CONF-830576--2

A MICROPROCESSOR MULTI-TASK MONITOR

C. A. Ludemann
Oak Ridge National Laboratory*
Oak Ridge, Tennessee 37830

DE83 012590

Summary

This paper describes a multi-task monitor program for microprocessors. Although written for the Intel 8085, it incorporates features that would be beneficial for implementation in other microprocessors used in controlling and monitoring experiments and accelerators. The monitor places permanent programs (tasks) arbitrarily located throughout ROM in a priority ordered queue. The programmer is provided with the flexibility to add new tasks or modified versions of existing tasks, without having to comply with previously defined task boundaries or having to reprogram all of ROM. Scheduling of tasks is triggered by timers, outside stimuli (interrupts), or inter-task communications. Context switching time is of the order of tenths of a millisecond.

Introduction

The speed and architecture of 8-bit microprocessors are adequate for most control and monitoring applications in the laboratory. These devices can be found incorporated in hardware systems of varying complexity. At the one extreme, they are integrated into a single circuit board with limited input and output capability. Programs are in ROM. At the other extreme, they support, or are supported by, a broad range of peripherals such as disks, printers, and graphics generating CRTs. The latter implementation typically provides the capability to "download" individual applications programs into RAM for execution. This flexibility in changing software makes these systems ideal for acquiring and processing data at moderate rates. The volatile nature of the software, however, can make such systems unsuitable for predictable and safe operation of equipment.

Our use of microprocessors in accelerator control applications involves a "black box" approach. This means that programs reside in ROM only and the full capability of a unit is initialized upon application of power or by use of a RESET switch. All perform well-defined functions with no operator involvement in software decisions. They are used to emulate hard-wired controllers, or monitor and report the status and safety of equipment. They are essentially of the single circuit board configuration. Use of microprocessors in the control and monitoring of experiments is usually relegated to the operation of a major piece of equipment (target chamber, spectrometer, etc.). Here, again, a "black box" approach is desirable. This is particularly true at a multi-user laboratory. Users should be provided with well defined operational characteristics (and well-tested firmware!) and perhaps a few optional features preprogrammed in ROM.

Raison d'Être

The usual method for programming a control and monitoring application, without a multi-task monitor, is to embody all of the software within the bounds of a single program. Typically, this program is a repetitive sequence of operations, e.g.: read data, display data, check to see if the operator has entered a command (via key-board, switch panel, etc.). If a command is present, the program branches back to the beginning to read data again. If there is a command present, the program branches to the appropriate subroutine to execute it, and then back to the beginning.

Implicit in the use of a sequence is the element of time. Consideration must be given in the program to such factors as the refresh rates of the data presented on displays, delays between the detection of an alarm situation and the beginning of the corrective action, and response times of the hardware. In order to avoid timing problems, it is imperative that the complete control and monitoring application be defined at the outset of programming the sequence. In this way, an evaluation can be made of the time dependence of each path in the program and its effect on the application as a whole. If this is ignored, a great deal of programming may have to be redone in order to introduce additions.

An example of how quickly difficulties arise with the sequential approach might be to examine the programming of the control software for a magnetic spectrograph and its ancillary equipment. To start out, one might just wish to read the status of valves; the pressures in the beam line and target chamber; and the current in the magnet. This information would then be displayed on a CRT terminal. It would be hard to see why the data could not be refreshed several times per second. Realizing that the resources of the microprocessor are being underutilized, you might now like to introduce a command from the terminal to set the current in the magnet. Because of the various time constants involved, this operation might take on the order of minutes, especially if one wished to cycle the magnet to minimize the effects of hysteresis. In the sequence outlined above, however, data would no longer be refreshed on the CRT until the operation has been completed.

To remedy this unacceptable situation one would most likely implement a hardware clock that could generate interrupts. The initial steps of the subroutine that executes the operation would set up the clock, enable interrupts, and then return to the beginning of the program to read data. The current in the magnet would be altered as each interrupt is recognized. Encouraged by the increased capability of the system, one now decides to add a command to rotate the spectrograph to a specific angle. This procedure, again, involves times on the order of many seconds to complete. Depending upon the microprocessor it might be possible to add another hardware clock. However, attention would have to be given to the enabling and disabling of interrupts in order to permit angle and magnet current setting to occur "simultaneously". If a single clock and interrupt is used, the capabilty of "simultaneity" would be handled in a somewhat different manner.

It is prudent to step back and observe what has been programmed in the example above. Essentially code has been generated to permit four separate programs to run "simultaneously" (read and write data, interpret commands, set the magnet current, and set the angle). In addition, the details of the clock interrupt code include many of the considerations that are found in a multi-task monitor. Unfortunately, this code is buried in the text of a specific

*Research sponsored by the Division of Basic Energy Sciences, U.S. Department of Energy, under contract W-7405-eng-26 with the Union Carbide Corporation.

application. It would be very difficult to sort out if someone wished to use it for some other purpose. Furthermore, any changes or additions that one would like to make require a detailed consideration of the entire code. It would be possible to add correct software for a new command that could introduce errors in previously operational routines. Finally, it is likely that reprogramming of all of ROM would have to be performed for every change.

The usefulness of the multi-task monitor to be discussed in the following sections depends upon the compexity of the application. However, such a monitor permits the generation of control and monitoring software far more complex than the spectrograph example discussed above. Such application can be accomplished in a more transparent and modular manner than by using the standard sequential approach.

Monitor Basics

The monitor was designed for the "black box" mode of operation discussed earlier. It permits programs to run in RAM, but this mode of operation is generally used for software check-out prior to placing the code in ROM. The only hardware requirement for the monitor to operate is that the microprocessor have one continuously running hardware clock which generates an interrupt.

The following definitions should make the discussion of the monitor more readily understandable.

Task A task is a complete program. It can be a repetitive loop such as the "read and display data" program in the spectrograph example. It also can have a "single-shot" nature as the program used to set the magnet current.

Task Control Block The monitor builds a Task Control Block (TCB) in RAM for each task in ROM. This block of data includes information such as the starting address of the task and its status (e.g., whether it is capable of running or is dormant).

Queue The monitor links the TCBs in a continuous chain. The first link in the chain is labeled TCB0 and is the "top of the queue". Within TCB0 is the address of TCB1. TCB1 has the address of TCB2, TCB2 points to TCB3, and so forth to the last link in the chain. The last TCB points to TCB0.

Taskmaster Within the monitor is a section of code that examines the TCB chain and determines how the microprocessor should be spending its time. This code is the Taskmaster. It looks at the status word in a task's TCB. If it finds the task is supposed to be dormant it locates the address of the next TCB in the chain. If this task is capable of running, the Taskmaster restores the contents of all registers to their values when the task was last operating. It then turns control of the CPU over to the task at the point where it last left off.

Common Services Within the monitor is a set of subroutines that all tasks can use. These are called Common Services. They permit a task to interact with the Taskmaster without having to know any details of the monitor. Almost all Services cause "context switching". The Taskmaster records all pertinent information about the calling task in its TCB (status, address where the contents of all operating registers and point of return are stored). Then it will turn over the CPU to some other task depending on the Service that has been called. Two of the Services are the activation of another task (the command interpreter starting the magnet current-setting program in our example), and the setting of a timer (the time interval the current setting program would like to have between current increments). If none of the tasks in the Queue uses the Services, the Taskmaster gives each task an equal time slice within which to operate. One has, in effect, a round-robin or circular queue where there is no significance attached to being at the "top of the queue".

Monitor Initialization

The monitor is activated immediately upon the application of power to the microprocessor or after a *RESTART* occurs. It clears all RAM, sets up the TCB Queue and register save areas for the tasks (push stacks), starts the clock, and enters the Taskmaster. The Taskmaster begins its operation by examining the status of the task associated with TCB0. The system is running!

TCB Queue Set Up In order to establish the Queue, the monitor must know how many TCB's are required. It determines this by searching all of ROM for tasks. A task is identified by five bytes of information coded immediately before the program's starting location. There are two possible forms:

```
DB      'TSS'       I AM A TASK
DB      N           MY PRIORITY IS 'N'
DB      00H         START ME WHEN MY TIME COMES
```

or

```
DB      'TSS'       I AM A TASK
DB      N           MY PRIORITY IS 'N'
DB      80H         I'M INITIALLY DORMANT
```

The uniqueness of such byte patterns is perhaps questionable. They were chosen with two coding rules in mind: a) do not use message labels that contain the consecutive characters "TSS", and b) do not generate bizarre software ("TSS" = MOV D,H; MOV D,E; MOV D,E). Correspondence of these bytes with a table of addresses is not possible since the systems we have been using do not have valid memory addresses in the range 5000H to 9000H. This leaves the possibility of confusion with data tables in ROM, a one in $\sim 2 \times 10^9$ chance.

The value of "N" in the fourth byte determines the position in the Queue to which the task is assigned. If N = 0, the task is at the "top of the queue" and as one might expect has the highest priority. The monitor searches for the largest value of "N" and creates a TCB Queue with N(max) + 1 nodes.

It is not necessary for the tasks to be programmed in ROM in their priority order or at any specific addresses. Furthermore, it is permitted to have two tasks in ROM with the same value of "N". However, only one will be linked in the Queue as the monitor assumes the task located at the highest address is the desired version. This permits modified versions of "old" tasks to be placed in an empty ROM socket (usually at a higher address) without having to reprogram the chip that contains the original. It is also possible to create more TCB's than there are programs in ROM by having fewer tasks than N(max) + 1. These vacant TCB's are useful for linking tasks which need to be checked out in RAM.

The information contained in the fifth byte is used by the monitor to set the initial status in the task's TCB. The Taskmaster will start a task on its first pass if it was coded with the first form of

header. Use of the second form will cause the task to be bypassed until such time as another task requests it.

Register Save Area In the course of the TCB Queue search procedure, the monitor determines the storage area in RAM required for saving each task's "environment" (register contents and return address) during context switching by the Taskmaster. A "push-stack" of 64 bytes is reserved for each task. While a typical task usually uses ~20 stack locations, special circumstances required a larger allotment. An excellent ~2500 byte "debug" program with CAMAC test routines was obtained for checking software and hardware.[1] The original program had paths that permitted the stack to grow. Rather than reprogram the code, a patch was inserted in a common path that resets the stack pointer to the value established by the monitor upon initial operation. This experience is mentioned to make two points. First, programs written for non-multi-task use can be run with this monitor. Second, in using a multi-task monitor such as this, one must be exceedingly careful in stack maintenance. If you PUSH, you had better POP. It does not take very long to appreciate this fact.

Specific Common Services

Access to a specific Common Service provided by the monitor is gained by a subroutine CALL statement. While direct coding of the appropriate instructions and address is possible, what is requested becomes more transparent if the assembler being used supports Macro statements. The descriptions to follow use Macro names.

RELINQ This statement is used when the task has nothing useful to do for a while - it is relinquishing its CPU time. It instructs the Taskmaster to look down the queue for the next task that is capable of running. An example of its use is a task that is writing to a terminal at 9600 baud. A character has been sent to the UART and approximately one millisecond will pass before another can be sent.

```
OUT    UARTWR      SEND CHARACTER
RELINQ             GO AWAY FOR A WHILE
IN     USTAT       CAN I SEND ANOTHER?
```

When the task again has CPU time it will test to see if the next character can be sent. If the answer is negative, it would RELINQ again.

RELTOP This Service is used when a task has discovered or created an "event" that some higher priority task should know about immediately. The Taskmaster goes to the "top of the queue."

RELTIL The task relinquishes its CPU time until the byte in the RAM location specified by a register pair becomes zero. This Service provides a means for tasks to communicate with each other through a mutually agreed upon memory cell. Consider two tasks: the first task's only function is to monitor the pressure in a gas-filled detector. If it falls below a specified level, the program is to turn the detector's high voltage off and notify the experimenter. The second task's function is to write messages for a number of tasks. The code in the former program, after the voltage has been turned off might look like:

```
LXI    H,COMCEL3      WE AGREED UPON THIS CELL
MVI    M,23           WRITE MESSAGE 23
RELTIL                I'LL WAIT HERE TIL WRITTEN
```

A portion of the second task's code could be:

```
TST3   LXI    H,COMCEL3      DOES TASK3 NEED SOMETHING?
       XRA    A
       CMP    M
       JZ     TST4           JUMP IF IT DOES NOT
       CALL   WRITIT         WRITE MESSAGE
       MVI    M,0            TELL TASK I'M DONE
TST4   LXI    H,COMCEL4      DOES TASK4 NEED SOMETHING?
```

In this illustration the subroutine WRITIT would preserve the value of the register pair and also identify the message.

If the progammer knew that all messages to be written had starting addresses 100H or above, the use of this Service could be made more powerful. Rather than place a message code in the communication cell, tasks would insert the message address in two adjacent memory locations. The most significant byte of the address would be placed in the communication cell. The writing task could then be made more completely general. There would be no need for it to know all present or future messages. In addition, the programmer could set aside some surplus communication cells that future tasks could use. These cells would be cleared during monitor initialization and the overhead for the convenience would be small.

This Service is also useful in single- and double-buffering applications. If one task acquires data but another processes it, then RELTIL can be used to prevent the acquisition task from writing over the data before it can be processed, as well as prevent the processing task from beginning to operate on the data before the acquisition is complete.

TIMER A task can utilize the monitor's clock by the TIMER Service. It can be used in two ways. A single eight-bit register specifies the number of clock "ticks". If the most significant bit is set, the time is "absolute".

```
MVI    A, 80H + TICKS
TIMER
```

If TICKS is equal to the number of "ticks" in one second, the task will resume execution when the monitor clock strikes the second mark, even if the call was made a hundredth of a second prior to the event. If the most significant bit is reset, an "interval" timer is established. The task will resume execution after the interval has expired regardless of when the call was made with respect to "absolute" time.

As an example how TIMER might be used, consider a task that tunes the frequency of a laser with the voltage generated by a DAC and records the current through a photo-diode with an ADC.

```
       LXI    H,DATBUF       DIODE DATA START HERE
       MVI    B,0            INITIAL DAC SETTING
AGAIN  CALL   WRTDAC         WRITE DAC
       MVI    A,6            SIXTY MILLISEC TIMER
       TIMER
       CALL   READADC        READ AND STORE DATA
       INX    H
       INR    B
       JNZ    AGAIN          JUMP IF NOT DONE
```

This case assumes eight-bit resolution and accuracy, as well as a combined settling time of the laser and diode of sixty milliseconds.

EXIT If a task has completed its operation it may become dormant by using the EXIT Service. The task is not removed from the Queue and may be brought back into service at some future time. If this happens, the Taskmaster begins execution of the task at its starting address. The register storage area is initialized to the point assigned by the monitor when it started operation. This Service is used when a task must be assured that certain hardware or software initial conditions are met before it executes its purpose.

SUSPND This Service performs a function similar to that of EXIT. A task which has completed its operation can become dormant by using SUSPND. However, when some other task requests that it be brought back into service, it continues its operation at the point it left off. All registers are preserved and the task can bypass or perform whatever portions of initialization it wishes.

ACTIVA A task may request the execution of another task by using ACTIVA. The desired task is specified by entering its priority ("N") in a register. This is the only Service that modifies the contents of a register between the time of its call and subsequent return to the calling task. This change indicates the status of the desired task at the time ACTIVA was used (task non-existent, already started, etc.). ACTIVA is usually called to activate tasks that have become dormant via the EXIT or SUSPND Services, or which have never been active.

This Service would be used by a command interpreter such as would appear in the software for the control of the spectrograph in our earlier discussion. This task would determine the magnet current desired by the experimenter, place the appropriate DAC setting in RAM where the current setting task could find it, and then activate that task.

Taskmaster Operation

The Taskmaster has the responsibility for determining which task is to use CPU time. It also must maintain each task's environment during a context switch. This means that when a task releases its CPU time (voluntarily if it calls a Service; involuntarily if, for example, a timer expires), the Taskmaster must save all operating registers and the location to return to when the task is brought back into service. Once the decision is made as to what task to start or restart, the Taskmaster must restore all operating registers to the entering task's saved values and begin execution at the correct location.

The Taskmaster uses the tasks' TCBs to store and retrieve all pertinent information about the programs. The structure of the TCB is shown in Table 1.

Table 1. Task Control Block (TCB) Structure

BYTE	USE
11,12	NEXT TCB ADDRESS
9,10	INITIAL STACK POINTER
7,8	"RELTIL" CELL ADDRESS
5,6	STARTING ADDRESS
3,4	CURRENT STACK POINTER
2	TIMER IDENTIFICATION
1	I/O IDENTIFICATION
0	STATUS

The Taskmaster identifies the departing task's TCB by referring to the Current TCB (CTCB) cells in RAM that it maintains. It also knows the reason for any context switch and stores this in the departing task's TCB status byte. Each task has its own stack space in RAM, and the value of the stack pointer is stored in TCB(3,4).

The Taskmaster examines the CPU Status Word (CPUSW) to decide in which direction to go in the Queue to find the next task to use CPU time. If, for example, a timer has expired or the departing task has called the RELTOP Service, the Taskmaster will immediately examine TCB0. If there is no request to "go to the top", it will extract the next TCB address from the departing task's TCB (bytes 11 and 12).

The decision to start a task is based upon the bit pattern found in the status byte (see Table 2). If the task is dormant (bit 7 set); because it has never been started, or has called the EXIT or SUSPND Service, the Taskmaster will immediately move on to the next task in the queue. If the task is not dormant, the remaining bits are examined in sequence to determine if it can be restarted. If the task is waiting for an I/O operation to come to completion, or a TIMER to time out, bits 6 or 7 will be set. In the former case, the bit found in byte TCB(1) is compared with cell IOSTA, and in the latter, the bit found in TCB(2) is compared with cell TMSTA. These cells record the completion status of I/O operations and timers, respectively. If there is agreement in either case the appropriate bit is cleared both in the cell and the TCB. The appropriate "wait" bit in the status byte is cleared and the task is restarted. If there is no agreement the next TCB is inspected.

The next bit to be tested, bit 4, would be set if the task had been interrupted by the completion of an I/O operation or any timer expiration. It would have been set by the task itself had it used the RELINQ, RELTOP, SUSPND, or ACTIVA Service. If the bit is set, the Taskmaster will start the task.

If the task had used the RELTIL Service bit 3 would have been set. The Taskmaster will test the cell whose address is recorded in TCB(7,8) to determine whether or not to resume this task.

Table 2. Bit Assignments in Status Byte

BIT	IF BIT SET, TASK IS
7	DORMANT
6	WAITING FOR I/O
5	WAITING FOR TIMER
4	RELINQ OR INTERRUPTED
3	RELTIL
2	STARTING UP
1	LOADED
0	(UNUSED)

Description of TImer

The TIMER Service allows the programmer to use the clock without having to be concerned about the details of the hardware or the enabling or disabling of interrupts. However, in order to reduce the overhead involved to provide this convenience, certain compromises had to be made.

The monitor has only eight timers that it can allocate to tasks. These are assigned on a first-come, first-served basis. If all have been allocated, the ninth task's TIMER call is treated as a modified RELINQ request. It is modified in the sense, that when the Taskmaster returns to the task, it will restart it at the TIMER call. The potential for introducing delays of high priority tasks is alleviated somewhat by the fact that the expiration of any timer causes a context switch to the top of the Queue. If the ninth task happened to be the one with the highest priority, it would automatically aquire the first available timer.

One other timing feature is built into the monitor. There is a PUMP CLOCK that is reset on every context switch. This clock will time-out, however, and force a switch if none has occurred after a specified period of time.

I/O and Interrupts

Up to this point, very little has been written about I/O operations except that,

a) there is a RAM cell labeled IOSTA that identifies the completion status of up to eight I/O operations,
b) there is a bit in the TCB status byte that indicates that the task is waiting for I/O completion, and
c) there is a byte in the TCB that the Taskmaster can compare with IOSTA to determine which I/O operation is being sought by the task.

Furthermore, nothing has been written about interrupts, except that the TIMER service permits access to the clock without having to deal with them.

In many applications, timers and I/O status checking are all that is necessary to implement a successful control and monitoring system. However, there are cases in which sophisticated I/O and interrupt handling are required. Since the programming of these important elements are hardware specific, no attempt has been made to incorporate any options within the monitor (e.g., CAMAC READ or WRITE Services); rather, a general approach is outlined.

There are two aspects of the monitor that must be understood before one implements I/O or interrupt routines. First, interrupts are disabled during the entire time the Taskmaster is performing its deliberations. The period of time they are disabled depends on what caused the context switch (Service call or timer completion) and the number of TCBs scanned before the Taskmaster enables interrupts upon its restarting of a task. Empirically, one can expect to be able to handle interrupts at a kHz rate and still have the system perform its function.

The second point to be understood, is that the IOSTA cell and the status bits mentioned at the beginning of this section, are designed for message completion purposes. If a single interrupt, or a single byte received by a USART, represents a message, these "hooks" into the monitor should be used. However, it is doubtful that reliable data transmission would take place at a kHz rate, if the monitor message completion features are used for every interrupt. Furthermore, low priority tasks would most likely be locked out.

The I/O and interrupt software can be embodied within the framework of a task. This task can be structured in several ways. It could operate completely independent of the monitor by making itself dormant (EXIT) after it performs its initialization process. An example might be an interrupt-driven routine that continuously refreshes a CRT with data from RAM which are updated by some other task. Another structure could be that of a task and an I/O handler combined. The handler would manipulate IOSTA and the task's TCB status and I/O identification bytes. The monitor would resume the task at the completion of the I/O operation. Other tasks could make use of this I/O capability by calling the ACTIVA Service.

A third structure could be that of either of the previous forms but it would manipulate IOSTA and the TCB of any task. This task would provide one or more Common Services. A discussion of this form of task will not be entered here as it involves detailed knowledge of the Taskmaster.[2]

Actual Implementation

The monitor was first installed in a Kinetic Systems Model 3885 Microcomputer.[3] The code uses 831 bytes of ROM. The amount of RAM required by the monitor depends on the number of tasks to be handled.

$$\text{RAM Storage} = 29 + (N + 1) * (13 + 64)$$

where "N" is the priority of the lowest priority task. These locations are allocated from the top of memory downward. The first 29 cells are CPUSW, CTCB, timers, etc. The remaining storage is for the TCBs and push stacks. Since this hardware implementation starts RAM at location 0, the clock interrupt and Service vectors are "down loaded" to the first 64 locations.

The hardware clock was set to provide a 'tick" every 10 milliseconds. Therefore, the maximum interval the TIMER Service can furnish is 1.27 seconds. Longer intervals must be obtained by repetitive calls for the Service. The PUMP CLOCK expires after five "ticks" or 50 milliseconds.

The amount of time to perform a context switch depends on whether a Service or timer expiration caused it. In order to provide an example of the time periods involved, the following measurements were made using two tasks that produced timing pulses on an oscilloscope. The first task cleared a memory cell, generated a pulse, then called RELINQ. The second task was waiting for the cell to clear (RELTIL). Once the task was restarted by the Taskmaster, it generated another pulse and stored data in the communication cell. Each task was a consecutive loop. The time interval between pulses was ~190 μsec. The second task was then moved down the queue with dormant task TCB's in between it and the first task. Each decrease in priority resulted in an ~18 μsec increase in the interval between pulses.

The K.S. 3885 is in a CAMAC crate with an auxiliary crate controller. The crate is on a serial highway of the Oak Ridge Isochronous Cyclotron control system. Part of the application was to replace a digital voltmeter and mechanical cross-bar scanner with a faster, more modern, piece of equipment. The system was to permit the control computer to continuously scan the 96 data channels at a 100 Hz rate (the old system permitted one channel every two seconds and could not be operated continuously). A manual control panel and LED display were also to be provided for off-line operation. The actual digitization of the analog signals is performed by an integrating ADC at a 15 Hz rate. These are incorporated into eight

commercial data scanners[4] that provide the information in ASCII form over a 9600 baud 20 mA current loop. The command-reply sequence to poll each of the scanners and the structure of the messages is such as to keep almost continuous traffic on the loop. The goals of the project, as well as some additional features are easily handled by the monitor. Its usefulness can be demonstrated by a description of each of the tasks:

- a data acquisition task that interrogates the eight scanners.
- data acquired by the previous task are double buffered. This task performs checksum verification of the data in one buffer while the other is being filled. It also interprets error codes from each scanner.
- the data are converted to binary for use by the control computer.
- a communication task that passes the data to the control computer when requested.
- a task that supports the manual control panel and display. The operator can view any channel's data at a 1 Hz refresh rate.
- a task controls a phoneme-based voice synthesizer. This hardware is in a CAMAC module and can vocalize messages locally or over the public address system. These messages can be initiated by the control computer or the task described next.
- the status of 64 alarm indicators is examined by a task that passes the information to the voice synthesizer task if the operator has enabled the feature via the control panel.
- the debug program,[1] which was expanded to permit the manipulation of the Queue. The ability to allocate TCBs to RAM-resident tasks provides a valuable means for program check-out.

A measure of the effect of the entire system's overhead on a low priority task can be seen in the following example. When the system is operating, the debug program will display a CRT page of memory cells in 11% more time than it would in a non-multi-task system.

Acknowledgements

The author wishes to express his appreciation to B. J. Casstevens of the Computer Sciences Division for useful discussions during the development of the monitor and D. C. Hensley for the time he has taken to understand this work, improve the manuscript, and present it at the Conference.

References

1. C. N. Thomas, private communication.

2. C. A. Ludemann, ORNL Technical memo (to be published).

3. Manufactured by Kinetics Systems Corporation, Lockport, Illinois.

4. Manufactured by Analog Devices, Norwood, Massachusetts