Nav and Control Software

Rev 8/18/05

Comments/corrections to Alex Brown rbirac@cox.net

The Nav and Control software provides several functions for the Leaf robots:

communications with the microcontroller; receiving data from the microcontroller and making it available to the Lisp software as well as internal use, and transmits commands to the micro.
converts low-level motion commands from the Lisp software to microcontroller commands.
performs navigation based on high-level commands from Lisp.
provides an optional simplified animation image
provides a simple terminal like console on which status and warning messages may be printed.
provides a data logging function.

The top level function ("main") is NavAndControl.exe.

This process contains several separate functions. These include:
Robot Controller. Creates Windows used by various functions and provides for initialization, periodic execution and shutdown of all the following subfunctions.
MicroUSB. Provides communications between the microcontroller and the laptop via USB.
Comm. Provides data to other processes via shared memory in a dll.
Animation. Provides animated display of a face on the laptop screen.
Navigation. Provides for navigation and mapping of the environment as directed by the Lisp function.

RComm.dll This is an independent dll () which is called by processes which wish to communicate (such as Comm above).

All of the above software was generated using Microsoft Visual C++ 6.0 (professional edition).

Main classes:

NavAndControl.cpp:
This module is the main controller. It provides generation of windows on the laptop for the animation display and a scrolling terminal display which can be used for failure warnings, troubleshooting, status etc. It also provides for periodic execution of each of the other NavAndControl subfunctions.

This program is derived from a book called "Windows Game Programming for Dummies" by Andre LaMothe. For those of us who are newbies to Windows programming, this book provides an excellent short introduction to how Windows programs work. In addition, game programming has a lot in common with robotics programming in that the goal to provide fast efficient real-time operation. Part of the goal of the book is to provide skeleton frameworks of Windows programs which can easily be modified using simple C code. That objective is similar to the objective of this program; to provide a prototype robot program which can easily be modified by others.

So, what does this program do?

1. It reads in various configuration (*.ini) files for use in configuring the windows generated and for configuring the microcontroller etc.
2. It defines and registers and shows the two windows: main and console
3. A function called RobotInitialization() is called at this point. This function creates instances of each of the main functional classes and calls initialization routines in them when necessary. Pointers to

each class are defined and made global for use by the other functional classes. It also downloads the configuration data to the microcontroller. Finally, sensor objects are created for each input.

4. At this point, the Windows program enters its main event loop. This loop normally waits for a Windows message to arrive (e.g. key pressed) and then processes the rest of the loop. For a real time program, this loop is modified so that a Windows message is tested for and processed if it exists. If not, the loop continues anyway and calls a real-time executive program called RobotMain(). RobotMain() controls execution of all subfunctions. It concludes with a Sleep (or equivalent) function which controls the rate at which RobotMain() is called. In Windows, a Sleep function relinquishes all remaining time allocated to RobotController to other processes. Right now, it performs the following functions:
   Animation (if turned on),
   Receive microcontroller data message,
   Check that microcontroller is initialized, if not send config data.
   Read command message from Lisp,
   Process all commands (from Lisp and other sources), do arbitration,
   Navigation,
   Send microcontroller command message,
   Send data message to Lisp.
   Sleep until end of 100 msec. frame.

5. When the main window receives a WM_QUIT message (like when you click on the exit "X" button on the top of the window), it leaves the main event loop and calls a RobotShutdown() function before the program closes. The RobotShutdown() function calls shutdown functions in the various subfunctions, when necessary, then deletes objects which have been created. Finally, the program returns to Windows.

A “feature” of this setup is that, if you are happy with the basic windows screen layout, you don't have to modify the Windows stuff, you just write code in the RobotInit, RobotMain and RobotShutdown functions

A general note on real-time control programs. The intent is to provide the same effect as if all functions are running continuously. For example, the Navigation function must be continually evaluating sensor data and the robots motion. If the Animation function decided to spend 2 seconds reading some files, the robot could crash in the meantime. If Robot_Main() is being executed at 10 times per second (as controlled by the Sleep() statement), all functions called by Robot_Main are meant to execute every 100 milliseconds. As far as possible within the Windows environment (which is still a bit of a mystery to me), none of the functions called should have any delays that cause the next cycle to be delayed. Basically, that means software should never WAIT for something to happen unless you KNOW that the time delay is trivial. If some data is not ready, skip it and look again on the next cycle.

MainWindow.cpp

This class defines, registers and shows the main window. The size and location of this window is defined in NavAndControl.ini and the parameters are sent by NavAndControl.exe when the Create() method is called. The window is normally intended to be the full screen.

The main window event handler is also included in this file. It doesn’t do anything at this time.

ConsoleWindow.cpp

This class defines, registers and shows the console window. The size and location of this window is defined in NavAndControl.ini and the parameters are sent by NavAndControl.exe when the Create() method is called. The window is normally intended to be at the bottom of the full screen.

The console window event handler is also included in this file. Its primary function is to implement a simple scrolling text display for data, status, messages etc. Such messages are sent to the console as strings using the Print() method.

One additional function is to read the “R” and “S” keys that call for the data logging operation to be started and stopped. (‘R’ is for Record, ‘S’ is for Stop).

MicroUSB.cpp

This class handles all the communications with the microcontroller.

The init() method finds the FTDI USB interface chip on the bus and opens it.

The shutdown() method closes the FTDI USB inteface.

The receive() method reads whether there are enough characters in the USB buffer to provide a full microcontroller data message. If so, it dumps excess characters until only the most recent message remains. It then reads that message into a structure USBDataMsg. Finally, some data conversions and calibrations are performed, generally by calling a class object with the microcontroller data.

The send() method compiles command messages and sends them to the microcontroller. One command message can be transmitted per real-time cycle. Several different messages may be transmitted selected by the message type sent as a parameter of the method. Three of the message types send configuration parameters to the microcontroller. These config messages are requested by NavAndControl.exe at RobotInitialization time, or at any time during operation that the configuration data is lost (e.g. if the microcontroller is reset). The remaining message is the Command message which sends motion commands and other data to the microcontroller to take action on.

A message is transmitted every cycle and is (or will be) used by the microcontroller to indicate that the laptop is operative (the idea being that if it is not, the microcontroller should bring the robot to a stop). If a new message is available, that message will be transmitted. If not, the previous message will be repeated. A timetag (sequence number) in the message tells the micro whether the message is new or not. A repeated message keeps the same timetag as previous transmissions.

Comm.cpp

This class handles communications between the NavAndControl software and the Lisp software. This communication takes place through shared memory in a .dll called Rcomm, which will be described elsewhere.

The SendLispData() method compiles a message from the data received from the microcontroller and other data generated locally. This message structure is then copied to the RComm shared memory.

The ReadLispCmds() method reads the shared memory in Rcomm and stores the Lisp message to a LispCmdMsg structure.

ProcessCommands.cpp

This class takes the commands sent by the Lisp software and/or other sources and performs arbitration to determine which source has control and subsequently passes the commands on to other functions for handling.

Most low level commands are passed directly to the MicroUSB class for transmission to the microcontroller. All navigation commands are passed on to the Navigation class for handling.

Generally, commands received from Lisp will have priority over commands from other sources. The arbitration logic isn’t fully implemented yet.

Navigation.cpp

Now the plot thickens! This class will perform all the navigation. Currently, the class is very simple and just reads low-level (such as "forward" and "stop") commands from Lisp and converts them into command messages to the microcontroller. This is done with the Update() method.

In past development testing, this class also performed higher level navigation functions such as: wander around avoiding objects, or run a specific contest sequence. These functions were implemented with case statements and were getting way too messy. They will be put back in using, perhaps, a table driven method.

Other navigational functions such as maintaining the Directional Gyro in the microcontroller, updating the X/Y coordinates in the microcontroller, using sensors to do localization and navigation will also be in this class.

The only other method is Init(), which sets the current command to Stop.

Animation.cpp

This class provides a simple animated face for the robot. This optional face may be used if you have a laptop with relatively low performance and can't afford the throughput required for the CSLU animation package. My 2.4 GHz celeron with no graphics accelerator needs 80% throughput to run the CSLU package, and less than 5% for this simple image.

This class is chosen to be operative by the [Animation] On parameter in NavAndControl.ini. A value of 1 turns the animation on and a value of 0 turns it off.

Emotion data is received from the Lisp software via shared memory in the RComm.dll.

A collection of bmp images is in the AnimationImages folder. In general, these images provide a number of versions of what a particular feature may look like. For example, a mouth transitioning from smile to frown.

Animation is achieved by using the emotion data to compute a new target value for each feature, then to sequence through the versions, one at a time each cycle, until the new target is reached. This method was derived from Sam's Teach Yourself Game Programming by Michael Morrison. The Bitmap.h and .cpp files are taken from the book.

The Start() method loads all the bitmaps into memory, creates buffers to draw the animation to, draws the basic background head as a starting point, reads the emotion matrix data from the ini file and sets all feature indexes to zero so that they will all be drawn on the first pass of update().

The Update() method reads the emotion values from the RComm dll, computes the new targets and ramps the features to the targets. The ramping is done by advancing one frame per Update call until the feature moves from the current value to the new target. It also controls the red beacon display on the robot's head, which rotates when the microcontroller is active.

The End() method deletes the bitmaps and associated buffers.

Sensors

This class processes sensor data. Each sensor is represented by an instance of a class defining that sensors characteristics; e.g. battery, GP2D12, SRF04, RCservo. Each sensor has a Set method which is used to supply it with the associated data from the microcontroller; and a Value method which returns the computed value. Each instance of a sensor converts the data to engineering units and performs calibrations as defined in the sensors.ini file.

Datalog

This class creates disk files with data stored down every cycle. Data logging is started and ended by hitting keys on the laptop. The "R" key to start recording, and the "S" key to stop. Note that focus must be on the console window when the key is hit.

When Init() is called (when "R" is pressed), a data file is created in the local directory. Each file automatically is named "Data" + the date and time + .txt. E.g. Data 05-18-05 08;49;25.txt . A set of header data is sent to the file which defines how many parameters are being recorded, the name of each parameter and default ranges for each. There is a separate program called Dataplot that can read these files and display the data on the screen.

Once data logging is started, a set of data is sent to the file for every executive frame (i.e. every 100 milliseconds).

When logging is stopped, by hitting "S", recording is stopped and the data file is closed.

INI

This class reads configuration (.ini) files associated with this program and makes the data available to other functional modules via global variables. Some other functions read ini data for themselves including the Animation.ini file which is read in directly by Animation.cpp since it is an optional feature and various sensor calibration data which is read by specific sensor instances.

Messages:

Commands from Lisp to Navigation (An array of 32 bit integer words) 8/18/05

This array is sent from Lisp to navigation through the RComm.dll.

Word	Value	
0	time tag	from WIN32 GetTimeTick() (milliseconds since machine was started) Note: this may be used by NavAndControl to determine whether the message has been changed, and may be used in return message from navigation to indicate which command it is referring to.
1	Message type	(provisions for sending different commands on same message. Make it zero)
2	Fwd Cmd	This is an integer value selecting the commanded mode which may be high level (e.g. explore) or low level forward/reverse motion commands (e.g. forward 1 meter). A later word will cover turn commands.
	0 None	No forward or high level command is specified in this message.
	1	Proceed straight forward or backwards measuring from current location (see words 3 -5 below for parameters)
	3	Modify an on-going "1" command. Measured distance will continue to be referenced to the original start point. (see words 3-5 below for parameters)
	5 Stop	(see words 3 for parameter) If robot is backing up, the direction "right" is as if the robot is looking backwards.

(examples of high level commands follow; none are currently implemented)

100 Contest go straight down hallway and through door

101 Contest go straight up hallway stopping 8 feet before start circle

105 Explore mode

3 Acceleration A positive number in mm/sec^2 (normally 100 to 1000) indicating the acceleration to be used for forward or reverse acceleration or deceleration or stopping. If set to zero, the micro will supply a default.

4 speed command A positive number in mm/sec indicating the maximum speed applied to commands 1 and 3 above. Robot will accelerate or decelerate to commanded speed at the acceleration in word 3.

5 distance command distance in mm. that the robot should proceed at the commanded speed. Robot will automatically come to a stop at the specified distance. To travel "forever", set the distance to a LARGE number, e.g. +/- 2000000000. Distance command is + for forward and – for backwards. Distance is measured from the last time a distance command was set using command = 1. If the distance is changed (command = 3) while a distance is already set, the robot will proceed to the new distance; the zero point will remain where it was originally set. If the distance is revised to a point behind where the robot has already reached, it will back up to the new point.

6	Turn Mode		Specifies type of turn. The turn commands will be added to the forward motion commands above. Hence, if the robot has zero forward speed, it will rotate in place. If moving, it will turn toward the left or right.
	0	None	No turn command is specified in this message
	1		Turn right or left at a specified acceleration, rate and distance in degrees. (see words 7-9 below for parameters)
	3		Modify an on-going "1" command. Measured distance will continue to be referenced to the original start point. (see words 7-9 below for parameters)
	5		Stop turning (see word 7 for parameter)
	10		Heading Select: Turn to a specified gyro based heading at the acceleration, rate and heading specified in words 7-9 below.
	11		HeadingHold: Controls to the existing gyro based heading +/- an optional value provided in word 9. Acceleration and rate should be specified as above.
7	Turn Accel		A positive number in deg/sec^2 * 10 (i.e. 10 deg/sec^2 is "100") Normally 100 to 1000 indicating the rotational acceleration or deceleration or stopping. If set to zero, the micro will provide a default.
8	Turn Rate		A positive number in deg/sec*10 indicating the maximum speed applied to commands 1 and 3 above. Robot will accelerate or decelerate to commanded turn rate at the acceleration in word 7.
9	Turn Dist		For command mode = 1 or 3, Distance in deg*10 that the robot should turn at the commanded speed. + is a clockwise turn looking down on the robot, - is CCW. This direction of turn applied when moving either forward or reverse. Robot will automatically come to a stop at the specified distance. To turn "forever", set the distance to a LARGE number, e.g. +/- 2000000000. Distance is measured from the last time a distance command was set using command = 1. If the distance is changed (command = 3) while a distance is already set, the robot will proceed to the new distance; the zero point will remain where it was originally set. If the distance is revised to a point behind where the robot has already reached, it will back up to the new point. For command mode = 10, A heading is specified in the range of 0 to 3600 (360 degrees). The robot will Turn the shortest direction to this new heading. For command mode = 11, The basic mode 11 causes the robot to track to the current heading. To do so, Word 9 should be equal to 0. A value in word 9 causes the robot to turn to a Heading to the left or right of the current heading. E.g. if word 9 = 100, the Robot will turn to the current heading plus 10 degrees. Word 9 will normally Be in the range of +/- 1800 (meaning +/- 180 degrees).

Note: Additional turning command modes may be added such as turning at a specified radius rather than a specified rate. This may change the definition of (at least) word 8 for those commands in the future.

Radio Control Servo commands:

10	RCservonum	Identifies an RC servo output for a command message. Will be an integer from 0 to 7 representing the microcontroller port to which the servo is attached. Numbers outside this range should have no effect.
11	RCparam0	There are three multi-purpose parameters for each RC servo command. This one specifies the rate of turn of the servo. A zero in this position means that no servo command is selected in this message (since a command to move at zero rate is meaningless anyway). Actual commands will be positive integer numbers indicating the command rate of turn in degrees per section times 10. E.g. 200 means to turn at 20 degrees/second.
12 13	RCparam1 RCparam2	There are two modes of RC servo operation. GoTo command tells the servo to go to a specified angle at the rate specified in RCparam0. The second mode is to scan back and forth between two specified angles at the specified rate. For GoTo mode, RCparam1 parameter specifies the target angle in degrees * 10. As a flag to indicate GoTo mode, RCparam2 is set to 9999. In Scan mode, RCparam1 is the rightmost (clockwise) limit of scanning and RCparam2 specifies the leftmost angle. Examples: RC servo 2, Go to 30 degrees clockwise at 100 degrees per second: RCservonum 2 RCparam0 1000 RCparam1 300 RCparam2 9999 RC servo 7, scan between +80 and –90 deg at 30 degrees per second: RCservonum 7 RCparam0 300 RCparam1 800 RCparam2 -900

Discrete I/O commands:

The microcontroller board provides 12 bits of discrete I/O which may be used as inputs or outputs. The status of each bit may be read in the Lisp data message at any time regardless of whether the bit is an input or an output.
If bits are configured as outputs, Lisp may choose to take control of some or all of the output bits. The next two command words implement that control.

14 15	DIOactive DIOcommand	Specifies which I/O bits are being controlled as outputs by Lisp. The 12 bits are designated by the lower order 12 bits of both the active and command words. I.e. binary 0000111111111111 (showing least significant 16 bits). Bits 0 to 3 specify the four output bits in the DIO0 port, bits 4-7 in DIO1 and 8-11 in DIO2. If bits are set to zero in this word, the ProcessCommands function will allow other users to control the status of these bits. If bits are set to one, then ProcessCommands will set the bit status as indicated by the corresponding bits in the DIOcommand word; where a zero sets the output to ground, and a one sets it to high.

These two words must be transmitted in each Lisp message to retain command. Otherwise, ProcessCommands will see zeros in the active word and stop Lisp command of the bits.

Example:

Assuming bits 0 to 7 are configured as outputs in the NavAndControl .ini files, Lisp takes control of bits 0 and 3 and sets them to 1 and 0 respectively.

DIOactive = 0000000000001001
DIOcommand = 0000000000000001

DIOcommand can be changed at any time. DIOactive should be maintained as long as Lisp wants to keep control of specific bits.

Command message from navigation to microcontroller revised 8/18/05

Note: Words are 16 bit (2 bytes) each

Word	Value	
0	start word	currently FFA5
1	time tag	Sequential number of command message. Unique to each new message (with new data). Same number should be used on repeated transmissions of same message.
2	msg type	Message type identifier.
3	spare	
4	FwdCmd	(this will be an integer value selecting the commanded mode of forward or reverse motion which may be high level (e.g. explore) or low level (e.g. forward 1 meter) This command is also used to select high level activities.
	0	None this message has no command for forward motion
	1	Proceed straight forward or backwards starting from a stopped condition. Starts distance reference at zero. (see words 3 and 4 for parameters)
	3	Modify the acceleration, velocity or distance of an on-going FwdCmd. This command will work even if the on-going command has completed and stopped. But the initial distance reference remains the same as the original command.
	5	Stop causes an on-going FwdCmd to halt immediately. Only the deceleration rate is specified (see word 4 for parameter)
5	spare	
6	accel command	sets maximum acceleration to be used during both acceleration and deceleration. Defaults to 300 mm/sec^2 if not set (zero value). Sign does not matter. Normally in the range of 50 to 500.

7	speed command		maximum speed applied to commands 1 and 3 above. speed is in millimeters/second . Robot will accelerate or decelerate to commanded speed at the specified accel rate. Sign does not matter. Normally in the range 10 to 400.
8/9	distance command		distance robot should proceed at the commanded speed. Robot will automatically come to a stop at the specified distance (32 bit long int). To move without having a specified distance, command a distance which is very far; e.g. +/- 200000000. Distance command is + for forward and – for backwards. Distance is measured from the last time a FwdCmd = 1 was sent . If the distance is changed with FwdCmd = 3 while a distance is already set, the robot will proceed to the new distance; the start point will remain where it was originally set.
10	Turn Cmd		This is an integer number selected a particular mode of steering. Turning may be done while the robot is moving forward or stopped. If stopped, the result is to pivot in place.
	0	None	no steering command is sent in this message
	1	Turn	This mode controls direction with respect to wheel encoder data. The distance to turn is always with respect to the starting point when the mode is engages. (see words 9 thru 11 for parameters)
	3	Modify turn	
	5	Stop turn	
	10	HdgSel	This mode controls direction with respect to the heading signal derived from the yaw rate gyro and compass. Heading can be specified from –360 to +360. Robot will turn the shortest direction from the current to new heading. (see words 9 thru 11 for parameters)
	13	Modify hdg (not implemented)	
	15	Stop hdg turn (not implemented)	
	20	Radius turn (not implemented)	
	23	Modify radius turn (not implemented)	
	25	stop radius turn (not implemented)	
11	spare		
12	Turn Accel		turn acceleration rate in deg/sec^2 *10 (normally about 300). Sign doesn't matter.
13	Turn rate		rate at which robot should turn or pivot in degrees per second *10. Normally in range of 50 to 600. Sign doesn't matter.
14/15	Turn distance		how far the robot should turn in degrees * 10. 30 degrees is "300". + is to the right, - to the left. (clockwise is to the right) If want to turn continuously, use a very large number; e.g. +/ 200000000

16	DG control	(provisions)
17	DG set	(provisions)
18/19	Xcoord set	(provisions)
20/21	Ycoord set	(provisions)
22	DIOcommand	
23	RC0param0	RC servo 0, rate select
24	RC0param1	
25	RC0param2	
26	RC1param0	RC servo 1, rate select
27	RC1param1	
28	RC1param2	
29	RC2param0	RC servo 2, rate select
30	RC2param1	
31	RC2param2	
32	RC3param0	RC servo 3, rate select
33	RC3param1	
34	RC3param2	
35	RC4param0	RC servo 4, rate select
36	RC4param1	
37	RC4param2	
38	RC5param0	RC servo 5, rate select
39	RC5param1	
40	RC5param2	
41	RC6param0	RC servo 6, rate select
43	RC6param1	
43	RC6param2	
44	MCDataLogCmd	microcontroller datalog message select

Notes on use of robot motion control commands.

Forward and reverse motion may be commanded and modified at most any time.

Motion may be commanded in either the forward (+) or reverse (-) directions. The Distance parameter specifies the direction of motion. The Acceleration and Velocity maximum parameters apply to both forward and reverse motion and acceleration and deceleration. The sign of these two parameters does not affect the results. Normally no sign is used.

A value of distance must be supplied, if only to specify direction of motion. Normally the robot will accelerate to the maximum specified velocity at the specified acceleration. It will travel toward the target distance at the maximum velocity and compute when to decelerate (at the specified acceleration) to stop at the specified distance.

If you want the robot to continue travelling for an indefinite distance (until you tell it to stop), choose a very large value for distance that the robot will never get to. A value of +/- 2000000000 (2000 KM) should do.

If the robot is traveling forward, you can send modified parameters using FwdCmd = 3. The robot will continue motion but will transition to the new parameters. Maximum velocity can be modified to be larger or smaller and the robot will speed up, or slow down. Acceleration can be modified and will affect any ongoing or future speed changes. If the robot cannot achieve the commanded velocity due to motor speed limitations, ramps, deep rugs etc, it will continue at the highest speed it can manage.

Distance can be changed at any time. Direction can even be reversed and the robot will slow to a stop and then proceed toward the new target. Even after the robot has completed a forward motion command and come to a stop, specifying a new distance with FwdCmd = 3 will cause the robot to restart and go to the new target.

If an explicit stop command has been sent (FwdCmd = 5), subsequent FwdCmd = 3 commands will have no effect. You must use a new FwdCmd = 1 message to start a new motion.