Manual v0.2

Joseph O'Sullivan
josullvn@cs.cmu.edu

Karen Zita Haigh
khaigh@cs.cmu.edu

G. D. Armstrong
roboman@cs.cmu.edu

April 21, 1997

Abstract

Xavier is a mobile manipulator developed at CMU that is being used for research in robot learning and self-reliant autonomous operation. It is built on an RWI B24 base and includes bump sensors, a laser range sensor, sonars, a color camera on a pan-tilt head, and an arm. Control, perception and planning are carried out on two on-board, multi-processing 486-based machines. The robot can communicate with humans both graphically and through natural language.

The software controlling Xavier combines both reactive and deliberative behaviours. The reactive behaviours enable the robot to handle real-time local navigation, obstacle avoidance, and emergency situations. The deliberative behaviours, which operate concurrently with the reactive behaviours, maintain occupancy grid and topological maps, plan paths, interpret visual information, and do task-level planning. The behaviours are integrated using the Task Control Architecture, which provides facilities for scheduling and synchronizing tasks, resource allocation, monitoring the environment, and handling exceptional situations.

In addition, Xavier has a detachable arm which can pick up boxes. This arm has a large V-shaped end effector with roller bearings to guide the box to the center, where two electromagnets were used to hook onto metal plates mounted on the corner of the boxes. The arm enables Xavier to lift the boxes over its "head," simplifying planning of subsequent movements.

This manual briefly reviews the physical components of Xavier, presents a simulation environment, and introduces a programming tutorial and overview for the novice roboteer.

Contents

Chapter 1

Introduction

Motivated by the desire to replace a Hero 2000 robot in Carnegie Mellon University's Learning Robot Lab, a robot was designed by a group of several students (under the direction of Reid Simmons) to provide a new test-bed for research and to participate as Carnegie Mellon University's entry in the 1993 AAAI Robotics competition. This required a tight time bound of 9 months from initial design to competition entry.

The name Xavier was chosen in honor of the leader of a band of comic super-heros named X-men – Professor Charles Xavier, wheelchair bound yet with 'astounding mental powers and abilities'.

Xavier is built on an RWI B24 base. The base is a 24" diameter, four-wheeled, synchro-drive mechanism. Sensors include bump panels, a Denning sonar ring, Nomadics laser light striper, and a color camera mounted on a Directed Perception pan/tilt head. On-board computation consists of two 200 MHz Intel 6x86 computers and a 486 laptop, connected to each other via Ethernet and connected to the outside world via a 2 Mb/sec wireless Ethernet. Xavier has a distributed, concurrent software system, which runs under the Linux operating system. All programming is done in C, and processes communicate and are sequenced and synchronized via the Task Control Architecture.

Communication with Xavier is mainly speech-driven. An off-board Next computer runs the Sphinx real-time, speaker independent speech recognition system and a text-to-speech board provides speech generation. Thus, we can give verbal commands to the robot and the robot can respond verbally to indicate its status. In addition, a graphical user interface is available for giving commands and monitoring the robot's status.

This manual will present a tutorial for altering basic hardware behaviours and will provide a synopsis of research which has been undertaken within the Xavier domain and an outlook on ongoing research and future directions.

1.1 Credits

The TCA overview is from the introduction to TCA programmers manual. This manual was compiled by Joseph O'Sullivan from the collective folklore of the Xavier team. In alphabetical order, with a list of specialties, they were:

Rich Caruana	cameras selection, laser system installation.
Lonnie Chrissman	frame-grabber selection, Motherboard.
Domingo Gallardo	base controller, basic navigation, simulator.
Rich Goodwin	system supervisor, software control procedures, simulator interfaces, arm driver, hardware maintenance.
Karen Haigh	planning, monitoring execution
Herman Herman	Box alignment and pickup arm design.
Wes Huang	power system.
Mark Ollis	simulator, grid based maps, laser enhancements.
Joseph O'Sullivan	speech, laser, pan-tilt, Mach installation, frame-grabber, serial hardware interfaces, hardware maintenance.
Reid Simmons	TCA, corridor navigation, team management.
Sanjiv Singh	arm design, box collection.
Hank Wan	basic simulator, topological maps, sonar and serial drivers, corridor navigation, hardware maintenance.
Andrew Weinstein	simulator enhancements

1.2 Obligations

Xavier and its software system is a constantly changing and growing environment thanks to its pool of users. By using Xavier and its source code for your research, we hope that you will put back in as much as you take out. As such, any utilities, libraries or programs which you construct should be documented, both internally and in the form of an overview README file. If additions are significant, extensions to this manual should be considered.

1.3 Selected Bibliography

If you would like to know more about Xavier, the following are a selection of papers which either involve Xavier, or aid in understanding some of the subsystems.

[1] Denning. *Denning Ultrasonic Ring*. Denning Mobile Robotics, Inc., Wilmington, MA, 1987.

[2] Directed Perception, 1451 Capuchino Avenue, Burlingame, CA 94010. *'Pan-Tilt Unit Model PTU User's Manual*, 1.0b edition.

[3] Illah Nourbakhsh *et al.* The winning robots from the 1993 robot competition. *AAAI Magazine*, 14(4):51–62, Winter 1993.

[4] Richard Goodwin. Reasoning about when to start acting. In K. Hammond, editor, *Proceedings of the Second International Conference (AIPS94)*. Artificial Intelligence Planning Systems, June 1994.

[5] Richard Goodwin and Reid Simmons. Rational handling of multiple goals for mobile robots. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, June 1992.

[6] Richard T. Goodwin. A formal specification of agent properties. Technical Report CMU-CS-93-159, Carnegie Mellon University, Pittsburgh PA. USA, May 1993.

[7] Kurt Konolige. Designing the 1993 robot competition. *AI Magazine*, 15(1):57–62, Spring 1994.

[8] David P. Miller. The long-term effects of secondary sensing. *AI Magazine*, 15(1):52–56, Spring 1994.

[9] Tom M. Mitchell, Joseph O'Sullivan, and Sebastian Thrun. Explanation-based learning for mobile robot perception. In S. Mahadevan, editor, *Proceedings of the MLC-Colt '94 Workshop on Robot Learning*, New Jersey, NB, July 1994. Available on the web via URL, `file://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/mind-matters.ps.Z`.

[10] MuTech. *Image / VGA-Composite Board User's Guide*. ScanTech, Inc., 1993.

[11] Nomadics. *Sensus 500 Users Manual*. Nomadics Technologies, Inc., 1992.

[12] Joseph O'Sullivan. The CMU learning robot laboratory data toolkit. In S. Mahadevan, editor, *Proceedings of the MLC-Colt '94 Workshop on Robot Learning*, New Jersey, NB, July 1994. Available on the web via URL, `file://www.cs.cmu.edu/afs/cs.cmu.edu/project/robocomp/xavier/www/papers.html`.

[13] RC Systems, Inc, 121 West Winesap Road, Bothell, WA 9801, Phone (206) 672-6909. *V8600/1 Speech Synthesizers Databook*, 1991.

[14] Real World Interface, Inc., P.O. Box 270, Main Street, Dublin, NH 03444, Phone (603) 563-8871. *B12 Mobile Robot Base Guide to Operations*, 2.1 edition.

[15] Reid Simmons. Expectation-based behavior. In *Proceedings of the International Symposium of Robotics Research*, Hidden Valley, PA, October 1993.

[16] Reid Simmons. Becoming increasingly reliable. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, Chicago, IL, June 1994.

[17] Reid Simmons, Richard Goodwin, Chris Fedor, and Jeff Basista. *Task Control Architecture: Programmer's Guide*. Carnegie Mellon University, School of Computer Science / Robotics Institute, Pittsburgh, PA, 7.7 edition, May 1994.

[18] Mark Stemm. Using artificial neural networks and image segmentation to assist in mobile robot navigation. Master's thesis, School of Computer Science, Carnegie Mellon University, ??? Year. Bachelor's Thesis.

[19] Telesystems SLW Inc., Toronto, Canada. *Arlan 140 User Manual*, 1990.

[20] Sebastian Thrun. An approach to learning robot navigation. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems*, Munich, Germany, September 1994.

[21] Sebastian B. Thrun. A lifelong learning perspective for mobile robot control. In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, September 1994. (to appear).

[22] David S. Touretzky, Hank S. Wan, and A. David Redish. Neural representation of space in rats and robots. In Jacek M. Zurada and Robert J. Marks, editors, *Computational Intelligence: Imitating Life*. IEEE Press, 1994.

[23] Hank S. Wan, David S. Touretzky, and A. David Redish. Computing goal locations from place codes. In *Proceedings of the 16^{th} Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates, 1994.

[24] Hank S. Wan, David S. Touretzky, and A. David Redish. Towards a computational theory of rat navigation. In Michael Mozer, Paul Smolensky, David Touretzky, Jeffrey Elman, and Andreas Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, pages 11–19. Lawrence Erlbaum Associates, 1994.

Chapter 2

Hardware

Our design aim when building Xavier was to use 'off the shelf' components. The resulting design meant that each individual component was mass-produced and well tested. If any individual component failed, maintenance or replacement was trivial. The only exceptions to this design philosophy occurred in the power system and in the interconnection between components. As such it is not surprising that the majority of down-time since Xavier has been in operation can be traced either to failure in the homebuilt power system or due to interference between components. The original power distribution has since been replaced with one designed and built by RWI, Inc. (the makers of the robot base), for use with their B21 robot system. There have been fewer faults with the power systems since then. Where component failure has occurred, it has generally been straightforward to remedy by referral to the manufacturer/supplier of the part.

In this chapter, each of Xavier's component systems will be described.

Range-finding Devices:

These are the devices which yield range information of objects relative to Xavier. They come in three forms, differing in sensitivity, noise resistance and usefulness;

Sonar:

This has a broad range (extending about 25 feet in a 360 degree sweep), is fairly insensitive, and is highly susceptible to noise. Due to its broad range, it is currently the most useful sensor.

Laser:

Narrow range (extending about 10 feet, in a 15 to 20 degree arc) aimed in the direction of motion, extremely accurate (under three centimeters error), and high noise resistance. Due to its narrow range, it is currently being used for precise placement tasks.

Touch:

Broad but close range, most accurate, most resistant to noise. Due to its distance limitations, it is the range device of last resort.

Vision System:

We will describe the color camera in current use, the hardware being used to capture images, and the hardware used for pointing the camera.

Motion:

We will describe the status of the base in terms of power, range, and maintenance.

Communication:

We describe both off-board language recognition and on-board speech synthesis hardware.

Computing Power:

The current computing machinery will be reviewed in conjunction with its operating system and patches necessary for robot control.

Wiring:

This has proven to be the most unreliable component on Xavier - its flaws will be discussed.

In addition, this chapter will conclude with a trouble shooting section which distills the experiences of the design and user teams.

2.1 Range-finding Devices

Xavier is approximately 2 feet in diameter, over a meter in height and is capable of moving in excess of 3 feet per second. At top speed close up sensing becomes less effective. Generally, Xavier should not be driven faster than it can sense. Xavier's sonar system offers 360° coverage but can presently check this area only 3 times per second. As such, Xavier could move up to a foot in this time. *JOSEPH: the math doesn't add up – sonar's OK to 25 feet, so need explanation.* The laser system provides a more directed range of view, covering 45° of view directly in front of Xavier 10 times per second – this is the area most likely to be next entered. In addition, the laser system provides less noisy data with enhanced resolution. (Whereas each individual sonar reading covers 15° corresponding to an accuracy within 2 feet when obstacles are seen at 10 feet, each individual laser reading is accurate to within 3 centimeters.)

The sonar system has been chiefly utilized for gross map making, while the laser system has been applied to precise placement relative to objects (such as for box picking

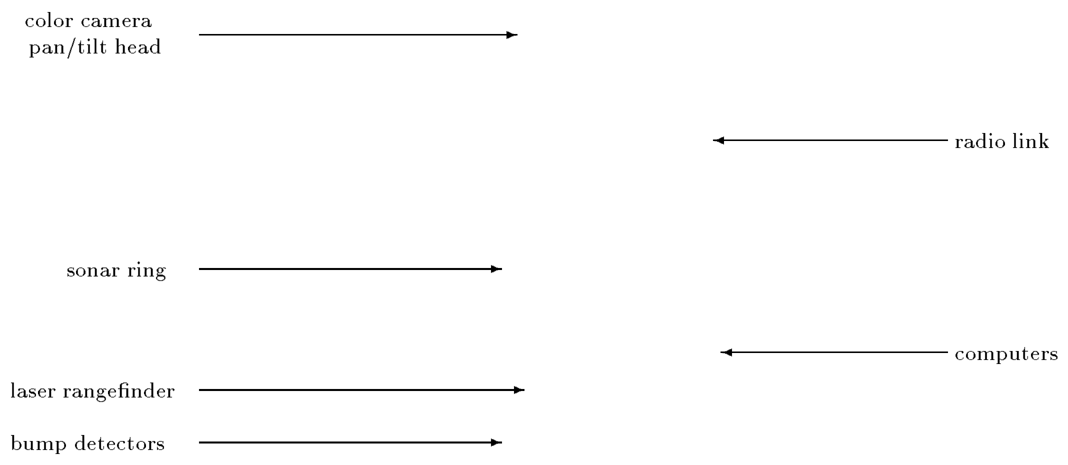

Figure 2.1: **Xavier** A side view of an open Xavier.

up). In addition, the laser system is highly configurable. The lasers can be placed at varying heights in different directions. If there is a specific object of interest in the environment, the laser system can be adjusted so that object appears uniquely. This method has been used to isolate chairs by placing the laser system parallel to the floor and precisely at the height of the chair seat.

We now proceed to a more detailed description of both systems and to describe the range-finding system of last resort: the bump panels which detect collision.

2.1.1 Sonar

Xavier uses a "Denning" supplied sonar system which is low power ($\approx$ 4 Watts), has a range from 4 inches to 20ft and can fire all of its 24 Polaroid sonars simultaneously [1]. It is seen as a proven design, and easy to control using a command language via an RS-232 interface. On the other hand, it is not capable of dynamic sonar sensing (sending out a sonar pulse immediately upon return of a previous pulse as opposed to waiting a fixed amount of time).

We first describe some of the constraints of the sonar technology. We then describe the decisions made in choosing the specific tradeoffs for Xavier. Finally, low level device management is described.

2.1.1.1 Sonar Questions and Answers

What's hard about using sound to sense? With any sensing device, we wish to read information as quickly as possible so that our information about the world does not lag reality. The problem with rapid sonar updating is that if a sonar can read a distance of 25ft, then it can take .05 seconds for the return pulse to arrive. To be safe, a potentially interfering sonar cannot be fired during this period. If we do, we may spuriously read the return pulse of the first sonar with the high power pulse of the second. Either of these errors can lead to a large error in the measured distance.

Note that interference usually causes false too-near readings because once a sonar receives an echo, it stops listening for more echoes. For interference to cause an error it must usually arrive before the true echo and hence lead to a false too-near response.

A safe sonar reading rate is therefore about .05 seconds per sonar. 24 sonars read at this rate takes 1.2 seconds. This is too slow for a robot moving at 3 feet per second if the sonar is the main obstacle avoidance detector. Sonars that are unlikely to interfere with each other can be fired simultaneously. Sonars that are most likely to interfere are near neighbors on the ring (see figure 2.2) and those 180 degrees (and, to a lesser extent, 90 degrees) away. A nice case is reading three sonars at a time, each 120 degrees away from each other. Then a 24 sonar ring can be updated at 2.5 Hz.

An alternate approach to rapid ring update is to selectively read the forward sonars at a higher rate and to only occasionally read the other sonars. For example, the five forward pointing sonars span 90 degrees or more and can safely be updated at 4 Hz.

One more trick is to trade off long range capability for speed. If the sonars are modified so that the maximum range that can be sensed is about 12 feet, then a 24 sonar ring can be read at 1.7 Hz if sonars are fired singly, at 3.3 Hz if sonars are fired 2 at a time, and at 5 Hz if sonars are fired three at a time. Also, 5 forward pointing sonars can be read at 8Hz (See table 2.1).

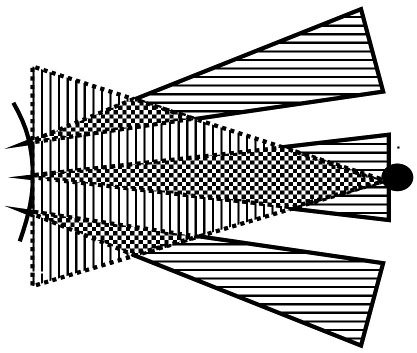

Figure 2.2: **Sonar interference** Reflections from a sonar may interfere with adjacent sonars. We use a 3 way interleaved pattern when reading sonars to reduce this problem.

No. sonars (dist)	1	2	3
24 (25ft)	0.80 Hz	1.60 Hz	2.4 Hz
16 (25ft)	1.25 Hz	2.50 Hz	3.75 Hz
5 (25ft)	4.00 Hz	-na-	-na-
24 (12ft)	1.67 Hz	3.33 Hz	5.00 Hz
16 (12ft)	2.50 Hz	5.00 Hz	7.50 Hz
5 (12ft)	8.00 Hz	-na-	-na-

Table 2.1: **Multiple Sonars** This table illustrates the trade off between having many sonars, and the firing rates achieved when avoiding interference.

Xavier fires four sonars spaced 90 degrees apart simultaneously, rotating in a counterclockwise pattern. We have found that the interferance is acceptable in exchange for a higher update speed.

Why were Polaroid vs Seimen sonars used? With Polaroid sensors, a tradeoff can be made between minimum sensing range and maximum sensing range by adding a dampening resistor to each transducer. The minimum range can thus be reduced to 3 inches this way while still maintaining a maximum range of 10 feet or more. (While this reduction should allow the sonars to be read faster, in practice the far range is more critical to us so this feature is seldom used.)

The Seimen sonars did not seem to be as viable an option for the sonar ring. They are fairly thick, and if mounted on the outside, add a total of about 4 inches to the diameter. Also, the Seimen sonars with sufficient maximum range has poor close range ability: it can't read closer than 2 feet.

Why use so many sonars? Several issues were considered in choosing an appropriate configuration. Since Xavier is large (more than 2 ft in diameter), 16 sonars leave gaps in sensing both near the robot and far away, though coverage at intermediate distances is complete. 24 sonars are sufficient to eliminate most of these gaps (see figure 2.3).

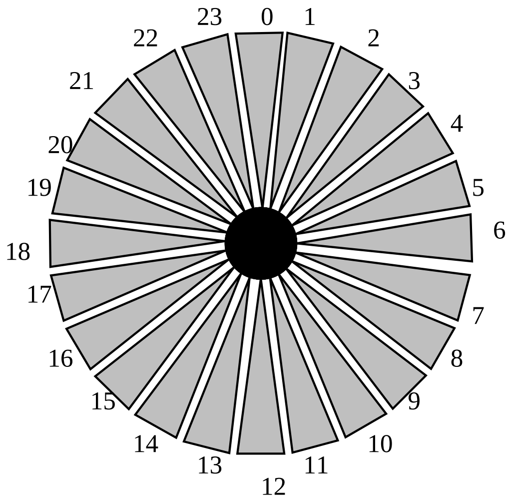

Figure 2.3: **Sonar coverage** Each sonar covers a span of 15 degrees and each are separated by 15 degrees about Xavier's circumference.

A robot with a diameter of 2 ft and 16 equally spaced sonars spaces sonars 4.7 inches apart. This spacing is reduced to 3.1 inches when 24 sonars are used. Reduced spacing minimizes gaps but makes crosstalk of simultaneously fired sonars more likely.

Sonar systems with 24 sonars typically require from .75 - 1.5 amps at 5-12 volts (5-15 watts). Roughly half of this power is used for the control and interface boards. the other half is used to generate the acoustic pulses and is only expended while sonar is running.

Such sonar rings can typically be read at rates between 1 and 3 Hz. Reading faster than this usually requires advanced techniques but is possible. The fastest readings published are about 6Hz. Admittedly, using fewer sonars allows the ring to be read faster, but the limiting factor seems to be interference (electrical and acoustic crosstalk) between adjacent sonars, not just interfering reflections.

Why are the sonars placed on the turret? The recommended height of the sonar ring is about 24–30 inches above the floor to eliminate the problem of false readings from cracks in the floor, floor molding, etc. (The ring could be placed lower if the sonars are aimed up). 24–30 inches is also a good height for detecting table-tops, chairs, etc.

The control boards for the sonars can take up a fair amount of space, requiring a total volume of about 8" × 8" × 8". Placement however is flexible as long as they are near the sonars they are controlling. The maximum length of wiring distance advisable is 18". Thus, we constructed a shelf specifically for carrying the sonar controllers.

What alternatives have been considered? The old HERO robot uses a single sonar on the manipulator to do sweeps. This is probably still a nice idea as it would make the location and identification of nearby objects (for example, bottles and cans) easier.

A close-range Seimen sonar (2.4" - 12") may be considered for this purpose. This sonar will output a logic level signal that can be timed by a general purpose I/O control board. Also, the short range of the sonar should let us read it fairly quickly (8Hz or better?).

We could also consider mounting 1–3 sonars forward looking downwards to recognize dropoffs such as stairwells. Again the Siemens sonars look like a good choice due to their reputed reliability.

Most controllers seem to allow flexible tasking of the sonar ring. For example, most allow a single sonar to be read separately. Reading multiple sonars simultaneously is a different story.

Other approaches are to use a single broadcast sonar pulse to "illuminate" a large part of the environment at one time. Then, use individual sonars in a conventional ring arrangement as detectors. The geometry is a little more complex for the time-to-distance calculation but this appears tractable. The main problem would be in generating a broad even pulse from a location on the robot that can "see" everywhere we want to sense.

Alternatively, we could fire all sonars at the same time. The main problems here are that sonars crosstalk via electrical connections, via acoustic transmission in the structure, and via acoustic transmission of their sidelobes, and that the electronics must exist to fire and read all sonars at the same time.

2.1.1.2 Sonar Hardware

The 24 Polaroid ultrasonic transducers are glued to a metallic ring. Dual wired cables clipped to the back of these transducers connect each to one of three transducer control modules. These contacts can become loose, especially when the ring is moved during re-assembly. They should be checked when a particular transducer consistently returns poor range data.

The ordering of the hookup is such that the 'zeroth' transducer is pointing directly forward and each subsequent unit is mounted in a clockwise fashion. The three control modules are connected to the sonar CPU card. Separate cables power the CPU card and the 3 control modules.

The sonar computer has a debugging port at 9600 baud and a high speed binary port at 19200 baud. While the debugging port is quite robust, any error in data to the high speed port can cause communication to get out of sync resulting in loss of response from the CPU card . When this happens, the only known way to re-establish communication is by resetting the sonar CPU card: an on-off switch is made to allow temporarily grounding of the active-low reset line on the cage on which the sonar CPU card is mounted. The switch needs be in the off position during normal operation.

In addition, the high speed port was found to have problems communicating to the PC serial boards. Because the serial boards use an older set of serial communication chips (UARTs) that do not support hardware buffering, they are unable to maintain communication with the sonar's high speed port. Instead, 16550 UARTs that support hardware buffering were used with the serial drivers of the Unix kernel being modified to enable this buffering (see section **??**). With this latter setup, there has been no problems with the high speed sonar data communication.

The transducer control modules have jumpers for selection of pulse width and receiver gain. The tradeoffs between near and far modes set by these jumpers are discussed in and briefly summarized in the sonar questions and answers. Throughout this project these jumpers are left in their factory default configuration.

2.1.1.3 Sonar Device Interface

Low level device interface routines, which are documented in `interface/include/sonar-interface.h`, allow access to various sonar firing parameters. These include specifying the number of sonar transducers to fire simultaneously, the interleave sequence, selection of sonar transducers to use, the delay between successive firing, and the maximum duration to wait for an echo to return.

The parameters in current use were empirically determined so that the sweep rate is 2.5 Hz. One out of each group of four sonar transducers is made to fire at any given time with an interleave factor of three. These are selected to yield the largest spacing between two transducers firing consecutively.

The sonar CPU board cannot be made to report range data continuously. Also, it makes a new set of readings available only when a full sweep is completed. A scheme is implemented in the software driver to poll the sonar CPU board periodically. The rate of polling is made to approximate the sweep rate.

2.1.2 Laser

A modified Nomadics laser system is used by Xavier[11]. The basic Nomadics system weighs 4–5 pounds, has a 45° field of view, is over a foot tall and is designed to be incorporated on top of a superstructure.

In typical lighting, we discovered that the effective plane sweeped by the laser was only 15 to 20 degrees and the overall range information produced not sufficiently accurate. The basic mechanism consists of a video camera scanning a scene. A laser produces a plane of light in the scene. The frequency of this light is in the high red range and is filtered by a custom built lens on the camera. Based on the knowledge of the offset of the laser from the camera, and the deviation of both, standard trigonometry yields range information. The greater the offset between camera and laser, the more accurate the range information (see figure 2.4).

There is a tradeoff due to the mechanical difficulties of building a structure suitable for maintaining a rigid deviation and the desire for this structure to be as long as possible, and yet freely adjustable. With Xavier, the

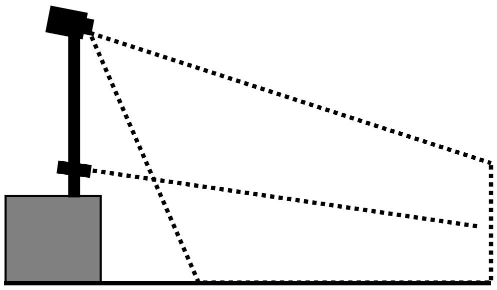

Figure 2.4: **Laser Range** The camera filters out all light except in the laser frequency. Calibration tables allow for the derivation of the exact locations of any objects intersecting the laser plane. The near point currently is 60 cm, the far point 300 cm.

use of a scientific optical bench pieces as mounting points proved to be a suitable compromise.

Xavier uses two lasers: one horizontally mounted to detect objects in front of the robot; and a second mounted diagnally to detect drop-offs in front of the robot.

2.1.2.1 Usage

Before use, ensure that the lasers are switched on. Each laser can be disabled by a switch near the laser mounts. The lasers may be switched on and off at anytime but should be on during robot use. Running CTR will cause a laser initialization sequence to be executed. This sequence will calculate an environment appropriate threshold level for laser filtering and will ensure that both lasers are switched on by software. CTR should remain running while the lasers are in use. CTR must be run on the machine which has the sonars and lasers (currently Wolverine). You must have the file “RangeTable” in the directory from which you are running CTR. A copy of this file can be found in `/afs/cs/project/robocomp/xavier/bin/RangeTable`.

The rangetable contains calibration information which allows rapid lookup of range information for the laser readings as opposed to performing triangulation operations based on the deviation and offset. Alternative range-finding mechanisms for the Nomadics system can be found in the Nomadics supplied documentation.

2.1.2.2 Calibration

As stated above, all calibration parameters for new configurations are contained in the range table file. Thus when the lasers and camera are set up in a new configuration, an appropriate range table must be generated.

1. A “comb” shaped target has been set up in the Learning-Robot laboratory for this purpose. Set the target out in a ‘V’ shape, with an interior angle of 120 degrees.

2. Place the robot facing the point of the ‘V’, with the center of the robot 60 cm from the front-most pole (see below). Make sure the robot has 2.5 m of free space behind it, as it will move during the calibration. **The quality of the calibration is very sensitive to the accuracy of this starting configuration**

```
x           x
  x (120) x
    x   x
      x    _
           |
           | 60 cm.
      O    -
```

3. Start up TCA and CTR; then run:

   `new_laser_calibrate`.

   (This is in the standard bin directory). The robot will back up 24 times in 10 cm increments, and write out the calibration table to “RangeTable”.

2.1.2.3 Statistics

1. The laser should be able to register obstacles with an RMS error of 3 cm, everywhere within its field of view (15 degrees, from a depth of 60 cm to 3 m). If your errors appear larger than this, try recalibrating.

2. There is a hardware bug in the way the threshold parameter works, so that it only returns reasonable results for a few (more or less random) values.

   The threshold can be set using

   ```
   LASER_set_threshold(unsigned char t).
   ```

   The default value is 62, which is reasonably good; for cleaner results at the expense of a smaller field of view, try using 80. Other reasonable settings are 126 and 254.

3. The quality of the results is not particularly sensitive to camera aperture; but they seem to be best at a setting somewhere between 2.8 and 4, regardless of lighting conditions.

2.1.2.4 Current problems

1. The field of view is slightly smaller now (about 3 degrees) than it used to be, because no extrapolation is performed outside the convex hull of the sample points. A good, numerically stable extrapolation routine would of course fix this.

2. As mentioned before, the calibration routine is extremely sensitive to the starting configuration, especially to being placed slightly off to one side or the other. There should be a way to make the robot line itself up automatically.

2.1.3 Bump panels

We use a bumper supplied by RWI, the base manufacturer, which is segmented into eight parts. These are OR-ed to give a signal to the base controller. Each can also be accessed sensed separately by the turret.

2.2 Vision System

The key criteria we required of a vision system was to have a head mounted color camera on a pan-tilt head capable of 340 degree pan and 78 degree tilt. A capability (unused) exists for a monochrome camera to be mounted on the arm to more easily determine the locations of objects relative to the manipulator. (This functionality will also be described).

2.2.1 Camera Requirements

Cameras were investigated in order to fill three sensing niches on Xavier:

1. An arm mounted camera which will allow the robot to perform close-up investigation and active sensing. Typical uses would be examining a stack of books in a library, examining button numbers in an elevator or checking to see if the grippers were closing around an object being picked up in front of the robot. A head mounted camera would not be as effective, since many applications exist where the arm is hidden from the head. This has been illustrated with the Hero robot. In summary, the ideal camera here would be small, low-power, high-resolution. Mounting difficulties will worsen as camera size increases due to more complex kinematics required to prevent the camera being damaged during arm movements.

2. A head mounted camera which would be used initially for a "Where" visual search followed by a "What" recognition. This camera would ideally have a deep depth of field and be wide angle in order to view as vast a panorama as possible. In order to aid in initial discrimination, as many visual senses as possible should be utilized in a coarse search of the scene. This would imply that a color camera would be beneficial.

3. In addition, a monochrome very-high resolution head mounted camera would be used to enhance capabilities on "What" tasks. Whilst the color camera should prove sufficient in most situations, instances are conceivable where higher light sensitivity and superior resolution would be of benefit (a poorly lighted room in one instance, supporting an ambiguous decision of the color camera in the second). With a B/W's low power (1.4W), low cost, and low weight (140g), the outlay is easily offset by the increased robustness and flexibility.

Since our laser system is being used for active 3-D range finding, the usage of stereo cameras seemed redundant and ineffective in comparison.

After an extensive search of the field of BCD cameras, two bony camera were found to fit our needs. These were the Sony XC-75 and the Sony XC-999.

The XC-75 is a high-resolution monochrome camera.

- It provides 768(H) x 494(V) pixel resolution, which translates into a TV image of 570x485 lines. This is as high as we have found and should be suitable for most tasks.

- In addition, it has a 1/3" pickup device. 1/3" as opposed to 1/2" means that comparable lenses on large devices have less field of view. Thus, the 1/3" pickup device will help in affording us maximum focal depth.

- The sensitivity is 3 Lux with an IR cut filter (About 0.5 lux without) which is superior to other cameras.

- Even more crucially, the Signal to Noise ratio is 56dB, 10dB better than most other cameras compared.

- It is compact, 44(W)x29(H)x71(D)mm and 140g, suitable both for the arm and the head.

- A final factor in the decision was the power consumption, 1.4W.

- It has output for an automatic mechanical iris. This is preferable to electronic gain control because depth-of-field increases as the iris is closed but does not change with electronic gain.

- It has an electronic shutter (1/125 - 1/10,000 sec). This may be essential to reduce blur during motion.

- The XC-75 uses a C-Mount lens fitting. This is the most common lens type, vastly increasing the number of lenses available for enhancing the optical input.

The XC-999 is a medium resolution color camera which is still low power.

- It provides 768(H) x 494(V) pixel resolution, which translates into a TV image of 470 lines (Horizontally). This should be adequate for most "Where" tasks.

- It has a 1/2" pickup device. While not as small as the XC-75, it is not significantly at a disadvantage and approaches the XC-75 closely wrt maximum focal depth.

- The sensitivity is 4.5 Lux which is worse than the XC-75 by about an F-stop, but superior to most other color cameras by the same margin.

- The Signal to Noise ratio is 48dB which is one of the best for a color camera.

- It is compact, 22(W)x22(H)x120(D)mm and 99g, which means that it can fit on the head along with another camera comfortably.
- It has a low power consumption, 3.5W.
- Similarly, the XC-999 has a number of features which should be noted.
- It has an auto-iris adapter available.
- It has an electronic shutter (1/1000 sec).
- It has a C-Mount adapter available.

In addition, lenses have to be considered. Lenses are required for three specific purposes.

- Arm-mounted camera. This requires an extremely close focal point and a depth of view which should be as large as possible.
- Head mounted camera (Where). Wide angle, good depth of field.
- Head mounted camera (What). Narrow angle, depth of field not as crucial.

The 'What' head mounted camera was deemed less important, and we use a 6mm F1.4 lens on the color camera providing a field of view of 58 degrees horizontally and 43 degrees vertically, with focus from about 3.5 feet to infinity.

The arm mounted camera must have a closer focus and so a 3.5mm lens with focus from 9" to 18" was purchased.

2.2.2 Pan-tilt Head

In order to effectively utilize the head mounted camera, a pan/tilt head is essential. Fortunately, cheap, effective pan/tilt heads do exist which are light weight and low power. Three manufacturers were considered; Directed Perception, Pelco and Zebra with the Directed Perception pan/tilt unit being eventually chosen, principally for its simple interface and word of mouth evaluation. In practice, the unit presented initial problems due to poor cabling. These problems have never been fully resolved.

The Directed Perception unit is a computer controlled pan-tilt unit which weighs 1 kg and can support about 3 lbs camera payload. It provides 200 deg/sec slew. The range can be specified with in the range of (Tilt) 78 degrees to 111 degrees and (Pan) 280 degrees to 340 degrees, *i.e.* it can't turn around totally. The unit promises between 3 to 4 degree accuracy in all directions. Power consumption varies from 16W-7.5W in motion to just 1W when holding.

Most importantly, the unit can be communicated with over an RS-232 interface. A driver resides within CTV (see section 4.2.3) which implements the command protocol.

2.2.3 Frame-grabber card

A frame-grabber card is required to interface the camera with our pentium pro machines. Requirements considered were quality of image and speed of acquisition. The board we have isa Matrox Meteor, with the highest quality image one can get at standard video resolution. The next step up is the super-high resolution boards that some scientific instrumentation use with on the order of 10K pixels per line and special cameras.

The frame-grabber uses NTSC input or RGB, and can multiplex up to 4 inputs. It has been discovered that the Natoma P6 chipset, which Xavier's pentium pro's use, will lockup when a meteor is run in the YUV_PLANAR mode at any speed and resolution. The solution is to use the YUV_PACKED mode instead.

CTV contains a driver for the card so CTV must be run on the machine with the frame-grabber (at this time, **Ariel**).

2.2.4 Adding additional cameras

Up to 4 cameras can be connected to the frame-grabber. Currently, the color camera uses the "red" cable. The "blue" and "green" cables can be immediately connected up to a monochrome or color cameras and TCA commands have been provided to switch between them.

2.3 Motion

The B24 robot base is capable of two types of movements: rotation and translation. They can be performed simultaneously, allowing a curve movement. And they can be modified on the fly, allowing changes in the velocity and direction of the robot's movement without stopping the motion.

The correct use of the commands can make the robot move in fancy figures. Some examples are shown in the RWI manual, where there are sample programs to make the robot to do a square, a circle, a rounded square and a 4 point start.

The base is currently set up to constantly returns information relating to current encoder positions, the status of bump sensors, and battery voltage level.

The base is also set up at 32% of its maximum torque. This limits its speed of travel but prevents it from crushing doorways.

2.3.1 Batteries

The base contains 4 batteries, totalling 500W. This has been found in practice to be sufficient for 2 to 3 hrs of heavy usage. It may then take 10 hrs to fully re-charge them. The batteries should be charging anytime Xavier is stationary. The charger is easily attached to the red connector on the base, it is behind the small hinged door on the bump panel.

2.3.2 Dead reckoning

A dead reckoning function is executed each time a status report from the base occurs. It estimates the current base position from the current velocity (read from the base) and from the time difference between the previous status reading and the current one. The dead reckoning has been optimized for "on-carpet" usage. If Xavier is to be used for an extended period of time in different terrains (eg. a tiled corridor), it may be appropriate to generate specific dead reckoning heuristics.

2.4 Verbal Communication

Xavier is capable of both generating speech and understanding spoken communication.

2.4.1 Speech Generation

An inexpensive speech board from RC Systems, the V8601, is on **Wolverine** currently. It possesses a custom algorithm for converting ASCII English to a recognizable accented speech, and also provides the (unsupported) capability to use any CMU software speech programs that are developed with a "PCM" playback format.

Operation It has four operating modes (and a standby mode) which determine how the data is processed by the internal speech algorithms. These modes are "Text, Character, Phoneme, PCM" described below. The operating mode can be changed at any time, even within the same string of text. Thus whilst speaking we can alter the system used for synthesis.

Text Mode: All text sent to the V8601 is spoken as complete sentences. Punctuation is also taken into consideration to give natural intonation (asking questions, etc).

Character Mode: This spells out words as they are received.

Phoneme Mode: This disables the text-to-speech translator, allowing phonemes to be directly accessed. This means that incorrectly pronounced words can be fine-tuned.

PCM Mode: Data is directly written to the digital to analog converter, producing the highest quality speech (digitized sound), allowing any improved software versions developed at CMU to be used.

Standby Mode: The V8601 enters a low-power standby state when no data is being written. Its status can nonetheless be ascertained in this mode.

The pitch, frequency and volume can be adjusted at any stage. This allows male/female voices to be mimicked and for the board to actually sing. Three programmable tone generators are available which can easily be used for creating music and sound effects.

In operation, the card is plugged into a half-slot on an ISA bus and is then controlled by sending ASCII characters to a specific location in memory until a null character or a carriage return is reached. This seemed simple and foolproof, reducing the complexity of the device driver. Nonetheless, at present only the text mode is fully operational.

2.4.2 Natural Language Recognition

It was acknowledged at design time that the leaders in this field are here at CMU with Sphinx II. A problem existed at that time with everything being Next specific with no i*86 platform support.

Since then, the Navigator project has ported the technology to Mach 3.0 based *JOSEPH: need TM & citations?* i386s. A major catch turned out to be that the implementation was tremendously CPU intensive. Since Sphinx II is entirely software based, everything runs thorough the processor. This was found in the Navigator to slow the system every time someone spoke. A system was required whereby we have a processor dedicated to speech (admittedly, possibly only while someone is talking).

The current approach, but not totally in keeping with the autonomous theme, was to borrow a Next computer and use the current state of the art Sphinx system. This permits a 1000 word speaker independent vocabulary with minimal development time and training.

Work is currently in progress to port Sphinx II into a Linux environment, and thus onto the robot.

A controller program is written for the Next which parses input and then communicates with Xavier through TCA. Instructions for compiling new versions are on Odysseus, Xavier's offboard Next.

2.4.2.1 Performing Speech Recognition

To perform speech recognition, log onto Odysseus.

*** start Central program: /Apps/Manager.app:

Either double-click on the wave-symbol on the right, or start this program directly. After some time, the manager window will show up.

*** start Robot application:

Click once on the robot icon (in the manager window). After some time, the robot application asks you for a TCA hostname. If you leave this question open, the system will startup in stand-alone mode, otherwise it will connect to the TCA process on the machine name you give. The TCA process must be established by that time.

*** Software changes:

Everything robot-specific is in:

`AST_Robot/src/Robot/` (linked to `/Robot` for impatient people).

Important files:

- SpeechListener.m main program, in C Every change in programming is done here.
- Robot.bnf, Robot2.bnf are robot grammars in Backus-Naur-Form.
- grammars.h contains a list of valid grammar names. It also contains a short list what to do in order to define a new grammar.

After changing anything, the new robot program can be installed by:

1. Click the robot mouse button on the right side. Click "quit" on the menu which shows up on the left.
2. Call "make install" on Odysseus - don't forget install
3. Click the robot icon in the manager window - a message will ask whether to disconnect, click yes.
4. Click the robot icon again, which causes the manager to reload the robot application.
5. Give the name of a machine running central, or nothing (stand-alone, see above)

2.5 Computing Power

Xavier was 486 based, in keeping with the intention to use cheap off the shelf components. Early in 1997, the main 486's were upgraded to Pentium Pros, to keep Xavier up to date with current technology.

2.5.1 486 machines

The machines in the Xavier group are :

Charles : Offboard, available for dependable compilation.

Wolverine : In Xavier, robot control, sensors.

Ariel : In Xavier, vision stuff.

Gambit : Lap-top with color display, controls wireless communication.

2.5.1.1 Raw speed

Two different motherboards are used on Xavier - 2 MicroStar motherboards onboard and a slower SI2 board on **Charles**. In compute bound tasks, **Charles** scores 111K Dhrystones, whereas **Ariel** and **Wolverine** achieve 49.7K[1].

[1] It is interesting to put the capabilities of the original onboard system into perspective. The official figures from the "Intel486(TM) DX2 Microprocessor Performance Brief" (order number: 241254-001 from Intel Literature) lists the Dhrystone 1.1 Performance of the INTEL486 DX2-66MHZ CPU as 94786 Dhrystone/sec or 54 Dhrystone 1.1 MIPS (page 16). Furthermore, the performance of our system is 4 times faster than a 386 based machine, 97 times faster than a IBM/PC AT, 4 times faster than a Vax, and just over half the speed of a Sparc ELC.

2.5.1.2 Card insertion

Due to the placement of the motherboards and cages, card insertion and removal is tricky. Unscrew the sonar ring and lift it as high as possible. This should allow access to the screws on the cards. They may then be unscrewed and removed as normal. Also ensure that cards are firmly seated and securely screwed in during operation.

2.5.2 Operating System

Mach OS was used on Xavier originally due to familiarity and in-department expertise. Facilities installed the standard Mach 2.5 configuration, and we then amended the kernel for optimized serial drivers and frame-grabber, laser, and speech device hacks. **Gambit** used Mach 3.0 due to specific LCD screen and PCMCIA driver requirements.

The Linux operating system is now used on Xavier.

2.5.3 Wireless communication

Being a mobile machine, we use two means of communicating with Xavier on the run. One means is with a wireless Arlan, which corresponds to an invisible network connection at 1M, the alternative is using a 9600baud radio modem and logging in over the serial port.

2.6 Assembly/Disassembly

Hopefully, you will never have to reduce Xavier to his component parts. However, it is often necessary to remove the enclosure from the base. To do this, remove the skin (see below), to allow access into the enclosure. Turn off main power both at the white top-panel switch next to the laptop, and at the breaker marked "Enclosure" next to the charger connection on the base. In the center of the bottom of the enclosure, remove the ribbon cable connecting the base to the enclosure. The two screws in line with the connector do not hold the enclosure to the base, and should not be removed. One of the functions they provide is protection for the plastic connector. The other four screws, the flat-head hex screws, can be removed. Once that is done, two people can remove the enclosure by gripping the uppermost plate and lifting straight up before setting it aside. When replacing the enclosure, ensure that the notch on the plastic connector on the base (mentioned above, for the ribbon cable) is facing the front of the enclosure. Then reverse the process of removal.

2.6.1 Skin

Xavier's skin is in nine parts, not including the bump panels on the base. The three largest pieces cover the main body, and are suspended from the sonar ring with velcro. Two of these pieces have cuts in them to allow for access to the lasers and associated switches. These belong in front. Three small pieces are braces for the upper skin. They are attached between the sonar ring and the enclosure top using velcro. One goes to the side of the laser-rangefinder mount, the other two at 4 and 8 o'clock around the robot. The last three pieces of the skin attach to these braces, again using velcro. The smaller of these, with a fan, goes above wolverine. There is a small mini-phono jack mounted onto the support post that the fan plugs into. The other piece with the fan goes over Ariel, again, plug the fan into the jack mounted on the support post. The last piece goes in front of the sonar and pan-tilt controllers.

2.6.2 Wiring

In the summer of 1995, Xavier's power distribution was replaced by a new system developed by RWI for their B21 robot systems. This resulted in more reliable power with less wiring throughout the robot. There are now panel switches on the upper body of the robot which allow for partial shut-down of the robot, as well as a covered switch for main enclosure power. Programmable switches are also available.

2.7 Problems

In the event of a serious problem arising, mail to "robocomp@cs" will be answered in prompt fashion. This may be necessary in the event of machine crash, power outages or such problems.

2.8 Quick Troubleshooting

If you don't have the time to read the full manual, we try to provide in best kitchen equipment style likely causes and fixes to some common problems.

Symptoms/Hints

1. **Low grinding noise:** Adjust drive belts - they may be too tight. Take off turret, and loosen the belts so that they are tight, yet no grinding occurs. Read the RWI B12 manual for advice.
2. **Shakes a lot:** A wheel has fallen off/come loose. Check wheels. Drive belt snagging internally. See "Low Grinding Noise."
3. **Refuses to rotate** Check rotate belts in base. Or, six hex screws through the spindle gear need to be tightened. These must be VERY tight!
4. **Moves fail:** Kill Switch may be on. Reset by pressing red button on the base. Base may be switched off. Switch on, and make sure that green light appears. Rotational velocity set too low. Adjust this in CTR. Drive belt broken. Spare belts are available, replace, and see "Low Grinding Noise". Base connector working? Test by trying to talk to the base.
5. **Won't speak:** Volume off. The volume control is on the speech card on Ariel. Twiddle to the required level. Someone tried to make the card sing. This isn't supported and the card must be reset by rebooting the machine.
6. **Pan-Tilt unit spins:** This is usually an indication that the batteries are running low. Reconnect the charger.
7. **Machines reset themselves:** Power spike or Power low. Connect to the charger. If problem persists, there may be a loose power connector.
8. **Slows down:** Disk space filling up. This commonly occurs if the paging file grows too large. Either clear up disk space or reboot. Power running low. Connect to AC and recharge. Obstacle noise slowing potential field - see "Spurious obstacles" or reduce the potential of obstacles.
9. **Sonar acts poorly:** Interleave rate for sonars altered (check in sonar_interface.c) ? All switched on (check output from CTR)? Sonars switched on? Check on/off switch on top panel. They may have gotten out of sync. Reset using on/off switch.
10. **Spurious obstacles appear:** :Laser only, glare from ceiling lights is appearing in the camera- add hood. Alternatively, rerun CTR so that a new threshold is generated. Sonar only, faulty sonar - check that all are working in CTR.
11. **Central fails to run:** Someone else running it? Did not set centralhost?
12. **Can't communicate with onboard machines:** Ethernet OK? machines on? pingable?
13. **Can't log into onboard machines:** Have account?
14. **Sonars aren't clicking:** See "Sonar acts poorly". Make sure that all the sonars are connected by wire to the correct transducer board.
15. **No lasers come on:** Switches near lasers must be "up". Make sure laser connector plugged into card.
16. **Only one laser:** Both switches up? Run CTR.
17. **Pan-Tilt head stops moving:** Reset pan-tilt manually - due to noise on wire.
18. **Machines won't reboot:** May have to run `fschk` manually. Connect monitor and keyboard. Answer yes to all questions. Alternatively, switch disk drives by swapping the IDE cables. This will result with **Wolverine** rebooting as **Ariel** and vice versa. `fschk` can now be run and state returned to normal.
19. **Machine complains of hard disk controller error** Disk cable plugged in upside down? The side marked "up" on the cable should be visible.

20. **Machine runs slowly** (This is most relevant when a new machine is purchased). Check on-board jumpers and oscillator. A 66 MHz oscillator should be present with the jumpers are set correctly. Played with the CMOS settings a little... the bus speed can be adjusted (that is, 8 MHz may be recommended, but it is possible to set it to 11, 13, and some other settings in-between). This may involve also changing memory wait states around a little.

Chapter 3

Simulation Environment

The simulator has evolved from the days of the HERO-2000 robot and has recently been substantially improved in terms of responsiveness and speed. This chapter assumes that the CMU CS environment is available. The simulator and TCA have been tested for DecStation 2100/3100/500, Dec Alphas, Intel 468 machines, Sun 4 series and Sparc 10.

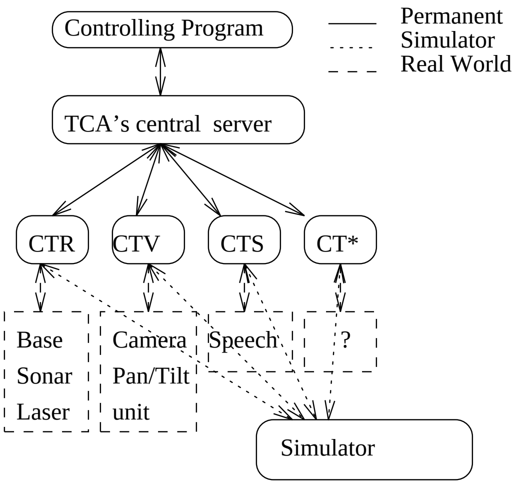

Figure 3.1: **Schema of Agent Architecture**. The bold lines are the "permanent interconnections." The dashed and dotted lines indicate that the simulator replaces the hardware devices exactly and transparently to the controlling program.

The simulator duplicates the devices on Xavier (see figure 3.1). The testing of application code on the simulator is exactly as it would be on Xavier. While the simulator attempts to provide a close approximation to the real world, deviations do occur. We shall first describe how to run the simulator, what special features are available in the simulator, and then how the devices are modeled and where the modelling fails to capture the real world.

3.1 Starting up

As has been seen in figure 3.1, the simulator provides the same interface as the real robot. As such, nearly equivalent software procedures are performed as with Xavier.

We must first set the central server running. Then we initialize the simulator in a particular environment. We can then run all necessary devices and finally communicate with the simulator through these devices with a controlling program.

```
Window 1:
 machine1%setenv CENTRALHOST <machine1>
 machine1%cd /afs/cs/project/robocomp/tca/bin
 machine1%central -lmstd 1

Window 2:
 machine2%setenv CENTRALHOST <machine1>
 machine2%cd /afs/cs/project/robocomp/xavier/bin
 machine2%xavier ../params/event1.param &
              [start with some param file]

Window 3:
 machine3%setenv CENTRALHOST <machine1>
 machine3%cd /afs/cs/project/robocomp/xavier/bin
 machine3%CTR -s <machine2>
```

We advise these to be run in different windows. Each process directs error messages and other information to the standard output, so for your own sanity it is more convenient to separate the displays. The processes can be on the same machine or distributed over different machines. The simulator, and controllers must all be run on the same architecture. Note that the machine names are described simply in terms of the Ethernet address. Do not append ":0" to the names! Also the ordering is important. The central server is started first, then xavier (the simulator), then all the required devices.

The devices available are in detail in chapter 4. Briefly:

CTR: Base, sonars and laser.
CTS: Speech.
CTV: Vision.
CTA: Arm.
CTRECORD: Record sensations.
CTREPLAY: Replay previously recorded sensations.

CTR may take a few seconds to start up. It will place a window with empty buttons on screen which are slowly filled in.

Error messages may be reported from CTR. This will be due to CTR testing for parameter files associated with the real hardware. These error messages should be ignored.

Occasionally, the socket used by the simulator for communication will be in use. (This will especially occur if multiple simulators are in action on the same machine). In this situation, a port specifier is provided, which allows a different socket to be used. The "-p" option should be provided to the simulator and all devices. The number given to "-p" is any random number, such that it does not appear in the fourth column of the output when you run "`netstat`". The same number should be given to all devices.

3.2 The Central Server

The TCA central server provides the interface between a controlling program and the robot's hardware. Sections 4.1 and 4.2 discuss this interface in more detail.

It *must* be started before the simulator, the devices, and any controlling programs.

The command format for `central` follows. `central` without parameters will print a short help file.

```
central [expected-modules]
        -l[options]
        -L[options]
        -f[filename]
```

expected-modules minimum number of modules that will be connected.

-l: logging onto terminal.
options: **mstdihrpx**
- m show message traffic
- s show status of TCA
- t time messages are handled
- d data associated with message
- i ignore logging registration and deregistration messages
- h summarize incoming message handle time
- r log the reference Id as well as the message name
- p log the reference Id of the message's parent
- x no logging

-l is equivalent to **-lmstdh**
default: **-lmsi**

-L: logging into file.
options: mstdihrpxF
same as above, plus
- F don't flush file after each line

default: **-Lx**

-f: filename to use for logging. If not specified, name is automatically generated.

-v: display server version info.

-r: try resending non-completed messages when modules crash

-p: specify the server port. The compiled port is: 1381

-d: flag random debugging.

-q: test quantity lattice insertions for task tree nodes.

-s: silent running, don't output to stdout.

`central` will report an error if a copy is previously running in the background. The error reported will be:

```
ERROR: serverInitialize
: Address already in use
```

Either kill the old process, or use the old process as server.

3.3 Look and Feel - the X Window environment

Following the above setup, two windows will appear on screen. One window will be the device window, in this case CTR. This will be discussed in the individual device section. The other window is the simulator display (Figure 3.2).

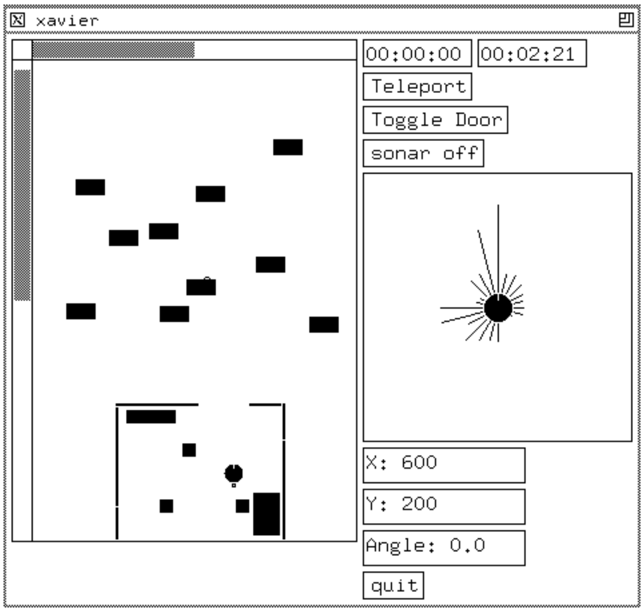

Figure 3.2: **Sample simulator display** This is the world in event1.param.

The simulator window scales itself to the size of the environment, with there being a fixed maximum size. If the environment is larger than this size, the whole world may be viewed using the scrollbars in the display. The simulator window also places itself on the screen. You can control its placement using the geometry resource in your `.Xdefaults` file.

In the top left hand corner of the display is a scrollable view of the world specified in the parameter file. In this instance (namely, `event1.param`), Xavier is in a small room inside a larger room, with obstacles scattered around. There are 3 doors in the smaller room, one of which is open. By clicking on "Toggle Door" and then clicking on a door, the state of the door way can be changed.

Xavier is located at X = 600, Y = 200. All measurements are assumed to be in centimeters. These x,y, and angle coordinates are in a reference frame specified by the simulator parameter file. The simulator will display actual world coordinates, while Xavier's dead reckoning senses will report the believed position; initially this is (0,0).

Xavier can be interactively moved to an arbitrary location. Press the teleport button and then press the mouse button when the pointer is over the location in the environment bitmap in which Xavier is to be teleported. If you can't move the robot there because it would overlap with an obstacle or it wouldn't be in freespace, an appropriate error message is output. The teleport button remains black while waiting for you to select a place to teleport. This lets you know what "mode" you are in.

The sonar displayed is scaled such that its limit is about 25 feet . When the sonar beams and robot are scaled to fit within the sonar widget, the robot becomes only a few pixels wide. This has been solved by the sonar beams being scaled correctly but the display of the robot being at three times its correct size.

Two times are maintained. One is the real world time, the other is the time in the environment. By adjusting the parameter file, the environment can be made to run faster/slower than real time.

The quit button available in the simulator is the recommended means of closing the simulator. Shut down CTR first. Alternatives risk upsetting the socket connection. If a crash occurs, it may be necessary to wait for approximately a minute to allow UNIX to reset the socket address.

If you find that sockets are often left in the system, you may find the following command useful:

```
/afs/cs/usr/dkindred/bin/killit RFS\ 128
```

`killit` kills all processes matching the pattern in the arguments. You might want to do a `ps -aux` to see what the best pattern for you is.

3.4 Modeling of Xavier's hardware devices

An important feature of the simulator is that it mimics devices at the hardware level. Interaction at all but the very lowest level is identical whether or not the simulator is in use.

3.4.1 Range-finding

All range-finding devices are supported. Each are mimicked at the speed of the hardware. Thus the sonars are updated 3 times per second, the lasers 10 times per second and the bump panels are constantly available. The sonars and lasers are implemented by ray-tracing in the environment. Gaussian errors are added to the readings as specified in the current parameter file.

3.4.2 Motion

The simulated base provides dead-reckoning information just like it would in the real world. In addition, errors may introduced (by setting the parameter file), which model basic slippage of wheels. By default it is assumed that Xavier is accurate to within 5% in all actions.

3.4.3 Power

Testing the base power in the simulator will always return full power. An enhancement for a future date is to model the power loss, so that a long term simulated agent need also be concerned about charge.

3.4.4 Vision

Basic vision stubs are not implemented in the simulator. Providing some type of ray-tracing capabilities seem unnecessary. However, the simulator does allow markers to be searched in the environment - first grab an image using ACQUIRE_IMAGE_COMMAND, then apply FIND_MARKERS_QUERY to the image (see section 4.2.3). The pan/tilt head may also be rotated and tilted to scan the full environment. As such the x, y and z locations of the marker are crucial. It is assumed that if the marker location can be reached with a sonar, the marker can be read (that is, it is in line of sight!).

3.4.5 Speech

Attempting to produce speech on random workstations was not attempted. The speech output commands are simulated by printing output to the screen.

3.5 Extending the simulator

Working is currently ongoing into improving the simulator. Comments and opinions are always appreciated and may be mailed to "robocomp-simulator@cs".

3.6 Brave new worlds

The simulation environment is fairly amenable to alteration. New worlds can be developed, perception noise can be changed (based on a Gaussian model of noise), and speed of response can be varied (note that since many controlling programs expect Xavier to be operating at a real world speed, faster-than-real-time is not a good default).

3.7 Redefining World Specification

The simulator acts upon a world environment which is defined in a .param file. While several such worlds are provided and there is an ongoing process to describe Wean Hall in a simulator readable fashion, it may frequently be required to construct addition settings. The best way to begin creating a new world is by modifying one of the many predefined environments. A simple language is used to construct these files. Each file contains lines with commands of the following forms (note that all dimensions are in centimeters and that each command is one line with no line breaks - these have been introduced here for clarity):

#

```
# (followed by space)
```

This allows for comments. All of the line containing a # is treated as a comment.

door

```
door <ul-x> <ul-y>
     <lr-x> <lr-y>
     <close/open>
```

Makes a door. For example, with wall thickness = 1, then neighboring rooms have gaps of 1 possibly occupied by a door. Doors can be initially closed or open and the state of doors can be changed during simulation runs, using the "ToggleDoor" button.

environment

```
environment  <name> <scale-factor>
```

This construct controls display parameters. The name of environment will be used as the name of the simulator window, with the scale-factor being the number of pixels per 1 unit of simulated space.

goal

```
goal <x> <y>
```

A goal is placed at x,y. x,y must be in a previously defined room/corridor. Once the robot reaches the goal during simulation, the goal is removed from the environment. (This was developed to model food which is eaten when robot finds it.)

marker `marker <mark> <x> <y> <z> <attachedTo>`

As stated above, marker recognition is supported in the simulator. "Mark" may be one of +,o,x, and "attachedTo" is one of door, room, corridor, obst.

obst

```
obst <shape> <status>
     shape==rectangle   <ul-x> <ul-y>
                        <lr-x> <lr-y>
     shape==circle <mid-x> <mid-y> <diam>
```

Obstacles can be picked up with the arm. Reverse the robot to an obstacle and issue a grasp command. The obstacle will be removed from the simulator.

robot

```
robot  <position_x> <position_y>
       <orientation> <radius>
```

This specifies robot's initial position and orientation. The (x,y) specifying position should be in a room earlier defined. The orientation is specified currently in radians. `Radius` corresponds to the robot's radius.

room

```
room <ul-x> <ul-y>
     <lr-x> <lr-y>
```

Make room with specified dimensions. Coordinates are defined such that (-oo,-oo) is upper left, (+oo,+oo) is lower right.

units `units_per_cm <p>` (or) `units_per_ft <p>`

Specifies the conversion rate between the units used in the file and the robot size and speed.

3.7.1 Redefining Xavier Specification

In all the cases described below, noise is modeled as being Gaussian. Most of these parameters should be used unaltered, with noise adjustment and speed being the most likely candidates for variation.

forward_noise_level

```
forward_noise_level <p>
```

real forward = asked forward + $p * [-1, 1]$.

laser

```
laser <resolution> <fov> <height>
      <distance> <yaw> <pitch>
      <noise level>
```

The resolution corresponds to the maximum possible number of bits of information. The laser is assumed to be adjusted such that the field of view is fov, the height above the base platform being $height$, and the laser pointed at the given yaw and $pitch$. The $noise$ is generally set to an extremely small value due to the accuracy of the device. $height$ and $pitch$ are currently ignored.

nodisplay

```
nodisplay
```

If this is set, no graphics will be displayed. It should be generally ignored in parameter files unless speed is critical and the controlling program is providing feedback.

refresh_rate

```
refresh_rate <n>
```

Graphical screen refresh occurs every n simulation steps.

sensors

```
sensors  <#number> <measuremts_per_sensor>
         <distance> <cone> <display_mode>
```

#number is the number of sensor (sonar) measurements
in one 360 degree rotation. Each sensor measurement is calculated from `measurements_per_sensor` samples near the angle of the sensor measurement. For example, if #number = 2 (at 0 and 180) and `measurements_per_sensor` = 2, then sensor result at 0 is the min of that at -45 degrees and 45 degrees. The distance specifies the maximum distance, the cone specifies the field of view in degrees and the display_mode specified whether the widget produced will be robot_centered (1) vs world_centered (0).

sensor_noise_level

```
sensor_noise_level <p>
```

reading = real + $p * [-1, 1]$.

sensor_zero

`sensor_zero <degrees>`
Actual angle corresponding to the zero-degree reading. Defaults to 0.

time_increment

`time_increment <milliseconds>`
Maximum time between simulator updates.

time_scale

`time_scale <milliseconds>`
Each simulated second is this many CPU timer milliseconds.

turning_noise_level

`turning_noise_level <p>`
real turn = asked turn in degrees + $p * [-1, 1]$.

Chapter 4

Basic Robot Programming

This chapter describes everything required to develop a high level control module for Xavier. A high level module is a program that "sits on top" of Xavier and assumes command. Such a program could just constantly monitor sensations and thus build a map or perhaps steer Xavier through a maze. This section will not deal with combining multiple modules, acquiring resources to remove competition and deadlock, developing low-level device dependent modules, or some of the more arcane (but highly useful!) features of TCA such as exception handling. These will all be dealt with in the Advanced Robot Programming chapter or can be discovered in the TCA manual.

First, we introduce the task control architecture – TCA – the underlying mechanism allowing control to be distributed across Xavier's multiple processors. TCA, when requested, routes messages and data between these multiple processes, roughly implementing a remote procedure calling functionality. We describe such messages and data structures as implemented in Xavier.

To ground this information, simple application modules are presented which cause Xavier to perform simple tasks. Using these modules, the simulator will be presented and used to test how well this module works in a "clean" world. Then, we'll step through the running of the equivalent module on Xavier itself.

4.1 TCA in the context of Xavier

The Task Control Architecture[1] (TCA) simplifies building control systems for mobile robots such as Xavier. It allows you to construct a distributed system without having to build your own remote procedure call mechanism. It provides orderly access to robot resources so that you don't have to build your own scheduling mechanism. On Xavier, these features are especially important given that processing power is distributed across two machines, **Wolverine** and **Ariel**, each having their own unique hardware capabilities.

Using this base of control, the user can then construct *deliberative* behaviour for Xavier. The TCA manual describes tools allowing you to build *task trees*, which specify hierarchical plans of action. The order of planning and execution can be controlled by applying temporal constraints to task tree nodes. Constraints can also be incrementally relaxed to take advantage of concurrency wherever possible.

Tools also exist for incrementally adding *reactive* behaviours to the control system without disturbing the basic, deliberative algorithms. Monitors test for some condition – often periodically over some time interval – and specify appropriate action when the condition holds. Exception handlers are invoked when something has interfered with the robot's plans; error recovery strategies are implemented by manipulating the task tree to "re-plan" robot behaviour. Wiretaps provide the ability to listen for particular messages in the system. You can specify that some action be taken each time a certain message is sent.

By providing separate mechanisms for deliberative and reactive behaviours, TCA facilitates incremental development. Building deliberative behaviours and then adding reactive behaviours as needed, is particularly useful given that Xavier may be required to operate in dynamic or unpredictable environments, because it is impossible to anticipate all possible problems that may be encountered. This is one of TCA's key features: separating the "normal" and "abnormal" algorithms of a control system for Xavier.

4.1.1 Capabilities

The following are key capabilities of autonomous robots such as Xavier; the Task Control Architecture provides support for all of them.

Achieving Goals

The basic need of a robot is to construct and execute plans to achieve given goals, such as navigating to a particular location or collecting a desired sample. A robot must construct plans based on current (or projected) environment, available resources, and its other goals and beliefs. It should be able to execute parts of its plans before specifying them completely – for example, orienting Xavier to pick up a box before determining an exact point to place the arm.

[1] This introduction derives from the TCA manual, which should be referred to if more detail is required.

Reacting to Environmental Changes

If Xavier's battery is low, or its movement uncovers interesting objects, these changes should be noticed in a timely manner and should be attended as necessary. To handle such changes, Xavier may have to add new subgoals (e.g. examine and perhaps pick up the new object), suspend current tasks (e.g. stop picking up the box and recharge), or replan current goals.

Recovering from Errors

One type of reactive behaviour is noticing when a plan of action is failing. The robot should have general mechanisms for recovering from both execution and plan time errors. For example, if an obstacle appears in the robot's pat, it might want to detour around the object or plan a new path to its goal. Similarly, if the robot planner is unable to find a clear path, it may try re-planning with less restrictive constraints (e.g. allowing more tolerance), or may attend to a different goal altogether.

Coordinating Multiple Tasks

For autonomous robots with many goals but limited resources, it is crucial to prioritize and schedule its various tasks. The robots should make decisions about which goals to attend to by comparing their relative costs, benefits, likelihoods of success, etc. For example, if Xavier's batteries are getting low, it might decide to stop what it's doing and recharge. However, if it estimates that it can finish its current task in the next few minutes, it might decide to continue and then recharge.

Communicating with External Sources

No robot should be fully autonomous; interaction with humans is a necessity. The robot should be able to explain its decisions and actions. It also should request assistance when needed and allow people to add goals and to alter plans and decisions made by the system. This is especially important for robots, such as planetary rovers, that experience a significant communication delay. To this end, the architecture allows humans to input information on any level of the planning and control hierarchy, including viewing and manipulating autonomously created plans.

4.1.2 Limitations

TCA is not an appropriate framework for real-time control systems, It is not intended for robots requiring fast (millisecond) reflexes. On Xavier, subsystems such as the sonar and laser requiring fast response are catered for in hardware, with localized control routines providing buffering to allow TCA management.

TCA provides a general control framework, and it is intended to be used to control a wide variety of robots. Thus, TCA has no built-in control functions for *particular* robots. That is, TCA itself has no code specific to Xavier. We shall next describe code *using* TCA, which *is* specific to Xavier.

4.2 Xavier Control

TCA on Xavier provides a means for a high level program to communicate to a number of low-level modules distributed over the system. Specifically, a high-level program can communicate with:

CTR: Controls the base, moving the robot, performing local navigation, and responding to queries relating to battery life, sonar/laser/bump panel readings or dead reckoning location.

CTV: Controls the pan-tilt head and up to three cameras. The pan-tilt head can be moved/queried, pictures can be taken/saved to file, or a marker (a plus sign) can be searched for in a given picture.

CTS: Controls Xavier's speech module, performing English to sound mapping in hardware. It also has un-debugged capabilities for playing music (once!) and for replaying recorded sounds.

CTA: Controls Xavier's detachable arm, either performing a raising or a lowering action.

CTRECORD: Can record selected/all sensations and actions to a data file.

CTREPLAY: Can replay these sensations/actions for off-line investigation.

In this section, we shall provide a comprehensive listing of the commands available to the high level program, grouped according to the module wherein they are implemented. We present the commands in terms of C TCA calls. It is also possible to use Lisp.

TCA allows for two basic means of scheduling any call; either for blocking notification, with the TCA command not returning until execution has completed, or for nonblocking notification, with the TCA command returning directly after the TCA server has been notified. The execution of the command then completes in parallel with the controlling module, and there exists TCA constructs to query the success and return values of such instructions.

In the following descriptions, we assume that for the most part blocking notification is required for TCA queries (commands that interrogate Xavier) and the TCA commands for Xavier can either be blocking or nonblocking depending on context. The appropriate constructs are:

tcaWaitForCommand(): Executes a command, waiting for completion. Returns *Success* if the command completed successfully as defined by the module, otherwise returns *Failure*. Example usage:

```
status=tcaWaitForCommand(SOME_COMMAND,
                         &someData);
if (status == Success)
    DoSomeThingForSuccess();
else /* Failure returned */
    DoSomeThingForFailure();
```

tcaExecuteCommand(): This routine is non-blocking. Success means that the request was successfully sent to the central server. It does not mean the the request has been handled. Example usage:

```
tcaExecuteCommand(SOME_COMMAND,
                  &someData);
```

tcaQuery(): Sends a message to the central server, which then routes it to the relevant module. This call blocks until the message is handled and reply data produced. Typically, the return value of a call to `tcaQuery()` will be *Success*. If the servicing module decides that the query should fail, tcaQuery() returns the value *NullReply* and the value of the reply data is undefined. Example usage:

```
status=tcaQuery(SOME_QUERY,
                &someQueryData,
                &someReplyData);
if (status != NullReply)
    print_data(someReplyData);
else /* Failure returned */
    DoSomeThingForFailure();
```

Section 3.2 discusses how to start the TCA central server running.

4.2.1 CTR – the Base Module

CTR has specific control over the base involving the movement of XAVIER, the performance of local navigation (reactively avoiding obstacles), and the response to queries relating to battery life, sonar/laser/bump panel readings or dead reckoning location.

When any of the CTR commands or queries are used, CTR must be running. If the real hardware is being used, CTR must be run on **Wolverine**, the computer carrying the laser card. A copy of CTR is available in:

/afs/cs/project/robocomp/bin

(as are copies of all programs referenced).

```
machine% setenv CENTRALHOST machinewithcentral
machine% CTR
>
```

Currently, CTR expects only the following options:

-x = No X11 window will be displayed. CTR will communicate just using the keyboard.

-t = No attempt is made to connect to TCA. This is useful when a robot is to be "joysticked" by the user without a higher controlling program being involved.

-s machine = Attempts to connect to a simulator on this machine

-p num = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

-r = Does not attempt to connect to the base. Actions commands are ignored.

+r dev = Connects to the base using a different port. The default is "/dev/com0"

-l = Does not attempt to start the laser. Laser readings return no objects.

-n = Does not attempt to start the sonar. Sonar readings return blank space.

+n dev = Connects to the sonar using a different port. The default is "/dev/com1"

If CTR is connecting to the hardware, it will read in configuration files which should be in the same directory (see Chapter 2).

If the **-x** option is not set, a X11 screen will appear, allowing graphical control of the robot (see Figure 4.1). This screen is duplicated by the keyboard prompt, which also provides for controlling the base. (The keyboard also has additional commands for going to specific goals and directions).

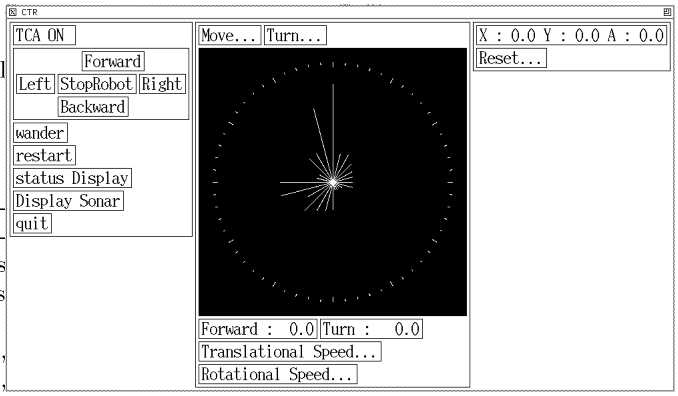

Figure 4.1: **CTR X11 Display** A graphical interface for controlling and monitoring the base. This display is currently in the process of being improved.

Clicking on the "Display Sonar" button causes the sonar display to appear, displaying the results of 24 sonar readings, plotted from the center of the robot (see Figure 4.1). This sonar display window can be used for controlling the robot. Pressing and holding the mouse in the black area will cause a line to appear. Xavier will translate and rotate depending on the angle and length of the line. A long vertical line upwards means that Xavier will go forward quickly without turning. A long horizontal line to the right means that Xavier will turn fast clockwise without any rotation. (Rapid clicking in this area **will not** cause the robot to move faster).

Clicking on the "status display" button yields a separate more stylized perception view (see Figure 4.2). Sonar readings are explicitly displayed as circles of uncertainty, with laser readings showing up as persistent red pixels.

If the -x option is set, no window is displayed and control is effected using either the keyboard or the CTR-TCA suite.

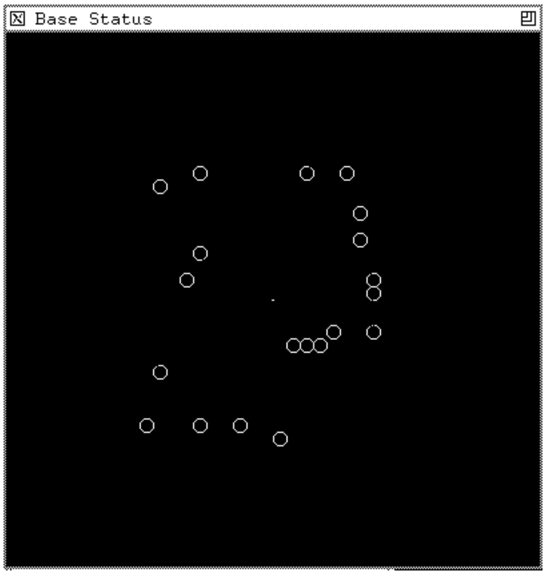

Figure 4.2: **Status Display Window** A more stylized view of current perceptions.

4.2.1.1 Commands

Described below are the following basic commands, in the following order:

- Reset
- Dead reckoning
- Stop
- Teleport (simulator only)

- Move (Translate)
- Set maximum velocity for move
- Set velocity for move
- Set acceleration for move

- Rotate
- Set maximum acceleration for rotation
- Set velocity for rotation

Resetting low level devices.

```
tcaWaitForCommand(CTR_RESET_COMMAND, NULL);
```

During operation, any or all of the low level devices (base, sonar, laser) can come out of sync. For the base, this could occur if the kill switch was depressed, or the base manually servoed using the joystick – communication cannot resume until an initialization sequence is resent. For the laser, entering a radically brighter/darker environment would require recalculation of an appropriate threshold. For the sonar, if it is manually reset during operation, the startup sequence has to be resent.

This message also causes the internal dead-reckoning location to be reset to the origin. If Xavier has already been in operation for a while, it may be appropriate to explicitly change this location (to maintain mapping coherence normally).

Change the internal dead reckoning location.

```
POS_TYPE newPosn;

newPosn.x = ?; newPosn.y = ?;
newPosn.orientation = ?;
tcaWaitForCommand(CTR_SETPOS_COMMAND, &newPosn);
```

As the base moves, a dead reckoning mechanism maintains a rough sense of the robot's location with respect to an origin defined when the `CTR_RESET_COMMAND` was executed or when CTR was started. Over time, drift occurs. The dead-reckoning is quite accurate in terms of distance travelled but less so in terms of orientation. As such, this command allows for the dead-reckoning position to be re calibrated with landmarks in the world.

Stopping the robot

```
tcaWaitForCommand(CTR_STOP_COMMAND, NULL);
```

This command immediately halts rotation and translation. It should always be executed in blocking mode. In the event of failure, a user should be immediately notified by some means. Failure would occur if the base failed to maintain communication with the processing turret.

Teleport robot to arbitrary free location

```
POS_TYPE newPosn;

newPosn.x = ?; newPosn.y = ?;
newPosn.orientation = ?;
tcaWaitForCommand(CTR_HYPERJUMP_COMMAND,
                  &newPosn);
```

During initial testing, it is frequently convenient to move to a specific "problem" location. Not surprisingly, the hyperjump command has no effect outside the simulator. In the simulator, it attempts to move Xavier to the newPosn, relative to the origin. It will fail only if that newPosn is blocked by an obstacle.

Translation Speed	Rotation Speed
30	50
40	60
50	70

Table 4.1: Suggested matched pairs of speeds which allow the Xavier to remain reactive.

Base translation

```
double cm;

tcaWaitForCommand(CTR_MOVE_COMMAND, &cm);
```

This call will attempt to move the base *cm* centimeters forward. If *cm* is negative, the base will move backwards.

During the translation, the local navigation emergency features will be enabled. This means that both before and during the action the handler will poll for obstacles. If an obstacle has already hit Xavier, or if an obstacle would obstruct Xavier before the next possible handler poll (given that the current trajectory, velocity and acceleration would be maintained), Xavier will halt and this command will return *Failure*.

This function will also fail if the emergency stop button has been pressed.

If *cm* is less than the current goal tolerance, this command will prove to be unreliable in the simulator. ALWAYS test to ensure that this condition does not hold beforehand.

Set maximum translation velocity

```
MAX_SPEED_TYPE maxSpeedData;

maxSpeedData->trans_speed = ?;
maxSpeedData->rot_speed   = ?;
tcaExecuteCommand(CTR_SET_MAX_SPEED_COMMAND,
                  &maxSpeedData);
```

Both parameters must be positive. If not, the base will reset to default values. Modified by negative potential due to the presence of obstacles, these speeds are used for the reactive navigation.

For a smooth navigation it is advisable that the rotational speed be greater than translational. The settings in table 4.1 have been empirically noted to work adequately.

Be warned that once the translation velocity is greater than 50, the risk of bumping into objects becomes appreciable, since the current obstacle avoidance polls approximately five times per seconds, not managing well at such speeds.

The speeds will never exceed these programmed velocities unless they already do at the time of this call, in which case Xavier will decelerate to the new velocity. This command may affect some higher level motions already in progress (such as wandering, going to a goal, following paths), but it alone does not cause the base to move. It will not have an effect on current motions such as move and turn, but will effect future motions.

Set current translation velocity

```
double TransVelocity;

tcaExecuteCommand(CTR_TRANS_VELOCITY_COMMAND,
                  &TransVelocity);
```

The velocity value is in centimeters per second. This has a local effect, affecting only the current set of actions. The setMaxSpeed command should be used to change speed global. While this command has effect, the velocity will never exceed this programmed value unless the velocity is already over it, in which case, the translation will decelerate until it is at the new velocity. This command affect motions already in progress, but it alone does not cause the base to move.

The command will cease to be effective on the next wander, follow or goto command.

Set maximum translation acceleration

```
double maxTransAcceleration;

tcaExecuteCommand(
            CTR_TRANS_ACCELERATION_COMMAND,
            &maxTransAcceleration);
```

This command sets the internal translational acceleration variable. The parameter is in centimeters per second squared. This command may affect motions already in progress, but it alone does not cause the base to move. High settings will leaded to jerkiness in movement. Low settings may prevent the base from braking quickly enough to avoid obstacles.

Base rotation

```
double d;

tcaExecuteCommand(CTR_TURN_COMMAND, &d);
```

This call will attempt to turn the base *d* degrees clockwise. If *d* is negative, the base will move anti-clockwise.

During the rotation, the local navigation emergency features will be enabled. This means that both before and during the action the handler will poll for obstacles. If an obstacle has already hit Xavier, or if an obstacle would obstruct Xavier before the next possible handler poll (given that the current trajectory, velocity and acceleration would be maintained), Xavier will halt and this command will return *Failure*. Since Xavier is round, there is no risk of collision due only to the rotation – this event will only occur in conjunction with a concurrent translation action.

This function will also fail if the emergency stop button has been pressed.

If d is less than the current minimum angle tolerance, this command will prove to be unreliable in the simulator. ALWAYS test to ensure that this condition does not hold beforehand.

Set maximum rotation acceleration

```
double maxRotAcceleration;

tcaExecuteCommand(CTR_ROT_ACCELERATION_COMMAND,
                  &maxRotAcceleration);
```

This command sets the internal rotational acceleration variable. The parameter is in degrees per second squared. This command may affect motions already in progress, but it alone does not cause the base to move. High settings will leaded to jerkiness in rotation. Low settings will prevent the base from changing direction sufficiently fast for effective navigation.

Set current rotation velocity

```
double RotVelocity;

tcaExecuteCommand(CTR_ROT_VELOCITY_COMMAND,
                   &RotVelocity);
```

The velocity value is in degrees per second. This has a local effect, affecting only the current set of actions. The setMaxSpeed command should be used to change speed global. While this command has effect, the velocity will never exceed this programmed value unless the velocity is already over it, in which case, the rotation will decelerate until it is at the new velocity. This command affect motions already in progress, but it alone does not cause the base to move.

The command will cease to be effective on the next wander, follow or goto command.

4.2.1.2 Reactive Commands

When Xavier navigates using reactive commands, the potential field is used to compute the next rotational speed whereas the translation speed is computed from the absolute distance to the closest obstacle cell.

Note: Blocking and non-blocking commands that otherwise appear to be the same are differentiated for implementation purposes.

- Goto absolute goal
- Initiate goto absolute goal (non-blocking)
- Goto relative goal
- Initiate goto relative goal (non-blocking)
- Goto multiple, relative goals *JOSEPH blocking or not?*
- Navigate in specific direction
- Concatenate directions
- Change current direction
- Set "goto" accuracy
- Set "goto direction" accuracy
- Random walk
- Timed Random walk
- Fine tune reactive behaviour (set preferred direction for obstacle avoidance).
- Follow Wall
- Follow Path
- Follow Path (non-blocking)

Goto specific absolute goal

```
GOAL_POS_TYPE absGoal;

absGoal.x = (double) ?;
absGoal.y = (double) ?;
absGoal.tolerance = (double) ?;
absGoal.turn_to = (int) ?;
tcaWaitForCommand(CTR_GOTO_GOAL_WORLD_COMMAND,
                  &absGoal);
```

This is intended to be called as a blocking command. Xavier will commence navigating towards *goal*, which is an *absolute* position in the world. A shortest path heuristic is used in conjunction with local navigation. As such, this command is effective for simple goal traversal but not should be abused and expected to navigate complex paths (for example, through mazes!).

`absGoal.turn_to` corresponds to the four cardinal directions (N, S, E, W), but both `absGoal.tolerance` and `absGoal.turn_to` are currently ignored. There is in fact currently no way to specifiy orientation, and tolerance has to be specified with an explict set tolerance command.

On termination, Xavier will either be at the goal (within the goal tolerance or else *Failure* is returned.

Begin going to absolute goal

```
GOAL_POS_TYPE absGoal;

tcaExecuteCommand(CTR_SET_GOAL_WORLD_COMMAND,
                  &absGoal);
```

Xavier will commence navigating towards *goal*, which is an *absolute* position in the world. A shortest path heuristic is used in conjunction with local navigation. As such, this command is effective for simple goal traversal but not should be abused and expected to navigate complex paths (for example, through mazes!).

A constraint message will notify when the goal has been attained successfully (within the goal tolerance). See the example program for an illustration of how this is handled.

See above for comments on `GOAL_POS_TYPE`.

Goto specific relative goal.

```
GOAL_POS_TYPE goal;

tcaWaitForCommand(CTR_GOTO_GOAL_COMMAND, &goal);
```

This is intended to be called as a blocking command. Xavier will commence navigating towards *goal*, which is *relative* to the position of Xavier when the central server passes the call to CTR. A shortest path heuristic is used in conjunction with local navigation. As such, this command is effective for simple goal traversal but not should be abused and expected to navigate complex paths (for example, through mazes!).

On termination, Xavier will either be at the goal (within the goal tolerance), or else *Failure* is returned.

See above for comments on GOAL_POS_TYPE.

Begin going to relative goal

```
GOAL_POS_TYPE goal;

tcaExecuteCommand(CTR_SET_GOAL_COMMAND, &goal);
```

Xavier will commence navigating towards *goal*, which is *relative* to the position of Xavier when the central server passes the call to CTR. A shortest path heuristic is used in conjunction with local navigation. As such, this command is effective for simple goal traversal but not should be abused and expected to navigate complex paths (for example, through mazes!).

A constraint message will notify when the goal has been attained successfully (within the goal tolerance). See the example program for an illustration of how this is handled.

See above for comments on GOAL_POS_TYPE.

Go to multiple goals

```
GOAL_POS_TYPE goal;

tcaExecuteCommand(CTR_ADD_GOAL_COMMAND, &goal);
```

If Xavier is not carrying out a task, it will commence navigating towards *goal*, which is *relative* to the position of Xavier when the central server passes the call to CTR. As such it will be the same as going to a relative goal.

However, if Xavier is already going towards goal, this new goal will be followed once previous goals have been achieved.

A mechanism for performing this in an equivalent fashion to the set goal command is not present as path following fills that role.

Navigate in specific direction

```
double dirn;

tcaExecuteCommand(CTR_GO_DIRECTION_COMMAND,
                  &dirn);
```

Xavier will commence navigating parallel to *dirn*, an absolute direction in the world. Local navigation is used to avoid small obstacles. If a large obstacle appears which Xavier cannot circumnavigate without turning to far from the *dirn*, that is, if Xavier falls into a local minima, a constraint message is raised.

Success is returned when Xavier begins navigating.

Concatenate directions

```
double dirn;

tcaExecuteCommand(CTR_ADD_DIRECTION_COMMAND,
                  &dirn);
```

If Xavier is not already following a direction, Xavier will commence navigating parallel to *dirn*, which is relative to the current robot orientation. Local navigation is used to avoid small obstacles. If a large obstacle appears which Xavier cannot circumnavigate without turning to far from the *dirn*, that is, if Xavier falls into a local minima, a constraint message is raised.

If Xavier is already following a direction, this new direction *dirn* will be taken when a local minimum is encountered. *dirn* will be relative to the robot orientation at that time.

Change current direction

```
double dirn;

tcaExecuteCommand(
     CTR_INCREMENT_DIRECTION_COMMAND,
     &dirn);
```

If Xavier is not already following a direction, Xavier will commence navigating parallel to *dirn*, which is relative to the current robot orientation. Local navigation is used to avoid small obstacles. If a large obstacle appears which Xavier cannot circumnavigate without turning to far from the *dirn*, that is, if Xavier falls into a local minima, a constraint message is raised.

If Xavier is already following a direction, this new direction *dirn* will be taken when a local minimum is encountered. *dirn* will be relative to the robot orientation at that time.

Set the accuracy of "goto" commands

```
double goalTolerance;

tcaExecuteCommand(CTR_GOAL_TOLERANCE_COMMAND,
                  &goalTolerance);
```

All commands which attempt to move Xavier to some specific point (go to goal, move, the individual components of follow path), will attempt to get within *goalTolerance* at each point.

Setting *goalTolerance* to too high a value will lead to sloppiness in navigation. Setting it to too small a value risks time wasting as Xavier attempts to shuffle to the correct location. Also, if a goalTolerance lies between encoding readings (the smallest distance Xavier can discriminate between), there is a risk of hovering about a goal without ever reaching it. A "high" value might be 50-100cm, and a "small" value might be 5-10cm.

Set the accuracy of "goto direction" commands

```
double angle;
tcaExecuteCommand(CTR_MIN_FACING_ANGLE_COMMAND,
                  &angle);
```

All commands which attempt to rotate Xavier to some specific orientation (go to dirn, rotate, the individual components of follow wall), will attempt to get within *angle* at each point.

Setting *angle* to too high a value will lead to sloppiness in navigation. Setting it to too small a value risks time wasting as Xavier attempts to shuffle to the correct location. Also, if a goalTolerance lies between encoding readings (the smallest distance Xavier can discriminate between), there is a risk of hovering about a direction without ever reaching it.

Random Exploration

```
tcaExecuteCommand(CTR_WANDER_COMMAND, NULL);
```

When wandering mode is enabled, Xavier will constantly move towards the largest opening in front of it. If no clear space is available in front, it will rotate until an opening appears. This rotation direction can be amended by fine tuning the reactive behaviour.

Note that the wandering routine is purely local and does not try to maximize exploration (that is, it does not guarantee that it will do the most effective exploration of a domain in a given time).

The wandering mode will continue until a further command effects the position or motion of the base is issued (halt being the most appropriate). Alternatively, if an obstacle is hit, the wander command will terminate.

Timed random exploration

```
int seconds;

tcaExecuteCommand(CTR_TIMED_WANDER_COMMAND,
                  &seconds);
```

This performs the same actions as normal wander, but will always terminate after a specific time.

Fine tune reactive behavior

```
int turn_type =
      {RIGHT_TURN | LEFT_TURN | WHATEVER_TURN};
tcaExecuteCommand(
      CTR_PREFERRED_DIRECTION_COMMAND,
      &turn_type);
```

On encountering an obstacle, the local reactive navigation function could either turn right or left to avoid the obstacle. When an obstacle can be avoided by turning less than 135° in either direction, there is an obvious choice of what direction in which to turn. However, whenever the local behaviour discovers that it needs to turn more than 135°, the direction of turn is arbitrary.

By default, `WHATEVER_TURN` is enabled and Xavier will continue turning in the direction it turned last. That is, if it last had done a small turn right, the preferred direction would be towards the right. In certain environments, such as a maze, forcing Xavier to always turn in one direction can be beneficial. The preferred direction only has effect on turns greater than 135°.

This command should always be executed using the non-blocking TCA call.

Initiate wall following

```
FOLLOW_WALL_TYPE fw;
fw.side = {RIGHT_WALL | LEFT_WALL};
fw.speed = ?;
tcaExecuteCommand(CTR_FOLLOW_WALL_COMMAND, &fw);
```

Following this call, Xavier will commence continuously taking the readings of the sonars at the chosen side, measuring the wall direction and distance, and setting the current direction to be the same as the wall direction. Xavier will then move parallel to this wall.

The construction of the command is such that it expects to be executed in a hallway setting. In an open space, it will attempt to go to the middle of a room, and then follow the wall.

Execute path following

```
POINTS_TYPE path;
points->num=?
for( i = 0; i < points->num; i++){
     points->obs_x[i] = x[i];
     points->obs_y[i] = y[i];
  }
tcaWaitForCommand(CTR_FOLLOW_PATH_COMMAND,
                  &path);
```

The path is specified as a series of points relative to the current robot location. Each point in the path will be treated as an individual goto goal operation. The series of goto's starts immediately. When all points in the path have been reached successfully, *Success* will be returned. If any individual segment fails, the call will return with *Failure*.

Initiate path following

```
tcaExecuteCommand(CTR_SET_PATH_COMMAND, &path);
```

The path is specified as a series of points relative to the current robot location. Each point in the path will be treated as an individual goto goal operation. The series of goto's starts immediately, and *Success* will be returned.

When all points in the path have been reached successfully, a constraint message will be raised. If any individual segment fails, the constraint message will contain false. (See example programs.)

4.2.1.3 Queries

This section describes the set of queries that can be posted to the CTR device. They appear in the following order:

- Current position
- Current direction
- Motion Report
- Status of laser and sonar
- Status of laser
- Status of sonar
- Sonar broadcast
- Status of battery

Current position

```
POS_TYPE curPosn;

tcaQuery(CTR_POSITION_QUERY, NULL, &curPosn);
newx = curPosn.x;
newy = curPosn.y;
neworientation = curPosn.orientation;
```

Ask for the current dead reckoning position (x, y, orientation) of the base.

Current Heading

```
double dirn;

tcaQuery(CTR_DIRECTION_QUERY, NULL,
         &dirn);
```

This describes the current direction in which the robot *may* be moving. The robot need not actually be moving at this point in time. The direction represents absolute degrees relative to the origin. Also, this direction is not-necessarily the current orientation.

Motion Report

```
MOTION_TYPE motionStatus;

tcaQuery(CTR_MOTION_REPORT_QUERY, NULL,
         &motionStatus);
```

The motion report query returns the robot's motion state: Current direction, current rotation speed, rotation acceleration, commanded rotation speed, current translation speed, translation acceleration and the robot's commanded translation speed.

All information

```
STATUS_TYPE robotStatus;
/* defined in CTR-commands.h */

tcaQuery(CTR_STATUS_REPORT_QUERY, NULL,
         &robotStatus);
```

This yields the fullest information available to a controlling program relating to the state of the robot. The position information is updated every call. The sonar information is updated every time a new reading occurs, as indicated by the sonar frame number incrementing. When the sonar number increments, `data` will contain the current sonar readings for that position, clockwise from sonar 0. This data should only be referred following a sonar frame transition. Finally, the laser readings contain the latest number of obstacles read by the sonar. They are in x,y locations relative to the current location.

Laser information

```
int resolution;
LASER_READINGS_PTR current_readings;

tcaQuery(CTR_LASER_READINGS_QUERY,
         &resolution,
         &current_readings);
```

This will return in `current_readings`, a listing of obstacles viewed under the current laser scan. The resolution defines the fineness of the laser reading sampling. If the resolution is large, the probability of small or far away objects slipping through gaps is greater, but the query will also be faster. *example range of resolution*

The final data returned may be treated as the laser readings in the basic status report query.

Sonar and position information

```
SONAR_STATUS_TYPE sonarStatus;

tcaQuery(CTR_SONAR_STATUS_QUERY, NULL,
         &sonarStatus);
```

The sonar status query returns a subset of the basic status report with no laser readings present.

Initiate sensation notification

```
int activate;
tcaExecuteCommand(CTR_SONAR_STATUS_COMMAND,
                  &activate);

CTR_SONAR_STATUS_BROADCAST
```

Twice a second a new sonar reading is available. It is possible to continuously query the base for this information, but this places undue load on the system. Instead, in situations where every sonar reading is useful, it is better to let the base notify your program when one is ready. This call will either activate or deactivate this ability. It is then your programs responsibility to construct a handler for this message. See the sonar status query for further notes of the information this message would yield. In addition, Xavier can automatically notify the controlling process when new readings arrive. The sample code in section 4.4.2 (page 46) provides a template for mapping utilities.

Battery indication

```
int battery_level;

tcaQuery(CTR_BATTERY_QUERY, NULL,
         &battery_level);
```

This returns the current battery level on Xavier. Units are 1/10 of a volt.

4.2.2 CTS – the Speech Module

CTS controls the speech card. When any of the CTS commands or queries are used, CTS must be running. If the real hardware is being used, CTS must be run on **Ariel**, the computer carrying the speech card. A copy of CTS is available in:

```
/afs/cs/project/robocomp/bin
```

(as are copies of all programs referenced).

```
machine% setenv CENTRALHOST machinewithcentral
machine% CTS
>
```

Currently, CTS expects only the following options:

-x = No X11 window will be displayed. CTS can still be controlled with the keyboard.

-t = No attempt is made to connect to TCA. This is useful when speech is to be typed by the user without a higher controlling program being involved.

-s **machine** = Attempts to connect to a simulator on this machine

-p **num** = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

If the **-x** option is not set, an X11 window will appear, allowing graphical control of the speech board (see Figure 4.3). This window is duplicated by the keyboard. Clicking on the windows will pop up widgets allowing sentences to be spoken or PCM files played.

If the **-x** option is set, no window is displayed and control is effected using either the keyboard, or the CTS TCA suite.

4.2.2.1 Commands

This section describes the commands to use CTS, in the following order:

- Text to Speech
- Say Hello
- Sing Song
- Play File

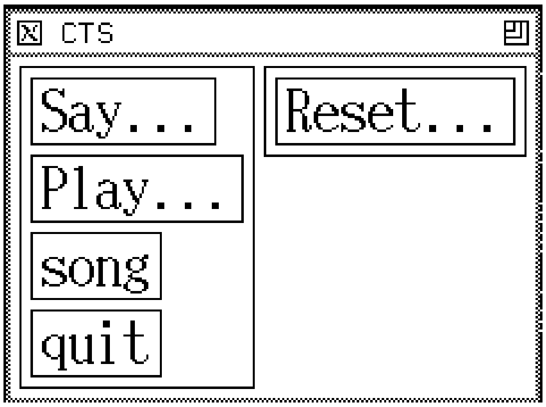

Figure 4.3: **CTS X11 Display** A graphical interface for controlling and monitoring the speech unit.

Text to speech

```
char stringToSay[]="A string to say.";
tcaExecuteCommand(SAY_COMMAND, &stringToSay);
```

The stringToSay can be of arbitrary length. It will be parsed in hardware, with American English pronunciation rules applied. Normal punctuation should be used in the string, as the parser will inflect questions, statements, etc. as well as possible. The string should not be terminated with '`\n`' but simply null-terminated, as is standard C convention.

Preprogrammed introduction

```
tcaExecuteCommand(SAY_HELLO_COMMAND, NULL);
```

Says Xavier's preprogrammed welcome message of the moment. Currently this is "Hello. My name is Xavier. I am very happy to be here today. I hope you enjoy the demonstration."

Preprogrammed song

```
tcaExecuteCommand(SING_SONG_COMMAND, NULL)
```

Xavier will 'sing' a prepared message. Currently this message is "I am done. Time for a Guinness". This is typically used for a finale at task completion – at AAAI-93, Xavier would spin and sing to indicate success (failure can be swiftly communicated by hitting a wall!).

File replay

```
char soundFile[]="Name of File";
tcaExecuteCommand(PLAY_SOUND_COMMAND,
                  &soundFile);
```

Attempts to open up a PCM coded sound file. There are several of these available in `~\xavier\doc\speech` including laser zaps, Star Trek door openings, and energizer wooshes. One can also use the standard "sndrecord" program on a Next computer to produce a PCM formated sound file. The file name should be fewer than 128 characters long. If the soundFile cannot be opened, a tcaFailure will be returned.

Due to an unresolved bug, this command is only effective once - after it has completed successfully, the speech board will be in an unstable state and will have to be reset.

4.2.3 CTV, CTPan, and CTCamera – the Vision Modules

The vision module CTCamera controls the frame-grabber and the module CTPan controls the pan/tilt unit. The module CTV can take the place of both of these on Xavier. When any of the following commands or queries are used, the appropriate vision module must be running. If the real hardware is being used, these modules must be run on **Ariel**, the computer carrying the framegrabber card. Copies of CTV, CTPan, and CTCamera are available in:

```
/afs/cs/project/robocomp/bin
```

(as are copies of all programs referenced).

```
machine% setenv CENTRALHOST machinewithcentral
machine% CTV
>
```

Currently, CTV expects only the following options:

-x = No X11 window will be displayed. CTS can still be controlled with the keyboard.

-t = No attempt is made to connect to TCA. This is useful when camera contrast is being tested, or the pan/tilt unit being manually servoed into position.

-s **machine** = Attempts to connect to a simulator on this machine

-p **num** = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

-**P** = Do not attempt to use the pan/tilt unit.

Currently, CTPan expects only the following options:

-x = No X11 window will be displayed. CTPan can still be controlled with the keyboard.

-t = No attempt is made to connect to TCA. This is useful when the pan/tilt unit being manually servoed into position.

-s **machine** = Attempts to connect to a simulator on this machine

-p num = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

Currently, CTCamera expects only the following options:

-x = No X11 window will be displayed. CTCamera can still be controlled with the keyboard, however you cannot display an image.

-t = No attempt is made to connect to TCA. This is useful when camera contrast is being tested, or the pan/tilt unit being manually servoed into position.

-s machine = Attempts to connect to a simulator on this machine

-p num = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

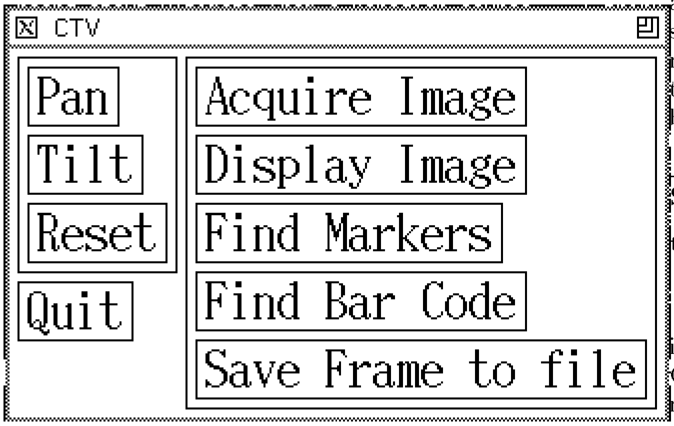

Figure 4.4: **CTV X11 Display** A graphical interface for controlling and monitoring the frame-grabber and pan/tilt unit, with the added bonus of saving TIFF snapshots. **CTPan**'s X11 display is the left portion of this, and **CTCamera**'s is a modified version of the right, allowing a file to be saved in either a GIF or a TIFF format.

If the **-x** option is not set, an X11 window will appear, allowing graphical control of the speech board (see Figure 4.4). This window is partially duplicated by the keyboard, which fails to provide file saving options in CTV, and for displaying images in both CTV and CTCamera.

Always acquire an image before displaying, saving or attempting to find markers.

If the **-x** option is set, no window is displayed and control is effected using either the keyboard or the CTV-TCA suite.

4.2.3.1 Commands

This section describes the commands to use these modules, in the following order:

- Reset pan/tilt head (p)
- Shutdown pan/tilt head (p)
- Set (pan and tilt) velocity (p)
- Pan (p)
- Pan relative (p)
- Tilt (p)
- Tilt relative (p)
- Move (pan and tilt) (p)
- Move relative (pan and tilt) (p)
- Grab snapshot (p)

All of the above are usable with CTV. Those marked with a (p) are used with CTPan, and the last marked with a (c) is used by CTCamera. Reid Simmons has added further commands to CTCamera which performs modifications on the captured image for better clarity.

Reset pan-tilt unit

```
tcaExecuteCommand(PAN_TILT_RESET_COMMAND, NULL);
```

Causes the pan-tilt head to return to its home position and to recheck its limits. This is achieved by the unit servoing in all degrees until the limit switches are set, then returning to the midpoint. It has been found in practice that this operation occasionally fails, so tcaSuccess should be checked.

Shutdown pan-tilt unit

```
tcaExecuteCommand(PAN_TILT_SHUTDOWN_COMMAND,
                  NULL);
```

This does not change the position of the pan-tilt head in any way but should be called prior to closing CTV or CTPan. The pan-tilt unit uses some temporary system resources, principally a file descriptor and a serial port, and this command ensures that they are freed. When the module is shut down cleanly they are also freed.

Pan-tilt Speed

```
PAN_TILT_SET_VELOCITY_TYPE p;

p.panVelocity  = (double) 20.0;
p.tiltVelocity = (double) 20.0;
tcaExecuteCommand(PAN_TILT_SET_VELOCITY_COMMAND,
                  &p);
```

This sets the speed at which the pan-tilt unit will servo to a new location. The minimum and maximum speeds are volatile and should be checked using `PAN_TILT_LIMITS_QUERY`. If the new velocity attempts to exceed these limits, nothing will change and a tcaFailure will be returned.

Pan

```
DEGREES d;

d = 20.0;
tcaExecuteCommand(PAN_TILT_PAN_COMMAND, &d);
```

This pan-tilt unit will rotate in a plane horizontal to the floor to the degree, d, specified. If d exceeds the limits of the unit, nothing will change and a tcaFailure will be returned. The speed at which the unit servos can be adjusted using `PAN_TILT_SET_VELOCITY`.

Pan relative

```
DEGREES d;

d = 20.0;
tcaExecuteCommand(
    PAN_TILT_PAN_RELATIVE_COMMAND, &d);
```

This pan-tilt unit will rotate in a plane horizontal to the floor to the degree, d, relative to its current rotation. If d+current rotation exceeds the limits of the unit, nothing will change and a tcaFailure will be returned. The speed at which the unit servos can be adjusted using `PAN_TILT_SET_VELOCITY`.

Tilt

```
DEGREES tiltDegree;

tiltDegree = 45.0;
tcaExecuteCommand(PAN_TILT_TILT_COMMAND,
                  &tiltDegree);
```

This pan-tilt unit will tilt in a plane vertical to the floor to the degree, $tiltDegree$, specified. If $tiltDegree$ exceeds the limits of the unit, nothing will change and a tcaFailure will be returned. The speed at which the unit servos can be adjusted using `PAN_TILT_SET_VELOCITY`.

Tilt relative

```
DEGREES tiltDegree;

tiltDegree = 45.0;
tcaExecuteCommand(
    PAN_TILT_TILT_RELATIVE_COMMAND,
    &tiltDegree);
```

This pan-tilt unit will tilt in a plane vertical to the floor to the degree, $tiltDegree$, relative to the current tilt. If $tiltDegree+$ the current tilt exceeds the limits of the unit, nothing will change and a tcaFailure will be returned. The speed at which the unit servos can be adjusted using `PAN_TILT_SET_VELOCITY`.

Move (pan and tilt)

```
PAN_TILT_MOVE_TYPE p;
p.panAngle  = (DEGREES)20.0;
p.tiltAngle = (DEGREES)45.0;

tcaExecuteCommand(PAN_TILT_MOVE_COMMAND, &p);
```

Move relative (pan and tilt)

```
PAN_TILT_MOVE_RELATIVE_TYPE p;

p.panAngle  = (DEGREES)20.0;
p.tiltAngle = (DEGREES)45.0;

tcaExecuteCommand(
    PAN_TILT_MOVE_RELATIVE_COMMAND,
    &p);
```

Grab snapshot

```
tcaExecuteCommand(ACQUIRE_IMAGE_COMMAND, NULL);
```

JOSEPH: where can they then find the image?

4.2.3.2 Queries

This section describes the commands to use CTV, in the following order:

- Information
- Current position
- Marker location in current image
- Barcode presence in current image

Basic information

```
PAN_TILT_LIMITS_TYPE p;

tcaQuery(PAN_TILT_LIMITS_QUERY, NULL, &p);
/*
 p.minPanAngle, p.maxPanAngle,
 p.minTiltAngle, p.maxTiltAngle all degrees
 p.maxPanVelocity, p.maxTiltVelocity both double
*/
```

Current position

```
PAN_TILT_POSITION_TYPE p;

tcaQuery(PAN_TILT_POSITION_QUERY, NULL, &p);
printf("Camera is pointed (P: %d, T: %d)\n",
  (int)p.pan, (int)p.tilt);
```

Marker location in current image

```
MARKERS_DATA_TYPE m;
int i;

tcaQuery(FIND_MARKERS_QUERY, NULL, &m);
printf("Found %d markers\n", m.num);
printf("Robot was at (%d, %d, %d)\n",
       (int)m.robotPos.x, (int)m.robotPos.y,
       (int)m.robotPos.orientation);
printf("Camera is pointed (P: %d, T: %d)\n",
       (int)m.panTiltAngles.pan,
       (int)m.panTiltAngles.tilt);
for (i=0; i<m.num; i++)
    printf("Marker `%c' at (%d,%d,%d)-polar, \
             (%d,%d,%d)-cartesian\n",
           m.markers[i].symbol,
           m.markers[i].dist,
           m.markers[i].rho, m[i].markers.psi,
           m.markers[i].x, m[i].markers.y,
           m.markers[i].z);
```

Find the markers present (if any) in the current image. This assumes an image has already been acquired using ACQUIRE_IMAGE. If not, there is no guarantee of the accuracy of the result.

These barcodes are available in the xavier subdirectory ~/doc/rules/markers2.ps.

Barcode presence in current image

```
BARCODE_DATA_TYPE b;

tcaQuery(FIND_BARCODE_QUERY, NULL,  &b);
printf("Barcode type %d\n", b.barcode);
```

Attempts to find a barcode in the current image. As such, it assumes an image has already been acquired using ACQUIRE_IMAGE. If not, there is no guarantee of the accuracy of the result.

If a tcaSuccess is returned, *b.barcode* will contain the type of the barcode found:

0 for no barcode,
1 for a single vertical bar,
2 for two vertical bars, side by side,
3 for three horizontal bars, one on top of the other.

Typically, these barcodes would be placed next to markers. The FIND_MARKERS_QUERY returns the position of a marker. Thus the camera can focus on the expected area where the barcode may be, heightening the reliability of the query.

These barcodes are available in the xavier subdirectory ~/doc/rules/markers2.ps.

4.2.4 CTA - the Arm

This section describes the commands to use CTA, in the following order:

- Lower arm
- Raise arm
- Grasp object
- Ungrasp object
- Arm status

Lower arm

```
tcaWaitForCommand(ARM_EXTEND_COMMAND, NULL);
```

The arm is stored in an upright position. This command will drop the arm parallel to the ground. If the arm is already on the ground nothing will happen. There is no sensor on the arm, so this command simply assumes success and that no blockages occurred.

Raise arm

```
tcaWaitForCommand(ARM_RETRACT_COMMAND, NULL);
```

The arm is stored in an upright position. This command will return the arm to that vertical position. If the arm is vertical nothing will happen. There is no sensor on the arm, so this command simply assumes in success and that no blockages occurred.

Grasp object

```
tcaWaitForCommand(ARM_GRASP_COMMAND, NULL);
```

This will turn on the magnets at the tip of the arm. It doesn't matter whether the arm is up or down.

Ungrasp object

```
tcaWaitForCommand(ARM_UNGRASP_COMMAND, NULL);
```

This will turn off the magnets at the tip of the arm. It doesn't matter whether the arm is up or down.

Arm status

```
ARM_STATUS_TYPE arm;

tcaWaitForCommand(ARM_STATUS_QUERY,
                  NULL, &arm);
if (arm.armStatus)
/*Extended*/;
if (arm.magnetStatus)
/*Magnet on*/
```

This will return the current status of the arm. There are no sensors on the arm, so this is not guaranteed to be correct – it simple replicates the projected effects of the various calls to the arm module.

4.2.5 CTRECORD – Data Acquisition

CTRECORD is implemented as a TCA controller which should be started up along with CTR, CTS, CTV etc. It acts as an transparent eavesdropping agent, wiretapping all commands sent to Xavier from controlling programs. If it is recording from hardware, it must be run on the machine the frame-grabber resides (currently **Ariel**).It can be run as follows:

```
machine% setenv CENTRALHOST machine
machine% CTRECORD filename
...
> r             (for record)
> a -Hi there. (Add message to file)
> p             (to pause recording)
> a -Getting close to that door.
> a -Past it.
> r             (to resume)
> q             (to quit)
machine% [filename is all ready to play with]
machine% CTREPLAY filename
```

CTRECORD will continuously monitor Xavier for changes in location/environment/settings. All changes will be entered into the log file in the format*JOSEPH: hunh?*. In situations when a TCA command to Xavier is being executed but hasn't completed yet (eg. wander, follow path), a polling monitor will take snapshots at regular intervals. Comments can be inserted into the file using the "a" command. These comments will be ignored by the parser. In addition, the faint of heart can explicitly fake commands by starting comment lines with #*KEYWORD*. For instance, to add a dummy robot position momentarily to the file:

```
> a "#ROBOT x y theta 0"
```

If an incorrect command is added to the file, it will interfere with the succeeding environment changes until the next perception of the world occurs.

Alternatively, CTRECORD can explicitly be commanded by the controlling program using TCA, RECORD_SAVE_FRAME_COMMAND. This will save to the record file the current sensations of Xavier. It will not save actions taken between calls. If this is required, then the RECORD_GO_RECORDING_COMMAND, RECORD_PAUSE_RECORDING_COMMAND (as used by the text interface) should be used as before. This will allow Xavier to automatically record interesting periods of its life (sort of, "here's what I did on my holidays when I visited the 6th floor"). RECORD_ADD_COMMENT_COMMAND does the expected.

This data file can be parsed using functions available in log_status.o, reviewed using replay, or fed to your control program using CTREPLAY. This is useful for debugging programs using real world data. You can control Xavier as you wish while running CTREPLAY. However, Xavier will ignore commands off its path, and will return sensor information according to the previous readings taken during the real run. In fact, running CTREPLAY should be a good test of your controlling code, where you really don't know whether or not an action will succeed the way you expected (*i.e.* go forward 2 doesn't succeed and instead Xavier goes backward 10).

Currently, CTRECORD expects only the following options:

-t = No attempt is made to connect to TCA.

-v = No vision data is saved.

-c = continuous recording. Pictures will be taken even when the robot is not moving.

-d = Displays time-stamping information as each snapshot is grabbed.

-s **machine** = Attempts to connect to a simulator on this machine

-p num = Uses **num** as the port to connect on this machine. Useful if multiple simulators are running. **num** is any number that does not appear in the fourth column of the output when you run "`netstat`".

BUGS: Some TCA commands aren't being saved to file yet. If an unknown command occurs, the polling monitor begins running until the command completes. If any important commands appear to be missing (*i.e.*, your controlling program complains) let me know and I'll add them in.

Note that you occasionally will get messages such as "WARNING: Message 'R_SonarStatus' received but not processed: Handler 'l_updateMapHnd' not registered". This is just CTRECORD's mechanism for flushing data at work.

Also, CTRECORD may grab one frame on calling rec_pauseRecording. As such, always do waitFor rec_PauseRecording when you want to be certain that recording has actually stopped!

4.2.5.1 Commands

Below are ways for your program to explicitly save features of interest, in the following order:

- Save sensation
- Begin recording data
- Stop recording data
- Add comment to data file
- Change category in data file data file

Save sensation

```
tcaWaitForCommand(RECORD_SAVE_FRAME_COMMAND,
                  NULL);
```

This will save to the record file the current sensations of Xavier. It will not save actions taken between calls.

Begin recording data

```
tcaWaitForCommand(RECORD_GO_RECORDING_COMMAND,
                  NULL);
```

Stop recording data

```
tcaWaitForCommand(
       RECORD_PAUSE_RECORDING_COMMAND,
       NULL);
```

Add comment to data file

```
char comment[50];

tcaExecuteCommand(RECORD_ADD_COMMENT_COMMAND,
                  &comment);
```

Change category in data file

```
int category;

tcaExecuteCommand(
    RECORD_CHANGE_CATEGORY_COMMAND,
    &category)
```

JOSEPH: what does this mean?

4.2.6 CTREPLAY – Off-line Data Review

CTREPLAY mimics CTR, CTV and CTS just from the recorded data. This is useful for debugging programs using real world data. You can control xavier while running CTREPLAY as you wish. However, xavier will ignore commands off its path, and will return sensor information according to the previous readings taken during the real run. As such, CTREPLAY emulates the commands of all other packages (except CTRECORD).

'CTREPLAY' allows for graphically viewing this data. Monochrome, 8 bit greyscale and 8 bit color monitors are supported in replay. This code has been compiled for DEC/5000, DEC/alpha, Sun and i486 machines. It is not advised that a slower machine such as a Dec/3100 be used.

To replay a run (we assume TCA is already going...):

```
machine% setenv CENTRALHOST somemachine
machine% CTREPLAY filename
>
```

This will open 2 windows on the X window display entitled 'Map' and 'Last image' respectively.

You control the rate of motion through the data file. The movie command 'm' will advance forward at real-time. The reverse command 'M' reverses in real-time.

The Camera View is toggled using the verbose flag ('v' on the CTREPLAY command line) and displays the current visual perception - an intensity display or actual image on a color monitor. Using 't threshold' varies the quality of this picture on a monochrome monitor.

The map window depicts the position of the robot relative to the start of a run (located at (0,0)). 24 rays protrude from a central location - these signify the sonar reading at any point in time. By entering 'm' for movie, you can see how the sonar readings change as the robot goes past obstacles (press any key to stop the movie sequence). In addition, by entering 'e', toggling erase mode, old lines are not erased, which leads to a map of free space (save for noise and sensor error), being built up. If you just wish to see the objects detected by the laser and the sonar, 'l' toggles line/point mode. In point mode, the laser is also visible as a horizontal dot (the sonar being a vertical dot). As the robot moves, if 'e' is off, walls and obstacles will be traced out. In addition, the window can zoom in and center on portions of the world. On a color screen, the robot path will appear in green, the laser will always be present as a yellow trace, with the sonar being increasingly white, depending on the likelihood of an object actually having being detected.

4.2.7 Basic Commands

Described below are the following basic commands, in the following order:

- Navigate to Goal
- Navigate Path Goal
- Plan Path Query

- Distance Query
- Distance Reset
- Distance Update

- Near Room Query
- All Points Near Room Query
- Where Am I Query
- Where Am I Now Query
- Room Location Query

- Find Door Command
- Find Door Bound Command
- Where Door Query
- Go through Door Command
- Go through Door Cont Command
- Goto Door Goal
- Confirm Door Goal

- Approach Object Command
- Exit Room Goal
- Exit Room (Continued) Goal
- Explore Room Goal
- Wander around Goal
- Follow Wall Goal
- Aiming for biased wall Goal
- Searching Biased Wall Goal
- Traverse Corridor Goal

- Turn to face Command
- Head Toward Command
- Goto Map Location Command
- Traverse Segment Goal
- Start Traverse Goal
- End Traverse Goal
- Update Map Transform Command

Navigate to Goal

```
NAVIGATE_TO_GOAL_TYPE navigateData;
navigateData.mapX = (double) x;
navigateData.mapY = (double) y;
tcaExpandGoalWithConstraints(
    NULL,
    NAVIGATE_TO_GOAL,
    &navigateData,
    SEQ_ACH+DELAY_PLANNING);
```

This command will get Xavier to plan a path to `navigateData` coordinates and then navigate along it.

Navigate Path Goal

```
QMAP_PATH_PTR plannedPath;
NAVIGATE_PATH_TYPE navigatePathData;
plannedPath = QMAP_plan_path(curX,curY,
                             toX,toY,
                             CLOSE_NODE);
navigatePathData.path = plannedPath;
navigatePathData.startOrientation =
        curOrientation;
tcaExpandGoal(NAVIGATE_PATH_GOAL,
              &navigatePathData);
\index{NAVIGATE\_PATH\_GOAL}
```

This command will get Xavier to navigate along the path determined by `plannedPath`.

Plan Path Query

```
typedef struct { int id;
                 int room;
                 char* room_name;
                 double x, y;
                 int arcId;
           } QMAP_PATH_NODE_TYPE;

PLAN_PATH_QUERY_TYPE planData;
PLAN_PATH_REPLY_TYPE plannedPath;
planData.from.mapX = (double);
planData.from.mapY = (double);
planData.to.mapX = (double);
planData.to.mapY = (double);

tcaQuery(PLAN_PATH_QUERY,
        &planData,
        &plannedPath);

int plannedPath.numNodes;
double plannedPath.length;
QMAP_PATH_NODE_TYPE *plannedPath.nodes;
```

This call will plan a path from `planData.from` to `planData.to` and return it in `plannedPath`.

Distance Query

Distance Reset

```
tcaExecuteCommand(DISTANCE_RESET_COMMAND, NULL);
```

Distance Update

Near Room Query

```
roomLocation.mapX = (double);
roomLocation.mapY = (double);
tcaQuery(NEAR_ROOM_QUERY,
         &roomLocation,
         &nearbyCorridor);
double nearbyCorridor.mapX;
double nearbyCorridor.mapY;
double nearbyCorridor.dirToRoom;
```

Given a room location, returns a nearby corridor location, and the direction from the corridor point towards the door.

All Points Near Room Query

```
ALL_POINTS_NEAR_ROOM_QUERY_TYPE
   roomLocation;
ALL_POINTS_NEAR_ROOM_REPLY_TYPE
   allPointsNearRoomReply;

roomLocation.mapX = (double);
roomLocation.mapY = (double);
tcaQuery(ALL_POINTS_NEAR_ROOM_QUERY,
         &roomLocation,
         &allPointsNearRoomReply);

int allPointsNearRoomReply.num_doors;
double allPointsNearRoomReply.location[ct].mapX
double allPointsNearRoomReply.location[ct].mapY
doubleallPointsNearRoomReply.dirToRoom[ct]
```

Given a room location, returns all the nearby corridor locations.

Where Am I Query Translates input (x,y) coordinates to map position. Returns room name.

Where Am I Now Query

```
WHEREAMI_NOW_REPLY_TYPE whereami;
tcaQuery(WHEREAMI_NOW_QUERY,
         NULL,
         &whereami);
char *whereami.roomName;
double whereami.mapPose.x;
double whereami.mapPose.y;
double whereami.mapPose.orientation;
```

Finds out where robot is now, returns location & room name.

Room Location Query

```
char *room_num;
MAP_LOC_TYPE roomLocation;
tcaQuery(ROOM_LOCATION_QUERY,
         &(room_num),
         &roomLocation);
double roomLocation.mapX;
double roomLocation.mapY;
```

Find Door Command

```
FIND_DOOR_TYPE findDoor;

findDoor.startAngle = (double);
findDoor.endAngle = (double);
findDoor.minWidth = (double) MIN_DOOR_WIDTH;
findDoor.maxWidth = (double) MAX_DOOR_WIDTH;
findDoor.idealWidth = (double) IDEAL_DOOR_WIDTH;
tcaExecuteCommand(FIND_DOOR_COMMAND, &findDoor);
```

Find the door by using the laser to scan once straight ahead, once left and once right. Returns success/failure.

Find Door Bound Command Seems to find the door with more limited/focussed scanning.

Where Door Query

```
WHERE_DOOR_REPLY_TYPE whereDoor;
tcaQuery(WHERE_DOOR_QUERY, NULL,
        &whereDoor);
(double) whereDoor.robotPos.x;
(double) whereDoor.robotPos.y;
(double) whereDoor.doorLoc.x;
(double) whereDoor.doorLoc.y;
(double) whereDoor.doorWidth;
```

Returns the information about where exactly the door is. It seems that `FIND_DOOR_COMMAND` must be called beforehand.

Go through door Command

```
TCA_REF_PTR goThruDoorRef =
    tcaCreateReference(GO_THROUGH_DOOR_GOAL);
GO_THROUGH_DOOR_TYPE goThruDoor;
goThruDoor.startAngle = (double);
goThruDoor.endAngle = (double);
goThruDoor.endAngle.startLocPos.x = (double);
goThruDoor.endAngle.startLocPos.y = (double);
goThruDoor.endAngle.startLocPos.orientation =
                                      (double);
tcaExpandGoalWithConstraints(
     goThruDoorRef,
     GO_THROUGH_DOOR_GOAL,
     &goThruDoor,
     SEQ_ACH);
```

This command finds the door and then calls GO_THROUGH_DOOR_CONT_GOAL.

Go through door Cont Command

```
GO_THROUGH_DOOR_TYPE goThruDoor;
tcaExpandGoal(GO_THROUGH_DOOR_CONT_GOAL,
              &goThruDoor);
```

This command goes through the door.

Goto Door Goal

Confirm Door Goal Get as close as we can to the door and then go through it.

Approach Object Command

```
approachData.minDistance =
    (double) MIN_DOOR_APPROACH_DISTANCE;
approachData.maxDistance =
    (double) MAX_DOOR_APPROACH_DISTANCE;
approachData.maxMove =
    (double) 100.0;
approachData.searchWidth =
    (double) 2.0*ROBOT_RADIUS;
tcaExecuteCommand(
    APPROACH_OBJECT_COMMAND,
    &approachData);
```

Exit Room Goal

```
tcaExpandGoal(EXIT_ROOM_GOAL, NULL);
```

Finds walls, looks for doors, etc, then finally leaves if it thinks it can.

Exit Room (Continued) Goal

```
segmentRef =
tcaCreateReference(EXIT_ROOM_CONTD_GOAL);
    tcaExpandGoalWithConstraints(
segmentRef,
    EXIT_ROOM_CONTD_GOAL,
    NULL,
    SEQ_ACH+DELAY_PLANNING);
```

Does the actual exit.

Explore Room Goal Doesn't seem to really exist.

Wander around Goal

```
tcaExpandGoal(WANDER_AROUND_GOAL, NULL);
```

Follow Wall Goal

```
(double) wallAngle;
segmentRef =
    tcaCreateReference(FOLLOW_WALL_GOAL);
tcaExpandGoalWithConstraints(segmentRef,
                             FOLLOW_WALL_GOAL,
                             &wallAngle,
                             SEQ_ACH);
```

Aiming for biased wall Goal

Searching Biased Wall Goal

Traverse Corridor Goal

```
traverseCorridorData.landmark = (int);
traverseCorridorData.minDistance = (double);
traverseCorridorData.maxDistance = (double);
tcaExpandGoalWithConstraints(
   NULL,
   TRAVERSE_CORRIDOR_GOAL,
   &traverseCorridorData,
   SEQ_ACH+DELAY_PLANNING);
```

Turn to face Command

```
double exitDir;
tcaExecuteCommand(TURN_TO_FACE_COMMAND,
                  &exitDir);
```

Head Toward Command

```
double newDir;
tcaExecuteCommand(HEAD_TOWARD_COMMAND, &newDir);
```

Goto Map Location Command

```
MAP_LOC_TYPE mapGoal;

mapGoal.mapX = x;
mapGoal.mapY = y;
tcaExecuteCommand(GOTO_MAP_LOC_COMMAND,
                  &mapGoal);
```

Traverse Segment Goal

```
TRAVERSE_SEGMENT_TYPE traverseData;

traverseData.landmark = (int);
traverseData.minDistance = (double);
traverseData.maxDistance = (double);
traverseData.turn = (double);
traverseData.onMap = (int) TRUE;
   /* TRUE if landmark is a place on the map */
traverseData.mapPose.x = (double);
traverseData.mapPose.y = (double);
traverseData.mapPose.orientation = robotDir;

segRef =
    tcaCreateReference(TRAVERSE_SEGMENT_GOAL);
tcaExpandGoalWithConstraints(
    segRef,
    TRAVERSE_SEGMENT_GOAL,
    &traverseData, SEQ_ACH);
```

Start Traverse Goal

```
START_TRAVERSE_TYPE startTraverseData;

startTraverseData.maxTransSpeed =
        (double) transVelocity;
startTraverseData.maxRotSpeed =
        (double) rotVelocity;
tcaExpandGoal(START_TRAVERSE_GOAL,
              &startTraverseData);
```

End Traverse Goal

```
tcaExpandGoal(END_TRAVERSE_GOAL, NULL);
```

Update Map Transform Command

```
TRAVERSE_SEGMENT_PTR travData;
UPDATE_MAP_TRANSFORM_TYPE updateData;

travData = (TRAVERSE_SEGMENT_PTR)data;

pastFeatureRetries = 0;

traverseAndTurn(travData->landmark,
                travData->minDistance,
                travData->maxDistance,
                travData->turn);

if (travData->onMap) {
  updateData.mapPose = travData->mapPose;
  updateData.landmark = travData->landmark;
  tcaExecuteCommand(
           UPDATE_MAP_TRANSFORM_COMMAND,
           &updateData);
}
```

4.2.8 Exceptions

- Not in Goal Exception
- No unknown area feature Exception
- Past Feature Exception
- Lost and Confused Exception
- Corridor Blockage Exception
- Door Blockage Exception

Not in Goal Exception

No unknown area feature Exception

Past Feature Exception

Lost and Confused Exception

Corridor Blockage Exception

Door Blockage Exception

4.2.9 Monitors

- Go Direction WireTap

- Monitor Distance Query
- Monitor Distance Goal
- Monitor Distance Monitor
- Traverse Monitor Distance Goal
- Traverse Distance Monitor
- Track Monitor Distance Goal
- Track Distance Monitor
- Metric Information Goal
- Metric Information Monitor
- Stat Command
- Init stats Command

- Feature Monitor Setup Query
- Feature Monitor Action
- Markov Monitor Setup Query
- Markov Monitor Setup Action
- Markov Monitor Setup Cancel
- Update Robot Position Monitor
- Update Robot Position Query
- Update Robot Position Action

- Initial Progress Goal
- Progress Monitor Query
- Progress Monitor Action
- Monitor Progress Command
- Wandering Progress Goal
- Wandering Progress Monitor

- Handle Blockage Goal

- Local Unknown Goal
- Global Unknown Goal
- Unknown Area Goal
- Unknown Area Monitor
- Local Area Known Goal
- Local Area Known Monitor

- Try Opening Goal
- Verify Opening Goal
- Verify Opening Monitor
- Opening Goal
- Opening Monitor

- Wall Goal
- Wall Monitor
- Corridor Goal
- Corridor Monitor
- Corridor Direction Goal
- Corridor Direction Monitor
- Corridor Landmark Goal
- Corridor Landmark Monitor
- Turn 360 Goal
- Turn 360 Monitor

Go Direction WireTap

Monitor Distance Query

Monitor Distance Goal

Monitor Distance Monitor

Traverse Monitor Distance Goal

Traverse Distance Monitor

Track Monitor Distance Goal

Track Distance Monitor

Metric Information Goal

Metric Information Monitor

Stat Command

```
tcaExecuteCommand(STAT_COMMAND, NULL);
```

Init stats Command

```
tcaExecuteCommand(INIT_STATS_COMMAND, NULL);
```

Feature Monitor Setup Query

Feature Monitor Action Goal

Markov Monitor Setup Query

Markov Monitor Setup Action

Markov Monitor Setup Cancel

Update Robot Position Monitor

Update Robot Position Query

Update Robot Position Action

Initial Progress Goal

Progress Monitor Query

Progress Monitor Action

Monitor Progress Command

Wandering Progress Goal

Wandering Progress Monitor

Handle Blockage Goal

Local Unknown Goal

Global Unknown Goal

Unknown Area Goal

Unknown Area Monitor

Local Area Known Goal

Local Area Known Monitor

Try Opening Goal

Verify Opening Goal

Verify Opening Monitor

Opening Goal

Opening Monitor

Wall Goal

Wall Monitor

Corridor Goal

Corridor Monitor

Corridor Direction Goal

Corridor Direction Monitor

Corridor Landmark Goal

Corridor Landmark Monitor

Turn 360 Goal

Turn 360 Monitor

4.3 Software Modules

This section describes (briefly) each of the major software modules, and who is (are) the primary maintainer(s).

Each of these modules can be found in the directory: `/afs/cs/project/robocomp/xavier/src`.

4.3.1 Common

The common module contains useful utility code that is used by the other modules. If you develop some generally useful utility that is not specific to a particular module, it should be placed in the common directory. The code is made available in `libcommon.a` and `libezx.a`.

The directory currently includes the following utilities:

- ezx : A set of routines that provide a simplified interface to the X11 windowing system. These routines can be mixed with direct calls to the X11 primitives.
- LIB-math : Some useful math utilities.
- lists : A list data structure and associated routines.
- vectors : A 2d vector data structure and associated routines.
- getbool : A routine to prompt the user for a boolean response that provides a default.
- libc.h : This is a header file that provides ANSI C definitions of system calls and routines in libc.a.

Maintained by Everyone.

4.3.2 Controller

The controller module provides module to run the robot and perform behaviours. These abilities are made available to other modules through TCA interfaces. Devices are grouped by function. CTR (section 4.2.1) provides access to the base, the sonar and the laser. It implements behaviours involving moving the robot such as obstacle avoidance and wall following. CTV (section 4.2.3) provides access to the robot's vision system which includes the camera and the pan/tilt head. It implements low level vision routines. CTS (section 4.2.2) provides an interface to the speech synthesizer. CTA (section 4.2.4) provides access to the arm.

The controller module bundles related device interfaces and provides a TCA interface.

Maintained by Rich Goodwin and Joseph O'Sullivan.

4.3.3 DBMalloc

DBMalloc is a debugging version of the standard system routines for memory management. It can be vary helpful when trying to track down memory problems. You can compile xavier code with this library by specifying "`DBMALLOC=dbmalloc`" on the command line call to gmake.

The code was not written locally. The following description comes from the man page for dbmalloc.

This malloc library is a replacement for the standard library to be used during software development and debugging. See the standard malloc(3) pages for more information on the use of the following functions: `calloc()`, `cfree()`, `free()`, `malloc()`, `realloc()`

This library differs from the standard malloc library in the following ways:

1. Each malloc segment contains a magic number so that free can verify that the pointer passed points to a valid malloc segment.
2. Each malloc segment is filled with a non-zero pattern so that code that depends upon malloc segments being null will fail.
3. The size of each segment will be at least 1 byte larger than requested and the extra bytes will be filled with a non-zero pattern. When free is called, it will verify that you did not go beyond the number of bytes you asked for.
4. When a segment is freed, it will be filled with a different non-zero pattern to ensure that the program doesn't depend upon the use of already freed data.
5. Whenever any of the string or memory functions (`str`, `b`, `mem`) are called with a pointer that is within the malloc arena, the operation is checked to verify that it does not overrun the malloced segment. A failure of this check is considered a "warning level error" (described later) and is handled accordingly.
6. Run time checking can include verification of the malloc chain at each and every call to one of the malloc functions or manually by calling the `malloc_chain_check` function.

When a problem is found, the following error message is displayed:

```
MALLOC Warning from funcname()
(called from filename.c line ###):
Warning message goes here
```

`funcname` is the name of the function that has found the problem and will usually be an entry point into the library. The information that identifies where the function is called from will only be available if the source module was compiled with the `malloc.h` file included.

If the error is caused by a problem in the malloc chain and the offending chain element can be identified, the following information is also displayed (NOTE: this is just a guess by the software, it may not be the actual culprit):

```
This error is *probably* associated with the
following allocation:

   A call to malloc for 33 bytes in program.c
                                   on line 834.
   This was the 172nd call to malloc.
```

The above example assumes that `program.c` #included the debugging library `malloc.h` file. If it did not, the identification information will be as follows:

This error is *probably* associated with the following allocation:

```
A call to malloc for 33 bytes in an unknown file.
   This was the 172nd call to malloc.
```

The identification of which call to malloc is associated with the problem is helpful in that it gives you the information necessary to set the breakpoint on the allocation function for that particular invocation (breakpoints usually can have counters associated with them). The counters for the three primary allocation entry points (malloc, calloc, and realloc) are managed separately.

Not maintained locally.

4.3.4 EV-Grids

Maintained primarily by Jim Blythe and also Barry Belboz.

4.3.5 Grid-Map

The grid map module provides an implementation of a simple occupancy grid built up from sonar readings. It has been superseded by the evidence grip implementation in the ev-grids module.

Not Maintained.

4.3.6 Interface

The interface module provides interfaces to each of the devices on the robot. These interfaces also provide low level reactive behavior, like obstacle avoidance. Any low level behaviours that needs direct access to the hardware devices should be added to this layer. Care must be taken that any additions don't adversely affect other behaviours or the real time requirements of the device. Any new behaviours added should be made available via TCA to other modules.

Most robot applications access the devices through the controller via TCA. Only rarely will new behaviours have to be added.

Maintained by Rich Goodwin, Joseph O'Sullivan and Jim Blythe.

4.3.7 Learning

Maintained by Joseph O'Sullivan.

4.3.8 Mapping

Maintained by Reid Simmons.

4.3.9 Markov

Maintained by Sven Koenig.

The programs in the "markov" directory implement two different, but complementary functionalities: probabilistic position estimation and passive distance learning.

4.3.9.1 Probabilistic Position Estimation

This module performs deadreckoning in the absence of complete metric information about the environment and in the presence of noisy effectors and sensors. It maintains an estimate of Xavier's position in form of a probability distribution over all possible locations of the robot.

The position estimation module works as follows: It discretizes the topological map and compiles it, together with common sense knowledge (such as: office doors are usually more then 3 and less than 100 feet apart) and other initially known, although possibly uncertain metric information, into a Markov model (to be precise: a Markov Decision Process model). This has to be done only once. While Xavier operates, this Markov model is used to project Xavier's location after every incoming action or high-level sensor report. High-level sensors, such as a "left wall sensor," are virtual sensors that are derived from the occupancy grid - they collect different information from Xavier's physical sensors over time.

The output of the position estimation module is used by the passive distance learning module to refine the Markov model and by various high-level planning modules, for example to derive plans for reaching a goal location or localizing the robot (minimizing the uncertainty about the robot's location).

4.3.9.2 Passive Distance Learning

This module learns and improves distance information, such as how long certain corridors are. It does not assume that it has control of Xavier at any time; it learns purely by observing the actions that Xavier executes and the sensor information that it receives in return. The module operates in the presence of noisy sensors and inaccurate dead reckoning. It maintains (and periodically updates) estimates of the distances between any two adjacent nodes of the topological map in form of probability distributions over all possible lengths.

The passive distance learning module uses an almost pure Bayesian approach. The prior probability distributions over the lengths are derived from the Markov chain. They are then updated using both the position estimates provided by the position estimation module and the high-level sensor reports.

The output of the distance learning module is used to update the Markov model used for position estimation. Since the position estimation module provided information that was used for learning the distances in the first place, we use an EM algorithm: Given an execution trace, we use the position estimation module to predict Xavier's location at every point in time, which allows us to use the distance learning module to improve the distance estimates. We use this distance information to compile a more accurate Markov model, and can then use the improved Markov model on the same execution trace to predict Xavier's locations again, etc.

4.3.10 Navigation

Maintained by Reid Simmons.

4.3.11 QLattice

Maintained by Reid Simmons.

4.3.12 Qual-Map

Maintained by Reid Simmons.

4.3.13 Simulator

The simulator module provides the code to simulate the robot at the device level for all devices except the camera. The camera is simulated at a higher level and provids the same interface as the vision primitives implemented in the interface module. The simulator module also provides code to read and display parameter files that describe a floor plan and the initial position of the robot. This is currently used by the navigation module to display the believed position of the robot and its intended path.

Maintained by Rich Goodwin, Joseph O'Sullivan and Reid Simmons.

Described in more detail above, in section 3.

4.3.14 Tasks

The tasks module is a collection of high level robot controllers that were implemented for the AAAI robot competition and various demos.

pickBoxDemo : The demo where the robot can be directed to find and pick up boxes with different types of marks.

pplan, dynprog2 : Sanjiv's fast dynamic programming path planner.

event1 : Event 1 from AAAI93. Escape from office by watching the doors.

event3Seq : Full blown version of the box manipulation task. Never worked.

find-door : Program to locate left and right edges of the door by scanning forward +/-45 degrees using laser.

fixate : Have the robot fixate the camera on a specific location as it moves around.

get_grid : Get the old style occupancy grid.

idenbox : Laser code to identify the corner of a box.

laser_switch : Example of how to switch between two lasers.

mapping, mapping2 : Used old grid map routines to build an occupancy grid.

marker_map : Occupancy grid for markers.

processSpeech : Interpret input from the speech recognizer.

say_speak : Have to robot repeat what you say to the voice recognition system.

sentry : Simple interactive program to notice the approach of people (using sonar) and to turn cameras to face them and say appropriate phrases.

move-examp, move2-examp, status-examp : Simple TCA examples.

wander : Wander around.

Maintained on an emergency only basis by Rich Goodwin, Joseph O'Sullivan and Reid Simmons.

4.3.15 Vision

Maintained by Joseph O'Sullivan.
Described above, in section 4.2.3.

4.3.16 Zephyr

Maintained by Soshi Iba.
`xavz` is a Zephyr interface for Xavier's navigation module. It is capable of answering basic questions such as position inquiry and can also deliver requests. Capabilities such as sending out emergency zephyrs have not been adopted; similarly there are many other thing zephyr is capable of doing, but have also not been implemented yet.
If you have a specific task in mind, send requests to Soshi Iba.

4.3.16.1 Instructions

How to use `xavz`:

1. Login as 'robocomp' to either `ariel.xavier` (or `charles.xavier`). Basically, you need a machine that runs zephyr. Otherwise, `xavz` won't start.
2. `setenv CENTRALHOST <somewhere>`
3. Start simulator, CTR, and navigate. Use a param file with roomnames in it. (`navigate -q <param file>`).
4. Then start `xavz` on ariel or charles. No parameters are needed for `xavz`.
5. From here on, if you zwrite to 'robocomp', it will respond as if it is the robot. Your name must be on xavz_valid_user_table (which is in `xavz.h`) to communicate with Xavier. Some example messages you can send are:

   location : 'where are you'; 'locate yourself'; 'which room are you in'

   move request : 'come to weh5302'; 'go to wean 5101'; 'come here'; 'goto bh211'

   job queue : 'show queue'

   yes/no : 'yes'; 'yeah'; 'no'; 'nop'

6. To add more words that Xavier can understand, go to `xavz.h` and add sentences into parse tables.

   To add more functionalities, add the appropriate language parse table to `xavz.h`, and change code in the `process_msg()` function within `xavz.c`

   For example, if you wanted Xavier to deliver your mail to you, you should add the following table to `xavz.h`:

```
char *deliver_mail_table[] = {
  "deliver to weh @@1",
  "deliver to wean @@1",
  "deliver to @@1",
  0 };
```

   In `process_msg()` you need to add something similar to the following lines:

```
parsed_str = parse( msg_lwr,
                deliver_mail_table);
if (parsed_str) {
  do_all_the_stuff_to_deliver()
}
```

   As a note: if you have a suggestion for a more efficient way to add functionalities, please contact the maintainer.

4.4 Writing a Program for Xavier

4.4.1 A simple example

If you need to do tests in which you have to move Xavier a number of cm, wait for it to finish the movement, and do something after that, then the TCA commands implemented in CTR are the thing that you are looking for. Here we present an explanation about them, and about how to use TCA. The source code is in:

```
/afs/cs/project/robocomp/xavier/
   src/controller/example1.c
```

4.4.1.1 The code

This program moves Xavier forward 20 cm, then turns it 45 degrees, and then asks for and prints the current position.

```
#include <stdio.h>
#include "CTR-messages.h"
#include
 "/afs/cs/project/robocomp/tca/include/tca.h"

main()
```

```
{
    double   cm, degrees;
    int      status;
    POS_TYPE position;

    printf("Connect ...\n");

    /* Tell TCA that you are ready */
    tcaConnectModule("Example A",
                     tcaServerMachine());

    /* Wait for tca to be ready */
    tcaWaitUntilReady();

    cm = 20.0;
    status = tcaWaitForCommand("C_move",
                               &cm);
    if (status == Success) {
      /* Yeah */;
    } else {
      /* Failure returned */
      /* Could retry, or plan alternative */;
    }

    degrees = 45.0;
    status = tcaWaitForCommand("C_turn",
                               &degrees);
    if (status == Success) {
      /* Yeah */;
    } else {
      /* Failure returned */
      /* Could retry, or plan alternative */;
    }

    status = tcaQuery("Q_position", NULL,
                                    &position);
    if (status == Success) {
        printf ("X:%f, Y:%f, Orientation:%f\n",
                position.x, position.y,
                position.orientation);
    } else {
      /* Failure returned */
      /* Could retry, or plan alternative */;
    }
    printf("Finished\n");
}
```

4.4.1.2 Compiling

gcc -o example1 example1.c -ltca
-I/afs/cs/project/robocomp/xavier/include
-L/afs/cs/project/robocomp/tca/lib

You probably want to modify the `GNUmakefile` to avoid retyping these commands ad nauseum:

```
...
SRCS = example1.c ...
...
_OBJECTS = example1.o
...
$(BDIR)/example1: $(ODIR)/example1.o $(LIBDEP)
$(LINK.c) -o $@ $(ODIR)/example1.o $(LIBRARIES)
```

4.4.2 A more complex example — Interrupt Handlers

This piece of code shows how to install a interrupt handler for the CTR_SONAR_STATUS_BROADCAST signal.

```
#include "tca.h"
#include "CTR-messages.h"

void myHndFunction (TCA_REF_PTR ref,
                    SONAR_STATUS_PTR sonar)
{
  /* Deal with sonar readings here */
  /* See CTR-messages.h for SONAR_STATUS_PTR
     format */

  tcaFreeData(tcaReferenceName(ref), sonar);
}

main()
{
  tcaConnectModule("MyModule",
                   tcaServerMachine());

  /* Tell TCA when this handler should be
     called */
  tcaRegisterHandler(
        CTR_SONAR_STATUS_BROADCAST,
        "My_handler", myHndFunction);
  tcaWaitUntilReady();

  /* loop forever in the main device loop, which
     collects and processes input from all
     devices.  You can put your own code here.
  */
  devMainLoop();
}
```

This file is compiled with the same commands as `example1.c`.

4.4.3 Running On the Simulator

The following series of commands will run `example1` on the simulator:

```
[start TCA]
machine1> setenv CENTRALHOST machine1
machine1> cd /afs/cs/project/robocomp/tca/bin
machine1> central 2

[start Xavier, another terminal]
machine2> setenv CENTRALHOST machine1
machine2> cd /afs/cs/project/robocomp/xavier/bin
machine2> xavier ../params/event1.param
```

```
[start CTR, another terminal]
machine3> setenv CENTRALHOST machine1
machine3> cd /afs/cs/project/robocomp/xavier/bin
machine3> CTR -x -s machine2

[start example1, another terminal]
machine4> setenv CENTRALHOST machine1
machine4> example1
```

4.4.4 Running On Xavier

The following series of commands will run `example1` on Xavier:

```
[start TCA]
machine1> setenv CENTRALHOST machine1
machine1> cd /afs/cs/project/robocomp/tca/bin
machine1> central 2

[start CTR, another terminal]
machine2> setenv CENTRALHOST machine1
machine2> cd /afs/cs/project/robocomp/xavier/bin
machine2> CTR -x

[start example1, another terminal]
machine3> setenv CENTRALHOST machine1
machine3> example1
```

In both environments, you can quit from `example1` and restart it again without restarting the rest of the system.

4.5 Trouble-shooting

See forthcoming manual.

4.6 On writing Xavier Software

There is a methodology to the Xavier hierarchy. The software to control and interface to the devices on the robot is organized into three layers.

- **Interface Level** The direct interface to the hardware. It provides functions to control the hardware (eg. set velocity of the base, start moving forward ...). This layer will not be considered in this document.
- **Behaviour Level** Uses the low level hardware routines to provided "behaviours." These are routines that provided a higher level of functionality (eg. follow a path, avoiding obstacles, find targets that are currently visible. ...) This layer will also not be considered specifically in this document, but the CT* levels subsume the code.
- **CT* Level** Connects behaviours to TCA messages and provides a user interface. It includes the "main" procedure for a process.

Two key directories are the `interface` subdirectory, which contains the low-level device control, and the `controller` subdirectory. These are the top level routines that interface between the device and the rest of the system providing high level functionality. This is subdivided into a "behaviours" layer that implements more functionality and an interface level that connects to TCA.

4.7 Compiling and Maintaining your own version

If you plan to modify Xavier code , then you'll need your own copies in your own directory. In order to allow several people to simultaneously access and amend Xavier's source libraries independently, CVS was chosen to maintain authoring revision policies.

We have set up shell scripts and makefiles that take care of the routine aspects of using cvs and compiling code. Most people will not have to learn any of the details of how to use cvs to develop and maintain Xavier code. The following sections outline how to do the basic steps in compiling, modifying and adding new code.

4.7.1 Getting Started

If you want to modify some Xavier code, you will need a local copy. The basic idea is that you get a copy, work on it until you are happy that the changes you have made are working and then release it so that other people can use it. We can think of the software development process as a co-op where everyone is adding to the base of software and functionality and we can all benefit from and use each others code. To keep additions compatible and to allow better code sharing, we all have to be working on the same base of code.

There is a script that you can run to get copies of some or all the of Xavier code and documentation. The script will prompt you for which directories to get copies of. If you later decide that you don't want a copy of a directory, simply delete it with -r -f ¡dir¿ If you decide that you need additional directories or subdirectories, run the script again. It will allow you to get additional directories (or update existing ones) without replacing or destroying any existing directories or the changes you have made.

Suppose H.Q. Bovik want to get a copy of the navigation code directory to modify. Note '`machine>`' is the shell prompt and indicates something you enter into the shell.

```
machine> cd /afs/cs/user/bovik
machine> /afs/cs/project/robocomp/xavier/etc/setupXavier
Is  /afs/cs/user/bovik the correct directory to put Xavier files in?
Y or N (y)
y
Release directory is  /afs/cs/project/robocomp/xavier
Checkout module xavier/doc ?
Y or N (y)
n
Checkout module xavier/etc ?
Y or N (y)
n
Checkout module xavier/params ?
Y or N (y)
n
Checkout module xavier/src ?
Y or N (y)
y
Checkout module xavier/src/aaai94 ?
Y or N (y)
n
...
Checkout module xavier/src/navigation ?
Y or N (y)
y
/afs/cs/project/robocomp/xavier/etc/xcvs checkout xavier/src/navigation
cvs checkout xavier/src/navigation
U xavier/src/navigation/GNUmakefile
U xavier/src/navigation/README
U xavier/src/navigation/distance.c
U xavier/src/navigation/distance.h
...
Checkout module xavier/src/qlattice ?
Y or N (y)
n
...
/afs/cs/project/robocomp/xavier/etc/xcvs checkout xavier/src/README
cvs checkout xavier/src/README
U xavier/src/README
/afs/cs/project/robocomp/xavier/etc/xcvs checkout xavier/src/GNUmakefile
cvs checkout xavier/src/GNUmakefile
U xavier/src/GNUmakefile
/afs/cs/project/robocomp/xavier/etc/xcvs checkout xavier/README
cvs checkout xavier/README
U xavier/README
/afs/cs/project/robocomp/xavier/etc/xcvs checkout xavier/GNUmakefile
cvs checkout xavier/GNUmakefile
U xavier/GNUmakefile
done.
```

In the setup we are using, you can only checkout entire modules or submodules (cvs terminology for directory/subdirectory). This is good because it makes sure that you keep all the code in a directory consistent. When you checkout code, only checkout the modules you need. You can always checkout other modules later.

4.7.2 Compiling Code

The xavier code needs to run on a number of machines and operating systems. One problem that we have had in the past is that each system has its own version of "make" with incompatible extensions. In order to avoid this problem, we have switched to using the gnu version of make, gmake. This allows us to create GNUmakefiles that can be used on all the machines and OS's that we support.

The directory for each modules, such as src/navigation, contains the code files for related libraries and programs. In the case of the navigation directory, it is code that deals with high level navigation of the robot from one location to another in a building. Modules can contain "public" programs an libraries that are exported for use by other modules and for general use in running the robot. In addition, each module can have private programs and libraries. These are generally used for testing or debugging the code in the module and don't have to be compiled to use the module and should not be exported to other modules.

In the GNUmakefile for a module, there is a list of the "PUBLIC_BINS" "PUBLIC_LIBS" and "PUBLIC_INCS". To compile the code for a module, use the command "gmake all". This will copy the "PUBLIC_INCS" to your xavier/include directory, compile the libraries and install them in your xavier/lib directory and compile the programs and install them in your xavier/bin directory. The "PUBLIC_INCS" are the include files that the module exports as its interface for other modules to include. Any code that is exported to other modules must be included in a library. When compiling your code, the compiler will first look for local copies of the include files and libraries. It will then look in the release directory in "/afs/cs/project/robocomp/xavier". If you are modifying multiple modules, you will want to make sure that you are getting the latest copies of the include and library files. The GNUmakefiles work recursively, so you can issue the "gmake all" command in your xavier/src directory to recompile all the modules you have copies of. The GNUmakefile ensures consistency by updating the include files before the libraries are compiled compiling the libraries before the programs are linked.

To compile the private programs and libraries for a module, use "gmake everything". The private programs will be put in your xavier/src/module/bin directory.

Note that you must be running X11-R5, and if you get errors saying `X*.h` file not found, you can automatically include this version by using the command:

```
/usr/cs/etc/modmisc - -release beta cs.misc.X11
```

Similarly, you may get an error message referring to an internal compiler error. Check which version of `gcc` you are using by typing the command "`gcc -v`". If the version number is less than 2.4.5, then you should type the command:

```
/usr/cs/etc/modmisc - -release alpha
                                cs.misc.gnu-comp
```

to get an up-to-date version.

4.7.2.1 Automatic Dependency Generation

The program "makedepend" is used to automatically create dependency files for C code by examining the "#include" directives in the *.c files. These files tell gmake which files need to be recompiled when header files change. Issuing the command "gmake depend" will generate the file: "GNUmakefile.¡machine¿". Since each machine and OS have different locations for the system include directories, system specific depend files are created. If the dependency file is missing, you will get a warning when you compile.

NOTE: As of this writing, the makedepend programs on the i386_mach machines and the pmax_mach machines don't work correctly and core dump.

4.7.2.2 Compiling for Multiple Machines

AFS, the Andrew file system, is used to allow compilation for multiple machines from the same source directory. The GNUmakefile programs detect if the code directory is on afs and creates symbolic links for the bin, lib and obj directories that use the "@sys" expansion. This is useful for interleaving of debugging code using the simulator on a sunos machine with running the code on the i386 machines on the robot.

The GNUmakefiles also allow for machine dependent programs and libraries. For example, the code that controls the video digitizer can only run on the i386 machines because it requires the video digitizer hardware. For example, the private test program "recognize_camera" uses the camera to recognize special targets. It is compiled only for the i386 machines by putting "recognize_camera" on the "PRIVS_Intel" list rather than on the "PRIVS" list of programs. The same thing can be done for public bins and libraries.

4.7.2.3 Updating Your copies

When you have a local copy of a module, you will periodically want to update it to get any changes that other people may have made and saved to the repository. This will allow you to get bug fixes and extensions created by others. To get the changes, use "gmake update". This command works recursively to update any subdirectories. The cvs command will print a list of updated and locally modified files. Files preceded with "U" are updated from the archive. Files preceded by "M" are locally modified. When a file have been modified in the archive and locally, cvs will attempt to merge the changes and will print out a message about merging the changes. If there are overlaps, then a message about conflicts will be printed and cvs will put both sets of changes in the file, as indicated below.

```
<<<<< 2.1
your changes
======
others changes.
>>>>>>
```

You have to edit the files with conflicts and manually merge the the changes. Conflicts usually only happen if two people modify the same lines.

To find out the status of the files you have, you can issue the "gmake status" command, although this provides minimal information. The "gmake diff" command will give the lines that you have changed locally and the set of lines that you and others have changed since your last "gmake update". It can be useful when you want to see what you have changed recently.

4.7.2.4 Committing Changes

The "gmake commit" command will take any changes that you have made in the current directory and subdirectories and save them to the central repository. In addition, the release directory will be updated and compiled. The cvs commit command first checks to see that your copy is up to date. If it is out of date, then you must do a "gmake update" and resolve any conflicts before you can save your changes. Before saving the changes, cvs will open a file using the editor specified in your environment variable "EDITOR" for you to add comments. The file will include a number of lines that start with "CVS" and indicate which files have been changed, added or deleted. You add a comment at the top of the file that describes the changes you have made and exit the editor. The "gmake commit" command should be issued from the command line because the cvs command may prompt for input and you can't input anything to the "*compilation*" buffer in gnu-emacs.

Note that you can commit specific files by typing "`xavier/etc/xcvs commit filname`". It is usually better to commit all the changes in a directory at once to make sure everything is consistent. The only way to "uncommit" changes is to undo the changes in your file and then recommit.

4.7.2.5 Adding a new Program

To create a new program, you simple create the new .c files and the new .h files and modify the GNUmakefile. Add any new .h files either to the PUBLIC_INCS or PRIVATE_INCS list. Add any new .c files to the SRCS list and any new .o files to the OBJS list. In the "User Defined Targets" section of the GNUmakefile, add the instructions for compiling the new program or library. Follow the examples given in the comments. The command "gmake add" will then create the information needed to add the new files to the archive. When the program is ready to be released, issue the "gmake commit" command from the shell.

4.7.2.6 Adding a new Module

Whenever you are creating a new program or library that does not fit into one of the existing modules, you should create a new module. Simply create the new subdirectory in the src directory and add a GNUmakefile. Use one from another module as a guide. Then add the name of the new subdirectory to the list of subdirectories in the GNUmakefile in the src directory. To add the module, use "gmake add" in the src directory. Then a "gmake commit" in the src directory will add the new module to the archive and update the release directory.

4.7.3 Gmake targets

The following is a brief description of the targets available when using gmake.

add: Schedule any new source, header or other files for addition to the cvs achieve. The files are added using the commit target.

all: The default target. It installs the public include files, compiles the public libraies and the public programs. It is a synonym for install.

install: Same as all.

everything: Same as all with the addition of compiling the private and test programs.

clean: Remove junk and object files from the directory.

clean_depend: Remove the dependencies generated by makedepend.

commit: Commit any changed or new files to the cvs archive and update the release directory. The updated release directory is then compiled.

depend: Generate dependencies using makedepend.

diff: Use cvs to get a list of changes to the local copy and to the archive since the last update of the local copy.

status: Get the status of local files. The status information includes the revision number and whether the file is out of date, locally modified or up to date.

update: Merge any changes in the archive with the local copy of the files.

veryclean: The same as clean with the addition that the program and library files are removed.

4.7.4 Gmake Variables

The following is a brief description of the targets available when using gmake.

PUBLIC_BINS: The set of programs to be compiled and put in the `xavier/bin` directory. These are intended for general use.

PUBLIC_LIBS: Libraries exported for use by other modules. These are installed in the `xavier/lib` directory.

MODULE_LIBS: Like public libraries, but with a module qualifier. There are installed in the `xavier/lib/<module>` directory. Not currently used.

PUBLIC_INCS: These are c header files that are exported for other module to use. They are copied to the `xavier/include` directory so that other modules can include them. Modify the files the the `src/<module>` directory and not the ones copied to the `xavier/include` directory. The copied files are write protected to help prevent inadvertent modification.

PUBLIC_DOCS: There are documentation files that are to be installed in the `xavier/doc` directory.

MODULE_INCS: Like PUBLIC_INCS, but installed in the `xavier/include/<module>` directory. Can be useful when include file names conflict. Similarly to the `include/X11` directory structure used for X11 includes.

MODULE_DOCS: Documentation for a module that is installed in the `xavier/doc/<module>` directory.

PRIVS: Private programs for testing or debugging. These are only compiled in the `src/<module>/bin` directory with the command "gmake everything".

OBJS: A list of object (.o) files. Used when cleaning the directory.

SRCS: A list of source (.c) files. Used to add new files to the archive.

PRIVATE_INCS: A list of include files that are not exported for public use.

MISC: A list of files that are to be included in the archive but are not source, header or documentation files.

DELETES: A list of files to delete when the directory is cleaned.

4.7.5 CVS

This section contains a bare bones description of how the CVS software version management system works. If you want more information do '`man cvs`' or look in '`doc/cvsGuide.fm`'. The cvsGuide contains a tutorial, but was written for the Erebus project.

There is a central archive that stores the current and previous versions of all the source code. This directory is:

/afs/cs/project/robocomp/Master

DO NOT change any files in this archive directly.

Each user maintains their individual "working" copy of the source code. Each person only changes and edits these copies. Periodically, one synchronizes with the archive to get the latest versions of other peoples stuff and to save ones changes. If more than one person changes the same file, then the system will attempt to merge the changes. If it has problems, it will tell you. You must edit any problem files and do a manual merge.

There directory `/afs/cs/project/robocomp/xavier` contains the current release of the software. Don't modify these directories. This directory is updated automatically when changes are committed to the archive.

If you have any questions, check the man page, the `cvsGuide.fm` or send email to `robocomp@cs.cmu.edu`.

Index