Multi-Vehicle Wireless Testbed User's Guide

Lars Cremean, Steve Waydo
lars@caltech.edu, waydo@cds.caltech.edu

April 15, 2003

Abstract

This document is intended to familiarize the reader with aspects of developing and running control algorithms for the Caltech Multi-Vehicle Wireless Testbed (MVWT).

Contents

1 Getting started

1.1 Basics

The MVWT vehicle consists of essentially a laptop (computing) sandwiched in Plexiglas and run on low friction ball casters, two ducted fans fixed to the vehicle (actuation), with wireless ethernet capability (sensing and communication). State information is received by the vehicle in a GPS-like fashion from an off-field vision system which uses four overhead cameras.

1.2 CVS web & M: drive

All relevant code to the project should exist in one of two places. The first is a CVS (Concurrent Versions System) repository at mojave.cs.caltech.edu called 'mvwt', which keeps track of all revisions of vehicle code, vision code, and PIC microcontroller code. You need an account on mojave to checkout and use this repository.

For viewing, the entire repository, with code and revision information, is available via the CVS Web on mojave.

The second is a user directory (mapped network drive) on the CDS cluster at /home/users/mvwt/, which can be mapped in Windows using the folder

`\\pcfiles.cds.caltech.edu\mvwt`

and username 'mvwt' (or your own CDS login). This is commonly called the 'M:' drive, and keeps documentation, saved data files and movies, a public_html directory, and other non-source code files.

Here are the names and descriptions of all of the top-level folders that are located on the MVWT mapped network drive.

- `documentation`: This is where you should find documentation about all of the different parts of the project.

- `dspace`: Thisis an old folder for code required to run closed loop control with dSPACE and radio control.
- `public_html`: This is where to put any MVWT data you wish to make publically accessible. HTML files in this directory are viewable at http://www.cds.caltech.edu/ mvwt/*. This folder is currently maintained by Lars Cremean (lars@cds.caltech.edu).
- `simulation`: Repository for simulation code and data.
- `user interface`: Directory for files related to an off-field command console.
- `users`: For your own files you want to keep handy.
- `vision`: Vision code resides on the vision computer and in the CVS repository. This folder is for saved data files, data processing tools, and other output files.

1.3 CVS Primer

If you want to create or change source code that runs on the vehicles, you need to be familiar with using CVS. The most important commands to know are

- `cvs checkout` (to get the repository for the first time),
- `cvs update` (to update your local copy to be the same as the repository),
- `cvs commit` (to commit your changes to the repository), and
- `cvs add` (to add new files to the repository).

The command for checking out the mvwt repository from mojave is

```
# cd /home/yourusernamehere/
# cvs -d:ext:yourusernamehere@mojave.cs.caltech.edu:/cvsroot co mvwt
```

http://www-pat.fnal.gov/muSim/cvs_prim.html has a nice quick reference to the basic commands.

Directories stick around in CVS, even after deletion (they are just marked as deleted). Using the `-d` and `-P` options will make sure you have only the current directory structure (and not the old ones):

```
# cvs update -dP
```

The `commit` and `update` commands are recursive from the directory in which they are executed, i.e.

```
# cd /home/lars/mvwt
# cvs update -dP
```

will update the entire local copy of the mvwt repository to be synchronized with the latest version on the server.

The reader is referred to the CVS Manual for more information on CVS and its full capabilities.

2 Network section

2.1 Local wireless network

We run a local wireless (802.11b) network separated from the CDS network via a NAT (Network Address Translation) box which serves to keep our local set of IP addresses invisible from outside the local network. The current set of machines on the local network and their static IP addresses are included in Table 1.

Name	Computer	IP Address	Notes
	Local Gateway	192.168.1.1	
vision	Vision Computer	192.168.1.100	Windows 2000
mvwt1	Laptop 1	192.168.1.101	On Kelly 1, QNX 6.1
mvwt2	Laptop 2	192.168.1.102	Unmounted, QNX 6.1
mvwt3	Laptop 3	192.168.1.103	On Kelly 3, QNX 6.1
mvwt4	Laptop 4	192.168.1.104	On Kelly 4, QNX 6.1
mvwt5	Laptop 5	192.168.1.105	On Kelly 5, QNX 6.2
mvwt6	Laptop 6	192.168.1.106	On Kelly 6, QNX 6.1
mvwt7	Laptop 7	192.168.1.107	Unmounted, QNX 6.1
mvwt8	Laptop 8	192.168.1.108	On Kelly 8, QNX 6.2
mvwt9	Laptop 9	192.168.1.109	Development, QNX 6.2
mvwt10	Laptop 10	192.168.1.110	
command	Command Computer	192.168.1.111	Desktop machine, QNX 6.1/6.2
bcast		192.168.1.255	Broadcast IP address

Table 1: The computers, network names and IP addresses of all of the machines that are connected to the MVWT local area network.

2.2 Addressing the vehicles

Each vehicle laptop is assigned the host name mvwt#, which conveniently is mapped to its IP address in each of the vehicles. Therefore, the following are all valid (and convenient) commands from a QNX prompt:

```
# telnet vision
# ping mvwt8
# rsh mvwt3 "ls -l /home/exec/localdata/"
# rcp myfile mvwt4:/tmp
```

3 Running the control software

This section should familiarize you with the software framework that runs on the vehicles.

3.1 Compiling

Nearly all of the vehicle code is C++ code written for the RHexLib programming suite and is therefore in the mvwt/rl directory (rl for RHexLib). All of the code compiles with Makefiles. After checking out a local copy of the CVS repository as explained in the CVS Primer section, you should be able to compile all of the code by:

```
# cd mvwt/rl
# make
```

to make the common modules and then executing 'make' in all of the directories that are in mvwt/rl/controller.

3.2 Running code on the vehicle

Rather than compiling code on every laptop that runs it, we debug, compile and test code on one machine and send only the executables and the .rc resource files they depend on, and run those remotely. The directory /home/exec/ exists on each vehicle laptop as a folder for executables and the resource files they use.

We use the remote copy command 'rcp' to send executables. General usage is 'rcp [source] [destination]'. For example, to send my_executable to mvwt4, and subsequently run it:

```
# rcp my_executable mvwt4:/home/exec/
# rsh mvwt4
mvwt4# cd /home/exec
mvwt4# ./my_executable
```

See 'man rcp' in Linux or 'use rcp' in QNX for more help on how to use this function.

3.3 CircLQRController

CircLQRController is a module in rl/controllers/circLQR/ which executes an LQR controller based on the error dynamics of the vehicle travelling around a circle of constant radius and angular velocity, as written in cylindrical coordinates. It accepts as parameters the radius, angular velocity and center of the circle about which to control the vehicle. These parameters can be changed at runtime.

CircLQRController uses a gain-scheduled array of optimal gain matrices, which are currently scheduled on thirty values of desired angular velocity ($\dot{\xi} = [0.1, 0.1, 3]$) and ten values of desired radius ($\rho = [0.1, 0.1, 1]$). This matrix array is automatically generated in RHexLib resource file format by the MATLAB script

`$CVSROOT/mvwt/users/lars/matlab/lqr_circles.m`

for fixed values of vehicle parameters and weighting matrices Q and R.

3.4 Using sendexec.sh

A script has been written that will run a series of 'rcp' commands for a list of vehicle numbers that are provided to the script as arguments. This script is at rl/controllers/.../sendexec.sh, and the files that are rcp'd are hard-coded into the script. Example usage to send the files indicated in the script to the executable directory of vehicles 3, 4 and 7:

```
# cd rl/controllers/MyController/
# ./sendexec.sh 3 4 7
```

3.5 Using master_control

The current preferred method of running any number of vehicles is the program `rl/controllers/.../master_control`. This program takes vehicle numbers as arguments in the same manner as `sendexec.sh` (see section 3.4). `master_control` attempts to start the vehicles corresponding to its input arguments using the script `mvwtstart`. The program to remotely execute on each vehicle is hard-coded into `mvwtstart`.

4 Writing custom control software

This section should be able to get you started with writing, compiling and running your own custom controllers to run on the vehicles.

4.1 RHexLib

RHexLib is a C++ software suite used to write vehicle control code. It is *module based*, where a number of modules can be run at specified rates in essentially real-time. A controller is an example of a module, as are Interfaces to other vehicles, a human user, the vision system, or the PIC electronics.

The reader is referred to (in particular, Chapter 5 of)

`M:\documentation\RHexLib\RHexManual.pdf`

for more details on programming in RHexLib.

4.2 Directory substructure

The mvwt repository includes the following structure:

```
/mvwt
  /rl          <--- RHexLib directory for executables, source and headers

    /exec      <--- commonly used executables code, e.g. usb\_test.cc
    /src       <--- commonly used RHexLib modules, e.g. VisionModule.cc
    /include   <--- header files for the modules above, e.g.
                    VisionModule.hh

    /controllers <--- library of vehicle controllers
      /circLQR   <--- controller that executes specific function
      /ctrlr2    <--- controller that executes a different function
      /ctrlr3    <--- these directories contain modules, headers and
                      executables
  /users
    /user1       <-- user1's repository directory
      /my_ctrlr1 <-- directory for user1's controller1 files while
                     developing the code and testing
      /ctrlr2fn  <-- directory names should be descriptive of the controller

    /user2...
```

RHexLib modules and main() functions (e.g. usb_test.cc) that are likely to be reused by multiple users should be placed under the 'rl' directories 'exec', 'src' and 'include'. Code with a specific developed function (e.g. LQR control about linearized polar error dynamics with circular reference trajectory) should be placed in a separate directory in the 'controllers' directory.

Custom RHexLib modules that are likely to be used by only one user should go under that user's directory. For development of new code/controllers, it is suggested that one use his/her own directory, and transfer the final (minimal) version to rl/controllers/ after sufficient development and testing. See 'controllers/MyController/' as an example.

4.3 Controller base classes

To speed development of controllers and to eliminate the need to write a new interface for every controller, two base classes have been created to standardize this system, Controller and ControllerShell. Controller handles the low-level controller tasks such as getting data from the VisionModule and sending forces to the fans. ControllerShell handles communication with MasterControl and the Real-Time Monitor and interfacing with Controller. Both of these classes contain virtual functions that must be defined in a derived class.

4.3.1 Controller

The Controller class contains 5 protected virtual void functions:

```
virtual void initController( void )
virtual void activateController( void )
virtual void updateController( void )
virtual void deactivateController( void )
virtual void uninitController( void )
```

which are called during the corresponding ModuleManager calls to Controller::init(), Controller::activate(), etc. (see the RHexLib documentation for details) Any or all of these can be left empty, but they must be declared by the derived class. The most important of these is the updateController function, which should at a minimum look at the vision data stored in VEHICLE_STATE veh (see VisionModule.hh for the definition

of this struct) and compute fan forces fleft and fright, which the parent class takes care of sending to the fans. Ideally, this function should also update the (port and starboard) nominal forces Fp_nom and Fs_nom and error forces Fp_e and Fs_e, which are broadcast to MasterControl and logged by the ControllerDataLogger.

Additionally the Controller class contains 1 public virtual void function:

```
virtual void updateParams( void )
```

This function gives the user the ability to define controller parameters which can be reconfigured through the MasterControl interface. The Controller class has a protected parameter array (double params[9]) which is updated by ControllerShell when new parameters are received from MasterControl. updateParams is then called by ControllerShell when the "commit" message is received from MasterControl. This function should then take the data from the params array and put it into the corresponding local variables. The names and units of these parameters should be stored in "controller_params.rc"

4.3.2 ControllerShell

The ControllerShell class also contains 5 protected virtual void functions:

```
virtual void initShell( void )
virtual void activateShell( void )
virtual void updateShell( void )
virtual void deactivateShell( void )
virtual void uninitShell( void )
```

which are called during the corresponding ModuleManager calls to ControllerShell::init(), etc as with the Controller class. These functions would generally be used to set up vehicle-to-vehicle communications, and for a single-vehicle control algorithm may be left entirely blank.

The ControllerShell class contains 1 more protected virtual function:

```
virtual void setDebug( void )
```

This function gives the user the ability to define debug parameters which will be sent to the MasterControl interface. The ControllerShell class has a protected debug array (double debug[5]) which is sent to MasterControl when the debug level is 2 or higher. setDebug() should place the data from whatever local variables the user wishes to be able to see in this array. The names and units of these variables should be stored in "controller_params.rc"

The ControllerShell class also incorporates the ability to load different configuration files while a controller is running. The names of any files to be loaded should be stored in "controller_params.rc"

4.4 Creating your own Controller

The MyController and MyShell classes (located in controllers/MyController) are a basic template for implementing your own controllers. This controller is derived from the Controller base class and contains most of the basic functionality a user may desire for controller implementation. In particular, this template contains built-in interfaces to the gyroscope, vision system, command computer and other vehicles.

To create your own controller, first copy the "makeNewController.sh" script from mvwt/controllers/MyController/ to your user directory, i.e. `/home/<username>/mvwt/users/<username>/`. When executed *in your user directory*, this script creates a local copy of the "MyController" files in a subdirectory and modifies all files to a user-defined controller name. The usage syntax is given by:

```
./makeNewController.sh <Name>
```

The two main resulting files of interest are `./<Name>/<Name>Controller.cc` and `./<Name>/<Name>Controller.hh`. The updateController() method of the first file is where you will implement your controller design. The variables and methods available to you are described in comments contained in the second file.

4.4.1 Modifying the MyController and MyShell classes

Simply modify the functions described above to implement your controllers functionality. Be sure to update `<Name>Controller/controller_params.rc` to reflect your changes so MasterControl knows how to handle things.

4.5 The main() function

The my_shell.cc file (which you should rename) contains the main() function. Required arguments to your derived Controller class are int args[] and int num_args, which must be passed in turn to the parent Controller. args[0] is 1 if the simulator is being used and 0 if the real vision system is being used. MyController reads this from the "vehicle_params.rc" file as "simQ," which is a convention that should be followed by new controllers. args[1] should be 1 if your controller relies on having a fixed time step (which we will define in a moment) and 0 if you want your controller to update as soon as new vision data are received. No other arguments are currently implemented. If you want to pass arguments to your derived Controller, *do not* use this data array, as more entries may be used in the future and we want to preserve backwards-compatability. Instead simply pass more arguments to your constructor. If more entries are added in the future, the Controller class will be designed to operate as it did before if they are omitted.

Only one more part of my_shell.cc should need to be modified: scheduling of the controller. The controller is added to the ModuleManager list and scheduled by the line:

```
MMAddModule( &ctrl, 16, 0, 30 )
```

The arguments to MMAddModule are (modulepointer, period, offset, order). If you are implementing a fixed time step controller, period should be set equal to the time step. If the controller should update as soon as new vision data are received, period should be set to 1 ms. The Controller update function will then check ever ms to see if new vision data have been received (about every 16 ms) and if so, call updateController() and send forces to the fans. Note that the SerialWriter module is set to run at 16ms period (about 60Hz); this is the frequency at which commands are actually sent to the fans. For a fixed time step controller the SerialWriter should have the same period (or an integer multiple of the period) as the Controller and a higher order so it will be sure to execute as soon as the Controller::update is complete.

4.5.1 Making your executables

The next step is to modify the Makefile to make your controller executable. Each set of code has a Makefile or set of Makefiles that store the options for compiling RHexLib code. These Makefiles compile the RHexLib modules indicated in the 'SOURCES =' line of the Makefile and the executables that use the modules, indicated on the 'EXEC =' line of the Makefile. To make your own controller, simply replace 'my_shell' with your main() function .cc filename (without the extension) on the 'EXEC =' line, and replace 'MyController.cc' and 'MyShell.cc' with the appropriate names on the 'SOURCES =' lines. This may already have been done by makeNewController.sh.

The Makefile, and sendexec.sh and master_control scripts should already be modified appropriately by makeNewController.sh. When run from their directory, these should already know which files to send to the vehicles and what executable to start.

4.5.2 Running your Controller

To run your controller, you need to send it to a vehicle and run master_control. To send to a vehicle, modify "sendexec.sh" and replace the executable and parameter file names with those of your controller. Then type:

```
# ./sendexec.sh veh#
```

where veh# is the number of the vehicle on which you are going to run. Now modify "mvwtstart" to contain your executable name.

The Makefile will also have copied the master_control executable from mvwt/rl/exec/ to your controller directory (if that failed, you need to build the master_control executable and move it there yourself). To run, type:

```
# ./master_control veh#
```

and have fun!

4.6 Resource files

should talk about MM calls to read values into controller from resource file; vehicle_params.rc; adjacency matrix; parameter arrays.

5 Running the vehicle hardware

This section should familiarize you with running the vehicle hardware

5.1 Preparing to run

To run a vehicle for which you want to test a controller, follow the following checklist:

- Copy your executable to the vehicle /home/exec directory, and make sure it runs properly.
- Run the vision system code on the vision computer at

  ```
  C:\vision\vision.exe
  ```

- Turn on the switches that connect the main battery packs to the motor controllers. You should hear the motor controllers beep once to indicate that they are initialized properly.
- Place the hat corresponding to the vehicle number on the vehicle. You should now be ready to run.

5.2 Battery procedures

This section gives a brief description of the batteries being used on the vehicles and how to charge them.

5.2.1 General Information

Each fan on the vehicle is powered by a pack of ten (10) sub-C size NiMH rechargeable batteries (Panasonic or Sanyo). Two 10-packs are taped together and typically recharged at the same time.

5.2.2 Charging Procedures

When the 10-pack battery voltage gets below 12.5 volts or so, it is time to recharge the pack. To do so:

- Turn on the 12V power supply
- Plug in the battery packs.
- Write down today's date (on sheet near charger).
- Press the start/stop button.
- The battery charger will display an "F" (for Finished) when the batteries are fully charged (about 1.5 hours).

5.2.3 Charger settings

We're using Robbe Power Peak Infinity 2 battery chargers. The following settings should be pre-set on the chargers:

- D.P. SENSITIVE (delta peak detection - see Robbe Power Peak Infinity 2 Operating Instructions manual)
- SET CHAR: (charging rate): 2.0 amps
- SET DISC: (discharging rate): 5.0 amps
- S.M.CUR: set at 0.0 amps. The eliminates "trickle" or "maintenance" charging (recommended for NiMH batteries).
- S.DC.CUT: 10V times the number of packs. For two packs, this should be 20V.
- S.PAUSE: 10 MIN. This allows the packs to cool between charge/discharge cycles.
- Mode: Set on DISC=¿CHAR mode and set cycles to "1 TIME". Set on this mode, the charger first discharges the battery packs to the S.DC.CUT value (10V per pack in our case), then fully charges the packs. First discharging to 10V (per pack) increases battery performance (according to Panasonic - see www.panasonic.com/industrial/battery/oem/chem/nicmet/index.html).

5.2.4 Notes

- DO NOT CHARGE OR DISCHARGE A WARM BATTERY PACK. Heat damages batteries. Before charging or discharging a battery pack, make sure it is cool to the touch (this includes discharging the pack by running the fans with it - make sure the pack is cool before you use it on the vehicles).
- DO NOT TRICKLE ("MAINTENANCE") CHARGE. As stated above, this is not recommended for NiMH batteries (however, it IS recommended for NiCd batteries).
- Dead Batteries: Just after fully charging, the battery pack should be between 14 and 14.5 volts (this voltage quickly drops to 13.5 volts during the first 24 hours after charging). If the pack does not charge to 14.0 volts, one of the cells is bad and the pack should be disposed of. Recycle if possible.
- Connectors: The connectors are Astro zero-loss connectors, part number 525, from Astro Flight, Inc., 310-821-6242.
- Deep Discharged Batteries: If a battery pack is below 8 volts, change the charge rate (SET CHAR) to .1 amps. This "balances" the cells, resulting in better battery performance.
- New Battery Packs: Charge at .1 amps, then perform 6 discharge/charge cycles (at the normal settings, i.e., 5.0 amp discharge 2.0 amp charge) to "balance" the cells.

5.3 Laptop troubleshooting

If a laptop does not respond to a 'ping' command, it probably needs to be restarted. It may need to be restarted twice if it is off because the battery died.

5.4 Motor controller troubleshooting

Motor controllers may not initialize properly every time. If one of them beeps continuously, it is not initialized properly and you should follow this procedure:

- `# killupic`
- `# startupic`

- Run `usb_test` up to signal greater than 10.
- Turn off the switches.
- Turn on the switches.

If the motor controllers do not respond at all and all connections are properly made, try the following:

- Turn off the switches.
- Unplug the motor controllers.
- `# killupic`
- Plug the motor controllers back in.
- Turn on the switches.
- `# startupic`

Alternatively,

- Turn off the switches.
- Unplug the motor controllers.
- `# killupic`
- `# startupic`
- Plug the motor controllers back in.
- Turn on the switches.

If this doesn't work, find out what does and write it here.

6 Datalogging

This section should show you how to customize, create and retrieve data files from the vehicles.

6.1 Datalogging basics

Data files are produced by the DataLogger module or a class derived from it, and should write to a date/time/vehicle-stamped file in the `/home/exec/localdata/` directory. The filename format is

`YYYY_MM_DD_HH_MM_SS_mvwt#.data`

Data files worth saving should be copied to `M:\vision\Interesting Logs\`. One way to do this is

```
scp file.data user@cds.caltech.edu:public_html/more_descriptive_name.data
```

and then save it from a web browser to a computer with the M: drive mapped.

6.2 Data column formats

To be consistent with other data files and the tools used to process them, data files should be saved with the following column format:

Column 1: time (seconds)
Column 2: vehicle x position (meters)
Column 3: vehicle y position (meters)
Column 4: vehicle orientation (θ, radians)
Column 5: $\dot{x}$, (m/s)
Column 6: $\dot{y}$, (m/s)
Column 7: $\dot{\theta}$, (rad/s)
Column 8: reference x position (x_{ref}, meters)
Column 9: reference y position (y_{ref}, meters)
Column 10: reference orientation (θ_{ref}, radians)
Column 11: reference $\dot{x}_{ref}$, (m/s)
Column 12: reference $\dot{y}_{ref}$, (m/s)
Column 13: reference $\dot{\theta}_{ref}$, (rad/s)
Column 14: commanded port (left) force, (F_p, Newtons)
Column 15: commanded starboard (right) force, (F_s, Newtons)
Column 16: reference/nominal port (left) force, ($F_{p,ref}$, Newtons)
Column 17: reference/nominal starboard (right) force, ($F_{s,ref}$, Newtons)

Additional columns of data can be added at the user's behest. If any of the above information is unavailable (e.g. nominal forces), fill the corresponding columns with zeros.

6.3 Data post-processing

Two tools are available for processing data files saved to the vehicle in the format specified above. These are `procdata.m` and `animgui.m` and are available at `M:\vision\Interesting Logs\`.

7 Simulation

7.1 Simulator details

The simulator runs a simulation of up to MAX_AGENTS (currently 8) vehicles, and broadcasts the state information of these vehicles in exactly the same manner as the vision system, but on port 2005 rather than 2001.

7.2 Running the simulator

7.3 Additional notes

We have found that if we run the simulator on a laptop using a wireless connection to send state information, then a very large percentage (25%-75%) of simulated vision packets were not successfully transmitted over the network. Running the simulator from a machine that is directly plugged into the wireless hub (as the vision system is) resulted in successful transmission of virtually all packets.

A Installing QNX 6.1/6.2 and MVWT Software

This section details the step-by-step instructions for installing QNX 6.1 real-time operating system (RTOS) on an MVWT laptop and configuring the laptop with RHexLib and other MVWT-related software. Here are the instructions (instructions specific to QNX 6.1 or 6.2 are marked **[6.1!]** or **[6.2!]**):

1. **[6.1!]** Install QNX 6.1 on laptop either by using the CD in the lab (labeled QNX RTP 6.1.0), by burning your own CD from the ISO available in M:\installation, or by upgrading using the Package Manager of QNX 6.0. You can check your version of QNX by typing "uname -a" in a terminal.

   [6.2!] Install QNX 6.2 on laptop by using the CD in the lab. You can also burn your own CD from the ISO available in M:\installation.

   [6.2!] Note: You will probably need to delete and replace the current QNX partition. If you can figure out how to do an install to a partition in addition to a QNX 6.1-installed partition, please send an email to mvwt@cs.caltech.edu!

2. **[6.2!]** Install the Momentics development suite after the first boot-up by going to the Installer (CD Repository), selecting "QNX Momentics NC for Neutrino (x86)", and selecting "Install".

3. Plug one of the wireless ethernet cards into the PCMCIA slot of the laptop and mount it by invoking the command:

   ```
   # mount -Tio-net -o "network=caltechmvwt" /lib/dll/devn-orinoco.so
   ```

4. Enable the wireless card by doing the following: Select the "Network" button. You should see an "en1" device. Deselect "Enable Device" for en1, change "Connection" to "DHCP", reselect "Enable Device" and click "Apply". Test the wireless card by pinging something in a terminal.

5. **[6.1!]** Go to Installer (QNX WWW Repository), select "Show: All Packages" and "Target: x86", and scroll down and check the following boxes if an 'x' does not already appear, and "Install" after all are checked:

   ```
   * QNX Realtime Platform > QNX (Patch A) for x86
   * QNX Realtime Platform > Software Development >
     - C/C++ Toolset for x86 targeting x86
     - Development (libs and headers) (Patch A) targeting x86
     - Development (libs and headers) targeting x86
   * Text Editing and Processing > Text Editors >
     - Vim for x86
   * Software Development > Programming Languages and Parsers >
     - Perl for x86
     - Libtool for x86
   * Software Development > Programming Languages and Parsers >
     Parsing and Lexical Analysis >
     - GNU m4 for x86
   * Software Development > Build Tools >
     - Automake
     - Autoconf
   ```

   [6.2!] Run the *old* package installer by typing "pkg-installer" in a terminal. Select (QNX WWW Repository), select "Show: All Packages" and "Target: x86", and scroll down and check the following boxes if an 'x' does not already appear, and "Install" after all are checked:

   ```
   * Text Editing and Processing > Text Editors >
     - Vim for x86
   * Software Development > Programming Languages and Parsers >
     - Perl for x86
     - Libtool for x86
   * Software Development > Programming Languages and Parsers >
     Parsing and Lexical Analysis >
     - GNU m4 for x86
   ```

```
* Software Development > Build Tools >
  - Automake
  - Autoconf
```

6. Get the mvwt-kit installer (RHexLib/ssh/cvs) by browsing the CVS 'mvwt' repository at

   `http://mojave.cs.caltech.edu/cgi-bin/viewcvs.cgi/mvwt/qnx/Attic/mvwt-kit.tar.gz`

   Right-click revision number 1.3 and select "Save Target As...". Save the file to /tmp/. Run the following:

```
# cd /tmp
# gzip -dc mvwt-kit.tar.gz | tar xvf -
# cd mvwt-kit
# ./kitinstall.sh
```

7. Log out of QNX Photon (Launch ¿ Shutdown ¿ End Photon session) and log back in as root. This sets the environment variables expected by the make code.

8. Install RHexLib. Note: you need an account on mojave to do this, and you should be prompted for a password at each 'cvs' command.

```
# export NAME=lars (in place of "lars", enter your own Mojave account user name)
# cd /home/rhex
# cvs -d:ext:$NAME@mojave.cs.caltech.edu:/cvsroot co RHexLib
# cd RHexLib
# make depend; make
```

9. Test the success of the installation:

```
# mkdir /home/$NAME
# cd /home/$NAME
# cvs -d:ext:$NAME@mojave.cs.caltech.edu:/cvsroot co mvwt
```

10. Reconfigure the wireless card to use static IP addressing as follows.

```
# cp /home/$NAME/mvwt/qnx/hosts /etc/
# cp /home/$NAME/mvwt/qnx/.rhosts /root/
```

    - Select the "Network" button on the desktop menu. In the "en1" section of the "Devices" tab, select "Connection: Manual", "IP: 192.168.1.1xx", where "xx" is replaced by the two digit laptop number (e.g. 192.168.1.109 for laptop 9), "Netmask: 255.255.255.0"
    - In the "Network" tab, set

      ```
      "Host Name:       mvwt#" (e.g. mvwt5 or mvwt10)
      "Domain Name:     cds.caltech.edu"
      "Default Gateway: 192.168.1.1"
      and add "131.215.42.28" as a name server.
      ```

    - Click "Apply" to save changes.

11. Reboot to reset IP configuration ("Launch ¿ Shutdown... ¿ Shut down and reboot").

12. Compile and install libcomm (communications library):

```
# export NAME=lars (in place of "lars", enter your own Mojave account user name)
# cd /home/$NAME/mvwt/comms/libcomm
# ./autogen.sh
# make
# make install  (as root)
```

13. Compile and install rhexcomm (communications RHexLib module):

```
# cd /home/$NAME/mvwt/comms/rhexcomm
# ./autogen.sh
# make
# make install   (as root)
```

14. Compile the existing vehicle control code:

```
# cd /home/$NAME/mvwt/rl; make
```

15. Copy over /root/.profile from another machine, and edit the prompt name (PS1).

    If you are able to reach this last step, and got no errors along the way, then your installation and setup was successful!

B Installing OCP Build 2.1 Binaries on QNX 6.2

This section details the step-by-step instructions for installing the Open Control Platform (OCP) build 2.1 on a machine running QNX 6.2 real-time operating system (RTOS).

1. Set up an OCP install directory:

```
# mkdir /usr/local/SEC
# mkdir /usr/local/SEC/OCP_B2.1
# export OCP_ROOT=/usr/local/SEC/OCP_B2.1
```

2. Download the following .tgz file to $OCP_ROOT:

```
http://www.cds.caltech.edu/~mvwt/OCP/OFPqnx6162binaries.tgz
```

3. Unzip the OCP binaries:

```
# cd $OCP_ROOT
# gzip -dc OFPqnx6162binaries.tgz | tar -x
```

4. Fix a directory path issue:

```
# mv $OCP_ROOT/OCP/OFP/* $OCP_ROOT/OCP/
# rmdir $OCP_ROOT/OCP/OFP/
```

5. Set up additional environment variables:

```
# export TARGET=QNX62
# . $OCP_ROOT/setup_sec_qnx
```

6. Make the environment variables automatic on reboot, by making sure the following lines are in /root/.profile:

```
export OCP_ROOT=/usr/local/SEC/OCP_B2.1
. /usr/local/SEC/OCP_B2.1/setup_sec_qnx
export TARGET=QNX62
```

7. Build the core OCP code:

   ```
   # cd $OCP_ROOT
   # make
   ```

   Note: 'make' may produce errors if TARGET=QNX62 is not set.

8. (Optional) Make modification to Makefile to avoid infinite loop bug:

   ```
   Edit $OCP_ROOT/OCP/Examples/ControlsAPI/APIFeatures/Process1/Makefile,
   remove first item of "DIRS =" line ("../../ \")
   ```

9. Make some of the OCP examples:

   ```
   # cd $OCP_ROOT/OCP/Examples/ControlsAPI/Hello_World; make
   # cd $OCP_ROOT/OCP/Examples/Infrastructure/EventCorrelation; make
   ```

10. (Optional) Note: This example produces ERROR! on make:

    ```
    # cd $OCP_ROOT/OCP/Examples/ControlsAPI/APIFeatures/ModeManager; make
    ```

B.1 Short Wish List for Next Build of OCP

1. Fix error produced by step 10.
2. Replace corrupted images in documentation.
3. Restructure documentation to be indexed and less monolithic.
4. Eliminate step 4.
5. Eliminate step 8.

C Vision System

Details about the vision system can be found in the old documentation by Jason Meltzer.

D Delay Estimation

This section contains information about estimation of the closed-loop system delay and its components.

E Sensing and Kalman Filtering

Tim is going to write some fabulous wisdom here!