STR 96-026

SNO Source Manipulator Control Code

T. J. Radcliffe
Department of Physics,
Queen's University at Kingston,
K7L 3N6, Canada

1 Introduction:

This document describes a set of **Hardware**-derived classes for control of the SNO calibration source manipulator. It is intended as a guide to the code, which is heavily documented internally, rather than an exhaustive description. Please see A HARDWARE CONTROL CLASS HIERARCHY for details on the **Hardware** classes. The calibration source manipulator is an over-constrained physical system: when moving in a single plane it has three ropes attached to it, and when moving out of plane it has five ropes attached to it. The primary task of the control code is to deal with possibly conflicting constraints as intelligently as possible. Note that the way dealing with this problem naturally incorporated the possibility of out-of-plane motion: nothing has been added to the code to support this capability.

A secondary task of the code described in this document is to communicate with the Data Acquisition computer (DAQ.) This is done via a TCP connection over an Ethernet line. The connection must be able to carry commands from the DAQ to the calibration computer and return status information from the calibration computer to the DAQ. The DAQ is a Macintosh, the calibration computer is an IBM PC clone.

The **Hardware** class hierarchy does not support multiple inheritance, so the class hierarchy described here, which is a set of **Hardware**-derived classes, consists of classes that contain each other rather than inherit from each other. For the most part this is appropriate: an **Axis** contains a **Motor**, an **Encoder** and a **Loadcell**, rather than IS a **Motor**, an **Encoder** and a **Loadcell** . The term "class hierarchy" is used throughout to include both "has a" and "is a" relationships.

Section 2 describes the philosophy behind the classes developed for manipulator control. Section 3 describes the sort of thing the system is intended

to do, including some of the things it can't yet do, but should. Following this is a section which gives an overview of the code structure. After this comes series of sections dealing with the **Hardware**-derived classes in descending order of abstraction, starting with the **PolyAxis** class and winding up with the low-level classes that deal directly with boards and chips. Section 18 discusses the main program that puts all this stuff together at the moment. Section 15 describes the **Server** class and the work that still needs to be done on it. Section 19 describes the Borland C++ IDE settings required to compile the code. Section 16 describes various utility functions for doing things like dealing with object databases. Section 20 deals with the oft neglected but ever important subject of co-ordinate systems. The final section discusses changes and improvements that are still required before the code goes underground.

Note on capitalization etc.: there is a widespread practice in object-oriented programming circles for the first letter of class names to be upper case, and the first letter of object names to be lower case. This is a practice that is slowly being built into the existing **Hardware**-derived code, and I heartily encourage everyone to follow it. It may be mindless conformance to arbitrary, socially defined conventions, but then again, so is speaking English. In this document, class names will be mostly **bold face** and function names will be mostly *italic*. Code examples and variable names will be `typewriter`. I tend to use "class" and "object" interchangeably, although one is collective and the other is singular.

Note on standards: C++ is not very standardized just yet, and all of this code was meant to run on IBM PC compatibles when compiled with the Borland C++ compiler. Parts of it have also been complied with the GNU g++ complier version 2.6.8 under LINUX, and I found that a few minor changes had to be made to do this. When I refer to "ordinary C++" I mean whatever the people at Borland think of as ordinary. If you try to port this code to another platform or even another compiler on the same platform, you may find a number of errors due to small differences in standards.

2 Code Organization and Philosophy

The code is organized in a fairly hierarchal way on the basis of how abstractly each class interprets the raw bits passed to it from the hardware or from classes that access the hardware. Looked at from a reductionist point of view, everything a computer deals with is "really" a stream of bits, in the same way everything around us is "really" a collection of atoms, uh, nucleons and electrons, uh, quarks and leptons, uh, supersymmetric technicolour

monads.... I put "really" in quotes because I think this is a stupid idea: stuff exists, and how we analyze it into concepts is up to us (in technical terms, I am a non-eliminative reductionist.) No particular level of abstraction or reduction is privileged. A stone is neither more nor less real than the atoms it is made of, as anyone who has stubbed a toe knows. The point of all this is that we are free, within certain constraints, to choose how we interpret the bits in a register or memory location. Different levels of abstraction are the result of interpreting the bits in different ways: low levels treat them as bits (possibly related to some chip or board-level function,) and higher levels treat them as being meaningfully related to some physical quantity like the pressure being measured by a transducer. Because no level of abstraction is privileged, there is no "one right choice" for dividing the code into abstract levels, and so some kind of objective design principles are needed to guide our decisions about where to put what kind of object in an abstract hierarchy.

The principles used in designing classes for controlling the manipulator result in roughly four levels of abstraction:

Level 1 classes that deal directly with hardware registers and treat register values simply as collections of bits

Level 2 classes that deal with hardware registers, but which impose an abstract meaning on the register values

Level 3 classes that deal with hardware registers only through Level 1 or 2 classes, and typically convert register values to physically meaningful numbers

Level 4+ classes that deal with hardware only through Level 3 or higher classes, and whose main function is to co-ordinate the actions of several Level 3 or higher objects.

Level 1 classes are those that represent a single chip or board, such as the AM9513 timing controller chip or the TIO10 board. These classes don't care what the values they write out to the boards mean apart from board-level or chip-level functionality. They contain no information that is in any way dependent upon the USE to which the board or chip may be put.

Level 2 classes also deal with boards and chips directly, but they have an awareness of the meaning of various inputs and outputs. Thus, a Level 1 class might read a register and put the value into a variable called `register1` but a Level 2 class would put it into a variable called `counter` if it was a counter value. Level 2 classes impose a meaning on the bits themselves without applying any transformation to them. Thus, a read from a particular

memory location may be recognized as an ADC output, but no commitment is made as to what might be going into the ADC input.

Level 3 classes typically describe single pieces of external hardware, such as a motor or a position encoder or other instrument. They contain Level 1 and 2 objects to access the hardware, as well as higher level information about the meaning of bits sent and received from those objects. Thus, the output of a counter may be interpreted as a shaft position, for instance, which allows other objects to ask the encoder sensible things like "What position are you reading now?" rather than "What are the bytes in register base+offset, uh, how do I convert those to a floating point value that represents position relative to, uh, what was the zeropoint again....?" Level 3 classes implement TRANSFORMATIONS – via some calibration constants – of raw input data into physically meaningful terms.

Level 4+ classes describe systems of Level 3 objects or above. Two examples from the manipulator control classes are the **Axis** class and the **PolyAxis** class. Each **Axis** consists of a **Motor** an **Encoder** and a **Loadcell** as well as various state information. A **PolyAxis** consists of an array of **Axis** objects and related state information. The **Encoder** and **Loadcell** classes themselves are Level 3 classes.

The notation "Level 4+" is refers to all classes of Level 4 or higher; at this point in the hierarchy a class's level is one higher that of the highest-level class it contains. It is a very bad idea to create classes that cannot be assigned an unambiguous level. If two classes contain references to each other it is impossible to assign them to a level within this scheme because they can't both be one level higher than the other. This means you have created two structures that describe a single concept, and you should think about re-writing them so the dependencies sort themselves out at lower levels.

Level 4+ classes are where the interesting parts of the control algorithm is typically implemented. Level 3 classes can do things like set a single motor moving, and perhaps tell it to stop after a predetermined time or number of steps, but they cannot implement feedback control because they only describe a single bit of hardware: one can't describe both a motor and a position encoder, for instance, or it would be a Level 4 class.

One could break this scheme, and write a single class that accessed both a motor and a position encoder simultaneously, but this would radically reduce the modularity of the code. It would become impossible to use the code to control a motor in the absence of an encoder (which is useful at least for debugging purposes) and impossible to use an encoder without a motor. Low coupling, or high modularity, is one of the most powerful aspects of object-oriented programming, and the levels described here can serve as a

set of design principles that should ensure that the highest possible degree of modularity is retained.

Note that some low-level classes were written before the level scheme described here was imposed on the code, and so there may be minor deviations from this scheme in those classes.

3 Desired Functionality

The purpose of the manipulator control classes is to allow a source to moved around inside the SNO detector at the request of the DAQ system, to report the status of the source back to the DAQ system upon request, and to ensure that the chance of damage to the detector or loss of a source inside the vessel is the minimum possible. The two basic disasters that can happen are the tension becoming too large on a rope and a rope going slack. The first of these is of relatively minor concern from the point of view of software: the mechanical design of the system is intended to prevent excessive forces from acting on the acrylic vessel; the motors are supposed to stall well before the tension is in the danger zone. This has not been tested.

A rope going slack is the major danger, as it can lead to tangling, which may prevent a source from being lifted out of the vessel. This would be very bad. A set of air-cylinder tensioners has been added to the mechanical system to help prevent this, but I am still not convinced that it is adequate to do the job. The mechanics of the system must undergo many weeks of reliability testing prior to going underground. The control algorithm, described in detail in Section 6 is intended to be a backup to the mechanical systems that prevent slack ropes. Ideally, one would like a tensioning system that only came into play when the tensions were near the allowed bounds. The air-cylinder system does not do this: it looks like a spring with a discontinuously variable spring constant. When a source is stationary the air-cylinder system is in equilibrium. When the manipulator starts to move a source it increases the tension on some ropes, decreases it on others. This causes the air-cylinders to move. In their simplest mode, the air cylinders look like constant force devices within the bounds of the limit switches, although the finite reservoir size gives them some spring-like characteristics as well. In a typical move the rope tensions change until the air-cylinder limit switches are reached, and then the fresh flow of air brings the cylinders back into the middle equilibrium position again. This looks to the control algorithm like a mysterious source of movement, as the air-cylinders are between the motors and the position encoders, and this may cause the system to generate a warning about the possibility of motor stall or a slipping encoder shaft.

The communications aspects of the manipulator control classes break into two parts: command processing and communications. The command processing part is handled by the **Hardware** framework as described in A HARDWARE CONTROL CLASS HIERARCHY. The communications part is supposed to be handled by the **Server** class, but this is not yet fully implemented.

4 Overview of the Manipulator Classes

A schematic of the manipulator control classes is shown in Figure 1. At the highest level is the **PolyAxis** class. As its name suggests it is for dealing with multiple **Axis** objects and co-ordinating their movement. An **Axis** object consists of a **Motor**, an **Encoder** and a **Loadcell** object and thus constitutes the primary object for feedback control of a single rope. The job of the **PolyAxis** object is to determine the position of the source and set the error inputs to the feedback algorithm in each **Axis** object. The **Axis** objects that constitute a **PolyAxis** are stored in an array which will typically have three or five elements for in-plane or out-of-plane motion respectively.

PolyAxis also contains an **AV** object, which supplies it with the current state of the acrylic vessel as measured by the AVPsense system. The hardware interface classes for this system are shown in a vapour cloud because they have not been written yet, as the DataConcentrator system for interfacing these boards into the PC is not yet complete. Note that when this hardware becomes available the **DigitalChannel** and **AnalogChannel** classes will have to be substantially re-written as well. It may be worthwhile to substantially increase the degree of modularity in these classes at that time. In particular, the card that fits into the PC backplane, the DataConcentrator backplane card and the DataConcentrator cards themselves should all have their own classes. This was not done in the current incarnation because it would just have to be redone completely in the next incarnation.

As mentioned above, the **Axis** class contains the Level 3 classes **Motor**, **Encoder** and **Loadcell**. Because the **Hardware** class hierarchy does not support multiple inheritance these subclasses are contained in – rather than superclasses of – the **Axis** class. They themselves communicate with the hardware via Level 2 classes like **AnalogChannel** and Level 1 classes like **TIO10**. There is at the moment only one **TIO10** card in the system. This is a general purpose National Instruments timing control card. It supplies both a globally accessible realtime clock and eight clock outputs for driving motors. The TIO10 board includes, a pair of Advanced Micro Devices

AM9513 timer/counter chips and a pair of Motorola MC6821 Peripheral Interface Adapter (PIA) chips. The **TIO10** class contains objects to describe these chips, and the **AM9513** class includes a set of five **AM9513Chanenl** objects that correspond the the five channels available on each chip. These chips and channels are referenced by other objects in the class hierarchy.

The purpose of the **Axis** class is to provide feedback control to a motor based on either error signals set by a **PolyAxis** object that contains that **Axis** or based on a simple feedback algorithm internal to the **Axis** class. The latter allows standalone operation. One of the basic constraints on **Axis** design was that no **Axis** object should have to know what any other **Axis** object is doing, thus maintaining hierarchy as described in Section 2. The **Axis** class implements different control algorithms depending on the control mode (i.e. PolyAxis or standalone). In standalone mode the algorithm is a simple position feedback algorithm that updates the number of steps it wants the motor to take until it gets close to the end, and then it just lets the motor complete the travel itself. The **Motor** object in this case is in step control mode, in which it tries to match the actual number of steps taken with the desired number, which is set by the **Axis** object. When the actual number matches the desired number the **Motor** stops itself.

When an **Axis** is in PolyAxis control mode it uses a fuzzy logic algorithm to control the speed of its motor. In this case the motor is in velocity control mode. The fuzzy logic algorithm, discussed in detail in Section 6 is a nonlinear PID controller; the fuzzy logic aspect is more of a design tool than an implementation tool in this case.

The **Encoder** and **Loadcell** objects deal with calibration constants as well as hardware registers. They access hardware via the **DigitalChannel** and **AnalogChannel** objects respectively. These Level 2 objects currently mix up different hierarchal levels fairly badly, and need to be re-written when new hardware is put in place in any case. A strong case can be made for breaking them up into separate PCcard, DCcard and DCinterface classes.

The **Server** class is shown at Level 3, as it accesses hardware via lower-level entities from the PCTCP socket library. It is part of the **Hardware** class hierarchy because it needs to be polled periodically to ensure communication with the DAQ system is maintained. At the moment the **Server** class is in an incomplete state: it appears to work in a standalone test setup, but does not behave properly in the manipulator control program.

The **Clock** class has global scope and is designed to return the time in seconds since the program started running. This time is a double precision real number. As time is a continuous global quantity the principle of verisimilitude argues that the object that supplies knowledge of it should also

be global and represent that quantity as continuously as possible. Once the **Server** class is made to work properly it may be worth adding the capability of referencing the **Clock** time to the GPS clock as seen by the DAQ. The **Clock** class uses two channels on the TIO10 board to count ticks of the tio10's 5 MHz clock. The **TIO10** object itself has to have global scope so that it can be defined prior to the **Clock** object.

There is also a global **Display** class that is not part of the **Hardware** class hierarchy as it is effectively polled at the system level. It serves as a standard output interface for all the **Hardware**-derived objects using various DOS screen-control functions. There are a number of helper objects, such as the **Keyboard** class, that deal with I/O and other management problems. Many of these are implemented in ordinary C subroutines.

The main program defines an **AV** object, a **TIO10** object and a **Clock** object as well as a **PolyAxis** object. The **PolyAxis** constructor takes care of calling constructors for all the lower level objects, as discussed below, but takes pointers to the **AV** object and the **TIO10** object as arguments. The main program loop consists of an inquiry to the **Keyboard** object to see if a complete command string is available, where "complete" means it ends in a carriage return. If one is, then the command string is passed to the *Hardware::doCommand()* routine for parsing and, if possible, execution. On every pass through the main loop the *Hardware::doPoll()* routine is also called, which runs the *poll()* member function of all **Hardware**-derived objects. At the moment the main program also displays various status information about the **PolyAxis** object, such as the estimated position of the source and the residual force on it.

5 The PolyAxis Class

The **PolyAxis** class consists of an array of **Axis** objects, and **AV** object and collection of information about the source. Each **PolyAxis** object has its own entry in the database file POLYAXIS.dat. This entry is read by the constructor based on the **PolyAxis** name. The constructor also takes as arguments pointers to the **AV** object and the **TIO10** object that are created at the top level. The format for the database entries is shown in Table 1.

The first line in the database file is the version identifier, which is the string `VERSION` followed by `1.00` with no spaces. This is matched with a version number supplied by the code to ensure the format matches what is expected. The version number is local to each object type, so that one may be updated without affecting the others. A **PolyAxis** entry is identified

```
VERSION1.00

POLYAXIS: PROTOTYPE
AXIS: AXIS0P // prototype axis objects
AXIS: AXIS1P
AXIS: AXIS2P
POSITION: -0.199943 0.000000 95.569618419 // last known position
SOURCE_MASS: 5.3357 kg // true mass of source, units required
SOURCE_VOLUME: 0.0 cm^3 // volume of source, units required
END;
```

Table 1: PolyAxis database entry

by the string `POLYAXIS:` followed by a name. The name comparison is case in-sensitive. Following the **PolyAxis** name is a list of **Axis** object names. These names are used to find **Axis** objects in their database file (see Section 6.) The `POSITION:` line gives the most recently known source position in centimetres from the centre of the detector co-ordinate system. This entry is updated once per second by the **PolyAxis** *poll()* routine when the source is moving, and once every ten seconds when the source is stationary, so the **PolyAxis** object will know where the source should be when it starts up. If for some reason you move the source by hand you will have to change this entry of the source-finding routine of **PolyAxis** will probably fail. The last two lines give physical parameters of the source: its mass and volume. These are needed to estimate the forces acting on the source. For sources in air, just set the volume to zero to eliminate the buoyancy correction. The density of D_2O is taken to be 1.10 g/cm^3. The mass and volume of the source should have their units specified (the mass can be in g or kg, the volume in `cm^3, cm**3, cc, l, m^3 or m**3`.) Masses are converted internally to kg, volumes to cm^3.

The **PolyAxis** object has the responsibility of figuring out where the source is and how to change the lengths of the various ropes to get it somewhere else. The **Axis** objects have the responsibility of changing the rope lengths while maintaining rope tensions within their upper and lower bounds. The basic tool the **PolyAxis** object uses to track the source is the function *findPositionL()* which finds the position of the source by looking at the lengths of the **Axis** ropes. For the best estimate of the source position the sum-squared length error is a minimum. The minimization is done by iterat-

ing on a linearized version of the problem, which increases speed a reliability over doing the full non-linear minimization. Various approaches to non-linear minimization were taken prior to settling on the iterated linear algorithm. The Marquardt-Levenberg algorithm was tried, but the surface is funnel-shaped with very small parabolic bottom. The M-L algorithm wandered badly: a bad parabolic step was followed by a good steepest-descents step that failed to get close enough to the parabolic region for the next parabolic step to be any good. The downhill simplex algorithm was also tried (*amoeba()* from NUMERICAL RECIPES IN C) but was too slow for reliable control. A simple steepest-descents algorithm was used with some success, but it was neither as robust nor as fast as the iterated linear algorithm.

The iterated linear algorithm begins with the equations:

$$L_r^2 = \sum_{i=1}^{axisNumber} (L_i - (\vec{xt}_i - \vec{xs} - \vec{\Delta}_i) \times \hat{ut}_i - (\vec{xb}_i - \vec{xs} - \vec{\Delta}_i) \times \hat{ub}_i)^2 \quad (1)$$

where L_r^2 is the squared length residual, $\vec{xt}_i$ is the position of the top pulley of axis i, $\vec{xb}_i$ is the position of the bottom attachment point of axis i, $\vec{\Delta}_i$ is the offset of the source-carriage pulley from the source-carriage centre point, and $\hat{ut}_i$ and $\hat{ub}_i$ are the directions of the top and bottom segments of the rope. For the central rope, which does not have an attachment point in the vessel, $\vec{xb}$ is equal to $\vec{xs}$, and $\vec{\Delta}$ and $\hat{ub}$ are zero, so the second term drops out. L_i is the measured length of the i^{th} rope. The second and third terms in the equation will be readily recognized as the lengths of the rope above and below the source. The sum of these terms for a given axis is just the total length of rope expected for the source at position $\vec{xs}$. A schematic representation of these quantities is shown in Figure 2.

The minimization is carried out by taking the derivative of the squared length residual with respect to the each direction with the assumption that the directions of the upper and lower sections of rope remains constant while the source position changes. This results in the set of linear equations for the source position:

$$\begin{bmatrix} -2(ut+ub)(\hat{ut}+\hat{ub}) \\ -2(vt+vb)(\hat{ut}+\hat{ub}) \\ -2(wt+wb)(\hat{ut}+\hat{ub}) \end{bmatrix} \vec{xs} = 2(\hat{ut}+\hat{ub})(L-(\vec{xt}-\vec{\Delta})\hat{ut}-(\vec{xb}-\vec{\Delta})\hat{ub}) \quad (2)$$

where for each term a sum over i is implicit.

Solving these linear equations is handled by the ThreeVector class. The solution has one problem: for motion in a single plane there is a tendency

for the solution to wander out-of-plane to make up for any errors in the rope lengths. This is dealt with by the relatively harsh expedient of zeroing any components of the equations that are out-of-plane. This is done by the **PolyAxis** constructor creating a vector called `freeze` during startup that is used to mask off any out-of-plane components. `freeze` has a zero component for any direction that has a sum of absolute values of the bottom attachments of less than 1 cm. If the acrylic vessel moves significantly it may be necessary to relax this standard somewhat.

The linearized equations are iterated until the change in source position between iterations is less than 0.1 cm. Ideally one would like to minimize the tension error at the same time as the length error (in particular, this would eliminate the problem of wandering out-of-plane.) Unfortunately, I haven't been able to figure out how to cast the tension equations in a similarly linearized form, so there is no set of grand linear equations to do this particular job.

As well as knowing what the source position is, the **PolyAxis** object has the responsibility of changing it. This is done using the *to()* member function, which takes a string argument containing the position in three-space of the desired position. A few simple tests are applied to this position to ensure it is inside the vessel and can at be reached, at least in theory, without violating any tension constraints. If the point meets the constraints a path is generated from the current position to the final position. This path consists of at most two straight line segments. If the start and end positions are either both in the vessel or both in the neck only one segment is generated. If the source is to move from the vessel into the neck or vise versa then two segments are generated: one to a point just below the neck, the other away from this point to the end position. There is a member function called *neckIntersection()* that determines if and where a rope intersects the neck ring. It is based on the assumption that there is no friction between the neck ring and the rope, so that the rope will always lie in a plane that contains a radius of the neck ring.

Prior to returning control the main loop, *to()* calls the *polyActivate()* member function of each of the **Axis** objects that make up the **PolyAxis** . This puts the **Motor** object in each **Axis** into velocity control mode, and places them in standby mode. It also sets the expected length for each rope, turns off command acceptance for the **Axis** and its sub-objects, and initializes some arrays used to store past values of length and tension for rate-of-change calculations needed for damping rules (see Section 6 for details.) The program is then returned to the main loop, and the rest of the **PolyAxis** control sequence is carried out by the *poll()* function.

The basic tasks of the PolyAxis *poll()* function are to move a point along the line segments calculated by *to()* and to set up error values for each Axis object based on the difference between the position of that point and the estimated position of the source. The point is the "hare" of a hare-and-hound controller, with the source itself playing the role of hound. The point is moved according to a hardcoded velocity profile such that the point velocity increases linearly for the first 20 cm of path to a maximum of 2.0 cm/s, and decreases similarly at the end of the path. For paths that turn at the neck, the source is brought to a stop at just below the neck, and then a similar velocity profile is followed for the second part of the path.

The Axis object errors are set by the function *errorSignal()*. There is a related function *errorSignalAll()* that will be discussed in more detail below. *errorSignal()* sets both error tensions and length. The error lengths are simply the difference between the actual length and the desired length of any given rope for the current point position. The error tensions are set differently depending on the status of the rope: if a rope has a bottom attachment that is on the far side of the central axis relative to the current source position, the error tension is set in such a way as to move the rope toward a tension of 10 N. The error tensions of the other ropes are set to zero, which is a signal for the code to let them take on whatever values they like. This mechanism of setting the off-side rope tensions to 10 N effectively makes them idlers, or nearly so, and brings the control algorithm back into the realm of constrained rather than over-constrained systems.

The function *errorSignalAll()* is used to set errors for all ropes, and is useful for attempts at full over-constrained control. It was found that the loadcells are not generally accurate enough to do this effectively. Note that there are two rather similar terms in the code that are really quite different: and `errorLength` is the difference between the actual length and the desired length, and a `lengthResidual` is the amount a rope contributes to the RMS residual that was minimized to find the source position. All errors are defined to be `(actual - desired)` but the residual length is defined with the opposite sign to make the derivatives work out properly.

There a number of sloppy usages in the code: the terms `distance`, `position` and `length` are used interchangeably in some cases and with distinct meanings in others. `Force` and `tension` are also used interchangeably sometimes and not others. There may be some idle functionality left over from earlier versions of the code, although I've tried to eliminate this where it seemed likely it would never be needed again.

The list of commands the PolyAxis object will accept from the keyboard is shown in Table 2. Required arguments are given following the command

to [x y z]	move to (x,y,z)
tensionFind	find and display position based on tensions
tensions	find and display desired tensions
netForce <x y z>	find and display net force magnitude at (x,y,z) or current position if none given
pattern [fileName]	run through a pattern of endpoints
stop	stop all motors
tensionStop	toggle stop-on-tension condition

Table 2: PolyAxis commands

name in square brackets, optional arguments in pointed brackets.

The `tensionStop` command toggles a flag whose state determines how the code reacts to tension-out-of-bounds conditions. If the tension goes out of bounds this flag can be unset to allow individual **Axis** objects to be controlled without the **PolyAxis** object stopping the whole show on every poll. This flag is set automatically whenever the *to()* function is called, to provide some protection against forgetting reset it.

The `pattern` command allows the source to run through a pre-defined pattern of points listed in the file given. The first line of the file is the name of the pattern log file to be used, which records where the source stopped for each endpoint and why. The rest of the lines are endpoints. The pattern following code moves to an endpoint, pauses for about 10 seconds, then moves on until the pattern is complete.

One of the more problematic aspects of the control algorithm is knowing when to quit. There are five stop conditions:

ENDPOINT_STOP source is within the lesser of `END_ERROR` and `distErr` of the endpoint, but in no case is the endpoint condition tighter than 0.1 cm. `distErr` is the RMS length residual from all **Axis** objects.

STUCK_STOP source is more than 1/2 way to the endpoint and hasn't gotten any closer in the last 10 seconds.

NET_FORCE_STOP `NET_FORCE_LIMIT` has been exceeded

LOW_TENSION_STOP tension is below `LOW_TENSION` on an **Axis**

HIGH_TENSION_STOP tension is above `MAX_TENSION` on an **Axis**

Depending on the frictional forces and hydrodynamic damping in the final system some of these conditions may need to be changed. The numerical values for the various quantities like END_ERROR are defined in the file PolyAxis.h.

6 The Axis Class

The **Axis** class is home of the fuzzy logic system that tries to maintain the **Axis** conditions such that the error terms set by the **PolyAxis** code are kept small. It also handles the tension constraints and includes damping terms in both length and tension to improve the stability of the system. The **Axis** constructor is typically called by the **PolyAxis** object. It is passed the name of the **Axis**, a pointer to the **AV** object that exists at the top level scope and a reference to the **TIO10** object that also exists at the top level.

The database utilities search the file AXIS.dat for the named **Axis** , and then read the fields shown in Table 3. The MOTOR: LOADCELL: and ENCODER: fields name the **Motor, Loadcell** and **Encoder** objects that are part of this **Axis** . The LENGTH: is the length of the rope from the upper pulley to the bottom end. The DEADLENGTH: is the length of rope between the spindle and the top pulley. This length should be small in the final installation, but is large in the prototype. It is used by the code that corrects the length of the rope for the effects of elasticity. Note that the **Axis** object should always use the length returned by *Axis::getLength(void)*, which includes a correction for rope stretch, and not *Encoder::length(void)* which returns the unstretched length. The OFFSET: is the offset in centimetres from the central point of the source carriage to the centre of the pulley on the source carriage through which the rope passes. At the moment the rope length calculation does not account for the varying amount of rope passed around the source carriage pulley as a function of distance from the central axis of the detector. The TOP: and BOTTOM: positions are the locations of the point where the rope comes off the top pulley and where it meets the bottom attachment point. For a central rope there is no BOTTOM: attachment point given, and the OFFSET: is zero. The LENGTH: field of the Axis object database entry is updated every second when the **Axis** is moving and every ten seconds when it is stationary.

The sub-object names passed to the Axis object are used to search the appropriate database files for the named objects, but the names these objects are given in the code are not the names used in the files; instead they are given names that are a compound of their class name with the name of the

```
VERSION1.00

AXIS: AXIS0
MOTOR: motor0
LOADCELL: loadcell0
ENCODER: encoder0
LENGTH: 293.4052730000000 // re-written length
DEADLENGTH: 453.0 // unchanging length
OFFSET: 0. 0. 0. // Delta in Equation 1
TOP: 0. 5.0 389.0 // top pulley position
END;

AXIS: AXIS1
MOTOR: motor1
LOADCELL: loadcell1
ENCODER: encoder1
LENGTH: 614.24914660000000 // re-written length
DEADLENGTH: 453.0 // unchanging length
OFFSET: 0. 0. 0. // Delta in Equation 1
TOP: 50.0 0. 391.0
BOTTOM: 317.0 0. 89.53 // bottom attachment position
END;
```

Table 3: Axis database entries for central and side ropes

Axis that they are part of. Thus, the **Motor** object for `axis0` is given the name `axis0Motor` and so on.

There are two **Axis** control modes, corresponding to the control modes of the **Motor** object: `STEP_MODE` and `VELOCITY_MODE`. The first of these is used for controlling a single axis: the number of steps the motor should take, and the direction to take them in, is calculated on each poll, and the **Motor** object is left free to work out how to take them. When the **Axis** gets to within 1 cm of the desired endpoint it stops recalculating the number of steps and lets the **Motor** finish the move.

`VELOCITY_MODE` is used for **PolyAxis** control. In this case the **Axis** object employs a fuzzy logic algorithm to estimate the change in velocity a motor should have, and then sets the desired motor velocity accordingly. The **Motor** object will change its velocity to match the desired velocity the next time it is polled. There are ten fuzzy rules in operation in the current controller:

- If tension is low then increase tension
- If tension is high then decrease tension
- If corrected length is short then increase length
- If corrected length is long then decrease length
- If not near end and relative tension is low then increase relative tension
- If not near end and relative tension is high then decrease relative tension
- If velocity is high then decrease velocity
- If errorLengthVelocity is high then decrease errorLengthVelocity
- If errorTensionVelocity is high then decrease errorTensionVelocity
- If dv is high then decrease dv

Some of the rules may look a little obscure, so some justification for using them is given toward the end of this section. The meanings of the various terms is given by reference to Figure 3. The "corrected length" is the measured length with a correction for the amount of the length residual. The reason for putting this in is so that inaccuracies in the source position that arise from imprecise measurements of rope length or the positions of the bottom attachment points don't appear as error terms in the control algorithm. The rules themselves are coded in a set of **Axis** member functions that

have names like `isLowT(float tension)`, which returns the membership of `tension` in the set `LowT`.

A fuzzy set is defined along an axis (not an Axis!) that represents a physical quantity like tension. The "membership" of a tension value in a fuzzy set is the magnitude of the set at that value. In Boolean logic membership values are always zero, one or undefined (at the transition between zero and one.) Fuzzy sets, unlike Boolean sets, obey the law of noncontradiction everywhere: they vary smoothly between one and zero over some boundary region.

Each rule has a premise (the bit before the "then") and a conclusion. The truth value of the premise is calculated using various fuzzy operations. The truth value of a simple premise is just the membership of the input value in the set it is associated with (for example, the truth value of the premise `tension is low` is just the membership of `tension` in `LowT`.) For compound premises the individual assertions like `tension is low` are connected by fuzzy operators such as AND, OR and NOT. There are various ways of defining fuzzy operations, and I've chosen ones here that are easy to code and computationally fast. The fuzzy AND operation I've represented using multiplication, whereas the usual way is to take the smaller of the two membership values. The fuzzy OR is the larger of the two membership values (which is not used in the rules at the moment) and the fuzzy NOT is just one minus the membership value.

The truth value of the conclusion of a fuzzy rule is usually calculated by truncating the output set at the truth value of the premise. One then does some kind of averaging to "defuzzify" the output and produce the desired control value. In this case, efficiency considerations have led me to put the dufuzzification step into the output directly. The conclusion of each rule (the bit after the "then") in each case is translated into a change in the motor velocity. To decrease the tension, for instance, the motor velocity is increased (positive velocity means the rope is getting longer) and to decrease the length the motor velocity is decreased. The velocity change values are supplied by the functions `deltaV(int indicator)` and `deltaDV(float dv, int indicator)` where the flag `indicator` selects what kind of velocity change you want. Both the magnitude and direction of the velocity change depends on the value of `indicator`; for instance, to increase tension a velocity change is -0.4 cm/s is returned, and this value is multiplied by the truth value of the premise, then added into the total velocity change. The output of all the rules is just the premise-weighted sum of the velocity changes: this is a very simple defuzzification method that more than makes up in efficiency for what it loses in generality.

up [dist]	move up dist cm
down [dist]	move down dist cm
to [len]	change length to len cm
calibrate	toggle loadcell calibration mode
point [load]	enter a loadcell calibration point
maxspeed [speed]	set maximum motor speed in cm/s
zero	reset encoder counter
stop	stop the motor

Table 4: Axis commands

The fuzzy rules currently in use are suitable for the prototype manipulator in air with the old source carriage. They may have to be modified for the new source carriage, the full detector geometry and the effects of water.

The high and low tension and length rules should be self-explanatory, but some of the other rules warrant comment. The relative tension rules (that is, the rules that deal with the tension relative to the desired tension set by the code) are relaxed toward the end of the path. This relaxation was added because it was found that errors in the tension measurements resulted in these rules preventing the source from reaching the endpoint in some cases. The rules are only partially relaxed, however; otherwise the side rope tensions tend to get very high for endpoints in the neck (why this is so is not clear.)

There are two damping rules, one based on the rate of change of the length error and one on the rate of change of the tension error. Both sets of rules are necessary, especially to prevent interactions between ropes from driving each other into oscillation. A common scenario is that a side rope finds itself with too much tension and slacks off and the opposite side rope finds itself with too little and tightens up. While the ropes are stable individually, the interaction between them leads to more rapid changes in tension than the rules are designed to compensate for, and so the system oscillates. Adding damping based on the rate of change of tension has eliminated this phenomenon. The damping constant for the length error velocity was calculated by observing the period of the oscillations and treating the system like a free oscillator. The damping constant for the tension error velocity was determined empirically.

The **Axis** object takes the commands shown in Table 4. When running an **Axis** from the command line the motor is in `STEP_MODE` and its acceleration profile is determined by simply decreasing the count-down time of the associated clock channel by a fixed amount every step. This amount is set to ten at the moment, but may be as little as one. Once the maximum speed is reached the motor runs at constant velocity. During the acceleration

```
VERSION1.00

AV: PROTOVESSEL
NECK_RING_STATIC_POSITION: 5.7 0. 364.0
NECK_RING_RADIUS: 42.0 cm
NECK_RADIUS: 42.0 cm
NECK_LENGTH: 27.0 cm
VESSEL_RADIUS: 3.20 m
END;
```

Table 5: Acrylic vessel database entry

phase the number of steps is counted, and when the motor gets to within this number of steps of the endpoint it goes into deceleration, increasing the count-down time by the same increment on every poll. The calibration commands for the loadcell will be described in Section 10: they are just calls to the **Loadcell** calibration commands in any case.

7 The AV Class

The AV class is still only partially implemented. It is intended to supply the rest of the code with information about the state of the acrylic vessel. To do this is must know the geometry of the vessel and know its position and orientation in the global co-ordinate system. The vessel geometry is described by entries in the database file AV.dat, as shown in Table 5. The **AV** constructor just takes the name of the vessel to be used as an argument, and searches the AV.dat file until it finds an entry for an **AV** object with this name.

The `NECK_RING_STATIC_POSITION:` entry gives the position of the centre of the neck ring when the centre of the vessel is at the origin of the global co-ordinate system and the neck is pointed straight up. The `NECK_RING_RADIUS:` is the interior radius of the neck ring. The radius of the allowed region of the neck is given by the `NECK_RADIUS:` entry: this is the radius within the neck that the source is not allowed to move outside of. The `VESSEL_RADIUS:` is just the radius of the vessel (and here we set x equal to five....)

At the moment the **AV** object is almost entirely non-functional. The functions it needs are prototyped but don't do anything, mostly because the

```
VERSION1.00

MOTOR: motor0
CHANNEL: 7
START_SPEED: 0.5
CRUISE_SPEED: 7.0
END;
```

Table 6: Motor database entry

hardware to measure the vessel position has not yet been integrated into the prototype system. The **AV** object does not take any commands, although a dummy command has been built into it to satisfy the constraints of the **Hardware** hierarchy, which expect at least on command per class.

Calls to the **AV** member functions *vtog(ThreeVector x)* and *gtov(ThreeVector x)* have been put in the **Axis** object and the **PolyAxis** object at the appropriate places. These functions are supposed to take a position vector x and transform it from the vessel to the global co-ordinate system (*vtog()*) or vice versa (*gtov()*). Their return value is the transformed vector. The functionality exists in the **ThreeVector** class to do this transformation, but I've not got around to implementing it here.

8 The Motor Class

Motor objects control stepping motors by setting the period of the clock on an output channel of the TIO10 card. The motor takes a half-step for every clock pulse. Several types of motor may be used in the current system: one type for manipulator control, another for the laser system and a third for the stearable source. Some motor parameters are read in from the MOTOR.dat database file entries to facilitate the use of different motor types. A typical database entry is shown in Table 6.

The database name of a **Motor** object is not the same as the name it is referred to in the code. The **Motor** constructor takes two names as inputs: one is the database name and one is the name used in the code. The latter is generally constructed by the object that the **Motor** object is part of. In the current set up, the **Motor** for `axis0` would be named `axis0Motor`.

The `CHANNEL:` is the TIO10 channel for this motor, which is determined by

the wiring of the external hardware. The START_SPEED and CRUISE_SPEED: are precisely what you would think they are, given in cm/s. 7.0 cm/s is the maximum allowed cruise speed and 0.07 cm/s is about the minimum allowed start speed. These limits are enforced by the code.

The **Motor** object has a number of features that are hardcoded to reflect the current configuration. The PIA chip on the TIO10 card is used to set bits that control the direction of a motor and turn all the windings off when the motor is idle. Two arrays, `dirBit[]` and `awoBit[]` contain the bit positions associated with each output channel. If the outputs are rewired for any reason these arrays will have to be changed. Note that channels 4 and 5 are reserved for the realtime clock, and this condition is also imposed by the **Motor** code. The final hardcoded constant is STEPS_PER_CM which reflects the mechanical connection between the motor and the rope spool.

As mentioned above, the **Motor** object has two control modes: STEP_MODE and VELOCITY_MODE. In STEP_MODE the motor's acceleration profile is determined by simply decreasing the count-down time of the associated clock channel by a fixed amount every step. This amount is set to ten at the moment, but may be as little as one. Once the maximum speed is reached the motor runs at constant velocity. During the acceleration phase the number of steps is counted, and when the motor gets to within this number of steps of the endpoint it goes into deceleration, increasing the count-down time by the same increment on every poll.

In VELOCITY_MODE the variable `desiredPeriod` is assumed to have been set by a preceding call to *setDesiredVelocity()* or *changeDesiredVelocity()*, both of which set the desired period as well as the desired velocity. The desired direction is also set by these functions; note that the velocities are signed quantities, but the the period is always positive. In STEP_MODE the motor's direction is never allowed to reverse; in VELOCITY_MODE it is. To facilitate reversal the **Motor** object has three possible states: on, off and standby. The variable `onFlag` is used to store this state information, with a value of 0 meaning off, 1 meaning on and 2 meaning standby. This allows a motor to be slowed to a halt and then restarted (possibly in the opposite direction) if the **Axis** control algorithm is not done with it. Note that when the motor has been turned off by setting `onFlag` to zero it can't be restarted without initiating a new command sequence from the keyboard or **Server** object.

As well as `onFlag` there is a variable called `stopFlag` which in the past was used to communicate with the **Axis** object to allow the **Axis** object to stop a motor when the **Motor** object thought it was done with it. I don't think this flag is used anywhere in the code any more, but have kept

```
VERSION1.00

ENCODER: encoder0
SLOPE: 0.2392  // conversion from ADC value to cm
ADDRESS: 168  // encoder board address
ZEROLENGTH: 747.045410 // rope length when encoder reads zero
DATE: 00-00-0000 // last calibration date (not used)
END;
```

Table 7: Encoder database entry

it around as it may be useful again in the future.

The rate at which the motors are polled has an effect on the control algorithm. At the moment they are still being polled a little faster than need be: the interval between polls must be at least `LOOP_TIME`, which is 10^{-4} seconds at the moment. It could probably be increased by a factor of ten without significant loss of performance, and because the current fuzzy rules were developed for a longer polling time (because of the use of non-linear minimization algorithm in **PolyAxis** to find the source position) they may not work quite so well under the current circumstances. But as they will have to be changed when we go to full scale it is probably worth waiting until then to change them. A related quantity is `DV_LIMIT`, the maximum change in motor velocity allowed per polling cycle. If this is too large the motors may stall. It is currently set at 0.5 cm/s.

The only command the **Motor** object accepts from the keyboard is `maxspeed [speed]`, which sets the maximum speed in cm/s.

9 The Encoder Class

The **Encoder** class reads the output of a shaft encoder via custom SNO electronics. The **Encoder** class itself is a Level 3 object, and it uses a rather messy Level 2 object, the **DigitalChannel** class, to do actual hardware access. **Encoder** objects know about the calibration of their encoder, and a hardware address they tell their **DigitalChannel** object to read values from. A typical entry from the ENCODER.dat database file is shown in Table 7.

The `ADDRESS:` is the address of the encoder board on the dataConcen-

reset	reset counter to zero
read	read and display value
readloop	enter read/display loop
stoploop	quit read/display loop
setZeroLength [len]	set zero length to len cm

Table 8: Encoder commands

trator chain. The whole addressing scheme will have to be revised when the dataConcentrator hardware is ready in its final form, so not a lot of attention will be paid to it here. The ZEROLENGTH: of the rope is the total length including the dead length specified in the associated **Axis** object. Because the **Encoder** class is at Level 3, it does not know anything about the **Loadcell** class, which is also at Level 3, or the **Axis** class, which is above it at Level 4. Therefore all it can know about it the total length of its rope, without any correction for stretch. The ZEROLENGTH: database field is updated once per second when the length is changing and not at all otherwise. The value used to update ZEROLENGTH: is the CURRENT LENGTH. This is because the **Encoder** constructor resets the counter to zero at startup, and so the old current length becomes the new zero length.

The list of commands the **Encoder** object will accept from the keyboard is shown in Table 8. The readloop command causes the **Encoder** to print its value to the screen every time it is polled.

10 The Loadcell Class

The **Loadcell** object is much like the **Encoder** object: it is a Level 3 object that accesses hardware via the **AnalogChannel** object. The and entry in the **Loadcell** database file LOADCELL.dat is shown in Table 9. Like the **Encoder** , some of the hardware-access data will have to be modified when the final dataConcentrator hardware becomes available.

The most important feature of the **Loadcell** object is its calibration facility. Calibration mode is turned on by entering running the "calibrate" command from the keyboard. Subsequent "calibrationPoint" commands are given with various masses hung on the rope. The value of each mass is given on the command line as well. When two or more calibration points have been given, the user enters the "calibrate" command again and the new calibration constants are written to the database file.

Tests have shown the loadcells to be quite linear over most of their range.

```
VERSION1.00

LOADCELL: loadcell0
MAXLOAD:  50 lb // maximum load in pounds
SLOPE: 0.6777730537 // conversion from ADC value to newtons
OFFSET: -482.535980 //  according to N = slope*adc + offset
CHANNEL: 3     // ADC channel setting
DATE: 00-00-000     // unused calibration data
END;
```

Table 9: Loadcell database entry

However, for very small loads the electronics they are connected to is quite non-linear. For this reason it is necessary to add a 75 k resistor between the red and green leads of each loadcell to provide a small bias that will prevent the amplifiers from being driven into the non-linear regime. The addition of this resistor adds about 0.7 % non-linearity to the overall response of the system, which is an acceptable loss to avoid the region of much higher non-linearity.

There are still some bits of old code hanging about from an earlier incarnation of the code: the *setTensionLimits()* function is no longer used and may be discarded. Its role has been taken over by the net force limits in the **PolyAxis** object.

11 Low Level Classes

There are a whole bunch of Level 1 and 2 classes for dealing directly with hardware. I didn't write most of them, and those I did are going to have to be re-written in fairly short order. Here are a few comments about some of them.

11.1 The DigitalChannel Class

The **DigitalChannel** class accesses the counters on the dataConcentrator board via a simple PC interface card that will be replaced by the real dataConcentrator interface card soon. It does not deal with the encoder boards out on the axis hardware at all, but just calls in the value from one of the

counter channels on the dataConcentrator card itself. No facility for handling multiple dataConcentrator cards exists at the moment, but one will have to be written when the new hardware arrives.

11.2 The AnalogChannel Class

The **AnalogChannel** class handles the A/D converter on a dataConcentrator card. Like the **DigitalChannel** class it has no way of dealing with multiple dataConcentrator boards. When the new hardware arrives these two objects should, at best, be made subobject of a new **dataConcentratorBoard** class which itself would be a subobject of a **dataConcentratorChasis** class. At worst, these classes should be scrapped altogether and something more sensible put in their place.

11.3 The TIO10 Class and Its Components

The **TIO10** class handles the National Instruments TIO10 card. It implements essentially all of the functionality described in the TIO10 manual. It was written almost entirely by Aksel Hallin. The TIO10 class has subclasses that describe the main chips: the **AM9513** timer and its channels, **AM9513Chanenl** objects, and the **pia** class to handle the MC6821 peripheral interface adapter. These Level 1 classes access the card registers directly. The only way they need to be extended is to add some more functions for checking the status registers so that one can tell that the card is actually there an functioning.

Note that the **TIO10** class and its subclasses are not part of the **Hardware** hierarchy. They are not polled and do not accept commands.

12 The Display Class

Display is a simple screen-handling class that is globally accessible to allow all **Hardware**-derived objects to print stuff to the screen in a semi-organized way. The top level code sets up the **Display** class to put the cursor on line 19 of the screen, and I try to restrict error messages and other outputs from **Hardware**-derived classes to the five or six lines below this. The **Hardware** class hierarchy itself uses the middle part of the screen for error messages, and the very top of of the screen is used by the top level code to display the source position and various error terms, as well as the time.

Messages are sent to the **Display** object, which is called `display`, by first printing the string you want to output to the public member `display.outString`

and then calling the function `display.message(int x, int y)` where x and y give the location on the screen to start the message. An alternative command is `display.messageB(int x, int y)` which blanks the line first, to eliminate overlap with the tag end of old messages.

13 The Keyboard Class

The **Keyboard** class takes input from the command line, including handling backspaces and the like. It does not care for arrow keys much. A useful bit of added functionality would be a command history.

14 The Clock Class

The **Clock** class is a realtime clock that uses channels 4 and 5 of the TIO10 card. A **Clock** object called RTC for Real Time Clock is defined with global scope at the top level. The time in seconds from program startup is returned as a double precision number by *RTC.time()*.

15 The Server Class

The **Server** class needs some work. It is supposed to connect the calibration PC with the outside world, and allow a privileged client to send commands to the manipulator control code and make them look like they came from the keyboard. The **Keyboard** class should probably be called by the **Server** , rather than the *Hardware::doCommand()* function as is now the case. The real problem with **Server** is that the communications function are still not working, for reasons that are obscure. A standalone test code in C:motors comm seems to work fine, but the behaviour is not the same when integrated into the main code.

Note that there are bunch of special libraries that have to be set in the makefile to handle the **Server** class. Some of the definitions in these libraries or their associated header files conflict with similar Borland C++ definitions. For this reason the actual **Server** object in the main code is defined as a static variable in separate file: servprot.cpp.

Note also that conflicting definitions from the PCTPC stuff produces three link-time warnings during compile.

16 Helper Classes and Functions

There are a whole bunch of helper functions that are part of this system. The most important are the **ThreeVector** class, which handles all the geometric transformations, and the various database utilities in ioutilit.cpp. All I have time to say is that you should look at the way these helpers are used in the existing code to deal with geometric and I/O problems and see if you can use their functionality rather than rolling your own.

17 Mechanical Considerations

There are whole bunch of features of the mechanics that have not been discussed here. A short list is:

- air cylinders or alternative tensioning devices
- D_2O inventory and control
- role of umbilical in control problem
- the role of the neck in control problem
- lots of others that have slipped my mind just now

We need to keep these problems in mind and worry about them whenever possible. In particular, we need to know how the system behaves in the neck and in the lower half of the vessel. The fuzzy rules may need to be modified to ensure stable control in these regions.

18 Top Level Functionality

The top level code at the moment traps a few commands before they can be passed to the **Hardware** parser. They are:

quit stop all motors and exit program

log toggle logging state

debug toggle debugging state

Turning on logging opens a file where objects like **PolyAxis** and **Axis** can dump state information while they are moving. This is particularly useful in adjusting the fuzzy rules in **Axis** . There is a standalone routine called logsort.cpp that breaks the log file into axis-specific parts that contain state information and rule outputs. The files have names like a0rules.dat and a0state.dat. There are Origin spreadsheet files called a0.org and the like that allow these data files to be read in and automatically plotted, so that you can see what rules are active when, and what state information is really causing that interesting oscillation.

The debugging toggle just sets or unsets a global variable called debugging that can be used by object in polling functions to decide if they want do dump some data to the screen.

There is also one command line argument that the main code accepts: "-forceMap" This generates three files that contain the force vector from each rope on the source for a range of positions. The files are named "axis0.map" and so on. The current force map code is set up for the prototype: hardcoded limits must be changed to map the real detector.

The code sometimes generates floating point expections. I think all the sources of this have been found, but just in case a floating point exception handler has been written, and is loaded by a call to *signal()* in the top level code. When an FPE occurs the exception handler stops all the moters before exiting. Although hardware protection for the motors is planned, one may as well have as many levels of protection as possible.

19 Compiling

There are several settings that have to be changed to compile this code. The file builtins.mak has to be updated according to the specifications in the PCTCP manual. The stack length has to be changed by setting the variable _stklen in the top level code (this generates a compiler warning) and the correct PCTCP libraries have to be linked in the correct order (these libraries and their associated header files generate three compiler warnings.) The large memory model should be used, with Borland C++ source set in the compiler options. I use the default optimization, and turn on all warnings except the one about "functions containing for-loops are not compiled inline." The code itself should generate no warnings. At the moment there is an "unreachable code" warning generated from the **Server** *poll()* function because the first line is a return statement, as the code itself does not work.

20 Co-ordinate Systems

There are two co-ordinate systems used in the code: the global system and the acrylic vessel system. The lower attachment points of the side ropes are fixed with respect to the AV, but the AV can move with respect to the global co-ordinate system. The global co-ordinate system is officially defined somewhere, but I don't know what it is precisely. The centre of the PSUP is approximately the origin, and the X-axis is along the electronics corridor, so the Y-axis is approximately north, I think. The Z-axis is local vertical, postive up. All the calculations are done in the global co-ordinate system: the positions of the lower attachement points are transformed before anything is done with them.

21 Things To Be Done

I've tried to indicate things that need to be done as I've gone along. There are a lot of them. The most important things at the moment are as follows:

- get the **Server** object working and discuss communications with John Wilkerson
- get the air cylinders working
- install and support full dataConcentrator hardware
- get the AVPsense boards integrated into the system, supported in software and tested.
- modify the new source carriage to let the weight swing free
- recompile the code with Borland 5.0 and run CodeGuard on it to ensure no memory leaks and other nasties.
- test/modify the fuzzy rules for the neck region and the lower half.

There are probably a lot of other things I've forgotten here as well, but that list appears sufficient to keep a few people busy for a while.

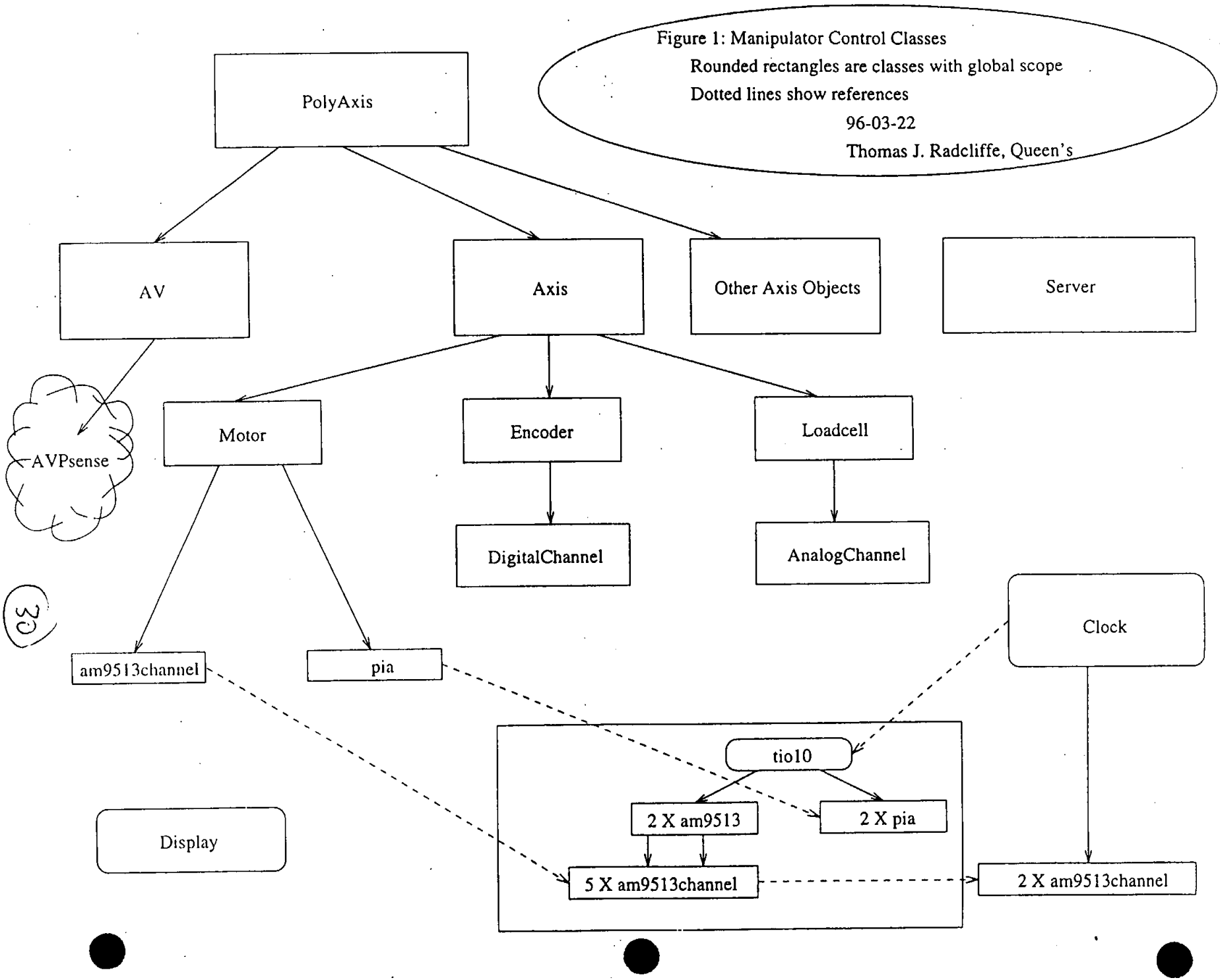
Figure 1: Manipulator Control Classes
Rounded rectangles are classes with global scope
Dotted lines show references
96-03-22
Thomas J. Radcliffe, Queen's
PolyAxis
AV
Axis
Other Axis Objects
Server
AVPsense
Motor
Encoder
Loadcell
DigitalChannel
AnalogChannel
30
Clock
am9513channel
pia
tio10
2 X am9513
2 X pia
Display
5 X am9513channel
2 X am9513channel

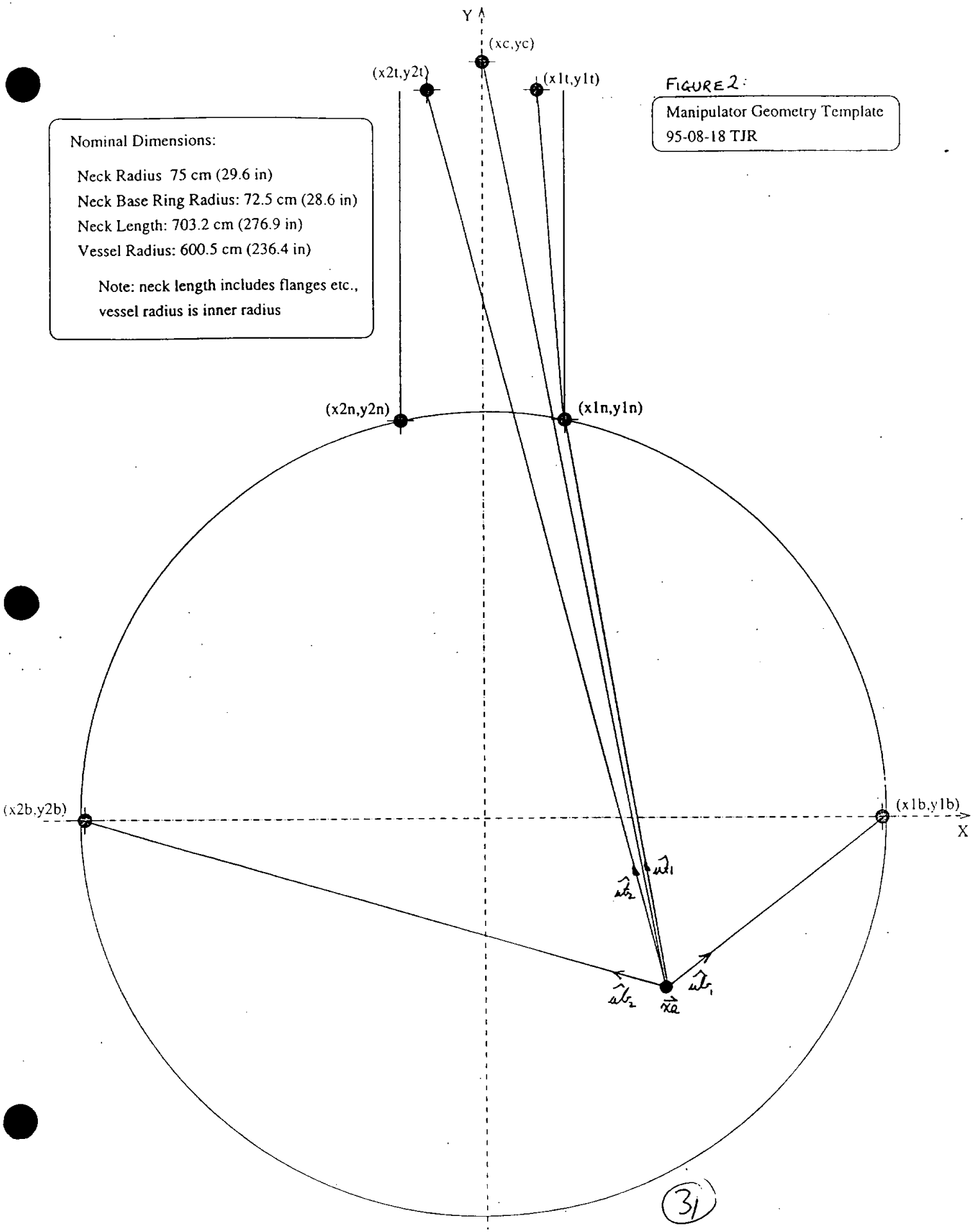

FIGURE 2:

Manipulator Geometry Template
95-08-18 TJR

Nominal Dimensions:

Neck Radius 75 cm (29.6 in)
Neck Base Ring Radius: 72.5 cm (28.6 in)
Neck Length: 703.2 cm (276.9 in)
Vessel Radius: 600.5 cm (236.4 in)

Note: neck length includes flanges etc., vessel radius is inner radius

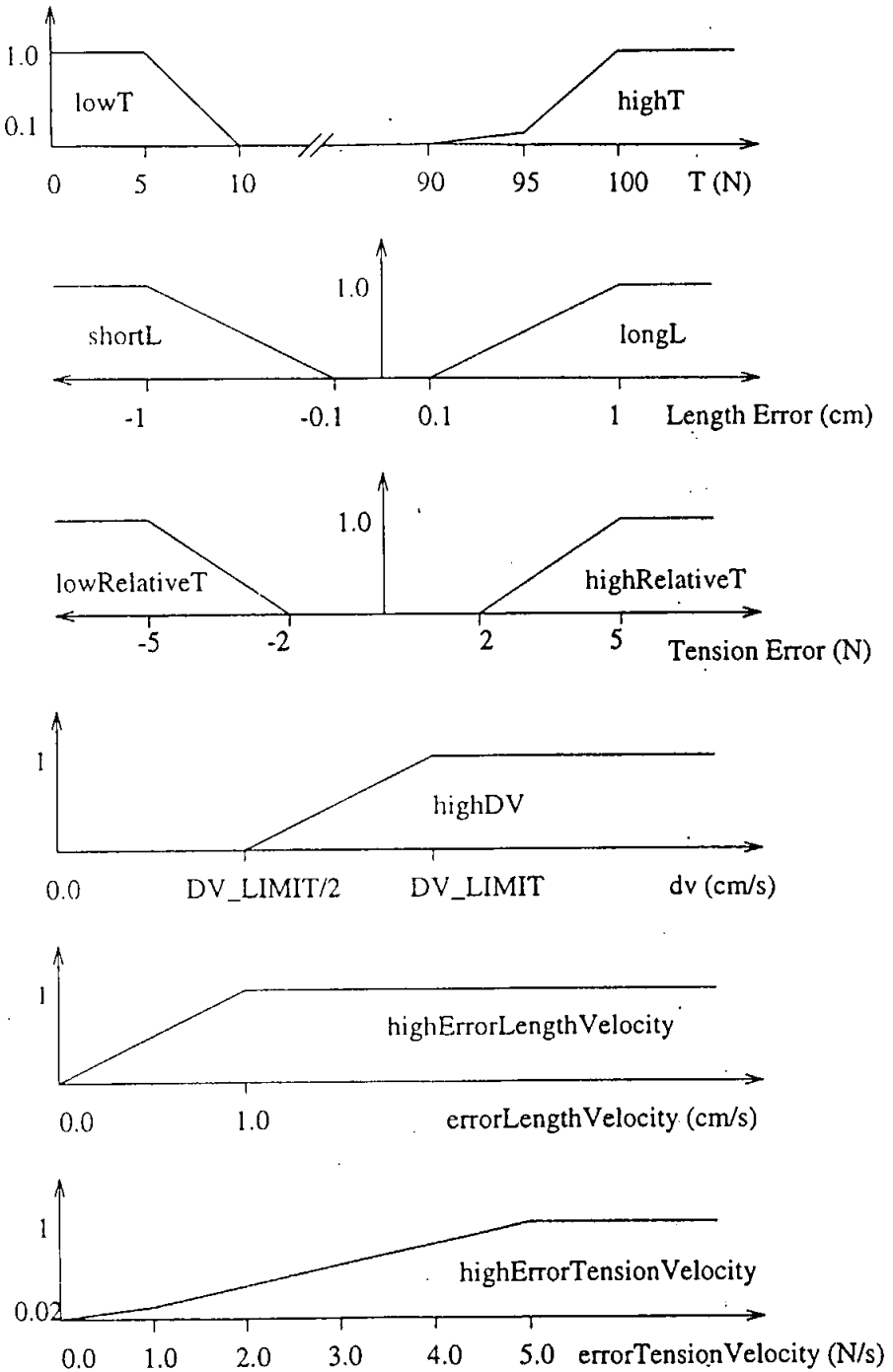

Figure 3: Fuzzy Sets for Axis Control
96-03-27 Thomas J. Radcliffe