Remote Real Time Milling

Dan Odell
Shad Roundy

ME230
Professor Auslander
12/9/00

Abstract

The remote real time milling project aims to create a software system that allows a user to design and mill a part concurrently over a network. The software was implemented in the Java programming language and uses the RTX real time environment and the TranRunJ scheduling package. The high level software design follows the task state methodology. The design environment runs on one computer, which communicates over a network with the other computer that controls the mill. The system is modular in nature and is capable of controlling multiple mills simultaneously over a network. The software controls a SpectraLight 3 axis mill from Light Machines Corp. Stepper motors actuate the x, y, and z axes of the mill. Because there is no hardware to output the square wave that runs the stepper motors, this must be done in software. Therefore, the feed rate is limited by the minimum task interval of the Cutter task, which sends the pulses to run the motor out the serial port of the computer. It was found that the maximum possible feed rate for this system is 1.77 inches per minute, which corresponds to a minimum task interval of 3 milliseconds. At this speed, the average latency for the Cutter task is 0.2 milliseconds, which corresponds to an average 6% velocity error. This velocity error, however, does not correspond to a position error because of the nature of stepper motors. The actual positions that the mill cuts precisely match those designed by the user. Although the final software system does not run as fast as originally hoped, it does allow the user to effectively design and mill a simple part in real time over a network.

Introduction

The overall goal of this project was to create an application that allows a user to design a simple part and mill the part at a remote location in real time. Generally, milled parts are designed and saved in a design data file, which is later read and planned for milling. The desire was to be able to plan and mill the parts in real time so that the user could see (possibly via a webcam) the features being milled while creating them. This goal, of course, requires that the design interface and the mill controller run on separate

computers connected over a network (or possibly the internet). It is also important that the mill cutter be able to run fast enough so that the user does not have to wait an excessive amount of time to see their design being milled. Originally, the plan was to try to move the mill cutter more or less in sync with user's mouse. However, it was quickly realized that this would require far more computing power and network bandwidth than was available, and a different type of mill. It was therefore decided that it would be sufficient to cut at approximately 3 inches per minute, which is a fairly standard feed rate. In the end, the mill was only able to cut at 1.77 inches per minute maximum. Another crucial goal is that the actual contours that the mill cuts match those drawn by the designer. A final goal was to develop the software in a modular fashion so that multiple mills could be controlled by a single user to manufacture multiple parts simultaneously.

The software was implemented using the Java programming language, the TranRunJ scheduler package, and the RTX real time environment. RTX is a real time addition to the Windows operating system that runs below Windows and doesn't allow Windows to interfere with the real time scheduling of the control software. This allows precise timing control by the software. The project software was designed using a task/state structure. The design interface and associated classes running on the user's computer run under the Windows operating system while the mill controller runs in the RTX environment. The TranRunJ scheduler handles both the scheduling of tasks and communication over the network between processes.

The SpectraLight 3 axis mill from Light Machines Corp. was used. A picture of the mill is shown in Figure 1.1. This mill has 3 stepper motors that actuate the x, y, and z axes and a servo motor that actuates the spindle. The spindle motor, however, is controlled by a dial on the machine tool and is not controllable by the computer. There are no feedback sensors on any of the motors and it is assumed that the when a pulse is given to the steppers, they respond accurately. Velocity control is possible only as open loop control.

Figure 1.1: Spectra Light Mill Used in Project

Implementation

A modified task diagram of the overall software implementation is shown in Figure 2.1.

The diagram is partitioned to show which parts of the software run on which computer, and in which process. Process One runs on the user's computer under Windows, Process Two runs on the mill's computer under RTX, and Process Three runs on the mill's computer under Windows. The blue boxes represent tasks, the yellow boxes represent important classes that are not implemented as tasks, and the green boxes represent hardware. The solid arrows represent data or command signal flow. The dashed arrows represent an instantiates or uses relationship. The user draws contours and pockets in the user interface, which is implemented primarily in the DrawWin and the TheGui classes. This data is stored in a ConnectedObject class that is managed by DrawWin. A ConnectedObject contains all of the data associated with one contour, or connected set of points. The DrawWin, therefore, contains a Vector of ConnectedObjects that collectively contains all of the design data. The PathPlanner class plans the paths of primitive objects, such as a rectangle, at a high level. The UITask takes the data stored in a ConnectedObject and sends it, one point at a time, to the DataReader task which reads that data and reconstructs a ConnectedObject on the computer that controls the mill. The Planner task takes the x, y coordinates and depth contained in a ConnectedObject and

converts them to actual pulses to send to the stepper motors. The Cutter task actually sends the pulses out the computer's parallel port to the mill. The DataLogger task logs the x, y, and z positions sent to the mill as well as latencies and task durations. Finally the BlackBox is a piece of hardware that takes 8 bits from the parallel port and routes the appropriate signal to the stepper motors.

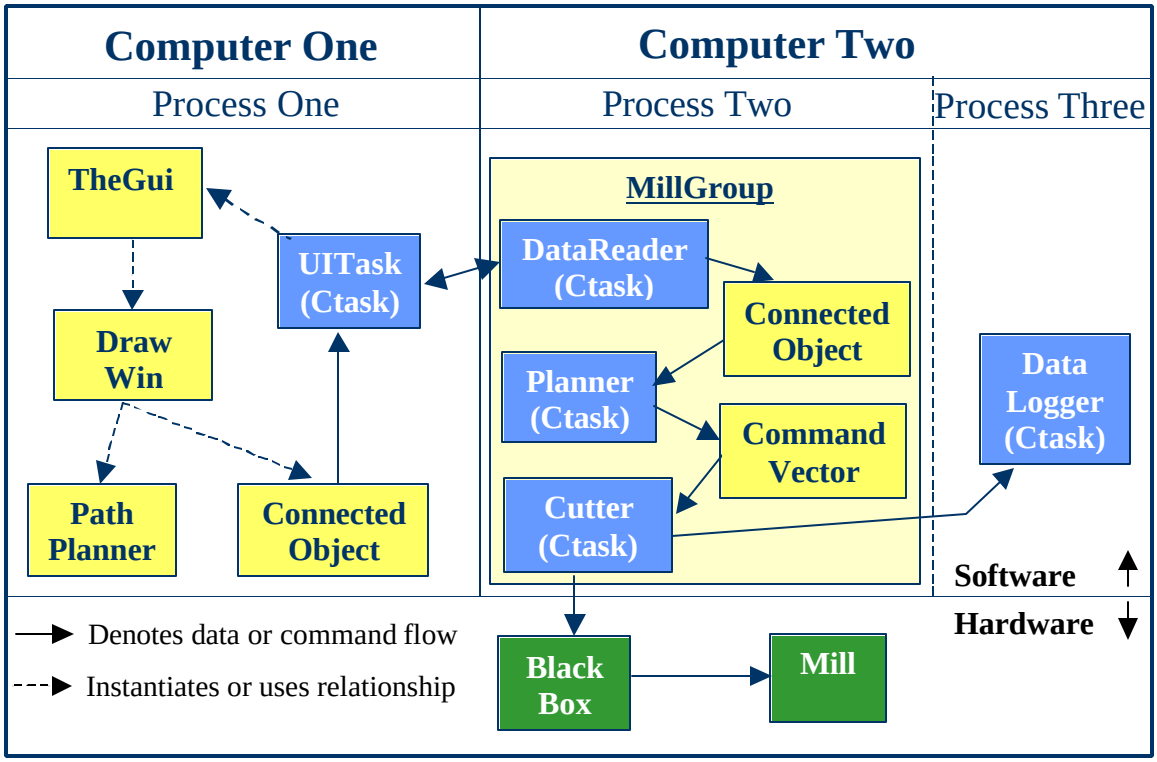

Figure 2.1: Class and Task Diagram for Software Implementation

For discussion purposes, it is convenient to partition the software into two parts, the design interface which runs on the users computer, and the mill controller which runs on the computer connected to the mill. A more detailed discussion or each of these parts follows.

Design Interface

The design interface is shown in Figure 2.2. There are four drawing tools available to the user. The "circle", however, is not yet implemented. Users can select the depth at which they wish to cut, and they can select a different depth for each contour.

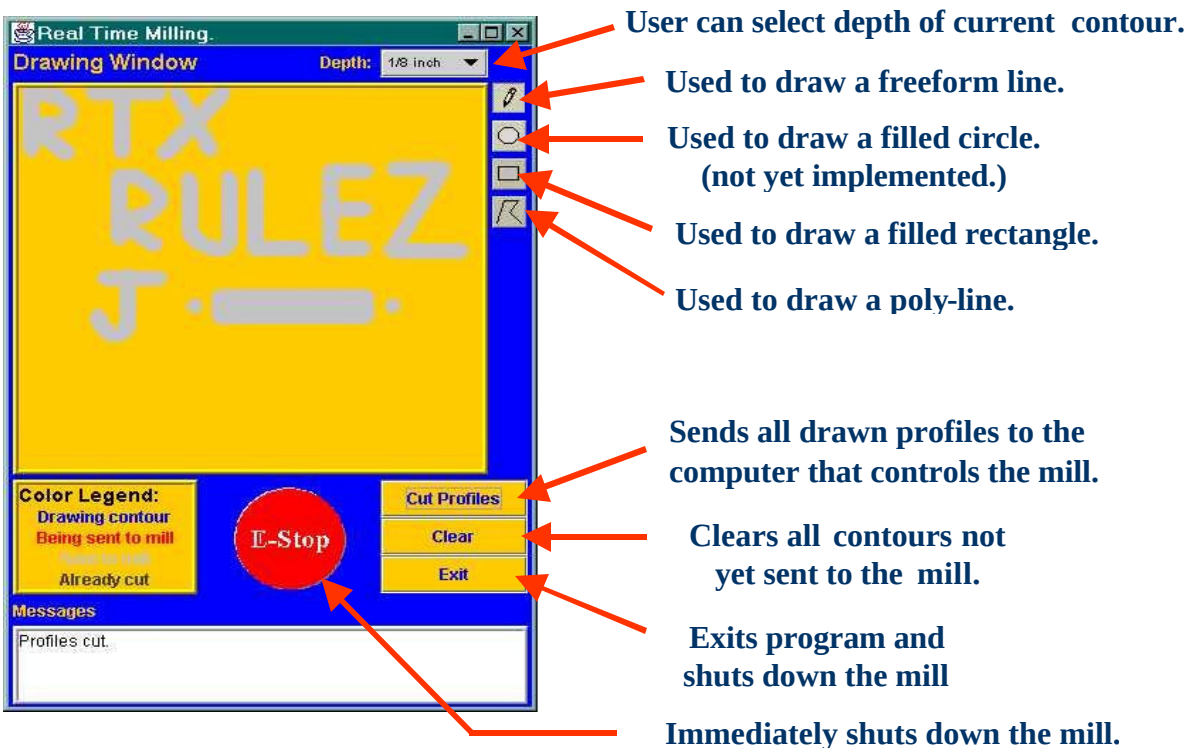

Figure 2.2: Design Interface

The tool diameter is known by the user interface, so as the user draws, the geometry of the actual cut (correct width and corner radii) is displayed in the drawing window. If the user selects the freeform line or poly line, the points defined by the user are put into a single ConnectedObject. If the user draws a pocket (rectangle), the PathPlanner creates x and y points for the entire tool path, and collects those points in a ConnectedObject. The ConnectedObject contains an array of x coordinates, an array of y coordinates, and a depth of cut. Each separate contour constitutes a single ConnectedObject.

When the user clicks the "Cut Profiles" button. All of the ConnectedObjects that have not yet been sent to the mill are marked. The UITask task has only two states: the Wait state and the Send state. In the Wait state the task checks to see if the DataReader is ready to read a ConnectedObject. (The DataReader sets a global variable indicating when it is ready to read another object.) If the DataReader is ready, it then checks to see if there are any ConnectedObjects which have not been sent, but which have been marked. If there are ConnectedObjects to send, the task transitions to the Send state. In

the Entry method, it sends the length (number of points) of the first ConnectedObject that needs to be sent to the DataReader via a data pack along with the depth of cut associated with the object. Note that this structure, along with the GUI design, constrains input designs to 2.5 axes of motion. In the Action method, the UITask sends one x point and one y point via data packs on each scan of the task. The UITask task then transitions back to the Wait state once it has sent all the points in the ConnectedObject. So, it only sends one ConnectedObject per state transition.

The "Clear" button clears all contours that have been drawn, but have not been sent to the mill. Finally, the "Exit" and "E-Stop" buttons are self-explanatory.

Mill Control

As previously mentioned, the SpectraLight mills used for this project consists of three stepper motors, one each for the x, y, and z axes, and a servo motor driving the spindle. The mill is controlled by sending an eight bit number to the black box controller from the computer's parallel port. Each bit corresponds to a particular command - global on/off, spindle on/off, and clock, or activation, and direction commands for the three axes (See Table 2.1 for details). The black box takes the command byte and actuates the proper component; the exact function of this box is hidden from the user, although it is most likely just an amplifier. The computer was connected to the mill Black Box via the computer's parallel port. Classes from the Embedded ToolKit that comes with the PERC software were used to write a class that accesses the parallel port.

As can be seen from Figure 1.1, all of the tasks required to run the mill are wrapped into a container class called "Mill Group." Wrapping the classes like this allows multiple mill controls to be instantiated easily, if it is desired to run more than one mill at once. The Mill Group contains several pre-set values for the specific mill. These include: the output bit number that corresponds with each mill command (see Table 2.1), the depth of cut of the mill for a given material (.06"), the maximum depth (.4"), the desired stock clearance height (.06"), and the conversion value for the number of steps required for each input unit (80 steps per pixel). The specific values for our program are included in parenthesis.

The constructor of this class could easily be modified to accept any of these values from the GUI if multiple mill types were to be used.

The Mill Group also contains a vector of connected objects received from the GUI, and a vector of mill commands for the corresponding object. The data in these vectors are adjusted by the tasks contained within the Mill Group. The Mill Group also contains a method called "System Shut Down." This method is called from the Mill Data Reader Task when an "exit" command is received from the UI Task. System Shut Down simply causes the RTX process running the mill group to exit.

Command	Parallel Port Pin Number	Output Bit Number
X Clock Command	2	0
X Direction	3	1
Y Clock Command	4	2
Y Direction	5	3
Z Clock Command	6	4
Z Direction	7	5
Global on/off	8	6
Spindle on/off	9	7

Table 2.1: Output Bit Values and Pin Numbers for Mill Commands

The first task contained by the Mill Group that manipulates the data is the Mill Data Reader task. This task receives data from the UI Task. A new connected object is instantiated on the RTX side when the object size is received. The data is then received at a rate eight times faster than it is sent to ensure that no data is lost. To prevent the program from hanging in the case of lost data, if no data has been received for 5 seconds, the object is truncated to this new length, and the state returns to idle. The idle state also looks for the exit command from the UI Task, and calls the System Shut Down Method in Mill Group if it detects it. The state transition logic for the Mill Data Reader is shown in Figure 2.3.

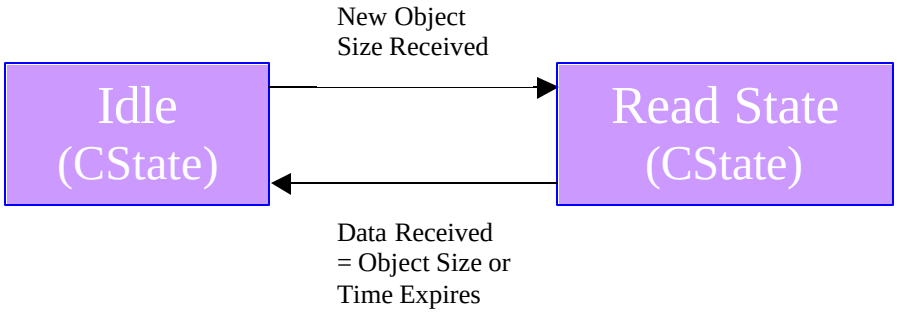

Figure 2.3: Mill Data Reader State Transition Logic

Paths that are deeper than the Mill Group specified maximum depth of cut are also handled in the Mill Data Reader. If a path is commanded that is deeper than the mill's depth of cut, the data reader will create separate objects of descending depths, incrementing by the depth of cut, until the desired depth is reached. While it would be more efficient to reuse the commands generated for the first object (which will simply be re-cut at a different depth), it was much simpler and more robust for us to handle deep connected objects in this way.

Once the contour data input by the user is reconstructed in a connected object on the RTX side, the Mill Path Planner task begins to generate the commands necessary to drive the mill to follow the path. There were three major challenges involved in the design of this task: setting the bits for the mill commands, generating the proper slopes, and the transition logic.

To set the bits of the command, a method called "setBit" was written (see below). This method takes a *number* to be modified, and a new *value* to set a specific *bit* to (either zero or one). It then treats the input number as a binary, and sets the given bit to the new value. If the value is zero, the bit is cleared by performing an AND operation with the number and a series of ones and a zero shifted to the proper bit. If the value is one, the bit is set to be a one by performing an OR operation with the number and a one shifted to the proper bit. This method is called eight times for each individual command number that is generated. The code is shown below.

```
Protected int setBit(int number, int value, int bit)
{
      if(value == 0)
      {
            number &= ~(1<<bit);
      }
      else if (value ==1)
      {
            number |= (1<<bit);
      }
      return number;
}
```

The bits for global on/off and spindle on/off were always set to one for every command. In retrospect, it may have slightly improved the speed of the program to simply initialize the command number to a decimal value of 192 (or 11000000 binary), which corresponds to a global and spindle on. This would have saved two calls to the setBit method.

Determining when to set each bit was done through a simple sloping algorithm of our own design. The steps required to move from the last position to the next position for each axis were calculated by taking the difference between the two positions and multiplying by the conversion factor of units to steps described in the Mill Group. The numbers of steps required to achieve the proper x and y-axis motions were then compared. The greater value determined the size of the required command array, and this clock (or actuation) bit was always set high. The lesser value was then divided by the greater value to generate the overall line slope to the next set point. A running count was then kept for each of the axes actuation commands. The current count corresponding to the sloping command axis was then divided by the current count of the always-on axis, and this value (corresponding to a instantaneous slope) was compared with the overall slope that was previously calculated. If the instantaneous slope was greater than the overall slope, the bit was set to zero (no stepping). If the instantaneous slope was less than the overall slope, the bit was set to one.

The maximum velocity for each axis was 1.25” per second (due to speed limitations of the computer. The maximum velocity of the system occurred while cutting a 45° angle, or both axes fully on. This corresponded to a value of $1.25*\sqrt{2}$, or 1.77” per second.

The transition logic for the Mill Path Planner state is shown in Figure 2.4. All segments consist of the same five operations: 1) Raise the tool to clear the stock, 2) Move to the first point, 3) Plunge the tool into stock, 4) Cut the desired contour, 5) Return the tool to the zero position. There can be some trickiness with the logic, particularly if the contour to be cut is just a single point (where the first and last point are the same). Specific checks had to be incorporated into the logic to handle such cases. Somewhat surprisingly, the most difficult object for our code to handle was a single point plunged to a deep depth. For this reason, this case was tested extensively before the code was declared complete.

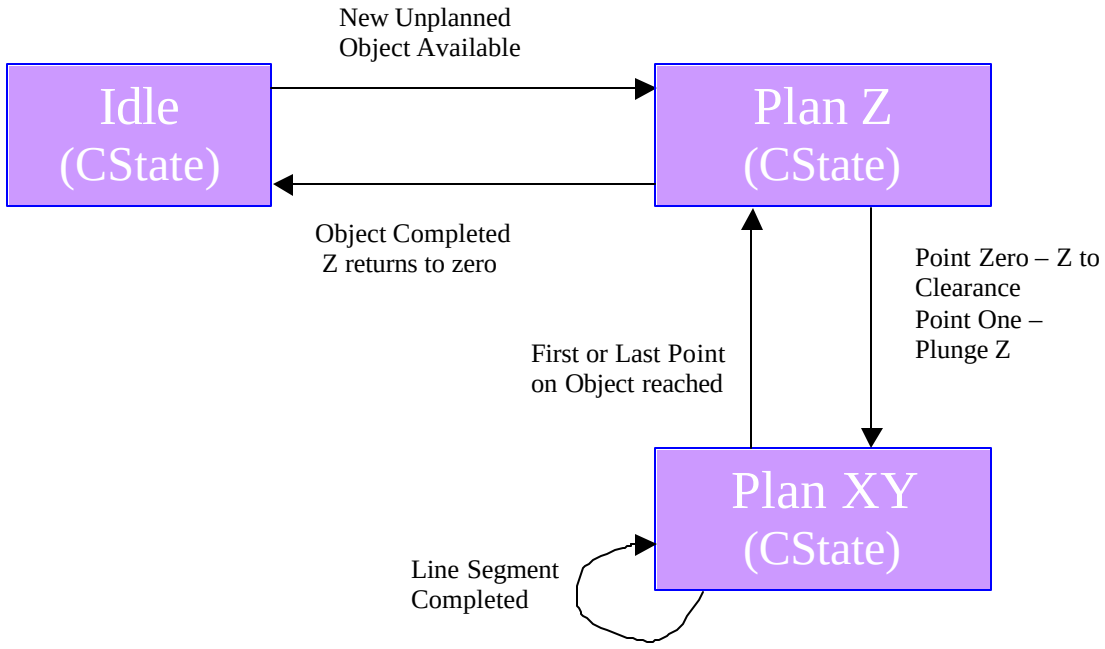

Figure 2.4: Mill Path Planner State Transition Logic

The final task enclosed in the Mill Group is the Mill Path Cutter State. This state is very simple, but is also the most time sensitive. This state simply spits out the eight bit command numbers generated in the path planner to the mill, and keeps track of position data while sending it to the data logger. The speed of this task limits the maximum velocity of the stepper motors in the mill, and therefore the maximum feed rate. For this reason, it is desirable to run this state as rapidly as possible. In our case, the minimum scan time was found to be three milliseconds. Since one on and one off bit must be sent

to the mill to complete one input signal (or square wave), this meant that we could at most send one full command every six milliseconds. To keep the stepper motors running smoothly, a consistent signal must be sent. This means that this task is also the most crucial from a latency standpoint, as any latencies translate into roughly running motors. Because of this, we took all of our latency measurements from this task.

To keep track of the x, y, and z position, we kept a running total of the clock commands multiplied by the direction of these commands for each axis, and then divided by the conversion factor of steps to input units. This data was then tracked by the data logger in process three, along with the duration and latency data. This process ran in Win32 which allowed it to write data (which can't be done in RTX), and allowed the computer to communicate with the GUI. Without a process running in Win32, the RTX process alone can't access the network card of the computer it is running on.

The cutter task also had a stop state in case an emergency stop was required. This state immediately sent out a zero value to the mill box, turning it off. This is really no substitute for a hardware emergency stop, but is the best we could implement in software. Once the restart command was sent from the GUI, the cutter state would begin where it had left off. The state transition logic for this task is shown in Figure 2.5.

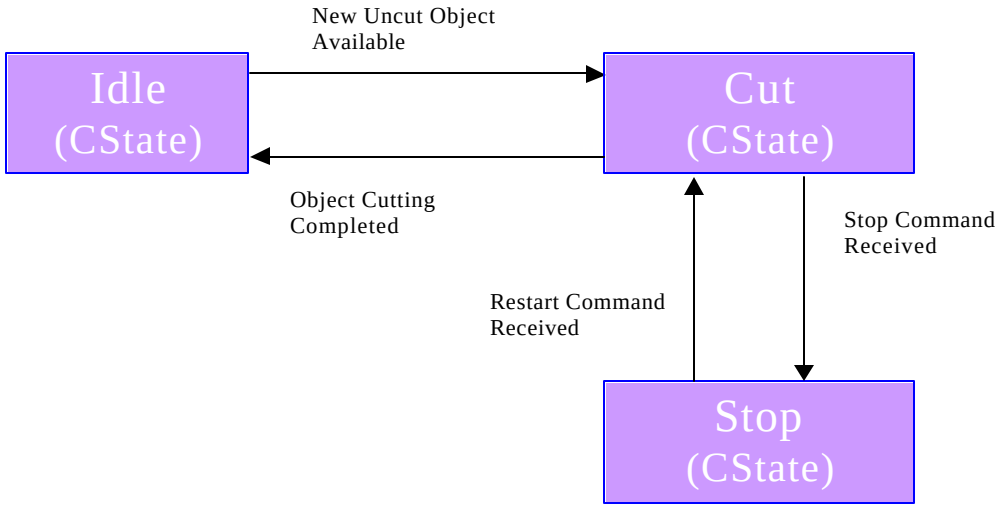

Figure 2.5: Mill Path Cutter State Transition Logic

Results

Test Case – "RTX Rules"

To test that our program was running properly, we cut stock material using a sequence that implemented all of our GUI tools (specifically: multiple depths, pencil tool, poly-line tool, and pocket tool). This test case was the phrase "RTX Rulez, " with a pocket, and two deep holes (the most difficult object for the path planner).

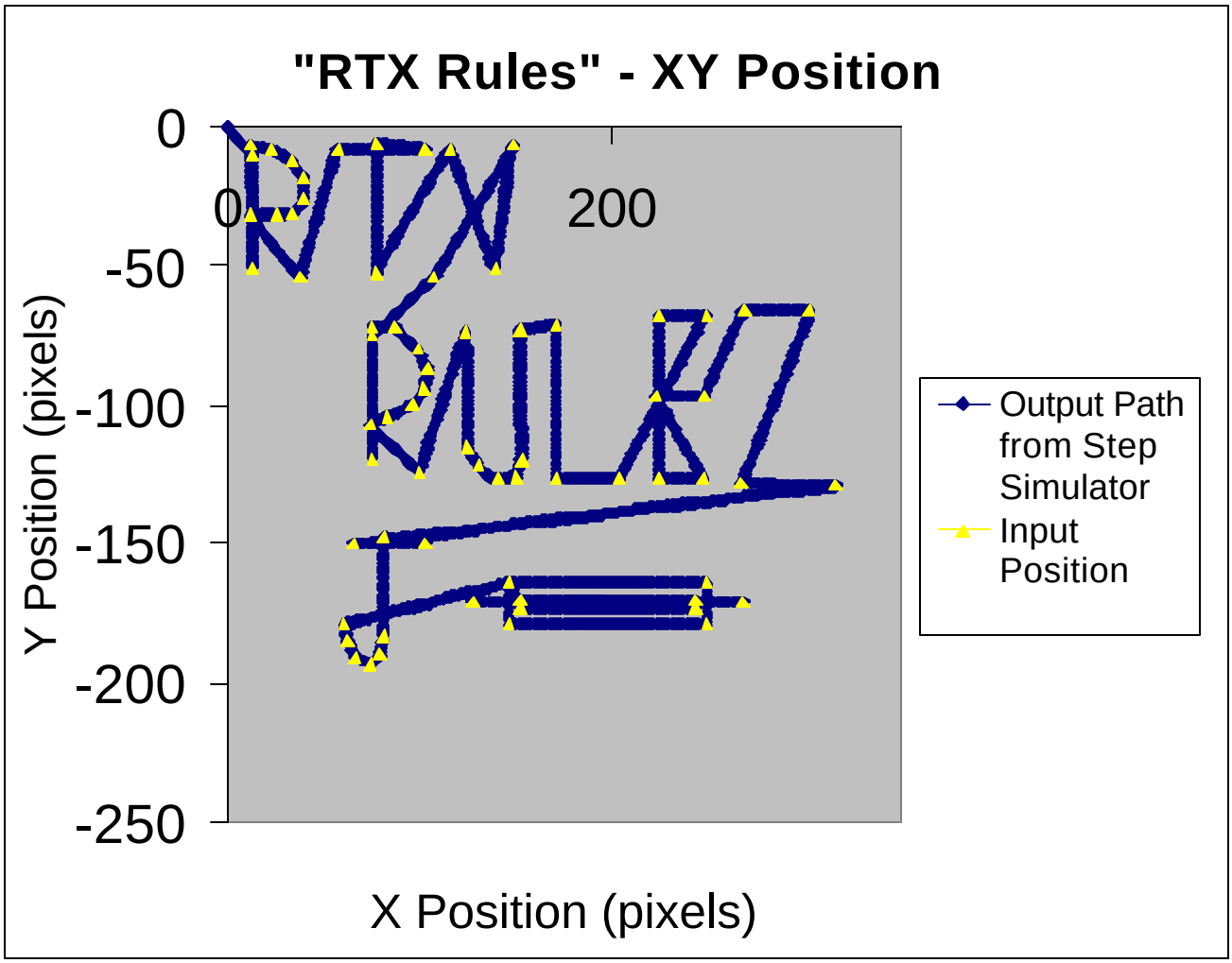

Figure 3.1: 2D Input Position and Resulting Cutter Output

Figure 3.1 shows the input values recorded by the UI Task and sent to the Mill Data Reader (in yellow), contrasted with the two-dimensional path plan generated by the Mill Path Planner (in blue). The original design input to this contour is shown in Figure 2.2. Clearly, the path cutter follows the input position precisely in two dimensions. Note that 100 pixels represent one inch.

Figure 3.2 is a three dimensional plot of the cutter path. A z position of zero represents the top of the stock, positive 6 (or .06") clearance for moving the tool to the next cutting point, and anything less than zero represents contours cut in the stock. The cutter motions of the tool obviously correspond to the desired output path.

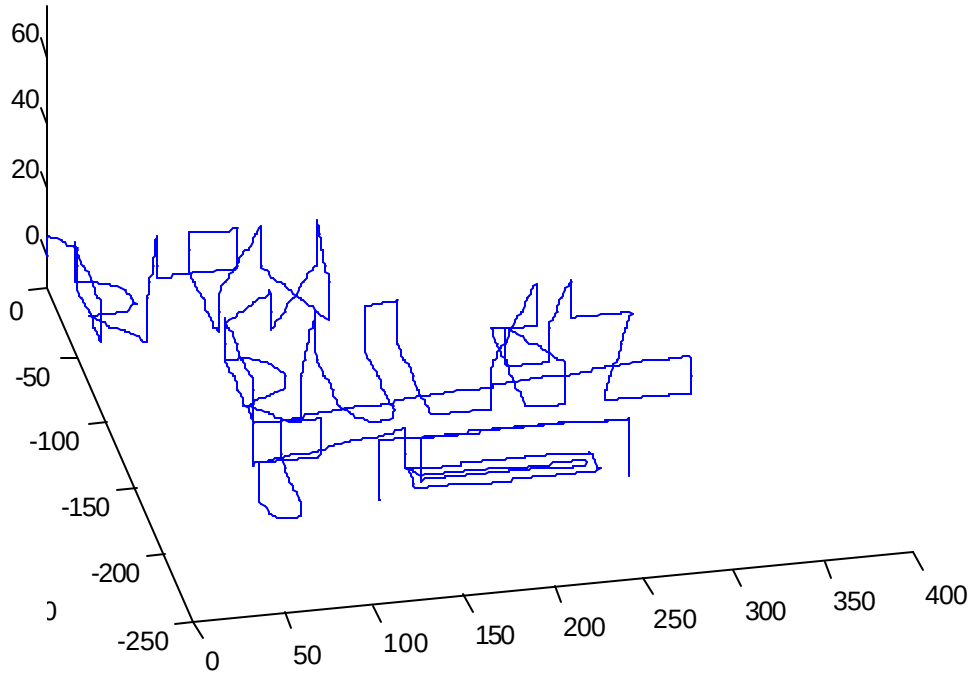

Figure 3.2: 3D Plot of Resulting Tool Path for "RTX Rulez"

Of course the final evaluation of any machine tool control software must be the parts that it is capable of generating. Figure 3.3 shows the resulting part from this operation. This part looks just like the original GUI input from Figure 2.2. The only variation is the line below the plunge hole to the left of the pocket. This was accidentally caused during the removal of the stock from the milling machine. Note the holes in the upper right and lower left were used to fixture the stock to the mill table.

Figure 3.3: Final Part Resulting From Test Operation of System

Timing Results

Latency and duration data were measured for the fastest and most time critical task, which is the Cutter task. This is fastest task because the feed rate of the mill is limited to twice the task interval of the Cutter task divided by the conversion factor (8000 for our case). It is the most time critical, because if latencies are large, then the feed rate of the mill is less that what is expected, and the motors will not run smoothly. Figure 3.4 shows the latencies of the Cutter task running at 10 millisecond, 5 millisecond, and 3 millisecond task intervals. The tests were taken under three conditions. The first set of tests were taken under standard conditions meaning that the design interface was running on the user's computer in a Windows process, and the mill control portion of the software was running on the mill's computer in an RTX process. The second set of tests was taken on two different computers, but with the mill control software running in a Windows process. The third set of tests was taken using a single computer with the mill control software running in an RTX process. It should be noted that the software was not able to run effectively with the Cutter task running faster than 3 milliseconds. With the Cutter task running at 2 milliseconds, the vast majority of the data sent from the user's computer

was lost in transit. However, at 3 milliseconds, all of the data got through as long as the mill control software was running in RTX. When using two computers with the mill control software running in a Windows process, some of the data was lost in transit at 3 milliseconds. The latency data for all the test cases is shown in Figure 3.4.

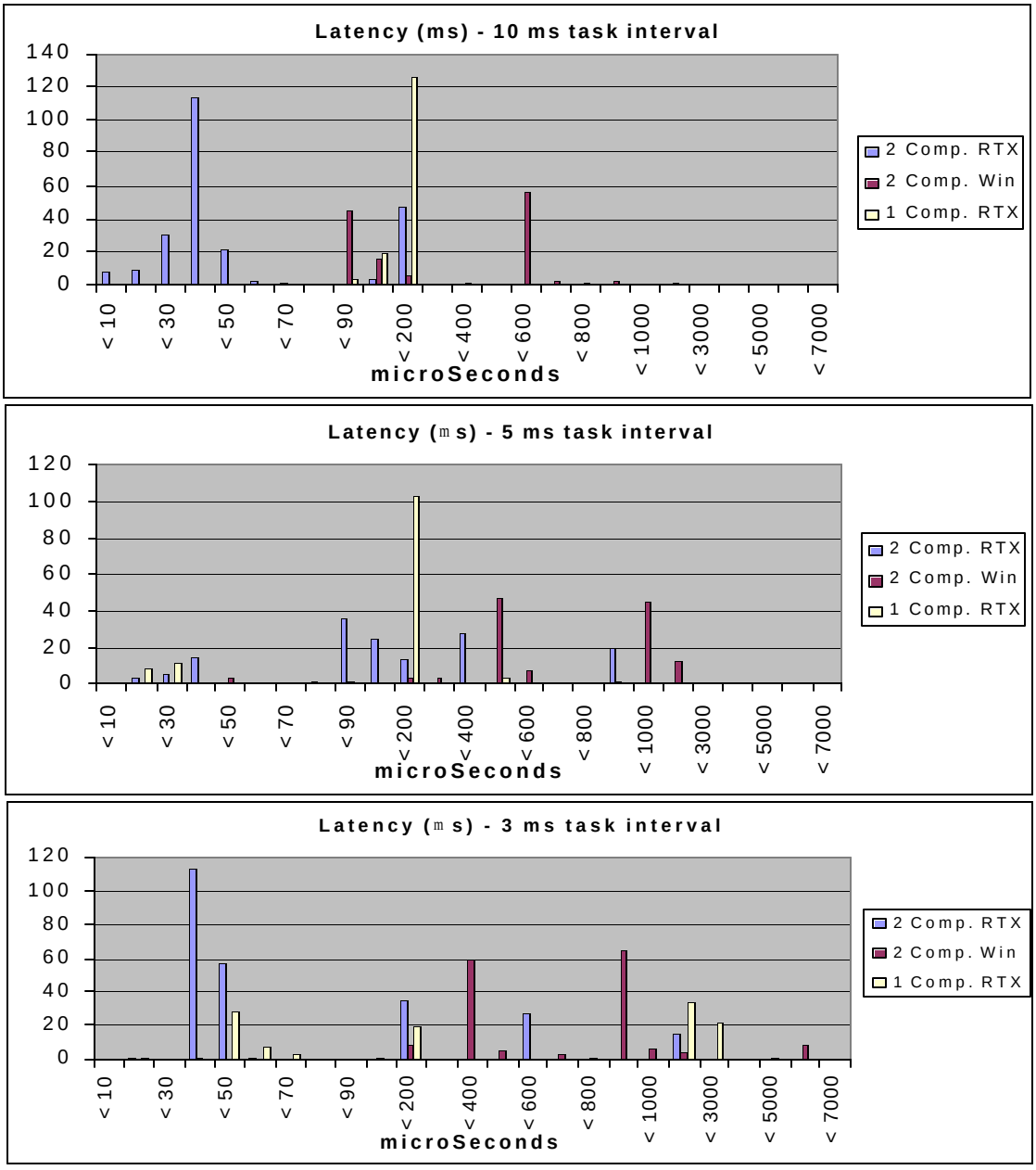

Figure 3.4: Latencies for Cutter Task at 3, 5, and 10 ms running under 3 different conditions: a- 2 computers using RTX, b – 2 computers using Windows, c – 1 computer using RXT

Figure 3.5 shows the Cutter task durations while running at a 3 millisecond task interval. Data was taken, but is not shown, while running at 5 and 10 milliseconds. There is very little variation in the task duration both across different task interval times and different operating conditions (one computer or two, RTX or Windows).

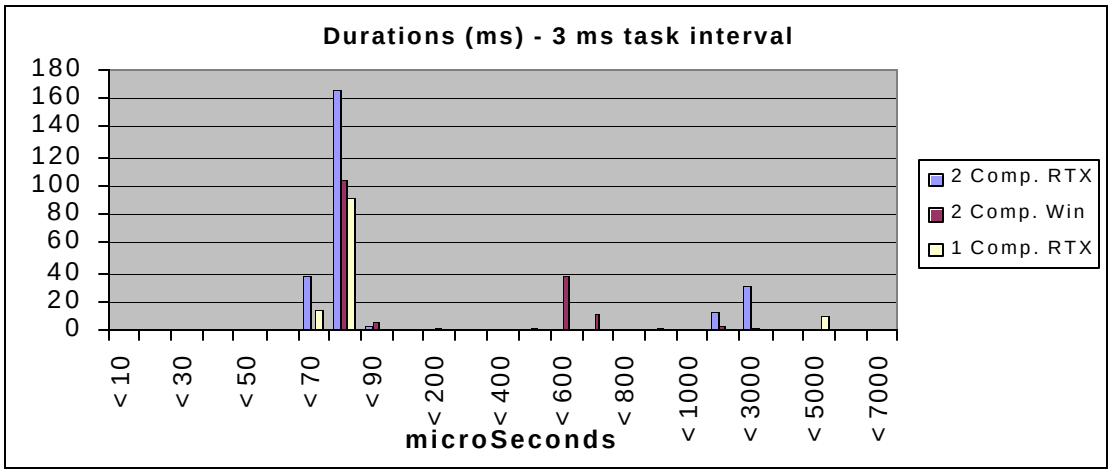

Figure 3.5: Durations for Cutter task running at 3 milliseconds under 3 different conditions: a – 2 computers using RTX, b- 2 computers using Windows, 3 – 1 computer using RTX

Discussion

Timing

The interval at which the Cutter task runs is of critical importance for this project because the feed rate of the mill is directly determined by the Cutter task interval, due to its use of stepper motors. Table 4.1 gives a brief comparison some of the important differences between stepper motors and servo motors in the context of this project.

Stepper Motors	Servo Motors
- Velocity determined by frequency of "square wave". - Max velocity limited by scheduler speed	- Velocity determined by input voltage. - Velocity is limited by voltage and friction
- Relative position always known (no feedback). - Feedback control is usually not necessary. - Limited Precision.	- Need feedback sensor to know position or velocity. - Feedback control crucial. - Infinite Precision.

Table 4.1: Stepper Motors vs. Servo Motors

As mentioned earlier, the fastest that TranRunJ can schedule the Cutter task is every 3 milliseconds. This means that the fastest square wave going into a stepper motor is 167 Hz., which corresponds to 1.25 inches per minute. If the mill is cutting at a 45 degree angle, and both motors are moving as fast as the scheduler will allow them to move, the maximum resultant feed rate is 1.77 inches per minute. It should be noted here that a simple program was written that runs one single RTX process and one task and does nothing but move one axis a predetermined distance. The task in this program was effectively run at a task interval of 0.5 milliseconds, which corresponds to 7.5 inches per minute. However, when multiple processes are run with added complexity of inter-task communication, 3 milliseconds is the fastest possible task interval that was accomplished using TranRunJ and RTX. When the Cutter task interval was dropped to 2 milliseconds, very little of the data got passed from the design interface to the DataReader. It is believed that this occurs because the minimum sleep time for an RTX process is 2 milliseconds. Therefore, when a task is trying to run at 2 milliseconds, there is very little or no time available for the scheduler to check the network sockets for new data packs.

The latencies of the Cutter task are also critical because, in the absence of feedback sensors, the error in the feed rate depends directly on the Cutter task latencies. Latencies also affect how smoothly the stepper motors run (specifically how much they vibrate during operation). Referring to Figure 3.4, the average Cutter task latency at an interval of 3 milliseconds while running under RTX is about 0.2 milliseconds, which corresponds to 6% error in feed rate. In other words, the actual feed rate is 6% slower on average than the controller thinks. And, because there are no feedback sensors there is no way to directly compensate for this error. It should be noted that this velocity error does not translate to position errors because of the nature of stepper motors. It may be possible to measure the latencies in the control software, and then compensate for them by speeding up the rate at which "on" pulses are being sent to the stepper motors. However, this would mean that the steppers would need to operate slower than the maximum possible velocity on average so that they could be speeded up to compensate for latencies. It was decided not to pursue this compensation because the maximum feed rate is so slow that one would always want to run the mill as fast as possible.

It is interesting to compare the average and maximum latencies under different operating conditions (See Figure 3.4). The Cutter task has much lower average latencies running in RTX than in Windows. This, of course, is expected. It is, however, interesting to note that running in Windows the latencies are less sensitive to the task interval. There is a dramatic difference in average latencies from 3 to 10 milliseconds in RTX, but in Windows this effect is dramatically reduced. It is logical to assume that the latencies in Windows are more a function of other Windows processes running, and less a function of the load on the TranRunJ scheduler. The maximum latencies are also considerably lower under RTX than Windows, however this affect is more noticeable at 10 milliseconds than at 3 milliseconds.

The other comparison shown in Figure 3.4 is between running the design interface on a separate computer and running all processes on one computer. The average latencies were larger using only one computer. Also, the average latencies using only one computer seem to be less sensitive to the task interval. RTX runs underneath Windows and gives a higher priority to RTX processes. So, it would be expected that the refresh rate of a Graphical User Interface running in Windows on the same computer as an RTX process would be quite slow. However, this was not the case. In fact it was the Cutter task latencies that suffered while running both processes on a single computer, not the refresh rate of the Graphical User Interface. The reason for this is not quite understood.

Networking

During the completion of this project, several difficulties were experienced with the networking aspect of the TranRunJ software running in real-time. While this caused us great difficulty and significant time loss, it did serve to help the author of that software (who is probably reading this paper) to make the scheduler more robust. Specific problems with TranRunJ included: some data packs were received multiple times when they had only been sent once, global box data was not transferring across processes, and data packs sent across processes were being lost at a very high rate. See the documentation of TranRunJ for definitions of these terms.

While the transfer of data across processes wasn't crucial from a control perspective, since all of the control occurred in the single RTX process, it was crucial for the proper operation of our software. In other words, data loss impacted the overall function of the project as the design intent of the user was lost with missing data packs. Global data loss prevented the processes from communicating properly. This often caused the software to hang as one process waited for the other to send data that never arrived. Particularly troublesome was the global data associated with the Mill Data Reader that signaled the UI Task that it was ready to receive more data. We verified that the problem was caused by the scheduler when we read the global data from two different processes and found that the same global box had a different value for each process. This problem was corrected quickly, but the nature of the problem remains unclear.

Our second problem was that data packs were being received multiple times when they were only sent once. As a result, three or four connected objects were often received to be planned when only one had been drawn. The number of data points in each of these connected objects descended with each receipt until some of the objects didn't have any data all. This was a puzzling problem as each data box is supposed to be flushed as soon as it is read. It turned out that the data was being immediately flushed on the receiver side, but wasn't being removed quickly enough on the sender side. As a result, when the receiving process would check to see if a new pack had been sent, it would interpret the old data that was still in the sender's memory as a new data pack. This problem was also corrected reasonably quickly.

The final, and most difficult, problem that was encountered was the loss of data packs being sent from process to process. Examination of the network revealed that all network data packets were being transferred properly from computer to computer, meaning that the problem existed in the scheduler. This problem turned out to be two fold. First, the global data boxes and the data in boxes shared the same socket for data transfer. This created a bottleneck that prevented data from getting through. To remedy this, two sockets were created for the program (by the author of the TranRunJ software): one each

for global data and data pack boxes. Also, the timing of the data sending/receiving was changed to make it more robust. Specifically, global data, which is typically much less time sensitive, was checked much less frequently to allow the data boxes to be checked more frequently.

Since the receipt of all data points (to preserve the user's design intent) was more important for our project than the speed of the data packs (since no control was taking place over the network) a TCP/IP protocol would have been preferable to the UDP protocol used by TranRunJ. This certainly would have mitigated some of the troubles described above. Ultimately, we found that scanning the receiver's data box at a rate of eight times faster than it was being sent was sufficient to ensure that almost all of the data was properly received once the scheduler software bugs were fixed. This was a crucial aspect to the proper function of our software across a network.

Conclusions

Timing is key in the control of stepper motors. The rate at which commands are sent limits the speed of the motor, and the consistency of the command signal determines how smoothly the motor runs.

The real-time operating system RTX offers fast time increments and lower latencies, making it well suited for the control of stepper motors. For higher speed applications, extra hardware must be designed and built.

Debugging in RTX mode is difficult as the Integrated Development Environment (IDE) debugger is unavailable, and RTX can't output data to a file. For this reason, code should be extensively tested and its use simulated before it is ported to RTX.

Reliability of the network is a large factor in the control of real-time systems. Even if your code is perfect, networking delays or lost data can destroy the function of your product.