Operating system design considerations for microprogrammed mini-computer satellite systems*

by J. E. STOCKENBERG, P. C. ANAGNOSTOPOULOS, R. E. JOHNSON, R. G. MUNCK, G. M. STABLER, and A. VAN DAM

Brown University
Providence, Rhode Island

INTRODUCTION

The operating system described in this paper was developed as part of research sponsored by The National Science Foundation and the Office of Naval Research on satellite processing and symbolic debugging of data structures. This system runs on a small (32K bytes), dual processor, microprogrammable computer equipped with a high-speed graphic display unit and attached in satellite mode to the multiplexor channel of an IBM System/360-67.

In addition to providing the support for these major research projects, the system is intended for use by a variety of applications, both stand-alone and satellite. To support these varied uses an operating system has been designed which attempts to tailor many features introduced in other, usually larger, systems to the particular needs of these applications, and at the same time introduce new features that better meet the research requirements.

The system, which has been in use since August 1972, is implemented in “levels of abstraction” as defined by Dijkstra. Firmware has been included to support the implementation of these levels and to provide automatic storage allocation and status-saving upon program entry. The lowest level of the operating system creates an extended machine which provides an environment that supports multiprogramming. This environment is similar in concept to that described by Hansen. Communication between the levels of the system is provided by the introduction of a data structure called the event table.

CONFIGURATION

The Brown University Graphics System (BUGS) configuration consists of dual microprogrammable Digital Scientific Corporation META 4 processors, sharing 32K bytes of memory. The ‘A’ processor, which has been microprogrammed as a general purpose computer, is attached to the multiplexor channel of an S/360-67 and has attached to it a card reader, a keyboard/console, and a megabyte removable cartridge disk. The ‘B’ processor is being microprogrammed with a subset of the general purpose instructions of the ‘A’ processor and a set of high level graphic display instructions. The ‘B’ has attached to it a high-speed, high-resolution Vector General CRT. The Vector General is equipped with a joystick, lightpen, function keys, control dials, and a keyboard. A small, very fast (less than 30 nanosecond instruction time) special purpose microprogrammable processor will also be attached to the ‘B’. This processor, which uses a writeable control store, can be dynamically set up to perform a variety of matrix and vector calculations including rotation, translation, perspective, windowing, scaling and clipping. The operating system being described runs primarily on the ‘A’ processor.

SPECIFIC GOALS

A simple general purpose operating system for use with predominately graphical applications

BUGS is intended to be a small general-purpose computing system to be used predominately by graphics applications. In general, graphics applications do not make any unusual demands on the system under which they are running. Graphics applications probably do more data structure manipulation and accessing, as well as more character manipulation, than a “typical” nongraphics application. However, in this respect they are not very different from a compiler. Since graphics applications tend to be interrupt-driven, they will run better and be easier to develop if the system has an efficient and easily interfaced interrupt mechanism. In this respect they are similar to heavily I/O-oriented applications. So while it is true that graphics applications are not really unusual, they do have a combination of characteristics that influenced the design of BUGS.

Investigation of the advantages offered by microprogramming

A major goal of this project is to investigate what advantages the flexibility of microprogramming offers to
both system and application programmers. The effects being examined concern program speed and size as well as certain harder to evaluate aspects such as programming and debugging ease. To facilitate experimentation in this area the operating system has been designed to allow operating system functions or parts of these functions to be moved from the software to the firmware with a minimum of change to the operating system and with no impact on user programs.

HARDWARE (FIRMWARE) CONSIDERATIONS

S/360-Like instruction set

The lowest level of BUGS is that presented by the firmware. A detailed description of the firmware implementation can be found in the META 4A Principles of Operation.1 Basically, the firmware-defined architecture is similar to that of the IBM System/360. Information is stored in main memory in individually-addressable eight-bit units called "bytes". Bytes may be accessed separately or grouped together. The most common grouping consists of two bytes and is called a "halfword". The sixteen registers and the fixed-point numbers operated upon by arithmetic instructions are halfwords.

Of the sixteen registers, three are special purpose, the rest general purpose. Register 0, called the Machine Status Register (MSR), contains information required for proper program control and execution. Register 1 is the Program Counter (PC). Register 15 is the Stack Frame Pointer (SFP). Its use is described in a later section. The system has no floating-point or decimal support.

Improvements over 360 instruction set

The system has S/360-like storage-to-storage instructions for handling variable length character strings. In addition to the standard S/360 version in which the string length is included explicitly in the instruction, each storage-to-storage instruction has a form in which the length is placed in a register and the register is specified in the instruction. This allows the handling of character strings up to 32K bytes in length.

To provide enhanced data structure and character-string manipulation facilities several instructions have been added to the standard S/360 set. These include a Search instruction for processing linked lists or tables for a key which holds some relation to a search argument. Enqueue and Dequeue instructions for manipulating the elements of linked lists are included for use in conjunction with the Search instruction.

The standard S/360 Translate and Translate-and-Test instructions are provided, as well as a Translate-and-Test which scans from right to left. Several additional scan instructions are included for scanning left or right for equality or inequality with a specified character. For manipulating pushdown stacks, Push and Pop instructions are provided which can be used for stacking registers or storage.

The machine includes other significant non-S/360 features. A group of instructions has been added which includes 16 bits of immediate data. Since this is the size of the registers and the fixed-point data items, these instructions are very useful and allow a four-byte instruction to perform the same function that would take six or eight bytes on an S/360.

360-Like PSW swap interrupts

The BUGS system has an S/360-like interrupt mechanism with the equivalent PSW swaps. There are four types of interrupts on the 'A' processor: SuperVisor Call (SVC), Program, Local I/O (which includes the 'B' processor), and those generated by the S/360.

Stack frames

Probably the most significant departure from standard S/360 architecture in BUGS is the use of stacks for subroutine calls. These stacks are similar in function to the Multics stack segment. Programs which require the saving of registers and optional dynamic work storage can have this service performed by executing an ENT (ENTer routine) as the first instruction. Upon execution of this ENT instruction the stack is updated in the following manner:

The first figure shows only the caller's stack frame. The second shows both the caller's and subroutine's stack frames. During the execution of his program, the user must not modify the SFP. The amount of dynamic storage needed by the routine is specified as an immediate field in the ENT instruction. To return to the calling rou
tine, a program issues a RET (RETurn from routine) instruction. This instruction restores the stack to its previous state and returns control to the calling program with all registers restored. The firmware controls only the allocation of storage within the stack. If, during the execution of the ENT instruction, the firmware determines that insufficient space remains in the stack to perform the required allocation, a program interrupt is generated and the operating system allocates a stack extension and completes the ENT processing.

I/O COMMANDS

Firmware support for the local I/O devices and the S/360 is very low level. A single command can be sent to a specified device by use of the Start I/O (SIO) instruction. Typical commands allow for the writing of a single character to the console, moving the disk arm a specified distance and direction, or the reading of a sector of information from the disk. The execution of most commands is performed asynchronously by the specified device. The completion of these commands causes an I/O interrupt.

BASIC CONCEPTS

Multi-level operating system

The specific goals outlined in the third section of this paper have led to a number of basic design conceptions. The first concept is motivated by a desire to produce a system in which the interfaces between the various parts of the operating system and between the user and the operating system are as precise and well-defined as possible. The need for such well-defined interfaces arises from the fact that the operating system is basically a research tool and will be continually changing. The system has been designed and implemented as a hierarchy of levels to minimize the effects of these anticipated changes.

The immediate effect of this design is to blur the boundary between firmware and software. The program on LEVEL1 "sees" only hardware below, with the characteristics of that hardware being simulated by LEVEL0 and the actual firmware. It is therefore possible to test the usefulness of a function by implementing it in LEVEL0, and later moving it into the firmware if that is found appropriate. The LEVEL1 program will not notice the change except in terms of the speed of the function. Thus the "extended machine" concept and microprogramming together provide an ideal environment for research into exactly what a "useful" machine for satellite graphics (or many other things) should look like. This is a significantly different approach from that taken in the Venus Operating System, which incorporates initially in LEVEL0. Rather than viewing the development of the firmware defined architecture as a step that must completely precede the development of an operating system, the BUGS project is attempting to overlap these two steps as much as possible.

The operating system consists of three levels. Lower levels are completely independent of the data structures and facilities of higher levels. Higher levels access the data structures of lower levels only through the use of the lower level routines.

To facilitate the implementation of the lowest level (LEVEL0) of the operating system, the firmware does not perform instruction parse and effective address computation prior to recognition of an illegal op-code. For illegal instructions, the firmware dumps the parsed fields and effective addresses into core, and initiates a program check interrupt. To utilize this feature, each of the functions of LEVEL0 is assigned an invalid instruction of the proper format. This instruction is coded and executed by higher levels of the system as if it were a valid machine instruction. The program check handler, part of LEVEL0, recognizes these instructions and invokes the proper LEVEL0 routine. Because the operands have already been computed (an activity that could take as long as the rest of the requested LEVEL0 function) these functions execute very efficiently. In addition, these functions can be placed in the firmware in the future with no effect on higher levels of the system. These LEVEL0 functions are called extended instructions.

In addition to making it easier to introduce changes into the system, it was expected that a multi-level system would be easier to implement and debug since the complexity of each separate level of the complete system is kept reasonably low. For the same reason it is easier to document the internals of such a system.

Event table

To meet the goal of providing good interrupt handling for graphics applications and also to provide a consistent interface between the various components of the system, a second concept, that of a special data structure called the event table, has been introduced. This structure is not really a table but rather sixteen separate unidirectional linked lists, many of which will be empty. The heads of these lists reside in fixed low-core locations. A typical element in the list will have the following format:

```
<table>
<thead>
<tr>
<th>link field</th>
<th>event name</th>
<th>flags</th>
<th>prty</th>
<th>entry point</th>
<th>stack size</th>
</tr>
</thead>
</table>
```
LEVEL0 uses the event table to provide a logical extension of the firmware interrupt mechanism to higher levels of the system. A more detailed description of this process is included in the next section of this paper.

All firmware interrupts cause LEVEL0 to gain control. Some interrupts, such as those caused by extended instructions, are handled completely by LEVEL0. Most interrupts, however, are reflected to higher level routines. This reflecting is done through the event table. In addition, the completion of asynchronous LEVEL0 functions is reflected through the event table. In this manner events represent interrupts from the extended machine (LEVEL0 plus firmware) to higher levels of the operating system. The event name can identify a specific "extended interrupt" or a class of extended interrupts.

Using facilities provided by LEVEL0, higher levels of the operating system and user programs build the event table. The parameter passing conventions and the environment created are identical for all events, creating one consistent interface between the system and the user. To aid in program checkout LEVEL0 can be made to invoke a specified event using parameters passed to it. This allows the creation of the exact environment that would exist if the actual extended interrupt had occurred. Since the actual scheduling of events occurs at only one point in LEVEL0 the tracing of system and user activity can be efficiently and easily accomplished.

**Multiprogramming**

The final basic design concept was that LEVEL0 should provide a flexible form of multiprogramming which would be fully utilized by higher levels. Because the system is designed for use by a single user or at most a few users, only very basic task coordination and communication facilities are provided by LEVEL0. Applications which require more sophisticated facilities can add this support to higher levels of the operating system, building on the functions of LEVEL0.

**LEVEL0**

LEVEL0 consists of an interrupt handler, a priority dispatcher, a group of extended instruction routines which provide storage management, a Wait/Post mechanism and a simplified I/O facility.

**Storage management**

Extended instructions are provided to obtain and release specified amounts of storage. In addition, an extended instruction is provided to obtain the largest block of free storage. Besides this explicit storage management, LEVEL0 allocates stacks whenever certain types of events are initiated and allocates a stack extension whenever ENT determines that there is insufficient space in the current stack. When RET determines that an entire stack extension is now free, LEVEL0 is notified and this storage is freed.

**Extended I/O**

In terms of the amount of storage occupied by code, the largest function in LEVEL0 is the extended I/O support for the S/360 and the local devices: disk, keyboard/typewriter, card reader, the programmer's panel, and the META 4B. This is due mainly to the very low level of the I/O commands accepted by the firmware.

The LEVEL0 extended I/O support simulates an intelligent channel. With this support higher levels of the system can accomplish a logical unit of I/O work with a single request to LEVEL0. A LEVEL0 I/O request is initiated by use of an EXecute Channel Program (EXCP) extended instruction. The operands of the EXCP instruction are the device number and the address of the start of a channel program. The channel program consists of one or more Channel Program Commands (CPC). CPC's have the following format:

<table>
<thead>
<tr>
<th>command code</th>
<th>flags</th>
<th>data area</th>
<th>data length</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;---1---&gt;</td>
<td>&lt;---1--&gt;</td>
<td>&lt;-----2-----&gt;</td>
<td>&lt;-----2-----&gt;</td>
</tr>
</tbody>
</table>

CPC's are logically equivalent to S/360 CCW's. The allowable command codes and their meanings vary from device to device. Upon completion of all processing required for a particular channel program, an extended interrupt is generated. The particular device and whether or not an error has occurred is reflected in the event name. Event naming conventions are described in a following section.

**Priority dispatcher**

The priority dispatcher allows for the execution of an essentially unlimited number of parallel tasks with priorities specified by the higher levels of the system. The only data structure that the dispatcher accesses is the stack. Each task in the system has its own stack. When the stack is created a header is attached with the following format:

<table>
<thead>
<tr>
<th>stack link</th>
<th>priority</th>
<th>remaining length</th>
<th>current SFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;--------2--------&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This stack is then linked into the stack queue according to its priority. The head of this queue is at a fixed low-core location. The priority dispatcher searches this queue for the first stack that is not in wait state (e.g., one which contains an MSR without the wait bit set). The user’s registers, MSR and PC, are loaded from this stack starting or restarting execution of the task associated with the selected stack. If all users are in wait state, the dispatcher loads the MSR, PC and registers from the last stack on the queue, placing the machine in wait state.

Extended interrupt generation/task creation

When LEVEL0 determines that an event has occurred which it does not handle internally, such as an SVC interrupt, completion of an I/O request, or an interrupt from the graphics processor, it uses information about the event to generate an event name. The event name consists of four hex digits. The first digit generally describes the type of event, e.g., 0 for a timer interrupt, 2 for a local I/O device completion, 'C' for an SVC interrupt. This digit also defines which of the sixteen lists is searched. The meaning of the three remaining digits varies depending on the first digit. For a local I/O completion the second digit is the device number, a third digit of 1 indicates successful completion, a 2 indicates an error, and for an error the last digit describes the type of error. For an SVC interrupt, the second digit is always zero; the third and fourth digits are the SVC number. After LEVEL0 has determined the proper event name, an extended interrupt is generated. This is done by searching the appropriate list in the event table for an entry with the specified event name. If an event entry is found for the specified name, the flag field in the event table element is checked. If this flag indicates that the event is to be processed in parallel with the other tasks running in the system, a stack is allocated for the new task. The size of this stack and its priority are obtained from the event table element. The stack is linked into the stack queue and initialized so that when it is dispatched, the PC will point at the entry point for the routine to handle the event as specified in the event list element.

Certain high priority LEVEL1 event routines do not wait for completion of I/O. (System performance would be degraded by allocating these routines their own stack since nothing is gained by executing them as parallel tasks.) Instead, they are allowed to execute immediately. They run using the stack of the routine that was executing when the event or interrupt occurred. These routines are run disabled and are allowed to use only a restricted set of system facilities.

A range of event names is reserved for applications programs. These programs are allowed to link in event list elements describing events that they wish to handle. These can be events that the system normally supports, such as the interrupt button on the panel; or they can be new events such as user SVC’s. Also, by adding an event table element with a user-specified name (with the entry point specifying the proper routine and with the event flag set to indicate parallel execution) an application program can initiate the routine as a parallel task by executing a ‘signal event’ extended instruction.

The event names are defined, and the event table searched, in a manner that allows a single event table entry to specify a class of events to be handled. If LEVEL0 fails to find an entry in the event table for a particular event that has occurred, the rightmost hex digit of the event name is zeroed temporarily, and the event table is searched again. This process is repeated up to four times, or until a match is found. It is because of this search technique that the event names are defined in the previously described fashion. In addition, no event names are defined with ‘trailing’ zeroes. The fourth search of the event table, if necessary, is always performed with an event name of X'0000'. LEVEL1 provides an event table entry for this event name. The entry point specifies the LEVEL1 supplied DEBUG package.

The result of all this is that the user can have both specific and general interrupt handlers, such as specific routines for function keys 5, 9, and 12 (entries X'nnn5', X'nnn9', and X'nnnC' in the event table) and a general routine for all other function keys (X'nnn0').

In discussing the creation of parallel tasks, the use of the term sub-tasks was specifically avoided. In this system all tasks are independent and equal. No information is retained concerning which tasks initiated which other tasks. In this way, a task’s life is not bound by that of its initiator, nor is its allowable priority range affected by that of its initiator. This does allow for more flexibility, but also requires that the programmer who utilizes multitasking provide his own means of removing unwanted tasks should the need arise.

Wait/Post

In order to coordinate the execution of tasks within this system, Wait and Post task synchronization primitives have been included in LEVEL0 as extended instructions. Both Wait and Post refer to a Wait Control Halfword (WCH). When a Wait is issued, LEVEL0 inspects the specified WCH; if the Post bit (the high order bit) is a ‘1’, control is returned to the user and execution continues. If the Post bit is ‘0’, then LEVEL0 stores the address of the stack header of the user in the WCH and turns on the Wait bit in the MSR in his stack frame. The dispatcher is given control to find another task to execute. When a Post is issued, LEVEL0 examines the addressed WCH to see if it contains a pointer to a stack header. If it does, LEVEL0 finds the MSR in the current stack frame and turns off the Wait bit. LEVEL0 then turns on the Post bit in the WCH and gives the dispatcher control. If LEVEL0 does not find a stack header address in the WCH, it just turns on the Post bit and returns.
LEVEL1

Introduction

The LEVEL1 routines, in combination with LEVEL0 and through the use of LEVEL0, are intended to provide an environment for execution of application programs. Most of the application programmers also use the Cambridge Monitor System (CMS) which runs on Brown's IBM System/360-67. These users are therefore accustomed to the facilities offered by CMS. Because of this, LEVEL1 provides 'CMS-like' support for the local I/O devices. LEVEL1 performs file management and space management for the local disk. LEVEL1 provides a linking loader and a command processing routine to initiate the execution of user programs and LEVEL2 utility routines. An automatically invoked debugging routine is also provided.

The LEVEL1 routines are invoked by issuing an SVC. All of these routines are passed a parameter list in which the first halfword is a WCH. The WCH is posted by the invoked routine when it completes the requested function. This Posting is necessary since these LEVEL1 routines run as parallel tasks after being invoked. This parallel execution allows the maximum overlap between the LEVEL1 routines, which normally perform I/O, and the user program. If the user wants to suspend execution until the LEVEL1 request is complete he must issue a WAIT on the specified WCH.

The LEVEL1 being described represents the standard or full LEVEL1. At IPL-time the user can specify that an alternate LEVEL1 should be loaded. This LEVEL1 may consist of a subset of the standard LEVEL1 or a completely different user written, program.

"CMS-Like" I/O support

For the card reader the RDCARD routine provides the user with up to 80 characters from the next card in the card reader. For the keyboard/console, the user can read a line (TYPEIN), type out a line (TYPEOUT), or type a line and read a reply (TYPEOUT, with reply option specified). Optional editing facilities are available.

RDBUF provides the user with a logical record from a specified file. Deblocking is provided automatically. The user specifies the logical record number so RDBUF can be used to read sequentially or randomly. WRBUF allows the user to write a logical record to an existing or new file. Again, blocking is performed automatically and the records can be written sequentially or randomly. A new file is allocated automatically when WRBUF is issued for a file whose file name is not found in the file directory. A file is extended automatically when a WRBUF specifies a record number beyond the current end of a file.

Disk management

LEVEL1 provides space management on the local disk through the use of an allocation map and a file directory. The allocation map contains a bit for each sector. This bit is on if the sector is allocated or defective. The file directory contains the file name and type and the starting sector number of each file on the disk. Additional information, including the logical record length and file length, is contained in the file header which occupies the first logical record of each file. LEVEL1 uses these data structures to allocate, delete, and extend files. The STATE routine can be used to obtain information from the file directory for a specified file.

Linking loader

The largest part of LEVEL1 is a linking loader. The input to the loader, called a module, is in the form of text decks stored on disk. These text decks are a compacted version of the standard OS/360 text deck. A text deck may represent the output of one or more compilations. The loader relocates the module into any available free storage. Cross-references between the CSECT's within a module are resolved.

Debugs

DEBUGS is invoked automatically when an event occurs for which there is no entry in the event table. The most common instance of this is a program check in the user program. When such an event occurs, DEBUGS informs the user of this fact through the keyboard/console. DEBUGS then provides the user with facilities to display and modify storage and registers, to set a breakpoint and/or origin, and to restart execution at an arbitrary point.

Command processing/parsing

The final function provided by LEVEL1 is command processing. User programs and LEVEL2 utility programs are stored on the disk in the module form. After the system has been IPL'ed and initialization is complete, the LEVEL1 command processing routine accepts a line from the console. The first field in the line typed-in specifies the file name of the user program or LEVEL2 program. This program is loaded and executed. The rest of the line is parsed and passed to this routine as parameters. When this program finishes, it returns to the command processor, which reads another line from the console and executes the next command.

LEVEL2 COMMANDS

The portion of BUGS that exists on LEVEL2 consists of a group of utility commands for file and directory manipulation and inspection. These commands are similar to a subset of the CMS command language. The
that, once allocated, the storage remains in a fixed location to provide storage management facilities for programs of this type of storage. The file record may be a program or termed "dynamically relocatable". Such storage is allocated and freed dynamically, but rather taken is similar to that taken in the SYSEX9 system. When the user requests the use of a record from a file or temporary storage is not allocated and freed dynamically, but rather and to reduce the amount of storage permanently required by the operating system. The approach being taken is similar to that taken in the SYSEX9 system.

Two types of storage are supported by these extensions. "Permanent" storage will be provided as it is currently done by LEVEL0. "Permanent" does not mean that the storage is not allocated and freed dynamically, but rather that, once allocated, the storage remains in a fixed location until freed by the user. The second type of storage is termed "dynamically relocatable". Such storage is allocated and managed by LEVEL0 when a higher-level routine requests the use of a record from a file or temporary work space which meets the conventions established for this type of storage. The file record may be a program or a data item. The program or data that occupies this storage may not contain any absolute pointers to itself or other dynamic storage. Also, these programs will not be allowed to modify themselves. At the time the higher-level or user requests this storage, a base register is specified. LEVEL0 initializes the register for the user and records the assignment in its internal tables. Only limited modifications may be made to these registers.

With the user following these conventions it is possible for LEVEL0 to dynamically relocate programs and data either to reclaim fragmented free space or when bumped items are reloaded. This capability, which was pointed out in the early sixties by Corbató[1], allows LEVEL0 to provide a form of virtual memory to higher levels.

Additional LEVEL0 facilities will be available to allow the user to provide additional information that LEVEL0 can use to determine what data items can be bumped when the need arises.

LEVEL1 disk support will be modified to use the new LEVEL0 support where applicable. This approach allows the programs that meet these conventions to be loaded with a simple disk read. The current LEVEL1 routines, including the loader, meet these requirements and can now be dynamically relocated, reducing the fixed storage requirements of the operating system.

CONCLUSIONS

The operating system has been in use by applications programmers since August 1972. Production usage was preceded by a 12 man/month design phase and an 18 man/month implementation phase. The implementation proceeded so smoothly that the operating system was usable a month ahead of schedule. It is felt that this is directly attributable to two basic design concepts, the more important being the division of the operating system into a hierarchy of levels and, to a lesser extent, the use of the Event Table as a canonical means of passing control between certain levels of the system.

Specifically, the division of the system into levels allowed each of these levels to be developed and debugged prior to its use by higher levels. This greatly reduced the overall debugging task. The Event Table, which was expected to ease the users programming effort, was felt to have provided much the same benefit to the systems programmers. In addition, it provided a convenient means of allowing access to debugging routines during the actual development of LEVEL1 and LEVEL2.

Much experimentation with the system is currently under way. This includes some of the projected experimentation in firmware implementation of operating system functions. So far, the work in this area has been limited to the addition or modification of instructions to improve certain common coding sequences.

The resident portion of the system, when a full LEVEL1 is included, occupies 8K bytes. Although, this is considered acceptable, a desire to reduce this figure is part of the motivation for the extensions outlined earlier.

The Event Table has proven very successful as a means of allowing the user to interface with the operating system. The extended interrupt mechanism using the Event Table meets the requirement of efficient interrupt handling. Less than .25 msec is required to process an interrupt and initiate the execution of the routine specified in the Event Table as a parallel task.

REFERENCES