Timesharing

The use of microprocessors in a timeshared computer facility for research, development, and teaching

GREGG T. VESONDER, ALAN M. LESGOLD, and RICHARD V. WOLF

Learning Research and Development Center, University of Pittsburgh, Pittsburgh, Pennsylvania 15260

The use of microprocessors in a timeshared computer facility at the Learning Research and Development Center, University of Pittsburgh, is described for applications in research, development, and teaching. Modes of usage are described, as are the advantages of microprocessor interfaces for on-line experimentation.

This paper explores some of the uses to which microprocessors might be put in the evolution of a medium-scale computer system that handles a mixture of teaching, research, and development tasks in a timesharing environment. The particular system that is the basis of our planning is the Experimental Time-Sharing System (ETSS), which has been extensively described in previous papers (Fitzhugh & Glaser, 1975; Lesgold & Fitzhugh, 1977; Fitzhugh, Note 1). This system supports a wide variety of tasks simultaneously, handling most of the timesharing and experiment-control needs of the Learning Research and Development Center (LRDC). However, like every successful system, it is becoming increasingly popular and is now heavily burdened. We are attempting to decide how to deal with our increased load and are particularly interested in the possibility that some of the load, as well as some equipment interfacing, might be more economically handled by microprocessors.

As it operates now, the system has 15-20 simultaneous users during peak business hours. This mixture, by late fall 1977, included users of a classroom information management system, an accounting system for our Center, a small computer-assisted instruction venture, and several on-line experiments. In addition, program development, routine file management, and data analysis needs are always present. Such a load can be managed partially by priority queuing (Fitzhugh & Glaser, 1975) and partially by restrictive scheduling of uses that place a heavy load on the system or that require rapid response from it. However, we are now faced with both a decrease in system responsiveness (due to increased usage) and demands by experimenters for more complex interfaces between experimental apparatus and the computer.

The use of microprocessors in interfacing to experimental apparatus has been envisioned for some time (Fitzhugh, 1974; Lesgold & Fitzhugh, 1977), and our approach to the interfacing problem within ETSS permits easy expansion in this direction. In the past, we have constructed interface devices that communicate to and from the computer with serial ASCII-character

messages that are identical in form to those that a standard terminal sends and receives. These devices are controlled via standard FORTRAN programs that read their responses and write commands to them with conventional READ and WRITE statements. Thus, our main computer system requires no special interface to operate with these devices, and they, in turn, are independent of the specifics of our host system. The devices turn apparatus on and off, sense apparatus performance states, and accept subject responses. For example, one of our devices can be commanded to advance a slide projector, start a clock when a photocell detects a new slide on the screen, and then stop the clock and report both the response and the response latency when the subject presses one of several keys.

These interface devices, combined with a small number of special programming techniques (see Lesgold & Fitzhugh, 1977) have served us very well. However, they have triggered an interest in even more complex patterns of stimulus and response events, which are difficult to control with a timesharing system that drives nonintelligent devices. The basic problem is the latency between an experiment event and a response from the system. The system, as currently functioning, logs all I/O times at the interrupt level, so response timing measurements are accurate. Further execution is priority queued, and experiments begin to compete for time when several experiments with complex timing demands are running simultaneously. The result is variance in stimulus pacing. One way to deal with this problem is to put the control of complex, high-speed activity sequences in an intelligent interface device. Increasingly, the optimal way to do this is with a microprocessor rather than a cluster of independent logic hardware.

ADVANTAGES OF MICROPROCESSOR INTERFACES FOR ON-LINE EXPERIMENTATION

The three primary advantages that result from a switch to microprocessor interfaces at LRDC are improved pacing of experiments, more flexible on-line

laboratories, and a reduction in the load that on-line experiments place on our system. All of these changes result from the fact that the pacing of experiments under microprocessor control is controlled via decisions made by executing programs rather than by discrete logic. Because the memory and processor necessary to execute a software-encoded set of experimental conditions is cheaper than the discrete logic needed to do the same job with hardware, more complex control can be placed in the interface. Since the microprocessor that provides this control is not timeshared, it can respond more rapidly in the real-time experiment environment than our timeshared main system can. This provides a relatively cost-effective solution to the timing difficulties that arise when usage of the timesharing main system increases. Because the microprocessor program is more readily alterable than is discrete logic, a single microprocessor-controlled interface can be put to several different uses without expensive rewiring. Thus, the change from one experiment that uses a slide projector to another that uses it differently is merely a change of program, not the design and implementation of new discrete logic. This is particularly crucial in experiments that require extensive pilot testing before exact conditions are determined. Of course, adjustable hardware is also possible, but it is expensive, more difficult to work with, and less flexible.

These advantages are illustrated with an example from current work. A number of discourse processing experiments at LRDC have sequences of two or three consecutive responses from the subject. Further, we need to have the stimulus materials change immediately after each response is made. The current system accurately produces stimuli with complex timing, but precision is possible only with short, ballistic sequences. That is, each new set of decisions that must be made in the midst of a trial requires additional execution of the experimenter's program by the CPU. These accesses currently take an average of .3 to .4 sec; the worst cases take almost 2 sec. Thus, even though we can provide accurate time records of stimuli and responses and produce short (1-3 sec) but accurate noncontingent stimulus timing, we are less able to produce response-contingent stimulus sequences that are precisely timed.

The solution is to use an interface that contains enough memory to be preloaded with the information needed for one trial. A trial is a segment of an experiment within which the times of various events must be precisely synchronized but which can be relatively asynchronous with respect to other such segments. Thus, the main computer system would provide the large memories needed for text stimuli and other complex pattern sequences and would collect the experimental data. The interface device would be sent enough information for a trial and would return the data for that trial when it was completed. The only reasonable way to construct such a device is with a microprocessor

that can be programmed with the required decision sequence.

A related case in which such a microprocessor-based interface device is useful involves interactions in which the subject produces continuous data, as in tracking tasks, for example. In order to do a tracking task study, one user of a timesharing system effectively has to lock everyone else out if detailed analyses of tracking performance (cf. Wickens & Gopher, 1977) are to be attempted. A microprocessor can easily and cheaply be interfaced to a joystick (a continuously moveable lever that can generate two-dimensional coordinates) or other response device and can, with a modest amount of memory, reduce and transmit tracking data to the host computer without heavily loading it.

The current cost of such microprocessor interfacing is about \$500-\$1,500 for components, which is about the same as our present nonintelligent devices cost. As a comparison, a simple hardware interface for laboratory equipment can easily cost \$3,000 assembled.

MODES OF USAGE

Microprocessors can play a number of different roles in a computer-based laboratory. These different modes of usage can be classified in terms of the relationships between the microprocessor and the host computer (in our case, ETSS). At one extreme there is the host computer without microprocessor involvement. At the other extreme there is a stand-alone microprocessor (no host computer) servicing a few users in a timesharing environment. The purpose of this section is to examine several points of the continuum and to relate our current plans to it.

Our past implementations, as described above, involved nonintelligent interface devices controlled by the main computer with limited capacity for storing commands received as character strings from the main computer and intercepting data from a terminal or other response devices for later transfer to the main computer. These customized I/O devices were fine solutions to the problems mentioned above. They surely decreased apparatus response time through temporary storage capacity and provided excellent interfaces to non-standard I/O devices.

A paradox developed in our uses of these nonintelligent devices, however. The specialized instrumentation for stimulus control also afforded a rather uncomfortable degree of inflexibility. If another experimental application was desired, a new device with a new design was required. The more complex the experiment, the more complex the hardware design. Designing and building a new piece of electronic hardware for each new experimental paradigm was a nontrivial task, and the use of discrete logic did not seem to be as cost-effective as we had once hoped. An alternative was necessary.

The alternative is an intelligent interface device

whose main component is a microprocessor (hereafter referred to as an m-device). The advantage of the m-device is readily apparent. Internal logic can be altered through software control; an m-device for one application can be altered simply by changing its internal program. Also, the m-device can handle more of the decision-making load (dependent upon the extent of its read-write memory) and can thus provide more responsive computation to the experimenter, regardless of the host system load.

Our m-devices fall into two categories, dependent upon how the program controlling the execution is stored. The two types differ in the type of memory used to store programs. One type is a semiconductor read-write memory. (It has most of the properties of large computer memories, though it is more volatile. Some read-write memories require continual electronic refreshing and thus are sensitive to power loss.) The other type of memory used for program storage is the read-only memory (often called ROM). The two basic types of read-only memories are erasable and permanent. Erasable memories are somewhat more expensive, though future price reductions may make this less of a concern. Erasure is effected via ultraviolet light. Read-only memories can be programmed with a device that is a simple component of our microprocessor development station.

The simpler m-devices we plan will have programs on read-only memories. However, they will also need read-write memories for storage of intermediate results and to accommodate minor additions or options to the semi-permanent program. The amounts of read-only and read-write memory needed for a particular application will be determined during preliminary development of the m-device on a microprocessor development station (to be described below). For fixed applications, the m-devices will be at least one step ahead of nonintelligent devices, because new applications for the same peripheral equipment can be implemented simply by reprogramming or replacing the read-only memory.

Read-write memory for program storage offers greater flexibility, via a different mode of operation. With read-write memory, the program is acquired by downloading, which refers to the process of having the host computer system transmit a precompiled program into the m-device's memory. After downloading, the m-device is immediately ready for a new experiment. Therefore, at the end of an experimental session, the m-device can transfer the collected data to the main computer, which can, in turn, download a program for a different experiment into the m-device. Even when read-write memory is used for storage of m-device programs, it may still be desirable for certain subroutines to be put on read-only memories. For example, if the routines that interact with the host computer are on read-only memories, the host is protected from situations, such as

too rapid data flow, that it may have difficulty handling without disruption to other timesharing users.

During performance of an experimental trial, m-devices operate autonomously; stimulus response sequences are timed, appropriate stimulus contingencies dealt with, and data preprocessed and reduced. In many applications, the host computer system can be used only for downloading of a program at the beginning of an experiment and receiving data for bulk storage and analysis at the end. Thus, m-devices would act as satellite processors handling a variety of real-time experimental applications.

At the far end of the continuum mentioned above, it is conceivable to design an independent microcomputer system. For such a microsystem, the host computer is used mainly for the development of the program to be run and for the manipulation of the data that has been collected. Even program development can take place in the system. It is highly feasible that in such a stand-alone microsystem, bulk data storage devices (diskettes, bubble memories, etc.) can hold the data from several subjects. The data can then be transferred to the host computer for analysis at a later time. Even with such an extensive "peripheral" system, there may be need for development work on a different type of microcomputer, the m-device development station described in the next section.

THE MICROPROCESSOR DEVELOPMENT STATION

The preceding sections described a proposed system and the rationale for developing microprocessor-based experimental stations. Because a number of these stations are planned, we plan to concentrate design and debugging of the facilities for particular experiments in a central microprocessor development station. This section outlines the development station and explores its potential in developing m-devices of various sorts.

The microprocessor development station may be arbitrarily segmented into several sections: (1) the microprocessor, (2) utility programs on read-only memories, (3) read-write memory, (4) interface hardware and software for peripheral devices, (5) interface hardware and software for interaction with the host computer system, (6) a real-time clock with associated hardware and software to permit complex response timing and stimulus pacing, and (7) mass storage devices. Read-only memory containing service routines such as standard I/O routines, elementary editing capabilities, and necessary monitor commands (e.g., the capability to examine the contents of specified registers and of specified locations in memory) are currently available. Since the amount of read-write memory available is the limiting factor in the size of programs that can be run by the development

system, it seems sensible that it have as much memory as possible (e.g., at least 32K bytes).

The I/O capabilities of the development station will be standardized in two basic types of circuitry attached to the central bus (communication channel) of the microprocessor. One type of I/O circuitry will communicate with the host computer and with terminals. This type of communication is serial and asynchronous. That is, one bit is sent or received at a time, according to a timing scheme that is not strongly tied to the internal timing of the microprocessor. Our development station has two such asynchronous interfaces at the moment, one for communication with the host and one for the experimenter's interactions with the m-device. We may add a third, to facilitate applications in which the m-device needs to talk to the experimenter, the subject, and the host computer. The second type of I/O circuitry is a parallel interface, which can transmit the entire 8-bit content of a memory location at one time. Parallel interfaces of standard design are being built to permit standard connection of the station with slide projectors, tape recorders, bells, buzzers, voice keys, etc. There will be as many parallel data ports as are needed by the most complex lab set-ups currently envisioned.

Each I/O port is based upon a large-scale integrated circuit chip. Each such chip appears to the microprocessor as a few bytes of memory on our system (other systems have separate I/O instructions). Consequently, lab implementations can operate with only as many interface adapters as are needed for the particular application, just as they have only the amount of memory that they actually need.

We are currently in the process of designing a standard clock hardware arrangement for timing both stimulus presentations and response latencies. We plan to implement a standard clock arrangement that is accessed in the same manner as all other devices. In its simplest form, a clock can be set up to interrupt the microprocessor at the end of every time quantum (say, 100 microsec). It is then incumbent upon the software to figure out what to do on each clock tick. Alternatively, one can add to the clock circuitry the possibility that the software could specify the number of quanta that should elapse before the next interrupt. This greatly simplifies stimulus timing. It is also possible, though somewhat more cumbersome and expensive, to have a clock that can be read by the microprocessor.

One area in which we have not made final decisions is the question of mass storage devices. It is hoped that ETSS will provide most of the mass storage needs for the m-devices and the development station. However, there are two applications of m-devices that may necessitate some sort of local, self-contained, mass storage capability. The development station should also have this capability. One m-device that might need mass storage is a facility for host-independent conduct of an entire experimental session (not just one trial). If an

experiment entails the presentation of large amounts of text (as many of ours do), it may be desirable for the m-device to have access to a fast-transfer mass storage device that would store this text. Ordinarily, text will be transmitted on a trial-by-trial basis from ETSS, but if a constant, precise intertrial interval is needed, a heavy timesharing load could cause some undesirable delays. A diskette system (Shugart type) is appropriate for this application. (Diskettes are magnetic storage devices similar in appearance to a soft phonograph record, and are sometimes called floppy disks.) With appropriate interfacing hardware and software, a Shugart single-disk system can be implemented for under \$1,000.

Another type of host-independent m-device can be transported to schools and serve as a remote experimental station. Although the system can be downloaded over phone lines, it may be more desirable for it to be completely independent of ETSS during all phases of its operation. This can be facilitated by the interfacing of a dual-diskette system. The diskettes enable the system to have primitive (in present-day terms) file-handling capabilities. This permits the storage, in different blocks, of software for the experiment, of stimuli that are presented during the experiment, and of multisession data. A program, or a small operating system, is first loaded into the microcomputer from a diskette via a bootstrapping program on read-only memory. From then on, diskette-stored programs are overlaid into a read-write memory as needed. At the end of the day, a diskette containing the day's data can be transported to the development station and loaded into ETSS for permanent storage and more extensive data reduction. That is, the development station doubles as a diskette interface for the host computer.

Other peripherals being considered include: (1) a CRT for limited graphics, (2) analog-to-digital and digital-to-analog converters, and (3) a circuit for programming of read-only memories (often called a PROM burner). Some of these peripherals are attached directly to the development station at the outset (e.g., the read-only memory programmer). Others will be added when needed.

The microprocessor development station is attached to ETSS through a high-speed terminal port and looks, to the host computer, like any other timesharing terminal. Note that the development station may conceivably have more than one channel open to ETSS. This enables speedy transfer of instructions and data either way and affords the development station the capability to operate at the varying data transmission rates required to appropriately simulate the rates that different m-devices under development have to contend with. For example, a "stand-alone" system in a school may have a 30-character/sec transmission rate; a device in our laboratories may have a 480-character/sec line to ETSS.

This section has been primarily concerned with the

hardware associated with the development station. The next section outlines our software plans for the proposed microprocessor system.

SOFTWARE SUPPORT: INTERACTION OF THE DEVELOPMENT STATION WITH ETSS

Crucial to the usage of microprocessors is the ease of software development [see Toong's (1977) explanation of the "Triad" system]. Many effective uses of microprocessors at LRDC rest upon the capability of using the host computer to develop the programs necessary to run the microprocessors. This means that the microprocessor programs are not written using the microprocessors. Rather, they are written using the more powerful host computer. This is one way to increase the versatility of m-device implementations. There are several rather immediate software needs that we are pursuing in our microprocessor efforts. A cross-assembler, a variety of utility routines, and possibly a higher level language and/or a simulator are needed.

A cross-assembler is a program for the host computer that takes programs written in a mnemonic language (i.e., one in which a verbal code directly corresponds to each machine-language instruction) and translates them into machine language. There are several reasons for doing this on the host computer rather than on the microcomputer, even though it can be done either way. The biggest advantage of a cross-assembler is that the memory limitations of even large microsystems are more severe than on the host device. Further, the host computer is generally equipped with relatively high-speed bulk storage such as disk drives. Thus, the source program can be quickly accessed and the object program quickly written; also, the text editors available on the host machine are likely to be more convenient, so program input is less cumbersome.

Related to these concerns is another potential advantage of assembly on the host computer. Given that the microprocessor must currently be programmed in a low-level language, standard utility routines are particularly valuable. It is convenient to store routines that are regularly used in microcomputer programs in a library that can be accessed by the assembler. Then, these routines can be inserted into particular programs as macroinstructions or subroutine calls (a macroinstruction is one that is expanded by the assembler into a sequence of several machine-language instructions). We are developing several such utility routines including: standard routines for interaction between the host and the microprocessor, arithmetic and scientific routines, basic experiment-control routines, and routines for various timing needs.

A third area for software development is a higher level language. In order to make m-device-based experiments easily and completely accessible to psychologists, it is necessary to eventually have a higher level language

in which control programs can be written easily. At present, there is a strong preference among most microprocessor users for either assembly languages or BASIC-type languages. This is because programs written at that level tend to more efficiently use memory, and memory is an important component of the total cost of an m-device. However, memory is continually getting cheaper, and software production costs are very high. As microprocessor usage expands from the domains of hobbyists and mass-produced specialized control devices, higher level languages should become more prevalent. With the basic computational routines in prepackaged form (as discussed above), memory loss for higher level programs that make good use of these utilities will not be excessive.

The fourth possibility for software development is an m-device simulator that runs on the host computer and permits software development independent of the microprocessor development station. Initially, we need to develop experience with the aspects of m-device performance that need to be simulated (e.g., timing), but eventually such a simulator may be useful.

A HYPOTHETICAL M-DEVICE SCENARIO

It may be helpful to run through a hypothetical example of m-device development from initial conceptualization to final implementation.

The experimenter first communicates the design of the experiment to the computer facility staff, indicating the desired stimuli, response domain, contingencies, and timing patterns required. At this time the experimenter and the computer facility staff jointly make some decisions concerning the peripherals, such as slide projectors, voice keys, buttons, etc., required. The computer staff then codes a support program in standard FORTRAN for the host computer (ETSS) and constructs a control program (currently, in assembly language) for the m-device.

The m-device control program contains all the code necessary to control the critical time intervals, display stimuli, preprocess response data, etc. The support program serves a supervisory role in the experiment by creating data files, accessing stimulus files on ETSS, and otherwise serving as the intermediary between the m-device and the host's bulk storage facilities. The host program and the m-device program are then tested on the development station to ensure that the appropriate sequencing of events is occurring. If any nonstandard peripherals are required, they are attached or simulated on the development station. The result of this debugging and testing process is a set of working programs and information about the level of m-device needed for the experiment. An m-device meeting the experiment's needs is assigned or built at this point. Next, the m-device is downloaded (or its read-only memory programmed) with the already tested programs and tested

itself. Testing is extensive to insure the reliability of m-device and program operation and interaction with the host computer and peripheral devices. The development station is used not only for testing and debugging the m-device control program but also for programming read-only memory chips for the m-device.

During the running of the experiment, the experimenter logs into ETSS via the m-device. Then, if the m-device must be downloaded, a utility program located on ETSS can be used by the experimenter to transfer a machine-language control program from an ETSS file to the m-device. The experimenter then activates the support program and tests the subject. After the experiment is run, another experimenter may walk into the lab, attach the appropriate peripherals for his/her application, and then repeat the initializing sequence for the next experiment. During the experiments, the m-device performs the critical timing and stimulus functions and interacts with the ETSS through the support program for data and stimulus transfer.

COST OF THE DEVELOPMENT SYSTEM AND M-DEVICES

Our development system is based on the American Microsystems, Inc. EVK-series microcomputer system, which uses an M6800 processor. It contains 32K of read-write memory and 2K of read-only memory that contains a primitive monitor and utility routines. It also has a built-in read-only memory (PROM) burner, a prototyping board, serial and parallel I/O ports, and a clock. The current cost of the development station is approximately \$1,500 (excluding labor). If the development station is to mimic the operation of a stand-alone microcomputer system (and also provide for the transfer of its data to the main computer), then the development station should also have a dual-diskette system, which costs approximately \$1,500-\$2,000.

The simplest m-devices, containing a few thousand bytes of read-write memory, cost slightly under \$500. A major portion of this cost involves a standard packaging scheme that insures the device can be expanded as the need arises. The \$500-\$1,000 range covers most applications requiring greater complexity. The cost of m-devices beyond this level depends upon the amount of read-write memory required, the number of interface connections with peripherals, and the necessity for local, mass storage devices.

Through the extensive use of both ETSS and our development station, we will have the capability to design intelligent devices to perform complex experiments requiring precise timing. We will have this expanded capability with little additional cost and minimal effort through the judicious use of our existing ETSS resources and a well-planned development station.

REFERENCE NOTE

1. Fitzhugh, R. J. *A general-purpose timesharing system for a small- or medium-scale computer*. Pittsburgh, Penn: University of Pittsburgh, Learning Research and Development Center, 1973.

REFERENCES

- FITZHUGH, R. J. Laboratory control with a medium-scale time-sharing system. *Behavior Research Methods & Instrumentation*, 1974, **6**, 131-137.
- FITZHUGH, R. J., & GLASER, R. A general-purpose computer system for a research and development center. *American Psychologist*, 1975, **30**, 219-225.
- LESGOLD, A. M., & FITZHUGH, R. J. An on-line psychological laboratory for teaching and research. *Behavior Research Methods & Instrumentation*, 1977, **9**, 184-188.
- TOONG, H.-M. D. Microprocessors. *Scientific American*, 1977, **237**(3), 146-161.
- WICKENS, C. D., & GOPHER, D. Control theory measures of tracking as indices of attention allocation strategies. *Human Factors*, 1977, **19**, 349-365.

(Received for publication November 8, 1977;
accepted November 9, 1977.)