RoboFlag for MVWT Software Users' Guide

Robert Christy

January 27, 2008

Contents

1 Overview

This document is intended to give the reader a thorough understanding of what infrastructure has been developed to allow Cornell-developed *RoboFlag* games to be played on the Caltech-developed *MVWT* testbed. It should be noted that in this document, *MVWT* refers specifically to "*MVWT I*," the testbed located in room 12 of the Steele building and is not necessarily relevant to the *MVWT II* testbed.

2 Setup

2.1 Requirements

In order to run a *RoboFlag* game on the *MVWT* testbed, you need the following things:

- 1 *MVWT* testbed
- 2[1] or more of the following types of robots: "Steelebot" kinematic robots, "Bat" hovercraft, or "Kelly" fan-driven vehicles.
- 1 or more computers running *Windows® 2000* or *Windows® XP*. These computers are necessary to run the *RoboFlag* software. Since this software can be rather computationally intensive, multiple computers may be necessary for larger numbers of robots.
- 1 computer running Linux. The software intended for this computer has only been tested under Redhat 9, but should work for any installation using the 2.4 kernel.
- A copy of the *RoboFlag* software. The testbed has been tested with the 2.1 beta version of the *RoboFlag* code. To test compatibility between the a version of the *RoboFlag* software and the *MVWT* software, see appendix A.

2.2 Setting up the *RoboFlag* software

Clearly, a working version of the *RoboFlag* software must be acquired and setup in order to run an *RoboFlag* game. The instructions for setup differ per *RoboFlag* version, however, the basic instructions should be present on the *RoboFlag* website: `http://roboflag.mae.cornell.edu/RoboFlag.html`. The only *MVWT* specific directions are below, otherwise the *RoboFlag* software should be configured normally.

1. In ConnectionParameters.txt, set the parameters *REAL_WORLD_HOST* to the hostname or IP address of the computer where the "Real world server," *MVServer*, will be running (ie. the Linux machine mentioned in section 2.1). Also, be sure to set the *REAL_WORLD_PORT* parameter to the port on which *MVServer* will listen for the Arbiter. The default port is 4545.
2. In RoboflagConstants.txt, make sure that the constant *UseWireless* is turned off (ie. has a value of zero). Although setting this value should not have any adverse effects, the RF-based wireless communication system used by the Cornell *RoboFlag* testbed is not used by the Caltech *MVWT* testbed, and therefore should be turned off.

2.3 Setting up the *MVWT* software

To set up the *MVWT*-specific software, first copies of the *MVWT* CVS module must be checked out of the central CDS CVS repository located at `mojave.cs.caltech.edu:/cvsroot/`. To get access to this repository, see whomever it

[1]The game can be played with only one vehicle, but it will, most likely, not be a very interesting game

is that is in charge of it[2]. Once a local copy of the module has been checked out, the source code relevant to the remainder of this document can be found in the directory `mvwt/users/roboflag/`. Note that, in fact, one copy of the CVS module will need to be checked out on each computer where code will be compiled. At the moment, almost all of the code can be compiled natively on a Linux machine[3]. The only exceptions being the Bat controller, which can be compiled on a Linux machine with the appropriate cross-compiler, and the Kelly controller which requires a QNX 6.1 machine to compile. See section 2.4 for more on compiling specific controllers.

The first thing that needs to be done with a fresh checkout of the *MVWT* module is to build the *MVServer*, as the *RoboFlag* Arbiter cannot interface with the *MVWT* testbed without it. To do this, change to the directory `mvwt-/users/roboflag/MVServer` and run the command `make`. This will automatically compile *MVServer* and link it into an executable. Additionally, several utilities will be built. These include a utility to interface with the robots once they are running *RoboFlag* controllers (see appendex B for more information on the `sendcommand` utility) as well as several debugging tools. For more information on these utilities, read the `README` file located in that directory or run the command in question with the `-h` command-line option.

Next, *MVServer*'s configuration file `vehicle.conf` must be modified to indicate which vehicles will be used in the game, which (vision system) hats they are wearing, and which *RoboFlag* team they should be assigned to. The file is whitespace independent and comments begin with a #. In the file, each non-empty line represents a single vehicle with two integers and a hostname or IP address. The first of the two integers is the hat number (currently the hat numbers range 1-16), which is the number by which the vision system will identify that robot. The second integer is the number of the *RoboFlag* team to which that robot will be assigned. Currently, *RoboFlag* supports two teams of players, plus a third, "obstacle" team. The *RoboFlag* "agent id," which is used by *RoboFlag* entities, is simply determined by the order in which vehicles are specified in the file. See figure 1 for an example.

2.4 Setting up the controllers

In *RoboFlag* for *MVWT*, the programs that run on the individual robots are known as controllers. In order to play a *RoboFlag* game on the *MVWT* testbed, the controller for each type of robot to be used must be compiled and a copy of the resulting executable must be copied to each robot of that type that will be used. The code for all of the *RoboFlag* controllers can be found in subdirectories of the `mvwt/users/roboflag/controllers` directory.

Since each type has its own, intricate software infrastructure, each robot type has its own setup procedure. The setup procedure for each robot type is

[2]I have no idea who this person is. I gather that none of the grad students do either. In fact, I do not even have my own CVS user; I use Steve Waydo's

[3]See 2.1. It is assumed that this machine is of the x86 architecture

```
# vehicle.conf  -  Vehicle/team configuration for MVServer

# Team 1 (first robot team)
#   Vehicle     Team        Hostname/IP Address

    2           1           steelebot7      # agent id 1
    15          1           steelebot5      # agent id 2
    1           1           bat3            # agent id 3

# Team 2
#   Vehicle     Team        Hostname/IP Address

    13          2           steelebot2      # agent id 1
    14          2           bat1            # agent id 2
    16          2           steelebot4      # agent id 3

# Obstacles (team 3)
#   Vehicle     Team        Hostname/IP Address
    3           3           mvwt3           # obstacles don't
    6           3           mvwt6           # have agent ids
```

Figure 1: This is an example `vehicle.conf` file. The comments on the far right of each line indicate which agent id each vehicle gets as dictated by the ordering of the file.

outlined below. Typically, the setup procedure will need to be repeated for each robot, however, the controller will only need to be compiled once.

2.4.1 Steelebot controller

To compile the Steelebot controller and the necessary setup utilities, change to the directory `mvwt/users/roboflag/controller/Steelebot/` and type `make`. This step need only be done once. This step should take place on an x86-architecture computer running Linux with gcc.

The following steps should be performed for each Steelebot to be used in the *RoboFlag* game. There are several notes on the procedure located at the end of this section, be sure to read over them before you begin, they may save you some time and trouble.

1. Turn on the Steelebot to be set up and, when it has booted, login using either telnet or ssh. If, in the home directory, the executable `RFController` and the data file `anglesteps.data` already exist, then no further setup is necessary.

2. If `RFController` is not already present, copy it from the build machine to the Steelebot using `scp`. This executable is the *RoboFlag* controller.

It can be found among the results of the `make` performed earlier in this section. If, at this point, both `RFController` and `anglesteps.data` are present on the robot, no further setup is necessary.

3. If the executable `stepmap` is not already present, copy it from the build machine to the Steelebot using `scp`. This program is a utility that will generate the `anglesteps.data` file. It can be found in the results of the `make` as well.

4. Run the battery monitor program found on the Steelebot by typing `run battread`. The number on the far left should indicate the approximate battery power remaining. Make sure that there is enough battery power for another 20 minutes of operating time (ie. the battery power should be > 2150 or so). If there is not, power down (by running `run /sbin/halt`, waiting for one minute, then switching it off), replace the battery and boot again.

5. Now generate the `anglesteps.data` with the command `run stepmap`. This will take a while. The `stepmap` utility generates a look-up table that assists the *RoboFlag* controller in making precise, in-place rotations. To do this, however, the utility will cause the Steelebot to rotate, in sequence, 360 rotations of increasing angle. Be patient. When this step completes, a new file `anglesteps.data` should have been created and the Steelebot setup should be complete.

Notes:

- Each Steelebot is different. When choosing which Steelebots to use in your game, be sure to get an accurate listing of which are currently in a working state (this list is everchanging). Also make sure that you have the correct username and password for each robot. While the usernames and passwords are largely consistent among the robots, they are not entirely so.

- The software present on the Steelebot is not necessarily consistent either. Specifically, certain versions of `battread` operate differently than others. In some cases, the battery numbers will scroll by rather quickly and the program will terminate after 100 readings and, in others, the readings will progress slowly and `battread` will take an infinite number of them. In this latter case, feel free to use CTRL-C to terminate the program whenever you please.

- If you find yourself feeling lazy and not wanting to wait 20 minutes for a `stepmap` to complete, it is generally safe to copy an `anglesteps.data` file from another Steelebot. While, strictly speaking, `stepmap` generates a rotation profile accurate for each specific robot, most of the motor controllers across the Steelebots are consistent enough to get by with the same one.

- You must prefix most of your commands with `run` as demonstrated above. This is because many commands require root privileges to run, none of the programs have the setuid bit set and running them as root would be a faux pas. Therefore, the `run` command, which does have the setuid bit set (and is owned by root), exists to run whatever command it is passed as root.

2.4.2 Bat controller

To compile the Bat (hovercraft) controller, the ARM Linux cross compiler must first be installed. This must only be done once.

1. Check to make sure the cross compiler, libraries, header files and binary utilities have not already been installed. The standard target directory for their installation (on a Red Hat machine, anyway) is `/opt/Embedix/-tools/`. If they have, you may skip ahead to step 4.

2. The necessary software can be downloaded from `http://www.zaurus.com-/dev/tools/downloads/tools/` as RPM files. Listed here the are the versions of the necessary packages that are current as of the writing of this document:

   - The cross compiler, `gcc-cross-sa1100-2.95.2-0.rpm`
   - The standard C library, `glibc-arm-2.2.2-0.rpm`
   - Linux header files, `linux-headers-arm-sa1100-2.4.6-3.rpm`
   - Binary utilities, `binutils-cross-arm-2.11.2-0.rpm`

   Download and install these packages using `rpm` (see the `rpm` documentation for usage).

3. Modify your *PATH* environment variable to include the new cross-compiler binary directory `/opt/Embedix/tools/bin/`. This is necessary in order for the `Makefile` to work. The precise method for doing this varies depending on which shell you use.

   For users of `sh`, `bash`, or another `sh` variant : Add the line
   `export PATH=$PATH:/opt/Embedix/tools/bin/`
   to the `.profile` or `.bash_profile` file in your home directory. Then type that very same line at the command prompt and execute it (since the `.*profile` file won't be reread until the next time you login).

   For users of `csh`, `tcsh`, or another `csh` variant : Add the line
   `set path=($path /opt/Embedix/tools/bin/)`
   to the `.login` file in your home directory. Also type this same line into the command prompt and execute it (since the `.login` file won't be reread until the next time you login).

4. Change to the directory `mvwt/users/roboflag/controllers/Bat/` and run `make`. The Bat *RoboFlag* controller as well as associated utilities and debugging tools should build.

After building the controller. Copy the following files to each Bat to be used in the *RoboFlag* game:

- The *RoboFlag* controller executable, `RFController`
- The controller gains file, `gains.txt`
- The vehicle parameters file, `parameters.txt`

In order to perform this copy, make use of the ftp server that each Bat runs. See the documentation for the ftp client, `ftp`, for help copying these files. While it is not important where, specifically, you put these files, only a subset of the Zaurus' filesystem is writeable (much of the filesystem is actually on the Zaurus' ROM, thus making it read-only). One potential place to put the files is `/tmp`. Also, all subdirectories of `/home` are writeable. Finally, before the controller can be run, it must be made executable (as ftp often drops this file attribute). To accomplish this, log in to the Zaurus, change to the directory to where the files have been moved and type `chmod 755 RFController`. The Bat should now be ready to play *RoboFlag*.

2.4.3 Kelly controller

This controller requires special treatment as it is based upon the *RHexLib* modules developed for the *MVWT* testbed. As such, the code can only be compiled on a QNX 6.1 system with an *MVWT* system installation. See the *MVWT* documentation on how to perform this installation, if it is necessary. Since the *RoboFlag* controller is based upon the modules found in `libmvwt.a`, the *RHexLib* `Makefile`s are used to build it. Essentially, building the controller consists of changing to the directory `mvwt/users/roboflag/controllers/Kelly` and running `make`. For more detailed information, see the *MVWT* documentation.

The controller executable, `RFShell`, may be copied to the vehicle with `rcp` but not `scp` as the Kellys do not run ssh servers. Like the Bats, it is not of particular importance where the controller executable is placed, provided it is executable from that location. Also, the controller will require one parameters file, `vehicle_params.rc`, which should already be present on the vehicle. Simply make sure that a copy of (or symbolic link to) the file is in the same directory as the controller executable.

3 Running the game

In large part, running a *RoboFlag* game is simply a matter of following the instructions as laid out on the *RoboFlag* website. There are, however, several important steps that must be performed before the *RoboFlag* software may be started in order for *RoboFlag* to properly interface with the testbed.

1. Start all of the robots. Turn on each robot, making sure that all of the necessary batteries are charged and attached. In the case of the Kelly robots, make sure that the fan switches are switched on.

2. Login to each vehicle with `ssh`, `telnet` or `rsh` (typically each in a different window) and run the appropriate controller. For the Bats and Steelebots, the controller takes the vehicle's hat number as a mandatory command line argument. This is not the case for the Kellys as they determine their own hat number by assuming that it is correlated to their IP. Therefore, always make sure that Kelly 1 (mvwt1) is wearing hat 1, Kelly 2 (mvwt2) is wearing hat 2, etc. Also, do not forget to use the `run RFController ...` syntax on the Steelebots.

3. In the `mvwt/users/roboflag/MVServer/` directory, run *MVServer*. It should report a message indicating that it is waiting for the *RoboFlag* Arbiter to connect. Once the *RoboFlag* software is started, the message should change to indicate that the server is running.

4. Start the *RoboFlag* software as directed on the website, skipping the step that starts the simulator.

5. Play *RoboFlag* as usual.

Notes:

- *MVServer* will run until either 'q' is pressed on the keyboard or the *RoboFlag* Arbiter disconnects. In either case, the *RoboFlag* game will effectively be over (since one cannot play a *RoboFlag* game without seeing or controlling any robots) and *MVServer* will have to be restarted in order to restart the game.

- It is common to have many ssh/telnet/rsh clients open for the duration of the game as there should be one connection to each robot in play at all times. For the most part, this connection will be silent, that is, the *RoboFlag* controller will generally not report anything back to the user unless an error occurs. Furthermore, the *RoboFlag* controllers will not terminate upon the closing of *MVServer*. To terminate the *RoboFlag* controllers at the end of the game, use the `sendcommand` utility found in the `mvwt/users/roboflag/MVServer/` directory. See appendix B for more on `sendcommand`.

A Checking *RoboFlag*/*MVWT* software compatibility

*MVServer*interfaces with the *RoboFlag* Arbiter over the network to perform two tasks:

1. To forward vision information from the vision computer to the Arbiter

2. To distribute commands, received in bulk from the Arbiter, to each vehicle

To accomplish these tasks, *MVServer* and the Arbiter have to use the same data representation when exchanging data. This representation is given by the *RawVision* and *ObjectCommands* struct definitions found in `DataTypes.h`. In order for *MVServer* to work with a given version of the Arbiter, the definitions of *RawVision* and *ObjectCommands* used by *MVServer* and the Arbiter must match. Check `mvwt/users/roboflag/MVServer/RoboFlag/DataTypes.h` in the *MVWT* module against `datatypes/DataTypes.h` in your copy of the *RoboFlag* release. Be sure to check that the base data types and the data types of the elements match as well.

Additionally, when connecting, the *RoboFlag* Arbiter performs a "handshake" of sorts with the server. The constants and data types in the file `CommType.h` are needed to do this. Again, in order for *MVServer* to work with the Arbiter, the version of this file used by *MVServer* must be equivalent to the version used by the *RoboFlag* Arbiter. However, typically this file does not come with the *RoboFlag* release since the release does not include the Arbiter code, just the executable. This file seems unlikely to change and therefore it is not really necessary to check this file. However, if you have access to the source code of the *RoboFlag* Arbiter compare the *MVServer* file `mvwt/users/roboflag/-MVServer/RoboFlag/CommType.h` against `_common\network\CommType.h` in the Arbiter code.

B The `sendcommand` utility

One of the utilities that accompanies *MVServer* is `sendcommand`. It is useful for sending commands directly to vehicles without relying on *MVServer* or the *RoboFlag* Arbiter. From the command-line you specify which command to send and the vehicles to which the command. The syntax is:
`sendcommand [-h] [-i | -h | -x` *XVEL* `-y` *YVEL*`] [-p` *PORT*`]` *HOST* `[`*HOST*`...]`
Options:

`-i` Command the vehicle to go into idle mode, if possible

`-s` Command the vehicle controller to shutdown (this is the default)

`-x` Command the vehicle to move with an x-velocity of *XVEL*

`-y` Command the vehicle to move with a y-velocity of *YVEL*

`-h` Print a usage message

`-p` Set the port to which the command is to be sent (the default is 2020)

Notes:

- `sendcommand` only sends one command. Therefore, if multiple command types are specified on the command-line then only the last one is used.

The exception to this rule is that both `-x` and `-y` may be used to specify both components of the velocity command. If only one of these two is used, then the other component is set to zero.

- *MVServer* does not shutdown the controllers when it terminates. To terminate the controllers running on the individual vehicles, you must use `sendcommand`. Since the default action is to send a shutdown command, simply typing:
  `sendcommand` *`robot1 robot2 ...`*
  where *robot1*, *robot2*, etc. are the hostnames of the robots currently in use.