both the direction and duration of the turn based on the value of *`bump`* and then executes the turn for that direction and duration.

A solution to this exercise is `xbmp1tjp.c` in `TJPRO_Experiments` directory.

Questions

Qualitatively compare this fixed turning away behavior with the previous one with "random" motion. Which motion appears more sophisticated? More organic, animal like? Which program has fewer lines of code? Which behavior appeals more to your sense of esthetics, or which motion gives you more satisfaction? Which behavior appeared more interesting to you?

(There is no one "right" answer to each of the above questions, but can you defend your answers with reasoned arguments?)

Application of Sensors in Multiple Behavioral Contexts

One of the neat aspects of an autonomous robotic agent is the multiplicity of uses to which we can put its sensory and actuation capabilities. We have already demonstrated this with the bumper switches. The rear bumper switch, for example, has been used not only to detect rear collisions, but also to serve as a "GO" switch. It gets even better. The *bump-programs* use the back bumper switch both ways, depending on context. After the robot is told to "GO" by pressing the back bumper, the switch resumes its role as a collision detection sensor. The contextual meaning of a sensory input looms as an important concept in autonomous robotics. As you develop your own programs you can take advantage of this powerful idea and use the sensor suite in multiple ways.

10.4 Terminal Output without IC

You have already seen examples of how *persistent* variables can be used as program inputs, even dynamically, but, so far, none of the TJ PRO™ programs have output data to

a terminal. Unfortunately, IC is designed to output data to an LCD screen on the robot using a rather obscure feature of the MC68HC11, a rather inconvenient situation for TJ PRO™ which has no LCD screen! This aside will show you how TJ PRO™ can output to a VT100 screen simulated by the *Windows95* Accessory program *Hyper Terminal*. The sacrifice is that IC must be exited, *Hyper Terminal* opened and configured as follows:

Configure Hyperterm on Win95:
Direct to COM1, 9600 baud, 8 bits, no parity, 1 stop bit, no flow control.

Since IC must be closed to do PC screen output, you will not have access to persistent variables to input data. Apparently, the virtual *p-code* machine continues to look for input from IC, so your screen input, if not synchronized with the virtual machine, will not typically be accepted (sometimes you can fortuitously get a character inputted).

The process to write TJ PRO™ programs that write to your PC screen is described next.

TJ PRO™ PROGRAMS THAT WRITE TO YOUR PC SCREEN

1) Load *serialtjp.c* from *TJPRO_Libsrc* directory,
2) Load the TJ PRO™ program that uses the serial output routines in *serialtjp.c,* Make sure your program uses the back bumper switch to start it. This lets you reload IC with ease. Pressing the robot reset without touching the enabling back bumper switch will kill the serial output from your program. If this technique is not used, the serial output of your program will conflict with IC communication and prevent synchronization when you attempt to open IC again (You close IC in the next step). You will then have to go through the inconvenience of doing a first time load of IC again.
3) Close IC,
4) Open Hyper Terminal configured as described above,
5) Connect Hyper Terminal to COM1 (select the phone on-the-hook icon),
6) Press reset to begin program execution,
7) Press the back bumper to proceed with program execution (the test for the back bumper should be the very first statement in your program),
8) Behold! Your program should write to the screen.

Example TJ PRO™ Screen Output Progarm

1) Read the program *sensortjp.c* in the *TJPRO_Experiments* directory. Try to understand how it does what it does. This program outputs the bump and both IR sensors to the PC screen on a continuous basis. The program explains and uses VT100 character escape sequences to clear the screen and move the cursor to different lines and columns. To erase the data fields, the code prints blanks and then backspaces over them before the next data sample is printed (A *printf()* for an external terminal would obviate the need for this hack!).

2) Apply the above procedure to `sensortjp.c` and watch the data update on the screen as you put your hand in front of the IR sensors or push any combination of bumper switches.

10.5 Infrared Experiments

The IR emitters, the blue eyes on top of the plate, emit 940 nanometer electromagnetic radiation (infrared) modulated at 40KHz. This light expands out in cone, strikes an object. Some of the light is reflected back. Underneath the plate, on the right and left, the IR detectors receive the reflected light and output a voltage proportional to the received intensity. We will use this sensory capability in the following experiments.

For the IR experiments, use the program `sensortjp.c` in `TJPRO_Experiments` directory on the *tjproed01* diskette.

Experiments
Follow the procedure outlined in **TJ PRO™ PROGRAMS THAT WRITE TO YOUR PC SCREEN**

1. Explore TJ PRO™'s Sensory Capabilities.

Observe the sensor outputs printed on your terminal screen.

Place an object or your hand to the right front of the robot. Observe the change in the IR sensor readings. If the object is close enough, you will have certainly changed the right front IR detector reading. Depending on your placement of the object, you may or may not change the left front IR detector reading. Repeat with the object at various distances.

Press the back bumper and various combinations of the switches behind the front bumper. Explore the robot's "visual" limits. Try objects of various sizes and placements. These visual limits will help you understand why TJ PRO™ fails to detect objects sometimes and bumps into them.

a. Determine the minimum and maximum readings of the two IR detectors. Are they the same? What impact would any differences have on collision avoidance?
b. Can TJ PRO™ see the point of a small shoe on the floor in front of it? Can it see an object suspended several inches above its plate?
c. Determine the minimum height of a white wall that the robot can see.
d. Determine how low an overhang can be without the robot seeing it.
e. What is the smallest rectangular wall surface the robot can see?
f. What is the smallest right circular cylinder (chair leg!) the robot can see?
g. Explain why TJ PRO™ IR "vision" these and other "blind" spots.

2. In this experiment you will measure IR detector intensity as a function of the distance from a reflecting wall surface.

Procedure

Place a flat, white cardboard, paper, or plastic "wall" in front of the right IR detector. I recommend taping a white sheet of paper on the side of a cardboard box. Make the front surface of the right IR detector package parallel to your "wall" (the right IR detector can is the shiny metal cube underneath the top plate on the right-hand-side). A key to the success of this experiment is to keep this parallel alignment as you move the wall various distances from the robot. For example, you can slide the cardboard box along straight tile-lines on a tile floor.

Questions

a. *How can you tell the robot IR detector squares off against the wall? (Hint: Place TJ PRO™'s front about 8 inches from the wall and take an IR reading using IC. In place, manually rotate TJ PRO™ a small amount and take another reading. Be careful not to move the robot closer or further away from the wall! If the IR reading got bigger, rotate a bit more and take another reading. If it got smaller rotate back the other way and take another reading. Keep rotating in the direction that causes increases in the IR readings. The IR readings will then level off and not change for small rotations. Eventually the readings start to decrease to one side and then the other of the flat high region you found. Pick the middle of that region. That point should be the geometric arrangement you need between the wall and the IR detector. The robot sensor does not exactly square off with the wall, but does give the greatest measurement sensitivity. Can you explain why?*
b. *With the proper orientation of the robot and the wall. Measure the IR intensity with the wall touching the robot and in increments of 1 inch all the way out to where the reading no longer is affected. Plots this data on a graph. The IR readings on the y-axis and the distance in inches on the x-axis.*

Data Analysis (Advanced)

a. *Did you get a strange value for the wall up against the robot? Explain.*
b. *At what distance did the reflected IR light from the wall have no effect on the IR reading?*
c. *Consider only the part of the plot where the data changes each successive measurement. Solve for the coefficients (a,b,c) of the quadratic* $I = ar^2 + br + c$ *from three simultaneous linear equations. Obtained the three equations by finding I, on the curved part of your plot, for 3 widely separated values of known distance r. For more sophisticated users, use a minimum least-squares fit to a quadratic equation.*
d. *Plot the resulting quadratic with the coefficients you found, together with the measured curve. Compare at several points. Does the quadratic curve seem to fit the points in between the ones you actually selected to compute it?*
e. *What is the maximum error between the quadratic and the measured plot for the curved part?*

f. From your data, describe two ways to determine the distance to an object given the reflected IR intensity? Will your answer be correct if the object is not the same material and color as the one used to get the data? Explain.

11 APPLICATIONS

The directory *TJPRO_Applications* contains several TJ PRO™ "toy applications". You can approach these applications in several ways. One fun technique, perhaps the hardest, load a program you know nothing about (don't cheat and read the descriptions below, or the code!) and run it and try to guess what it does. This, in general, is not easy! Next, program the behavior you observed and compare the robot's behavior executing your program with its behavior executing the original. Does the robot really behave the same for the two programs? This question is not trivial. Comparing behaviors of robots is not yet a science. Now, compare your code with the original code. The two programs probably differ greatly in detail, if you didn't look-ahead!

Another approach, similar to what I have done throughout the manual, is for you to write a behavior program from behavior specifications. Here, too, you can compare the robot's behavior running your program and running the given solution. This approach is taken for the program `faceoff`.c in the paragraphs below.

A third approach to these programs: read the code and descriptions. Load the code and observe the robot in action. Was it what you expected? Does the robot do strange things in strange circumstances? Think of ways to improve the behavior program or expand the behavior capabilities using one of the given programs as a base.

Finally, of course, make copies of these programs, modify them to change the behaviors by a little or a lot, and test your changes. As you become more experienced you will write applications of your own, applications limited only by your imagination.

11.1 Programming a Behavior from a Specification

In this section you will program a behavior for the robot from a description of that behavior. I identify and name some key variables to provide a common basis for discussion and help you get started. You do not have to use those variable names, or even develop the algorithm in the direction that I layout, of course. But, the learning game requires you to meet specs, or have fun, or diverge in your own direction with an off-the- wall "non-solution"…as long as you get something out of the game!

.

Program a Face_Off Behavior

Write an IC program `faceoff`.c for TJ PRO™ that turns the robot to face a wall and stops it there.

Detailed Description of Face_Off Behavior

TJ PRO slowly rotates counterclockwise about the left wheel (`speedl=0, speedr=12`) until its front faces parallel to a wall. The robot rotates for `move_time = 50L` milliseconds and then stops for `ir_measure_time = 350L` milliseconds to give the IR detectors time to stabilize on their new values. This process repeats until both IRs read the same value.

I place the robot at an angle with the right IR sensor further from the wall than the left IR sensor. The left IR sensor is about 7 inches from the wall. As the robot rotates counterclockwise, the right IR values slowly increase, so when it matches the left IR, the motion is complete. You can actually place the robot at any orientation near the wall. It moves so slowly, however, that you may not have the patience to wait for it to rotate almost 360 degrees!

The value of the IRs must indicate the presence of an object. Since the baseline IR values hover around the low eighties, I selected `irr_threshold = 97`, meaning the right IR must read at least that amount or no wall is assumed to be nearby. By symmetry, if the two IRs are identical and the surface is flat with uniform reflectance, the robot's front-to-back diameter should be perpendicular to the surface when the IRs read the same value. Because the dc motors on TJ PRO cannot be precisely controlled, it is impossible to get the robot front-to-back axis exactly perpendicular, even for a really good surface.

Parameter Manipulation

You might want to play with the various parameters `speedl, speedr, move_time` and `ir_measure_time` to test the various tradeoffs. If so, be sure to declare all of them `persistent`.

Comment: Although you may have calibrated the servo motors, they still drift, so low speed control is iffy.

A solution to this exercise is `faceoff.c` in the `TJPRO_Applications` directory.

11.2 Application Program Descriptions

Brief Description of `attractjp.c`

TJ PRO™ will be attracted by anything that moves close to it, or it moves close to! If TJ PRO™ bumps into anything it becomes shy and backs off and then waits for something to move close to it again.

Playtime with `attractjp.c`

Put the robot in the middle of a room with lots of free space around it. Reset the robot and touch the back bumper switch to start the robot. Nothing happens. Wave your hand

in front of the robot. It's alive! Now, slowly move your hand about. TJ PRO™ will follow your hand motion unless it locks onto your leg and bumps you. Miffed, TJ PRO™ backs off and turns some random angle and pouts. After pouting for a second, the robot will head right for any close by object or, if none is near, wait until an object comes close.

Questions

a) Roughly, how close does your hand or object have to be to activate attraction*?*
b) What program parameter will change the answer to a)?[9]
c) Roughly determine how close does your hand has to be for the robot to track it?
d) What program parameter will change the answer to c)?[10]
e) Make a copy of this program and make changes to the parameters and observe the behavior changes. Try to predict what will happen beforehand. How often are you surprised at the robot behavior resulting from your changes?

Brief Description of *`avoidtjp.c`*

This program realizes a simple collision avoidance program. TJ PRO™ will read each IR detector, and turn away from any obstacles in its path. Also, if something hits TJ PRO™'s bumper, it will backup, turn, and go forward again.

Playtime with *`avoidtjp.c`*

Put the robot anywhere in a room. Reset the robot and touch the back bumper switch to start the robot. The robot will begin to move about avoiding bumping into obstacles. Sometimes the robot backs up and jams into an obstacle without responding until the backup action timeout occurs. The program doesn't account for that situation. In the section on multitasking we will see how to easily correct for this behavior.

The robot may get trapped between two obstacles or caught in a corner and oscillate indefinitely. I call this behavior the Braitenburg Trap, in honor of Valentino Braitenberg whose first vehicle in synthetic psychology will invariably oscillate in a corner until it dies of exhaustion.[11]

Caution!

Do not let TJ PRO™ oscillate too long as it can overheat and damage the motor control electronics. Oscillations also increase the wear on the motors and gearboxes, decreasing their lifetimes.

[9] LOW_IR 95
[10] MED_IR 99
[11] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, 1984.

Questions

a) *What obstacles in your room does TJ PRO™ get stuck on/under/against? Explain why the robot got stuck for those obstacles.*
b) *Explain the Braitenberg trap. Why does the robot oscillate?*
c) *With the existing program, TJ PRO™ will not see certain types of black surfaces and bump into them. What program parameter will change the sensitivity of TJ PRO™ so that it might possibly see the black surfaces?*[12] *If you program the robot to "see" the black surface, what happens when the robot approaches light colored surfaces?*[13]
d) *Make a copy of this program and make changes to the above parameter and observe the behavior changes. Try to predict what will happen beforehand. How often are you surprised at the robot behavior resulting from your changes?*

Evaluation of *avoidtjp.c*

The robot will not perform well in both a light colored environment and a dark colored environment using `avoidtjp.c`. *In environments that have both light and dark obstacles, the overall performance can be marginal. How can this be corrected or, at least, be improved? If the robot could somehow learn to adjust IR thresholds according to environmental circumstances, surely its performance would improve. This brings us to the next program.*

Description of *avcaltjp.c*

This collision avoidance program that learns to adjust its IR sensitivity based on environmental factors. The self-calibration is based on bumper activity and average translational speed (not velocity). TJ PRO™ will read each IR detector, and turn away from any obstacles in its path that reads greater than the learned threshold. Also, if something hits TJ PRO™'s front bumper, it will backup, turn "randomly", and go forward. If something hits TJ PRO™'s back bumper it will go forward only for half the time a front bump causes the robot to backup, turn "randomly", and continue forward.

The novel feature of this program is its ability to adjust its behavior based on environmental conditions. The form of learning employed, while simple, is quite powerful. Whenever the robot bumps an object from the front, the control algorithm "presumes"[14] that the IR threshold value (*avoid_threshold*) must be too high and the objects being bumped do not reflect enough light to exceed the threshold before the robot bumps into it. The *avcaltjp.c* algorithm corrects for the undesired bumps by increasing the sensitivity of the robot motion control to lower IR readings. The process continues until there are no further bumps. For low or overhanging objects that TJ

[12] AVOID_THRESHOLD 100
[13] The robot becomes "shy" and will not approach as close to obstacles as it did before.
[14] To be more precise, the programmer who wrote the algorithm, namely me, presumes! I will tend not to make such fine, semantic distinctions, but the reader should always be aware of their underlying importance.

PRO™ cannot see, lowering the IR threshold is counter productive as the bump has nothing to do with the threshold being too high. Such situations "mislead" the algorithm, but only temporarily, and insignificantly, if such situations are statistically rare.

A new problem now arises. The robot can avoid "painful" bumps by simply not moving or spinning in place! I discovered this in my first attempt to get it to automatically calibrate the IR threshold that invokes aversion or evasive moves by the robot. The robot would "panic" after a number of "painful" bumps in close sequence and simply spin in the middle of the floor! To avoid these behavior *attractors*, there must be a counterbalancing driving force "urging" the robot onward. The program does this by positively reinforcing average forward motion during a fixed-sized, periodic time window. I experimentally determined 3 seconds for mixing light and dark environments: my study (light environment with brown stained, wooden baseboards and scattered cardboard boxes) and my kitchen (black baseboards all around). I found 5 seconds too long and 1 second too short. With this countervailing force, the robot might "panic" for a while, but, continued urging of forward motion by steadly decreasing `avoid_threshold` would get it moving forward again.

I limited the dynamic range of IR threshold adjustment between 90 and 120. These constants can be changed through

```
#define THRESHOLD_HIGH 120
#define THRESHOLD_LOW   90
```

The IRs typically do not read below 84 (everything is an obstacle) and above 127 (nothing is an obstacle), so working thresholds should not be too "close" (experimentally determined) to those numbers.

Since this program produces extremely interesting robot behavior, you will likely want to play with this program extensively. I recommend that you convert the above constant parameters, as well as the others discussed here, into *persistent variables* so you can conveniently change them from your PC on the IC command line. Otherwise, you have to change the program, unload the old version and download the new version. If you take this approach, do not forget to initialize them appropriately.

```
 1  if(mseconds()-mark_cal_time > AVERAGING_TIME)
 2    /*Adjust IR threshold if average speed not up to snuff*/
 3    {
 4      if((average_speed < AVERAGE_SPEED_MIN)&&(avoid_threshold <THRESHOLD_HIGH ))
 5        avoid_threshold +=3; /*Not going forward. Get less sensitive*/
 6
 7    /*Reset calculation window for running average of robot speed */
 8     n=0;
 9     average_speed = 0;
10     mark_cal_time = mseconds();
11    }
12   else
13     /*Calculate the average speed during a five second time window*/
14     {
15       n++;
16       average_speed = (n-1)*average_speed/n + (speedr + speedl)/2*n;
17     }
```

Figure 5. This code urges the robot to move forward.

The most difficult part of the program appears in Figure 5. The instantaneous translational speed of the robot equals *(speedr + speedl)/2*. By taking a running average of the robot translational speed (lines 15-16),

$$average\_speed_i = (n-1)*average\_speed_{i-1}/n + (speedr + speedl)/2*n;$$

the program nominally can determine if the robot is moving about. The average_speed being large, close to 100%, does not guarantee the robot is moving about since its wheels could be spinning full forward with the robot hung up. In such situations you have to rescue the robot anyway, so a “hung up robot” little matters to the correct functioning of the program under normal circumstances.

The variable *mark_cal_time* records the instant of time the running average of the speed begins to compute. After *AVERAGING_TIME* seconds (line 1), the program

```
 1 /* This "if" statement checks the bumper. If the bumper is pressed, */
 2 /* Tj will back up, and turn.                                       */
 3
 4   if(FRONT_BUMP)
 5   {
 6     motorp(LEFT_MOTOR, -MAXSPEED);
 7     motorp(RIGHT_MOTOR, -MAXSPEED);
 8     if(avoid_threshold>THRESHOLD_LOW)
 9       avoid_threshold -= 3; /*Bumped something. Get more sensitive*/
10      sleep(0.6);
11      turn();
12    }
13 /*
14   Sometimes the robot gets so sensitive it wants to backup permanently!
15   the following code causes the robot to become less sensitive and, thus,
16   get it out of the "backup attractor".
17 */
18    if(BACK_BUMP)
19    {
20      motorp(LEFT_MOTOR, MAXSPEED);
21      motorp(RIGHT_MOTOR, MAXSPEED);
22      if(avoid_threshold<THRESHOLD_HIGH)
23         avoid_threshold += 3; /*Bumped something. Get less sensitive*/
24      sleep(0.3);
25      turn();
26    }
27
28  if(avoid_threshold >THRESHOLD_HIGH)
29    avoid_threshold=THRESHOLD_HIGH ; /*Don't get too insensitive!*/
30  if(avoid_threshold<THRESHOLD_LOW)
31    avoid_threshold=THRESHOLD_LOW; /*Don't get too sensitive!*/
32
33    sleep(0.033);
```

Figure 6. This code adjusts the IR threshold when bumper contact occurs.

stops computing *average_speed*, checks to determine whether the robot has performed better than *AVERAGE_SPEED_MIN* (line 4), and, if not increase the *avoid_threshold* by 3 (line 5). Next, the code initializes the parameters for computing *average_speed* once again (lines 8-10).

The code in Figure 6 adjusts the threshold whenever bumps occur (lines 8-9 for front-bumps and lines 22-23 for back-bumps). Front-bumps lower the IR threshold, making the robot more sensitive in avoiding obstacles. Back bumps increase the IR threshold, making the robot less sensitive to obstacles. The latter may seem counter intuitive or just simply wrong. Without it, however, the robot can get in such a “panicked” state that it backs up full speed and jams into a wall or other object and strains and whines as its wheels spin on the floor, not a pleasant sight or sound! The average speed

adjustment will eventually correct this behavior, but it takes 3 seconds per adjustment of 3 points, so the robot might be in "pain" for 12 seconds or more!

The rest of the front and back bumper code you have seen before. Lines 28-31 insure that the code does not generate values of the IR threshold above the specified range limits.

Playtime with *avcaltjp.c*

Put the robot anywhere in a room. Reset the robot and touch the back bumper switch to start the robot. The robot may begin to move about avoiding obstacles. Since it starts at the high IR threshold range, it will probably avoid ok, even in a cluttered environment. Restart executions cannot guarantee a high threshold, remember, the variable is persistent! In restart situations the robot may not behave as well in a cluttered environment.

Make the environment too cluttered and the robot oscillates with "indecision". There are actually two distinct causes for indecision:

1) Everything appears too close to the robot and it rapidly turns here and there, or spins, trying to find a free path, but getting nowhere.
2) When the indecision manifests as oscillations back and forth from left to right, the cause is the Braitenburg Trap. You probably saw this behavior when TJ PRO™ ran `avoidtjp.c`.

If there is only marginally enough room for the robot to pass between two obstacles with an IR reading of 120, the robot will oscillate a great deal before passing through, although the process is quite painful to watch. If the opening is too marginal, the oscillations will take too much time (>15 seconds). In such situations, rescue the robot to prevent possible motor burnout.

Watch the performance of the robot. Expose the robot to a variety of rooms and situations. If the environment changes radically from light to dark obstacles, for example, the robot will start bumping into obstacles for a while until it adjusts its threshold to accommodate the new situation. If the switch is from dark to light obstacles, the robot may back up in "fear" or spin indecisively until it eventually compensates.

In mixed environments the robot will forever be adjusting its threshold, occasionally bumping into things, occasionally being "shy" or "fearful". Depending on the detailed nature of the environment, the robot may have extended periods of nominal behavior as well.

Questions

a) Do you consider that the TJ PRO™ executing `avcaltjp.c` *actually "learns"? Carefully defend your position on paper and file it for future reading.*

b) Ask people who knows nothing about robots to observe TJ PRO™ in action. Do they detect any "learning"? (Their answer will certainly depend upon their observational capabilities). Help them (bias them?) to see how the robot changes its behavior. How do they respond now?

c) Change the following parameters to persistent variables. Be sure to declare them appropriately.

Parameters:

```
#define AVERAGE_SPEED_MIN  50    (Range: 0 to 100)
#define THRESHOLD_HIGH 120       (Range: 84 to 127)
#define THRESHOLD_LOW   90       (Range: 84 to 127)
#define AVERAGING_TIME  3000L    (Range: 0 to infinity)
```

Vary these parameters and observe robot behavior changes. Can you quantify what you see (To perform quantitative analysis of intelligent machine behavior, in general, is a research question!)? In particular, note what happens for extreme values of the various parameters. The upper bound on the parameter AVERAGING_TIME, in principle, does not exist. For practical matters, after a certain value (10000? Less?) the robot decision time frame becomes so large that, were it an insect, it would surely get eaten before it could "decide" about any behavior changes!

Brief Description of `racetjp.c`

TJ PRO will be attracted by 40KHz modulated IR light, say from your TV remote! If the robot bumps into anything it becomes shy and backs off and stops. Point your TV remote at it and press any key. The robot will start up again and follow the light from the remote. Make TJ PRO "heel" as a pet dog using your TV remote!

A really interesting thing about this program is that it is exactly the same as `attractjp.c` with the omission of one line, IR_ON! The robot does not look for reflected light generated by its own IR emitters, rather, it looks for an external source. Any 40KHz modulated IR will serve as a source, and your TV remote generates such a signal. Now you have an IR remote controlled robot. To stop the robot, just let it bump into something!

The program is called "race" because you can line up several TJ PRO™ robots and race them. Of course, the 'PRO™ cannot tell who is its owner controller, so it can make for some fun games with people trying to sabotage each other's control!

Games with `racetjp.c`

You can device an incredible number of games with TJ PRO™ executing this program. I will list two examples.

Single TJ PRO™

1. Develop an obstacle course with boxes in an enclosed arena, say 10 feet by 10 feet square. The arena can be any shape or size, but a light colored, six-inch wall should enclose it. Competitors cannot enter the arena. Set up a home position at one corner and a goal position at the diagonally opposite corner. See which competitor can take TJ PRO™ from home to goal the fastest using a TV remote.
2. Forget about the arena. Just make an obstacle course. My bedroom is a natural obstacle course, so this part does not take any effort! Mark off a home and a goal position. Race against time again. A problem with this version of the game is controlling the controllers. Where can the controllers legally position themselves during the course of a run?!

Multiple TJ PROs™ : ROBORACES™[15]

Make TJ PROs™ into *ROBORACERS™*[16]. Put racing stripes and numbers on them. Layout a 20 feet long, straight, racecourse with START, FINISH and several LANE lines. You can, at your own risk, use colored tape or washable colored ink to make the lines.

Be careful!
Make sure whatever you use to mark lines does not stain your floor!

Line up the robots on the START line and the competitors on the FINISH line, with remotes in hand and THUMBS-UP! The competitors cannot move from their posts during the contest, no matter how badly they want to help their robot or do something unspeakable to someone else's robot. Assign a Race Track Announcer to call the race This can be a most entertaining assignment! When the race track announcer commands THUMBS-DOWN, the competitors press the TV remote buttons and the race is on. The first TJ PRO™ to cross the finish line is the winner (regardless of who got it across!). Do not tell the competitors that they can control their opponents 'PRO. They will discover that soon enough amidst the laughter and heat of competition!

12 MULTITASKING BEHAVIORS

In this section you will learn to use the multitasking facility offered by IC. To quote the IC manual,

[15] ROBORACES™ is a registered trademark of Mekatronix.
[16] ROBORACERS™ is a registered trademark of Mekatronix.

"One of the most powerful features of IC is its multi-tasking facility. Processes can be created and destroyed dynamically during run-time.

" Any C function can be spawned as a separate task. Multiple tasks running the same code, but with their own local variables, can be created.

" Processes communicate through global variables: one process can set a global to some value, and another process can read the value of that global.

" Each time a process runs, it executes for a certain number of ticks, defined in milliseconds."[17]

Essentially, any C function `f()` may be defined as a process. What distinguishes such C functions from others? Nothing really, except that in `main()`, or some other function, the statement

```
start_process(f(),<ticks>, <stack _size>);
```

starts the process `f()` on a virtual stack-machine for `f()`'s own private use for `<ticks>` milliseconds (optional) with an optional stack size specified by the last parameter. The `<ticks>` default equals 5. The stack size default equals 256 bytes, an adequate amount for most TJ PRO™ applications, but sometimes you might want to specify smaller or larger stacks. Typically, `f()` runs forever own its own virtual stack-machine by enclosing all statements within the function by `while(1){….}`. But wait! There is only one real processor to run all the virtual machines! To be fair, the virtual machines share the real processor on a round-robin basis, in the order they were created, for the default of 5 ticks (5 milliseconds each), unless specified differently by `<ticks>` above.

12.1 IC Multitasking in Operation

To develop an appreciation of how IC implements multitasking, consider the program `multitask1.c` listed in Figure 7 and Figure 8. Four IC functions, `a(), b(), c()` and `d()` share the microprocessor on the robot. The IC scheduler executes each process the default 5 ticks, suspends execution of that process and passes control to the next process in line.

[17] IC Manual pp. 158-159.

```
/*****************************************************************************
 * Title        multitask1.c                                                 *
 * Programmer   Keith L. Doty                                                *
 * Date         August 27, 1998                                              *
 * Version     1                                                             *
 *                                                                           *
 * Description                                                               *
 *   Run four concurrent process, only once, at different rates, using       *
 *   msleep. Press the back bumper switch to START. Each process writes      *
 *   a unique two character string before the sleep and after its sleep      *
 *   statement. By watching the characters display on your PC screen         *
 *   you can get an idea how the multitasking facility in IC works.          *
 *   Each process only runs once.                                            *
 *                                                                           *
 *****************************************************************************/

/*************************** Includes *********************************/

/*
  Before loading this program, be sure you load the library files
      twoservo.icb
      twoservo.c
      motorp.c
      serialtjp.c
*/

/************************ End of includes *****************************/

/************************** Constants *********************************/

#define BUMPER      analog(0)

/************************ End of Constants ****************************/

void main(void)
{
 while(BUMPER<126);
 init_serial();

 start_process(a());
 start_process(b());
 start_process(c());
 start_process(d());

}
```

Figure 7. Structure of an IC program to multitask five processes. The code for each process is shown in the next figure.

```
void a(void)
{
  write("a1");
  msleep(10000L);
  write("a2");

}/*end a()*/
/*********************************************************************/


void b(void)
{
  write("b3");
  msleep(1000L);
  write("b4");

}/*end b()*/
/*********************************************************************/


void c(void)
{
  write("c5");
  msleep(100L);
  write("c6");

}/*end c()*/
/*********************************************************************/


void d(void)
{
  write("d7");
  msleep(10L);
  write("d8");

}/*end d()*/
```

Figure 8. Each of these four multitasked processes take different amounts of time to output two strings of two characters each. The `msleep()` command forces the processes to consume more than the allotted five ticks.

To help you visualize the execution sequence each process simply writes a string of two characters to the screen at two different times. Once before an `msleep()` function call and once after it. The duration of the `msleep()` function call exceeds the allotted time for each process. The IC scheduler will invoke process a`()`, for example, approximately 2000 times before the process completes execution.

Experiment with `multitask1.c`

The directory `TJPRO_Experiments` holds `multitask1.c` as well as `multitask2.c,` and `multitask3.c`. Refer to **10.4 Terminal Output without IC** to start up this program, which uses serial output to the PC screen. After you press reset and push the back bumper, the Hyper Terminal VT100 simulated screen should output the sequence `a1b3c5d7d8c6b4a2`. The sequence `a1b3c5d7d8c6` appears instantly. Approximately, a one-second delay separates the printing of `c6` and `b4` and nine more seconds elapse before `a2` prints. From the code you can see that each process starts, but, because the `msleep()` time statement takes more than the allotted five ticks, each process is suspended in turn to activate the next one. This is why you see `a1b3c5d7d8c6` almost instantly. The same four processes are activated over and over, in the order *abcd*, until all have run to completion.

Questions

1) *How long does it take for this program to execute?[18] You can use the WIN95 clock to measure seconds ticking as you watch the Hyper Terminal screen print out the characters.*
2) *Add the independent process*

```
void e(void)
{
while(1)
  {
  write("e9");
  msleep(1000L);
  write("e10");
  }/*end while(1)*/

}/*end e()*/
```

and call the resulting program `multitask2.c`.

3) *Before executing the change in step 2), predict what will happen on the screen. Were you surprised? If so, make other changes and try to predict what will print after those changes. You might want to do this until you feel comfortable with the idea of multitasking.*
4) *When you did this change, did you have to make any changes to the program other than add the new function and the statement* `start_process(e());` *?[19] In programming multitasked robot behaviors, seek to code independent processes that interact only through global variables, if at all. This decoupling of processes, a favorite software engineering discipline, can greatly simplify your coding life and reduce coding woes.*

[18] About 10 seconds, the same time as `a()` executing by itself.
[19] No.

Program Assignment
Change all the processes in multitask1.c to tasks that execute forever. Predict what the screen will print (write it down on paper, no cheating!) and then compare with the actual display.

Question
You will probably see an occasional splitting of two character sequences. Something you did not see before in the earlier experiments. Can you explain why this happens?[20]

12.2 Multitasking in Robotics

The scope of this manual does not permit a deep probe into the use of multitasking as a method of programming robotic behaviors. A sequel to this manual will take up the issues in depth.

Many, if not most, simple robot behavior programs do not require multitasking. The construction of complex robot behaviors from independent, simple behaviors running at the same time motivates and supports the argument for multitasking. Unless you have a great deal of experience with multitasking and real-time programming, however, you may find multitasking robot behaviors not for you. Subtle task interactions can lead to days of frustration and difficult to find bugs.

To wrap up this manual, I will integrate the avoid and attract behaviors along with the IR threshold auto-adjustment processes. The resulting program, `animatjp.c` , appears in the directory `TJPRO_APPLICATIONS`.

The program `animatjp.c` runs six processes

1. sense():
   Measures IR return.
2. average_speed_calculation(): state = 50
   Computes average forward speed of the robot.
3. arbitrate(): state = 40
   Controls which of the three processes, spin, avoid, or attract, executes next.
4. spin(): state = 30

[20] Obviously, the process times out before it completes the write statement. This answer is a bit too glib, however. If the IC scheduler could perform zero-time context switching, switching from one process to another, and if each processes took integer multiples of 5 ticks, then split writes would not occur for the sleep times chosen. But, context switching does take time. As this tiny amount of time accumulates, it eventually becomes large enough to affect the write output sequences.

Spins the robot in place while looking for a source of IR emissions with the robots IR emitters off.

5. attract(); state = 20

Moves the robot toward any 40KHz source of IR radiation, for example, a TV remote.

6. avoid(): state = 10

Moves the robot about, avoiding obstacles and auto-adjusting the robot's sensitivity to IR.

Program Operation

Process arbitrate() invokes spin() for a fixed amount of time. Process spin() checks for external 40KHz modulated IR radiation. If spin() finds IR, it will stop turning the robot. After spin-time-out, arbitrate() halts spin() and invokes the two processes avoid() and average_speed_calculation(), if spin() did not find IR , or attract(), if spin() did find IR. As long as the IR reception is above a certain minimum, the robot continues to executes attract(). If IR is lost, arbitrate() invokes spin() again and the process repeats.

Playing with `animatjp.c`

With TJ PRO™ on a stand, download the program, press reset and push the back bumper switch closed. Monitor *`avoid_threshold`* while the program executes. Notice that this variable decreases by 3 each time you press the front bumper and increases by 3 each time you press the back bumper. The former is "punishment" for the robot being too insensitive to its IR readings and bumping into too many objects. By decreasing *`avoid_threshold`*, the robot becomes more aware of objects further away. The latter is "punishment" for backing up and bumping into things, presumably because the robot is so "shy" (too sensitive to IR). Increasing the *`avoid_threshold`* decreases the range at which the robot can detect objects.

Monitor the variable *`state`*. Watch how the processes change. After you see the pattern, take any TV remote and hold it in front of the robot with any button pressed, say a channel-digit. Make successive reads of state until *`state = 20`*. Keep the button pressed and slowly wave the TV remote from side-to-side. The wheels should change their speed to effectively turn the robot towards the remote.

Since the TV remote's IR signal is so strong, the robot IR detectors may both saturate, even when you point directly only at one of them. In such cases, the remote appears to be straight ahead and the robot turns both wheels equally. You might have to point 45 degrees away from center to generate different IR detector responses, hence, an IR controlled turning motion. To gain better IR control, you can partially block the remote's output LED with black electricians tape to make the IR signal more collimated, hence, directional. You can examine the IR response through the variables *`irdl`* (left IR detector) and *`irdr`* (right IR detector).

Place TJ PRO into an environment with different colored walls. Watch the robot's response as it moves about.

Questions

1) *When the robot falls into a Braitenberg trap does it remain stuck there? Explain what happens.*[21]
2) *How long can the program stay in the* `attract()` *process?*[22] *You can hypothesize an answer based on experiments. What happens if you keep a TV remote button pushed down while pointing the remote at the front of the robot? You can also read the code for the* `arbitrate()` *process and find out.*
3) *Run the robot in a uniformly light-colored environment and then switch to a uniformly dark-colored environment. What happens initially?*[23] *Switch environments again and observe the initial behavior. If the light-colored environment is cluttered and the obstacles are separated by just a few robot diameters, what do you expect the initial behavior to be?*[24]
4) *If you run the robot in an environment with light and dark wall and obstacles, what behavior do you predict?*[25]

13 FURTHER EXPLORATION

This introductory manual only hints at the tremendous experimental possibilities available to you with your TJ PRO™ robot. The basic understanding of TJ PRO™'s capabilities that you have received from studying this manual and carrying out its experiments will help you to develop truly sophisticated robot behaviors.

Good luck and enjoy☺

[21] Typically, `spin()` will break the trap.
[22] The program stays in `attract()` as long as external IR is detected.
[23] The robot frequently bumps into walls, and then not at all as it "learns" the new environment.
[24] TJ PRO should "shy" away from the light obstacles at a larger distance than from dark obstacles. If the obstacles are several diameters apart from each other, the robot might tend to spin ineffectively until the average speed requirement kicks-in and makes the robot less sensitive to IR.
[25] The robot will "seek" an IR threshold sensitivity that will prevent bumping into obstacles and at the same time produce a net 50% average speed. This goal may or may not be consistent with reality.