Pluto Software Documentation

John Folkesson

May 13, 2003

1 Introduction

Pluto's architecture is based on mobility which is based on CORBA using the omni ORB. One can read the ATRV2 software manual to learn about mobility, the name service and MOM (mobility object manager).

To run anything on pluto one must first have the name service running, (type name -i or name -r). When I use the word 'module' below I don't usually mean a linux module but rather an ordinary process run as a mobility module.

The individul modules of pluto's software are run as seperate processes which communicate using the mobility corba classes. One can view the currently loaded modules in MOM. Most modules will try to start any modules they need, if they don't find them already running. This can lead to problems however so I suggest starting the lowest level modules first with the startbasic script or running the indiviual scripts. (see below for a discription of the scripts). The reason is that some of the servers need to do some initialization of the sensors before they are up and running. That delay can foul things up. Check that everyone is really sending data before starting anything higher, (use 'mom', 'ps -a -f' and, of course, 'killall -9').

2 Architecture

As of this writting, we have the following modules running on Pluto:

- Hardware:
  - ATRV2 (baseserver),
  - dmuModule (inertial sensor),
  - sickModule (laser scanner),
  - wristModule (pan/tilt unit) and

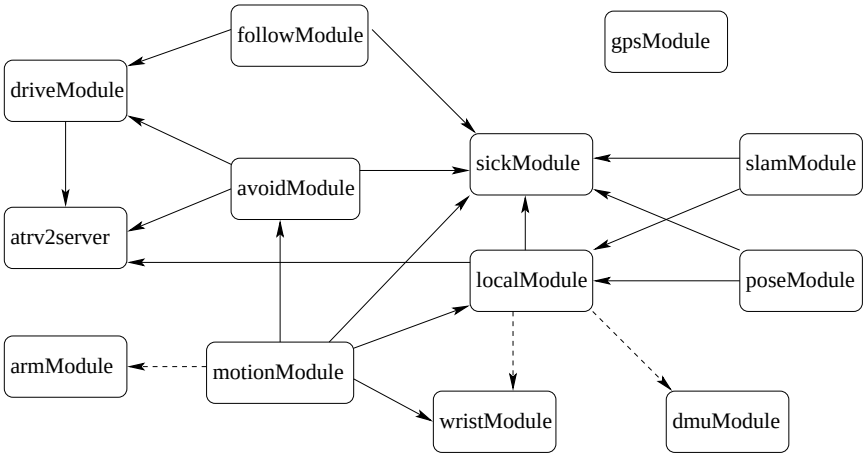

Figure 1: The arrows can be read ’knows about’ or ’needs’.

  - gpsModule.

- Navigation:
  - localModule (localization),
  - poseModule and
  - slamModule.
- Movement:
  - driveModule,
  - avoidModule and
  - motionModule.
- Other:
  - Follow Module

We have also had a compass and will soon add an arm, a weather and a gas server.

Then finally we have, not a module, (has no mobility interface), but rather a translation program between mobility and a much simplier xml interface using sockets: the xmlServer.

In additon there is a data collection program to save data to disk which also can be used to just start up a bunch of sensor servers.

Now I will try to lay out the big picture by describing the main parts of the architecture.

The most important module is the baseserver. This was provided by Real World Interfaces with the robot. It provides control of the motors, odometry, sonar and bumpers data. Its interfaces are described in the software manual pages for the ATRV2. It is important to understand these interfaces as the Navigation and Movement interfaces are based on the baseserver odometry and drive interfaces with enhancements.

3 Navigation Summary

All one needs to understand about the Navigation package is the localization service. The localization service, (localModule), uses the odometry, dmu and wrist orientation data along with timing information from the sick to provide five different pose interfaces. Besides fusing the odometer and inertial data it provides the other programs an interface where they can find the pose information they need. It also provides a way for them to share improved pose information with any other modules that may benefit from it. Thus, pose information is the best available regardless of whether the poseModule, the slamModule, neither or some new pose estimation process is running.

The five pose interfaces of the localModule have the CORBA names:

- Localization/Dead-Reckoning
- Localization/Laser_Dead-Reckoning
- Localization/Post_Pose
- Localization/Pose
- Localization/Laser_Pose

The first interface, "$Localization/Dead-Reckoning$", is just a dead reckoning estimate of the position of the robot since the localModule started. (The starting

pose is read from a file). This is useful if continuity of the pose is a requirement. This estimate will drift over time but will not suddenly jump to a new value. The pose is of a point on the ground where the 'X' formed by lines between opposite wheels, crosses, (in short, the center of the robot). The pose is 6 numbers (x, y, z, theta, phi, psi). The angles are defined as theta around the z-axis (z is vertical up in the robot frame), followed by phi about the y-axis (y is left in the robot frame), followed by psi about the x-axis, (x is forwards in the robot frame). Thus, the angles (0,0,0) have the robot flat facing the x direction.

The other interfaces are best understood by describing how the pose (and slam) module uses the interfaces.

The poseModule only communicates with the sickModule and localModule. No modules need to know about pose (or slam). The pose module reads from the "$Localization/Laser\_Dead-Reckoning$" interface which updates as each new laser scan becomes available. This interface gives the pose of the laser and the wrist at the time the laser scan was taken, (ie. before it was sent out over the serial cable). Also provided is the estimated error covariance in the difference between this pose and the previous one. This increamental covariance is needed to do the predict step in a kalman filter.

The poseModule then calculates an improved estimate of this pose and writes it to "$Localization/Post\_Pose$". The localModule reads this interface and uses it to calculate an adjustment to the dead-reckoning estimates. The last two interfaces then coorespond to the first two but after the adjustment and with the covariance matrix cooresponding to the best estimate of the absolute pose error.

The five interfaces are:

- (0) "$Localization/Dead-Reckoning$" - Dead-reckoning of the current pose from start of localModule.
- (1) "$Localization/Laser\_Dead-Reckoning$" - Dead-reckoning of laser and wrist at the time of the latest scan. This is different in space and time from the first interface. The wrist pan and tilt are appended to the 6 other pose dimensions for a 8-D pose vector. The covariance refers to (x,y, theta) only.
- (2) "$Localization/Post\_Pose$"– Input to the localModule of pose improvments. If cmdFlags<0 this is interpreted as a robot pose otherwise it is a laser pose. The pose uses cmdFlags 1 and the slam uses 2 and 3. cmdFlags==0 indicates that no one is updating this interface.
- (3) "$Localization/Pose$"– 0 adjusted by 2.
- (4) "$Localization/Laser\_Pose$"– 1 adjusted by 2.

I start numbering from 0 as this is how the loc_proxy numbers its interfaces. When one creats a loc_proxy one must specify which interface(s) one plans on using. This is done by passing an integer equal to the sum of $\mathbf{2}^n$ for each interface. So if you want interfaces 0 and 3 you pass 9 (=1+8) when creating the loc=new loc_proxy(argc, argv, 9); Thereafter one can call loc->get_pose(p_pose, 3); for instance to read the third interface.

Most other modules will want to use either interface 0 or 3, depending on whether continuity or global consistancy is important. If the module is using the laser it will need interfaces 1 or 4.

4 Movement Summary

The movement modules are arranged in a chain. The lower down the chain a module is the simplier the behaviour and the smaller the time and distance scales are. At the bottom is the baseserver. It has a command interface of type *ActuatorState_i*. The other modules mirror this interface but depending on the cmdFlags may modify the command as it passes thru them.

What I mean to say is that each link in the chain reads from its command interface (an *ActuatorState_i*) and writes to the next link's command interface, after possibly modifing the command. Thus, at the top of the chain (motionModule) the command has many possible cmdFlags, while at the bottom, (baseserver) only cmdFlags=0 is possible).

In order to control this chain some information must be sent the other way, up the chain. For that the status interface is used (a *FVectorState_i*). It starts out at the bottom with very little information (the baseserver has no status interface at all), and contains more information the as one moves up the chain. The chain links, along with their corba command and status interface names are:

1. "ATRV2/Drive/Command" (baseserver)
2. "drive/command" "drive/status" (driveModule)
3. "avoidance/command" "avoidance/status" (avoidModule)
4. "motion/command" "motion/status"; (motionModule)

The baseserver just provides velocity control of the motors. One can set a linear and angular velocity, (command.velocity[0..1]). Actually the base server can do more than this but this is all I use.

The drive server provides some collision protection, dynamic constraints and a regulation loop to hold a heading. It stops the robot if the bumpers are pressed and limits the accellerations. If one sets the cmdFlags to 1 one can specify a speed and an angle to make the the robot turn to the angle and move at the speed, (command.position[1] and command.velocity[0]). If one adds 10 to the cmdFlags the

drive adds sonar collision protection.

This is the general pattern but the list of options gets longer. The avoidModule can do goto commands and uses the laser and sonar to steer around obstacles (rather than simply stoping as drive does). One can now give an x and y to goto but if there are complex structures in the way the robot will not be able to drive around them.

The motionModule carries out relatively complex motion behaviours such as explore, moveto (goto with simple path planning) and investigate. It controls the wrist as well in order to do active scanning, moving the laser to see to the side or the ground.

5 File Structure

The programs directory of the mobility user contains all of our code. It has a src directory (headers and .cpp files), a lib directory (object code) and a bin directory (executable files). This is nicer than having everything mixed up as the object code is huge to back up and backups of it should not be needed.

Most of the modules have a similar file structure for the source code. The main program is in a file called *xxxModule.cpp*. This creates the corba interfaces and the control thread and then goes into a $default\_main\_loop()$. The control thread is usually in a file called *xxx_control.cpp* and may or may not start automatically. There are then other files to do the work of the module, typically a *xxxer.cpp* file and maybe more. In some cases the *xxx_control.cpp* file uses an assistant controller called *xxx_boss.cpp*, if the control would otherwise become too complex.

The make files are included in each source directory. For the higher directories there are three scripts included, makeit, cleanit and tarit. They do that to everything in the lower directories. Usually one does a backup or file transfer of just the source code by typing tarit; If you want to force a recompile of everything type cleanit; makeit; This of course can take some time so plan on doing something else while everthing recomplies.

To run each module a script is included that runs the module in the backround as a deamon. These scripts have names *base_server*, *dmu_server*, *avoid_server*, *pose_server* ... By specifing '-k' after the script name one can kill the deamon.

To facilitate using the modules, each module has one or more proxies. These are in the proxies directory. If one needs to communicate with a module one need only include the proxy file and then create a proxy object in the cpp code. The proxy will find or start the needed module and set up communication. It then translates simple c++ method calls into the neccesary sequence of corba messages to carry out the command. One then need not concern onself with the protocol of the indiviual modules just the simplier proxy methods.

For some sensors the proxies can be replaced by readers which read data from a file instead of live data.

In addition, each module has a simple user interface that allows control from the keyboard. These are in the sub-directories called interface. They have the names *xxx_i.cpp*. These interfaces are a good place to look to see how to use the proxies in a real program.

One more nice thing, in the dummies directory one can find a dummy base server, as dummy sick server and a dummy wrist. These can replace the real thing and will allow one to run all the pluto software as if the robot were moving without it moving. The odometry then will have no error and the laser scan is based on the odometer position and a map. The sonar is just 8 meters and the bumpers are always open.

By running the dummy servers one can test obstacle avoidance or other type control programs without having to take the robot out. That saves time believe me. One can even test navigation programs.

These simulations really are crude but one can work out most of the bugs with them. They are not suitable for inclusion in reports as they really don't give a picture of performance do to the ideal sensors. Actually as of this writting the dummies are so crude it seems that if things work on them then they are robost indeed and will work on the real robot.

Now I will try to describe how to use the individual modules in detail.

6 Movement

6.1 The Drive Module

The driveModule provides some collision protection, dynamic constraints and a regulation loop to hold a heading. It stops the robot if the bumpers are pressed and limits the accellerations. If one sets the cmdFlags to 1 one can specify a speed and an angle and the robot will turn to the angle and move along the direction at the speed, (command.position[1] and command.velocity[0]). If one adds 10 to the cmdFlags the drive adds sonar collision protection.

The driveModule has a status interface of type:
MobilityData :: *FVectorStatestatus_i* (corba name="status"). This interface has a time stamp, a vector of floating point numbers, data[2] and a vector of integers element[1].

The element[0] and data[0..1] are used to send information back to the controling object about obstacles that have caused the driveModule to cancel a command. The data[0] and data[1] contain the new, slowed down, velocity values. This is needed to avoid the situation of the controller thinking the robot is moving when it

is not. Without this the user interface feels very strange when drivig by hand. One hits the up arrow to go after the robot has timed-out and the interface would send a speed a little greater than the last command, which may be totally unexpected. By first reading the status the interface knows the working command value of the driveModule.

When the sonar (or laser using the avoidModule) detects an obstacle it slows the robot to a safe speed, eventually stoping the robot if needed. element[0] indicates where the obstacle was detected:

- element[0]=0 indicates normal operation.
- element[0]=-1 indicates drive timeout (about 4 sec).
- element[0]=-10 indicates the drive thread is stoped.
- element[0]=1 indicates the front bumper.
- element[0]=2 indicates the back bumper.
- element[0]=3 indicates the sonar front.
- element[0]=4 indicates the sonar back.
- element[0]=5 indicates the sonar sides.

The command interface is an *ActuatorState_i*. It has a time stamp, an int cmdFlags, a float position[2] and a float velocity[2]. The cmdFlags should be set to 0, 1, 10 or 11, as described above. For cmdFlags 0 or 10 the velocity values will be used to set the linear and angular velocities. For cmdFlags 1 or 11 the position[1] is used to set the direction (and angle defined by baseserver interface "ATRV2/Drive/Raw"), that the robot will drive in. While velocity[0] will set the linear velocity. In this mode the robot turns smoothly to the direction and then maintains a straight line path. This is used by the avoidModule goto.

The drive will also timeout after not getting any new command for about 4 seconds. If one is using the avoidModule goto this timeout will not occur but when driving by hand it can cause the robot to stop. This is in case the user interface crashes and stops passing commands to the robot.

One last thing the drve does is calibration of the baseserver odometry data. The calibration data are read from a file and written to the baseserver when the dreive module is started.

6.2 The Avoidance Module

The avoidModule has a status interface of type:
MobilityData :: *FVectorState* (corba name="status"). This interface has a time stamp, a vector of floating point numbers, data[2] and a vector of integers element[2].

The data and element[0] are mostly passed on from the driveModule's status. The element[1] indicates the simple avoid goto status. The avoid goto is a straight line goto with simple avoidance.

- element[1] = 1 goto point is reached, with cmdFlags=(02, 12, 22, 32).
- element[1] = 2 goto is stuck (sonar detected something too close).
- element[1] = -1 goto is normal.
- element[1] = -10 if avoid thread is stoped.
- element[1] = 0 otherwise.

When sending a goto command one sets command.position= x,y (this is in the "Localization/Pose" frame). Then one can check element[1]==1 to see when the robot is there.

One might want to stop the avoid thread if one wants to control the drive module directly without passing the command through the avoidModule. One can see if the thread is running or not by checking element[1]==-10. All threads can be started by sending a start_activity command, (see the proxy). Threads will not care if you try to start them twice, so don't worry about that. If you want to stop the avoid thread you can also do that by sending the release_robot command $(cmdFlags = -1)$.

The avoidModule also provides laser collision protection similar to the driveModule sonar protection. It then changes element[0] and data if an object is detected too close with the laser.

- element[0]=7 indicates the laser scanner has had a fatal error and is rebooting.
- element[0]=6 indicates laser slow/stop.

The value of data[0..1] is set to the new velocity command values.

The command structure is similar to the drive. The cmdFlags are comprised of two parts, the command type and the avoidance. These correspond to the ones and tens digit in the cmdFlags=10*A+T.

- A= avoidance,
- A=0 both off
- A=1 sonar detection only
- A=2 laser detection only
- A=3 both on
- T=command type,
- T=0 Velocity control of driveModule, specify velocity[0..1]
- T=1 Regulator mode of drive (specify velocity[0] and position[1]) =>drive will hold that speed and angle. (now relative to interface (3) (adjusted robot pose) of the localModule).
- T=2 goto mode (specify position[0..1], velocity[0..1] and force[0])

  position="Localization/Pose" x,y of the goal point.
  velocity=maximum linear and angular velocities.
  force =1 default
  >1 makes avoidance weaker (goal stronger)
  <1 makes the avoidance stronger

- T=3 wall follow

In addition, as described above, one can use cmdFlags=-1 to stop the avoid thread.

The goto command, $(cmdFlags = 2, 12, 22, 32)$, will try to go around obstacles using a vector summation rule. It reacts to near ($<$3m) obstacles using the laser and sonar. It then will steer around the obstacle. This works well for some obstacles such as people. For buildings the goto is rather clumsy driving right up to the wall then maybe following it slowly to the corner. It can easily get traped. The motion module uses this goto after determining that the path is probably clear.

The wall follow command is designed to hold a camera, extended from the back and to the left of the robot, at a safe but constant distance from the 'wall'. The robot will move along the wall in this posture. One should turn the wrist to the left so the laser gets a good view of the wall near the camera.

6.3 The Motion Module

As stated above the motion module has a status interface of type: *MobilityData* :: *FVectorStatestatus_i*(corba name="status"). This interface has a time stamp, a vector of floating point numbers, data[2] and a vector of integers element[4].

The data and element[0..1] are simply passed on from the avoidModule's status. The element[2] indicates moveto status. The moveto is a goto with better path planning. It will move the robot down the middle of the street and it will try to go around buildings by moving to intersection and then turning, rather than driving right at the building untill it gets very close.

The moveto mode is entered at the end of explore to find home. It can also be useful as a better goto but may not work as well as avoid goto in some special situations. By monitoring element[2] one can tell when moveto is finished or wether it has started.

- element[2]=-1 if moveto point set (cmdFlags 42 received)
- element[2]=-2 if nearing moveto point (now using simple goto)
- element[2]=2 if investigate done
- element[2]=1 if at goal
- element[2]=0 otherwise

The element[3] indicates the motion status it can be used to see if explore has started, the rectangle was set ... This is set in response to $40 <$ cmdFlags < 50. When just passing commands to avoidace element[3] will either be 0 or -1.

- element[3] = -10 if motion thread is stoped.
- element[3] = -1 not in motion after rectangle set
- element[3] = 0 if not in motion rectangle not set
- element[3] = 11 if investigate moving towards object.
- element[3] = 12 if investigate circling object.
- element[3] = 1 if scaning for new paths
- element[3] = 3 if following a path

One might want to stop the motion thread if one wants to control the drive or avoid modules directly without passing the command through the motionModule. One can see if the thread is running or not by checking element[3]==-10. All threads can be started by sending a *start_activity* command, (see the proxy). Starting the motion thread also starts the avoid thread. Threads will not care if you try to start them twice, so don't worry about that. Stoping the motion thread does not stop the avoid thread. If you want to stop both the motion and the avoid threads you can do that by sending the *release_robot* command (cmdFlags=-1).

The motionModule has a rectangle which can be set a few ways. This rectangle limits the motion of the robot to the region inside the rectangle. To see if the rectangle really was set one can check (element[3]==-1), but the robot must not be in motion at the time (send a stop to he robot first, then a resume after).

Otherwise one can see element[3]>0 means that something is going on, motion.

The command interface is an *ActuatorState_i* and has a time stamp, an int cmdFlags, a float position[2], a float velocity [2] and a float force[0]. The cmdFlags <40 are simply passed thru to the avoid module.

The commands that are processed by the motionModule are:

- 42 moveto using *ray_following* position give (x,y)
- 43 investigate
  move to closest object
  unfold arm
  circle the object.
- 44 explore:
  set rectangle: (b,h)=velocity[0..1] and
  home: (x,y,theta)=current robot pose and
  then => start explore
- 45 start/resume explore/goto
- 46 just set
  rectangle: (b,h)=velocity[0..1] and
  home: (x,y)=position[0..1], theta=force[0]

The release the robot was described above. One can set cmdFlags=42, position[0]=x and position[1]=y to start the moveto behaviour. The robot will try to move to point x,y (="Localization/Pose"). The robot will not try to go through a wall but will see

that the way is blocked and look for the best open path (outdoors defined as 1.5 meters on each side of the robot). It will then follow that path (ray) checking both sides, right and left, as it goes to find any openings. If it finds an opening it will stop and decide if it is better to turn here (set a new ray) or to continue.

The motion can be stoped by sending a -1 $<$ cmdFlags $<$ 40. The same motion can then be resumed by sending cmdFlags=44. The robot may become confused if the situation has changed a lot (ie. you drove the robot far from the current ray), but it will soon return to some kind of correct behaviour (ie. pick a new ray).

The rectangle is defined by a base and hieght (b,h) and a home (x,y,theta). The home is a point at the center of one side (base) of the rectangle with theta pointing perpendicular to the side into the rectangle. b is the distance from the home point to each of the sides perpendicular to the base, (ie. the base length is actually $2*$b). The height is then the length of the sides perpendicular to the base. By setting b=0 one can remove the rectangle and the robot can then move anywhere.

7 Navigation

7.1 The Pose Module

The pose module uses a priori map and the laser scanner to improve the dead-reckoning of the localization module. It first extracts features from the laser scan then matches them to the map. It then feeds this along with the dead-reconing pose into a Kalman Filter to get an improved pose which it writes to the localization module as described above.

When the thread is started the pose is read from the localization server. It then starts updating untill the thread is stoped. While the pose module is updaing the localization module the 'setter' (see *loc_proxy*) is 1. This can be read by looking at cmdFlags on interfaces "*Localization/Post_Pose*", "*Localization/Pose*" and "*Localization/Laser_Pose*". When the thread stops the setter is changed to 0.

Aside from stoping and starting the pose thread one can use the pose Module to find the longest wall in front of the robot. This is used by the *pose_i* to set the odometry calibration values used by the baseserver (read from a file by the driveModule and then set in the baseserver). One can use this to just measure angles and distances of walls to, for instance, set the pose. One should look in the *pose_i* and *pose_proxy* files to see how to use this. The calibration needs to be set after changes to tire pressure.

To set the pose one needs to first stop the pose thread. Then set the pose in the localization server, use setter=cmdFlags=-1 if you want a robot pose. Then start the thread again. One needs to wait between commands untill the localization interface has changed setter. The *pose_i* gives an example of how to do this.

Otherwise it is only to be sure that the pose thread and the slam thread are not both running at the same time. Note that it starts out with the thread active as default. It loads the map from an xml file (you can specify -m filename). There is a default mapfile if none is specified.

The poseModule uses setter = 1 (see *loc_proxy*). The pose has a command/status interface of type MobilityData::FVectorState. element[0] tells the status of the thread. It is 0 if the thread is stoped and 1 otherwise. When the thread stops it changes the setter to 0. Notice that there is only one command here. One should not change anything except element[1]=2 as described below. Otherwise this is really a status interface.

The element[1] is used by the find wall. Set element[1]=2 to start the find wall. The poseModule then sets element[1]=-1 while it searches for the wall. Then element[1] is set to the number of scan points in the wall (a non-negative number). The data vector contains 6 parameters of the wall (gamma,rho and covariance).

7.2 The Slam Module

The slam module uses a compressed Kalman Filter to creat a map while also using the map the improve the pose estimates. Aside from saving the map not doing odomtry calibration it works like the pose module. The slam thread does not start automatically when you run the slamModule script, one must start it when ready.

The slamModule uses setter = 2 (3 during save). The slam has a command/status interface of type MobilityData::FVectorState. element[0] tells the status of the thread. It is 0 if the thread is stoped and 1 otherwise. When the thread stops the setter is changed to 0. Notice that there is only one command here. One should not change anything except element[1]=2 as described below. Otherwise this is really a status interface.

Set element[1]=2 to save the map. The slamModule then sets element[1]=-1 while it saves the map. While the map is being saved the setter is changed to 3. When the map has been saved element[1] is set 0 and the slam estimate cycle continues.

When the slam thread is stoped the map is saved so one should wait.

8 Hardware

8.1 The Sick Module

The sick Module can setup the laser scanner baudrate, mm/cm mode and degree/half degree mode. The default is cm mode (80 m max), half-degree (361 scan points) and 38400 baud. This gives new scans about every 200 ms.

The sickModule has 3 different interfaces one is a copy of the original rwi interface. It was made to allow old programs to run with the new server without needing any changes. It forms 181 scan points with a time stamp at the time the laser data arrived at the serial port.

The second interface gives a timestamp at the time the scan was taken, (the center time of the scan. This time is about 200 ms earlier than the first interface. It also gives mm data and the number of scan points available.

The third interface is a timing interface. It is used by the localization service to find out when to interpolate the dead-reckoning to. The data contains one element the number of scan points.

All data is set to -1 if a fatal error occurs. This type of error requires the sick server to reboot the sick sensor and takes about 20 seconds. No laser data is available during this time. I should say that the Corba interface does not allow one to change any settings on the sick scanner. For that one must go in and change the source code and recomplile, easy for me hard for you.

8.2 The Dmu Module

The dmu has two interfaces. The DMUraw_state is just the data off the crossbow inertial sensor with the z axis changed to point up, (x forwards and y to the left). One then has the pitch and roll angles (kalman filtered in the sensor to agree with gravity in the long term) and the angular velocity and acceleration for the three axis (instantaneous).

The raw data are really hard to use right so use the DMUstate interface instead. There I low pass filter the velocity and acceleration. I then integrate out the heading, theta, (position[0]). This becomes the estimated total rotation about the vertical relative to the starting orientation. In other words, I can describe what it is without using my hands and feet. The pitch and roll angles are as before and correspond to the euler angles. The three position[0..2] are then theta about the vertical followed by phi about the y-axis followed by psi about the x-axis. The timestamp has been delayed somewhat to try and account for the filter delay.

8.3 The Wrist Module

The wrist has a can interface one must load the can driver (a linux module) with insmod before running the wristModule. The interface is quite simple. There is one command interface and one state interface. They both are of the type *ActuatorState_i*. The position, velocity and acceleration vectors are (pan, tilt) in rads, rad/sec, rads/(sec*sec). One must set both the velocity and accelleration >

0 if you want the postion to change. The proxy and interface are the best manual. The current position of the wrist can be read off the state interface.

8.4 The GPS Module

Ha! so you want to use this data. Don't use the proxy in gpsmeter to get data into a xy grid in meter units. Otherwise the interfaceis discribed in Mattias's README file withthe source code. Important information is in force[0]=number of satellites and force[3]=GPS quality. The number of satellites should be > 5 and the quality should be 2 for sub-meter accuracy.

8.5 The Arm Module

Olf and Andreas have given pluto an arm (not yet but soon).