UNIVERSITY OF CALIFORNIA AT BERKELEY
Mechanical Engineering

# Three Axis Milling Machine

ME230 Final Project
Fall 1998

Yitao Duan
Pete Retondo
Robert Hillaire

Dec 19, 1998

## Table of Contents

**List of Figures**

## Abstract

This projects hardware target is a three axis closed frame milling machine with a high-speed spindle. All of the components are off the shelf so that repair and maintenance is inexpensive and simple. The milling machine has a sturdy all aluminum frame. The aluminum frame is lightweight, and easy to manufacture using CNC machine tools. The revolutionary closed frame design increases rigidity and decreases the effect of thermal errors and Abbe' offset errors. Providing four posts for the Z-axis to ride on closes the frame. The working volume of this machine is 3"x3"x3".

The control software provides the best performance possible in the given time. The typical accuracy of the machine is 0.050 inches with speeds up to 15 inches per minute. The major features of the control software are:

1. It is able to coordinate all three axes so that the cutter will move smoothly at a velocity (also called feed in machine tool industry) as constant as possible;
2. It has the capability to cut a part with arbitrary shape and certain complexity;
3. It has a user interface that takes inputs and displays enough information.

The user interface anticipates the needs of the system's users, and to make the necessary functions accessible and understandable. From this point of view, two kinds of use were accommodated. The first set of operations has to do with basic machine control of the sort an operator would be concerned with. The second was seen as addressing the needs of designers of parts or objects. The ability of designers to have a software path of access to the operation of fabrication equipment is the essence of intelligent, integrated manufacturing.

## Introduction

Computer Controlled Milling Machines are prevalent in the manufacturing community. The one created for this project will not revolutionize the industry. Rather, it instanciates another example of how to create a machine tool. However, it does bring to bear one extension in the realm of manufacturing: a completely software driven control of a machine tool. In the manufacturing arena, this is what is known as a fully Open Architecture Controller. The User Interface also brings to bear an important issue in manufacturing. Creating a useful user interface which provides as rich as information as is desired by the operator is important. However, the real achievement is the GUI module that automatically recognized the shape of the object that the operator desires to cut and generating a tool path to make the machine tool trace this path.

## Body

### Hardware Design Details

The high-speed spindle is a Dremmel grinder with a three-foot flexible extension. The Dremmel runs from 5,000 rpm to 30,000 rpm. Several cutting bit designs are available for the Dremmel grinder.

This closed frame design produced some fundamental kinematic problems. The Z-axis is constrained vertically by the drive mechanism. The four post constrain the remaining five degrees of freedom. This lead to the fact that one of the posts has line contact (or two-point contact) while the other posts has single point contact. However, due to tolerance issues, single point contact is not assured for any of the post leading to an over-constrained kinematic device.

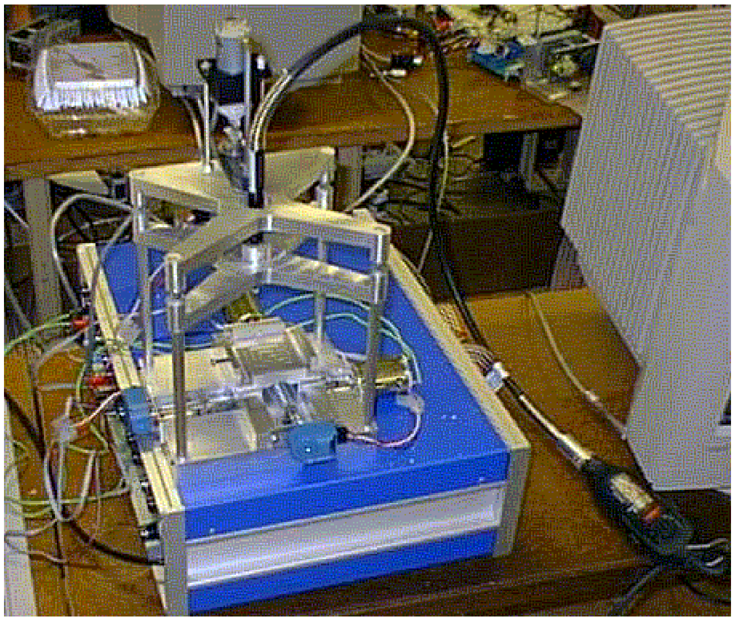


**Figure 1 Milling Machine**

The Z-axis, Figure 1, has an innovative "Centered off Center design" with six inches of travel. The motor unit is mounted off center from the Z-axis. The motor drives a lead-screw that is mounted near the center of the z-axis to provide a balanced thrust load. The lead screw is rigidly attached to the Z-axis carriage while a pulley connected to the motor rotates the drive nut. The Z-axis motor assembly is a 3V 4,800-rpm motor with a 50:1 double planetary gear reduction salvaged from an inexpensive cordless screwdriver. The motor mounting was machined with a three-axis CNC milling machine. The bearing mounted pulley shafts were manufactured on a manual flat bed lathe. The motor to pulley coupling is achieved with poly-vinyl tubing. A 400 count per revolution optical encoder is used to measure axis position.

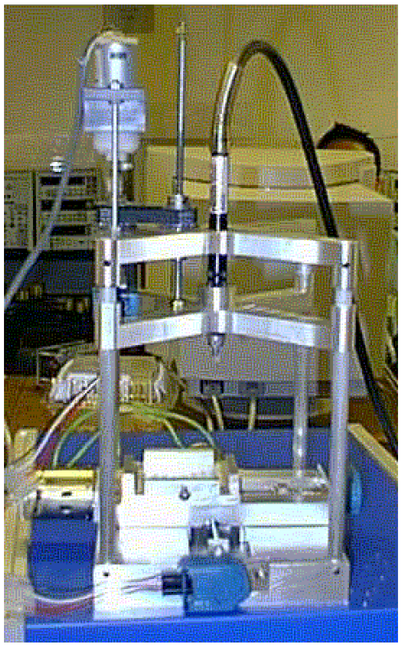


**Figure 2 Milling Machine, Side View**

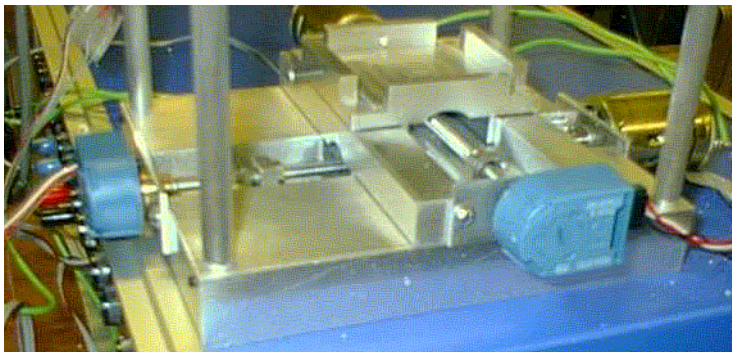


**Figure 3 Milling Machine, XY Table**

The XY table assembly, Figure 3, uses traditional Y-on-X Box-way guides for simplicity and rigidity. Machining all but two 0.050" runners along the edges of each slide relieved table slide contact. The lead-screw is a 10-32 threaded rod with 1/4" diameter bushings at either end to contain the rod radially. One jam-nut at either end of the threaded rob contain it axially. A 3oz-in 1800rpm motor drives each lead-screw through a poly-vinyl tube on one end, while a 800 count per revolution encode is attached to the other end of the lead-screw, also with poly-vinyl tubing. The resolution on the encoder translates to 0.00004 inches resolution on XY travel.

## Sample Parts

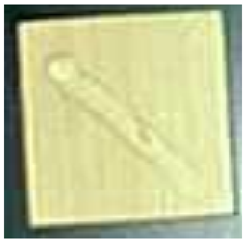


**Figure 4 Coordinated 3-Axis Slot**

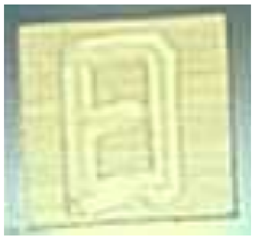


**Figure 5 Chinese Sun Symbol**

These two parts show typical machine tool performance. The first part, Figure 4, was made with a coordinated three-axis move. The slot is diagonal in the XY plane and in the XZ plane (or YZ plane). The second part, Figure 5, is a Chinese Sun symbol. The first part was made while Windows NT had three applications open. As a result, the servo performance was moderate to poor. At one end of the slot, the axis jitter resulted in a .050 inch gouge. However, this sun part was made while Windows NT was only running the control application. The servo performance was considerably better. No obvious gouging appears.

## Software Design

Based on the different functionality, the software can be divided in to two basic parts: user interface and control. Figure 6 illustrates this scheme.

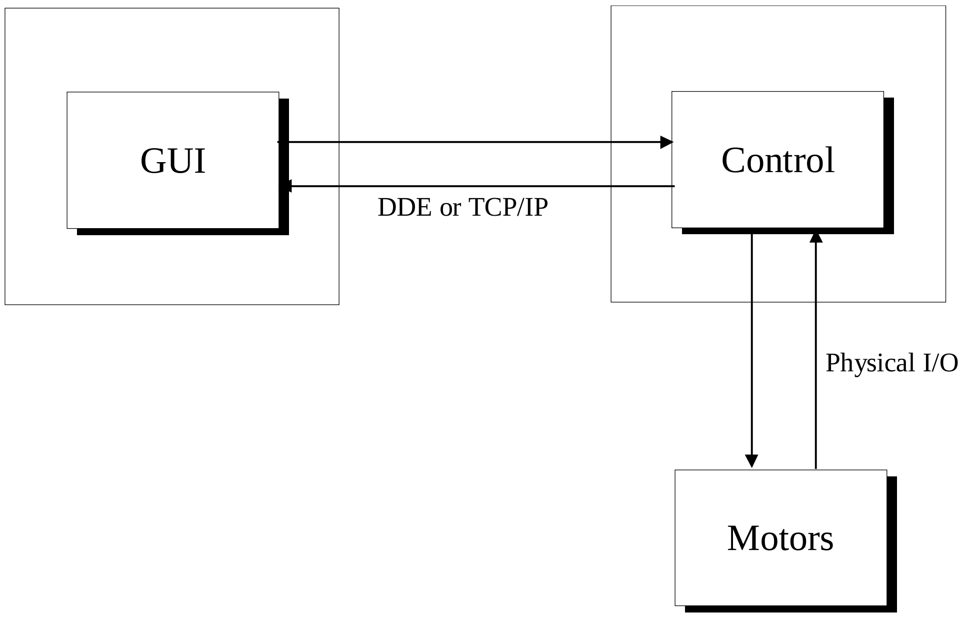


**Figure 6 Interconnect Scheme**

The two processes are connected either by Dynamic Data Exchange or TCP/IP network, depending on whether they are running on one computer or two. Best performance will be achieved if the control process runs on one computer under DOS or other real time operating system and the graphic user interface runs on another computer under Windows NT. This, however, is not implemented due to the limited time and the numerous problems we encountered while developing and implementing the control algorithm. At current stage, the two processes run on the same computer and communicate with each other via DDE and the performance is quite satisfactory.

## Control

Most of our time and energy is spent on this part trying to get the best performance. Several issues will be addressed in the following sections. Figure 7 shows the task structure implemented.

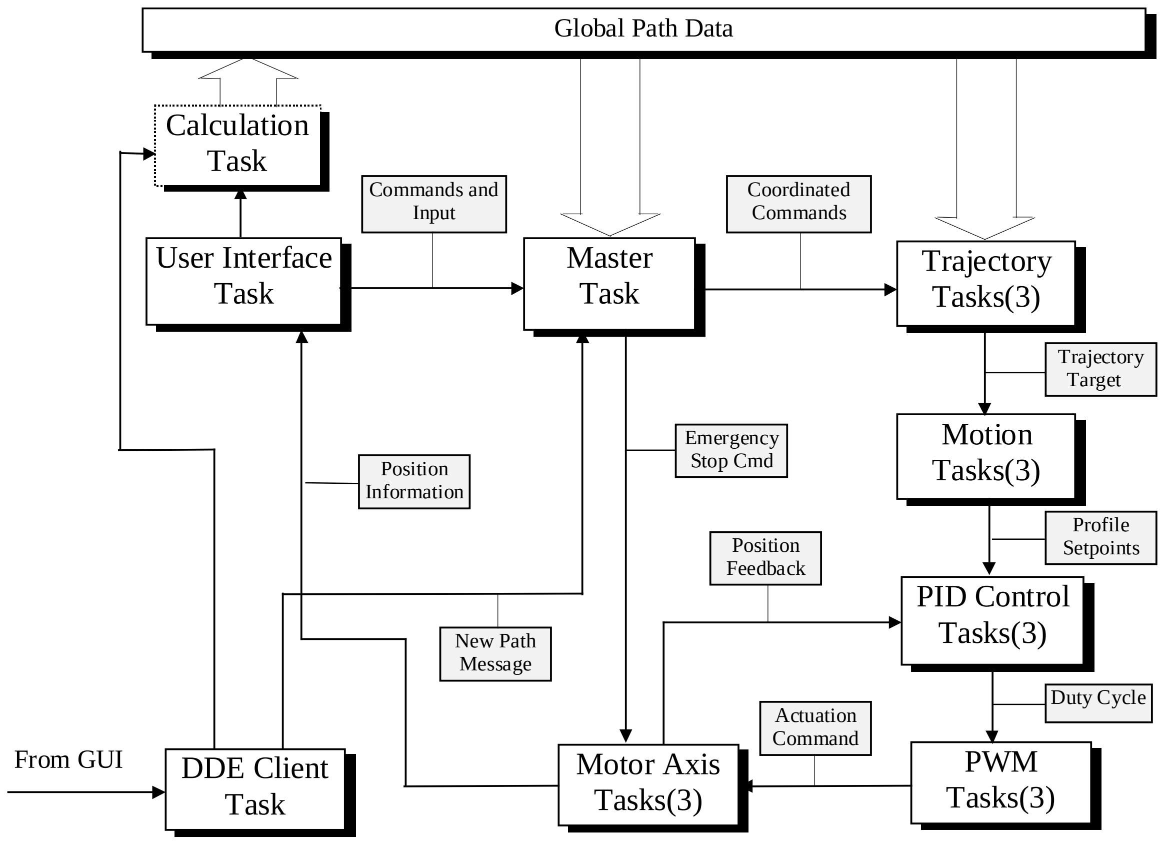


**Figure 7 Task Structure**

Note that Calculation Task is placed in a dashed box that indicates that this task is not in the execution loop. The responsibility of Calculation Task is to generate desired path date for the other tasks to carry out. It is different from the other tasks in that it is not in the execution list thus the task dispatcher does not scan it. What is more, it does not even override the BaseTask's virtual function Run(), which is supposed to be done by each derived classes. In fact, however, overriding Run function is only necessary for tasks that are meant to be scanned constantly, i.e. those that must be executed at a certain frequency. Here calculation needs to be done only once before each operation. Once it is done, keeping it in the execution loop is not only unnecessary, but also will increase the latency overhead. On the other hand, however, it is still necessary to keep calculation a task(i.e. it is implemented by a class derived from BaseTask.) so that it can access tasks' global data and also can take advantage the tasks' global data acquisition and message methods . Thus considered, in our design, calculation is a task but is not in the dispatching loop. It is invoked by UserInterface Task only when new path is assigned by the user. It then creates new path based on global data from UserInterface task which contain user specified path information. New path is assigned as a global data so all tasks can access. By keeping calculation off line, latency is minimized and performance improved.

## Tasks and Their Responsibility

User Interface: Interfaces with user, passes command and parameters to other tasks, and display feedback

information.

Master: Coordinates the movement of three axes. The major responsibility is to make sure that all

three axes go to the desired target together at the desired time.

Trajectory: Takes path information and sends targets to motion task to be realized into a point-to-

| | |
|---|---|
| | point move(to be described below). |
| Motion: | This task is modified from the motion task of past project to be able to take a new start point, which can be passed as a message or taken from a specified task, for each move. It can generate the desired profile with Trajectory task. |
| PID Control: | Generates actuation based on the setpoints and motor feedback. |
| PWM: | Generates PWM signals to drive encoder motors. |
| MotorAxis: | Actually controls the motor and sends back position feedback infomation. |
| Calculation: | Generates desired path date based on the user inputs |
| DDE Client: | This task emulates a DDE Client window and receives data from GUI which acts as a server. |

To be generic, the Trajectory takes a path object as parameter that tells it the next target. It also can be passed by message from master task. The generation of path object is the responsibility of the Calculation. UI should specify the type of path.

All these tasks are compliant with the task/state structure. The state transition logic structures of two most important tasks -- Master and Trajectory -- are given below. The new motion task, which is created to fit our particular need, will be discussed later.

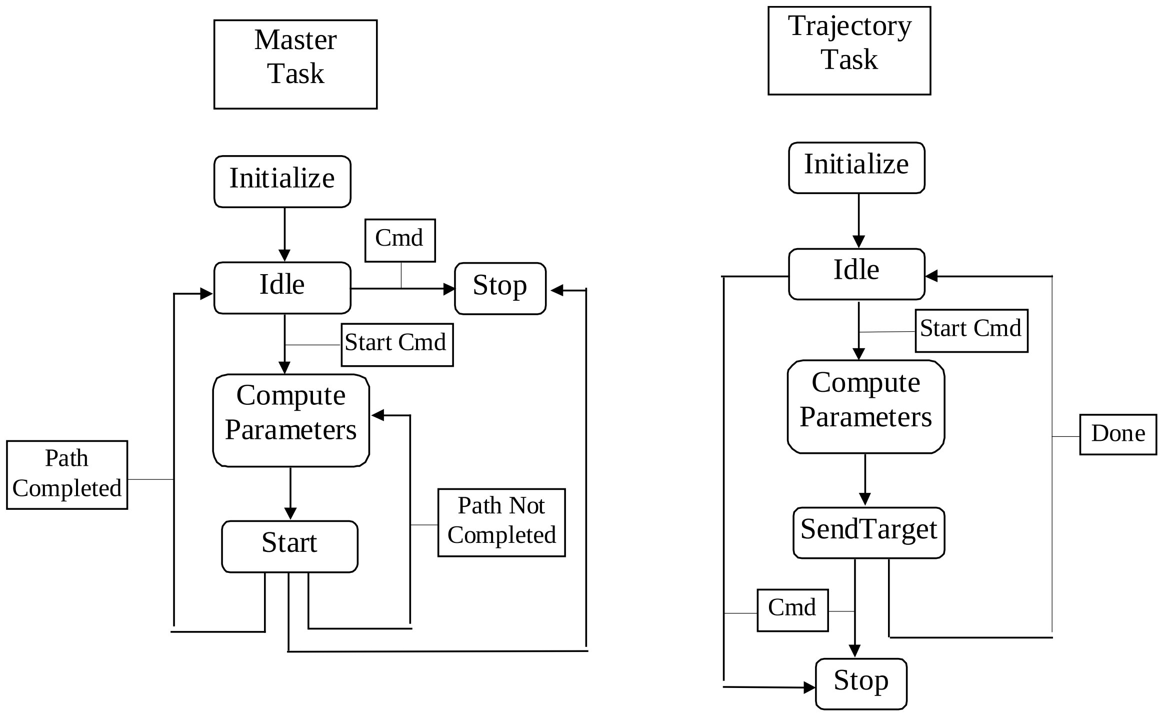


**Figure 8 Master-Trajectory Task**

## Motion Profile

To ensure smooth move and effective control, profiled move is a must. Some task similar to the Motion task used extensively in the previous projects is needed. The original motion task, however, has certain limitation and is not fit for this project. It can only generate such profile that is trapezoidal in velocity -- it consists of a period of constant acceleration, followed by a period of constant velocity, ending with a constant deceleration to the targeted position, as shown in Figure 9. This, however, is not what we want in that the move always stops at target position. What is desired in machine tool operation is that the cutter moves at a velocity as constant as possible. Stopping at each target will result in a bumping movement that is highly undesired in machine tool operation. A new motion profile-generating task must be used.

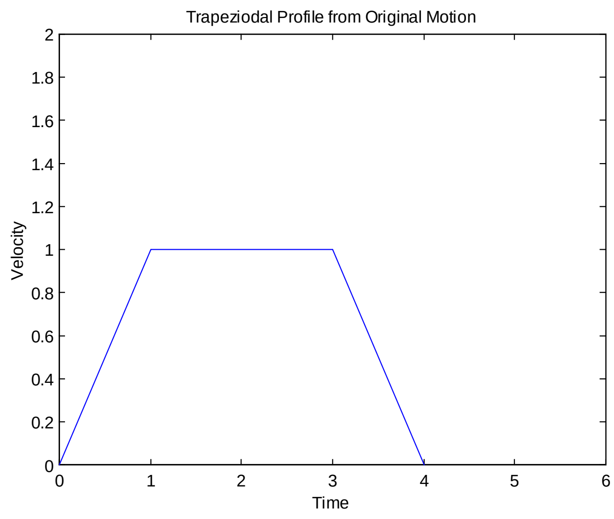


**Figure 9 Trapezoidal Motion Profile**

In designing such a motion task, one must bear in mind that although in an ideal situation the cutter moves at a constant velocity, each individual axis does not. Essentially the three axes on our milling machine represent three axes in a fixed rectangle coordinate system. To ensure the resultant overall speed is constant along a desired path, velocity of each individual axis, which is in a fixed direction, must vary. Therefore the desired velocity for each axis may change from one move to the next. To produce a constant overall velocity, we introduced blending between two moves.

The basic idea of blending is that, if the ideal velocity of the next move is higher than current one, do not stop at the target. Instead, hold current velocity until target is reached and start accelerating toward the new velocity. The same principle can also be applied to transiting from a higher velocity to a lower one. The idea is illustrated in Figure 10.

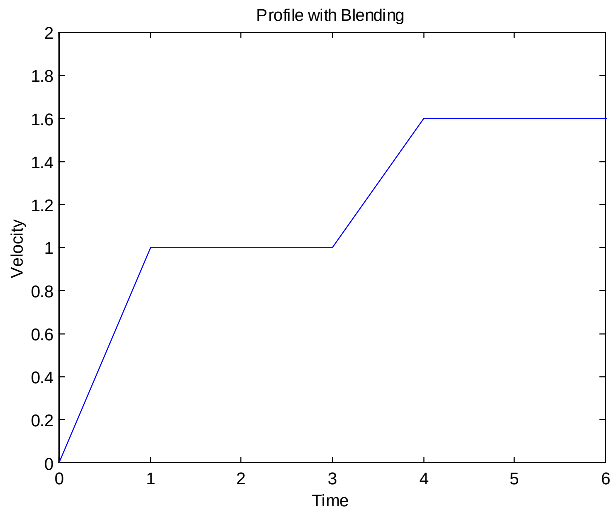


**Figure 10 Profile with Blending**

Blending, however, should be turned off if one or more axes are going to change directions. Experiments show that holding velocity in such situation will cause large overshot. The final motion task we used is a hybrid in that blending is turned on or off according to current condition.

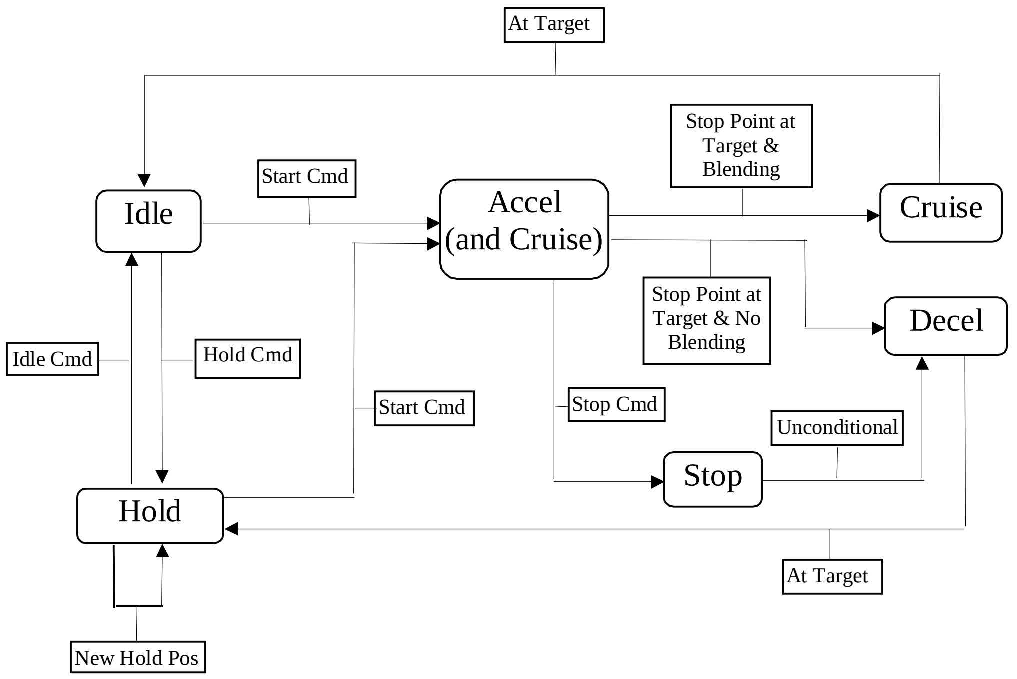


**Figure 11 Motion Task**

Figure 11 is the state logic for this new Motion task. Note that the Accel state actually includes part of Cruise stage. This is to enable blending to be turned off if desired. Motion won't go through a Cruise state without blending but a period of cruise is still needed for trapezoidal motion. By integrating a period of cruise into Accel state, trapezoidal motion can be generated even blending is off. Turning on/off of blending is done via message. Please see attached code for details.

Figure 12 shows a motion profile generated by this motion task. Only the motion of one axis is shown. The red line is command, the blue line is position profile and the green line is velocity. The blending effect can be clearly seen.

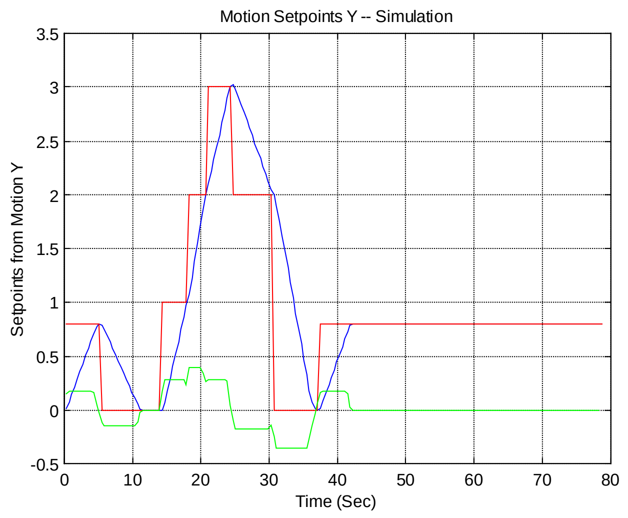


**Figure 12 Motion Set Points Y**

## Coordinated Movement

All three axes must move in concert so that the desired path is tracked. We have to make sure that:

1. At a higher level, none of them should start move if any of them have not finished the previous move;
2. All of them should transit from one stage of move -- acceleration, cruise, blending, etc-- to another at the same time. This is to ensure the cutter won't go off path.
3. Blending should be turned on or off for all of them during one move. If one axis moves with blending on and the other two with it off, the error will be too large. We have to relax constant velocity requirement to ensure adequate path tracking precision.

These are the responsibilities of Master task. The basic technique used here is time based rather that velocity based. In brief, the Master determines the deadline of the move for all the axes and let the axes themselves figure out the correct motion parameters (velocity and acceleration). The Master also decides whether to turn the blending on or off for all axes. The following segment of code taken from Master's Run function illustrates the ideas clearly.

```
case ComputeParas:
        // Entry Section
        if (RunEntry){
                // only run this on entry to the state
                // entry code goes here
        x2 = pPath->GetTarget(X);
                y2 = pPath->GetTarget(Y);
                z2 = pPath->GetTarget(Z);
                // Get target without setting home flag
                s = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)+(z1-z2)*(z1-z2));
                if(s < ErrTolerance){  // Too close. Same point
                        if(pPath->Home())
                                NextState = Idle;
                        else{
        pPath->Advance();
                                NextState = ComputeParas;  // This state again
                        break;
                        }
}

                DeadLine = s/Feed;
                if(DeadLine <= 8.*DeltaTaskTime) DeadLine = 8.*DeltaTaskTime;
                vx = (x2-x1)/DeadLine;
                vy = (y2-y1)/DeadLine;
                vz = (z2-z1)/DeadLine;
                // Velocity in each direction. They are signed!
                if(!FirstMove){
                        if(pPath->Home())
                                HoldCruise = false;
                        else{
                                pPath->Advance();
                                double xx = pPath->GetTarget(X,false);
                                double yy = pPath->GetTarget(Y,false);
                                double zz = pPath->GetTarget(Z,false);
                                // Get target without setting home flag
                                if( (xx-x2)*(x2-x1)<0. || (yy-y2)*(y2-y1)<0.
                                        ||(zz-z2)*(z2-z1)<0.)
                                        // At lease one axis will reverse direction in the next move
                                        HoldCruise = false;  // Won't hold velocity this time
                                else
                                        HoldCruise = true;
                                pPath->Back();  // This will reset home flag
                        }
                }
                else
                        HoldCruise = false;
                        // Have to move to start point. We want to stop there.

                pPath->ResetHomeFlag();
        }

        // Action Section
        // Test/Exit section
        NextState = Start;
        done = 0;               // Continue in this task
        break;
case Start:
        // Entry Section
        if (RunEntry){
                // only run this on entry to the state
                // entry code goes here
                MoveDone1 = MoveDone2 = MoveDone3 = false; // Reset
                if(AllReady()){ // All axes ready. i.e. at Idle or Stop state
                        // Send time limit for complete this move
                SendAllAxes(TRAJECTORY_MSG_DeadLine,0,DeadLine,
                MESSAGE_OVERWRITE);
                        SendRTMessage(Trajectory1, TRAJECTORY_MSG_Feed,0,vx,
                                MESSAGE_OVERWRITE);

                        SendRTMessage(Trajectory2, TRAJECTORY_MSG_Feed,0,vy,
                                MESSAGE_OVERWRITE);
                        SendRTMessage(Trajectory3, TRAJECTORY_MSG_Feed,0,vz,
                                MESSAGE_OVERWRITE);
                        if(HoldCruise)
                                SendAllAxes(TRAJECTORY_MSG_HoldCruise);
                        // Start:
                        SendRTMessage(Trajectory1, TRAJECTORY_MSG_Start,
                                MOTORMASTER_MSG_MoveDone1);
                        SendRTMessage(Trajectory2, TRAJECTORY_MSG_Start,
                                MOTORMASTER_MSG_MoveDone2);
                        SendRTMessage(Trajectory3, TRAJECTORY_MSG_Start,
                                MOTORMASTER_MSG_MoveDone3);
                        // Each has a different box to send completion msg
                }
                else{
                        SendAllAxes(TRAJECTORY_MSG_Home);
                        SendAllAxes(TRAJECTORY_MSG_DeadLine,0,1.,
                MESSAGE_OVERWRITE);
                        NextState = Start;              // Do RunEntry again
                }
        }
        // Action Section
        // Test/Exit section
        if(AllMoveDone()){
                x1 = x2;
                y1 = y2;
                z1 = z2;
                FirstMove = false;
                if(pPath->Home()){  // Job done
                SendCompletionMessage(UI,UIBox,MOTORMASTER_SUCCESS);
                        NextState = Idle;
                }
                else
                        NextState = ComputeParas;  // Compute parameters
        }

        if(GetRTMessage(MOTORMASTER_CMD_Stop,&MsgIndex,&MsgValue)){
                NextState = Stop;
                // Clear any old commands
                GetRTMessage(MOTORMASTER_CMD_Start,&MsgIndex,&MsgVal);
                GetRTMessage(MOTORMASTER_CMD_Idle,&MsgIndex,&MsgVal);
                GetRTMessage(MOTORMASTER_CMD_Reset,&MsgIndex,&MsgVal);
        }
        break;
```

**Figure 13 Master Task Code Excerpt**

The Trajectory task, on the other hand, is only responsible for one axis. Its task is to determine the parameters for motion task and trigger each targeted move according to Master’s command. The following code shows its work.

```
case ComputeParas:
        // Entry Section
        if (RunEntry){
        // only run this on entry to the state
                // entry code goes here
if(GetRTMessage(TRAJECTORY_MSG_HoldCruise,&MsgIndex,&MsgVal,
&FromTask))
                HoldCruiseNow = true;
                TargetPos = pPath->GetTarget(Axis);
                pPath->Advance(Axis);
                Accl = DefaultAccl;
                Vel = DefaultVel;
                DeadLine = -1.;
                Feed = -100000.; // Set to invalid value before intialize
        }
        // Check for new feed info

        if(GetRTMessage(TRAJECTORY_MSG_DeadLine,&MsgIndex,&MsgVal,&FromTask))
                DeadLine = MsgVal;
        if(GetRTMessage(TRAJECTORY_MSG_Feed,&MsgIndex,&MsgVal,&FromTask))
                Feed = MsgVal;
        if(GetRTMessage(TRAJECTORY_MSG_HoldCruise,&MsgIndex,&MsgVal,&FromTask))
                HoldCruiseNow = true;
        // Test/Exit section
        if(DeadLine>0. && Feed>=-100000.){ // Successefuly assigned
                // Compute Accl and Vel based on DeadLine:
                // The Motion is not capable of dealing with negative Vel and Accel
                Vel = Feed;
                Accl = fabs(Vel-VelCur)/(AccPeriod*DeadLine);
                if(HoldCruiseNow)
                        VelCur = Vel;
                else
                        VelCur = 0.;
                NextState = SendTarget;
                done = 0;       // Continue in this task
        }
        break;
case SendTarget: // Send target one by one until path is completed.
        // Entry
        if(RunEntry){
                // Send new start point:
                SendRTMessage(MotionTask,MOTION_MSG_STARTPOS,0,
                        CurrentPos,     MESSAGE_OVERWRITE);
                SendRTMessage(MotionTask,MOTION_MSG_ACCEL,0,fabs(Accl),
                        MESSAGE_OVERWRITE);
                SendRTMessage(MotionTask,MOTION_MSG_VCRUISE,0,fabs(Vel),
                        MESSAGE_OVERWRITE);
                SendRTMessage(MotionTask,MOTION_CMD_NEW_PAR);
                if(HoldCruiseNow)
                        SendRTMessage(MotionTask,MOTION_CMD_HOLD_CRUISE);
                // Send a move command to Motion task
                if(fabs(TargetPos-CurrentPos)<1e-7) // Same point
                        SendRTMessage(MotionTask,MOTION_CMD_HOLD_POSITION,
                        TRAJECTORY_MSG_MoveDone, // The msg box to receive completion msg
                        CurrentPos, MESSAGE_OVERWRITE); // Hold it
                else
                SendRTMessage(MotionTask,MOTION_CMD_START_PROFILE,
TRAJECTORY_MSG_MoveDone,
                                TargetPos);                     // Target point
                PrivateGlobalData[TRAJECTORY_GLOBAL_Status] =
                                TRAJECTORY_STATUS_SendTarget;
                AtHome = false;
        }
        // Action
        // Test/Exit
        if(GetRTMessage(TRAJECTORY_MSG_MoveDone,&MsgIndex,&MsgValue)){
                // Move is finished
                if(MsgIndex == MOTION_SUCCESS){// Successful Move
                        if(Boss >= 0 && BossBox >= 0)// Make sure boss exits
                                SendCompletionMessage(Boss,BossBox,TRAJECTORY_SUCCESS);
                                // Notify boss
                                CurrentPos = TargetPos;
                                NextState = Idle; // Wait for next command from boss
                }
                else
                StopControl("<Trajectory> Send target failed\n");
        }
        break;
```

**Figure 14 Trajectory Task Code Excerpt**

Note that after completing a move, Trajectory task will transit to Idle state waiting for Master's command instead of starting next move. This is to make sure the move of all three axes is indeed coordinated. Trajectories are isolated from each other so that each can focus on one single axis. Only Master has information of all axes that it uses to coordinate the operation. In this way, a clear line is drawn between of responsibilities of different tasks and the software structure is simple and robust.

### Motor Class

The motor class used in this project was developed for the earlier Motor-Lab project. It is a class with an overloaded constructor that allows for using it for a Simulated, Stepper, Analog, or PWM motor.

### PWM Class

The PWM class used in this project was developed so that a single class may be instantiated several time for different motors. It also has the feature of being able to operate in simulated or real mode.

### Dynamic Data Exchange

Dynamic Data Exchange is used to establish communication between GUI that acts as a server and control software that acts as a client. A task, DDEClient is constructed to do this job. There is no state for this task. All it does in its Run function is to check if new data has been sent from GUI. Currently only file name passing is supported. This is necessary as well as sufficient for cutting a part with arbitrary shape. Other data exchange will be added later. If file name is updated by server, DDEClient task will verify the existence of the file and invoke Calculation task to generate path from the file.

The control process has its own text user interface (Described in other section sections) and can run along without GUI.

## Results

### Simulation

The control algorithm and state logic as well as the motion profile was first tested in simulation. The following figures plot the simulation results, which show that the approach is feasible.

Figure 15 to Figure 19 are the results of a cutting polygon that is an excellent example to show the blending work.

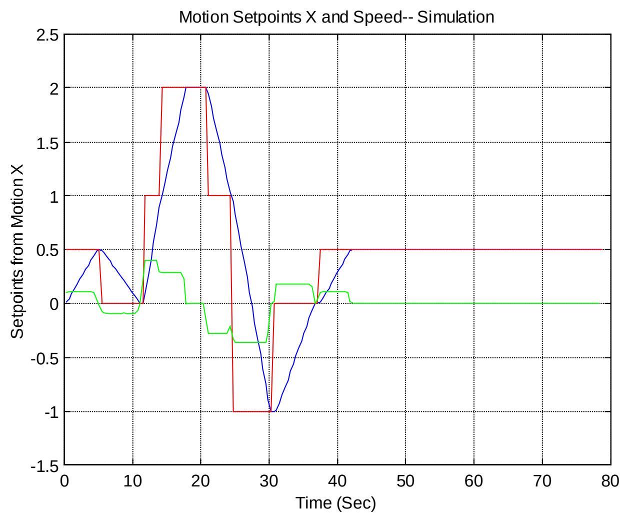


**Figure 15 Motion Set Points X**

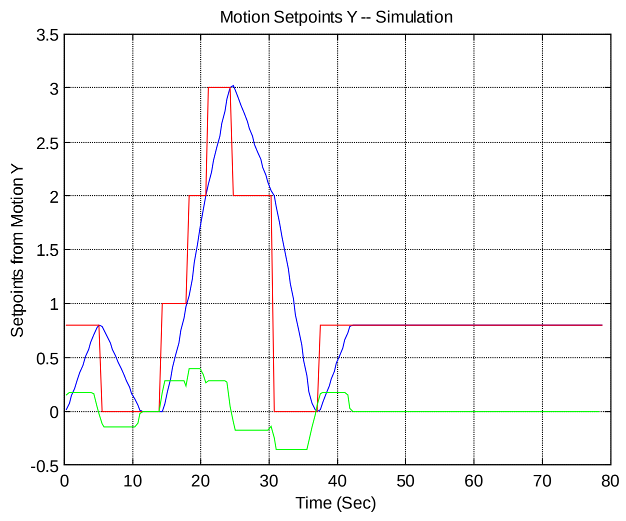


**Figure 16 Motion Set Points Y**

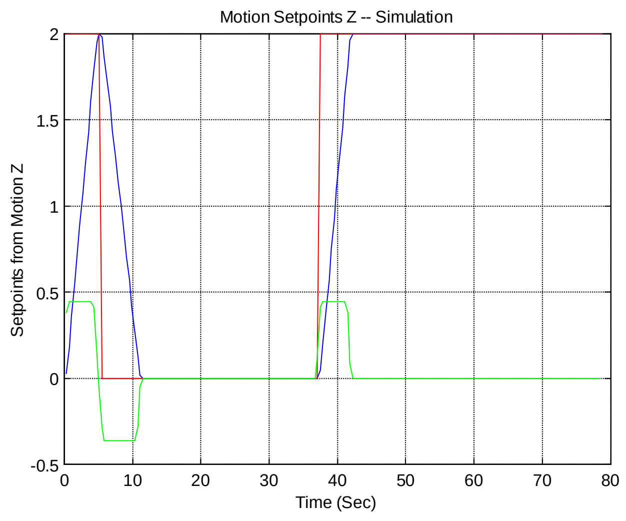


**Figure 17 Motion Set Points Z**

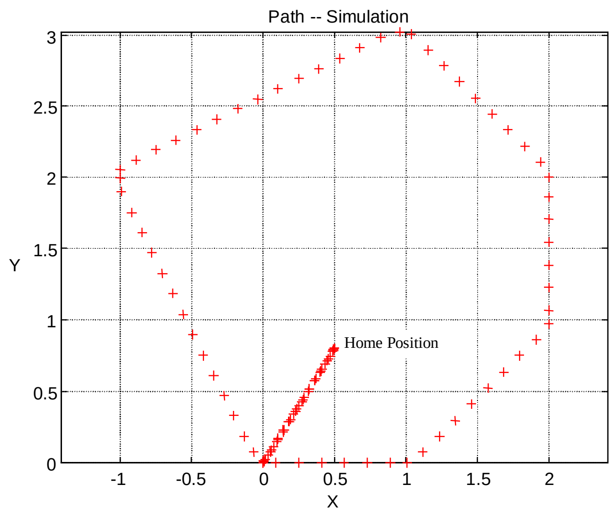


**Figure 18 Polygon Primitive Path**

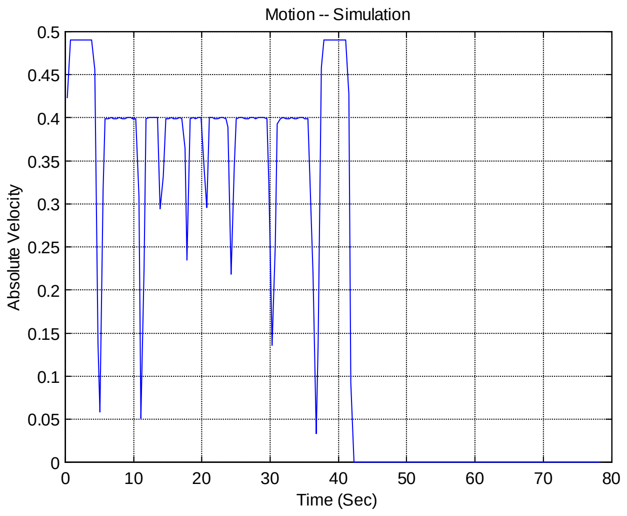


**Figure 19 Motion with Blended Velocity**

Figure 20 to Figure 23 show a smooth curve – a circle.

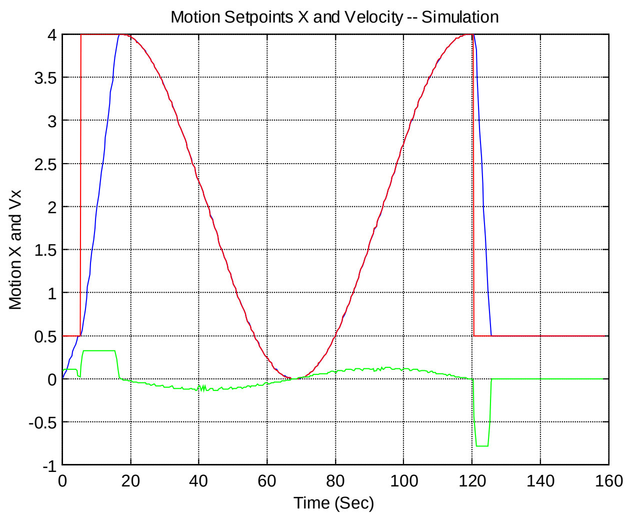


**Figure 20 Motion Profile for Circle (X-Axis)**

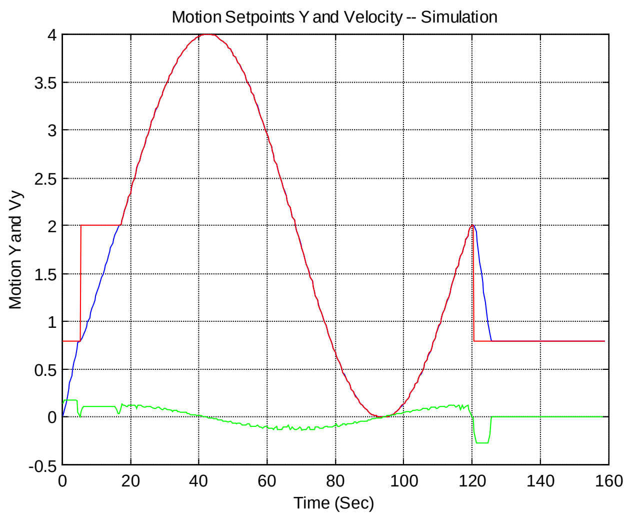


**Figure 21 Motion Profile for Circle (Y-Axis)**

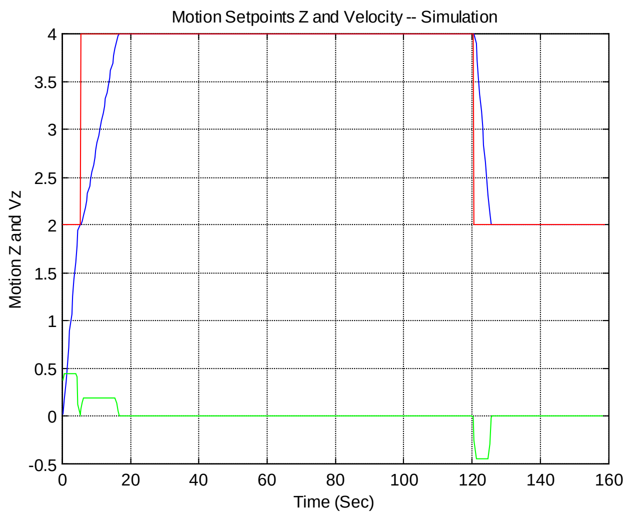


**Figure 22 Motion Profile for Circle (Z-Axis)**

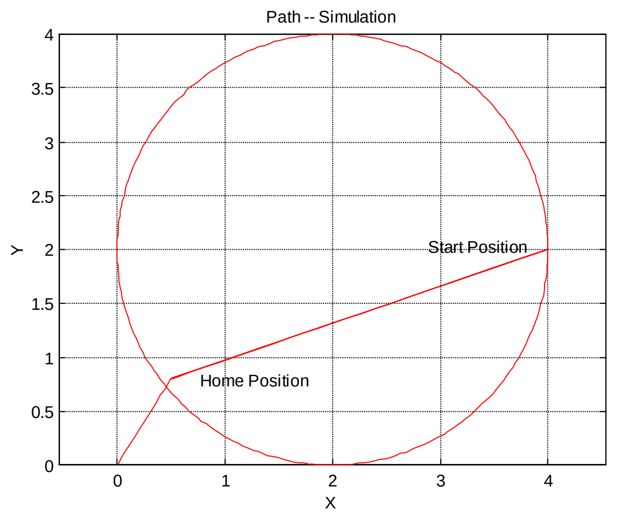


**Figure 23 Coordinated Circular Path (XY)**

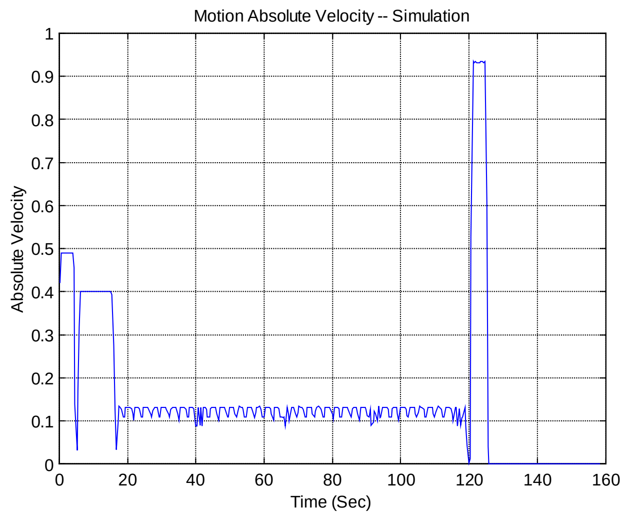


**Figure 24 Feed Profile for Chinese Sun Symbol**

Finally, Figure 24 to Figure 28 are the results of a path defined in a file. This is a Chinese character which means the Sun. The same file with modified scale is used to test our machine and software in real time.

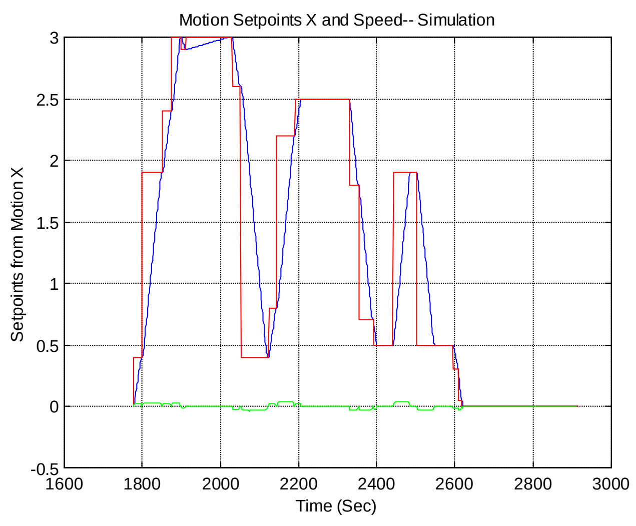


**Figure 25 Motion Profile for Chinese Sun Symbol (X-Axis)**

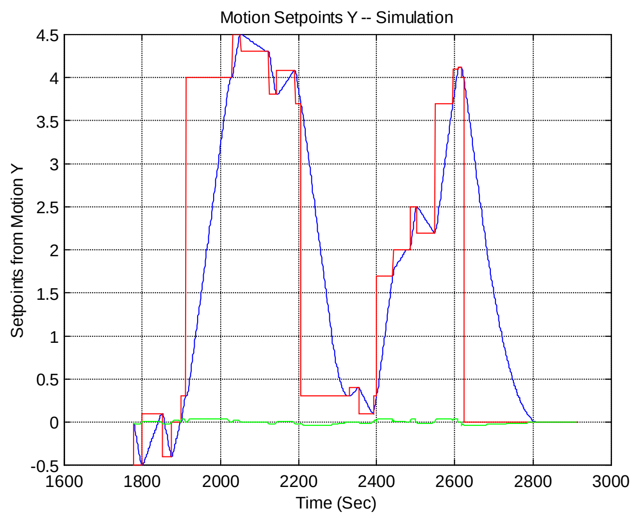


**Figure 26Motion Profile for Chinese Sun Symbol (Y-Axis)**

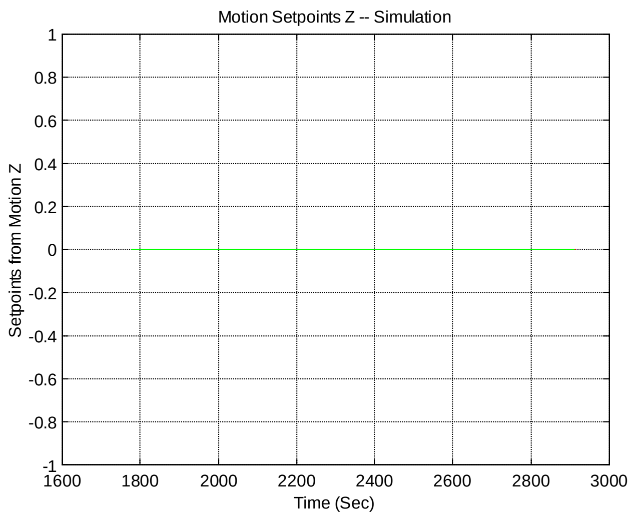


**Figure 27 Motion Profile for Chinese Sun Symbol (Z-Axis)**

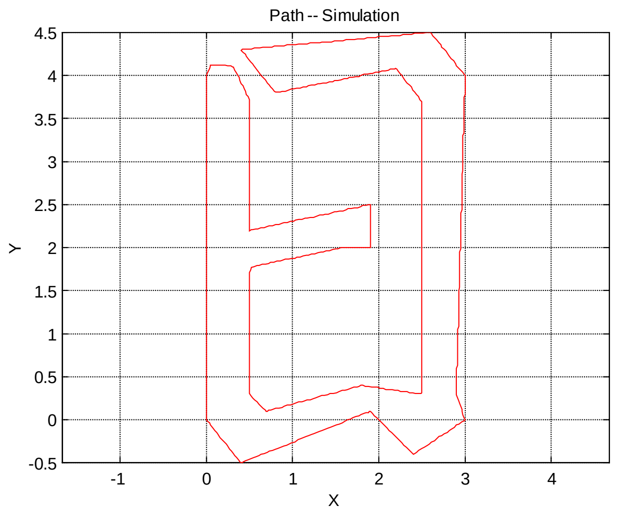


**Figure 28 Motion Profile for Chinese Sun Symbol**

## Real Time Performance

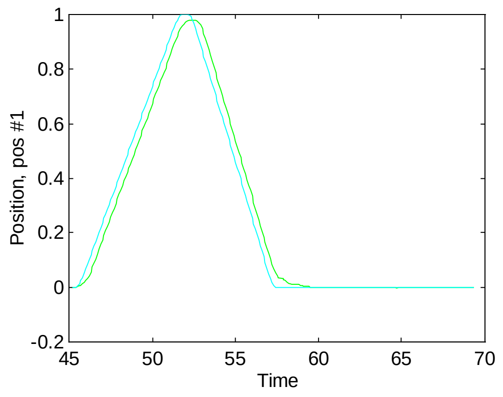


**Figure 29  X-Axis Linear Move**

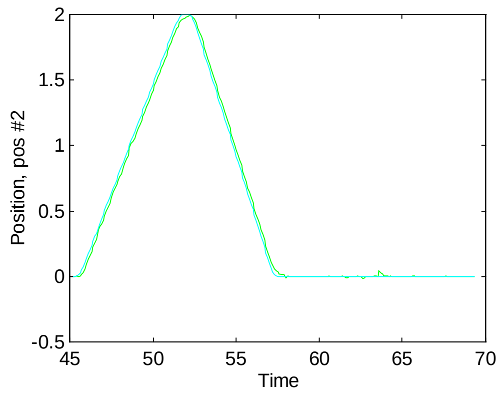


**Figure 30 Y-Axis Linear Move**

Figure 29 and Figure 30 show the typical servo performance of the machine tool during a coordinated XY move. Position 1 is the x-axis position. Since the x-axis carries the y-axis it is heavier. As such the proportional gain was set much higher. The control effort to overcome sticktion was small relative to the control effort to move the mass so that very little jitter occurs. Position 2 is the lower mass y-axis. In this case the gains had to be lower, and the control effort to overcome sticktion was not small relative to the control effort to move the mass. This is why jitter is present during the move and especially when the axis is holding position (60 s - 65s). Notice also that the larger mass system (x-axis) has a steady state following error, while the lower mass (y-axis) has relatively low following error. The steady state following error on the x-axis is less than 0.06 inches. The jitter error on the y-axis is less than 0.05 inches. Figure 31 shows how the coordinated XY trajectory looks. While not a perfect line, it does look good.

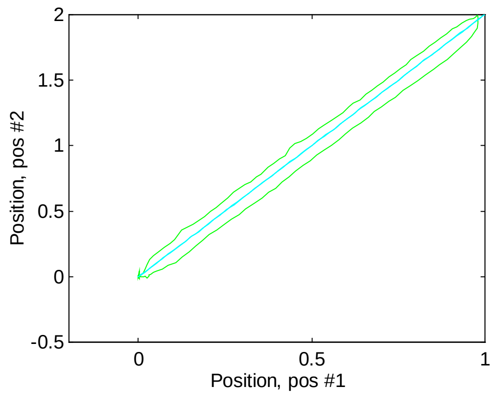


**Figure 31  XY Coordinated Linear Move**

This servo performance was achieved in Windows NT without interrupts. When running the system in DOS, no better performance was achieved compared to running in NT with no other applications running. Therefore, the controller was executed in the NT environment.

## User Interface

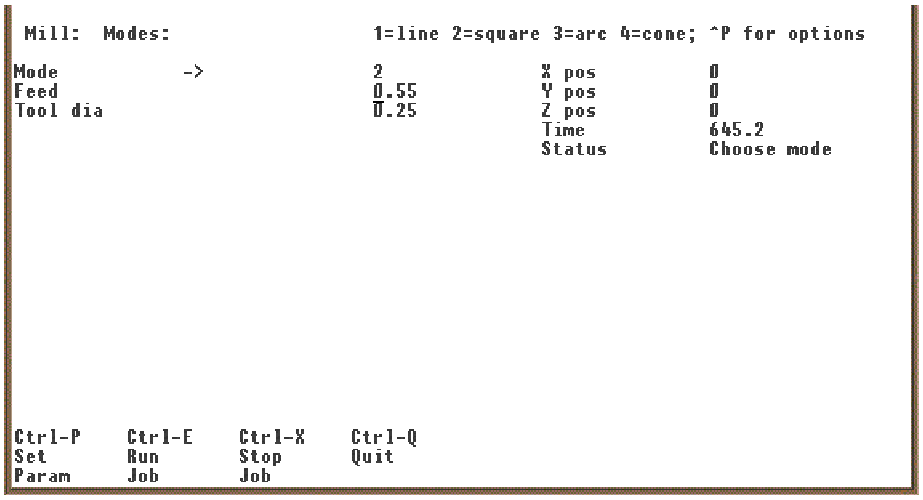
Two versions of the user interface were constructed, a text-based and a graphic version. We took the opportunity to create a graphic interface that contributed a unique capability to the system, rather than to just mimic the operations of the text-based interface. The graphic interface was constructed using Excel

97and Visual Basic for Applications, which is packaged with Excel and connected to it via an object library.

## Text-Based User Interface

The text-based interface consists of several windows or screen objects. The user may toggle between windows, which are set up on a hub basis, with the Main window as the hub and the various object windows accessible from Main but not from each other.

*Main* Window: Keyboard commands in the Main window (Figure 32) allow the user to control the start of a job (^E), the termination of a job (^X), and quitting the program (^Q). Not shown in the figure are two keyboard command features added late in the development of the program, one to adjust the feed rate while the job is in progress (^F), and another to allow the user to send the axes to home position (^O). All keyboard commands are transferred to other tasks in the program via single-bit command messages.



```
 Mill:  Modes:                1=line 2=square 3=arc 4=cone; ^P for options

Mode           ->             2              X pos          0
Feed                          0.55           Y pos          0
Tool dia                      0.25           Z pos          0
                                             Time           645.2
                                             Status         Choose mode

Ctrl-P    Ctrl-E    Ctrl-X    Ctrl-Q
Set       Run       Stop      Quit
Param     Job       Job
```

**Figure 32 Main Window**

There are three lines on the left for the selection of options, one of which has to do with machine control: Feed Rate, which is used by the program to determine the resultant speed at which the mill bed moves from point to point in three dimensions. Feed rate is also output as a message, using the slot for a double. On the right are five lines that are supervisory in nature. The first three give position feedback, which should come directly from the encoder-reading task; the fourth is time, and the fifth is job status, which also serves to deliver prompts and error messages (for example, the screen shown gives the prompt "Choose Mode," which is the next step the user needs to take). "Job Status" is not based on a message from other tasks; it is internal to the user interface task and is implemented as part of the key command function (for example):

```
//  Function:  RunJob
//  Function to start execution of job (^E)
void RunJob()
    {
        runJob = 1;
        strcpy(status,"Running");
    }
```

The Main window has two more lines of options on the left side of the screen . The first allows the user to select an object type (a summary of the options is given to the right in the title line). Two options were added later in the project: “File,” which allows the user to provide the name of a file containing point data

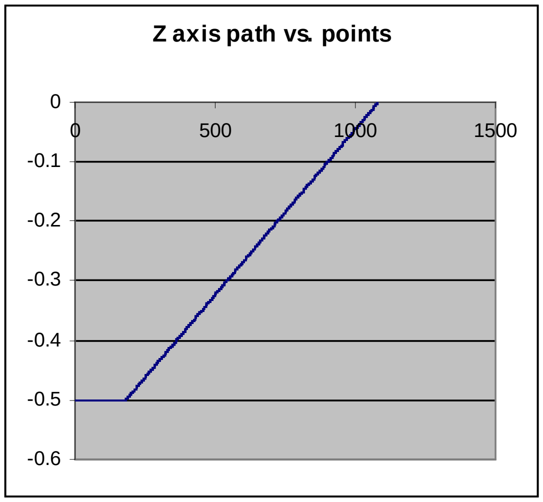


Figure 33 Z-Axis Path

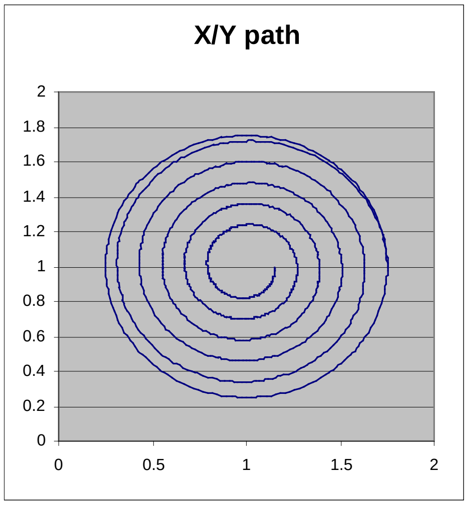
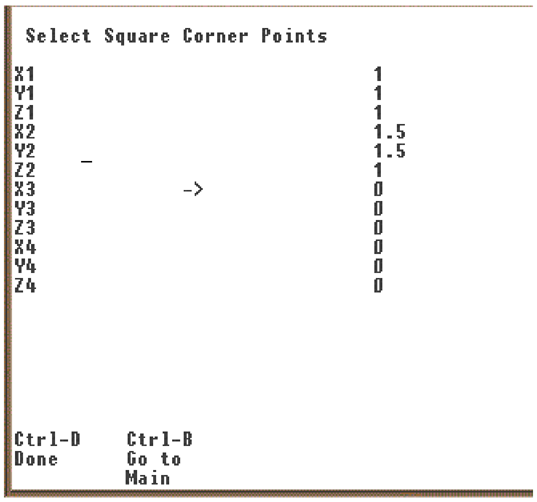


Figure 34 XY-Axis Path

constructed outside the context of the program; and “Cone,” which allows the user to enter parameters for the construction of a spiral cone (see Figure 33 and Figure 34for a plot of the (x,y) points and the z-axis path calculated by the Calc task for an object of this type). The keyboard command for switching to the object type window selected was originally ^P, but this turned out to have a conflict with some versions of DOS, and was changed to ^Y.

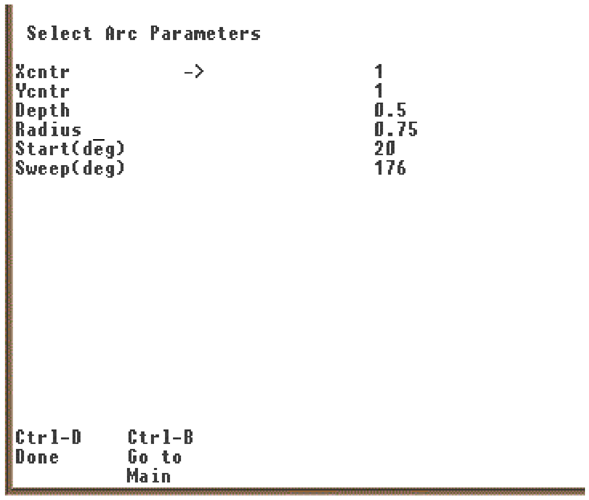


```
 Select Square Corner Points

X1                              1
Y1                              1
Z1                              1
X2                              1.5
Y2                              1.5
Z2     _                        1
X3              ->              0
Y3                              0
Z3                              0
X4                              0
Y4                              0
Z4                              0







Ctrl-D     Ctrl-B
Done       Go to
           Main
```

Figure 35 Square Object Window

*Line* and *Square* windows: These objects represent a primitive form of design input, which are appropriate only to simple forms. The user is allowed to enter directly the coordinates for the points that define the objects (see, for example, Figure 35). The Square type actually allows the definition of any quadrilateral. The fact that these object types are named as two-dimensional figures reflects a fundamental quality of three-dimensional orthogonal machines to which we have all become accustomed through practice, and seldom if ever think about – namely, that we can construct symmetrical operations in only two of three dimensions at one time. This fact of space leaves the third axis as “odd man out.” In the case of a milling machine, this fact is expressed in the symmetrical and interchangeable two-axis movement of the milling bed (horizontal x and y coordinates), and the unique tool-movement axis (z). Motions in the z direction create a type of action which is very different from the motions of the mill bed. X and y motions can be interchanged without affecting the ultimate form of the object, but neither x nor y can be interchanged with z. Thus the z axis is typically conceptualized as “depth of cut,” and x,y motions as a two-dimensional figure.

*Arc* and *Cone* windows: These parametric types bring a greater sense of the difference between the bed motions and the tool motion, in that height or depth is explicitly called out. The Arc window (Figure 36) is constructed in step-by-step fashion in one of several ways in which a designer thinks about arcs. This pattern, which defines the center location, the radius, the start angle (angle of the line between center and end point relative to “horizontal”), and the sweep (arc radius), is perhaps the most common way of thinking about arcs. The description given is also appropriate because it is path-oriented, as the movement of the tool through a path on the bed is the essential operation of a milling machine. A circle is a member of this type, being defined as a 360° arc. Note that a quadrilateral can also be defined parametrically, in similar fashion: first, start point (x, y coordinates); then direction and length plus change in depth three times (the fourth move is constrained to return to the start point, and should thus be generated automatically). It is the nature of a parametric description that the geometry is quantified in the simplest possible way, and that the software takes the task of converting these numbers to the sets of coordinate data used for motion control. Thus, for example, the arc data is sent to a calculation task which makes the following conversion to an array (number of points x 3) of doubles:



```
 Select Arc Parameters

Xcntr          ->             1
Ycntr                         1
Depth                         0.5
Radius _                      0.75
Start(deg)                    20
Sweep(deg)                    176











Ctrl-D    Ctrl-B
Done      Go to
          Main
```

**Figure 36 Arc Object Window**

```
case ARCTYPE:
    xc = GetGlobalData(MainOpInt,0);
    yc = GetGlobalData(MainOpInt,1);
    z = -(GetGlobalData(MainOpInt,2));
    R = GetGlobalData(MainOpInt,3);
    ThStart = GetGlobalData(MainOpInt,4); // Degrees
    Sweep = GetGlobalData(MainOpInt,5);
    numPoints = (int)Sweep*2;  // e.g., 720 points for a circle

    //   Create array in two steps
    Point = new double*[(numPoints+2)];
    for(i=0;i<(numPoints+2);i++)Point[i] = new double [3];

    //   Distribute array of coordinate values into 3x array of points
    //   Set first point to (0,0,0)
    for(j=0;j<3;j++)Point[0][j]=0;
    //   Calculate rest of points:
    for(i=1; i<(numPoints+2); i++)
    {
        Point[i][0] = xc+R*cos((M_PI/180)*(ThStart+(i-1)/2.0));
        Point[i][1] = yc+R*sin((M_PI/180)*(ThStart+(i-1)/2.0));
        Point[i][2] = z;
    }
```

The spiral cone object is similar in nature, but a little more complex. The parameters “Step Height” (=stepUp variable) and “Step Width” (=stepRadius variable) are an attempt to put the essential geometric information which defines the upward movement and the inward spiral in easily understood form:

```
case SPIRAL_CONETYPE:
    xc = GetGlobalData(MainOpInt,0);
    yc = GetGlobalData(MainOpInt,1);
    zHeight = GetGlobalData(MainOpInt,2);
    z = -zHeight;
    R = GetGlobalData(MainOpInt,3);
    stepUp = GetGlobalData(MainOpInt,4); // Degrees
    stepRadius = GetGlobalData(MainOpInt,5);
    numPoints = 180*(zHeight/stepUp+1);  // 180 points for each circle

    //   Create array in two steps
    for(i=0;i<(numPoints+2);i++)Point[i] = new double [3];

    //   Distribute array of coordinate values into 3x array of points
    //   Set first point to (0,0,0)
    for(j=0;j<3;j++)Point[0][j]=0;
    //   Calculate initial circle points in 2 degree steps:
    for(i=1; i<(182); i++)
    {
        Point[i][0] = xc+R*cos((M_PI/180)*2.0*(i-1));
        Point[i][1] = yc+R*sin((M_PI/180)*2.0*(i-1));
        Point[i][2] = z;
    }
    //  Calculate rest of points:
    for(i=182; i<(numPoints+2); i++)
    {
        R = R-stepRadius/180;
        z =  z+stepUp/180;
        Point[i][0] = xc+R*cos((M_PI/180)*2.0*(i-1));
        Point[i][1] = yc+R*sin((M_PI/180)*2.0*(i-1));
        Point[i][2] = z;
    }
```

## Graphic User Interface

This interface is called the “BlobGUI” because it attempts to make it possible to mill a shape that is either impossible or very difficult to describe in geometric terms. This interface represents a third distinct type. Unlike the coordinate-based and parameter-based types, which use numbers to describe geometry, this interface allows the user to create a drawing, which the program then processes into a set of numbers to describe the shape. Excel allows us to create a field of pixels which have graphic format, address (Row and Column numbers) and content. These properties make it possible to manipulate space in a way which is both visual and mathematically significant. Furthermore, the ability to link this field of objects to executable code via Visual Basic for Applications makes it possible to automate and make interactive the whole process.

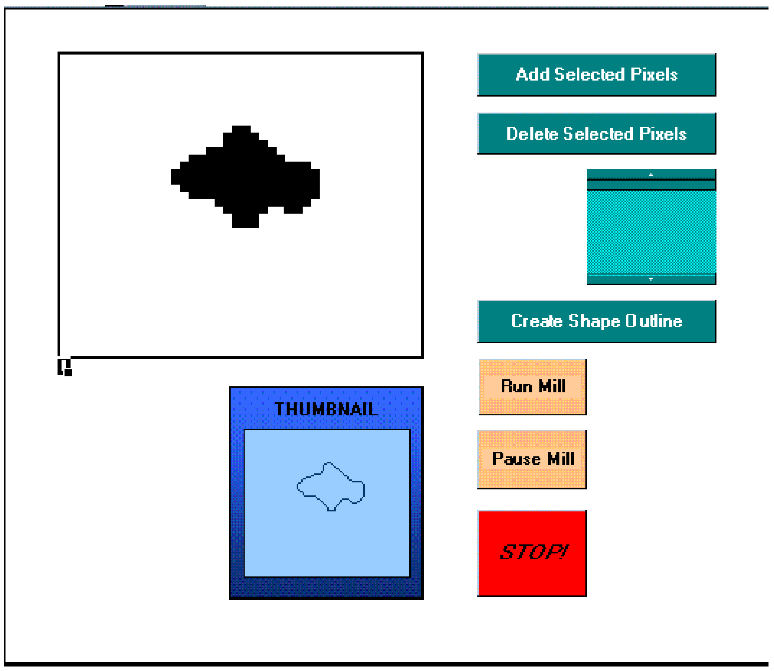


**Figure 37 Blob Graphical User Interface**

In its current form this object is two-dimension complex with a simple third-dimensional input, but it would be possible to profile the depth of cut using a similar technique. The window (see Figure 37) also contains simple machine controls to start, pause, and stop the cutting process. In the upper left quadrant, the user draws the object to be cut by selecting "pixels" (actually, Excel cells reduced in size) and pushing the "Add Selected Pixels" button. Pixels can also be removed using the "Clear Selected Pixels" button. The form can be either a solid shape or an outline, but the outline must close. The "Create Shape Outline" button then causes the program to select the boundary cells in counterclockwise fashion and to generate an array of coordinate values usable by the machine. Below this drawing field is a thumbnail graph which automatically shows the outline produced by this process. Note that the inside corners are "cut," reflecting the fact that these points are actually interior to the figure. The effect would become less noticeable as the grain of the drawing is reduced to something closer to screen pixel size. Since this is a demonstration project, the pixel size has been left large to make the process more visible and for greater ease of manipulation. This cutting of the corners is actually the first step in a smoothing process that is completed in the calculation and trajectory tasks, to create a series of line segments that approximate a curving shape. The thumbnail drawing is an interactive device which give useful feedback on what is actually included in the drawing (for example, "islands" will either not process or preempt the processing of the main object). It will automatically update as various shapes are tried out, and the set of points will not be sent to the main program until the "Run Mill" button is pushed.

The complete code for the Visual Basic program which makes all this work is quoted in Appendix X. Of note is the algorithm which finds the next boundary pixel in the same sequence which will be taken by the milling operation (the first point is just found by scanning the field). A series of "ElseIf" instructions test for the presence of a cell which is part of the shape in a counterclockwise fashion around the first point (see Figure 38). Depending upon which of these ElseIf instructions results in a cell being found, the algorithm knows whether it has turned a corner and must rotate 90° to another mode (the modes are "Right," "Up," "Left," and "Down"). In the example shown in the figure, the search for the next pixel ends at point 2, and the algorithm knows it must then switch from "Right" mode to "Down" mode.

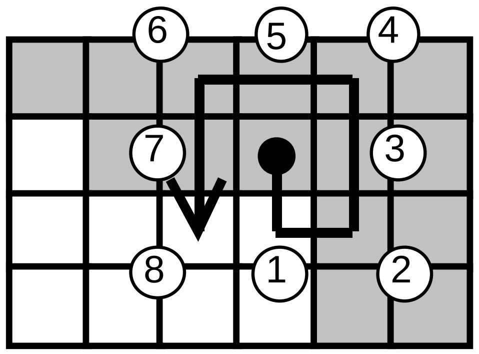


**Figure 38 Search Algorithm**

Note that this project was not completed to the point originally conceived. The buttons for machine control are not attached to any code for the production of commands, which would have defined DDE connection information. The "Run Mill" button, for example, would have produced a flag to start the machine operation, and named addresses for this flag and for the data array of coordinate information to define path geometry. There is also only the beginning of a button for selecting depth of cut (the green square below the "Clear Selected Pixels" button). Further, a window or windows for prompt and error messages would need to be created, as well as code which prevents the user from entering illegal actions.

## Discussion

The results from this project were extremely satisfactory. The performance of the servo system was better than the performance of the servo systems in the "MotorLab." While this performance does not rival that of an industrial quality machine tool, it is very good for a purely software driven electro-mechanical device which was developed in less than four weeks. The parts made by the milling machine provide concrete evidence of the utility of the system.

The mechanical system had some room for improvement. The slides exhibited some stiction that could have been reduced with better tolerances during fabrication. Some performance improvements could be achieved by following some of the machine tool builder's common practices. Such common practices include using dislike materials at sliding joints to reduce galling, kinimatically constraint moving parts with proper numbers of degrees of freedom, and drive mounting techniques.

The software could have been improved with more time. While the software developed performed well, the system could benefit from expanding the software capability. Such features as manual jogging motion program editor, and error monitoring could provide more utility.

Another set of user interface operations, which would have been very useful to this project team, was not anticipated. These would have allowed access to the variables that have to do with tuning the program's system and feedback modeling, and to monitor the handling of crucial system parameters to help understand and locate problems. It was particularly difficult to adjust the PID constants by closing down the program, then modifying and recompiling the code, over and over.

## Conclusion

The outcome of this project is a real reward for our time and effort – we built a REAL milling machine that, with the software we wrote, is capable of cutting REAL parts! The control is effective and the precision achieved is quite satisfactory. Also the communication between GUI and control works well.

The milling machine fabricates parts with arbitrary shape and complexity within the working volume of this machine is 3"x3"x3" and accuracy of about 0.050 inches. The material in which the machine can machine was limited to soft wood and perhaps wax (although not tested).

The user interface provided a means of monitoring the machine performance while also providing a flexible method for defining and transmitting arbitrary part geometries for the machine to fabricate.

While this project did not revolutionize the mechatronic world, it did provide some small ingenious new vantages to mechatronic/machine tool technology.