Generating exercise programs with the InMotion^2 robotic system

By
Saina Parizadeh

B.S. – Computer Engineering
University Of Kansas, 2001

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

---
Dr. Arvin Agah
(Committee Chair)

Dr. Nancy Kinnersley
(Committee Member)

Dr. John Gauch
(Committee Member)

Date of Acceptance

Abstract

InMotion^2, also known as MIT-Manus, is a robotic system that stroke patients work with to regain their arm movements. InMotion^2 can only help improve shoulder and elbow movement in patients and, to improve functioning, patients need to be able to use their hands. In this paper the overall programming structure, expected use of the InMotion^2 System and workings of the system are explained. In addition 5 therapy exercises are introduced for stroke patients that can possibly be implemented using the robot. Three of these exercises are explained in more details and one of them has been developed for the illustration purposes.

1. Introduction

The objective of this document is to provide a quick guide to creating stroke rehabilitation exercise programs for the InMotion^2 system. The important and necessary details of the system are outlined to give the reader a basic understanding of the system, as well as an organized method for creating an exercise.

This paper starts with a background in stroke and rehabilitation exercises to give a general overview of what the main goals of these exercises are. The first topic of discussion is an overview of the tools involved in this system followed with several sections outlining the basics of the InMotion^2 system. Next there is a section “Exercises” which contains some of the recommended standard stroke rehabilitation exercises. One of these recommended sample exercises is been implemented. The final section gives the necessary details for creating the algorithms to drive the InMotion^2 system. The example program is put together in a step by step manner to illustrate all aspects of the topics discussed. In the final section, some of the main code functionality of few files is discussed along with explanation of the changes for the sample exercise.

2. Background

Stroke is the third highest reason for death among Americans after heart disease and cancer but yet so many people don't know and don't take the sign of stroke as serious. If detected shortly after its occurrence stroke patients can be treated and may have a full recovery. This is the intent and goal of physical therapy exercises for stroke patients [4].

According to Dr. Edgar Kenton, a spokesman for the American Stroke Association, about 700,000 Americans have stroke each year and about 4.6 million live with the stroke related problems [7, 12]. Yet so many insurance companies don't cover the rehabilitation more than six week after the stroke. Stroke rehabilitation exercises are very effective in bringing a stroke patient to a full recovery. However, these exercises take up a lot of a physical therapist's time and as it is been mentioned about these exercises are the function of time. Therefore, a machine that can be configured to perform basic stroke therapy exercises would be extremely beneficial both financially and time wise.

The primary symptom in all stroke patients is the loss of normal postural reflex or reaction mechanism to initiate movement on the affected side [7]. Therefore, the main objective of the physiotherapist is to help the patient to gradually return the affected limb to the normal state and help the patient to regain control. Essentially, treatment aims at re-establishing the normal *postural reflex mechanism* [7].

The inability to move a part of the body results in poor blood circulation in that part of the body and for this reason a stroke therapy exercise needs to focus on improving blood circulation. However, it is important to keep in mind that the effect of these exercises should be evaluated over time. Basically, the recovery process is naturally a function of time and it changes gradually either slowly or quickly, but not suddenly [8, 12].

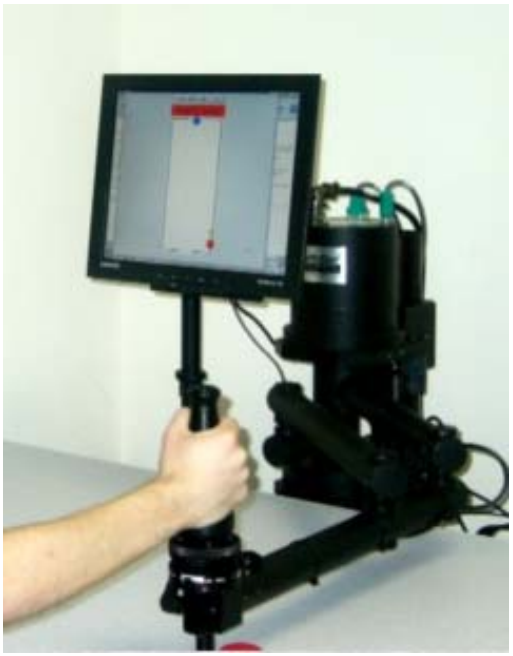
Although, based on some studies that were done using a novel robotic device called Assisted Rehabilitation and Measurement (ARM) Guide, the primary stimuli to recover can be the repetitive movement attempts by patient rather than the help from the robot [5], there are several studies that state using the robot can result in better improvement.

Based on some studies at Burke Rehabilitation Hospital in White Plains, New York, more than 200 patients used the InMotion^2 robot, and they all have regain motion while even for some of them the experts had lost hope for any improvements [7, 12].

3. InMotion^2 System

The InMotion^2 system combines a real-time Linux interface with a robotic arm that moves freely in the horizontal plane within two-dimensional space. The arm/handle is being controlled by two motors and encoders for sensing the x/y

position. The software controls the system by reading data from the sensors and writing data to the motors. Therefore the software can be manipulated to change and control the behavior of the system in a desired way (Picture 1 and 2) [6].



Picture 1

Picture 2

3.1. Tcl / TK

Knowledge of the Tcl scripting language is necessary to start experimenting with the InMotion^2 system. Tcl has a website that incorporates descriptions of all built in functions as well as a full description of syntax rules. The website is located at http://www.tcl.tk/.

The second tool being used is the TK windowing environment. The sample program in this paper illustrates an example of using TK with the Tcl scripting language.

3.2. System Basics

InMotion^2 hardware components are controlled through a data acquisition (DAQ) board in the PC computer, by reading data from and writing data to the analog to digital (a/d) and digital to analog (d/a) channels on the DAQ board [13].

The InMotion^2 software runs on Linux 2.4 kernel, and it has been improved with RTLinux3.1. The real time Linux provides low-latency interrupts for the system by running the Linux kernel as a subordinate task under micro-kernel. There are two different sets of programs, the User mode Linux programs and the control loop program. RTLinux3.1 and the robot control loop each run as a set of separate Linux Kernel Modules (LKM). At each sample period the InMotion^2 robot control loop performs the following major tasks:

- Read data from robot sensors
- Read data from reference sources (or files)
- Calculate controls based on input data and check safety
- Write control data to robot motors
- Write data to log channels (or log data)
- Write display data
- Wait for next tick

User mode programs such as reference source data or save log data which interact with the graphics display, communicate with the control loop using either

RTLinux real-time fifos or RTLinux shard memory buffers (mbuffs). A user mode C program can share memory areas with LKM [13].

In this system, the control loop program is written in the language of the Linux Kernel, C. The User mode programs (such as GUIs and data sources) can be written in both C and C++. However, the GUI of this system is written in TCL/Tk, because of the capacity and power of this language.

The changes in the system's motor torques will control the movement of the manipulandum. Therefore, to direct the movement of the robot's handle, the motor torques needs to change accordingly. InMotion^2 system is a robot that is used for interacting with the patient and guides patient to move his/her arm in certain patterns that are determined in the therapy exercises. There are many different exercises that a patient can do and should do to recover and regain his/her arm's motor skills. No matter what exercise and what pattern of movement the system is programmed for, the robot handle is the part that is moving around in order to execute the program.

Therefore there is a common action between all the exercises that are implemented by this robot, which is moving the handle around. However where the handle needs to move, depends the type of exercise, goal of the exercise, the feed back from patient therapist and perhaps the patient's condition.

3.3. The File Interface

The code to interact with the InMotion^2 system is separated out into three files of concern. These files are listed below:

Shm.c

This program gets and sets variables in the robot's shared memory *mbuffs*. It creates a hash table of string names, so that they can be searched quickly. It is a user mode program that provides command line access to a shared memory buffer.

shm.tcl

Tcl script functions that interface to the shm program (shm.c). This function library contains all necessary functions for interacting with the robot arm.

Vex

Tcl script sample, demonstrates how to interact with the system, as well as TK GUI interface that plots the position of the robot arm onto a 2D plane. In this file the state of variables such as force and acceleration are represented in the form of vectors.

3.4. The coordinate System

The vectors on the canvas panel start at the (0,0) origin point at the center of the canvas, which represents a point offset in Y .65 meters from the center of the robot motors. The variables in the text panel are shown with three digits of

precision after the decimal point, but they are stored in the software system as 64-bit floating point values with precision and accuracy bounded by the design of the sensors and algorithms used in the software system.

4. Methodology

The main focus of this paper is to show how to use the current functionalities and programs of InMotion^2 System to create and develop exercises that will help stroke patients in their recovery. The InMotion^2 system includes some working examples to choose from to start the development. As it has been mentioned in the previous section “vex” and “shm.tcl” are among the main programs and examples in this system. The *Vex* example was chosen as a starting point given that it exhibits the greatest insight to the inner workings of the InMotion^2 system.

The vex program is the GUI for a practice example that provides a real-time moving picture graphics or the state of the InMotion^2 system. In this section the general overview of how this program works is discussed as well as what has been modified to implement one of the sample exercises - Move from left to right. Vex program displays two panels. One is the canvas which displays the vectors and the other is the text window which shows the numeric representation of the vectors and their movements.

When the vex program starts, first the text window is displayed (Figure 1). This window has four buttons on the top, load, run, star and quit. *Load* button will load the LKM to he memory. The *run* button causes the control loop to start running and enable read/write commands from and to the system motors and sensors. The *Star* button runs the procedure that the vex GUI is representing. This program directs the robot handle between the origin and the eight compass directions that are represented by gray dots on the vex display (Figure 2). The *quit* button perceptibly, quits the vex application by pausing the robot, unloading the LKM and shutting down the shared memory.

Figure 1- Text window

As it is shown in figure 1, beside the buttons, the text window shows the value of different data resulting from the movement of the robot handle. Some of the variables it displays are such as x, y positions (mm), velocities (mm/sec), forces (Newton) and shoulder and elbow degrees.

The position of the virtual objects in the space is called *slots*. InMotion^2 system offers several slot controller functions. These functions control the motor torques of the robot based of the given data such as the current x, y position, velocity, stiffness torque and damping value [13].

One of the objectives of the InMotion^2 system is to have the patient move the manipulandum (the handle at the end of the arm) from point A to point B in preferably a straight line. Therefore, a slot controller function can be set up to control the motors and force the manipulandum back to the path when the patient moves it away. In general the slot control code in this system provides a framework to set up the desired slot paths as well as control its behavior.
One of the slot control examples is star. This example moves the manipulandum through a series of straight line slots back and forth from the center of the display to each of the eight compass points shown in the figure 2.
The star program will not run unless the LKM is loaded therefore the user should first press load button.

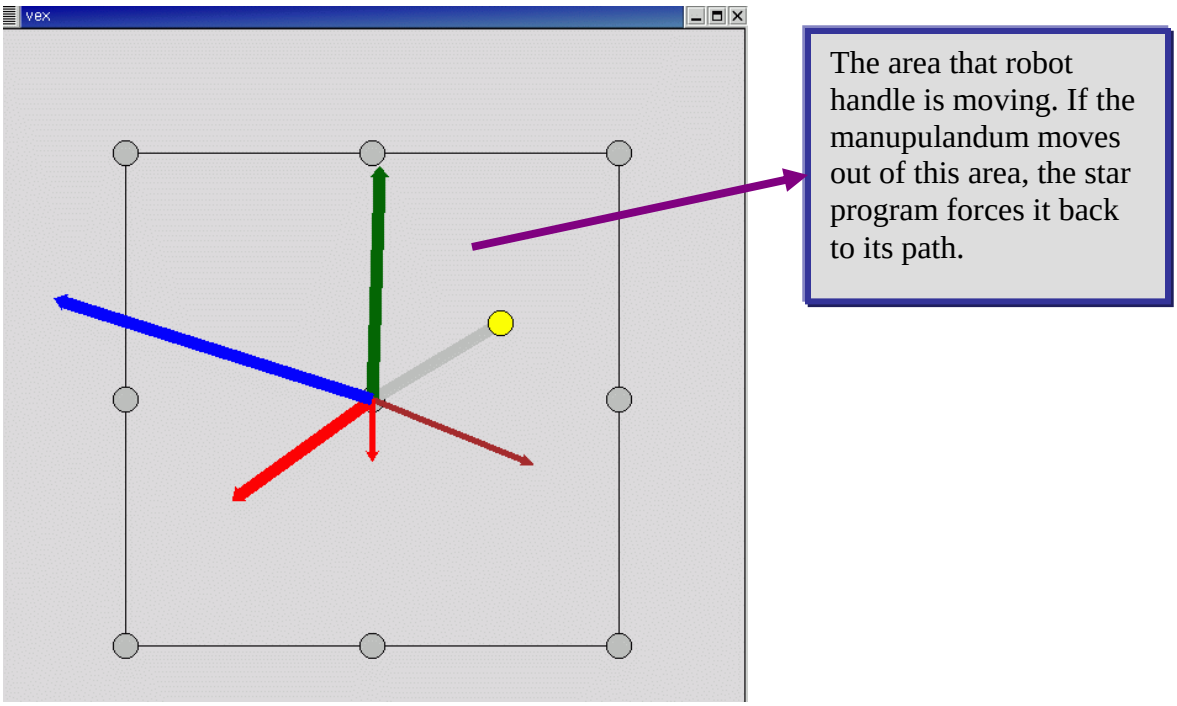


Figure 2- Original Vex GUI

As shown in figure 2, the vex program is representing the GUI that forces the user to move the robot handle to eight different positions.

For the illustration purposes and to step through the necessary changes for creating a new exercise, the second sample exercise-Move from left to right- has been implemented. For more details of the sample exercise refer to the section 5. The Vex has been modified to only move the robot from center to the left, back to the center then to the right and back to the center to replicate the sample exercise. This exercise only requires movement along the x axis and no movement in the Y direction as it is shown in figure 3.

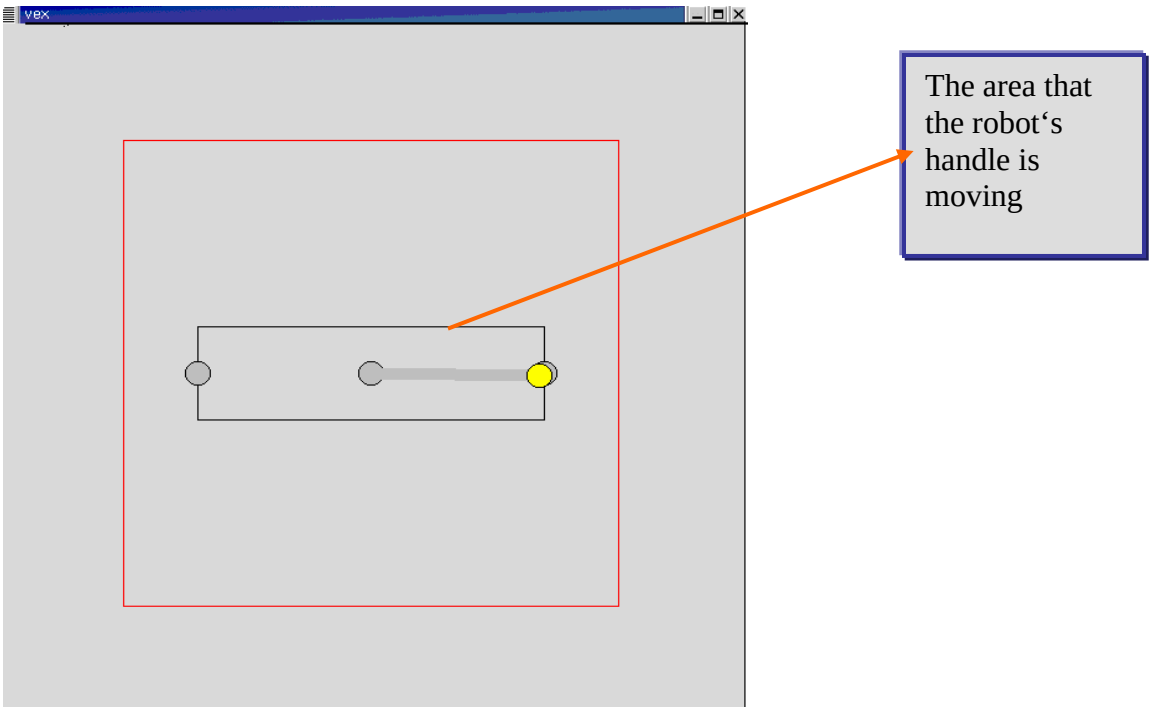


Figure 3- Modified vex GUI (representing the implemented example)

The black box in the center of vex represents the area that the movement of the manipulandum is controlled and if the user moves the robot arm out of this area, the controllers will draw the arm back to the path and within the area.

For this reason, for the sample exercise -Move from left to right- the GUI has been modified to only show the two targeted points that the manipulandum has to move to. Also since the arm needs to be moving only along the x-axis the height of the controlled area is reduced to enforce the movement of the maniulandum in a straight horizontal line. Any movement outside of this box will be met with an opposing force pushing the arm back into the area of the box. This reaction will force the robot handle to move in a straight line along the x- axis.

5. Exercises

As soon as an isolated muscle action is elicited, it must be practiced and extended into meaningful actions. This requires the patient gaining control over increasing ranges of a movement and be able to change to other movements which also requires the muscle to contract in its prime mover, synergist and fixture roles and shifting from concentric to eccentric in different parts of range at various speed.

Gross pattern of movement of the upper limb should be avoided as these will not allow either the therapist or the patient, to be aware of any minimal muscle activity present and they will tend to encourage only the more active muscles which may cause trauma around the shoulder [3, 6].

The major function of the arm is to enable the hand to be positioned for manipulation. Therefore, the essential components are:

- Shoulder abduction/hold
- Shoulder forward flexion/bend
- Elbow flexion and extension

The following are the questions that should be addressed for each exercise:

1. Description of the exercise.
2. When patients can do this exercise?
3. What are the factors/types of data that need to be recorded for this exercise?

- ❖ Age
- ❖ Time that patient needs to do the exercise (T_EX)
- ❖ Number of repetitions
- ❖ Time it takes for the patient to do one repetition (T_P)
- ❖ Force that the machine has to apply to the patient's hand
- ❖ Amount of the force that the patient is expected to apply while he/she is moving the handle

4. What type of data should each one of these factors be compared to?
   - ❖ Mostly to a healthy person the same age as the patient.
5. What are the graphs factors? What do these graphs represent?
6. How to determine the patient has mastered the exercise?

In this section, based on various researches that were done about recovery exercises for arm after stroke, several exercises are suggested that can be implemented and be used for our experiment purposes.

5.1 First Exercise

Converting the rolling chair exercise to the steps that can be implemented by the robot:

1. Description of the exercise:

Patient's arm position starts with 90 degree bend at the elbow (Figure 4-5). At that position the patient has to grab the handle (this is position Y =

0) and move the handle forward to position Y = y (along a straight line) and back to position Y=0.

Movement factors: the shoulder should not move. Only the hand should move forward and backward along a straight line.

Starting:

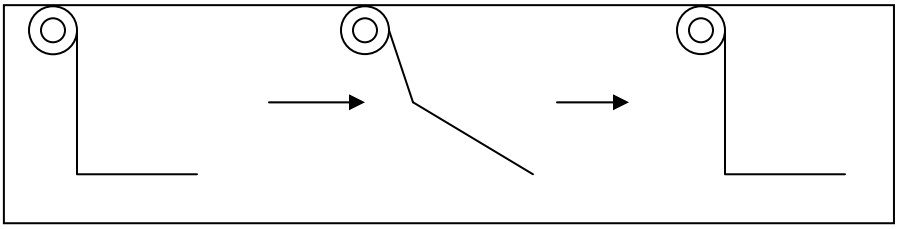


Figure 4

Eventually:

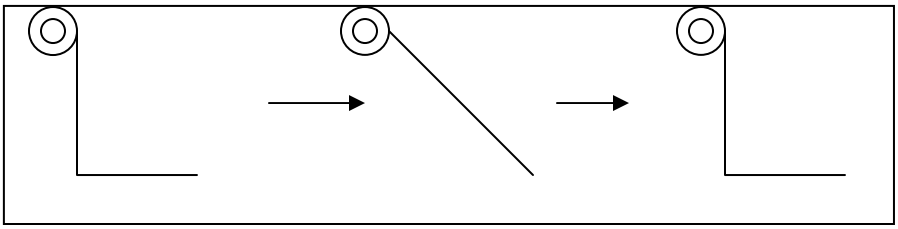


Figure 5

2. When patient can do this exercise?

When patient can grab the handle and be able to move their elbow from 90 degree to some larger angle where the elbow is extended.

3. What are the factors/types of data that need to be recorded for this exercise?

AGE: Depending on the age of the patient, different levels of force is expected to be applied by the patient. The force level should be compared to a healthy person of comparable age as the patient.

When a healthy person performs the exercise the amount of force that he/she applies to move the handle is measured. Keep in mind at this point the robot handle has the stiffness level equivalent to the force required to move a wheel chair. Therefore, a person has to apply the same amount of pressure to move the handle as if he/she had to move a wheel chair.

T_EX: 10 minutes (Can be varied).

NUMBER OF REPETITIONS: Record the number of times that a person moves the handle from Y= 0 to Y=y and back to Y=0.

T_P: How long it takes for the user to move the handle from Y= 0 to Y=y and back to Y=0.

FORCE THAT THE MACHINE HAS TO APPLY TO THE PATIENT'S HAND:

The robot handle needs a stiffness level equivalent to the weight of the force required to move a wheel by a healthy person, however depending on the age, the severity of the stroke and mobility/sensory level of patient the stiffness level should be adjusted.

4. What type of data should each one of these factors be compared to?

Same data as gathered from a healthy person of comparable age

5. What are the graph factors? What do these graphs represent?

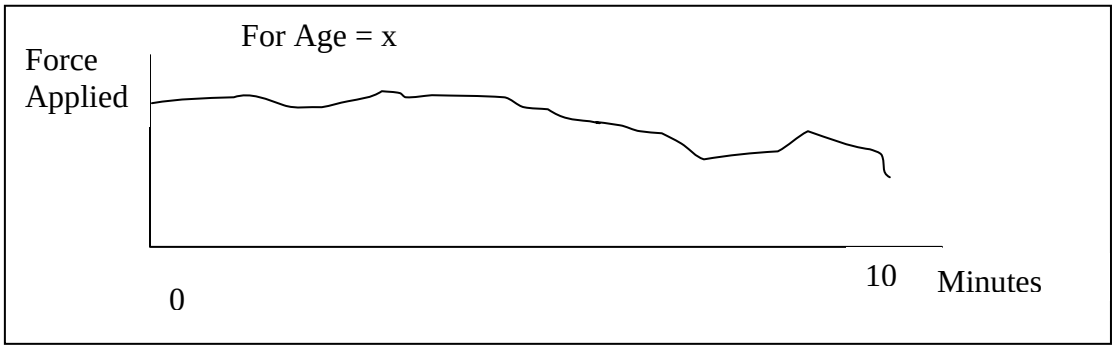


Figure 6: An imaginary graph – for illustration purpose only.

The force applied by a healthy person to move the handle back and forth for 10 minutes. Since after a few minutes, the person gets tired it is expected that the average force applied will go down over the time. Figure 6 displays an imaginary graph that demonstrates this pattern.

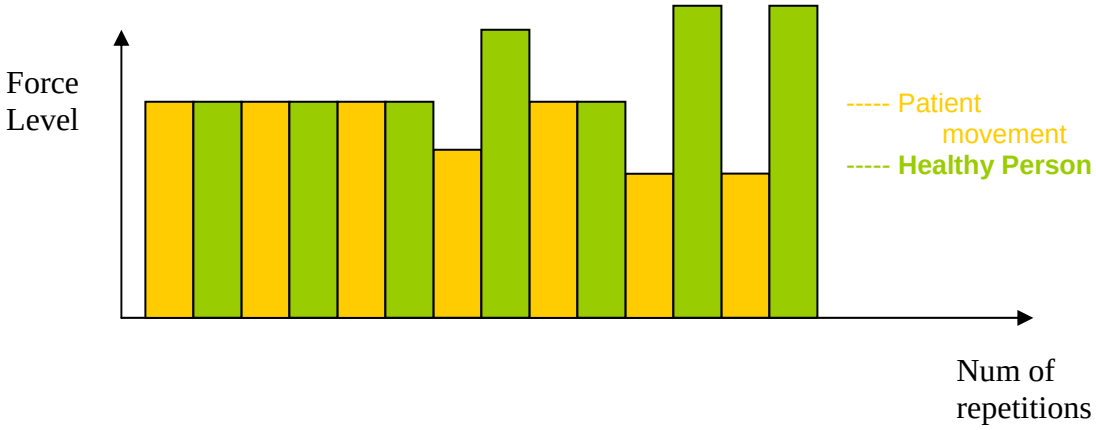


Figure 7: Number of repetitions in 10 minutes vs. force applied in each repetition.

We will generate the same type of graph (as the ones that were created for a healthy person) for the patient and we should compare the trace of the graph with the healthy person's graph (Figure 7). The key is the comparison of the pattern not the data for each patient. For instance if a healthy person applies less force after 6-7 minutes into the exercise because he/she got tired, we are looking for a similar decrease in force applied around the same time from the patient.

6. When patient has mastered the exercise?

As the patterns get closer to each other we are hoping that the patient is on their way to recovery.

5.2 Second Exercise

This is the example exercise, called "Move from left to right".

1. Description of the exercise:

Patient's arm position starts with 90 degree angle bend at the elbow.

At that position the patient has to grab the handle (this is position Y = 0, X= 0) and move the handle left to position X = -x (in the straight horizontal line) and back to position X=0 and again move right to position X= +x and back to X= 0.

Movement factors: Shoulder and elbow should not move. Elbow should maintain the 90 degree angle all the time. Only the forearm should move right and left in horizontal (X) direction.

2. When patient can do this exercise?

When the patient can grab the handle, position their affected hand at the 90 degree angle at the elbow and be able to move their forearm in right and left direction long the horizontal (X) axis

3. What are the factors/types of data that need to be recorded for this exercise?

AGE: Depending on the age of the patient and the severity of the stroke, different levels of force is expected to be applied by the patient as well as different range of motion. The force level and range of movement in the ("+" and "-"X) direction should be compared to that of a healthy person of comparable age.

A healthy person performs the exercise (for the same amount of time-10 min) and the amount of the force that he/she applies to move the handle is measured. We record the distance that he/she moves the handle in the x direction. At this point the handle has no stiffness. We can also record the time and calculate how fast the person moves the handle right and left. Then the same exercise can be repeated for different stiffness levels of the robot handle.

So important factor is how stiff the handle should be or what is the range of stiffness that the robot handles should have. We can compare the stiffness of the handle to the weight that a person can move around. The harder it is to move the robot handle, heavier the weight the person is moving around and therefore, the stronger force he or she is applying to the handle. We can measure and analyze the force as we increase the stiffness of the machine

T_EX: 10 minutes (Can be varied)

NUMBER OF REPETITIONS: Record the umber of times that a person moves the handle from X= -x to X= +x for each level of stiffness, as well as how far he moves the handle in both directions. Also we can measure the force that the patient applies. As the patient recovers he/she can move the handle with considerable ease and can apply more force for higher stiffness levels.

T_P: How long it takes for the patient to move the handle from X= 0 to X=-x to X= +x and back to X=0 depending on the stiffness of the handle.

FORCE THAT THE MACHINE HAS TO APPLY TO THE PATIENT'S HAND:

The handle should be programmed for different stiffness level. Also the maximum stiffness level should depend on the patient's age and the severity of the stroke.

4. What type of data should each one of these factors to be compared to?

As was mentioned before, we can measure the force that the patient applies to move the handle, how far he/she moves the handle along each x direction, how long it takes for him/her to complete one repetition of the exercise as well as how many repetitions the person completes within the given time for each level of the stiffness.

The same data should be recorded from the performance of a healthy person. Plot a graph of the recorded data from a healthy person and compare the pattern of the two graphs.

5. What are the graph factors? What do these graphs represent?

We are hoping to see the same pattern of movement of a healthy person for the patient over time. However since the patient might get tired faster and lose his/her energy, the force level might be lower than the healthy person.

6. When patient has mastered the exercise?

Patient should be able to move the handle to the desired X position given a particular level of stiffness with considerable ease.

5.3 Third Exercise

Straight hand

1. Description of the exercise:

Starting position: Patient's arm is outstretched (elbow is extended at180 deg) parallel to the floor and 90 degree angle with the body (Figure 8). At this position the patient grabs the handle (this is position X = 0, Y = 0) and move the handle back and forth in the x axis direction. Starting at position X = 0 go to position X = -x, back to X=0, then go all the way to going forward to X= +x and back to the original position X=0. The distance of –x and +x is determined by the movement ability of the patient. The goal is that the patient should eventually be able to move his\her arm in the full range of motions.

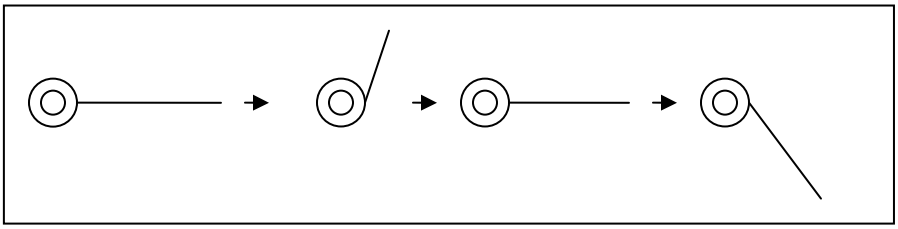


Figure 8- Patient's arm movement pattern.

2. When patient can do this exercise?

When patient can grab the handle and hold onto it, be able to hold his/her arm outstretched in front the robot and move the arm back and forth along the x direction.

This is perhaps a more advanced exercise compared to the previous two exercises since the patient should be able to hold the handle as well as be able to maintain the straight arm position.

3. What are the factors/types of data that need to be recorded for this exercise?

AGE: Depending on the age of the patient, and the severity of the stroke, the patient is expected to be able to move the handle to certain X distance. The movement of the patient's arm has to be compared to the movement of a healthy person of comparable age. Depending on how far the patient can move the handle while keeping his/her arm straight in front, the force that is applied by the handle should be adjusted. It is adjusted such that the handle can help the patient move his/her arm to the positions that a healthy person would. The patient should work within the range he/she can control the handle, and gradually increase the range.

T_EX: 10 minutes

NUMBER OF REPETITIONS: Record the number of times that a person moves the handle from X= 0 to X=-x, to X=0, to X=+x back to X = 0. Number of repetitions should be about 10-15 at a time.

T_P: How far the patient can move his/her handle along the x direction while he/she is maintaining an outstretched arm.

FORCE THAT THE MACHINE HAS TO APPLY TO THE PATIENT'S HAND:

The handle should have no stiffness at the beginning. However it has to apply enough force to direct the patient's hand to an acceptable +, - X position for that patient. Hopefully as the patient repeats this exercise he/she can eventually move his/her arm in full range of motion by just holding the handle and does not need any guiding force from the robot.

4. What type of data should each one of these factors be compared to?

The force applied by a healthy person to move the handle back and forth as well as the number of times he/she was able to do this exercise within the given time.

5. What are the graph factors? What do these graphs represent?

The graph should show the amount of the guiding force that the robot applies to move the hand to the desired x position for each set of exercise. It can be the average force for 15-20 set repetitions or the average of several sets of 15-20 repetition for a day of exercise. Hopefully as the patient repeats the exercise, the affected arm recovers and can move the handle with less help from the machine.

Also compare how far in the X direction can the patient move the handle with that of the desired distance that has been measured from a healthy person.

6. When patient has mastered the exercise?

The patient has mastered the exercise when he/she can move his/her arm in a straight line in the full motion range from +x to –x with the arm outstretched.

5.4 Fourth Exercise

Repeating the same movement as the third exercise but this time the outstretched arm should be at a 45 degree angle from the shoulder.

5.5 Fifth Exercise

Patient's arm position starts with 90 degree bend at the elbow.

At that position the patient has to grab the handle (this is position Y = 0, X=0) and rotate the wrist, from left to right and back.

Movement factors: Through out this exercise the entire hand does not move and only the wrist rotates [10]**.**

6. Code Explanation

In this section the function calls that are required to move the robot's arm are discussed. First section explains commands in general. Second part explains the

function calls that are required to move the robot handle from one place to another, regardless of what the exercise is.

6.1 Commands Explanation

The “canvas” command creates a new window and makes it into a canvas widget (w). Additional options in the command line can specify the aspects of the canvas such as its color or the size [13].

set c [canvas $w.c -height 600 -width 600]

This line displays a square canvas window with the dimensions of 600 in height and width.

chwin $c [winfo width $c] [winfo height $c]

“Winfo” returns the window related information, such as the height and the width. “Chwin” is a function that handles window resizing.

Centxy[$x $y rad]

“Centxy” function is defined in shm.tcl file. This function returns a list of four elements x-rad, y-rad ,x+rad, y+rad given the x , y and the rad value. For instance if 100,100 and 10 is passed, the list (90, 90, 110, 110) is returned.

$c create rect [centxy 0 0 .14]

The Key word “create” creates the object specified after it. Therefore, this line creates a square with .14 meters in dimensions. This code is used to draw a black box (or a square) where the dots of the compass are lying (Figure 2). Therefore, if the size of this area needs to be changed, the values passed to this function have to be changed accordingly (Figure 3).

$c create oval [centxy $I $j .01] -fill gray

“Fill” Simply fills in the created area in this case the oval shaped object with the color that is specified after it which in this case is gray color.

Each item in the canvas widget is named in either by *id* or by *tag.* Each item has its own unique identifying number *id* which gets assigned to it when it is created. This number never gets changed or reused within the lifetime of a canvas widget. In addition each item can also have a *tag* number associated with it, which is not a unique number to the item. A *tag* may be associated with many different items to group them in different ways.

*$c create line {0 0 0 0} *

-tags "kscale linep" -width $linewidth -fill gray -capstyle round

The above statement simply draws a gray line at the given line width and a round cap at the end of it (Figure 2) for the items with same *tag* name "*kscale linep"*.

proc init {}{ body.}

The **"**proc**"** command creates a new Tcl procedure for instance named init, and replaces any existing command or procedure that may have been referred by that name.
Whenever the new command is invoked, the contents of *body* will be executed by the Tcl interpreter.

gets *channelId* ?*varName*?

"Gets" command reads a line from a channel. This command reads the next line form *channelId*, returns everything in the line up to (but not including) the end-of-line character(s). If *varName* is omitted the line is returned as the result of the command. If *varName* is specified then the line is placed in the variable by that name and the return value is a count of the number of characters returned [19].

lindex list {indice}

The” lindex” command accepts a list parameter and zero or more indices into the list and returns the requested index of the list. Keep in mind that 0 refers to the fist element of the list. The indices may be presented either consecutively on the command line, or grouped in a Tcl list and presented as a single argument [19].

set what [lindex $istr 3]

return $what

In general, “lindex” retrieve an element from a list. For instance, the above statement sets the value of “*what*” variable to the 4th element of the “*istr*” list and returns the value of “what”.

foreach *varname list {body}*

foreach *varlist1 list1 ?varlist2 list2 ...? {body}*

The *“foreach”* command implements a loop where the loop variable(s) take on values from one or more lists. For each element of *list* (in order from first to last), *foreach* assigns the contents of the element to *varname*

Exp {int (x) + int (y)}

“int” returns the integer value of the variable and “exp” returns the final result of the mathematical expression. For instance, If the value of $sec is 100, the following expression will set the value of atime2 for star to 200000 msec.

```
set star(atime2) [expr {int(2 * $sec * 1000)}]
```

The following section explains some of the general functions that are involved in moving the robot's handle. For the purpose of the demonstration the second suggested exercise –Move from left to right -is implemented and it is used as an example to explain what needs to be changed to implement the exercise. The comments are included in the source code.

6.2 Function Explanation

VEX.tcl File:

The following lines actually draw the dots at desired places and fill them with gray color.

```
foreach I {-.14 0.0 .14}                    //Horizontal line, (i) values
 { foreach j {-.14 0.0 .14}                  //Vertical line, (j) values
          {.
              $c create oval [centxy $I $j .01] -fill gray
          }
}
```

This code draws the circles at compass points and the center. Given x, y and a value for radius of the dots, "centxy" returns x1 y1 x2 y2 where the ovals dots are drawn (refer to Figure 2). It draws a circle where the robot needs to be moved to which in the original code were the 8 dots along the square.

Since in the move from left to right exercise there are just two points that the arm needs to be moved to, the function has been modified to only display two dots on the sides of the rectangular box (Figure 3).

In order to have only two circles on the left (West) and the right (East) the functions is modified as follow:

```
foreach I {-.14 0.0 .14}
{foreach j {0.0 0.0 0.0}                    // We are not drawing any compass pint
        {
        $c create oval [centxy $I $j .01] -fill gray
        }
}
```

The following lines deal with creating and drawing vectors.

```
$c create line {0 0 0 0} \
        -tags "kscale linep" -width $linewidth -fill gray -capstyle round
```

This command creates a line (or vector) indicating the change of the variables such as force and velocity. Since in this exercise showing the change of x, y and the velocity vectors were not necessary, these lines are commented out.

shm.tcl file

The shm program allocates the shared memory buffers that are needed by the program. *Ob* (general objects) is one type of these allocations. Others are *Rob* (robot data), and *Daq* (data acq data).

A typical need in a robot GUI program is to query the control loop for the current position of the manipulandum. This is a similar idea to getting the x/y cursor position from a mouse driver on a PC windowing system [13]. For this reason every function that is involved with manipulating variables based on the interaction of the user and the robot, first grasps an object by declaring a global *ob*.

The main two functions in this program are “wshm” and “rshm”.
“wshm” writes to a variable or updates a variables(“puts”) by a value of the memory location. Basically, it writes system tcl variables from *where* the value is equal *what* (like /sbin/sysctl -w where=what).

```
proc wshm {where what {I 0}} {
   global ob
   if {![info exists ob(shm)]} {
         return
   }
   if [info exists ob(shm_puts_exit_in_progress)] {
         return
   }
   shm_puts "s $where $I $what"

   gets $ob(shm) istr
   set what [lindex $istr 0]
   if {[string equal $what "?"]} {
         puts stderr $istr          } }
```

“*rshm”* reads memory locations or system tcl variables(“return”) from *where* (like /sbin/sysctl where). The memory location, specified by values of x, y is passed on and the value of it is retuned.
This is an important function and it has been utilized through out this file several times.

```
proc rshm {where {I 0}} {
    global ob
    set what "???"
    if {![info exists ob(shm)]} {
        return "0.0"
    }
    if [info exists ob(shm_puts_exit_in_progress)] {
        return "0.0"
    }
    shm_puts "g $where $I"

    gets $ob(shm) istr
    set what [lindex $istr 0]
    if {[string equal $what "?"]} {
        puts stderr $istr
        return "0.0"
    }
    set what [lindex $istr 3]
    return $what
}
```

“movebox” is implementing the *Slot* technique by moving the robot’s arm from one spot to the other. This is one of the most important functions and has been called several times through out this file.

```
proc movebox {slot_id slot_fnid forlist box0 box1} {

    # the uplevel/subst allows users to put $vars in the lists.
    set forlist [uplevel 1 [list subst -nocommands $forlist]]
    set box0 [uplevel 1 [list subst -nocommands $box0]]
    set box1 [uplevel 1 [list subst -nocommands $box1]]

    foreach {slot_I slot_term slot_incr} $forlist break
    foreach {slot_b0_x slot_b0_y slot_b0_w slot_b0_h} $box0 break
    foreach {slot_b1_x slot_b1_y slot_b1_w slot_b1_h}

   foreach I {
        slot_id slot_fnid
        slot_I slot_term slot_incr
        slot_b0_x slot_b0_y slot_b0_w slot_b0_h
        slot_b1_x slot_b1_y slot_b1_w slot_b1_h
    } {wshm $I [set $I]}

    wshm slot_running 1
```

```
    wshm slot_go 1
    wshm slot_max 1}
```

slot_id is a slot management code function that stops slot # id which is default to zero. *Forlist* is a list of mainly three elements {0, $star(time), 1}, which represent the tick time. *Box0* is the current position of the handle (slot) and *box1*is the position that we want the handle to move to.

Uplevel evaluates commands in different scope. Therefore uplevel 1 means to execute the command in the scope of calling procedures or the user level.

Break returns a TCL_BREAK code, which causes a break exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then aborts its execution and returns normally. There is no need to make any changes to this function. This function is being called from several different functions such as “center_arm”, “star_proc” which will be introduced later on.

“center_arm” moves the robot’s arm from its current position to center, at a constant speed.

```
proc center_arm {{cx 0.0} {cy 0.0}} {
        set x [getptr x]                    //Gets the horizontal value of current
                                            position of the robot’s arm and sets it to x.
        set y [getptr y]                    //Gets the vertical value of current position
                                            of the robot’s arm and sets it to y.
        set dist [edist $x $y $cx $cy]      //Sets dist  to Euclidean distance of x, y, cx
                                            and cy.
        set ticks [expr {int($dist * 4000.)}]       //Sets the ticks value (which is the
                                                    tick time) to the integer value of
                                                    dist*4000.
        movebox 0 0 {0 $ticks 1} {$x $y 0.0 0.0} {$cx $cy 0.0 0.0}
}
```

At the end after knowing the tick time, start position and the end position this function calls the *movebox* function to actually move the robot's arm from its current position (start place) to the center (end place) within *ticks* time.

When the "star" button on the text window is clicked, there are three functions that will get called. First the "star_once" function gets called. This function initializes the star process and the compass points that the robot arm must move to. Next, "Start_star" function places the arm in its initial position which in the case of vex, it is set to the center of the box and calls "star_proc". Finally, "star_proc" function actually executes the star procedure which is directing the maniupulandum through all the compass points as it is shown in Figure 2.

"star_once", first checks to see if star is already in process or in the other word has the "star" button been already clicked. If not, it will initialize its variables.

```
proc star_once {} {

    global star
    if {[info exists star(l)]} {
    return                    // If the function is in process don't do anything
    }                         // Else
    set star(i) 0             //Sets star(i) to zero

    set star(hw) 0.005        //Set star(hw) to constant value 0 .005. hw is just a
                               constant
    set star(c)  [list  0.0  0.0 $star(hw) $star(hw)]    //Sets the center position to  0
    set star(s)  [list  0.0 -0.14 $star(hw) $star(hw)]    //The south position is at
                                                           -.14 below the center
    set star(n)  [list  0.0  0.14 $star(hw) $star(hw)]   //The North position is
                                                          at +.14 above the center
    set star(w)  [list -0.14  0.0 $star(hw) $star(hw)]   //And so on ..
    set star(e)  [list  0.14  0.0 $star(hw) $star(hw)]
```

```
    set star(nw) [list -0.14  0.14 $star(hw) $star(hw)]
    set star(ne) [list  0.14  0.14 $star(hw) $star(hw)]
    set star(sw) [list -0.14 -0.14 $star(hw) $star(hw)]
    set star(se) [list  0.14 -0.14 $star(hw) $star(hw)]
    set star(dirs) {n ne e se s sw w nw}          //"dirs" contains the list of all
the directions
}
```

The example exercise -Move from left to right.- has only two compass points (east and west) that the robot arm is moving between, however the starting position is still the center or at (0,0). For this reason the "star_once" function is modified as follow:

```
proc star_once {} {

   global star
   if {[info exists star(i)]} {
        return
   }
   set star(i) 0

   set star(hw) 0.005
   set star(c)  [list  0.0  0.0 $star(hw) $star(hw)]
   set star(w)  [list -0.14  0.0 $star(hw) $star(hw)]
   set star(e)  [list  0.14  0.0 $star(hw) $star(hw)]
  set star(dirs) {e w}
}
```

The start_star{x} function starts the actual star program after x amount of time by centering the robot arm. In this case the star program starts after 5 second.
To position the arm, "start_star" function sets the number of trips to zero, since we don't want the arm starts the movements yet and calls the "center_arm" function(this function has been explained already) to position the robot's arm in the center of the window**.** Then "star_proc" is called after a given time (for

instance 5 sec) to start the laps for vex program. There is no change to this function for the example exercise.

```
proc start_star {{sec 5}} {
        global star

        set star(trips) 0
        # takes 1 second
        center_arm                  // First trip (putting the handle at the center
                                    position) takes 1 second
        # after 2 seconds
        lappend star(afters) [after 2000 star_proc $sec]   //….
}
```

The **"**star_proc**"** function begins the star process by creating a *star* object. First check if the object already exists if not it starts the run. It calculates the speed at which the handle has to move by using the expression (2* $sec*100). Knowing that it will start from center, it calls the *movebox* function to get to the next direction.

```
proc star_proc {{sec 5}} {
        global star ob
        if {!$ob(running)} {     //If the star process is already running, meaning is in the
                                  middle of a trip through all the compasses, return.
                return
        }
        incr star(trips)          //Otherwise, increment the round trip number by one
        set star(sec) $sec
        set star(atime) [expr {int($sec * 1000)}]  //atime is a constant time for pausing
                                                    between the trips (mSec)

        set star(atime2) [expr {int(2 * $sec * 1000)}]

        set star(dir) [lindex $star(dirs) $star(l)]          //Finds the next direction that it has
                                                               to go
        set star(stime) [expr {int($sec * .9 * 200)}]        //stime is secs * sample time in Hz
                                                               (one trip time)

        movebox 0 0 {0 $star(stime) 1} $star(c) $star($star(dir)) //moves from center to
                                                                    the direction passed
```

```
// then after “atime”, from that direction move back to the center
set star(afters) [after $star(atime) {movebox 0 0 {0 $star(stime) 1}
$star($star(dir)) $star(c)}]
//After the puss time (atime2) call the star_proc again to move from the center to
//the next direction
lappend star(afters) [after $star(atime2) star_proc $star(sec)]

set star(I) [expr {($star(I) + 1) % 8}]          //calculate the next index of “dirs”
                                                  array to determine where to go next

}
```

Star(i) contains the index of directions array. Since in the example program there are only tow target points the star(i) is calculated based on that and it is modified to the following:

```
set star(I) [expr {($star(I) + 1) % 2}]
```

7. Conclusion

Using InMotion^2 system for stroke patient recovery has significant advantages in quality and cost of care. Using the robot can minimize involvement of the therapist and therefore dramatically decrease the cost for the patient or the insurance companies. The system would also have a higher degree of accuracy than the human would and therefore it will increase the effectiveness of the therapy.

The key objective of this work was to recommend some basic arm exercises for the stroke patients and to implement one of them as an example to show that exercises outlined can be used as a fundamental module for implementing more

advanced exercise. Therefore, once basic system movements are implemented the additional exercises can be developed fairly easily and that will increase sophistication of the system.

This system besides the vex example, includes several other simpler examples that requires the user interaction with GUI, such as following a ball or a dot shown in the screen that is moving in the pattern by the manipulandum. Some of these exercises are:

Chase.tcl- In this sample program there are two object balls on a canvas. The small yellow ball is a cursor that in this case follows the mouse. The large red ball is the target. The goal is to touch the target ball with the cursor ball. The Target ball is moving at random.

Xy1- This program print x/y in response to new line.

Xy2- script that prints position and velocity once per second.

One InMotion^2 robot, which now costs about $70,000 to install in a clinic, can provide intensive therapy for hundreds of patients. Even as an adjunct to a human therapist, the robot will be cost-effective, and will eventually pay for itself. A long-term goal is for people to be able to rent or buy the robot for home use, but that is maybe five years in the future according to Susan Fasoli, a post-doctoral fellow at MIT's department of mechanical engineering [4].

Appendix

Shm.tcl source code

```
# tcl i/o with shm (user mode shared memory buffer) program
# sourced by other tcl scripts

# InMotion2 robot system software for RTLinux

# Copyright 2003-2004 Interactive Motion Technologies, Inc.
# Cambridge, MA, USA
# http://www.interactive-motion.com
# All rights reserved

if {[info exists env(CROB_HOME)]} {
    set ob(crobhome) $env(CROB_HOME)
} else {
    set ob(crobhome) /home/imt/crob
}

proc every {ms body {id ::after_id}} {
    eval $body
    set $id [after $ms [info level 0]]
}

proc procname {} {return [lindex [info level -1] 0]}

proc cancel_afters {} {
       foreach id [after info] {after cancel $id}
       foreach id [after info] {after cancel $id}
}

# reap zombie processes after "exec &" commands exit
# see: http://mini.net/tcl/1039
proc reap_zombies {} {
    catch {exec ""}
}

# flip y coordinate

proc y_up args {
    set ret ""
    if {[llength $args]==1} {set args [lindex $args 0]}
    foreach {x y} $args {lappend ret $x [expr {-$y}]}
    return $ret
}

# given a center position and radius, like 100 100 10,
# centxy returns x1 y1 x2 y2, like 90 90 110 110.

proc centxy {x y rad} {
    set x1 [expr {$x - $rad}]
    set y1 [expr {$y - $rad}]
    set x2 [expr {$x + $rad}]
    set y2 [expr {$y + $rad}]
    list $x1 $y1 $x2 $y2
}

proc centertag {w tag} {
       foreach {x1 y1 x2 y2} [$w coords $tag] break
       set x [expr {$x1 + $x2 / 2.}]
       set y [expr {$y1 + $y2 / 2.}]
       list $x $y
}



# lkm loaded?
```

```
proc is_lkm_loaded {} {
    file exists /proc/pwrdaq
}


# load lkms

proc start_lkm {} {
    global ob
    if {![file executable $ob(crobhome)/go]} {
        error "start_lkm: could not run go"
    }

    # set status [catch {tk_exec sh $ob(crobhome)/go} result]
    set status [catch {exec sh $ob(crobhome)/go} result]
    if { $status != 0 } {
        stop_lkm
        error "start_lkm: could not start kernel module robot.o\n\
        result string:\n<<\n$result\n>>\n"
    }
}

# unload lkms

proc stop_lkm {} {
    global ob
    if {![file executable $ob(crobhome)/stop]} {
        puts "stop_lkm: could not run stop"
        exit 1
    }

    set status [catch {tk_exec sh $ob(crobhome)/stop} result]
    set status [catch {exec sh $ob(crobhome)/stop} result]
    if { $status != 0 } {
        puts "stop_lkm: could not stop kernel module robot.o"
        puts "result string:\n<<\n$result\n>>\n"
    }
}

# start shm - the shared memory buffer C program

proc start_shm {} {
    global ob
    if {! [file exists $ob(crobhome)/shm] } {
        puts stderr "start_shm: can't find shared memory program $ob(crobhome)/shm"
        exit 1
    }
    set ob(shm) [open "|$ob(crobhome)/shm" r+]
    fconfigure $ob(shm) -buffering line
    after 100
    set check [rshm last_shm_val]
    if {$check != 12345678} {
        puts "start_shm: bad shm check value."
        puts "make sure all software has been compiled with latest cmdlist.tcl"
        exit 1
    }
}

proc stop_shm {} {
    global ob
    if {![info exists ob(shm)]} {
        return
    }
    set ob(loaded) 0
    puts $ob(shm) "q"
    close $ob(shm)
    unset ob(shm)
}

proc start_log {logfile {num 3} {uheaderfile ""}} {
    global ob
```

```
    # puts "start_log $logfile $num"
    wshm nlog $num

    # make sure the dir is there
    file mkdir [file dirname $logfile]

    # write log header
    logheader $logfile $num $uheaderfile

    set ob(savedatpid) [exec cat < /dev/rtf1 >> $logfile &]
}

proc stop_log {} {
    global ob

    # puts "stop_log"
    wshm nlog 0
    if [info exists ob(savedatpid)] {
        exec kill $ob(savedatpid)
        unset ob(savedatpid)
    }
}

proc xyplot_log {filename} {
        global ob
        exec [file join $ob(crobhome) xygp] $filename &
}

proc plot_log {filename {plotcmd {}}} {
        global ob
        exec [file join $ob(crobhome) gp] $filename $plotcmd &
}

# if the shm process gets killed from outside, the puts here will fail.
# this will set shm_puts_exit_in_progress, and cleanup should happen.
# don't call stop_shm or stop_loop, since these just do more i/o to the
# now broken shm channel.

proc shm_puts str {
    global ob
    if [info exists ob(shm_puts_exit_in_progress)] {
        puts stderr "shm_puts error, exit in progress..."
    }
    if [catch {puts $ob(shm) $str}] {
        set ob(shm_puts_exit_in_progress) 1
        puts stderr "shm_puts error, stopping lkm."
        stop_lkm
        exit 1
    }
}

# i is array index in both.

# wshm writes systcl vars
# like /sbin/sysctl -w where=what

proc wshm {where what {i 0}} {
    global ob
    if {![info exists ob(shm)]} {
        return
    }
    if [info exists ob(shm_puts_exit_in_progress)] {
        return
    }
    shm_puts "s $where $i $what"

    gets $ob(shm) istr
    set what [lindex $istr 0]
    if {[string equal $what "?"]} {
        puts stderr $istr
```

```
    }
}

# rshm reads systcl vars
# like /sbin/sysctl where

proc rshm {where {i 0}} {
    global ob
    set what "???"
    if {![info exists ob(shm)]} {
        return "0.0"
    }
    if [info exists ob(shm_puts_exit_in_progress)] {
        return "0.0"
    }
    shm_puts "g $where $i"

    gets $ob(shm) istr
    set what [lindex $istr 0]
    if {[string equal $what "?"]} {
        puts stderr $istr
        return "0.0"
    }
    set what [lindex $istr 3]
    # set what [expr int($what * 1000)]
    return $what
}

proc start_loop {} {
    wshm paused 0
}

proc stop_loop {} {
    wshm paused 1
}

proc mouse_getptr {p} {
    expr {[winfo pointer$p $::tachw]}
}

proc mouse_getvel {p} {
    # delta motion since last tick
    set d($i) [expr {$p($i) - $::lastp($i)}]
    # velocity in pixels/sec
    set v($i) [expr {$d($i) * $::hz}]

    # smooth $v
    set v($i) [iirsmooth $v(i)]
}

# robot ptr/vel

proc getptr {p} {
    rshm $p
}

proc soft_getvel {p} {
    rshm soft_${p}vel
}

proc fsoft_getvel {p} {
    rshm fsoft_${p}vel
}

proc tach_getvel {p} {
    rshm tach_${p}vel
}

proc getvel {p} {
    rshm ${p}vel
}
```

```
proc gettrq {p} {
    rshm ${p}_torque
}

proc getvolts {p} {
    rshm ${p}_volts
}

proc getfrc {p} {
    rshm ${p}_force
}

proc getftfrc {p} {
    rshm ft_${p}dev
}

proc getwftfrc {p} {
    rshm ft_${p}world
}

proc f3k {n} {
    format %.3f [expr {1000.0 * $n}]
}

proc f3 {n} {
    format %.3f $n
}

# checkerror should be called from inside a user-mode program event loop,
# to make sure that the event loop isn't generating too many errors or
# warnings.  a few such errors are expected in normal operation, but if
# something goes wrong, they should happen on every sample and they will
# exceed the max (10, for example) quickly.

proc checkerror {{max 10}} {
    global ob
    set ob(errormax) max
    set nerrors [rshm nerrors]
    if {$nerrors > $ob(errormax)} {
        set i [rshm errorindex]
        set ei $i
        set error0i [rshm errori $i]
        set error0code [rshm errorcode $i]
        incr i -1
        set error1i [rshm errori $i]
        set error1code [rshm errorcode $i]
        incr i -1
        set error2i [rshm errori $i]
        set error2code [rshm errorcode $i]

        set estring "InMotion2 System, Pausing control loop.\n\
nerrors = $nerrors,\n\
last ($ei): iteration = $error0i, code = $error0code.\n\
last-1: iteration = $error1i, code = $error1code.\n\
last-2: iteration = $error2i, code = $error2code.\n\
You may run shm to analyze system state, then run ./stop ."
        stop_loop
        error $estring
        # stop_lkm
        # after 2000
        # exit 1
    }
}

# movebox to move a box
#              i f  forlist    from                  to
# e.g.: movebox 0 0 {0 1000 1} {0.0 0.0 0.005 0.005} {0.15 0.15 0.005 0.005}

proc movebox {slot_id slot_fnid forlist box0 box1} {
# puts "movebox $slot_id $slot_fnid $forlist $box0 $box1"
```

```
    # the uplevel/subst allows users to put $vars in the lists.
    set forlist [uplevel 1 [list subst -nocommands $forlist]]
    set box0 [uplevel 1 [list subst -nocommands $box0]]
    set box1 [uplevel 1 [list subst -nocommands $box1]]

    foreach {slot_i slot_term slot_incr} $forlist break
    foreach {slot_b0_x slot_b0_y slot_b0_w slot_b0_h} $box0 break
    foreach {slot_b1_x slot_b1_y slot_b1_w slot_b1_h} $box1 break

    foreach i {
        slot_id slot_fnid
        slot_i slot_term slot_incr
        slot_b0_x slot_b0_y slot_b0_w slot_b0_h
        slot_b1_x slot_b1_y slot_b1_w slot_b1_h
    } {wshm $i [set $i]}

    wshm slot_running 1
    wshm slot_go 1
    wshm slot_max 1

}

# stop a slot currently in progress

proc stop_movebox {slot_id} {
    wshm slot_max 0
    foreach i {
        slot_id slot_fnid
        slot_i slot_term slot_incr
        slot_b0_x slot_b0_y slot_b0_w slot_b0_h
        slot_b1_x slot_b1_y slot_b1_w slot_b1_h
        slot_running
    } {wshm $i 0}
    wshm slot_go 1
}

proc star_once {} {
    global star

    if {[info exists star(i)]} {
        return
    }

    set star(i) 0

    set star(hw) 0.005
    set star(c)  [list  0.0  0.0 $star(hw) $star(hw)]
    #set star(s)  [list  0.0 -0.14 $star(hw) $star(hw)]
    #set star(n)  [list  0.0  0.14 $star(hw) $star(hw)]
    set star(w)  [list -0.14  0.0 $star(hw) $star(hw)]
    set star(e)  [list  0.14  0.0 $star(hw) $star(hw)]
    #set star(nw) [list -0.14  0.14 $star(hw) $star(hw)]
    #set star(ne) [list  0.14  0.14 $star(hw) $star(hw)]
    #set star(sw) [list -0.14 -0.14 $star(hw) $star(hw)]
    #set star(se) [list  0.14 -0.14 $star(hw) $star(hw)]
   # set star(dirs) {n ne e se s sw w nw}
   set star(dirs) {e w}
}

star_once

proc start_star {{sec 5}} {
        global star

        #while (1) {
        #        set x [getptr x]
        #       set y [getptr y]
        #       puts "xvalue is: $x"

        #              if {$x > .14} {
```

```
        #              puts " iam here"

        #      after 50 {movebox 0 0 {0 0 0} {x y 0.0 0.0} {.14 0.0 0.0 0.0}}
        #              puts " now i ma herjelkjreklj"
        #              } elseif {$x < -.14} {
        #              puts " i am after -.14"
        #      after 50 {movebox 0 0 {0 0 0} {x y 0.0 0.0} {-.14 0.0 0.0 0.0}}
#puts " i am aftere hrlkejrlkejklejrlek"
        #      }
        #}


#theirs

        set star(trips) 0

        # takes 1 second

        center_arm

        # after 2 seconds

        lappend star(afters) [after 2000 star_proc $sec]


}

proc star_proc {{sec 5}} {
    #         x     y
    global star ob

    if {!$ob(running)} {
        return
    }
    incr star(trips)
    # puts "round trips: $star(trips)"
    set star(sec) $sec
    # after msec
    set star(atime) [expr {int($sec * 1000)}]
    set star(atime2) [expr {int(2 * $sec * 1000)}]
    # samples Hz

    set star(dir) [lindex $star(dirs) $star(i)]

    # stime is secs * sample time in Hz
    # set star(stime) [expr {int($sec * .6 * 200)}]
    set star(stime) [expr {int($sec * .9 * 200)}]
    # movebox 0 0 {0 $star(stime) 1} $star(c) $star($star(dir))
    # after $star(atime) [list movebox 0 0 {0 $star(stime) 1} $star($star(dir)) $star(c)]
    # after $star(atime2) star_proc

    movebox 0 0 {0 $star(stime) 1} $star(c) $star($star(dir))
    # puts "<after $star(atime) <movebox 0 0 <0 $star(stime) 1> $star($star(dir))
$star(c)>"

    set star(afters) [after $star(atime) {movebox 0 0 {0 $star(stime) 1}
$star($star(dir)) $star(c)}]

    lappend star(afters) [after $star(atime2) star_proc $star(sec)]

    set star(i) [expr {($star(i) + 1) % 2}]
}

proc star_stop {} {
    global star ob

    if {![info exists star(afters)]} {
        return
    }
    foreach i $star(afters) {
        after cancel $i
```

```
    }
}

# calculate Euclidean distance.  (Why not Pythagorean?)
proc edist {x1 y1 x2 y2} {
        expr {hypot($x1 - $x2, $y1 - $y2)}
}

# send arm from current position to center, at constant speed
proc center_arm {{cx 0.0} {cy 0.0}} {
        set x [getptr x]
        set y [getptr y]

        set dist [edist $x $y $cx $cy]
        set ticks [expr {int($dist * 40.)}]
        movebox 0 0 {0 $ticks 1} {$x $y 0.0 0.0} {$cx $cy 0.0 0.0}
}

# send arm from current position to center, taking two seconds.
proc center_arm_2s {{cx 0.0} {cy 0.0}} {
        set x [getptr x]
        set y [getptr y]

        set ticks [expr {2 * [rshm Hz]}]
        set dist [edist $x $y $cx $cy]
        movebox 0 0 {0 $ticks 1} {$x $y 0.0 0.0} {$cx $cy 0.0 0.0}
}

set ob(kd) .05

proc kick {{dir down}} {
        global ob
        set x [getptr x]
        set y [getptr y]
        set x2 $x
        set y2 $y

        switch $dir {

        west -
        left    {
                set x2 [expr {$x - $ob(kd)}]
        }

        east -
        right   {
                set x2 [expr {$x + $ob(kd)}]
        }

        south -
        down    {
                set y2 [expr {$y - $ob(kd)}]
        }

        north -
        up      {
                set y2 [expr {$y + $ob(kd)}]
        }

        default  { return }

        }

        movebox 0 0 {0 10 1} {$x $y 0.0 0.0} {$x2 $y2 0.0 0.0}
        after 50 {movebox 0 0 {0 0 0} {0.0 0.0 0.0} {0.0 0.0 0.0 0.0}}
}

# bias the ft, just one iteration, but the ft jitters anyway.
proc ft_bias {} {
        for {set i 0} {$i < 6} {incr i} {
                wshm ft_bias [rshm ft_raw $i] $i
```

```
	}
}

# shortcuts for start/stop

proc start_rtl {} {
	start_lkm
	start_shm
	start_loop
	after 100
}

proc stop_rtl {} {
	stop_loop
	stop_shm
	stop_lkm
}

# if there is no arm and you want to use the mouse as a pointer

proc no_arm {} {
	global ob
	# pixel offset between 0 and center, same here for x/y
	# this is a kludge, but close enough.

	set ob(ptroffset) 0

	# make all these do nothing
	proc start_lkm {} {}
	proc start_shm {} {}
	proc start_loop {} {}
	proc stop_lkm {} {}
	proc stop_shm {} {}
	proc stop_loop {} {}
	proc rshm {where {i 0}} {}
	proc wshm {where what {i 0}} {}

	proc start_log {logfile {num 3} {uheaderfile none}} {
		# make sure the dir is there
		file mkdir [file dirname $logfile]

		# write log header
		logheader $logfile $num
	}

	proc stop_log {} {}

	# make getptr read the mouse, and hack it into a screen positon.

	proc getptr {p} {
		global ob
		set w $ob(bigcan)
		set val [expr {[winfo pointer$p $w] - [winfo root$p $w]}]
		# flip y
		if {$p == "x"} {
			set val [expr {$val - $ob(half,x)}]
		} else {
			set val [expr {$val - $ob(half,y)}]
			set val [expr {-$val}]
		}
		# scale world to screen
		expr {$val / $ob(scale)}
	}

}

# write a log file header.
# pad with commented dots to 4096 bytes of ascii stuff
# (or truncate)
# make sure this is ascii, multi-byte chars will be messy here.
```

```
proc logheader {filename ncols {headerfile ""}} {
       global ob
       # puts "exec $ob(crobhome)/loghead $ncols $filename"
       exec $ob(crobhome)/loghead $filename $ncols
}


# exec with event loop, for tk.
# for long running progs, like go/stop
# http://mini.net/tcl/3039

# not using it right now, because it doesn't seem to handle
# error return from exec'd program correctly

proc tk_exec_fileevent {id} {
    global tkex

    if {[eof $tkex(pipe,$id)]} {
        fileevent $tkex(pipe,$id) readable ""
        set tkex(cond,$id) 1
        return
    }

    append tkex(data,$id) [read $tkex(pipe,$id) 1024]
}

proc tk_exec {args} {
    global tkex
    global tcl_platform
    global env

    if {![info exists tkex(id)]} {
        set tkex(id) 0
    } {
        incr tkex(id)
    }

    set id $tkex(id)

    set keepnewline 0

    for {set i 0} {$i < [llength $args]} {incr i} {
        set arg [lindex $args $i]
        switch -glob -- $arg {
        -keepnewline {
                set keepnewline 1
            }
        -- {
                incr i
                break
            }
        -* {
                error "unknown option: $arg"
            }
        ?* {
                # the glob should be on *, but the wiki reformats
                # that as a bullet
                break
            }
        }
    }

    if {$i > 0} {
        set args [lrange $args $i end]
    }

    if {$tcl_platform(platform) == "windows" && [info exists env(COMSPEC)]} {
        set args [linsert $args 0 $env(COMSPEC) "/c"]
    }

    set pipe [open "|$args" r]
```

```
    set tkex(pipe,$id) $pipe
    set tkex(data,$id) ""
    set tkex(cond,$id) 0

    fconfigure $pipe -blocking 0
    fileevent $pipe readable "tk_exec_fileevent $id"

    vwait tkex(cond,$id)

    if {$keepnewline} {
        set data $tkex(data,$id)
    } {
        set data [string trimright $tkex(data,$id) \n]
    }

    unset tkex(pipe,$id)
    unset tkex(data,$id)
    unset tkex(cond,$id)

    if {[catch {close $pipe} err]} {
        error "pipe error: $err"
    }

    return $data
}

# grasp sensor

# the quiet value of the sensor changes when it warms up.
# this squeeze code asks for an initial grasp voltage, which may vary.

proc start_grasp {{w .}} {
        global ob

        set ob(grasp_running) "true"
        set ob(grasp_state) "released"
        set gv [rshm adcvolts 8]

        # puts "calibrating grasp voltage $gv"

        # up is release, down is squeeze, akin to a mouse click.
        # up must be less than down, of course.
        set ob(grasp_up_thresh) [expr $gv + .2]
        set ob(grasp_down_thresh) [expr $gv + .3]
}

proc stop_grasp {} {
        global ob

        set ob(grasp_running) "false"
}

# game calls this proc from its main loop
# it generates
# <<GraspSqueeze>> when squeezed
# <<GraspRelease>> when released
# <<GraspMotion>> every sample in between

proc grasp_iter {{w .}} {
    global ob

    set ob(grasp_volts) [rshm adcvolts 8]
    if {$ob(grasp_state) == "released"} {
        if {$ob(grasp_volts) > $ob(grasp_down_thresh)} {
            set ob(grasp_state) "squeezed"
            event generate $w <<GraspSqueeze>> -x $ob(screen,x) -y $ob(screen,y)
        }
    } else {
        if {$ob(grasp_state) == "squeezed"} {
            event generate $w <<GraspMotion>> -x $ob(screen,x) -y $ob(screen,y)
```

```
            if {$ob(grasp_volts) < $ob(grasp_up_thresh)} {
                set ob(grasp_state) "released"
                event generate $w <<GraspRelease>>
            }
        }
    }
}
```

Vex source code

```
# tcl i/o with shm (user mode shared memory buffer) program
# sourced by other tcl scripts

# InMotion2 robot system software for RTLinux

# Copyright 2003-2004 Interactive Motion Technologies, Inc.
# Cambridge, MA, USA
# http://www.interactive-motion.com
# All rights reserved

if {[info exists env(CROB_HOME)]} {
    set ob(crobhome) $env(CROB_HOME)
} else {
    set ob(crobhome) /home/imt/crob
}

proc every {ms body {id ::after_id}} {
    eval $body
    set $id [after $ms [info level 0]]
}

proc procname {} {return [lindex [info level -1] 0]}

proc cancel_afters {} {
       foreach id [after info] {after cancel $id}
       foreach id [after info] {after cancel $id}
}

# reap zombie processes after "exec &" commands exit
# see: http://mini.net/tcl/1039
proc reap_zombies {} {
    catch {exec ""}
}

# flip y coordinate

proc y_up args {
    set ret ""
    if {[llength $args]==1} {set args [lindex $args 0]}
    foreach {x y} $args {lappend ret $x [expr {-$y}]}
    return $ret
}

# given a center position and radius, like 100 100 10,
# centxy returns x1 y1 x2 y2, like 90 90 110 110.

proc centxy {x y rad} {
    set x1 [expr {$x - $rad}]
    set y1 [expr {$y - $rad}]
    set x2 [expr {$x + $rad}]
    set y2 [expr {$y + $rad}]
    list $x1 $y1 $x2 $y2
}

proc centertag {w tag} {
       foreach {x1 y1 x2 y2} [$w coords $tag] break
       set x [expr {$x1 + $x2 / 2.}]
       set y [expr {$y1 + $y2 / 2.}]
```

```
        list $x $y
}



# lkm loaded?

proc is_lkm_loaded {} {
    file exists /proc/pwrdaq
}


# load lkms

proc start_lkm {} {
    global ob
    if {![file executable $ob(crobhome)/go]} {
        error "start_lkm: could not run go"
    }

    # set status [catch {tk_exec sh $ob(crobhome)/go} result]
    set status [catch {exec sh $ob(crobhome)/go} result]
    if { $status != 0 } {
        stop_lkm
        error "start_lkm: could not start kernel module robot.o\n\
        result string:\n<<\n$result\n>>\n"
    }
}

# unload lkms

proc stop_lkm {} {
    global ob
    if {![file executable $ob(crobhome)/stop]} {
        puts "stop_lkm: could not run stop"
        exit 1
    }

    set status [catch {tk_exec sh $ob(crobhome)/stop} result]
    set status [catch {exec sh $ob(crobhome)/stop} result]
    if { $status != 0 } {
        puts "stop_lkm: could not stop kernel module robot.o"
        puts "result string:\n<<\n$result\n>>\n"
    }
}

# start shm - the shared memory buffer C program

proc start_shm {} {
    global ob
    if {! [file exists $ob(crobhome)/shm] } {
        puts stderr "start_shm: can't find shared memory program $ob(crobhome)/shm"
        exit 1
    }
    set ob(shm) [open "|$ob(crobhome)/shm" r+]
    fconfigure $ob(shm) -buffering line
    after 100
    set check [rshm last_shm_val]
    if {$check != 12345678} {
        puts "start_shm: bad shm check value."
        puts "make sure all software has been compiled with latest cmdlist.tcl"
        exit 1
    }
}

proc stop_shm {} {
    global ob
    if {![info exists ob(shm)]} {
        return
    }
    set ob(loaded) 0
```

```
    puts $ob(shm) "q"
    close $ob(shm)
    unset ob(shm)
}

proc start_log {logfile {num 3} {uheaderfile ""}} {
    global ob

    # puts "start_log $logfile $num"
    wshm nlog $num

    # make sure the dir is there
    file mkdir [file dirname $logfile]

    # write log header
    logheader $logfile $num $uheaderfile

    set ob(savedatpid) [exec cat < /dev/rtf1 >> $logfile &]
}

proc stop_log {} {
    global ob

    # puts "stop_log"
    wshm nlog 0
    if [info exists ob(savedatpid)] {
        exec kill $ob(savedatpid)
        unset ob(savedatpid)
    }
}

proc xyplot_log {filename} {
        global ob
        exec [file join $ob(crobhome) xygp] $filename &
}

proc plot_log {filename {plotcmd {}}} {
        global ob
        exec [file join $ob(crobhome) gp] $filename $plotcmd &
}

# if the shm process gets killed from outside, the puts here will fail.
# this will set shm_puts_exit_in_progress, and cleanup should happen.
# don't call stop_shm or stop_loop, since these just do more i/o to the
# now broken shm channel.

proc shm_puts str {
    global ob
    if [info exists ob(shm_puts_exit_in_progress)] {
        puts stderr "shm_puts error, exit in progress..."
    }
    if [catch {puts $ob(shm) $str}] {
        set ob(shm_puts_exit_in_progress) 1
        puts stderr "shm_puts error, stopping lkm."
        stop_lkm
        exit 1
    }
}

# i is array index in both.

# wshm writes systcl vars
# like /sbin/sysctl -w where=what

proc wshm {where what {i 0}} {
    global ob
    if {![info exists ob(shm)]} {
        return
    }
    if [info exists ob(shm_puts_exit_in_progress)] {
        return
```

```
    }
    shm_puts "s $where $i $what"

    gets $ob(shm) istr
    set what [lindex $istr 0]
    if {[string equal $what "?"]} {
        puts stderr $istr
    }
}

# rshm reads systcl vars
# like /sbin/sysctl where

proc rshm {where {i 0}} {
    global ob
    set what "???"
    if {![info exists ob(shm)]} {
        return "0.0"
    }
    if [info exists ob(shm_puts_exit_in_progress)] {
        return "0.0"
    }
    shm_puts "g $where $i"

    gets $ob(shm) istr
    set what [lindex $istr 0]
    if {[string equal $what "?"]} {
        puts stderr $istr
        return "0.0"
    }
    set what [lindex $istr 3]
    # set what [expr int($what * 1000)]
    return $what
}

proc start_loop {} {
    wshm paused 0
}

proc stop_loop {} {
    wshm paused 1
}

proc mouse_getptr {p} {
    expr {[winfo pointer$p $::tachw]}
}

proc mouse_getvel {p} {
    # delta motion since last tick
    set d($i) [expr {$p($i) - $::lastp($i)}]
    # velocity in pixels/sec
    set v($i) [expr {$d($i) * $::hz}]

    # smooth $v
    set v($i) [iirsmooth $v(i)]
}

# robot ptr/vel

proc getptr {p} {
    rshm $p
}

proc soft_getvel {p} {
    rshm soft_${p}vel
}

proc fsoft_getvel {p} {
    rshm fsoft_${p}vel
}
```

```
proc tach_getvel {p} {
    rshm tach_${p}vel
}

proc getvel {p} {
    rshm ${p}vel
}

proc gettrq {p} {
    rshm ${p}_torque
}

proc getvolts {p} {
    rshm ${p}_volts
}

proc getfrc {p} {
    rshm ${p}_force
}

proc getftfrc {p} {
    rshm ft_${p}dev
}

proc getwftfrc {p} {
    rshm ft_${p}world
}

proc f3k {n} {
    format %.3f [expr {1000.0 * $n}]
}

proc f3 {n} {
    format %.3f $n
}

# checkerror should be called from inside a user-mode program event loop,
# to make sure that the event loop isn't generating too many errors or
# warnings.  a few such errors are expected in normal operation, but if
# something goes wrong, they should happen on every sample and they will
# exceed the max (10, for example) quickly.

proc checkerror {{max 10}} {
    global ob
    set ob(errormax) max
    set nerrors [rshm nerrors]
    if {$nerrors > $ob(errormax)} {
        set i [rshm errorindex]
        set ei $i
        set error0i [rshm errori $i]
        set error0code [rshm errorcode $i]
        incr i -1
        set error1i [rshm errori $i]
        set error1code [rshm errorcode $i]
        incr i -1
        set error2i [rshm errori $i]
        set error2code [rshm errorcode $i]

        set estring "InMotion2 System, Pausing control loop.\n\
nerrors = $nerrors,\n\
last ($ei): iteration = $error0i, code = $error0code.\n\
last-1: iteration = $error1i, code = $error1code.\n\
last-2: iteration = $error2i, code = $error2code.\n\
You may run shm to analyze system state, then run ./stop ."
        stop_loop
        error $estring
        # stop_lkm
        # after 2000
        # exit 1
    }
}
```

```
# movebox to move a box
#              i f  forlist    from                  to
# e.g.: movebox 0 0 {0 1000 1} {0.0 0.0 0.005 0.005} {0.15 0.15 0.005 0.005}

proc movebox {slot_id slot_fnid forlist box0 box1} {
# puts "movebox $slot_id $slot_fnid $forlist $box0 $box1"

    # the uplevel/subst allows users to put $vars in the lists.
    set forlist [uplevel 1 [list subst -nocommands $forlist]]
    set box0 [uplevel 1 [list subst -nocommands $box0]]
    set box1 [uplevel 1 [list subst -nocommands $box1]]

    foreach {slot_i slot_term slot_incr} $forlist break
    foreach {slot_b0_x slot_b0_y slot_b0_w slot_b0_h} $box0 break
    foreach {slot_b1_x slot_b1_y slot_b1_w slot_b1_h} $box1 break

    foreach i {
        slot_id slot_fnid
        slot_i slot_term slot_incr
        slot_b0_x slot_b0_y slot_b0_w slot_b0_h
        slot_b1_x slot_b1_y slot_b1_w slot_b1_h
    } {wshm $i [set $i]}

    wshm slot_running 1
    wshm slot_go 1
    wshm slot_max 1

}

# stop a slot currently in progress

proc stop_movebox {slot_id} {
    wshm slot_max 0
    foreach i {
        slot_id slot_fnid
        slot_i slot_term slot_incr
        slot_b0_x slot_b0_y slot_b0_w slot_b0_h
        slot_b1_x slot_b1_y slot_b1_w slot_b1_h
        slot_running
    } {wshm $i 0}
    wshm slot_go 1
}

proc star_once {} {
    global star

    if {[info exists star(i)]} {
        return
    }

    set star(i) 0

    set star(hw) 0.005
    set star(c)  [list  0.0  0.0 $star(hw) $star(hw)]
    #set star(s)  [list  0.0 -0.14 $star(hw) $star(hw)]
    #set star(n)  [list  0.0  0.14 $star(hw) $star(hw)]
    set star(w)  [list -0.14  0.0 $star(hw) $star(hw)]
    set star(e)  [list  0.14  0.0 $star(hw) $star(hw)]
    #set star(nw) [list -0.14  0.14 $star(hw) $star(hw)]
    #set star(ne) [list  0.14  0.14 $star(hw) $star(hw)]
    #set star(sw) [list -0.14 -0.14 $star(hw) $star(hw)]
    #set star(se) [list  0.14 -0.14 $star(hw) $star(hw)]
   # set star(dirs) {n ne e se s sw w nw}
   set star(dirs) {e w}
}

star_once

proc start_star {{sec 5}} {
        global star
```

```
        #while (1) {
        #        set x [getptr x]
        #       set y [getptr y]
        #       puts "xvalue is: $x"

        #              if {$x > .14} {
        #              puts " iam here"

        #       after 50 {movebox 0 0 {0 0 0} {x y 0.0 0.0} {.14 0.0 0.0 0.0}}
        #              puts " now i ma herjelkjreklj"
        #              } elseif {$x < -.14} {
        #              puts " i am after -.14"
        #       after 50 {movebox 0 0 {0 0 0} {x y 0.0 0.0} {-.14 0.0 0.0 0.0}}
#puts " i am aftere hrlkejrlkejklejrlek"
        #       }
        #}


#theirs

        set star(trips) 0

        # takes 1 second

        center_arm

        # after 2 seconds

        lappend star(afters) [after 2000 star_proc $sec]


}

proc star_proc {{sec 5}} {
    #         x     y
    global star ob

    if {!$ob(running)} {
        return
    }
    incr star(trips)
    # puts "round trips: $star(trips)"
    set star(sec) $sec
    # after msec
    set star(atime) [expr {int($sec * 1000)}]
    set star(atime2) [expr {int(2 * $sec * 1000)}]
    # samples Hz

    set star(dir) [lindex $star(dirs) $star(i)]

    # stime is secs * sample time in Hz
    # set star(stime) [expr {int($sec * .6 * 200)}]
    set star(stime) [expr {int($sec * .9 * 200)}]
    # movebox 0 0 {0 $star(stime) 1} $star(c) $star($star(dir))
    # after $star(atime) [list movebox 0 0 {0 $star(stime) 1} $star($star(dir)) $star(c)]
    # after $star(atime2) star_proc

    movebox 0 0 {0 $star(stime) 1} $star(c) $star($star(dir))
    # puts "<after $star(atime) <movebox 0 0 <0 $star(stime) 1> $star($star(dir))
$star(c)>"

    set star(afters) [after $star(atime) {movebox 0 0 {0 $star(stime) 1}
$star($star(dir)) $star(c)}]

    lappend star(afters) [after $star(atime2) star_proc $star(sec)]

    set star(i) [expr {($star(i) + 1) % 2}]
}

proc star_stop {} {
```

```
    global star ob

    if {![info exists star(afters)]} {
        return
    }
    foreach i $star(afters) {
        after cancel $i
    }
}

# calculate Euclidean distance.  (Why not Pythagorean?)
proc edist {x1 y1 x2 y2} {
        expr {hypot($x1 - $x2, $y1 - $y2)}
}

# send arm from current position to center, at constant speed
proc center_arm {{cx 0.0} {cy 0.0}} {
        set x [getptr x]
        set y [getptr y]

        set dist [edist $x $y $cx $cy]
        set ticks [expr {int($dist * 40.)}]
        movebox 0 0 {0 $ticks 1} {$x $y 0.0 0.0} {$cx $cy 0.0 0.0}
}

# send arm from current position to center, taking two seconds.
proc center_arm_2s {{cx 0.0} {cy 0.0}} {
        set x [getptr x]
        set y [getptr y]

        set ticks [expr {2 * [rshm Hz]}]
        set dist [edist $x $y $cx $cy]
        movebox 0 0 {0 $ticks 1} {$x $y 0.0 0.0} {$cx $cy 0.0 0.0}
}

set ob(kd) .05

proc kick {{dir down}} {
        global ob
        set x [getptr x]
        set y [getptr y]
        set x2 $x
        set y2 $y

        switch $dir {

        west -
        left    {
                set x2 [expr {$x - $ob(kd)}]
        }

        east -
        right   {
                set x2 [expr {$x + $ob(kd)}]
        }

        south -
        down    {
                set y2 [expr {$y - $ob(kd)}]
        }

        north -
        up      {
                set y2 [expr {$y + $ob(kd)}]
        }

        default  { return }

        }

        movebox 0 0 {0 10 1} {$x $y 0.0 0.0} {$x2 $y2 0.0 0.0}
```

```
	after 50 {movebox 0 0 {0 0 0} {0.0 0.0 0.0} {0.0 0.0 0.0 0.0}}
}

# bias the ft, just one iteration, but the ft jitters anyway.
proc ft_bias {} {
	for {set i 0} {$i < 6} {incr i} {
		wshm ft_bias [rshm ft_raw $i] $i
	}
}

# shortcuts for start/stop

proc start_rtl {} {
	start_lkm
	start_shm
	start_loop
	after 100
}

proc stop_rtl {} {
	stop_loop
	stop_shm
	stop_lkm
}

# if there is no arm and you want to use the mouse as a pointer

proc no_arm {} {
	global ob
	# pixel offset between 0 and center, same here for x/y
	# this is a kludge, but close enough.

	set ob(ptroffset) 0

	# make all these do nothing
	proc start_lkm {} {}
	proc start_shm {} {}
	proc start_loop {} {}
	proc stop_lkm {} {}
	proc stop_shm {} {}
	proc stop_loop {} {}
	proc rshm {where {i 0}} {}
	proc wshm {where what {i 0}} {}

	proc start_log {logfile {num 3} {uheaderfile none}} {
		# make sure the dir is there
		file mkdir [file dirname $logfile]

		# write log header
		logheader $logfile $num
	}

	proc stop_log {} {}

	# make getptr read the mouse, and hack it into a screen positon.

	proc getptr {p} {
		global ob
		set w $ob(bigcan)
		set val [expr {[winfo pointer$p $w] - [winfo root$p $w]}]
		# flip y
		if {$p == "x"} {
			set val [expr {$val - $ob(half,x)}]
		} else {
			set val [expr {$val - $ob(half,y)}]
			set val [expr {-$val}]
		}
		# scale world to screen
		expr {$val / $ob(scale)}
	}
```

```
}

# write a log file header.
# pad with commented dots to 4096 bytes of ascii stuff
# (or truncate)
# make sure this is ascii, multi-byte chars will be messy here.

proc logheader {filename ncols {headerfile ""}} {
        global ob
        # puts "exec $ob(crobhome)/loghead $ncols $filename"
        exec $ob(crobhome)/loghead $filename $ncols
}


# exec with event loop, for tk.
# for long running progs, like go/stop
# http://mini.net/tcl/3039

# not using it right now, because it doesn't seem to handle
# error return from exec'd program correctly

proc tk_exec_fileevent {id} {
    global tkex

    if {[eof $tkex(pipe,$id)]} {
        fileevent $tkex(pipe,$id) readable ""
        set tkex(cond,$id) 1
        return
    }

    append tkex(data,$id) [read $tkex(pipe,$id) 1024]
}

proc tk_exec {args} {
    global tkex
    global tcl_platform
    global env

    if {![info exists tkex(id)]} {
        set tkex(id) 0
    } {
        incr tkex(id)
    }

    set id $tkex(id)

    set keepnewline 0

    for {set i 0} {$i < [llength $args]} {incr i} {
        set arg [lindex $args $i]
        switch -glob -- $arg {
        -keepnewline {
                set keepnewline 1
            }
        -- {
                incr i
                break
            }
        -* {
                error "unknown option: $arg"
            }
        ?* {
                # the glob should be on *, but the wiki reformats
                # that as a bullet
                break
            }
        }
    }

    if {$i > 0} {
        set args [lrange $args $i end]
```

```
    }

    if {$tcl_platform(platform) == "windows" && [info exists env(COMSPEC)]} {
        set args [linsert $args 0 $env(COMSPEC) "/c"]
    }

    set pipe [open "|$args" r]

    set tkex(pipe,$id) $pipe
    set tkex(data,$id) ""
    set tkex(cond,$id) 0

    fconfigure $pipe -blocking 0
    fileevent $pipe readable "tk_exec_fileevent $id"

    vwait tkex(cond,$id)

    if {$keepnewline} {
        set data $tkex(data,$id)
    } {
        set data [string trimright $tkex(data,$id) \n]
    }

    unset tkex(pipe,$id)
    unset tkex(data,$id)
    unset tkex(cond,$id)

    if {[catch {close $pipe} err]} {
        error "pipe error: $err"
    }

    return $data
}

# grasp sensor

# the quiet value of the sensor changes when it warms up.
# this squeeze code asks for an initial grasp voltage, which may vary.

proc start_grasp {{w .}} {
        global ob

        set ob(grasp_running) "true"
        set ob(grasp_state) "released"
        set gv [rshm adcvolts 8]

        # puts "calibrating grasp voltage $gv"

        # up is release, down is squeeze, akin to a mouse click.
        # up must be less than down, of course.
        set ob(grasp_up_thresh) [expr $gv + .2]
        set ob(grasp_down_thresh) [expr $gv + .3]
}

proc stop_grasp {} {
        global ob

        set ob(grasp_running) "false"
}

# game calls this proc from its main loop
# it generates
# <<GraspSqueeze>> when squeezed
# <<GraspRelease>> when released
# <<GraspMotion>> every sample in between

proc grasp_iter {{w .}} {
    global ob

    set ob(grasp_volts) [rshm adcvolts 8]
    if {$ob(grasp_state) == "released"} {
```

```
        if {$ob(grasp_volts) > $ob(grasp_down_thresh)} {
            set ob(grasp_state) "squeezed"
            event generate $w <<GraspSqueeze>> -x $ob(screen,x) -y $ob(screen,y)
        }
    } else {
        if {$ob(grasp_state) == "squeezed"} {
            event generate $w <<GraspMotion>> -x $ob(screen,x) -y $ob(screen,y)
            if {$ob(grasp_volts) < $ob(grasp_up_thresh)} {
                set ob(grasp_state) "released"
                event generate $w <<GraspRelease>>
            }
        }
    }
}
```

References

[1] M.H. Beers and R. Berkow. Chapter 29. Rehabilitation For Specific Problems. In *The Merch Manual of Geriatrics* [online]. Available from: http://www.merck.com/mrkshared/mm_geriatrics/sec3/ch29.jsp

[2] C.G. Burgar and P.S. Lum. *Robot-Assisted Upper Limb Neuro-Rehabilitation* [online]. VA Rehab R&D, 2000. Available from: http://guide.stanford.edu/Projects/2kprojects/stroke05.html

[3] J.H. Carr, R.B. Shepherd, A Motor Relearning Programme For Stroke, Rockville, Maryland, London: Aspen publication, 1983.

[4] M. Johnstone and C. Livingstone, Restoration of Motor Function in the Stroke patient, London Melbourne and New York: 1983.

[5] L.E. Kahn, M. Averbuch, W.Z. Rymer and D.J. Reinkensmeyer. *Comparison of Robot-Assisted Reaching to Free Reaching in Promoting Recovery From Chronic Stroke* [online]. Available from: http://www.eng.uci.edu/~dreinken/publications/us03.pdf

[6] A.P. Olsson, C.R. Carignan and J. Tang. Cooperative control of virtual objects using haptic teleportation over the internet [online]. Available from: http://www.icdvrat.reading.ac.uk/2004/papers/S05_N2_Olsson_ICDVRAT2004.pdf

[7] R. Rabkin. *Robot helps strike Patients* [online]. Columbia News Service, April 2002. Available from: http://www.jrn.columbia.edu/studentwork/cns/2002-04-30/569.asp.

[8] J.G. Smits and E.C smits-Boone, Hand Recovery After Stroke- Exercises and Results Measurements, Boston Oxford Auckland Johannesburg Melbourne New Delhi: Butterworth- Heinemann, 2000.

[9] Arm exercises after stroke improve function. In *Carolinas HealthCare System* [online]. Your Health- Health information, March 2004. Available from: http://healthinfo.carolinas.org/HealthNews/Reuters/20040304elin017.htm

[10] Arm Exercises for Stroke Patients. In *TSAO foundation For Successful Ageing* [online]. Available from: http://www.tsaofoundation.org/healthtips/stroke04.html

[11] Recovering Arm and Hand function. *Regaining Arm Movement* [online]. Available from:http://www.strokesurvivors.ca/Regaining%20arm.htm

[12] Robotic Physical Therapy Improves Movement Long After Stroke. In *ScienceDaily LLC* [online]. February, 2002. Available from: http://www.sciencedaily.com/releases/2002/02/020211080854.htm

[13] Additional documentation can be found at the following location: \notes\index.html- InMotion2RTLinuxSoftware System Design overview

[14] An Overview of Basic Measurement Theories http://www.measurementexperts.org/learn/theories/theories.asp

[15] Expert System http://www.aaai.org/AITopics/html/expert.html

[16] Functional Status Measures http://www.vard.org/rorc/functional_measures.htm

[17] Medical Export Systems http://www.computer.privateweb.at/judith/

[18] Post-Stroke Rehabilitation Fact Sheet http://www.ninds.nih.gov/health_and_medical/pubs/poststrokerehab.htm

[19] Tcl/Tk 8.5 Manual. In *Tcl Developer Xchange* [online]. Available from: http://www.tcl.tk/man/tcl8.5/