SESSION XI

MICROCOMPUTERS—WHAT THEY ARE AND HOW TO USE THEM: A TUTORIAL

CARL M. BRANDAUER, *Bell Laboratories, Presider*

Microprocessor uses in a timeshared facility

JAMES L. FOX, JR., RICHARD V. WOLF, and ALAN M. LESGOLD
Learning Research and Development Center, University of Pittsburgh
Pittsburgh, Pennsylvania 15260

A number of topics are discussed that are related to developing computer-controlled laboratories in which microprocessors and host machines interact. Experiences are derived from work at the Learning Research and Development Center, University of Pittsburgh.

In a recent article, members of the University of Pittsburgh's Learning Research and Development Center (LRDC) computer facility (Vesonder, Lesgold, & Wolf, 1978) presented a discussion of our plans for incorporating microcomputers into our laboratory computer system. That article was written 1 year ago, and we are now able to report a number of our experiences in effecting this plan. In the course of the year, we have learned a great deal about the service a microcomputer can provide when used as the peripheral intelligence of a timeshared computer system. We have learned the importance of using the host computer when designing programs for a peripheral microprocessor and also to take advantage of microprocessors as a means of taking some of the load off our central system (described in detail in Fitzhugh & Glazer, 1975; Lesgold & Fitzhugh, 1977; it is a PDP-15-based system that has been customized to behave like a large timesharing minicomputer). In this discussion, we present some of our current tactics and procedures that might be of value to others who plan on developing laboratories in which microprocessors and a host machine interact.

The specific topics addressed are (1) the advantages and disadvantages of distributing control and data storage between a central (host) machine and a set of satellite microcomputers, (2) some of the applications we have developed, (3) the importance of making full use of the host system in microsystem development,

(4) the value of a good cross-assembler and how we developed ours, (5) simple protocols for sending data between host and satellite computers, (6) software techniques for simulating the automatic priority interrupt capabilities found on larger machines, and (7) the sort of human resources needed to undertake such a venture.

ADVANTAGES AND DISADVANTAGES OF HOST-SATELLITE ARRANGEMENTS

We initially planned to use microprocessors as a less expensive alternative to discrete logic in interfacing experimental laboratories to our main minicomputer resource. Soon, however, the need to move precision timing off of our overworked central machine led us to plan a series of intelligent microcomputer systems that control all parts of experiments for which precise stimulus or response timing is required. This move toward more sophisticated microcomputer systems in turn redirected our attention back toward the main machine as a source of programming conveniences and as a more effective and sophisticated data-base manager than our microsystems are.

One alternative to using microprocessors in the periphery of a larger machine is to rely entirely on the host machine to support peripheral worksites. Unfortunately, our central machine cannot handle the full load we now have, and its usage continues to expand exponentially. A second alternative is to build separate independent microsystems for each experimental laboratory. For a research laboratory such as ours, in which there are quite a number of separate labs, remote tasks in schools, word processing facilities, and the like, to support, the cost of separate facilities is too high. This is because the cost of large-scale data storage, back-up of data, and the user-oriented conveniences

Portions of the research reported herein were supported by grants from the National Science Foundation and the National Institute of Education, United States Department of Health, Education, and Welfare. The opinions expressed here do not necessarily reflect the position or policy of the sponsoring agencies and no official endorsement should be inferred. For reprints or more detailed information, contact the authors at Learning Research and Development Center, 3939 O'Hara Street, Pittsburgh, Pennsylvania 15260.

that make computer usage attractive in the first place would be prohibitive if multiplied by the number of separate systems we would need. We have therefore chosen to develop distributed systems, and we are now quite aware of the advantages and disadvantages of this approach.

The advantages lie in the symbiotic relationship that a large machine and a microsystem can have. The large machine tends to have substantial storage capabilities: disk drives, magnetic tapes, and so on. Further, in any well run installation, there is a systematic back-up and check-pointing procedure for this memory, so that it can be thought of as a quickly accessible and very secure data storage system. In contrast, if experiments were run directly on stand-alone microsystems, data integrity would be in the hands of a wide range of research assistants, some of whom might lack both the expertise required to discern potential threats to the data and the necessary clerical skills required to keep track of a growing library of diskettes or cassettes. A similar argument can be made for concentrating program storage in a central system, especially if programming is done by a central staff and not by each individual researcher. If programs are centrally stored, updates and corrections are automatically distributed to all users, since the program is always downloaded from a central source. On the other hand, if each laboratory has an accumulation of semilabeled cassettes or diskettes, there is higher probability of the wrong edition being used.

Another advantage of having a central machine supporting satellite microsystems is that there will be a variety of powerful support capabilities in the central machine that cannot be included in each microfacility. For example, a powerful cross-assembler on the main machine can supply quick and simple programming facilities, including a wide range of metainstructions (pseudooperations), the ability to define new macroinstructions, and other options. Available assemblers for microsystems are usually slow and often lack many of the more useful features larger systems provide. In the future, as the new PASCAL machines are developed, this will be less of a concern. For most of us, though, the resources of a central machine will be useful for quite a while, even if nothing but faster assemblies, compilations, and speedy line-printer listings are all that we gain from the mini-micro alliance.

An additional advantage of distributed systems lies in the possibility of simple interfacing of a wide range of devices to the microcomputer. Response panels, voice keys, terminals, slide projectors, and graphic display systems can be attached to the microcomputer very easily. In most cases, the interface is managed by one or two integrated circuit chips. Later in this paper, we discuss the kinds of programming support the microsystem must contain for this to be most effective.

Interfacing directly to a larger machine, in contrast, is quite painful and expensive, and there is a high cost attached to the proliferation of a large and varied set of devices that must share a complex, high-speed data channel.

For similar reasons, it is advantageous to have all timing functions kept in the microsystem, since it is a single-user computer and hence more available for time-critical work. When timing must be done by a shared system, it is always a source of competition between jobs. Either the job that does not need timing is slowed down excessively, or the timing becomes less precise. In contrast, the luxury of even a small dedicated computer for an experiment, such as the downloaded microsystem, permits timing to be given guaranteed high priority.

The most obvious disadvantage of a distributed system is that failure of either the host system or the peripheral machine will prevent operations from continuing. Also, since all programs must be downloaded (this and many other terms are defined in Vesonder et al., 1978) from the central machine, a heavy burden is placed upon the host's terminal interface component, and downloading is often relatively slow in times of heavy usage. A similar problem can arise for data transfers. In principle, all complex arithmetic should be performed in the host if it has extended integer and/or floating point capabilities. In practice, however, the time needed to transmit raw data to the host and/or processed data to the microsystem can be excessive if the channel bandwidth is low or if the central machine is overloaded. Indeed, we have faced some problems because the central machine cannot handle input at high rates when it is heavily loaded. In spite of these problems, a distributed system is the only cost-effective way to preserve the highly user-oriented facility we have developed in the face of increased and more complex usage.

The extent to which each computer in a distributed (host-satellite) system participates in the execution of an experiment is dependent upon the complexity and speed requirements of the task. At one extreme, the program in the host machine maintains strict control of the status of the experiment. Commands are sent to the microcomputer to initiate a sequence of events, and data are received and analyzed at the end of each trial. The host is the system master performing all of the computation and decision-making functions. The primary task of the microcomputer is to interpret commands from the host and control the operation of external devices through its hardware interface.

At the other extreme, all of the major phases of a given experiment are under the control of the microcomputer. The timeshared system is used only for initial set-up and for receipt and storage of accumulated data. In this case, the microcomputer program must not

only control external devices, but also manage control states and at least partially analyze the data. Our current implementations fall in between these two extremes.

APPLICATIONS

There are currently three projects running in a distributed host-satellite arrangement at LRDC: a three-channel tachistoscope laboratory, a graphics display terminal, and a text-processing system. The equipment in the tachistoscope laboratory was originally operated by a special-purpose, discrete logic controller. The controller could execute one of four hardwired display sequences under the direction of a FORTRAN program running on the timeshared system. Display interval times and slide numbers were sent to the controller at the beginning of each trial, and the selected display program was executed. Subject response data and latency were then sent back to the host. This arrangement was more than satisfactory until it became necessary to alter one of the hardwired display sequences. Any changes or modifications to the hardwired programs required major rewiring of the controller. To expand the flexibility of the laboratory, the discrete logic was replaced with a microcomputer, and the controller functions were simulated in software.

Wiring the existing laboratory equipment to the microcomputer required very little in the way of special hardware. The subject response panel and three random-access slide projectors with solenoid-operated shutters were interfaced through two 16-bit, parallel, programmable I/O ports. The ports were part of a standard microcomputer interface module that also contained a three-channel programmable timer and two serial ports, one for a local terminal and the other for communicating with the timeshared system.

The task of writing the software for the microcomputer was divided into three parts. First, a series of I/O programs were written, each one tailored to the operation of a specific peripheral device. Routine calls to transfer data to peripherals were handled through a system of software interrupts. The actual operation of each device was managed by a hardware interrupt polling scheme. This established a standard communication procedure between main-line programs and external devices that has been used in all subsequent applications. The next step was to develop subroutines that would allow an interactive dialog between the timeshared system and the experimenter, whose terminal was connected to the microcomputer. These subroutines also include a means of downloading object programs and device handlers. Finally, a main-line program was written that could interpret commands from the host machine, pass parameters to the device handlers, and generate the desired experimental sequence.

To create a new sequence of slide selection or shutter operation requires changing only a relatively small

portion of the main-line program. This arrangement maintains all of the capabilities of the original hardwired system, while eliminating its restrictions. The microcomputer looks exactly like any other terminal to the timeshared system. Since its operation very closely approximates that of the old, discrete logic-based controller, very few changes were necessary to the host machine's programs that originally controlled the experiment.

The tachistoscope laboratory is an example of integrating a microcomputer into an already existing system to increase its capabilities. The graphics terminal, however, was designed in response to a need for a device that was not currently available. Several projects within the center expressed the need for a computer-controlled graphics system that could be used both as a research tool and a teaching aid. Applications ranged from static displays of simple geometric figures for an analogies study to a dynamic simulation of a common physics classroom demonstration.

The terminal was built around a Tektronix Model 611 X-Y display scope, which we have had for almost 10 years. This high-resolution unit is capable of both static displays and elementary animation. The digital-to-analog converters required to drive the X and Y axes of the 611 were constructed as part of a programmable vector generator. With this device, only three parameters are needed to generate straight lines of variable length and display density. Once these data are transferred to the vector generator, the microcomputer is free for other operations until a hardware interrupt signals that the line drawing is complete. This helps to simplify programming and to reduce the amount of time the microcomputer must spend generating and/or maintaining a display.

In all other respects, the microcomputer is an exact duplicate of the one used in the tachistoscope laboratory, and most of the support software developed for that project was used here. Device handlers for the vector generator were written to conform to the standard I/O procedures. These routines were given the highest priority in the hardware interrupt service schedule to accommodate the high-speed requirements of continually refreshed displays.

In the current applications, the programs on the host machine contain all of the software needed to compute the X-Y coordinates of the line endpoints for a given display. A table of point coordinates is transmitted to the microcomputer, which stores it until a display command is received. Depending on the application, this command may originate either from the host program or from a student at the keyboard of the local terminal. The heavy dependence on the host machine was the result of a conservative estimate of the speed with which the microcomputer could handle display operations. In future applications, more of the coordinate computation will be performed in the micro-

computer, especially when higher level languages become available on the smaller machine.

In the previous two examples, the microcomputer served primarily as a peripheral device controller managing events in real-time that were beyond the capabilities of a timeshared machine. In a third application, text processing, the roles are reversed, and the major computing task is shifted from the host to the satellite. The text-processing software is a modified version of a readily available, inexpensive, commercial program. Its task is to format and print documents created on the host system with the aid of the system editor. Beyond the initial dialog, the microcomputer program is in control of the host machine, issuing commands to open and close files, requesting unprocessed text, and transmitting text back after processing. Pacing of data transfers is controlled by the microcomputer to avoid overloading to the host machine.

The only hardware modification to our standard microcomputer is the addition of a third serial I/O port for a typewriter-quality printer. Software necessary to support this printer is a duplication of the other serial line handlers with only status codes and addresses changed. Any of our standard microcomputer systems can function as a text-processing system if it has a terminal with adequate print quality.

MAKING USE OF THE HOST COMPUTER

One lesson learned very early is that the large minicomputer is a much more hospitable environment for program development than is the stripped-down microcomputer. This is especially the case for us because we have substantial and quickly accessed on-line storage facilities. Thus, we can edit, assemble, and store all of our microcomputer programs within the luxury of a decent timesharing system. One immediate advantage of this approach is that there is no time lost learning to use a new editor or a new assembler. All that a programmer who has used our central system needs to learn in order to write software for our microcomputers is the microprocessor instruction set. All assembler pseudooperations are the same as for the main machine. The programs are written on the larger machine, using the usual editor. The cross-assembler is used to generate object code for these programs in downloader format, store them in disk files, and print program listing on a high-speed printer. All of these operations are managed quickly and efficiently by the operating system on the host machine. To load and run a microcomputer program, the programmer does what appears to be an interaction with the larger machine. The downloading programs of the microcomputer are part of its permanent read-only memory, and the host's downloading routine is very simple.

There is a small debugging system resident in each of our microsystems that facilitates program testing.

Its format is close to that of the debugging facility that we have on the larger system, in keeping with our policy of coordinating all programming within our laboratory around a single set of conventions. In general, debugging is made much easier by the heavy commenting of assembly code that we insist upon. Because this commenting is processed only by the host machine, it costs little in extra assembly or printout listing times. Another contribution of the cross-assembler to debugging is its capability of producing, very quickly, cross-referenced symbol tables that show each location at which a particular symbol is used.

Much of the above discussion suggests that we are organized around the use of assembly language rather than higher level languages. This is currently true. However, we fully intend to move to higher level languages for the micro as soon as reasonable systems for the processors become available. We expect to continue the one-system philosophy, even if we change to a different central machine, probably by adopting PASCAL as the source language of both the microsystem and the host system. Further, as memory gets cheaper and address spaces of microsystems get larger, we are likely to shift the balance of microsystem-resident and main-system-resident software tools, although we will continue to make the location of these tools a matter with which the user need not be concerned.

CROSS-COMPILERS AND CROSS-ASSEMBLERS

Cross-compiling and cross-assembling are techniques of using one computer system (host) to produce machine code for another (usually smaller) system. The major goal at LRDC was to produce a cross-assembler that could be easily upgraded over time to reduce the number of program statements necessary to perform a particular function without requiring an overhaul of the assembler itself. Ultimately, less sophisticated programmers were to be provided with a high-level macro language and, concurrently, software engineers were to retain the bit manipulation capabilities they required.

To accomplish this goal, the basic elements of a macroassembler must be written. The design must include conditional assembly pseudooperations (assembler directives), as well as assembly-time macro-definition capability. Most macroassemblers available on larger systems meet these two requirements. This is an important point, since it can significantly reduce the development time required to produce a cross-assembler. Our cross-assembler was produced by strategically modifying the host language assembler to understand the microcomputer assembly language mnemonics. The specific conversion was from a PDP-15 MACRO assembler to a 6800 macro cross-assembler. This conversion took 3 person-weeks. A 4th week was required to add the features to be discussed below. (A disadvantage

to this approach is the lack of transportability. However, the time required for the conversion is minimal if the programmer is an accomplished tinker.)

The most significant deviation from the standard assembly procedure was the total elimination of "relative" code assembly. Traditionally, separate modules or subroutines are assembled, resulting in relocatable machine code that is later linked to the machine code of other separately assembled modules through the use of a linking loader. The LRDC assembler could be called a "linking assembler," since it accepts a sequence of assembly language data sets and assembles them as one program.

The obvious disadvantages to this procedure are that all source code must be read and assembled to produce each version of a program even though the intervening changes may have involved only one module, the assembler symbol table requirements are increased, and symbol definition conflicts may arise. The advantages are that the complexity of the assembler is significantly reduced, and no linking loader need be written.

The first two disadvantages are proportional to the throughput and memory limitations of the host system. Our host system's memory limitation of 16K 18-bit words for any user task (during normal hours) allows assembly of microcomputer programs of up to 2,000 six-character symbols. Since individual modules may be assembled separately for development purposes, large assemblies need only be made for final version production. Symbol conflicts between modules can be handled in several ways. A simple solution is to restrict symbol size (number of characters) to one less than the symbol table is designed to handle and then internally prefix each symbol with a character unique to each module before making it an entry. If global references in the source code are prefixed with a special character and the assembler is written to suppress the internal prefixing described above when these symbols are encountered, global references will then be entered in the table intact, and, consequently, will be addressable from all modules. Our assembler does this automatically when the appropriate pseudooperation is specified.

Since the assembler has been designed to accept multiple-file input, it was trivial to provide a library of subroutines and macros that perform commonly needed functions. The cross-assembler adds two names to the list of modules that the user requests to be linked and assembled. The first is a file containing definitions of all "system" macros. The second is a subroutine library containing globally named subroutines that are referenced by the macros in the first file. The macro file is the first source file the assembler processes, making all system macros available for use in the user's program modules. The subroutine library is introduced immediately following the last user-specified module. All subroutines in the library are delimited by conditional assembly directives. They are included in the

assembly only if reference is made to them. Consequently, the system may be upgraded by merely adding new MACRO definitions and/or subroutines to the two system files.

This has proven to be a flexible and powerful system for the microcomputer applications we have developed. The time expended in development was minimized by using the conversion approach described above, and the elimination of the concept of relocatable code significantly reduced the logical complexity of the conversion.

A further upgrading of the system might be the implementation of a precompiler for a higher level language that would output a series of macro calls defined in the cross-assembler's macro file. This approach would allow the designer of the precompiler to concentrate on the higher level language syntax and grammar, relegating the details of symbol manipulation and code production to the assembler.

SIMPLE INTERSYSTEM COMMUNICATIONS PROTOCOLS

The ETSS/6800 downloader is a short program, stored in the microsystem's read-only memory, that establishes the initial communication link between the host and satellite systems. It is designed to pass ASCII characters between the microsystem's terminal and the timeshared system and to provide an elementary means of data verification while loading programs from the host into the microcomputer memory.

Several months were spent studying the relative merits of various software procedures and line protocols that would increase downloading integrity. We considered buffered and unbuffered character handling, with or without interrupts, and several elaborate schemes of synchronizing and acknowledging data transfers. These discussions led to the development of a relatively simple program that could be used to load a more sophisticated procedure if it was found necessary. A read-only memory chip containing this simple downloader was programmed and installed in the first development station; it has been used exclusively ever since without difficulty.

When the downloader is started, it sets up the microcomputer stack pointer, initializes the serial I/O ports to both the terminal and the host system, and prompts the user by printing an asterisk on the local terminal. The basic function of this initial program segment is to simulate the operation of an ordinary user terminal. With it, the user can log onto the host system and have access to all of its resources.

Characters typed on the local terminal are passed through to the timeshared system without character checking or special structuring. The full duplex line mode established by the host system is maintained as the only means of verifying transmission integrity.

Characters received from the timeshared system, however, are examined for a special switch character, control-D, before they are printed on the local terminal. When this special switch character is detected, program control is diverted from this pass-through routine to the loading segment. This segment expects that all subsequent characters from the host will be organized in a standard record format that can be examined, decoded, and stored in memory as a part of a user program.

The record format, an adaptation of the standard format used in commercial 6800 operating systems, represents the contents of each memory byte as a pair of hexadecimal digits (in ASCII). Each record consists of a type code, record length, destination address, data, and checksum. All record types contain information in the same format. Each record begins with the letter S to identify the start of a record, followed by a one-digit number specifying the type of record being sent. This is immediately followed by two hexadecimal digits specifying the length (in bytes) of the record. Since each data byte is represented as two hexadecimal characters, the byte count must be multiplied by two to compute the number of characters to the end of the record. (Checksum and address data are included in the byte count.) The next four digits are the memory address at which the record is to be stored. The address is followed by pairs of characters, each pair representing 1 memory byte. The last record element is a two-character checksum. This is the one's complement of the modulo-2 addition of the data bytes, the address value, and the byte count. Examples of this record format are shown in Figure 1.

The downloader recognizes two types of records.

S1 RECORD		S9 RECORD	
FUNCTION	CHARACTER	FUNCTION	CHARACTER
START OF RECORD	S	START OF RECORD	S
RECORD TYPE	1	RECORD TYPE	9
BYTE COUNT	0	BYTE COUNT	0
	7		3
	A		F
	0		F
	0		F
ADDRESS	0	ADDRESS	F
	1		F
DATA BYTE 1	0	CHECKSUM	E
	1		
DATA BYTE 2	A		
	2		
DATA BYTE 3	0		
	2		
DATA BYTE 4	A		
	E		
CHECKSUM	4		

MEMORY CONTENTS AFTER LOADING S1 RECORD	
ADDRESS	DATA
A000	10
A001	1A
A002	20
A003	2A

Figure 1. Examples of S1 and S9 records that are used to download user-program object code from the host machine to the microcomputer.

Data to be loaded into memory are handled by S1 records. After each S1 record is processed, the downloader remains in the loading segment looking for additional records. An S9 record signifies the end of the data file. This record type does not contain data to be loaded into memory. Its primary functions are to mark the end of a file and to provide a starting address to which the downloading routine should transfer control. If the address portion of an S9 record contains the value FFFF, the program will return to its initial dialog procedure and await further instructions. If any other address value is received, the downloader program will set the stack pointer to the top of the reserved stack area and jump to the user program at the specified address. Several program segments can be loaded into memory by sending end-of-file records containing an address of FFFF after each segment. The complete program can be started immediately after downloading the last segment, by transmitting an S9 record containing the starting address.

Four forms of transmission error detection are built into this format, all of which are terminal. First, if an S is received followed by any character other than 1 or 9, the downloader sends an abort character, control-C, to the timeshared system and prints a "type error" message on the local terminal. If during the course of processing an S1 or S9 record a nonhexadecimal character is received, loading is aborted and an illegal character message is printed. Each time a data byte is loaded into memory, it is immediately checked for an illegal address or storage location errors. After each byte is stored in memory, it is read back and compared. If the value read from memory is not equal to the data that were stored, an illegal address error message is printed (either there was no memory corresponding to the specified address or there was a memory failure) and loading is aborted. Finally, if a checksum error is detected, it is reported to the local terminal and the loading process is ended.

Asynchronous data transfers from the host to the microcomputer are relatively unrestricted in that characters are transmitted as they become available to the timeshared system. The microcomputer can be programmed to respond as quickly and as often as is necessary to handle them. Transmission from satellite to host must be paced to the rate at which the timeshared system can accept and accumulate characters. The exact procedures used depend heavily on the idiosyncracies of the particular host system. A detailed description of the methods used with our computer will, no doubt, be of little value on other systems. A third record type, S2, has been defined that contains information in essentially the same format as the S1 and S9 types, but requires a more sophisticated program on both the host and satellite machines to effect acknowledgement and retransmission of inaccurate data transfers. This record type is used, for example, by our word processing system, but it is not required by the downloader.

SOFTWARE PRIORITY INTERRUPT SERVICE SCHEDULER

A modern minicomputer has a number of priority levels at which hardware devices and system software can request interrupt service. This allows devices needing quick service to preempt less critical background tasks. A microprocessor characteristically lacks more than one or two priority levels. In particular, the 6800 has only one level suitable for use by I/O devices. To overcome this problem, we have created a software solution that mimics multilevel interrupt hardware.

The design is straightforward and consists basically of two levels of service. The first level is initiated immediately following the hardware interrupt. Priority is established among devices at this level by ordering the testing of device flags (i.e., the highest priority device is tested first). For each device flag set, the appropriate Level 1 service routine is called as a subroutine. All first-level routines are required to at least clear the device interrupt flag. All time-critical functions are performed by Level 1 routines with hardware interrupts disabled. If any noncritical functions remain to be performed, a flag is set indicating that Level 2 service is required. When all devices have been serviced

at Level 1, the Level 2 request flags are checked. Each is tested in turn (highest priority first). If the flag is set, the active flag for that priority is set, program interrupt is enabled, and the appropriate Level 2 routine is called. If a Level 2 request flag is found not set and the active flag is set, the scheduler is exited through execution of a return from interrupt (RTI) command. If the Level 2 request flag is not set and the active flag is not set, the next device is checked. Note that Level 2 routines can be reentered and should have a check for this condition built in.

The key to the logic of the scheme is this reentrance (from the top) of the scheduler during Level 2 execution; Level 1 critical service always preempts Level 2 noncritical service and higher priority Level 2 service preempts lower priority Level 2 service. The only requirement of Level 2 service is that the program interrupt be enabled as much as possible.

The scheduling algorithm does not guarantee against timing problems, but it does provide an easy method of pinpointing such problems: When a Level 2 service routine is reentered, it is clear that its respective device is generating interrupts faster than they can be serviced. A listing of the interrupt service scheduler currently employed in our 6800 programs is given in Table 1.

Table 1
Listing of Interrupt Handler

.TITLE		INTERRUPT SERVICE SCHEDULER	
/			
/ SYMBOLS IN THE INSTRUCTION FIELD PRECEDED BY PERIODS			
/ ARE ASSEMBLER DIRECTIVES OR MACRO CALLS.			
/			
/ “*G” INDICATES ASSEMBLER GENERATED CODE.			
/			
0100	.LOC	\$100	/ ESTABLISH ORIGIN
/			
	.OCT		/ INTERPRET NUMBERS OCTALLY
/			
/ DEVICE ASSIGNMENTS			
/			
/ SET DEVIC[N] = N TO DEFINE A DEVICE SLOT			
/ EACH DEVICE IS ASSIGNED A DIFFERENT LEVEL.			
/			
/ SET DV[N] FLG = #ADDR OF INTERRUPT SVC FLAG			
/			
/ SET DV[N] SVC = #ADDR OF LEVEL-1 SVC ROUTINE.			
/			
/ THE FOLLOWING ILLUSTRATES A SCHEDULER OF 3 DEVICES.			
/			
0003	TOTLD=3		/LOWEST PRIORITY USED
0000	DEVIC0=0		/DEVICE 0 – PRIORITY 0
0001	DEVIC1=1		/DEVICE 1 – PRIORITY 1
0002	DEVIC2=2		/DEVICE 2 – PRIORITY 2
1000	DV0SVC=GRAPH1		/GRAPHICS SCOPE–PRIORITY 0
1545	DV1SVC=LCRTCK		/LOCALTERMINAL–PRIORITY 1
1681	DV2SVC=LNRTCK		/HOST LINE–PRIORITY 2
0FFE	DV0FLG=SCOPF		/GRAPHICS VECTOR INTERRUPT FLAG
0FFD	DV1FLG=LOCAL		/LOCAL TERMINAL FLAG
0FFC	DV2FLG=LINE		/HOST LINE FLAG

Table 1 Continued

```

/ LOCAL STORAGE
/ LEVEL 2 SUBROUTINE ADDRESS TABLE.
/MUST BE LOADED DURING INITIALIZATION OF PROGRAM.
/MAY BE MODIFIED BY MAIN LINE CODE.
/A ZERO ENTRY INDICATES NO SERVICE REQUIRED.
/
0100          LV2ADR  .RMB      2*TOTLD  /2 BYTES/DEVICE
0106          IORB   .RMB      TOTLD    /LEVEL 2 REQUESTS
0109          L2ACT   .RMB      TOTLD    /LEVEL 2 ACTIVES
                  .EJECT
/
/***** PROCEDURE IRQSCN *****/
/
/ IRQSCN DISPATCHES ALL SERVICE TO PROGRAM INTERRUPTS.
/ IT IS RE-ENTRANT AND ASSUMES EACH LEVEL 1 SERVICE
/ ROUTINE SETS THE APPROP. BIT IN LOCATION IORB+SLOT#
/ WHEN LEVEL 2 SERVICE IS REQUIRED.
/ R0,R1,R2,R3 ARE SAVED AND RESTORED BEFORE RTI.
/
/ THE FOLLOWING TWO MACRO DEFINITIONS DESCRIBE CODE
/ GENERATED FOR EACH DEVICE DEFINED ABOVE.
/
/      LEVEL 1 DEVICE FLAG TEST MACRO
/      .DEFIN      TSTFL      S
/      .IFDEF      DEVIC@S@    /ASSEMBLE ONLY IF DEFINED
/      TST         DV@S@FLG    /TEST THE DEVICE FLAG
/      BPL         .+5         /IF NOT SET, GOTO NEXT TEST
/      JSR         DV@S@SVC    /OTHS, CALL LEVEL 1 SVC
/      .ENDC       /END CONDITIONAL ASSEMBLY
/      .ENDM       /END OF MACRO DEFINITION
/
/      LEVEL 2 SERVICE TEST MACRO
/      .DEFIN      TSTL2      S
/      .IFDEF      DEVIC@S@    /ASSEMBLE ONLY IF DEFINED
/      TST         IORB+S      /TEST FOR L2 REQUEST
/      BNE         TX1@S@      /IF NON-ZERO, CALL L2 ROUT.
/      TST         L2ACT+S     /OTHS, TEST FOR ACTIVITY
/      BEQ         TXX@S@+3    /IF NONE, CHECK NEXT DEVICE
/      JMP         IRQEXT      /IF THERE IS, EXIT
/      TX1@S@      CLR         IORB+S    /CLEAR THE REQUEST
/      LDX         #S*2+LV2ADR  /LOAD L2 TABLE ADDR.
/      LDX         0, X        /LOAD ADDR. OF L2 ROUTINE
/      BEQ         TXX@S@+3    /IF NONE REQD, CHECK NEXT
/      LDA A       #1         /OTHS,
/      STA A       L2ACT+S     /SET THIS PRIORITY ACTIVE
/      CLI         /ENABLE PROGRAM INTERRUPT
/      JSR         0, X        /CALL LEVEL 2 SERVICE
/      SEI         /DISABLE PROGRAM INTERRUPT
/      TXX@S@      CLR         L2ACT+S   /CLEAR L2 ACTIVE FLAG
/      .ENDC       /END CONDITIONAL ASSEMBLY
/      .ENDM       /END MACRO DEFINITION
/      .EJECT
/
/      ENTRY FOLLOWING PROGRAM INTERRUPT
/
010C          IRQSCN .SAVER                      /SAVE EXTENDED RG.
010C 96        LDA A  *0
010E 36        PSH A
010F 96        LDA A  *1
0111 36        PSH A
0112 96        LDA A  *2
0114 36        PSH A
0115 96        LDA A  *3
0117 36        PSH A
/      INITIATE LEVEL 1 SERVICE
/      TSTFL      0          /DEVICE 0 SERVICE
0118 7D 0FFE *G  TST      DV0FLG
011B 2A 03  *G  BPL      .+5
011D BD 1000 *G  JSR      DV0SVC

```


Table 1 Continued

0120	7D	0FFD	*G		TSTFL	1	/DEVICE 1 SERVICE
0123	2A	03	*G		TST	DV1FLG	
0125	BD	1545	*G		BPL	.+5	
					JSR	DV1SVC	
					TSTFL	2	/DEVICE 2 SERVICE
0128	7D	0FFC	*G		TST	DV2FLG	
012B	2A	03	*G		BPL	.+5	
012D	BD	1681	*G		JSR	DV2SVC	
					TSTFL	3	/DEVICE 3 SERVICE
				/			
				/	INITIATE LEVEL 2 SERVICE		
					TSTL2	0	/DEVICE 0 SERVICE
0130	7D	0106	*G		TST	IORB+0	
0133	26	08	*G		BNE	TX10	
0135	7D	0109	*G		TST	L2ACT+0	
0138	27	19	*G		BEQ	TXX0+3	
013A	7E	0199	*G		JMP	IRQEXT	
013D	7F	0106	*G	TX10	CLR	IORB+0	
0140	CE	0100	*G		LDX	#0*2+LV2ADR	
0143	EE	00	*G		LDX	0, X	
0145	27	0C	*G		BEQ	TXX0+3	
0147	86	01	*G		LDA A	#1	
0149	B7	0109	*G		STA A	L2ACT+0	
014C	0E		*G		CLI		
014D	AD	00	*G		JSR	0, X	
014F	0F		*G		SEI		
0150	7F	0109	*G	TXX0	CLR	L2ACT+0	
					.EJECT		
					TSTL2	1	/DEVICE 1 SERVICE
0153	7D	0107	*G		TST	IORB+1	
0156	26	08	*G		BNE	TX11	
0158	7D	010A	*G		TST	L2ACT+1	
015B	27	19	*G		BEQ	TXX1+3	
015D	7E	0199	*G		JMP	IRQEXT	
0160	7F	0107	*G	TX11	CLR	IORB+1	
0163	CE	0102	*G		LDX	#1*2+LV2ADR	
0166	EE	00	*G		LDX	0, X	
0168	27	0C	*G		BEQ	TXX1+3	
016A	86	01	*G		LDA A	#1	
016C	B7	010A	*G		STA A	L2ACT+1	
016F	0E		*G		CLI		
0170	AD	00	*G		JSR	0, X	
0172	0F		*G		SEI		
0173	7F	010A	*G	TXX1	CLR	L2ACT+1	
					TSTL2	2	/DEVICE 2 SERVICE
0176	7D	0108	*G		TST	IORB+2	
0179	26	08	*G		BNE	TX12	
017B	7D	010B	*G		TST	L2ACT+2	
017E	27	19	*G		BEQ	TXX2+3	
0180	7E	0199	*G		JMP	IRQEXT	
0183	7F	0108	*G	TX12	CLR	IORB+2	
0186	CE	0104	*G		LDX	#2*2+LV2ADR	
0189	EE	00	*G		LDX	0, X	
018B	27	0C	*G		BEQ	TXX2+3	
018D	86	01	*G		LDA A	#1	
018F	B7	010B	*G		STA A	L2ACT+2	
0192	0E		*G		CLI		
0193	AD	00	*G		JSR	0, X	
0195	0F		*G		SEI		
0196	7F	010B	*G	TXX2	CLR	L2ACT+2	
					TSTL2	3	/DEVICE 3 SERVICE
				/			
				/	EXIT FROM IRQSCN		
0199				IRQEXT	.RESTR		/RESTORE R0-R3
0199	32		*G		PUL A		
019A	97	03	*G		STA A	*3	
019C	32		*G		PUL A		
019D	97	02	*G		STA A	*2	

Table 1 Continued

019F	32		*G	PUL A		
01A0	97	01	*G	STA A	#1	
01A2	32		*G	PUL A		
01A3	97	00	*G	STA A	*0	
01A5	3B			RTI		/RET FROM INTERRUPT
/						
/***** END OF PROCEDURE IRQSCN *****/						
.EJECT						

NEEDED HUMAN RESOURCES

It seems important at this point to provide some sense of the level of expertise of the people that have worked on development of the LRDC system. They include a software engineer and a computer hardware engineer with substantial systems development experience, probably more than will likely be available to the average psychological laboratory. However, the techniques we have developed are not that difficult to apply and should simplify the lives of any researchers trying to develop their own general computer-based laboratories. This is especially the case with the priority interrupt handler and the downloading and general intercomputer protocols, which are extremely straightforward. In the case of the cross-assembler, it may be more appropriate for many researchers to try to purchase such software rather than build their own—our only claim is that it is valuable. Finally, there is the

issue of timing for major laboratory overhauls. Quite probably, within the next year or so, such microcomputer systems as San Diego PASCAL will be more feasible, especially for stand-alone systems. Even then, going into microcomputers need not be a step backward in computing sophistication if one makes maximal use of available higher level systems for support.

REFERENCES

- FITZHUGH, R. J., & GLASER, R. A general-purpose computer system for a research and development center. *American Psychologist*, 1975, **30**, 219-225.
- LES GOLD, A. M., & FITZHUGH, R. J. An on-line psychological laboratory for teaching and research. *Behavior Research Methods & Instrumentation*, 1977, **9**, 184-188.
- VESONDER, G. T., LESGOLD, A. M., & WOLF, R. V. The use of microprocessors in a timeshared computer facility for research, development, and teaching. *Behavior Research Methods & Instrumentation*, 1978, **10**, 510-515.