Real Time Data Acquisition and Control

by

Jonathan W. Valvano, Bapi Ahmad and Jagadish Nayak
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712

Abstract

This paper presents a laboratory environment for the development of real time data acquisition and control on the IBM-PC platform. The laboratory station involves the integration of low-cost computer technology with powerful software components which empower the student to efficiently and effectively construct real time systems. The software base integrates an editor, a spreadsheet, and a real time programming environment built around Druma FORTH. We have written multiple FORTH libraries to assist the student in the translation of engineering concept into creation. Real time events are managed using a rich set of FORTH software routines which guarantee that time-critical software is executed on schedule. The real time color-VGA graphic library includes many types of windows. We have developed an extendible debugging tool called PROSYM (PROfiler and SYMbolic debugger.) PROSYM provides a simple set of primitives with a high expressive power that may be used singly or may be combined to construct customized debugging tools. In addition to providing basic debugging functions, PROSYM supports an event-action model of debugging. We have evaluated this development system on the full range of PC platforms from the original PC-XT to the newest 486 systems. The environment has been used for two years by Biomedical and Electrical Engineering graduate students performing both teaching and research projects.

Introduction

The purpose of any laboratory experience is allow the student to develop, apply and evaluate engineering concepts in a practical manner. A well-organized laboratory course can be an effective teaching experience, while a poorly-run lab will cause undue hardship on both the student and the faculty. We have attempted to assemble the hardware and software components for a graduate level class on real time data acquisition and control. The class combines both Electrical and Biomedical Engineering aspects as shown in Table 1.

Electrical Engineering	Biomedical Engineering
• microcomputer interfacing	• medical instrumentation
• real time data acquisition	• signal processing
• analog instrumentation	• transducer physics
• control systems	• patient safety
• quality programming	• effective human interfaces

Table 1. The objectives of the class involve the integration of EE and BME disciplines.

The key is to provide sufficient tools (with appropriate documentation) so that the student can quickly and effectively deal with the fundamental educational issues of the class (Table 1) without being overwhelmed with the complexities of the machine. On the other hand, we feel that complete isolation from the computer, as is the case with National Instruments Labview, inhibits the student from dealing with the Electrical Engineering aspects of the instrument. This problem is accentuated when the details of the hardware/software interface play a critical role in the engineering design decision. In the educational setting, it is particularly important for the student to have the power and control to manipulate the computer so that the trial and error experimental process of learning is

allowed to flow smoothly. In addition to control, there must be facilities for performance evaluation, so that the student can effectively compare and contrast alternative designs.

What is FORTH?

C.L. Moore created FORTH in 1972 as a programming language to control his telescope. Moore chose the name FORTH because he considered it to be a fourth generation programming language. Our FORTH is more than a programming language, it is an integrated software environment including an editor, an Intel 80x86 assembler, high level language compiler, interpreter, debugger, real time graphics, floating point, file system and real time operating system. The FORTH environment is efficient for the development of real time instrumentation and embedded-control systems. The FORTH interpreter, along with its simple structure, facilitate programming for the beginner as well as the experienced software engineer.

FORTH provides links to any editor on the PC. The editor is used to create and modify user programs. The FORTH compiler can then be invoked to produce new definitions with fast execution speeds. Even on a modest computer (12MHz 286), the edit/compile/download/run programming cycle is as fast as 20 seconds. The interpreter can be used to develop and test new functions. The interpreter allows for effective interaction between the student and the machine. The program development stage can be operated under DOS or Windows. In the real time execution phase however, the system runs only under DOS. This is because the scheduling of real time events can not be guaranteed under Windows. The real power of FORTH comes from its inherent extendibility. The user is provided with an initial set of language elements, and the ability to add new elements to the working set. The true beauty of FORTH programming lies in the fact that one has complete control over the computer hardware (e.g., I/O devices written in assembly language) but still has a rich and extensible set of high level language constructs.

FORTH provides links to any executable program on the PC (e.g., editors, spread sheets, networks.) Typically students transmit data into a spreadsheet (including those on the Macintosh) for analysis and report generation.

The version of FORTH that is used in our lab was developed at Druma Inc. of Austin Texas. One of the truly exiting aspects of Druma FORTH is its debugger, which is described later. Table 2 gives a list of library routines available.

Library	Features
Fixed Point	arithmetic, number conversion, and input/output
Floating Point	trig, ln, exp, arrays, number conversion, and input/output
Data Acquisition	A/D input, D/A output, and digital input/output
Timer	16/32 bits, stopwatch features, and real time synchronization
Communications	Serial/parallel input/output
DSP	FFT, PID controllers, curve fitting, thermocouple standards
Simple Graphics	points, lines, rectangles, ellipses
Window Graphics	scroll, sweep, bar, pie, text, annunciator, logic, and meter graphs
File system	save/load multiple text/data files, virtual memory

Table 2. List of available libraries and their features.

Why FORTH?

One of the fundamental reasons for choosing FORTH as the programming base for this lab is that FORTH supports the concept of hierarchically layered programming. In other words, on the highest level the beginning student can piece together existing library routines, and create a powerful data acquisition system in a couple of hours (similar to what one can do with Labview.) On the other hand, the advanced student has access in a hierarchical manner to all lower level routines including assembly language. This access provides the student the knowledge of how the computer works and power to actually

integrate the fundamental theories into an effective data acquisition system. The dual nature of FORTH (editor/compiler for execution speed and interpreter for ease of debugging) is quite useful in the teaching environment.

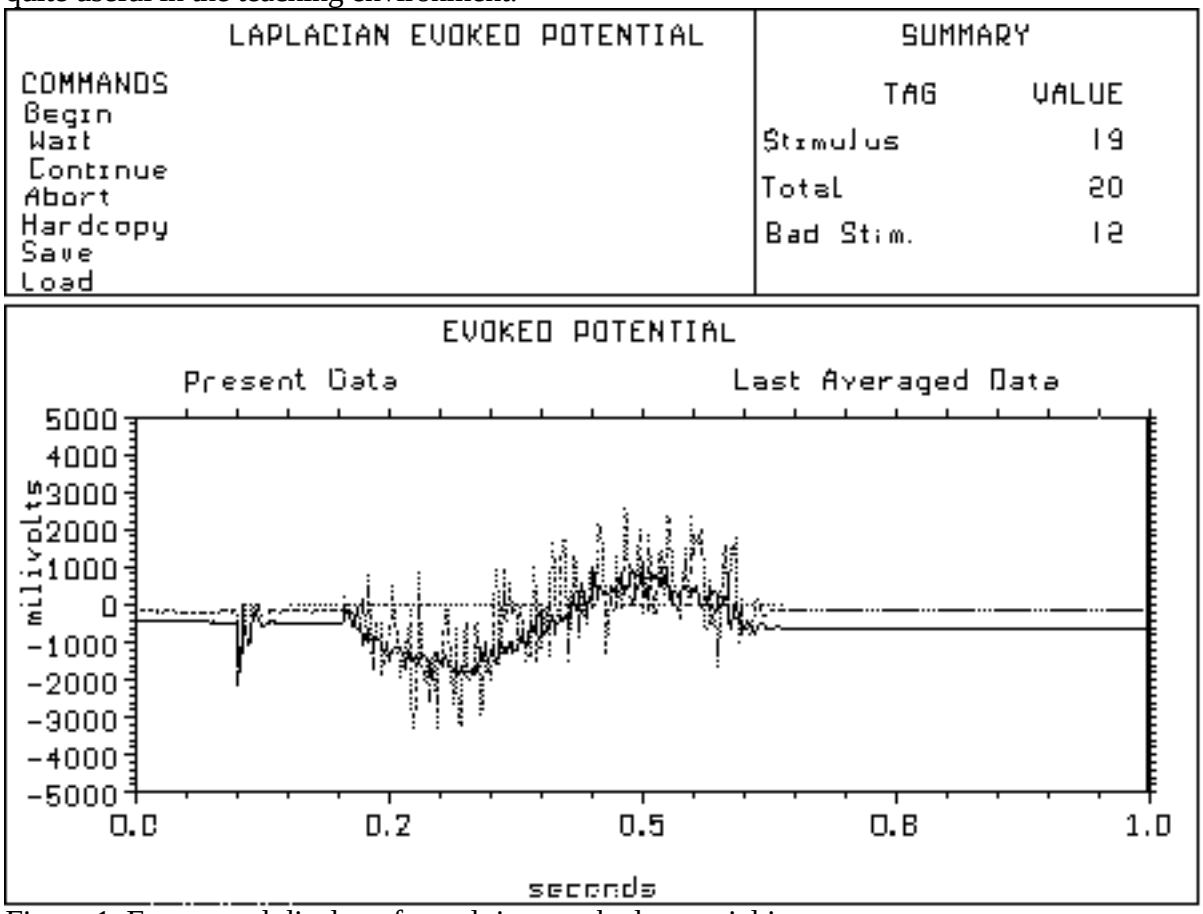

Figure 1. Front panel display of a real time evoked potential instrument.

Programming Style

Inherent in the development of software is the desire to create modular and structured procedures. Although assembly language is efficient both statically and dynamically, it suffers due to the difficulty in debugging, maintaining, and extending. FORTH is a structured language that facilitates modular programming. In FORTH, procedures are called **WORD**s. Each word should have a simple well-defined interface, and the body should be functionally complete with minimal side effects. Parameters are passed via the parameter stack or via global variables. The **DICTIONARY** is a linked list of all words. When a new word is defined, it is added to the dictionary hence extending the power of the system. We consider the one-pass nature of FORTH to be valuable asset, rather than a limitation. Since the compiler is one-pass, the programmer is forced to define words in a bottom-up hierarchical fashion. The FORTH assembler is also one-pass, substituting assembly language versions of the begin until and do loop structures. Without a GOTO the programmer naturally creates modular and structured programs. The programmer starts with about 1000 predefined system words.

Stacks

FORTH is a stack-based language. There are three stacks implemented in FORTH. The stacks are standard first in last out push down data structures. The **Parameter Stack** is the most visible stack. Most FORTH words use this stack for their inputs and outputs.

When a integer literal is executed, its value (16 bit) is pushed on the stack. When a integer constant is executed, its value (16 bit) is pushed on the stack. When a variable or array is executed its address (16 bit offset, with implied ES segment register) is pushed on the stack. It is implemented using the 80x86 SS:SP registers. To optimize for speed, the top of stack is stored in BX. The parameter stack is also used for **do loop** and **begin until** blocks. It contains the pointers and index values to implement nested loops.

The **Return Stack** contains return addresses when one FORTH word calls another. It is implemented using the 80x86 SS:BP registers. The execution of FORTH involves the indirect subroutine call of a list words.

The computer has an Intel 80x87 math coprocessor. External to the 80x87, floating point numbers are 32 bits or 64 bits. Internal to the 80x87 all numbers are 80 bit temporary reals. The 80x87 has an eight level hardware **Floating Point Stack**. The FORTH system uses this hardware stack for floating point parameters. When a floating point literal is executed, its value is pushed on the floating point stack. When a floating point variable or array is executed, its 16 bit address is pushed on the parameter stack.

Real time execution

Assembly language may be used for time critical portions of the program. The FORTH compiler is only one pass. Assembly language jumping is handled in a structured and disciplined manner, similar in syntax and function to the `begin end`, or `do loop` high level language structures. Assembly language labels and forward jumping are possible, although careless jumping or "spaghetti" code is strongly discouraged. Time is a critical parameter for most instruments. FORTH supports interrupt programming. The system includes a programmable real time clock. The following real time data acquisition program illustrates the programming style. Comments are *italicized*.

```
\ FORTH programs begin with the global variables.
\ Multiple access circular queue (MACQ)
variable x(n)     \ Current sample  Note: "x(n)" is its name
variable x(n-1)   \ Previous sample
variable x(n-2)   \ Previous sample
variable y(n)     \ Current filter output
\ The simple low level words are defined next:
:  ClearMACQ (  --  )   \ Initialize values in the MACQ to 0
   0  x(n) ! 0  x(n-1) !  0  x(n-2) !  ;
:  Enter  ( n -- )     \ n = new value,   put into MACQ
     x(n-1) @ x(n-2) !  \ Shift data down the MACQ
     x(n) @ x(n-1) !
     x(n)  !  ;         \ x(n)=new value
:  Filter ( -- )   \ 60 Hz Notch Filter y(n)=(x(n)+x(n-2))/2
     x(n) @ x(n-2) @ + 2/ y(n) ! ;   \ Update y(n)
\ High level words are last.
:  DAS ( -- )  \ Continuous Data Acquisition System fs=240Hz
     ClearMACQ              \ Initialize data structures
     0 Mux                  \ Set A/D channel 0
     rtSetupScrollGraph     \ initialize scroll graph window
     240 SetFs              \ establish 240Hz A/D sampling
     begin
       Wait                 \ Synchronize process to 240 Hz
       A/D                  \ Sample A/D
       Enter                \ Put into X array
       Filter               \ Execute digital filter
       rtUpdateScrollGraph  \ Update Scroll graph window
     key? until ;           \ loop until a key is pressed
```

Real Time Graphics

A library of real time graphics routines allow the student to create complex color displays. The real time VGA graphic library includes many types of windows: text, annunicators, bar graphs, scroll graphs, sweep graphs, arc meters, logic graphs, and x-y scatter graphs. Figures 1 and 2 illustrate the power and flexibility of these routines. These routines include curve-fitting, PID controllers and FFT calculations.

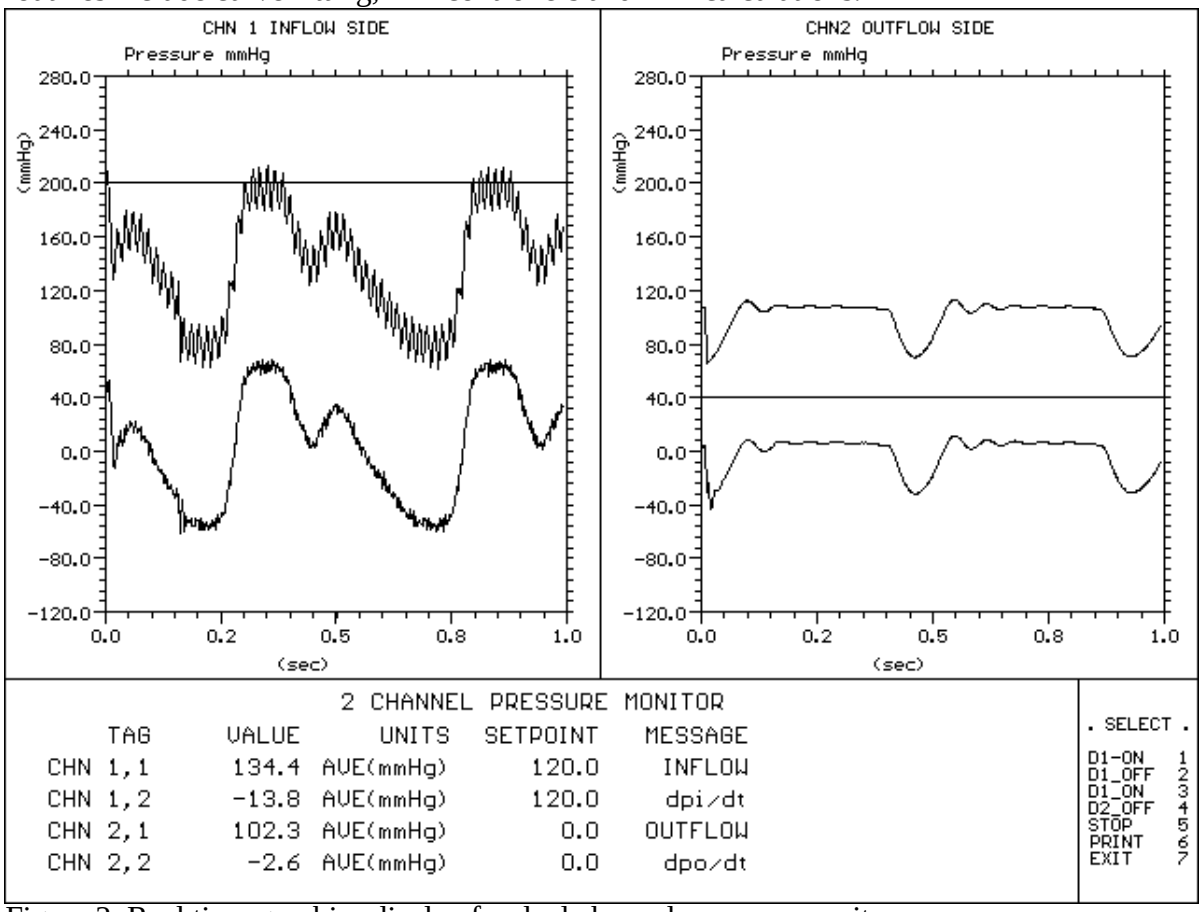

Figure 2. Real time graphics display for dual channel pressure monitor.

Prosym

PROSYM (PROfiler and SYMbolic debugger) provides a common methodology for program monitoring and debugging. PROSYM is an extendible debugging tool implemented as a test bed and research platform. We have had tremendous success using PROSYM both in teaching software techniques, and in developing embedded real-time instruments.

The key features of PROSYM are its interpreter, user-programmable conditional breakpoints, and the ability to append user-defined programs (in the native language) before and after each trap. It has the ability to analyze the real-time execution of multi-tasking software systems. PROSYM provides a simple set of primitives with a high expressive power that may be used singly or may be combined to construct complex debugging tools. The programmability feature allows the user to develop powerful customized debugging tools. In addition to providing basic debugging functions, PROSYM supports an event-action model of debugging. The event-action model provides mechanisms that allow reasoning about events and allow events to be traced and timed at different levels of abstraction. PROSYM allows correction via documented patches

without disrupting program structure. PROSYM is non-intrusive and non-invasive. The normal sequence for setting a trap instrument is as follows:

1. Identify an *instrumentation point*
2. Set *trap conditional, preattach and postattach*
3. Insert *trap instrument*

Instrumentation points are identified by name. They may be global or local. The `Trap` command identifies a global instrumentation point, and the `RTrap` command identifies a local instrumentation point. Global instrumentation points are defined on entry points to routines or identifiers (all calls to the routine), and local instrumentation points are defined on points of invocation (only calls to the routine from a specified point).

A trap instrument consists of a trap conditional, two attachments, and a trap handler. Trap conditionals are user or system defined routines that return a logical condition for evaluation by a trap instrument. Two system defined trap conditionals are `Halt` and `Continue`. `Halt` asserts "invoke trap handler" by always returning true and `Continue` asserts "bypass trap handler" by always returning false. Trap conditionals are set using the `Set_Conditional` (`SC`) command. The default trap conditional is `Halt`.

The two attachments, preattach and postattach, are arbitrary user defined routines. Preattach is set using the `Set_PreAttach` (`SPR`) command and postattach is set using the `Set_PostAttach` (`SPO`) command. System defaults for preattach and postattach are null routines.

The *trap handler* is a system routine that normally returns control to the command interpreter. Unless redirected, the command interpreter takes its input from the keyboard and sends its output to the screen. The trap instrument invokes the trap handler depending on the truth value returned by the trap conditional. A list of instrumentation points and their trap instruments is maintained in a *trap table*.

When the locus of execution reaches an active instrumentation point, the associated trap instrument is invoked. Upon invocation, a trap instrument does the following:

1. Invokes pre-attach.
2. Invokes trap conditional (returns a logical true/false).
3. Invokes the trap handler depending the trap conditional.
   - If true then the trap instrument invokes the trap handler (halts).
   - If false then the trap instrument bypasses the trap handler (continues).
4. Executes the routine at the trap point.
5. Invokes post-attach.

Pre-attach, therefore, executes at the instrumentation point before the trap conditional executes, and post-attach executes just before the trap instrument relinquishes control.

The following example illustrates the power and flexibility of PROSYM. The purpose is to verify the A/D sampling rate in the previous real time data acquisition system.

```
array Times 4800 allot    \ place for 2400 time measurements
PT variable               \ pointer to where to put next
Times PT !                \ initialize PT to beginning
: Next ( -- )
    ReadTime      \ current time (16bits) from hardware clock
    PT @ !        \ put into Times array
    PT @ 2+ Times 4798 + min    \ prevents overflow
    PT ! ;        \ update pointer to next measurement
RTrap A/D DAS     \ trap A/D word within DAS
SC continue       \ do not halt, i.e., keep on executing
SPR Next          \ (preattach) execute Next before each A/D
```

After the above debugger commands are executed, the timing experiment is performed simply by running DAS. After DAS is stopped, the array Times can be printed out in order to verify the accuracy of the "real time" process. The maximum time jitter is less than 5 µs on our 12 MHz 286 machine. The fact that debugging occurs without source code modification has two advantages. First, one is guaranteed that the actual system is being tested, and not one which has been modified by the edit/compile/download cycle. The second advantage is that it is easy to remove all debugger functions (with a Remove_All command) guaranteeing that the system is left with no unintentional debugger side-effects.

The following example uses the debugger to convert the 16 bit A/D system into 12 bits. In this way, the student can study the effect of A/D precision on system performance.

```
hex
: Reduce ( n1 -- n2 )  \ Convert 16 bit n1 into 12 bit n2
    FFF0 and ;     \ make the bottom 4 bits zero
Trap A/D           \ trap all calls to A/D
SC Continue        \ do not halt, i.e., keep on executing
SPO Reduce         \ (postattach) execute Reduce after A/D
```

Once this debugging instrument is invoked, the software system is run in its usual fashion. Every call to the A/D sample routine will now return a 12 bit instead of a 16 bit value.

This last example illustrates how the debugger can be used to measure software timing information in real time. The routines StartClk and StopClk start and stop an interval timer. Our system has both a 16 bit and a 32 bit timer, both with a time resolution about 1 µs. These functions use the existing hardware clock on every PC. The following debugger sequence performs multiple execution speed measurements:

```
array Times 4800 allot    \ place for 2400 time measurements
PT variable               \ pointer to where to put next
Times PT !                \ initialize PT to beginning
: SaveClk ( -- )
    StopClk               \ Stop time interval measurement
    CurrentTime @ \ current time measurement (16bits)
    PT @ !        \ put into Times array
    PT @ 2+ Times 4798 + min    \ prevents overflow
    PT ! ;        \ update pointer to next measurement
Trap Filter       \ Measure the execution speed of Filter
SC Continue       \ do not halt, i.e., keep on executing
SPR StartClk      \ (preattach) start timer before Filter
SPO SaveClk       \ (postattach) save time after Filter
```

Once this debugging instrument is invoked, the software system is again run in its usual fashion. The overhead involved in making these measurements ranges from 10 to 50 µs depending on the processor speed. An important advantage of this process is that the debugging is performed in real time on the actual hardware/software system. After the system has been executing a while, statistical analysis of the array Times will yield simple parameters like maximum and average execution speed, as well as more complex statistical information like probability density function.

Comparison between FORTH and Labview

Consider the analogy of the spreadsheet and the accountant. If the person understands the fundamentals of accounting, then the spreadsheet can be a very powerful tool. On the other hand, the spreadsheet by itself does not teach accounting. In a similar way, Labview can be a powerful tool for the experienced engineer, but not for the student. When developing a real time data acquisition or control system, the electrical engineer must be able to make design choices based on realistic limitations of the hardware and software. Therefore, the educational laboratory environment must empower the student with the ability to control and evaluate this hardware/software interaction.

Conclusions

This paper presents a laboratory environment for the development of real time data acquisition and control on the IBM-PC platform. The minimum hardware system includes a PC-clone, a math coprocessor, 640 Kbytes of RAM, 10 Mbytes of hard disk space, and VGA color graphics. The only add-on hardware that is required is an A/D&D/A data acquisition board. The system runs under both Windows and DOS. The software base integrates an editor, a spreadsheet, and a real time programming environment built around Druma FORTH. We have written many FORTH libraries to assist the student in the translation of engineering concept into creation. The system establishes software links to any commercially-available editor or spreadsheet program including those on the Macintosh. There are both fixed point and floating point math packages. These routines include curve-fitting, PID controllers and FFT calculations. Simple easy-to-use drivers interface both digital and analog I/O signals to the system. We can sample the A/D at rates up to 30 KHz on a 12 MHz 286 machine. Real time events are managed using a rich set of FORTH software routines which guarantee that time-critical software is executed on schedule. The maximum time jitter is about 5 µs on our 12 MHz 286 machine. These timing routines will run on any PC-clone without additional hardware. There are two stop watch timers (each with a time resolution of less than a 1µs) which can be used to make performance measurements. The 16 bit stop watch timer has a range of 0 to 56ms, and the 32 bit stop watch timer has a range of 0 to 1 hour. The real time VGA graphic library includes many types of windows: text, annunicators, bar graphs, scroll graphs, sweep graphs, arc meters, logic graphs, and x-y scatter graphs. We have developed an extendible debugging tool called PROSYM. PROSYM provides a simple set of primitives with a high expressive power that may be used singly or may be combined to construct customized debugging tools. In addition to providing basic debugging functions, PROSYM supports an event-action model of debugging. The event-action model provides mechanisms that allow reasoning about events and allow events to be traced and timed at different levels of abstraction. PROSYM allows correction via documented patches without disrupting program structure. PROSYM is non-intrusive and non-invasive. We have evaluated this development system on the full range of PC platforms from the original PC-XT to the newest 486 systems. The environment has been used for two years by Engineering graduate students performing both teaching and research projects.

Biographies

Jonathan W. Valvano was born in Clinton, CT in 1953. He received a B.S. degree in computer science and engineering, and a M.S. degree in electrical engineering and computer science from the Massachusetts Institute of Technology in 1977. He received his Ph.D. in medical engineering from the Harvard/MIT Division of Health Sciences and Technology in 1981. He is currently an Associate Professor at the University of Texas at Austin performing research in the fields of perfusion measurements, bioinstrumentation, and real time systems for embedded control.

Bapi Ahmad was born in Chittagong, Bangladesh in 1953. He received his B.S. degree in 1980 and his M.S. degree in 1983 in electrical and computer engineering from the University of Texas at Austin. His special interests are in programming tools for embedded real time systems.

Jagadish V. Nayak was born in Mangalore, India in 1967. He received his B.S. degree in 1988 in electronics and telecommunications from the Goa College of Engineering in India. He received his M.S. degree in 1992 in electrical and computer engineering from the University of Texas at Austin. His M.S. thesis research included the floating point and real time graphics libraries described in this paper.