*Int. J. Engng Ed.* Vol. 8, No. 4, pp. 288–302, 1992
Printed in Great Britain.

0742–0269/91 $3.00+0.00
© 1992 TEMPUS Publications.

# Microprocessor-based Control Systems: System Development in a PC Environment*

J. E. COOLING
A. H. WHITFIELD
G. M. AL-SADDIKI

*Department of Electronic and Electrical Engineering, University of Technology, Loughborough LE11 3TU, U.K.*

*This paper describes a general-purpose PC-based workstation for use with embedded control systems projects. Its purpose is to provide a self-contained environment for research, design and development activities for both undergraduates and postgraduates. Its requirements, structure and use, both in terms of hardware and software, are described in the context of an adaptive (self-tuning) control system research project.*

## INTRODUCTION

COMPUTERIZED tools for the analysis, synthesis and development of control systems have generally been based on the use of mainframe and minicomputers. More recently, however, much attention has been focused on low-cost alternatives using personal computers (PCs). In many cases the emphasis has been on training, analysis and theoretical synthesis [1]. Little has been reported concerning the use of PCs in the development of practical embedded systems, typified by servo applications. Yet the PC can provide many facilities needed for all phases of the research and design activity. In this paper we show how a PC-based development environment satisfies the requirements of the control student, from hardware design to system testing. The hardware and software configuration described here is specifically intended for use in practical microprocessor-based projects. These include:

- second-year undergraduate laboratory setwork;
- final-year projects (individual activities);
- postgraduate research work.

Our objective was to provide a comprehensive, stand-alone facility at a reasonable cost. Experiences gained in undergraduate (final year) and postgraduate project work were instrumental in developing this facility. Its form, function and features are here described in the context of work on practical embedded control systems—specifically a research project investigating the use and performance of adaptive control techniques in servo-type systems.

The work described here will be of particular interest to those involved in the development of undergraduate (and similar) courses in:

- practical digital control systems;
- real-time embedded systems;
- software engineering;
- microprocessor engineering.

## CONTROL SYSTEM DEVELOPMENT

### *General*

Control systems vary considerably. At one extreme are fast high-performance systems such as weapon stabilizers. In such applications many aspects of the design are new and unique, and the related control design effort is considerable. At the other end of the spectrum are process systems where the control engineer is asked merely to supply a three-term controller. Here the control design is minimal (though the system design may be considerable).

The system described here, together with relevant research and design techniques, falls into the first category. Figure 1 shows the main objectives of the programme, these being the development of the target system controller and the development of appropriate control techniques. The target system ('plant') is real as opposed to simulated, and is described in detail below.

In the past we had been faced with a number of difficulties when embarking on projects such as these. The development facilities were fragmented, with little interconnectivity. For instance, simulations were usually carried out on mainframe computers, using terminal access methods. Students thus had to acquire a working knowledge of the mainframe operating system (in our case Multics). Simulations were done using standard control

* Paper accepted 10 September 1992.

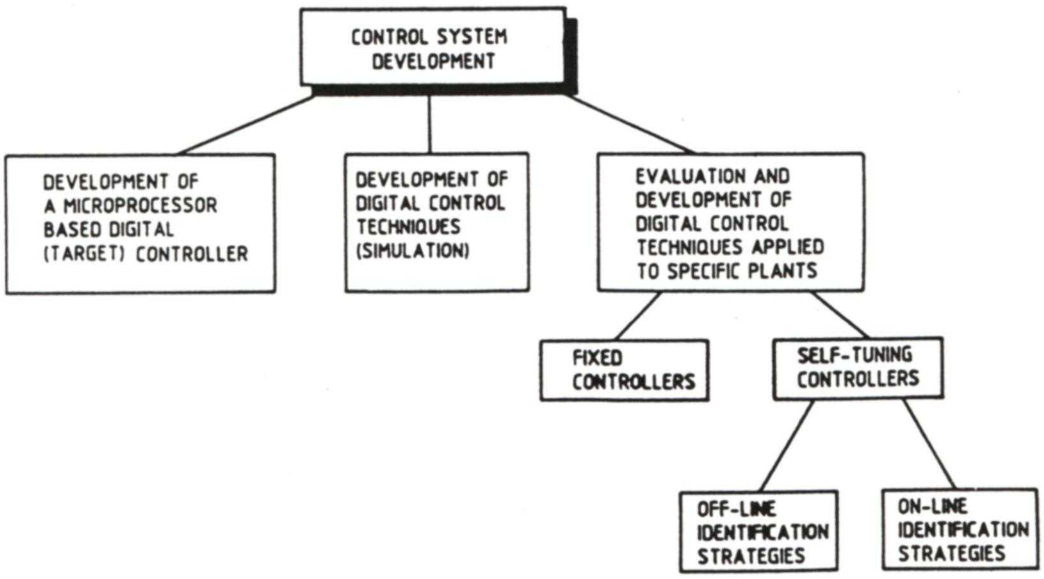

CONTROL SYSTEM DEVELOPMENT
DEVELOPMENT OF A MICROPROCESSOR BASED DIGITAL (TARGET) CONTROLLER
DEVELOPMENT OF DIGITAL CONTROL TECHNIQUES (SIMULATION)
EVALUATION AND DEVELOPMENT OF DIGITAL CONTROL TECHNIQUES APPLIED TO SPECIFIC PLANTS
FIXED CONTROLLERS
SELF-TUNING CONTROLLERS
OFF-LINE IDENTIFICATION STRATEGIES
ON-LINE IDENTIFICATION STRATEGIES

Fig. 1. System development programme.

packages or, more commonly, specially written simulation programs. For convenience these were usually coded in FORTRAN—thus enabling the use of the NAG numerical libraries. However, microprocessor-based software was generally produced on microcomputer development systems (MDSs), which have a quite different operating system (and text editor, etc.). Programming had, in the main, been done in assembly language.

Many of the projects called for application-specific hardware. This in turn led to the need to produce new designs. Unfortunately the testing and debugging of such designs frequently proved to be extensive and time-consuming. Final-year student projects were particularly prone to problems here—the limited supply of in-circuit emulators (ICEs) and logic analyzers caused regular bottlenecks.

Finally, the development and evaluation of digital control techniques tended to be a tedious and manually intensive process. Direct acquisition of plant and controller data was normally performed using chart recorders or $X$–$Y$ plotters. In general it was extremely difficult to analyse these results automatically—a necessity for system identification to work. And, on a mundane note, demand for such recorders and plotters frequently outstripped supply.

It was clear to us that this situation was adversely affecting our students' project work: significant improvements were needed. Central to these was the use of an integrated development facility.

## *The integrated workstation*

By 'integrated workstation' we mean a facility that supports all phases of the design, development and test activities. Until recently the high cost of such units has limited their availability to the researcher. Now, however, PC-based systems are changing this situation, providing a variety of tools relatively cheaply.

In developing the workstation the basic requirements of the development facility must first be established. The main criteria are:

1. The system must use a standard operating system to take advantage of standard software; this also saves users the effort of having to familiarize themselves with yet another system.
2. Software must be available for both high-level and assembly language support. Further, this *must* be able to generate object code for Eprom programming.
3. Software debugging tools should be part of the complete package.
4. A hard disk should be used whenever possible.
5. A good text editor should be included as standard.
6. The PC must have sufficient memory space to accommodate modern programming languages and computer-aided engineering/computer-aided design (CAE/CAD) packages.
7. A printer is an essential item.
8. Graphics facilities are highly desirable.
9. An Eprom programmer is essential; an Eprom emulator is desirable.
10. A logic analyzer is essential; one that is PC compatible is highly desirable.
11. An ICE is desirable but not essential.
12. Multiple standard serial communications ports should be available.
13. The station should be reasonably compact and mobile.

The essential points of the workstation developed for this programme are shown in Fig. 2; the actual equipment is shown in Fig. 3. The PC is capable of

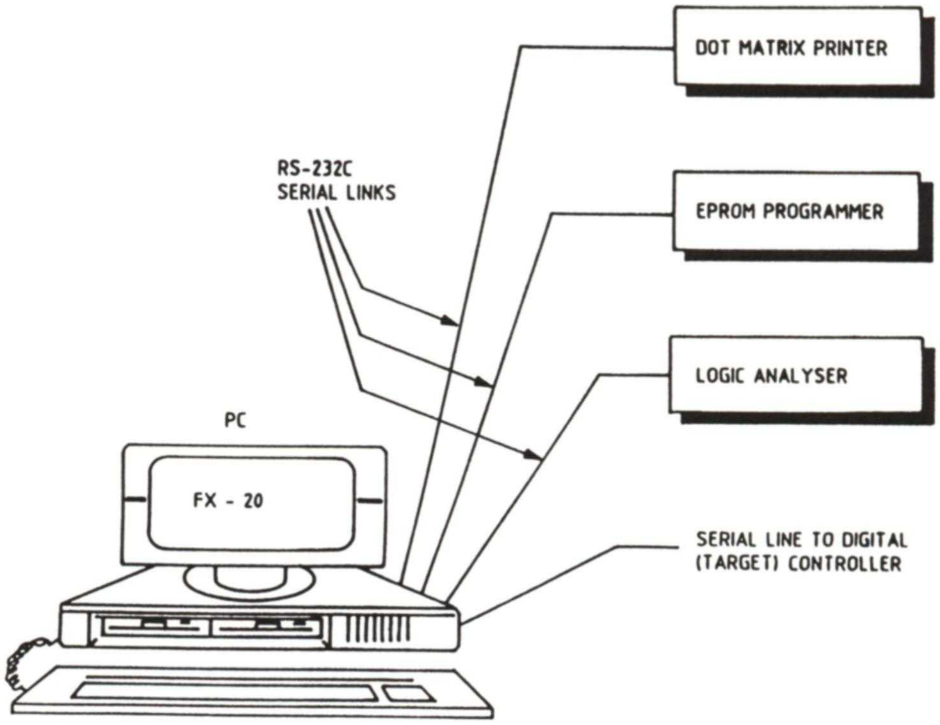


Fig. 2. PC-based integrated workstation.

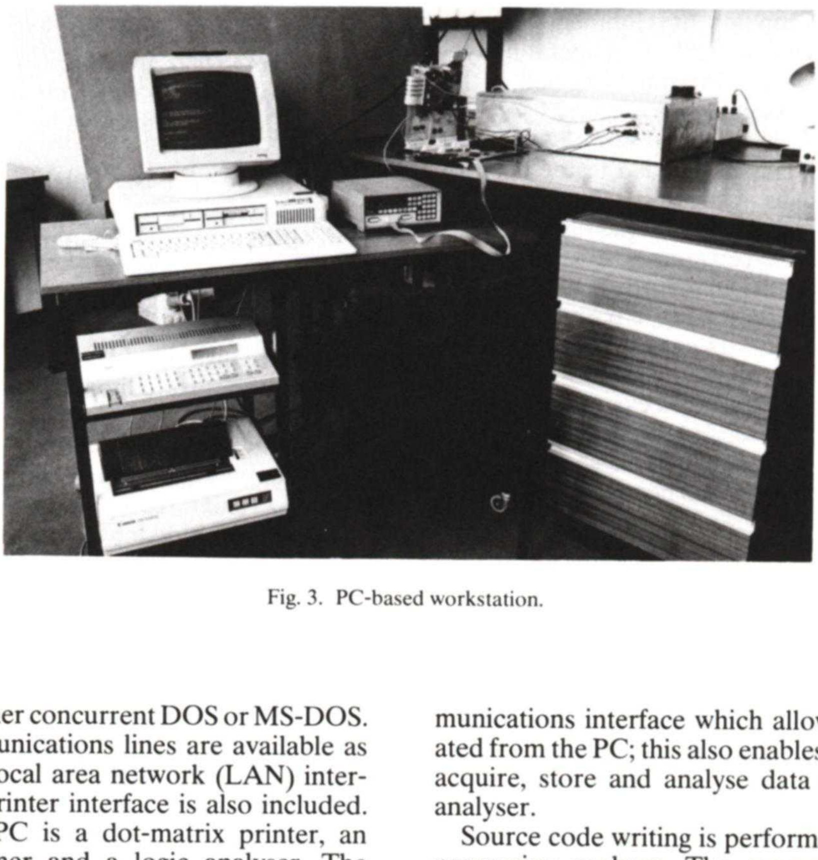


Fig. 3. PC-based workstation.

running under either concurrent DOS or MS-DOS. Two serial communications lines are available as standard, plus a local area network (LAN) interface. A parallel printer interface is also included. External to the PC is a dot-matrix printer, an Eprom programmer and a logic analyser. The programmer is capable of programming all standard Eproms and, with adaptors, single-chip micros and programmable array logic (PAL) devices. The logic analyser is equipped with a serial communications interface which allows it to be operated from the PC; this also enables the computer to acquire, store and analyse data gathered by the analyser.

Source code writing is performed using a word-processing package. The source code is subsequently compiled or assembled using Pascal/MT+ [2] or the 8086 assembler ASM86. Interactive test and debugging of programs may be carried out within the PC environment using the dynamic debugging tool Codeview [3]. Interactive development and test of the plant control algorithms is carried out from the PC using serial communications with the target system controller.

The relevance and use of the various hardware and software facilities are detailed below.

## THE CONTROLLED PLANT

### *Introduction to self-tuning systems*

Adaptive self-tuning systems have been the subject of a great deal of research effort in recent years (see [4] for a major review). Practical applications have lagged behind such work, being confined in the main to the process industries [5–7]. Few servo applications have been reported, though the advantages of self-tuners in these areas have been shown by Kanniah *et al.* [8] and Hope *et al.* [9]. The wide bandwidths and demanding performance specifications raise problems not found in the process world. In fact such problems have led researchers to develop alternative adaptive control techniques [10]. However, the object here is to develop and evaluate 'conventional' self-tuning techniques applied to electromechanical systems in the presence of stiction, friction and velocity saturation.

Conventional self-tuning methods consist of two parts: system identification [11] and system control [12]. Identification may be carried out continuously by the embedded controller as the plant runs; alternatively it may be computed from recorded plant data. The first case is known as 'on-line' identification, the second as 'off-line'. Here we describe the use of a PC, interfaced to the plant controller, to carry out off-line identification of the plant dynamics. It is also used as a software development tool for various identification techniques. This allows the student to evaluate the performance of different methods operating on the same set of data. The PC, though, is not limited to the role of data collector and analyser; it also enables the operator actively to control the test sequence via the plant controller.

### *The plant*

A block diagram of the 'plant' used for control system development is shown in Fig. 4; Fig. 5 is a photo of the actual test rig. This is an electromechanical actuator (suitable for use with process control valves) coupled to a mechanical load simulator. Closed-loop control on actuator position is provided by the digital controller described in this paper. The actuator motor and associated control/power electronics are considered to be part of the plant itself, as is the load shaft position sensor.

The drive unit of the actuator is a 1/6 h.p. induction motor, originally designed for 115 v, 60 Hz, three-phase operation. It is powered from a static inverter which, when used with its controller, provides full linear speed control from zero to maximum speed in both directions. Maximum motor shaft speed is 2000 r.p.m., though the maximum load shaft speed is 1.2 cm/sec.

A gearbox is used to translate motor shaft rotary motion to linear motion of the load shaft. Both the load resilience and viscous force can be varied by the rig operator. Furthermore, the effect of valve loading is simulated using a coulomb damper (a disc brake) on the motor shaft; this, too, is adjustable. Position sensing is carried out using a continuous-track rectilinear potentiometer. Motor speed control is carried out by a pulse-width-modulated (PWM) controller in conjunction with a three-phase static inverter [13]. The inverter uses power field effect transistors, connected in a full bridge configuration, to switch power to the motor. Motor speed is determined by the switching frequency of the transistors, this being set by the PWM controller. In turn this is determined by the analogue input signal to the controller, the switching frequency range being 0–1 kHz (approximately). Constant flux conditions within the motor are maintained by modulating the drive pulse width, though voltage boosting is used at low frequencies to compensate for shaft stiction effects. Thus the motor drive frequency is directly proportional to the input signal, setting the synchronous speed of the motor. Note that as the motor is an induction type its

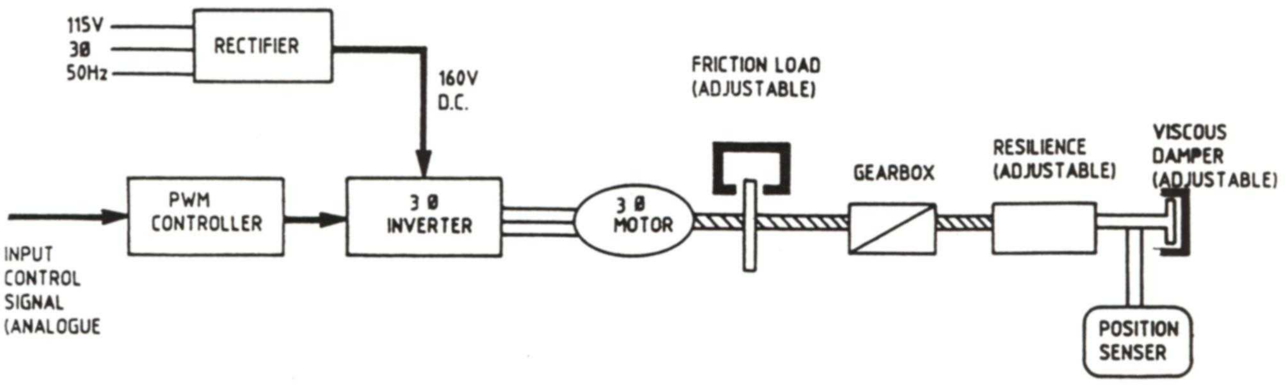


Fig. 4. Controlled plant.

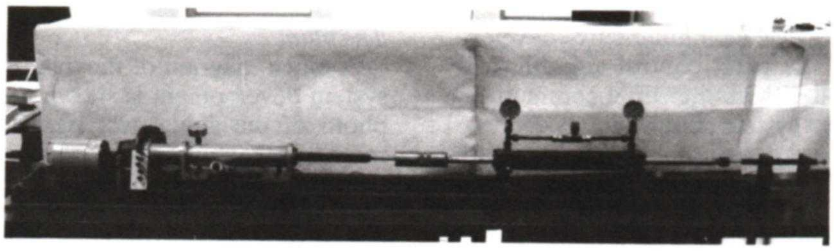


Fig. 5. The actuator rig.

actual speed depends on shaft loading, i.e. it exhibits slip.

## DIGITAL CONTROLLER—ELECTRONIC HARDWARE

### *General information*

The unit described here (Fig. 6) is designed to be used as a general-purpose digital controller in closed-loop systems. Although intended for use in laboratory conditions, its design reflects the requirements of real systems. It can be seen that the complete system consists of three main building blocks:

- microcomputer (CPU) section;
- analogue input/output (I/O) section;
- serial communications I/O section.

All digital electronics are housed on a single printed circuit board, the analogue subsystem being located on a small adaptor board. This technique enables a standard computer section to be tailored to meet specific I/O requirements. The complete assembly is defined as the 'target system', the micro being the target processor.

### *CPU section*

This is based on the use of the Intel 8088 microprocessor, augmented by an 8087 numeric data coprocessor. It is designed to handle a maximum memory address space of 64 kbyte, all devices being memory (not I/O) mapped. The address space is programmable through the use of PROM decoding techniques. Currently the system is equipped with 32 kbyte of Eprom for program and fixed data storage, together with the 16 kbyte of

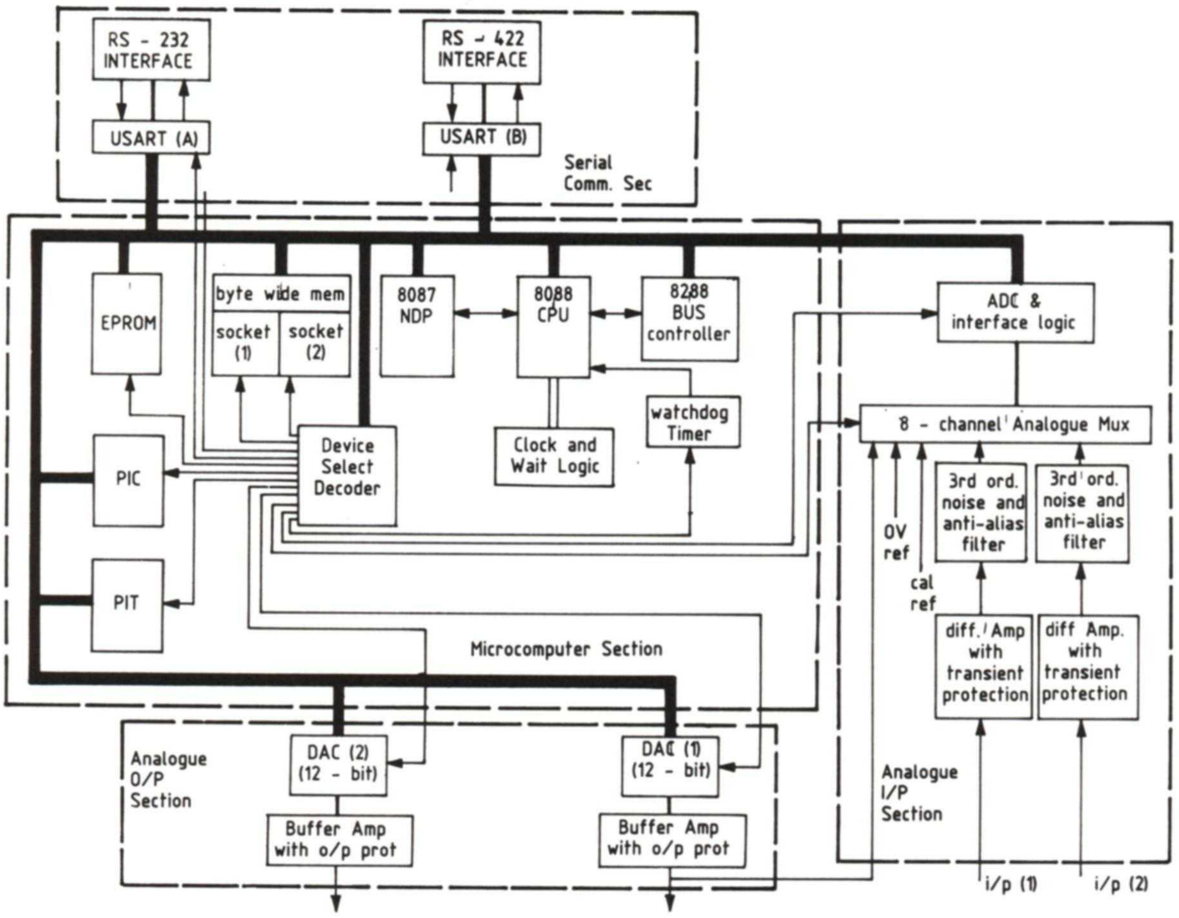


Fig. 6. Digital controller.

RAM for variable data storage and stack operations. The design also requires the use of timing and interrupt functions. These are implemented using standard microprocessor-compatible components, as these are programmable for flexibility. As a safety feature a watchdog timer is included in the CPU design. Also included is a wait-state generator to enable the processor to be single-stepped through its program sequence.

In normal circumstances restricting the address space to 64 kbyte allows the processor to operate in minimum mode [14]. Unfortunately, when using an 8087 the system has to be in maximum mode; as a result an 8288 bus controller must be added to the design.

*Analogue I/O section*

Two analogue input channels are provided. One is for the measured value variable (shaft position), the other being an optional extra for shaft velocity. Each signal is input via a differential amplifier, bandlimited by a third-order, low-pass anti-alias filter and digitized by a 12-bit successive approximation analogue to digital converter (ADC). Conversion time is insignificant when compared with computing activities. Additional internal analogue signals are digitized, including calibration reference values and DAC outputs. Signal selection is carried out using an analogue multiplexer, the output from this being applied to a sample-hold module prior to digitization. Two output stages are provided, each one consisting of a 12-bit digital to analogue (DAC) converter followed by a buffer amplifier. Simple (low-pass filtering is used to minimize the effects of DAC glitch spikes and to act in part as a reconstitution filter. Both short circuit and transient overvoltage protection are included for the output amplifiers.

*Serial communications*

Two full-duplex serial communication channels are incorporated into the design—a short distance one corresponding to the RS232C standard [15] and a longer distance RS422 version [16]. The RS232 version is designed mainly for interactive working with the PC. When used in this mode the maximum data rate is 9.6 kbaud, this being the limit of the PC. The RS422 channel is included to support controller operations within a distributed control system.

*The PC as an aid to hardware design*

The development philosophy used here is to bring the hardware to a fully functional state by incremental testing. That is, to start off with the minimum amount of circuitry on the board, verify its correct functioning, add more circuitry, repeat the process, and so on. The ICE is an ideal tool for use with this process. Unfortunately, because it is relatively expensive, its availability is limited; this creates a bottleneck in the development process. For hardware testing, the combination of a single-step facility used in conjunction with a bus monitor enables static testing to be carried out. For this approach it is highly advantageous to bring the RS232 link into operation as soon as possible. Further testing can be carried out interactively using a standard terminal device. Dynamic fault analysis (should it occur) is done with the aid of the logic analyser.

For this level of test it is essential to work in assembly language. Here the PC acts as the microprocessor development system, supporting source code writing, program assembly and PROM programming functions. Once the RS232 serial communications channel is operational, the PC can be used as an interactive terminal to assist in the functional testing of the remainder of the electronics. Moreover its data storage facilities, and the ability to print such information out, is an extremely useful tool for post-mortem analysis.

## FUNCTIONAL TESTING AND SYSTEM IDENTIFICATION

*Overview*

The ultimate objective of this test is to produce a mathematical model of the plant based on measurements of the plant input (control) and output (measured value) signals. Therefore it is necessary for the experimenter to control the test procedures, setting conditions such as signal type and duration, sampling rates and number of measurement points. The set-up used for these tests is shown in Fig. 7, the use of the PC being self-evident from the following text. Note that there are two distinct aspects of the operation. In the first place the plant has to be run to obtain data; subsequently, the information so obtained is used as part of the model identification process. This is carried out within the PC.

*Plant testing*

The control program, which actually runs the plant and collects data measurements, sits within the target controller in Eprom. The source code for the control program is written in Pascal/MT+; details are given below in the software development section. Program development, i.e. code writing, compilation, linkage and PROM blowing, takes place within the PC environment. During plant testing the PC functions as a terminal with data storage facilities, and communicates with the target controller using its RS232 serial data line. To start the test procedure the controller and plant are powered up. The PC must be connected to the controller, and set into terminal mode. Instructions from the controller to the operator are displayed on the PC screen, with responses being entered at the keyboard. These include plant test data and designation of the PC data file (on disk) that is to be used to hold the plant measurements. A typical test procedure is shown in Fig. 8 where, once the test parameters are entered by the operator, the controller runs the plant through its test sequence. During this, measurements are made of the control

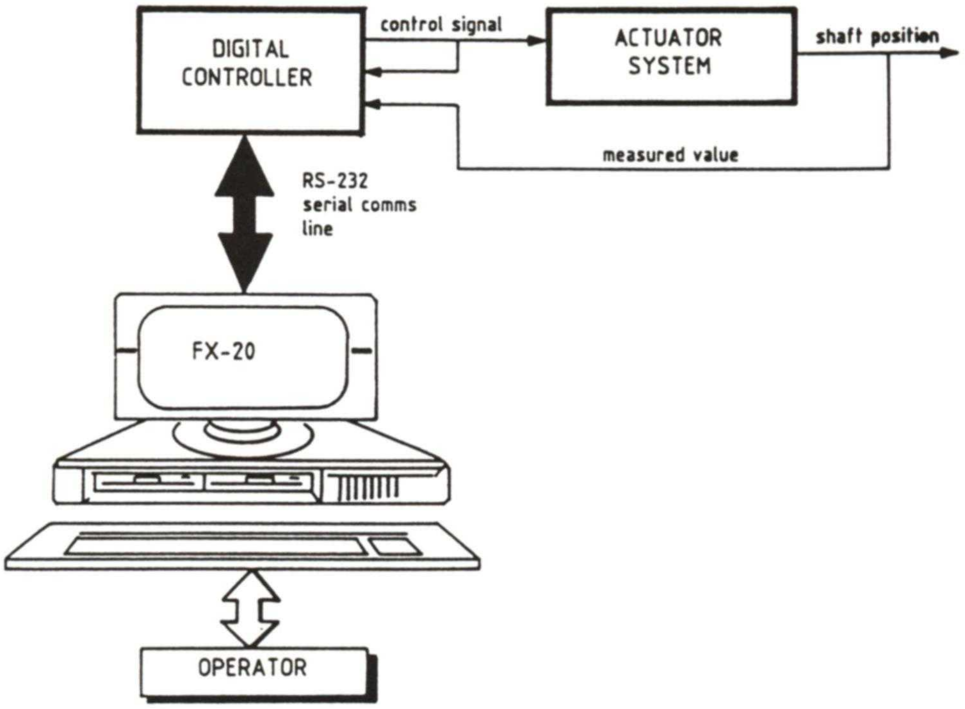


Fig. 7. System organization—functional testing.

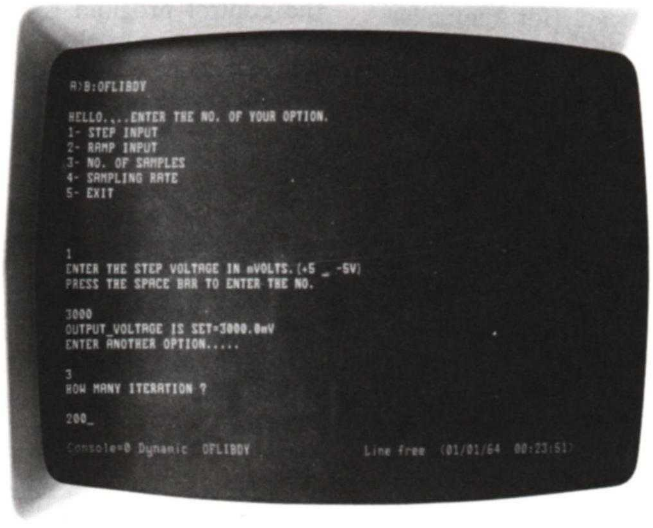
signal and measured value, and these are stored within the memory of the controller. At the end of the test the data are transferred to the named disk file in the PC. If required, the information can also be printed out for evaluation and review.

*Off-line identification*

The purpose of the off-line identification process is twofold. Firstly, using the data recorded during the test run, it enables a mathematical model of the plant to be generated. Secondly, using this same set of data, a number of models can be obtained by applying various identification schemes. By comparing these with the model derived from frequency response testing of the plant, the relative performance of the identification methods can be assessed.

Evaluation of a number of identification schemes may be carried out, including recursive least squares (RLS), recursive extended least squares (RELS), recursive maximum likelihood (RML) and recursive instrumental variable (RIV) methods. The process of plant identification can be better understood by considering one of these in more detail, RLS being a suitable candidate.

Figure 9 illustrates the concept involved in



```
A>B:OFLIBDY

HELLO.....ENTER THE NO. OF YOUR OPTION.
1- STEP INPUT
2- RAMP INPUT
3- NO. OF SAMPLES
4- SAMPLING RATE
5- EXIT



1
ENTER THE STEP VOLTAGE IN mVOLTS.(+5 _ -5V)
PRESS THE SPACE BAR TO ENTER THE NO.

3000
OUTPUT_VOLTAGE IS SET=3000.0mV
ENTER ANOTHER OPTION.....

3
HOW MANY ITERATION ?

200_

Console=0 Dynamic  OFLIBDY                    Line free  (01/01/64  00:23:51)
```

Fig. 8. Photo—Plant testing—Operator dialogue.

identification and model generation using RLS techniques. Here a program running under the PC operating system performs the following actions:

1. Sets up a preliminary (estimated) model of the plant using information supplied by the operator.
2. Reads the recorded plant control signal at a specific sample instant, applies this to the model, and calculates the resulting output.
3. Reads the recorded plant measured value at the same sample instant and calculates the error between this and the model output.
4. Computes the average error power and its gradient.
5. Adjusts the model parameters to reduce the power gradient.
6. Repeats the above steps (2–5) for all recorded values, working iteratively towards a condition of a zero power gradient.

As the program is executed sample by sample, the model parameters hopefully converge towards those of the plant itself. What we are left with is a best estimate of the plant transfer function, and this information is used in the implementation of an appropriate control scheme. The results may also be stored at each calculation interval, thus allowing the experimenter to review the convergence rate and accuracy of the identification process. It is, of course, necessary to design and write the identification program in the first case, and this is covered in the following section.

## SOFTWARE DESIGN AND DEVELOPMENT

### *Design techniques*

The basic software design method used here is that of structured programming [17] using a top-down development approach. Diagramming techniques are used throughout, specifically Jackson chart methods [18]. These diagrams, Figs 10 and 11, are read from top to bottom to obtain more detail on program activities and from left to right to get the time sequences. The recommended control structures of structured programming have generally been used in the writing of the program source code.

### *Programming language*

It would have been possible to develop the software for the target system entirely in 8086 assembly language. This was rejected, however, and the decision was taken to use a high-level language wherever possible, turning to assembler only as a last resort. Three factors influenced this:

- speed of development;
- problem (and not processor) orientation of high-level languages;
- inherent support by block-structured languages for structured design techniques.

In the context of the PC development environment it makes little sense to use a different language for the off-line programs. Software commonality is highly advisable. For this work the basic language selection criteria were that:

- The compiler must be capable of generating ROMable code.
- Code produced must also run under the PC operating system.

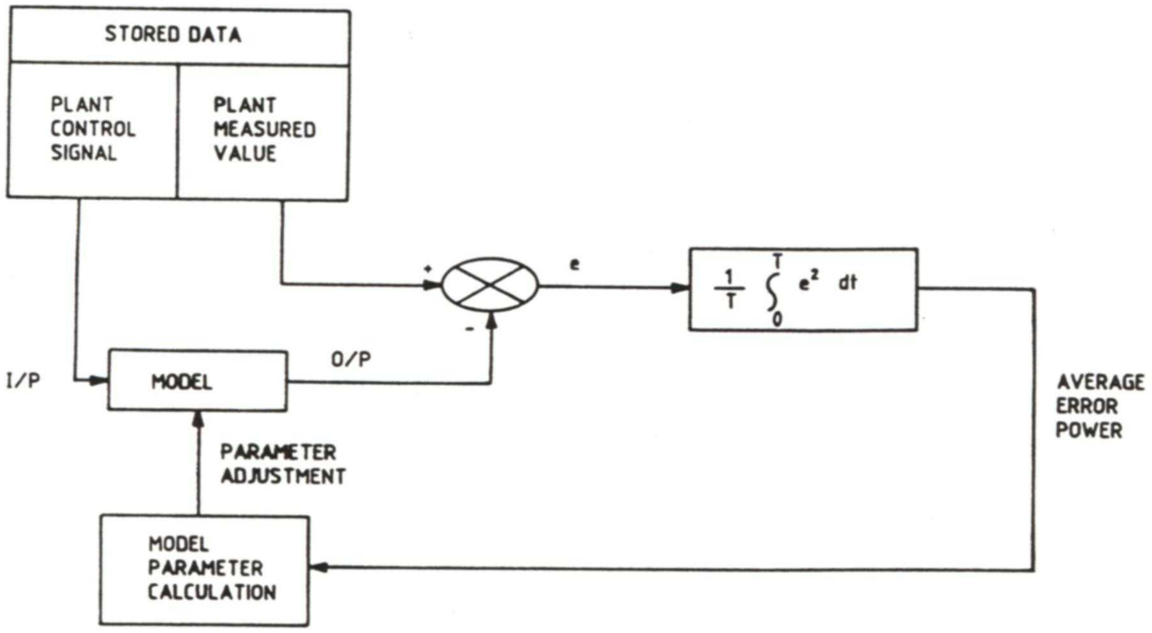


Fig. 9. Plant identification process.

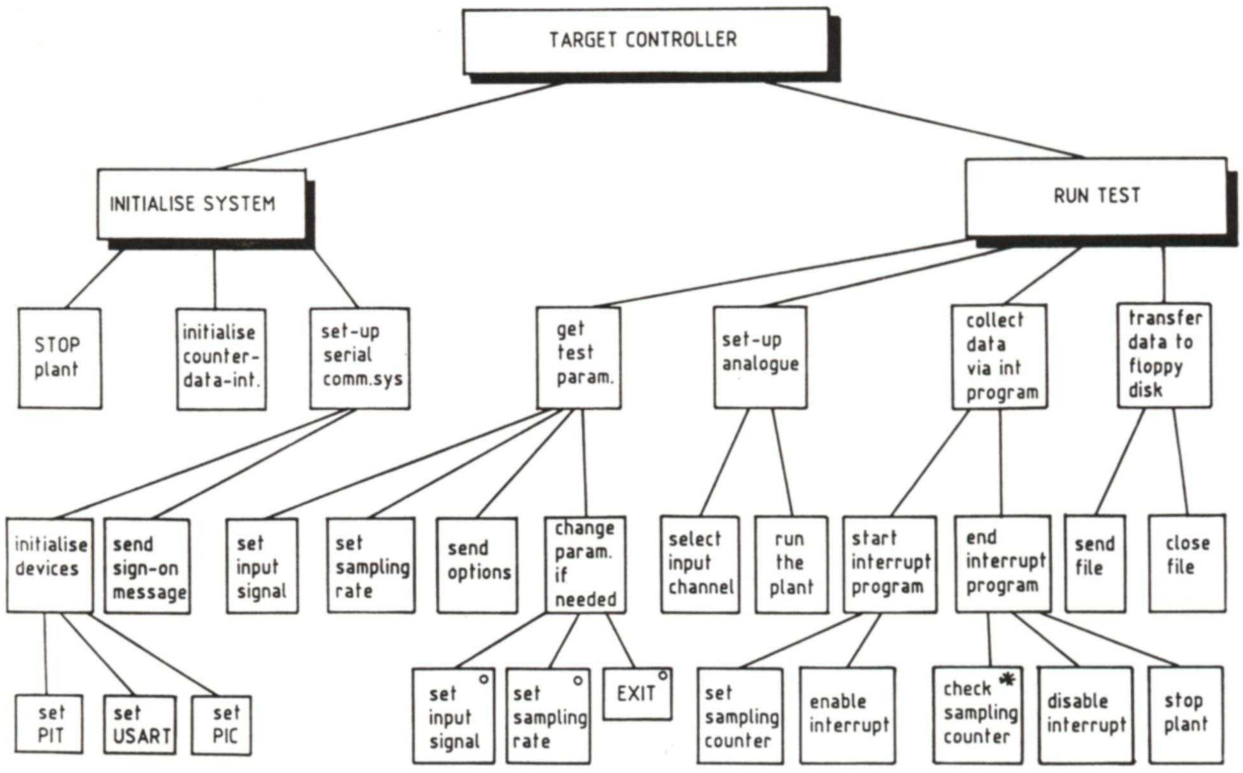


Fig. 10. Software design diagram—target controller.

- Object ('in-line') code inserts must be supported.
- Access of specific memory addresses and hardware devices from the source code (i.e. the high-level language statements) must be a standard feature.
- Access to the processor interrupt structure is essential.

Pascal/MT+ from Digital Research was the chosen language.

*Target controller software*

Figure 10, the structure design chart, is produced from the system requirements. As such it should reflect what needs to be done to run the plant and collect appropriate data. From this the source code (Pascal) is generated. At the higher levels of the chart, operations are fairly self-explanatory. At lower levels a detailed knowledge of the system and hardware is required to understand fully what is happening.

The Pascal program must correspond to the design diagram (otherwise there is not much point in having the diagram in the first place). As an aid to program visibility and clarity, modular programming techniques are used extensively. Fortunately Pascal/MT+ allows modules to be compiled separately, global variables being avoided through the use of 'external' procedure declarations. The top-level of 'program' module is essentially made up of a set of procedures that are located in lower-level modules, as shown in the Appendix (listing 1). Here each program statement consists of a procedure without parameters; as such the sequence is clear, readable and unambiguous. In turn these procedures call lower-level procedures, which, depending on the complexity of operations, may in turn call still lower-level operations. This is demonstrated in listing 2 in the Appendix, which is the code for the procedure 'SET_UP_SERIAL_COMMS'.

The major software design objectives for reliability and clarity are modularization, information hiding, loose coupling and high cohesion [19]. These are well supported by the structure and organization of Pascal. However, for embedded applications, the language must fulfil the following roles:

- Allow the programmer to access the various hardware devices, preferably from high-level language statements.
- Use meaningful and recognizable names and identifiers.

The procedure INITIALISE_DEVICES shows how Pascal/MT+ facilitates such requirements. It also shows that, to develop software for a target system, an intimate knowledge of the processor hardware is needed.

*Off-line identification on the PC*

The off-line identification program is derived from the software design diagram of Fig. 11. Only the higher levels are shown to maintain diagram

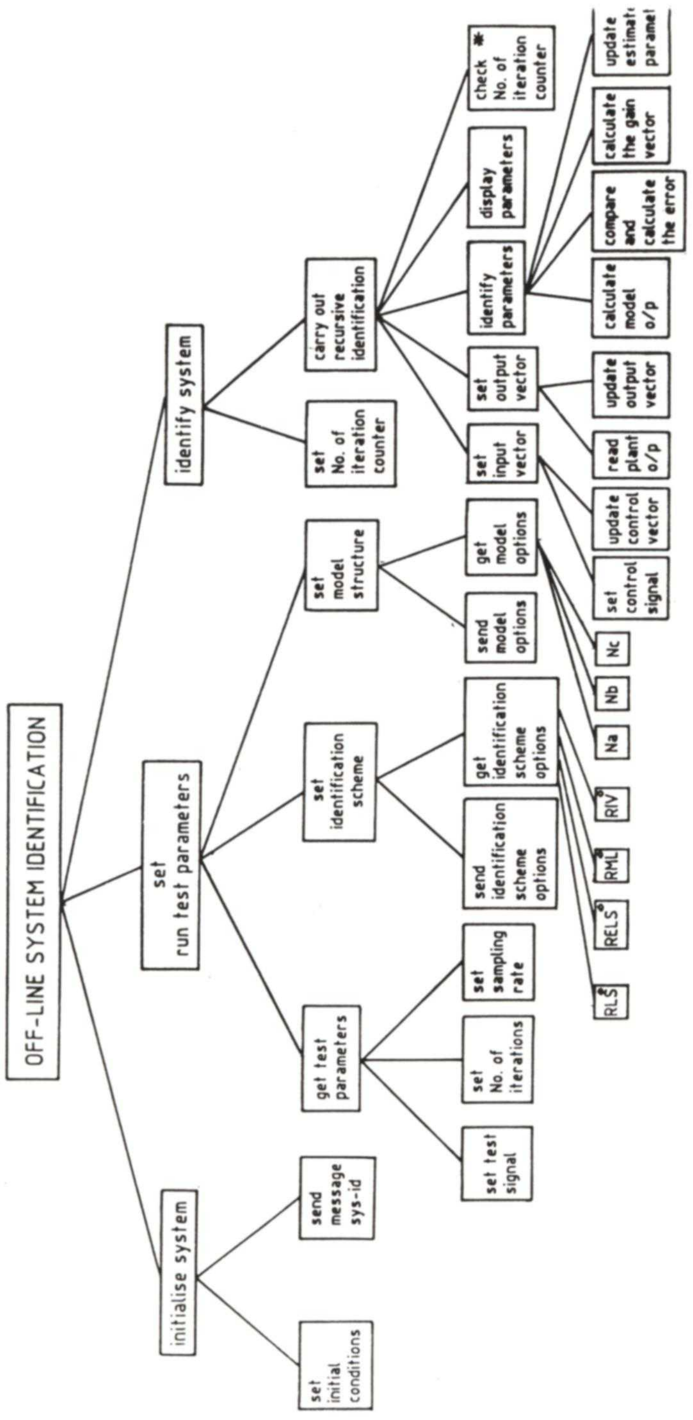


Fig. 11. Software design diagram—Plant identification.

clarity; the essentials of the identification process can, however, be deduced from this.

This program runs on the PC, having been developed on it in the first place. Thus, to produce the identification software, the programmer does not require any special knowledge of the target system. In fact test data are not even needed to evaluate the identification processes; simulations can be run on the PC using information provided by the programmer.

## GENERAL COMMENTS AND CONCLUSIONS

*Experiences—comments and conclusions*

The workstation described here has now been in constant use for over six years. It has been applied to a variety of embedded controller projects involving five different types of microprocessor and numerous programming languages. The majority of these projects have been carried out by undergraduates, at second-year and final-year level. In each case it was used as the basic development tool for both the hardware and software of the controllers. By comparison with traditional MDSs it is a much more cost-effective tool; moreover it is a general-purpose rather than a specialist item. Experience has shown that if the workstation meets the criteria defined earlier, then it provides an excellent environment for the development of practical control systems. However, four particular points should be noted.

Firstly, if the PC had not been able to communicate with the target system its capability would have been significantly impaired. In our own case it was necessary to develop specialist software precisely for this task.

Secondly, few compilers (relatively speaking) are available which generate ROMable code; even then, difficulties may be experienced in actually loading the object code into the Eprom programmer.

Thirdly, the addition of a maths processor to the PC is a worthwhile investment, especially in view of the large number of matrix operations involved in control system analysis and synthesis. Our own tests have shown that the speed-up in performance is, on average, by a factor of 30. For more complex maths operations (such as trancendental functions) the improvement is of the order of 100. Finally, a graphics screen plotting and hard-copy capability, though not essential, is very highly recommended.

*Teaching and course aspects—the future*

We are currently carrying out major changes to the software engineering section of our undergraduate course. In the past our approach to software-based systems has been somewhat fragmented: programming, software development and microprocessor systems have very much been treated as isolated topics. One of the primary aims of the new course structure is to integrate the design, development and programming of software-based systems. In particular, we intend to provide an integrated environment for the development of embedded (microprocessor-based) systems. The equipment (workstation) described here forms the basis of such an environment.

In the first year, students are introduced to programming and algorithm development. This involves both individual and group activities, using the PC for program development. Our standard teaching language is Modula-2 [20], using the Stony Brook compiler.

During the second year, software design and development is a major topic within the software engineering course. Students will be required to develop software to run on target systems, programming in both assembly language and Modula-2 (the emphasis is placed on high-level language programming). To do this they must produce ROMable code, which is placed into the target units. It is at this phase of the work that the full facilities of the workstation are required. Our experiences described in this paper have enabled us to define such facilities with great confidence. We have been able to identify the actual—not merely the perceived—requirements for practical embedded software development. Unfortunately, we have not yet been able to find a textbook that covers *all* aspects of the course.

In conclusion, our decision to develop a PC environment for practical microprocessor work has proved to be a sound one. A major advantage is that students do not have to cope with a multitude of systems; instead they can concentrate on the essentials of their subject. An added bonus has been the arrival of low-cost PC software for related topics: control system analysis and simulation, computer-aided software engineering (CASE) tools and software metrics packages, to name but a few.

*Acknowledgements*—The authors wish to acknowledge the work carried out by Mark Morgan Lloyd to develop the communications software used in this project.

## REFERENCES

1. IEE86, The use of personal computers in control systems analysis, IEE Colloquium, 3 May 1986, London. Colloquium Digest No. 1986/83 (1986).
2. Digital Research, *Pascal/MT+ Language Programmer's Guide for the CP/M-86 Family of Operating Systems*, Digital Research Inc., Pacific Grove, CA (1983).
3. Microsoft, *Microsoft CodeView and Utilities—Software Development Tools*, Microsoft Corporation, Redmond, WA 98073 (1987).
4. K. Warwick, Further development in self-tuning control. In C. J. Harris and S. A. Billings (eds) *Self-*

*tuning and Adaptive Control: Theory and Applications*, IEE Control Engineering Series 4, 2nd edn, pp. 332–360 (1985).
5. T. Cegrell and T. Hedqvist, Successful adaptive control of paper machine, *Automatica*, **11**, 53–59 (1975).
6. F. Buccholt and M. Kummel, Self-tuning control of a pH-neutralisation process, *Automatica*, **15**, 665 (1979).
7. D. W. Clark and P. J. Gawthrop, Implementation and application of microprocessor-based self-tuners, *Automatica*, **16**, 233–244 (1980).
8. J. Kanniah *et al.*, Microprocessor-based universal regulator using dual rate sampling, *IEEE Trans. Ind. Electron.*, **IE-31** (4), 306–312 (1984).
9. G. S. Hope *et al.*, Digital implementation and test results of a self-tuning speed regulator, *Can. Elec. Engng J.*, **6** (1), 9–15 (1981).
10. A. H. Jones and B. Porter, Expert tuners for PID controllers, *Proc. IASTED Conference on Computer-aided Design and Applications*, Paris (1985).
11. L. Ljung and T. Soderstrom, *Theory and Practice of Recursive Identification*, MIT Press, Cambridge, MA (1983).
12. G. Goodwin and K. Sin, *Adaptive Filtering, Prediction and Control*, Prentice Hall, Englewood Cliffs, NJ (1984).
13. Z. M. A. Ismail, Microprocessor control of electro-mechanical actuators, Ph.D. thesis, Loughborough University of Technology (1986).
14. Intel, *iAPX 86/88, 186/188 User's Manual, Hardware Reference*, Intel Corporation, Santa Clara, CA (1985).
15. EIA, Interface between data terminal equipment and data communication equipment employing serial binary data interchange, *EIA Standard RS-RS232*, Electronics Industries Association, Engineering Department, Washington, DC 20006 (1969).
16. EIA, Electrical characteristics of balanced voltage digital interface circuits, *EIA Standard RS-RS422-A*, Electronics Industries Association, Engineering Department, Washington, DC 20006 (1978).
17. S. Bennett, *Real-time Computer Control*, Prentice Hall, London (1988).
18. B. Sanden, *Systems Programming with JSP*, Chartwell-Bratt (1985).
19. J. E. Cooling, *Software Design for Real-time Systems*, Chapman and Hall, London (1990).
20. N. Wirth, *Programming in MODULA-2*, 4th corrected edn., Springer Verlag, Berlin (1989).

**Jim Cooling** is currently a senior lecturer in the Department of Electronic and Electrical Engineering, Loughborough University of Technology, Loughborough, UK. He has specialized in the area of real-time embedded microprocessor systems for 17 years, and is a consultant to a number of major UK companies. His publications include four textbooks: *Real-time Interfacing*, *Modula-2 for Microcomputer Systems*, *Software Design for Real-time Systems*, and *Introduction to Ada* (co-author).

**Alan Whitfield**, formerly of Loughborough University of Technology, is now a senior consultant with ICL (UK) Ltd. He is concerned with information technology in general, specifically with the analysis and specification of distributed computing systems.

**Ghassan Al-Saddiki**'s present post is that of Professor in the Department of Electric and Computer Engineering, King Abdulaziz University, Kingdom of Saudi Arabia. He specialized in the area of microprocessor-based adaptive control systems while researching for his doctorate. His research and teaching interests include those of real-time embedded systems.

## APPENDIX

```
(*********************************************************)
(                 LISTING 1                          *)
(* This is the body of the Data Collection program module *)
(*********************************************************)

BEGIN

   STOP_PLANT;
   INITIALISE_CNT_DATA_INT;
   SET_UP_SERIAL_COMMS;

   GET_TEST_PARAMETERS;
   SET_UP_ANALOGUE_SYSTEM;
   COLLECT_DATA_VIA_INT;
   TRANSFER_DATA;

END. (* End of main program module *)
(*********************************************************)
```

```
(***********************************************************)
(                 LISTING 2                    *)
(* This procedure is called by the main program module *)
(***********************************************************)


PROCEDURE SET-UP_SERIAL_COMMS;
BEGIN
   INITIALISE_DEVICES;
   SEND_SIGN_ON_MESSAGE;
END;
(* end of the procedure *)
```

Procedure INITIALISE_DEVICES is implemented as follows;

```
(* This procedure initialises all CPU hardware *)
PROCEDURE INITIALISE_DEVICES;
BEGIN

   (* Initialise the Programmable Interval Timer *)
   PITCONT:=CONWORD0;
   PITCTER0:=DATALOW;
   PITCTER0:=DATAHIGH;
   PITCONT:=CONWORD2;
   PITCTER2:=DATA2LOW;
   PITCTER2:=DATA2HIGH;
   PITCONT:=CONWORD1;
```

```
    (* Initialise the UART *)
    URTCOMD:=$00;
    URTCOMD:=$00;
    URTCOMD:=$00;
    URTCOMD:=$40;
    URTCOMD:=MODE_URT;
    URTCOMD:=TRANS_INS_URT;

    (* Initialise the Programmable Interrupt Controller *)
    PIC_COM0:=ICW1;
    PIC_COM1:=ICW2;
    PIC_COM1:=ICW4;
    PIC_COM1:=OCW1;

END;
(* End of hardware initialisation procedure *)
(**************************************************************)
```