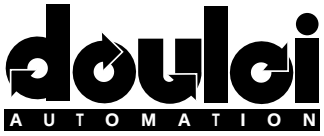
AN ADVANCED MOTION CONTROL SYSTEM ARCHITECTURE BASED ON A 386 PC

J. Randolph Andrews
Douloi Automation
740 Camden Avenue Suite A1
Campbell, CA 95008-4102
(408) 374-6322



Paper presented at the
1992 Incremental Motion
Control Systems and
Devices Symposium

Introduction

The equations behind digital motion control are continually finding better calculation vehicles as microprocessors and DSP's advance in performance and cost effectiveness. However solving a motion control application involves much more than timely execution of servo loops. There are many system level issues to be addressed. The system's objective needs to be expressed in at least one and perhaps several motion application programs. Information from non-quadrature as well as quadrature sensors needs to be interpreted and used by the motion controller. Other controlled devices need to be coordinated with respect to motion events. The operator needs to communicate to the system to request and alter system actions.

Many motion controller manufacturers respond to system level needs by making motion controllers more like computers. These motion controllers have the ability to run application programs, have IO buses that communicate to external hardware, and have serial ports that allow connection to operator interface terminals.

An alternative approach to making a motion controller more like a computer is to make a computer more like a motion controller. This paper describes an architecture based on a 386 PC which takes this second approach.

The following sections include an example problem, architecture description, problem solution, discussion of architecture contributions towards the solution, additional architecture benefits and a summary.

Example Problem

To provide a context for studying motion controller attributes consider the following Ferrari and Police Car pursuit problem:

1) You are driving a bright red Ferrari on Interstate 80 speeding through Nevada at 100 m.p.h..

2) A Police car hiding behind a billboard watches you flash by and begins pursuit.

3) The police car matches the speed of the Ferrari and follows maintaining a fixed distance while the Ferrari continues travelling at different speeds, occasionally slowing down and speeding up. The police car follows for a mile to document the Ferrari's speed violation.

4) After a mile of pursuit the police car starts flashing its lights and "catches up" to the Ferrari so that as well as travelling at the same speed both vehicles have the same position less the length of a car.

5) After catching up with the Ferrari, both cars slow to a stop.

Implementation Context

The illustration is to be represented with the following equipment:

• Two servo motors, one representing the Ferrari and the other representing the Police Car

• Lights which are switched on and off through industrial output modules controlled by a third party IBM PC IO board. These lights represent the police car's flashing lights.

Figure 1 shows example Ferrari and Police car velocity profiles if the Ferrari drives at constant speed.

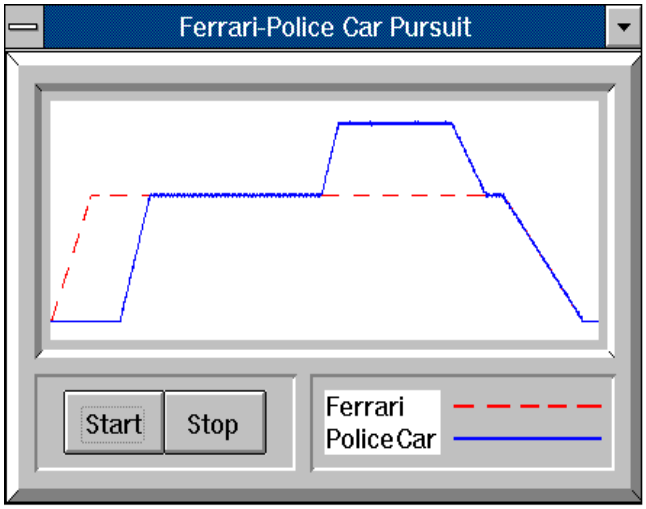


Figure 1. Constant Speed Ferrari

The Ferrari velocity is represented by the dotted line and the Police Car velocity by the solid line. The police car accelerates after the Ferrari and matches speed. After a period of time the solid line rises indicating that the police car is overtaking the Ferrari. After the police car catches up both cars travel at the same speed again and stop together. Acceleration may be different from deceleration. Acceleration and deceleration values for the Ferrari, police car, and "catch up" section may be independent of each other.

Figure 2 shows the velocity profiles when the Ferrari speed varies. Note that as well as tracking the Ferrari velocity during the initial pursuit period the police car is able to track the velocity while overtaking the Ferrari.

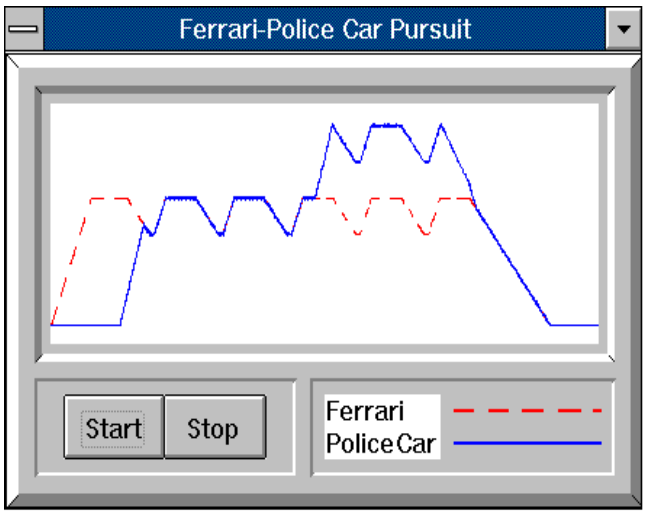


Figure 2. Variable Speed Ferrari

Motion Control Requirements of Problem

This example requires the following motion control system capabilities:

• Variable velocity profiles (Ferrari's joy ride)

• Ability to accelerate to a time varying speed and match it (police car pursuit)

• Position tracking or "electronic gearing" (police car "locking" onto the same speed as the Ferrari and maintaining a fixed distance from the Ferrari while the Ferrari continues to joy ride at various speeds)

• Trapezoidal phase advance on electronic gearing (police car catches up while continuing to track)

• Ability to motion mode "splice" from velocity mode to electronic gearing mode to stop while in motion

• Ability to coordinate IO with motion events (flashing lights when police car closes in)

• Ability to communicate to third party hardware (Industrial IO attached to PC add on board)

In an actual industrial application these capabili-

ties might solve a robotic assembly problem where multiple members need to “intercept” at a particular point to accomplish a part insertion.

Architecture Description

Figure 3 shows an overview of the 386 based motion control system architecture.

The system is composed of the following compo-

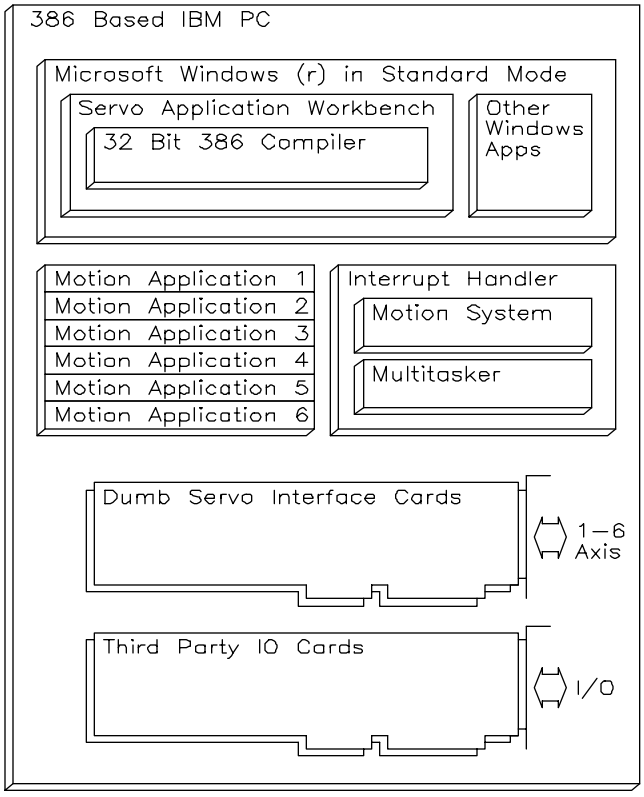


Figure 3. Architecture Overview

nents:

- 386 based PC as hardware platform
- “Dumb” Servo Interface Card
- Motion System/Multitasking Interrupt Handler
- Multiple motion application programs
- Microsoft Windows in standard mode
- Servo Application Workbench with Compiler

These elements are now described in detail.

386 based PC as Hardware Platform

The system architecture is based on a 386 PC with 64K cache and 32 bit wide data paths. This has become a commodity machine. A typical “base” system includes a 386 processor, 64K cache memory, 1 Megabyte of dynamic memory, floppy disk drive, floppy and hard disk controller, parallel and serial IO ports, keyboard, power supply, and enclosure. Street prices for such a base system are about $500 in March ’92 and continue to decrease. To make the computer useful requires additional memory, hard disk, VGA monitor, VGA graphics card, DOS software, Microsoft Windows, and Mouse. These extra components might cost about $800 for a computer system cost of about $1300.

“Dumb” Servo Interface Card

Installed in the IBM PC's IO expansion bus is a “dumb” servo interface card. This card contains:

- 8 MHz quadrature inputs with 3 bit filters
- High speed position capture
- 12 bit motor command outputs
- Uncommitted polling style inputs
- Uncommitted capture inputs
- Uncommitted outputs
- Watchdog safety system

This card does not contain a microprocessor and the associated ram, rom, communication chips, and device selection. These are not needed since the 386 itself will serve as the motion control processor.

Interrupt Handler

The 386 provides motion control functions by responding to a timer interrupt which occurs at 1 kHz. This interrupt handling routine performs three major functions.

The first function is servo control law execution. The current design controls from 1 to 6 axis of motion with the familiar zero, pole, integrator filter used in many motion control systems. This operates at a 1 kHz sample rate providing comfortable closed loop system

frequencies of 100 Hz and below.

The second function is motion profiling. The system is able to profile motion for 6 physical axis and 3 additional "virtual" axis. These 9 axis can be combined in different arrangements to form various multi-axis coordinated "machines". Any particular machine can perform coordinated motion along an arbitrary path. Multiple machines, each running multiple axis, can perform motion concurrently and independently. The motion profiler uses a dynamic profiling technique which permits changing profile parameters on the fly including acceleration, deceleration, slew speed, destination and in some cases motion type. This permits motion mode "splicing" without stopping. For example a positioning move can be changed to a jog at a new speed on the fly.

The third function is multitasking. Multiple motion application programs are resident in the computer. The interrupt handler contains a multitasker which activates and manages the operation of these programs.

Multiple Motion Application Programs

As many as 6 separate motion application programs (which are distinct from motion profiles) can be running concurrently and independently at any particular time. These programs can communicate to each other through shared memory and the file system. They can also access the motion control system, communicate to IO boards in the PC IO expansion bus and communicate with Windows applications created by the Servo Application Workbench.

Microsoft Windows in Standard Mode

When the computer is not responding to the motion system interrupt it is running Microsoft Windows in standard mode. "Standard mode" allows the 386 to access up to 16 megabytes of memory for Windows use. Windows provides the operating environment for motion application development and operation.

Servo Application Workbench

The Servo Application Workbench (SAW) is a Windows application which greatly simplifies the creation of multitasking motion application programs and operator control elements to direct them. Figures 1 and 2 are SAW applications that perform the pursuit example. Actual motion time history is collected and displayed as part of their operation.

Inside the Servo Application Workbench is a high level language compiler. The compiler changes the descriptions of the motion applications into native 386 32 bit object code which executes very quickly. The compiler "knows" about the motion system, the multitasking system, and Windows. This permits the application developer to access different system resources in a consistent way without having to worry about how these resources are being provided.

The Servo Application Workbench allows the developer to construct motion applications in a "clip art" fashion by pasting pre-fabricated parts and assemblies into the application. After "screen painting" the application and filling in the program's behavior the Servo Application Workbench compiles the motion application programs and creates the associated Windows application to operate them.

Problem Solution

The following code shows one possible solution to the Ferrari and Police Car pursuit problem:

```
var   Ferrari:T1Axis;
var   PoliceCar:T1Axis;
var   FollowingDistance:T1Axis;
const aMile=5280000;
const LengthOfCar=12000;
```

```
{-----------------------------------
-----}
procedure PursueFerrari;
{-----------------------------------
-----}

  begin
  Ferrari.Init(XAxis);
  PoliceCar.Init(YAxis);
  FollowingDistance.Init(RAxis);
  FollowingDistance.SetAccel(0.04);
  FollowingDistance.SetDecel(0.02);
  Ferrari.ServoOn;
  PoliceCar.ServoOn;
  BeginTask(Addr(FerrariJoyRide));
  Delay(5000);
  PoliceCar.SetAccel(0.04);
  PoliceCar.SetDecel(0.04);

  repeat
   PoliceCar.Jog(Ferrari.ProfileVelocity);
    Yield;
  until
  abs(PoliceCar.ProfileVelocity
    -Ferrari.ProfileVelocity) < 0.1;

  FollowingDistance.SetCommandedPosition(
    Ferrari.CommandedPosition
    -PoliceCar.CommandedPosition);

  ScheduleTask(Addr(PoliceCarTrackFerrari,
    InvokeEverySample));

  PoliceCar.WaitForDistanceChangeOf(aMile);
  ScheduleTask(
     Addr(FlashPoliceCarLights,1000));

  FollowingDistance.MoveTo(LengthOfCar);
  Delay(500);
  AbortTask(Addr(FerrariJoyRide));
  Ferrari.Stop;
  end;
```

The following routine is started by PursueFerrari as an autonomous task:

```
{-----------------------------------
-----}
procedure PoliceCarTrackFerrari;
{-----------------------------------
-----}

  begin
  PoliceCar.SetCommandedPosition(
     Ferrari.CommandedPosition-
   FollowingDistance.CommandedPosition);
  end;
```

The following routine performs IO control through the third party hardware:

```
{-----------------------------------
-----}
procedure FlashPoliceCarLights;
{-----------------------------------
-----}

 const IOBoardAddress=1000;
 var  TurnLightsOn:boolean; static;

  begin
  TurnLightsOn := not TurnLightsOn;
  if TurnLightsOn then
    PortWriteByte(IOBoardAddress,1);
  else
    PortWriteByte(IOBoardAddress,0);
  end;
```

This routine directs the Ferrari's joy ride:

```
{-----------------------------------
-----}
procedure FerrariJoyRide;
{-----------------------------------
-----}

  begin
  Ferrari.SetAccel(0.03);
  Ferrari.SetDecel(0.015);
  Ferrari.Jog(35);
  Ferrari.Delay(1000);
  while true do
   begin
   Ferrari.Jog(35);
   delay(1200);
   Ferrari.Jog(25);
   delay(800);
   end;
  end;
```

Explanation of Operation

Variables are declared to represent the different

objects in the problem. A Ferrari and Police car are each declared to be a one dimensional axis using one of the pre-defined object types of the language. Other machine dimensions are available, such as a T2Axis for a 2 axis machine or a T6Axis for a six axis machine. The following distance is also defined to be a one dimensional axis. The constant "aMile" represents the distance the police care will track the Ferrari. The constant "LengthOfCar" represents how close the police car will close in on the Ferrari.

Procedure PursueFerrari begins by initializing these variables. The Ferrari is associated with the XAxis, the PoliceCar is associated with the YAxis and the FollowingDistance is associated with the RAxis, one of the three available “virtual” axis which behaves like a normal axis but has no associated motor.

The Ferrari is started on its joy ride with the next line, and is given a five second head start on the police car with a Delay instruction.

The repeat loop tells the PoliceCar to match the Ferrari’s speed. Yield is an instruction which tells the task to “give up” execution to the rest of the system because the developer realizes that nothing interesting will happen until the next sample. When the speeds of the two cars are close execution continues with the next instruction after "until".

Now that the PoliceCar has matched the Ferrari’s speed the FollowingDistance is set to be the distance between the two.

At this point it is time for the police car to “track” the Ferrari’s speed. This is done by a separate task named “PoliceCarTrackFerrari” and contains one statement. The commanded position of the police car is set to be the position of the Ferrari minus the position of the Following Distance. Until the FollowingDistance virtual axis is asked to do something its commanded position will be the value that was last set, the initial following distance of the police car behind the Ferrari. To continuously maintain the commanded position of the PoliceCar we schedule this task to run every controller sample period. What this scheduled task has provided is a new sample-rate frequency criteria for the police car commanded position, effectively a custom profile algorithm.

After waiting for a mile of distance to go by the police car starts flashing its lights. The flashing of the lights is managed by another task and continues on its own at a different frequency. The utility of a virtual axis is now seen. By asking the FollowingDistance to MoveTo(LengthOfCar) the FollowingDistance commanded position changes from its initial positive value to LengthOfCar with the well behaved accel and decel of a trapezoidal move and the police car catches up with the Ferrari. A trapezoidal profile has been superimposed on top of electronic gearing criteria to achieve trapezoidal phase advance with only a few lines of code.

After the police car catches up to the Ferrari the Ferrari is asked to stop its joy ride and the fun is over.

The procedure to flash the lights simply toggles the third party industrial control module by assigning a new value to the card through the IBM PC IO port.

The procedure FerrariJoyRide is an endless loop of speeding up and slowing down. It would have continued indefinitely if it had not been aborted.

Architecture Contributions

What architecture attributes contributed towards solving this problem? The main attributes include:

- Descriptive domain terms language
- High speed application programs
- Multitasking application programs
- Hardware extensibility

These attributes are now discussed in detail.

Descriptive Domain Terms Language

Notice that the problem solution is expressed in the terminology of the problem. The problem is about Ferraris and Police cars and flashing lights. These different objects appear in the expression of the solution. The freedom to be expressive in both words and structure is an attribute enabled by the compiler. Many microprocessor based motion controllers which execute motion programs are based on interpreters. Interpreters must process the symbols of the program every time a program statement is executed. Descriptive language and expressive program structure are counterproductive to an interpreter since longer symbols require more processing time. A compiler incurs the cost of interpreting verbose symbols once, at compile time, and suffers no additional run time speed degradation.

The language system supports Pascal like variables, user defined record types, user defined object types (record structures with related manipulation routines), symbol scope, subroutines and functions with parameters, and type checking. The language system includes try-recover structured exception handling providing a powerful technique for managing errors that occur during application operation. The language also includes a library of predefined objects such as multi-axis machines, vectors of various dimensions, and controls that communicate with Windows for plotting and user interface interaction. Even if the developer chooses not to use these advanced language features directly many of the benefits are provided by the Servo Application Workbench automatically or in some cases through dialogs and fill-in-the-form prompts.

High speed Application Programs

This solution used 4 independent tasks, one executing its program body every millisecond. Being able to write application programs that operate at the controller sample rate is a very powerful developer capability. Previously achieving customized operations at the sample rate required specialized firmware in a microprocessor based controller with associated engineering costs and lead times. The ability of the 386 PC architecture to provide high speed operation is attributable to two things: resultant native 386 object code for the application programs and a 33 MHz 32 bit cached computer with a 32 bit hardware bus.

Conventional microprocessor based motion controllers with interpreters for program execution may take as much as a millisecond to execute a single statement because of command interpretation time. Using the 386 PC based architecture it is possible to write application programs which acquire information from multiple axis and sensors, process and combine that information, and create new control directives for multiple axis and other controlled devices and have that entire program execute 1000 times every second. The solution to the pursuit problem used an electronic gearing with trapezoidal phase advance technique however the system does not have an “electronic gearing with trapezoidal phase advance mode”. This capability was synthesized as an application program. The capabilities of this architecture are not limited by intrinsic “hard coded” controller capabilities. If a new capability is needed the developer can write it and achieve similar performance to firmware based solutions.

Multitasking Application Programs

High program execution speed would not be useful if the 386 “had its hands tied” during the execution of such a program. A system which executes quickly but fails to maintain a communication relationship with the operator is, from the operator’s point of view, not running at all. The 386 PC based architecture provides a multitasking manager which schedules and executes multiple tasks running at possibly different frequencies. While application programs are executing at 1 kHz invocation frequencies the operator is free to move about Windows, graphing information about motion behavior, clicking buttons that alter or redirect application behavior, providing parameters in preparation of starting a new

activity, and even running other Windows applications such as a spreadsheet.

Once a task has been scheduled it generally operates "on its own", occasionally requesting Windows services which it cannot perform by itself. This leaves the developer free to create other tasks for providing other functions without having the machine tied up performing the first task. Multitasking can simplify application development because it allows the developer to solve and "dismiss" an aspect of the problem from further concern. For example, once the police car "locked on" to the Ferrari it was not necessary to be concerned about that tracking relationship for the rest of the problem because it was already being managed. Without multitasking there is a much higher degree of coupling between different parts of the application because future sections of code need to constantly keep previous parts running while they are about the business of doing new and most likely unrelated activities.

Although Windows is called a "multitasking" operating environment Windows alone is not suitable for real time control. Windows is not preemptive meaning that once a Windows application starts executing other applications do not run until the first application releases control. The multitasker in the 386 based architecture adds preemptive scheduling to an otherwise non-preemptive multitasking environment. Without such a real time preemptive multitasking extension to Windows the 386 could easily be "distracted" from motion control activity and leave control matters unattended for durations of time ranging from milliseconds to minutes depending on what other Windows applications were doing.

Hardware Extensibility

The example solution used third party industrial IBM PC IO hardware. Many advanced microprocessor based motion controllers are able to communicate to external hardware through IO buses which are part of the controller. Unfortunately these buses typically have non-standard connectors and interface signals requiring that the developer build custom interface electronics.

The 386 PC based architecture has as its "external IO bus" the IBM PC's own IO Expansion Bus which has become a standard for data acquisition and control add-on boards. If someone makes a useful sensor most likely there is some type of IBM PC compatible board that can read that sensor. In the 386 PC based architecture the motion system is master of the IBM PC IO expansion bus rather than a slave as is the case for most microprocessor based control boards. Accordingly most any board can be accessed by real time motion application programs providing outstanding hardware extensibility.

Additional Architecture Benefits

Performance Extensibility

There are occasions when the system that has been built works, but wouldn't it be great if it had twice the throughput? Increasing system speed in motion control applications can be limited by motion controller attributes such as program execution speed rather than by the speed of the host. If a microprocessor based motion controller performs one instruction per millisecond in a 4.7 MHz PC it will perform one instruction per millisecond in a 33 MHz 486 because the speed of the motion controller is completely independent of the speed of the host. Some motion control manufacturers offer "speed up" options that allow developers to buy controllers which operate perhaps 50 to 100% faster than the nominal product and so help eliminate this bottleneck.

The 386 based motion control architecture, on the other hand, inherits the speed of the host because the motion controller is the host. If the project is moved from a 33 MHz 386 computer to a 33 MHz 486 computer more than twice the performance is achieved for a small incremental cost. The price performance ratio of this architecture improves with time simply because 386 PC's continue to have improving price performance.

Reduction of Communication Delays

Some systems which have the capability of recording real time information such as position time history must transmit that information to the host where it can be manipulated and displayed. This requires transmission time through the communication link connecting the controller to the host as well as possible program command interpretation time if the transmission is conducted by a motion controller application program.

When motion application tasks operating with the 386 PC based architecture need to communicate small amounts of information to control elements in the Servo Application Workbench they do so through a binary communication mailbox that SAW responds to. The motion application program "sends mail" asking for SAW to provide a service, such as displaying a particular number in a text control for example.

When it comes time to handle larger amounts of information such as position time histories, the 386 based architecture does not need to "transmit" information to the host. The motion application instead simply mails a reference to the information and SAW accesses the same data structure that the motion application filled. The management of mail and the associated data references are handled automatically by the compiler since it "knows" about these different elements and their intended relationships. The fact that this mechanism is allowing multiple motion application programs to communicate to a Windows operator interface applications is hidden from the developer.

Various Embodiments Available

The motion application may require operation in a particularly harsh environment, or require the use of a touchscreen rather than a mouse. There are many companies that provide 386 PC's in any form an industrial application may require. The amount of third party support directed to applying a computer to varied industrial and interface settings is very large for the IBM PC. This provides a solution for motion applications with challenging environmental requirements.

Dynamic Link Library Support

Some developers may elect not to use the Servo Application Workbench and to instead write Windows applications with their own preferred development environment. The motion system, multitasking system and compiler of this architecture can be accessed through a Windows Dynamic Link Library allowing functions to be used by a conventionally developed Windows application. Dynamic link libraries are a powerful feature of Microsoft Windows which greatly enhance cross-language interconnectivity. Windows is receiving a great deal of attention in terms of development tools, Dynamic Link Library support and enhancements. Language systems such as Visual Basic make creating Windows applications dramatically simpler than 18 months ago. This resource of tools benefits any developer working with Windows on the IBM PC.

Cost Effective

IBM PC based solutions are very cost effective because of the high volume commodity nature of hardware and software components for the PC. For the cost of a custom 4 line by 40 character LCD display that interfaces to a proprietary motion control IO bus it is possible to buy a VGA high resolution color monitor and display card. For the price of a microprocessor based motion controller speed upgrade option it is possible to buy a 386 base system. The cost effectiveness of IBM PC systems is remarkable and continues to improve.

Summary

Solving an advanced motion control application requires an advanced language system, multitasking, user interface support, high speed program execution and flexible motion capabilities. The benefit of this architecture is that these various diverse and necessary components have all been integrated together in a manner that gives the application developer simple access to

powerful capabilities for solving the entire system problem and not just the motion control part of the problem. Being based on commodity computing hardware and software components the architecture allows the creation of extensible, higher performance systems for less money than is possible through conventional microprocessor based controller architectures.

References

1) L. Fischer, R. LeBlanc, *Crafting A Compiler*, Benjamin Cummings Publishing Company, Inc., Menlo Park,California, 1988.

2) Intel Corporation, *i486 Microprocessor Programmer's Reference Manual*, Osborne McGraw-Hill Book Company, New York, 1990

3) E. Solari, *AT Bus Design*, AnnaBooks, San Diego, California, 1990

4) W. Brogan, *Modern Control Theory*, Prentice Hall, Englewood, New Jersey, 1991

5) G. Franklin, J. Powell, *Digital Control of Dynamic Systems*, Reading, Massachusetts, 1981

About the Author

J. Randolph Andrews received his B.S. M.E. in 1981, B.S. E.E. in 1981 and M.S.M.E. in 1983 from the Massachusetts Institute of Technology. He participated in the MIT Mechanical Engineering Department's DeFlorez Design Competition each undergraduate year winning 1st place '78, 1st place '79, 1st place '80 and 1st and 2nd place '81.

Andrews spent 4 years at Hewlett Packard's corporate research laboratory in the Applied Physics Research Center as well as the Manufacturing Research Center.

The following 4 year period was spent with Galil Motion Control.

In July '91 Andrews founded Douloi Automation to provide motion control hardware and software components primarily for use with Microsoft Windows.
Professional interests include motion control, software/electrical/mechanical system design trade-offs and high abstraction programming techniques and tools.