International Journal of Clinical Monitoring and Computing **10**: 147–154, 1993.

Hydra: a C-language environment for real-time DOS multitasking at the bedside

Andrea DeGaetano,[1,2] William P. Coleman,[3] Rita Pizzi,[4] Edoardo Tomasella,[4] Marco Castagneto[1,2] & Aldo V. Greco[2,5]

[1] *CNR Centro Fisiopatologia Shock e Clinica Chirurgica, Università Cattolica del Sacro Cuore, Rome, Italy;* [2] *Centro per la Modellistica dei Sistemi Fisiologici, Università Cattolica del Sacro Cuore, Rome, Italy;* [3] *MIEMMS—Shock Trauma Center and Dept. of Mathematics, University of Maryland, Baltimore, USA;* [4] *Dipartimento di Scienze dell'Informazione, Università degli Studi, Milano, Italy;* [5] *Clinica Medica, Gastroenterologia e Malattie Metaboliche, Universita' Cattolica, Rome, Italy*

Accepted 4 May 1992

Key words: microcomputers, mathematics, software, multitasking

Abstract

Patient monitoring at the bedside is an inherently parallel job, best handled by multiple individual tasks running concurrently. Cost and diffusion considerations strongly favor the use of PC's at the bedside, but their most widespread operating system, DOS, is not built for multitasking. Hence, a software platform in C language has been prepared, allowing the intermediate programmer to easily write independent modules which will then run simultaneously without conflicts.

Such a platform aims at allowing effortless sharing of data among concurrently running processes, while providing strong insulation between tasks, enough to allow multiple copies of any one task to run simultaneously unknown to each other. A cooperative, memory sharing multitasking paradigm has been chosen, which offers fine granularity of timeslicing and low execution overhead at the price of some loss in generality of design.

Speed, data exchange capability and number of stackable windows are greater than with commercial packages like Windows or LabWindows. Dynamical reprioritization of tasks is built in, allowing the computerized monitor to focus its attention and resources on urgent tasks.

Introduction

Computerized patient monitoring and instrument control necessitate an easy, inexpensive and effective computing platform in order to spread from the research laboratory to the clinical setting.

The usefulness of microcomputers in the clinical monitoring setting was recognized early [1–3], and there are more and more applications in which computing power is necessary for patient control. Several such applications have been described in the on-line respiratory monitoring of the intensive care patient [4–6], other applications have been described which greatly expand the scope of simple ECG monitoring [7, 8], such as continuous impedance cardiometry [9, 10]. Newer applications requiring additional computing power at the bedside could be renal function monitoring [11], or voice-mediated user interaction [12]. Whereas expert systems already exist in the intensive care setting [13, 14], a forthcoming necessity will be to have them running continuously on the latest patient's data.

Also, meaningful interpretation of acquired data must be preceded by computationally expensive signal conditioning [15, 16].

While massive computing power is available, it often comes at a steep price. Since Personal Computers are widely available, come at a relatively small cost, and are quickly increasing in power, they appear to be a natural choice if each bed must be equipped with a computer and if standard supporting software is still in the definition phase [17–19]. From a pure computational speed point of view, PC's are not too bad: a Toshiba 5200 (386 at 20 MHz) with a 387 coprocessor clocks at 5500 Dhrystones/sec, about the same as a VAX 785. Some practical problems have to be solved, however, before PC's can become effective tools in intensive care.

A bedside computer faces an inherently parallel job, where distinct tasks or processes have to be given attention to by the CPU: different instruments communicating at different rates, the user interface to be kept active, background computing processes like on-the-fly statistical analyses and expert system interrogations, disk updates, and other housekeeping chores. Other authors have already proposed parallel architectures for clinical monitoring systems [20, 21].

The purpose of the present work is to describe the Hydra platform developed at UCSC in Rome. This C-language software skeleton allows an intermediate level programmer to quickly create as many separate processes as he deems fit and as the hardware will tolerate, making them run on an IBM-PC compatible machine under MS-DOS. An application of this platform, currently monitoring a calorimetric chamber, is described.

Materials and methods

The Hydra multitasker is a Microsoft C large memory model template for the development of C programs consisting of several independent but freely communicating tasks.

The programmer is supplied with a header file (multitsk.h) containing the definitions of the global symbols used in the template, an object file (multirou.obj) corresponding to the compiled set of routines which handle the transitions between the pieces of code written by the programmer himself, and a manual. He then tackles each task separately as if it were an isolated program to be run alone, writing the relative code in standard C language.

In other words, if the computer programmer is able to write single small programs that execute the things he wants done one at a time, Hydra will run them simultaneously, each in its own window on the screen. If desired, the small programs (the separate tasks) will be able to call each other, exchange data and influence the behavior of one another in a very simple and powerful way. On the other hand, if insulation between tasks is desired, Hydra will make them so unaware of one another's workings that even arbitrarily many copies of the same task will run concurrently without conflict.

The separate programs are merged into a final source according to the following criteria. A task is logically divided into three components:

1) directives, pragmas and declarations;
2) a task body;
3) service routines (functions).

Directives, pragmas and declarations for all tasks are collected at the external level or at the beginning of the main ( ) function of the final merged source file. A call SETUP_MULTITASK is placed after the declarations under main ( ). All task bodies, delimited by

```
label: BEGIN-TASK;
  < statementblock >
END-TASK;
```

are then added to the merged main ( ) function. Accessory functions can be placed in the main file or in separate modules; legible coding practice suggests that most of the work be delegated from the task bodies to accessory functions.

At run-time, control is passed to the task bearing the label 'main-task'. From it, other tasks may be forked or chained as needed, by means of instructions like

```
fork (< task label >, 'task name', < task priority >);
```

Whereas chaining a child task suspends execution

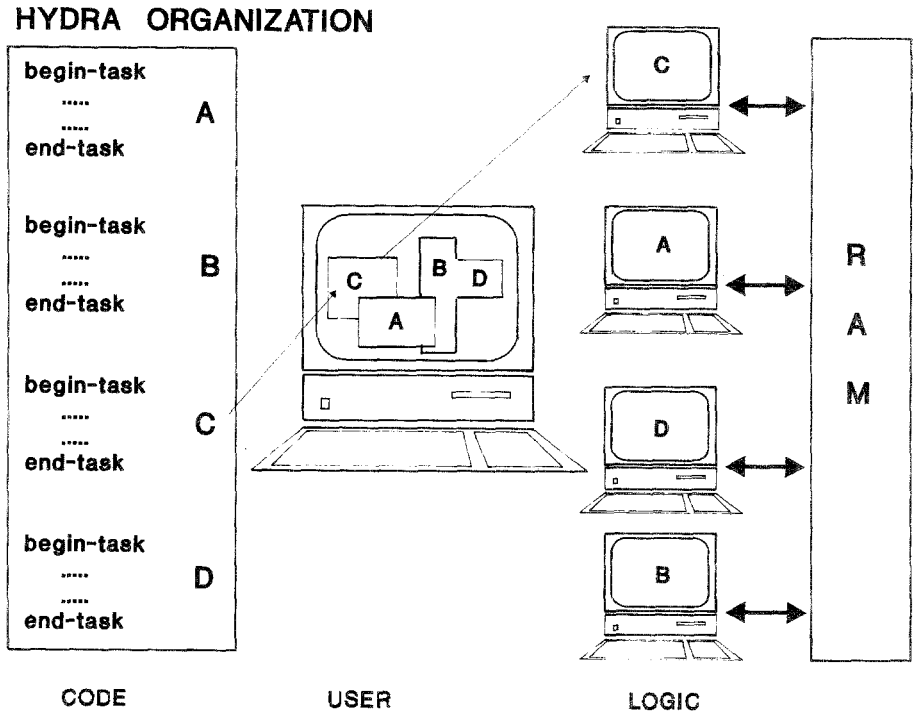

Fig. 1. To each Hydra task there corresponds a screen window and keyboard buffer, making it a separate virtual computer program. All such programs share all data in the host computer's RAM.

of the parent task until the child has terminated running, forking a child does not suspend execution of the parent.

The merged code can then be compiled and linked, together with the multirou.obj file, to produce a program in which each task runs independently in its own window.

Under Hydra, to each task is associated its own screen window (Fig. 1). Each task's window is updated independently of all others, even when it is completely or partially in the background, so that leafing through the windows on the screen brings up instantaneously the current window's contents.

To each task is also associated its own keyboard buffer and echo, so that the user talks to the foreground window. The user can leave the current input incomplete, bring some other window to the foreground, work on that, then come back later and restart from where he or she left off.

Appropriate input functions are provided, which act in a non-suspensive way: while the user completes input to a window, all the rest of the system continues working unimpeded. Windows are moved and restacked independently of any ongoing input process.

Task priorities are not fixed: supervisory tasks can be written that modify the system priorities on the basis of current variable values.

Extended memory is used to store large arrays, freeing conventional memory for counters and control variables. Appropriate functions are provided to handle transparently extended memory arrays.

At runtime multiple copies of the same task can be called and made to run concurrently and independently of each other. There is no fixed limit to the number of replicates of each task, which may be determined at runtime on the basis of the available data. This feature is useful when writing shared servers that are forked or chained by several other tasks. For instance, multiple copies of an inference engine can simultaneously roam the same rule base with different goals.

The multitasking kernel on which Hydra operates works under the cooperative and memory sharing paradigms.

As a cooperative multitasker, Hydra does not use a clock interrupt signal to stop task execution and delimit timeslices. Instead, it depends on each task generating an appropriate 'end-of-timeslice' signal, by calling the function preempt ( ). This arrangement has both advantages and disadvantages: on one hand it provides for task-defined timeslices allowing greater flexibility and easier data integrity maintenance. It also makes for faster context switching, finer granularity and smoother operation. On the other hand the whole system becomes only as robust as the worst task running under it, and task design requires some careful consideration on the part of the programmer.

Hydra is also based on a shared-memory as opposed to a message-passing architecture. In memory sharing all 'common data' among different tasks are simply declared at the external level and are indifferently accessible by everybody in the system. Such an arrangement makes condivision of information easier, but makes code encapsulation more difficult; it is more efficient and produces less overhead than a message-passing system, but it is not as easily upward scalable, does not enjoy as much generality of design and must be somewhat tailored to the application at hand.

Results

One current implementation of the Hydra multitasker is at the calorimetric chamber of the Department of Gastroenterology and Metabolic Diseases, Catholic University, Rome. This is a somewhat unusual patient monitoring application, where the object is to continuously measure metabolism, movement and heart rate in order to estimate real life energy expenditure and related factors. However, there is no difference in either concept or software between this application and a more traditional intensive care situation: all that would vary would be the type of some of the connected instruments. In the following description, the overall metabolic monitoring job is split into the several different concurrent tasks composing it. The relationships between some of these tasks are also sketched.

One of the several tasks running in this application talks to the user and modifies accordingly the computing environment (length of sampling brackets, depth of data smoothing, measured urinary Nitrogen Excretion, frequency of disk data saves, and other similar keyboard-input information). Another task polls the 16 channel A/D converter, to which O_2 and CO_2 concentrations within the chamber, flow through the chamber, patient's heart rate, radar output, temperature, humidity, pressure and other signals are brought; this task also stores smoothed data onto a large circular buffer. A metabolic computations task sleeps the allotted period of time, then wakes up, performs the computations, displays the results, stores them on disk, clears the buffer of the used data and goes back to sleep. There is a task which monitors a treadmill through an RS232 port and which records and shows speed, miles run, and slope. A radar motion detector task is being implemented, relying on a simple neural network simulator to gauge the amount of actual patient movement from the output of two ultrasound detectors; for the moment this task only adds the motion signals for the two detectors and averages them out relative to the chosen time period. There are also a system clock task and a fleeting (self-terminating) instrument calibration task. Invisible to the user, there is the windower task (part of the kernel).

Figure 2 shows the time tracing of some of the measured and on-line computed variables from an obese diabetic patient examined in the calorimetric chamber. It must be appreciated how the derived parameters of interest, like metabolic rate, are computed using information coming in from different instruments (flowmeter, oxymeter, temperature and humidity probes) as well as from the keyboard (Nitrogen Excretion). This integration of different sources of information is as easy as writing the formulae including the variables of interest, provided that the multitasking environment is in place.

Now that the skeleton works reliably, adding further tasks is very easy: a small library of them is being built and they can be plugged effortlessly in the main module. A case in point is the Siemens 900C Servoventilator, which outputs sixteen differ-

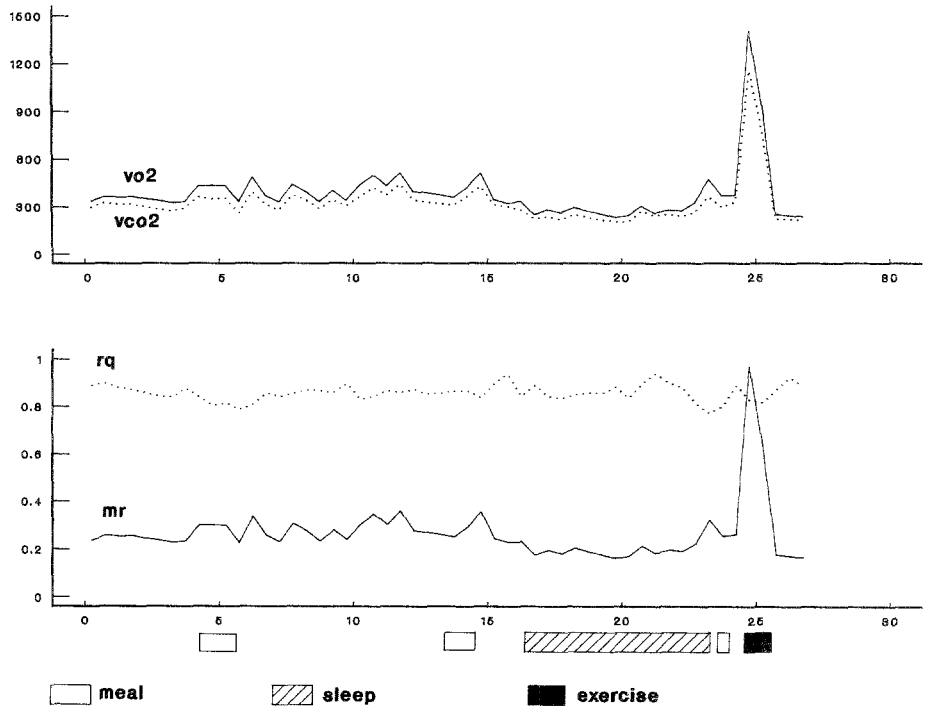

Fig. 2. Time tracings for a 33 yr old male type 1 diabetic patient studied in the Calorimetric Chamber. Time is expressed in hours from beginning of the experiment (at 8:00 A.M.). VO_2 and VCO_2 are expressed in ml/min. RQ is the respiratory quotient (VO_2/VCO_2), which varies from 0.7 (when the subject burns only lipids) to 1.0 (when the subject burns only carbohydrates). Metabolic Rate (mr: Kcal/24 hr) has been divided by 10,000 in order to superimpose its graph to that of RQ. Each graphed point is the average value of half an hour recording at five samples per second.

ent pieces of information through analog 0-10 Volt lines (Table 1): with the A/D board already in place, adding a ventilator task is a matter of merely declaring to the system the meaning of the sixteen pieces of information and the rate at which the user wants them polled from the ventilator.

Another example of the kind of problems which Hydra solves is the following: the physicians in our team perform euglycemic hyperinsulinemic clamp studies, where high doses of insulin are infused to stop hepatic gluconeogenesis and a continuous infusion of glucose must be maintained at varying rates in order to keep glycemia constant. A simple BASIC program was used to perform the clamp computations for the physician, indicating the necessary rate of glucose infusion for the next five minutes, given the target glycemia, current glucose infusion rate, size of the patient, etc. A decision was then made to associate metabolic monitoring, with a Deltatrac Metabolic Cart (Datex, Finland), to the clamp study. Hydra runs the clamp computations in a window, automatically acquires data from the Deltatrac in another window, corrects raw metabo-

Table 1. Data available on-line from the Siemens 900C Servoventilator as 0–10 V analog signals.

Airway flow
Tidal volume (inspired)
Tidal volume (expired)
Airway pressure (waveform)
Airway pressure (peak)
Airway pressure (mean)
FiO_2
Respiratory rate (set)
Respiratory rate (real)
SIMV rate (set)
Expiratory time
Inspiratory time
Pause pressure
Expired minute volume
PEEP level (set)

lic measurements in a third window, and time-stamps and saves together the clamp and (corrected) metabolic data in a fourth. In a fifth window the user is invited to express possible dissatisfaction with the current state of things, so that corrective measures can be taken.

With the above architecture running an on IBM PS2 System 80 computer (386 at 20 MHz, 6 MB RAM, 120 MB HD, no coprocessor), more than 200 task switches per second have been observed.

Discussion

A sizeable amount of effort has been expended in making it possible for a PC to perform in a reasonably efficient way as a parallel controller for bedside operation under the standard MS-DOS operating system. The rationale is that in most institutions big workstations are not cost-effective, and that PC's perform acceptably many functions with widespread cheap software. The same computer used to monitor a critical patient for a few days can then be used to draw diagrams, analyze historical data, do word processing or develop some more software.

Whereas the described windowing behaviour is more artfully implemented in commercial multitasking packages like Microsoft Windows or Quarterdeck Desqview 386, Hydra adds painless data condivision among tasks and dynamic, software controlled task reprioritization: for instance, the user interface can be slowed down in favor of instrument control processes when no keypress has been detected for ten seconds, to be accelerated again at the first keypress. A clinically more interesting example is that in which a physiologic abnormality in one organ system, say circulation, is detected; Hydra then increases the priority allocated to expert system interrogations referring to that specific organ system. This feature provides an analogue for attention focusing in information processing [22].

Faster PC's, like 486's, can run this system faster,

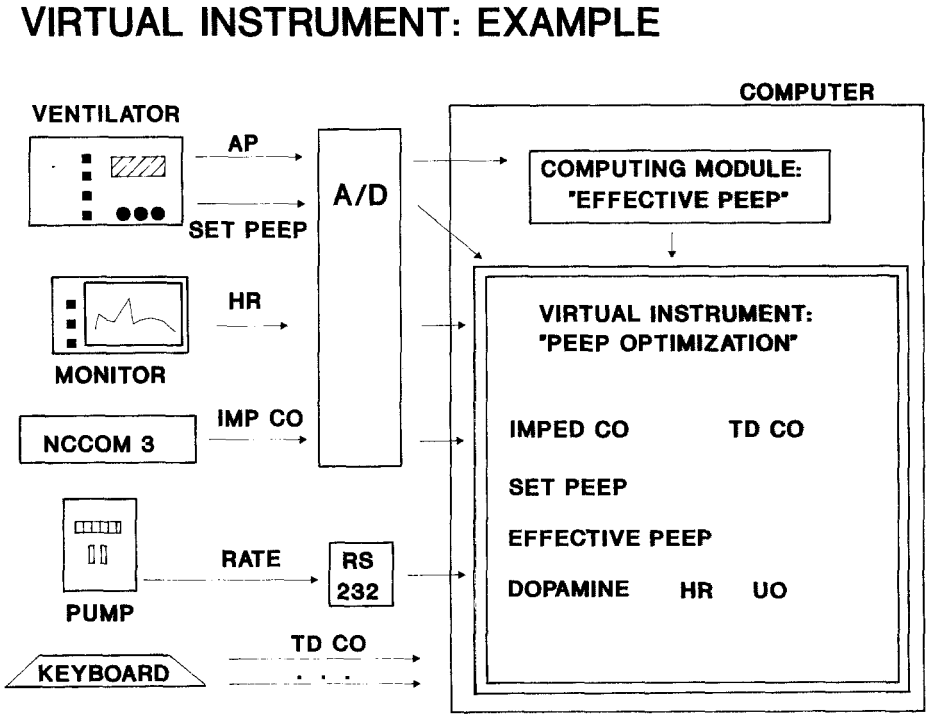

Fig. 3. Data items from several physical instruments and from the keyboard, after possible processing, are represented in the window dedicated to the control of one physiologic situation, realizing a computerized Virtual Instrument monitoring that physiologic situation.

and a physician's real time is usually slow enough to obviate the need of a dedicated, high performance workstation. What is more important, the availability of programmers versed in the standard PC languages is far greater than that of programmers able to quickly produce code for workstations, and software customization is thereby greatly facilitated.

Using the memory sharing parallel features of this software it is very easy to set up what may be called a 'Virtual Instrument' to operate at the patient's bedside (Fig. 3). A Virtual Instrument looks to the operator like a single isolated instrument, its frontpiece consisting of one window on the screen on which switches and dials are present. In the background, the Virtual Instrument task collects from shared memory whatever ingredients it needs, causes dependent tasks to be activated if necessary and displays as a finished product the computed quantities. In the foreground, it prompts for and accepts instrument-specific input.

Commercial software packages for process control, like LabWindows or Asyst, also provide tools for building virtual instruments, with much better graphical displays and semi-automated connection to vendor-specific hardware (which, however, does not eliminate the need to write programs). On the other hand, Hydra allows stackable instruments (more instruments than can fit in a single screen), native C programming, dynamically reprioritizable concurrent execution and multiple simultaneous copies of servers.

A Virtual Instrument in the anesthesiologic setting could be, for instance, a collection of computing routines specifically designed for PEEP optimization in the ventilated patient. This optimization could be brought about by recording simultaneous Cardiac Index and Effective PEEP values from an impedance cardiometer and the ventilator respectively, and by periodically calling a numerical fitting routine to find the point at which an increase of PEEP has relatively larger negative effects on CI.

In our application, such a Virtual Instrument is the Metabolic Computer, which updates in real time the fuel mix and rate of metabolism of the patient, accepting such keyboard inputs as the daily average nitrogen elimination and finding, in shared memory, the necessary chamber gas concentration and flow data.

In the preparation of this software, a deliberate course of action has been taken, minimizing graphical niceties and nonessential components in order to produce as fast a system as possible. The decision to start from native C code instead of building over a commercially available platform stems from this requirement. Substantial computing power is now needed at the bedside of the critically ill patient, and this need will increase in the near future. A trimmed-down, lean, efficient software can enable PC's to economically provide such computing power.

References

1. East TD. Microcomputer data acquisition and control. Int J Clin Monit Comput 1986; 3: 225–38.
2. Stoodley KD, Crew AD. Lu R et al. A microcomputer implementation of status and alarm algorithms in a cardiac surgical intensive care unit. Int J Clin Monit Comput 1987; 4: 115–22.
3. Farrell AP, Bruce F. Data acquisition and analysis of pulsatile signals using a personal computer: an application in cardiovascular physiology. Comput Biol Med 1987; 17: 151–9.
4. Chambrin MC, Ravaux P, Chopin C et al. Computer-assisted evaluation of respiratory data in ventilated critically ill patients. Int J Clin Monit Comput 1989; 6: 211–5.
5. Jenkins JS, Valcke CP, Ward DS. A programmable system for acquisition and reduction of respiratory physiological data. Ann Biomed Eng 1989; 17: 93–108.
6. Rudowski R, Skreta L, Baehrendtz S et al. Lung function analysis and optimization during artificial ventilation. A personal computer-based system. Comput Methods Programs Biomed 1990; 31: 33–42.
7. Kamath MV, Fallen EL, Ghista DN. Microcomputerized on-line evaluation of heart rate variability power spectra in humans. Comput Biol Med 1988; 18: 165–71.
8. Pinciroli F, Pellegrini A, Falcetti G et al. Electrocardiomultigraphimeter using a home computer. Comput Methods Programs Biomed 1988; 26: 1–10.
9. Jossinet J, Leftheriotis G, Vernier F et al. A computerized bioelectrical cardiac monitor. Comput Biol Med 1990; 20: 253–60.
10. Wang XA, Sun HH, Adamson D et al. An impedance cardiography system: a new design. Ann Biomed Eng 1989; 17: 535–56.
11. Koning HM, Mackie DP. Is on-line monitoring of renal function possible? Int J Clin Monit Comput 1989; 6: 243–6.

12. McMillan PJ, Harris JG. Data Voice: a microcomputer-based general purpose voice-controlled data-collection system. Comput Biol Med 1990; 20: 415–9.
13. van den Heuvel J, Stemerdink JD, Bogers AJ et al. GUUS an expert system in the intensive care unit. Int J Clin Monit Comput 1990; 7: 171–5.
14. Winkel P. A programming language and a system for automated time- and laboratory test level dependent decision-making during patient monitoring. Comput Biomed Res 1990; 23: 426–46.
15. Ciarlini P, Barone P. A recursive algorithm to compute the baseline drift in recorded biological signals. Comput Biomed Res 1988; 21: 221–6.
16. Sittig DF, Factor M. Physiologic trend detection and artifact rejection: a parallel implementation of a multi-state Kalman filtering algorithm. Comput Methods Programs Biomed 1990; 31: 1–10.
17. Cesarelli M, Clemente F, Bracale M. A flexible FFT algorithm for processing biomedical signals using a personal computer. J Biomed Eng 1990; 12: 527–30.
18. Mustard RA, Cosolo A, Fisher J et al. PC-based system for collection and analysis of physiological data. Comput Biol Med 1990; 20: 65–74.
19. Petrini MF, Dwyer TM, Wall MA et al. Communication between the PC and laboratory instruments. Comput Appl Biosci 1990; 6: 161–4.
20. Westdijk JA, van Alste JA, Schoute AL. Multi-tasking control system for real-time processing of biomedical signals. Comput Methods Programs Biomed 1988; 26: 153–8.
21. Factor M, Sittig DF, Cohn AI et al. A parallel software architecture for building intelligent medical monitors. Int J Clin Monit Comput 1990; 7: 117–28.
22. Hayes-Roth B, Washington R, Hewett R et al. Intelligent monitoring and control. Proceedings IJCAI 89 1989, Detroit, 243–9.

Address for offprints:
A. De Gaetano,
CNR Centro Fisiopatologia Shock,
Università Cattolica del Sacro Cuore,
L.go Gemelli,
8 – 00168 Rome, Italy