DATE: September 22, 1983

TO: RD&E Personnel

FROM: Jerry Kazin

SUBJECT: Software Interrupt Mechanism

REFERENCE: Specifications for PRIMOS Condition Mechanism
PE-T-468
Software Interrupt Control Module Proposal
PE-TI-1004
Software Interrupt Control Module Functional Spec.
PE-TI-1005

KEYWORDS: checks, faults, aborts

ABSTRACT

It has been noted that most of the code in the PRIMOS ring 0 kernel needs to be executed in an atomic fashion. Prior to Rev. 19.0 unless specifically inhibited, certain asynchronous events would have interrupted the actions of a module in ring 0. These events fall into a class known as software interrupts. A software interrupt mechanism which keeps these events normally inhibited in ring 0 has been created. This paper discusses software interrupts and this mechanism.

Please note that this revision has been renamed and completely supercedes the prior release, PE-TI-879.

Table of Contents

1 INTRODUCTION

There are three different ways in which a PRIMOS user process may be interrupted, i.e., asynchronously have its path of execution altered.

1.1 The Check

The first and most critical way is via a check. There are currently four check types:

1) power fail
2) memory parity
3) machine check
4) missing memory

They are caused by some trouble with the hardware. Checks cannot be handled by a user process and, most often, cause the machine to halt.

1.2 The Fault

The second way a user process may be interrupted is via a fault. There are currently 11 fault types:

1) restricted instruction mode
2) process
3) page
4) supervisor call
5) unimplimented instruction
6) illegal instruction
7) access violation
8) arithmetic
9) stack overflow
10) segment
11) pointer

They are caused by various events all within the user process except for process fault. Some may be handled by the system on behalf of the user. An example of this type is page fault. Some may be handed to the user via the condition mechanism. An example of this type is access violation fault. The stack overflow fault will cause the user's environment to be initialized. Finally, the process abort may or may not ever be seen by the user. It is currently divided into five different abort types:

1) software interrupts
2) I/O alarm
3) disconnect alarm
4) time-out alarm
5) timeslice end alarm

The first four abort types may be seen by the user while timeslice end

is never seen.

1.3 The Software Interrupt

The third way a user process may be interrupted is, in fact, a subclass of fault. One of the process abort types is handed to the user via the condition mechanism. It is the software interrupt.

The rest of this paper is devoted to descibing the software interrupt mechanism with emphasis placed on how to use the mechanism in ring 0.

2 THE PROBLEM

At Rev. 18, some code was introduced into PRIMOS which allows for the generation of asynchronous software interrupts in a process' execution space. These interrupts usually result in condition signals and subsequent crawlout from ring 0. The crawlout causes the ring 0 execution to be aborted. Such aborts may be quite hazardous to the integrity of a user's process as many pieces of code in ring 0 will produce non-reliable results if they are exited before completion.

3 THE SOLUTION

In the past, these events were normally accepted while in ring 0. One had to explicitly disable these kind of events around critical sections of code that needed to be atomic. This was done by explicitly incrementing a counter named QUITF. The new software interrupt mechanism normally disables software interrupts while in ring 0. When one of these events is now detected while in ring 0, it is normally deferred until execution returns to the outer ring.

4 WHY AND HOW ARE SOFTWARE INTERRUPTS CREATED?

Software interrupts are created by PRIMOS processes which need to interrupt other processes due to certain external events. These events must in some way be relayed to a particular process known at the time of the event. The software interrupt mechanism provides for this function.

The following sections describe these various external events, the software interrupt classes, interrupt priority, the control data base, and how a software interrupt is set up.

4.1 Software Interrupt Types

In Rev. 19, there are currently seven defined software interrupts with associated condition names. They are

1) terminal quit (QUIT$),
2) phantom logout notification (PH_LOGO$),
3) cpu watchdog time out (CPU_TIMER$),
4) real time watchdog time out (ALARM$),
5) cross process signalling (CPS$),
6) the logout condition (LOGOUT$), and
7) IPC message waiting (IPC_MSG_WAITING$).

The first six types are defined for all Rev. 19 while the last type is only defined for Rev. 19.3. All of these events are handled by the software interrupt mechanism.

4.2 Software Interrupt Classes

There are three different classes of software interrupts found within PRIMOS. These classes came into being at different times within the evolution of the system. The following sections describes these classes and when each appeared:

4.2.1 Simple On/Off Interrupt

The easiest class to explain is the simple on/off interrupt. The CPU watchdog timer is one of these. These interrupts take on only an on or off state. There is no chance that multiple instances of these interupts will occur. This class of software interrupt was introduced at Rev. 18.

4.2.2 The Counted Interrupt

The second form of software interrupt is the counted interrupt. Presently, there is only one of these, terminal quit. A counter for each ring is used to determine how many times at a given command level quits have been turned on/off. One may think of these counters as a stack. The software interrupt mechanism itself does not maintain these counters. Separate modules/mechanism which the software interrupt mechanism calls do so. BREAK$ is the module which does this for QUITS. This class of interrupt was introduced at Rev. 17.

4.2.3 Queued Data Interrupt

The third type of interrupt is the queued data interrupt. Phantom logout notification is one of these. The queuing mechanisms for these interrupt types are responsible for queueing the data. The software interrupt mechanism is only responsible for turning the interrupt on/off. This class of interrupt was introduced at Rev. 19.

For the most part, software interrupts will fall into the first class, the simple on/off interrupt. In particular, the introduction of this new mechanism provides the PRIMOS programmer with an alternative which takes the place of the second class of interrupt, the counted interrupt. This is a bonus since more interrupts types can now be easily defined and no new counter management mechanisms need be built.

4.3 Interrupt Priority

Interrupt priority is divided into two catagories, ring priority and signalling priority. The following sections describe these two catagories.

4.3.1 Ring Priority

PRIMOS together with PR1ME hardware has the capability of addressing three rings of operation, ring 0, ring 1, and ring 3. Ring 0 contains the lowest level primitives of the operating system. These primitives encompass such things as process control, file system access, and networking capability. Therefore, this ring is granted the highest privileges with regard to execution environment. Ring 1 is currently not implimented within the software. When implimented it will contain PR1ME subsystems. It will have the second highest execution privilege. Ring 3 is the user ring. It contains PRIMOS items such as the command processor and the CPL processor, and it contains the user application programs. Ring 3 has the lowest execution priority.

As we grant the user the ability to turn on and off software interrupts and since a user resides in ring 3, in order to recognize this enabling setting we must give it priority over anything that the operating system does in ring 0. Remember, software interrupts are normally not seen in ring 0. Disregarding the setting in the outer ring would allow the interrupt to be seen in the outer ring thereby overriding the ring 3 user's setting. Therefore, we have made the rule that all software interrupts will behave in the same manner with regard to ring priority and outer rings have have higher priority than inner rings. In other words, if a software interrupt is seen in ring 0 while enabled in ring 0, before the interrupt condition is signalled we must make sure that the interrupt is enabled in the outer rings.

4.3.2 Signalling Priority

Due to the nature of a time shared operating system, it is possible that more than one software interrupt may be pending at one time. That is, while a user is waiting for the machine, more than one interrupt may occur. For example, a user may type quit and their spawned phantom may complete while the user is waiting for the machine. This leaves two interrupts pending. When the user gains control of the machine, we must see one of these interrupts. The order in which we see software

interrupts is known as signalling priority.

Signalling priority as currently set up within PRIMOS at Rev. 19 orders the acceptance of software interrupts as follows:

1) Logout

2) CPU Timer

3) Real Time Timer

4) IPC Message Waiting

5) Phantom Logout Notification

6) Cross Process Signalling

7) Terminal QUIT

This order is based upon an educated guess at the importance of the various types of interrupts. It is therefore somewhat arbitrary. The order is imposed by the software interrupt handler module, SW$ABT. It can be varied by rearanging the order within SW$ABT. SW$ABT has been constructed to make any reordering of priorities very simple.

4.4 The Software Interrupt Control Data Base

The data base used by the software interrupt control and handler modules, is presently contained within a user's ring 0 stack base, PUDCOM. It is found here because the information relates to a user on a per-user basis, and it must be available to ring 0. All control and handler modules reside in ring 0. The control modules SW$INT, SW$RAOF, and SW$ON are accessable to the outer rings. The control modules SW$MKRCS and SW$ROOFF are available only to ring 0. SW$ABT can only be see in ring 0.

There are five kinds of information stored within PUDCOM that the software interrupt mechanism uses.

1) The first information type tells the mechanism whether or not an interrupt type is pending. Its form is as follows:

```
dcl 1 b_swityp based,          /* pending software interrupts */
      2 terminal bit(1),       /* terminal quit */
      2 cpu_time bit(1),       /* cpu timer */
      2 alarm bit(1),          /* real time timer */
      2 logout bit(1),         /* logout condition */
      2 lon bit(1),            /* phantom logout */
      2 cps bit(1),            /* cross process signalling */
      2 ipcmw bit(1),          /* IPC message waiting */
      2 nu bit(9);             /* not used */
```

Its name is PUDCOM.SWITYP (Software Interrupt Type).

If an interrupt is pending, its associated bit will be on. If an interrupt is not pending, its associated bit will be off.

2) The second information type tells the mechanism, on a per ring basis, which of the on/off interrupt types are enabled or disabled. Its form is like B_SWITYP defined above with the exception that the terminal field is ignored. The names corresponding to the three words in PUDCOM which contain this information are R0SWIN (Ring 0 Software Interrupt Enabled), R1SWIN (Ring 1 Software Interrupt Enabled), and R3SWIN (Ring 0 Software Interrupt Enabled). For each ring, if the bit corresponding to an interrupt type is on, then the interrupt is enabled in that ring.

3) The third information type tells the mechanism, on a per ring basis, whether or not terminal quits are enabled. Its form is that of an integer as it contains a count of how many times quits have been disabled in ring 1 and 3 or enabled in ring 0. Remember, software interrupts are normally on in the outer rings and off in ring 0. The names corresponding to the three words in PUDCOM which contain this information are R0QUIT, R1QUIT, and R3QUIT. The form of these three words is as follows:

```
dcl r0quit fixed bin,        /* ring 0 quit enable count -
                                > 0 quits enabled
                                = 0 quits disabled
                                < 0 should not be this value */
    r1quit fixed bin,        /* ring 1 quit inhibit count-
                                > 0 quits inhibited
                                = 0 quits enabled
                                < 0 should not be this value */
    r3quit fixed bin;        /* ring 3 quit inhibit count-
                                > 0 quits inhibited
                                = 0 quits enabled
                                < 0 should not be this value */
```

4) The fourth piece of information that the software interrupt mechanism uses is the deferred interrupt flag in the FLAGBT word in PUDCOM. It indicates that an interrupt has already been deferred. It only has meaning in ring 0.

5) The fifth piece of information is the not alright to take software interrupts now flag in the FLAGBT word in PUDCOM. It is used as a timing lock and indicates whether or not it is alright to process a software interrupt. It only has meaning in ring 0.

4.5 How A Software Interrupt Is Set Up

The mechanism is broken into two sides, the interruptor side and the interruptee side. The interruptor side is responsible for detecting the external event in question, determining which process should be told of this event, and starting off the software interrupt process. The interruptee side is responsible for noticing that the interruptor

has started the software interrupt process, fetching the interrupt type, and handing this infromation over to the user code.

For purposes of clarity the following discussions will be presented in a procedural fashion. Also, the terminal quit event will be traced. We will assume that the terminal quit will occur when the user is performing file I/O and did not create an on-unit for the QUIT$ condition.

4.5.1 Starting The Process Off - SETSWI

1) When a process notes an external event which is one of the software interrupt types it must first determine to which process this event belongs. The AMLC detects that the user on line 1 enters a terminal quit. It then looks up line 1 and finds that it belongs to user 2.

2) The process calls SETSWI with the proper key for the interrupt type in question and with the user to be interrupted. The AMLC calls SETSWI with the terminal quit key for user 2.

3) SETSWI, based on the passed interrupt key, first sets the proper bit in the selected user's software interrupt type word (PUDCOM.SWITYP). It then calls SETABT with this user number and the software interrupt abort key to setup a process abort for the selected user. SETSWI sets the terminal quit bit on in user 2's PUDCOM.SWITYP and then calls SETABT with user 2 and the software interrupt key.

4) SETABT, based on the passed abort key, sets the proper bit in the selected user's process control block (PCB) abort word. Additionally, if SETABT finds that the user to be aborted is waiting on a software interruptable semaphore, that semaphore is notified. SETABT sets the software interrupt bit on in user 2's PCB abort word and notifies the semaphore user 2 is waiting on if user 2 is waiting on a software interruptable semaphore.

4.5.2 Noticing The Interrupt - The Dispatcher

1) The next time the microcode dispatcher detects that a user is to run, it checks the user's abort flags in its PCB. If it finds any of these flags on, it interrupts the user's normal flow of control by executing the process abort code. The process abort code is defined to be at a location in memory pointed to by the fault table pointer in a user's PCB. The dispatcher notes that an abort flag in on in user 2's PCB. It interrupts user 2's normal flow of control and executes the process abort code.

2) The process abort code fetches the abort flags from the user's PCB and stores these flags in the user's PUDCOM.ABSAVE. Whenever a bit is on in PUDCOM.ABSAVE, we say that an abort is pending. The process abort handler, PABORT, is then called. User 2's abort flags are saved in user 2's PUDCOM.ABSAVE and PABORT is called.

3) PABORT scans PUDCOM.ABSAVE to determine which abort is pending. It then calls a specific abort type handler based on this type. PABORT finds that user 2 has a software interrupt abort pending and calls the software interrupt abort handler, SW$ABT.

4.5.3 Handling The Software Interrupt Type - SW$ABT

1) SW$ABT scans PUDCOM.SWITYP to determine which software interrupt is pending. SW$ABT finds that user 2 has a terminal quit pending.

2) Based on the interrupted ring of execution and the software interrupt control words, PUDCOM.R0SWIN and PUDCOM.R3SWIN, SW$ABT will either immediately signal the condition associated with the detected interrupt, leave the interrupt pending, or defer the interrupt. The rules for signalling, deferring, and pending are as follows:

1) The interrupt is immediately signalled if in ring 3 and R3SWIN indicates alright to signal or if in ring 0 and both SWIN words indicate it is alright to immediately signal.

2) The interrupt is deferred if in ring 0 and R0SWIN indicates not to signal.

3) The interrupt is left pending if R3SWIN indicates not to signal.

For terminal quits, PUDCOM.R0QUIT and PUDCOM.R3QUIT are examined instead of the SWIN words.

For either the immediate signal or the deferal case, the detected interrupt's PUDCOM.SWITYP bit is cleared, i.e., made not pending.

SW$ABT defers the terminal quit as the user is executing file I/O in ring 0. SW$ABT also clears the terminal quit bit in PUDCOM.SWITYP.

4.5.4 Deferring A Software Interrupt - SW$ABT

1) SW$ABT finds the entry frame into ring 0. SW$ABT finds the frame of PRWF$$ through which file I/O is being done.

2) SW$ABT checks to make sure no other faults have happened while in ring 0. SW$ABT finds that no other faults have occurred while in PRWF$$.

3) SW$ABT sets up its stack frame as a condition frame which indicates the approriate software interrupt condition to be eventually signalled. SW$ABT sets up its stack frame for the terminal quit condition.

4) CRAWL_ is then called with the defer crawlout key and the software interrupt crawlout fault intercept monitor (SWFIM_) to build a deferred crawlout frame for the appropriate condition. CRAWL_ is requested to build a deferred crawlout frame for the terminal quit condition.

4.5.5 Building The Deferred Crawlout - CRAWL

1) CRAWL_ builds a new frame on the outer ring into which it places all the infromation passed to it by its caller. CRAWL_ builds a new frame and puts the infromation about the terminal quit into it as passed to it by SW$ABT.

2) The return stack base (SB) and program counter (PB) in the base ring 0 stack frame are copied to the new stack frame. The SB and PB from PRWF$$'s stack frame are copied to the new stack frame.

3) The return SB of the base ring 0 stack frame is set to the new stack frame. The return SB for PRWF$$ is set to the new stack frame.

4) The return PB of the base stack frame is set to the first instruction of the passe crawlout fault intercept monitor. The return PB for PRWF$$ is set to the first instruction of SWFIM_.

5) CRAWL_ then returns to its caller. CRAWL_ returns to SW$ABT.

4.5.6 Making The Deferred Software Interrupt Happen - SWFIM

1) SW$ABT returns to its caller, PABORT.

2) PABORT returns to its caller, the process abort code.

3) The process abort code returns to the interrupted code. The process abort code returns to PRWF$$.

4) When the normal flow of control in ring 0 reaches the base stack frame, flow of control will procede to the software interrupt crawlout monitor. PRWF$$ will return to SWFIM_.

5) SWFIM_ will first call a module in ring 0, SW$RST, to reset the ring 0 part of the software interrupt mechanism, and then signal the condition that CRAWL_ set up. SWFIM_ will call SW$RST and then signal the QUIT$ condition.

6) The condition will be raised and handled by either the user if the user has an on-unit built for this condition or by the default on-unit handler. QUIT$ will be raised and handled by the default on-unit handler as the user does not have a QUIT$ on-unit built.

4.5.7 What If The User Was In Software Interruptable Ring 0 Code?

If the user had been in some piece of ring 0 software interruptable code such as C1IN$, then everything would procede as above except for the following:

1) SW$ABT would detect that the interrupt could be taken immediately.

2) CRAWL_ would be called without the defer key.

3) CRAWL_ would set up the for the normal crawlout fault monitor (CRFIM_), not for SWFIM_.

4) CRAWL_ would not return to SW$ABT. Instead, it would call UNWIND_ to unwind the ring 0 stack. Unwinding is the process of insuring that no further action be taken in the current ring except for final return from that ring.

Note that CRFIM_ would be returned to from ring 0. It would call SW$RST and signal the QUIT$ condition.

4.5.8 What If A Software Interrupt Was Disabled?

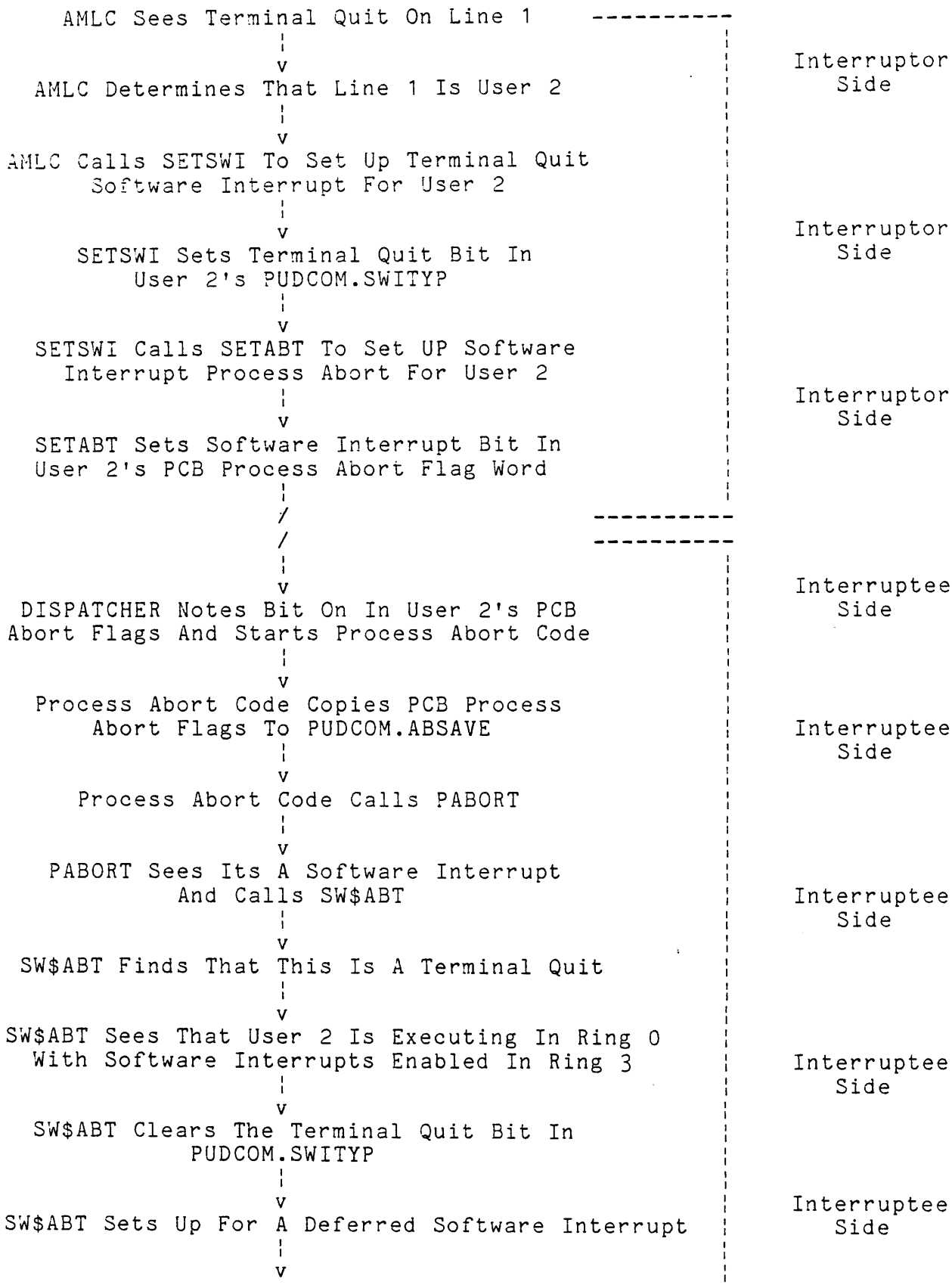
A user can disable a software interrupt. This is done by calling either SW$INT, SW$RAOF, or BREAK$ for terminal quit. See Software Interrupt Control Module Functional Spec. PE-TI-1005 for further details. If this is true then everything would procede as above except for the following:

1) SW$ABT would detect that the interrupt was disabled. R3QUIT would not be 0.

2) SW$ABT would then return without performing any other actions.

Note that the next time this user goes through the dispatcher, the above sequence will be repeated. In fact, it will be repeated until the interrupt is enabled by the user and the signal raised.

The following figure illustrates the sequence of events that occur within the software interrupt mechanism for a deferred terminal quit interrupt.


AMLC Sees Terminal Quit On Line 1
AMLC Determines That Line 1 Is User 2
AMLC Calls SETSWI To Set Up Terminal Quit Software Interrupt For User 2
SETSWI Sets Terminal Quit Bit In User 2's PUDCOM.SWITYP
SETSWI Calls SETABT To Set UP Software Interrupt Process Abort For User 2
SETABT Sets Software Interrupt Bit In User 2's PCB Process Abort Flag Word
DISPATCHER Notes Bit On In User 2's PCB Abort Flags And Starts Process Abort Code
Process Abort Code Copies PCB Process Abort Flags To PUDCOM.ABSAVE
Process Abort Code Calls PABORT
PABORT Sees Its A Software Interrupt And Calls SW$ABT
SW$ABT Finds That This Is A Terminal Quit
SW$ABT Sees That User 2 Is Executing In Ring 0 With Software Interrupts Enabled In Ring 3
SW$ABT Clears The Terminal Quit Bit In PUDCOM.SWITYP
SW$ABT Sets Up For A Deferred Software Interrupt
Interruptor Side
Interruptor Side
Interruptor Side
Interruptee Side
Interruptee Side
Interruptee Side
Interruptee Side
Interruptee Side

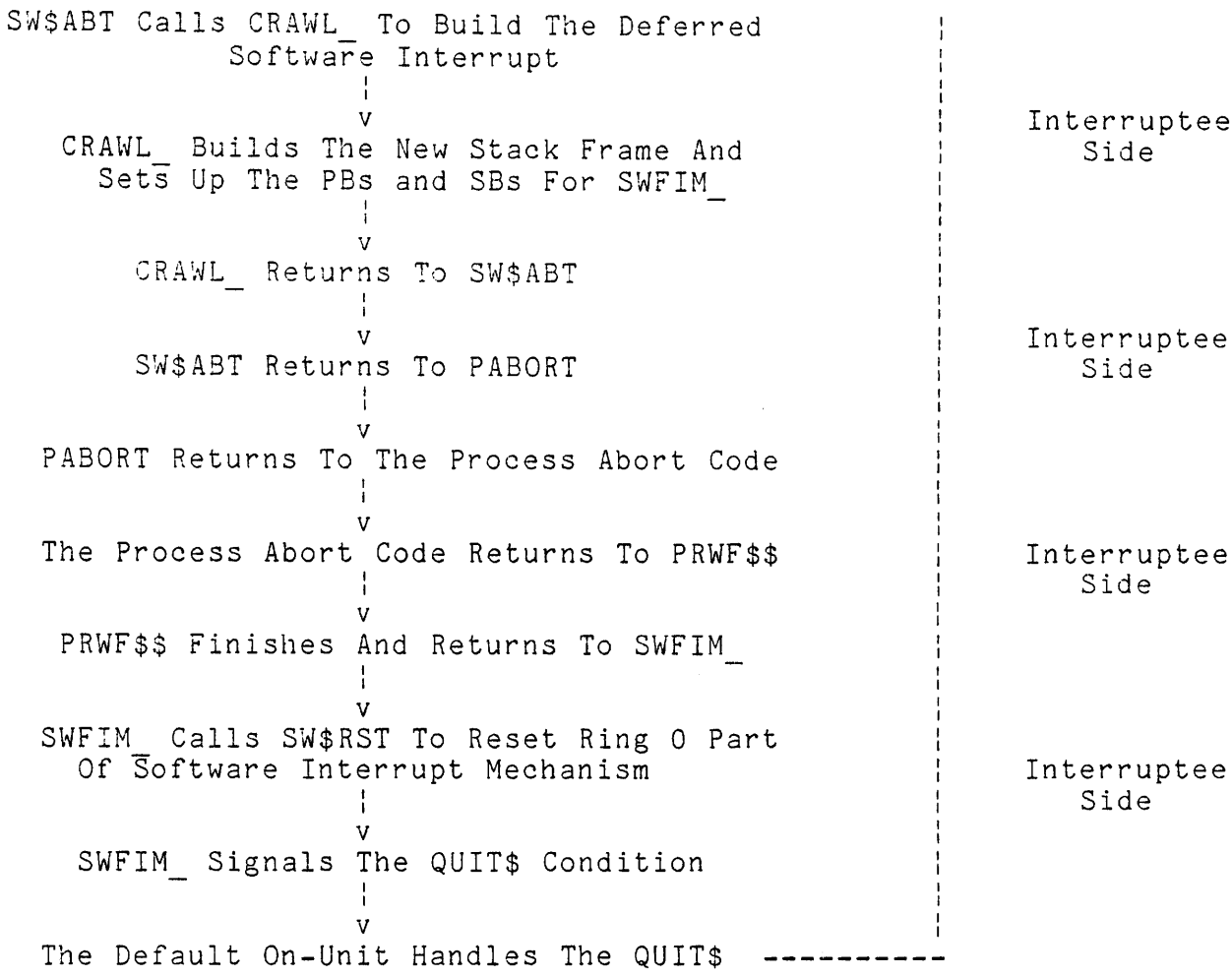


Figure 1 - History Of A Terminal Quit

5 HANDLING SOFTWARE INTERRUPTS IN RING 0

5.1 Comparison Between Old And New Ring 0 Software Interrupt Handling

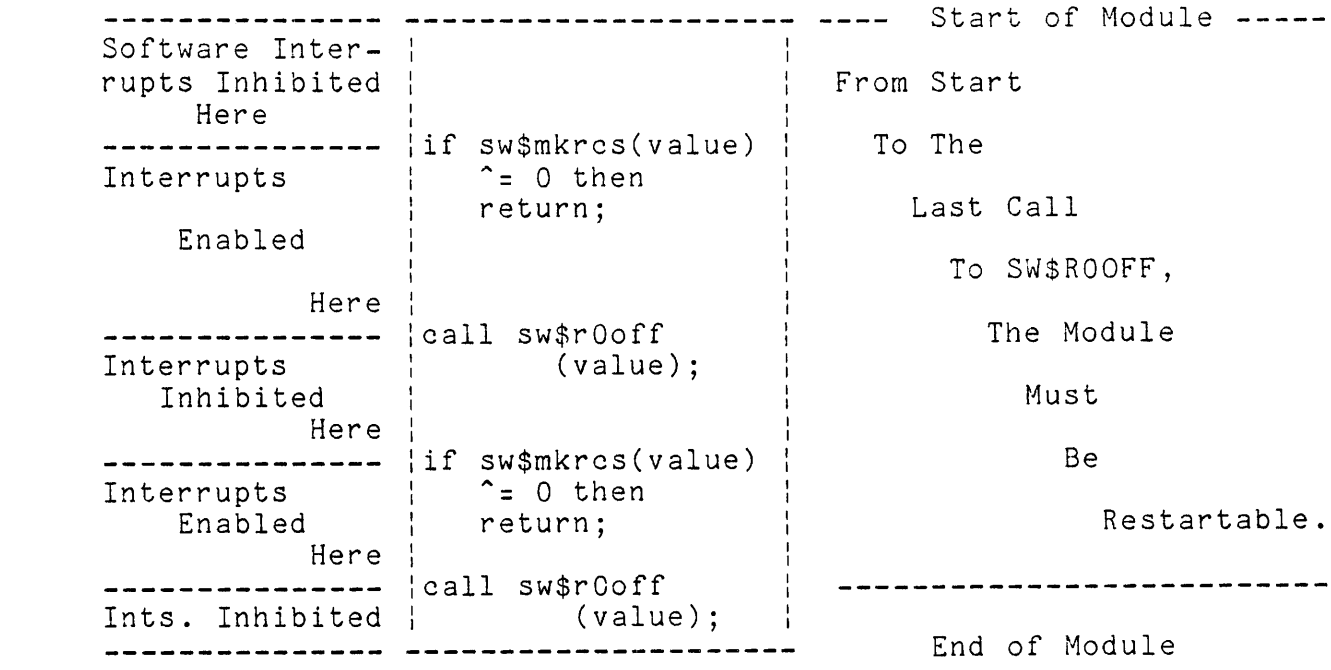
The old mechanism in ring 0 which disabled software interrupts in ring 0 was known as the ring 0 quit mechanism. It allowed a ring 0 module to inhibit software interrupts by incrementing the ring 0 quit inhibit counter, QUITF. When this was done, quit events were ignored. To turn software interrupts back on, a call to QUITON had to be made. It was noted that some of the modules in ring 0 that needed software interrupts inhibited, such as T$AMLC, did not use the mechanism properly. To make sure that these modules used the old quit inhibition mechanism properly would have required a major code audit. After considering the problem and realizing that most of the kernal code does not want to be interrupted by the these events, it was decided to normally inhibit all software interrupts in ring 0.

5.2 Enabling Receipt Of Software Interrupt In Ring 0

To enable the acceptance of software interrupts one must invoke the function SW$MKRCS before entering the interruptable section. One must also invoke the procedure SW$ROOFF to disable software interrupts when leaving this section. This section is called a reverse critical section. Because of the internals of the new mechanism, surrounding an interruptable section with calls to SW$MKRCS and SW$ROOFF is not sufficient. One coding rule must be followed.

> Between the top of a module that is going to allow software interrupts and the final call to SW$ROOFF, the module must be restartable.

Restartable sections of code are those that may be completely scrapped and may be resumed from their beginning. Visually, the following diagram depicts this rule:



Note that between the calls to SW$MKRCS and SW$R0OFF the module is enabled.

Figure 2 - Making Reverse Critical Sections

5.3 Making A Critical Section In Ring 0

Although the software interrupt mechanism normally disables interrupts in ring 0, on rare occasions it may be necessary to insure that ring 0 is in a critical section. This may occur because a prior module in ring 0 has enabled any or all of the software interrupts. To make sure a critical section exists, the following procedure must be followed:

1) Get and save the present value for all counted software interrupts

2) Set the value for all counted interrupts to 0

3) Call SW$INT with the read all off key to get present enabled state which must be saved

To end the crtical section, do the following:

1) Call SW$INT with the on key and the saved present enable state

2) Restore the saved value for all counted software interrupts

5.4 Examples Of Ring 0 Users

As was mentioned above, most code in ring 0 must have software interrupts turned off while executing. Examples are PRWF$$ and TNOU. PRWF$$, the file system read, write, and position a file routine takes certain locks and transfers data to or from a user buffer to a file system buffer. If it were interrupted this tranferal could be incomplete. TNOU prints characters to a terminal. If it were to be interrupted during printing, the possibility exists that duplication of printed characters could occur.

The new software interrupt mechanism insures that the integrity of this class of routines is maintained.

Some ring 0 code must have software interrupts enable somewhere during the course of their lifetime. An example of this kind of code is T$AMLC. When in T$AMLC, a user is waiting for terminal input. During this wait, it must be possible for software interrupts to be handled immediately. Therefore, T$AMLC makes use of SW$MKRCS and SW$ROOFF to allow receipt of software interrupts. The simplest event to understand is real time alarm. This event must be recognized by T$AMLC. If it weren't and the assigned AMLC line went dead, then the process running T$AMLC could be caught waiting forever for characters that would not come. Therefore, T$AMLC makes the reverse critical section.

CRAWL_ is a routine which may execute in ring 0. When performing its normal operations, it must be sure that a critical section exists. Therefore, CRAWL_ presently must explicitly create a critical section even though it is in ring 0. It does this by following the procedure noted in Creating A Critical Section In Ring 0. Since there is only one counted interrupt today, terminal quits, only it is saved and

restored. This is done by use of the BREAK$ interface.

6 MORE INFORMATION ON USING THE SOFTWARE INTERRUPT MECHANISM

This paper explains why the software interrupt mechanism was created, how it works, and how it is sometimes used in ring 0. It does not explain the general usage of the mechanism especially as relates to the ring 3 user. Specifically, this paper does not discuss how to control the enabling/disabling of software interrupts. To find out more about why the enable/disable feature is needed read Software Interrupt Control Module Proposal PE-TI-1004. To find out more about how to use the enable/disable features read Software Interrupt Control Module Functional Spec. PE-TI-1005.