# *TAKING A SHORT BREAK ...*

Michel Kohon
Tymlabs Corporation
811 Barton Springs Road
Austin, Texas 78704

**"Please Do Not Interrupt."**

Tacked to a door, this notice might be appropriate for a Meeting of the Board, or a poetry reading. But if it appeared on a computer, it would certainly belong on the HP3000 under MPE.

Although the architecture of the HP3000 is very well suited for handling interrupts, starting or stopping a process is a painful endeavor for MPE. The :RUN command costs so much that new MPE versions have an auto-allocate procedure. This may be fine for starting a program, but it is no help when switching from one program to another.

I spent a long time at a merchandising company where people work in an interrupt-driven environment. Phones ring, display monitors update commodity prices every 15 seconds, and decisions are made in a very short time. If the merchant is looking at his long/short position and a customer calls about the status of his current shipment, do you think the merchant will wait one minute while his currently-running program shuts down and a new one begins? No! He won't use the computer. But if it took only a few seconds he would.

"Process switching" is the ability to access one program while running another one, a facility which can provide both software developers and end-users with increased power and flexibility. For process switching to be useful, it must be fast, utilizing system resource in a very efficient manner. Could such a facility be implemented on the HP3000?

The MPE BREAK function provides an interrupt capability which is inexpensive in terms of system resource. This paper reports on the inner workings of BREAK, and suggests how this knowledge might fit into the overall prospect for implementing process switching on the HP3000.

**BREAK: A Guided Tour**

To understand precisely how BREAK works, you might begin by looking for some documentation, as we did. Will you be surprised to hear that we didn't find any

documentation on BREAK in the whole MPE manual library? By a stroke of pure luck, however, we managed to find the section in the MPE source code itself which contained the information we needed.

My description of the BREAK function will take us through the intricacies of the MPE operating system, revealing both its sophistication and its weaknesses. Remember that MPE is multi-process oriented; be ready to follow BREAK into several sub-processes.

There are two ways to call BREAK — either by pressing the BREAK key on your terminal, or programmatically by calling the intrinsic CAUSEBREAK.

Let's begin with what happens when you hit the BREAK key on your terminal. CAUSEBREAK will come into play in step 4.

1. The terminal generates an extended spacing condition (a hardware, rather than character, condition) and the terminal driver detects the BREAK, causing the ADCC, ATP, etc. to interrupt with a BREAK status.

2. The driver process calls the procedure BREAKJOB. BREAKJOB is an uncallable, privileged procedure which organizes the user process into the BREAK state.

3. BREAKJOB first hibernates all the user's sons, then sets the user in a BREAK state by turning on a bit in the 1st word of the MPE Logical Physical Device Table (LPDT). Next, BREAKJOB calls the procedure CAUSESOFTINT, which runs on the Command Interpreter stack. CAUSESOFTINT is a general procedure in charge of all types of pseudo-interrupts.

   In order to switch processes, BREAKJOB sets on the user's process pseudo-interrupt bit in the PCB (Process Control Block) using the procedure SETPSIF. It then awakens the C.I. whose PCB offset is passed as a parameter to the AWAKE procedure. The C.I. starts executing the procedure CAUSESOFTINT, as its PCB is now marked as being in BREAK.

4. CAUSESOFTINT calls the entry point SYSBREAK located in the actual C.I. code.

   If a program had used CAUSEBREAK to break, this step would be the entry point in the BREAK circuit. CAUSEBREAK takes care of hibernating the process's sons and awakens the C.I. via a pseudo-interrupt. From here on, things happen in the same way, whether you used the BREAK key or called CAUSEBREAK.

5. But what if OPTION NOBREAK were specified in one of the active UDCs? In that case, SYSBREAK would call FCONTROL to disable BREAK, and return to CAUSESOFTINT, which would re-activate the user's process.

   BREAK is disabled when the UDC is executed, as follows: The UDCINIT procedure calls the SETSERVICE procedure, which sets the LPDT break bit to 1. This indicates that the terminal is already in BREAK (although it is not), effectively voiding any other BREAK.

   This explains why you don't see anything happening when you hit BREAK while under the influence of an OPTION NOBREAK. It also means that you cannot enable BREAK with FCONTROL when OPTION NOBREAK is in effect.

6. If there is no OPTION NOBREAK, word 32 of the C.I. PXFIXED (a MPE table built in every stack) is set to -1.

7. What if there is an ongoing read at the terminal, or the program is waiting for a read (FREAD, READ, ACCEPT, etc.)? This means the user's program is engaged in a procedure called IOMOVE which interfaces the high level intrinsic with a low level one dealing with logical devices.

   This low level procedure is ATTACHIO. ATTACHIO controls the logical transfer of data between any physical I/O device and the stack. It is one of the most complex procedures in MPE. Actually, ATTACHIO is the tip of a deep iceberg which reads from, writes to, and locks any physical device (terminal, HPIB, ADCC, ATP, etc.).

   A word to the wise: If you have to deal with ATTACHIO, make sure the logical device number is valid — unless you want MPE to grant you a System Failure 206!

   ATTACHIO is called by IOMOVE to read the logical device attached to the terminal (for example, LDEV 20). When you hit BREAK on your terminal, ATTACHIO detects it as an error in I/O, and IOMOVE unlocks the terminal file control block (ACB) to let the C.I. process run. (If it is the C.I. itself which is running, the BREAK request is voided).

8. But let's return to SYSBREAK, from step 6. The uncallable FBREAK procedure is called by SYSBREAK (on the C.I. stack) in order to set the terminal file control block (ACB) break bit. During this time, the ACB is locked, to insure that no conflicting access to the file takes place. FBREAK is running on the C.I. stack.

9. Eventually, FBREAK releases the ACB. If there is a read pending on the terminal, IOMOVE tries to lock the $STDIN ACB. But since we are now running the C.I., this attempt impedes the user's process in a low priority queue internal to the file control block vector ("long wait" state).

10. Once FBREAK is completed, the C.I. checks to make sure the calling process was a session before displaying the expected colon. If it was a job from which BREAK was called with CAUSEBREAK, the output buffer is flushed.

11. Now the C.I. is waiting for an acceptable MPE command. Most commands are permissible; however, the :RUN command, and others which imply :RUN, are not. This is understandable since :RUN would create a brother process rather than a son. (Don't forget that the C.I. is now active.) But wouldn't it be nice to have some kind of mechanism to run a program from within a BREAK?

12. When the C.I detects :RESUME, :ABORT, :BYE, or :HELLO, word 32 of the C.I. PXFIXED is set to zero (no longer in BREAK) and the procedure FUNBREAK is called.

13. FUNBREAK locks the terminal file control block (ACB) and sets its ABORTREAD flag to FALSE. This bit will be used by any waiting IOMOVE. An ATTACHIO is fired to tell the terminal driver to clear its BREAK DIT (Device Information Table) so that the current read (if any) begins with data that has already been input and stored in a pending buffer. The ACB is unlocked.

14. If the C.I. detects a RESUME command, it returns to the procedure CAUSESOFTINT. This brings all user's sons out of hibernation and resets the user's PCB to a no-BREAK state.

15. The user's process is no longer impeded, and the program resumes.

    If a read was in progress, IOMOVE can now lock the terminal ACB to complete that read. But before re-initiating the read, IOMOVE displays the familiar (and hard-coded!) "READ pending". (How about a beep, instead of this message printed in the middle of a formatted screen?)

This is the end of your guided tour. BREAK-time is over; your program has resumed.

As you can see, a tour of the BREAK function is almost as complex as a visit to the Hearst castle, and I cannot even be sure that my description is perfect since no documentation is available. In fact, I would be very interested in your comments if you happen to know more about what is actually going on during specific phases of BREAK, and have documentation of it.

Given what we now know about BREAK, how can it be used to implement process switching? BREAK itself doesn't provide a way to actually run a second program; it merely gives access to a subset of MPE commands, which are, for the most part, useless in a business application environment. Two steps are still required: We must seize control of the terminal at some point in the BREAK process, and, once in control, we must make it possible to run a second application.

**The Riviera Interface to BREAK**

As of this writing, the Tymlabs product which will take advantage of this research is not yet on the market and doesn't have a name. For now, I will use the code name Riviera (that is where many Frenchmen like to take their breaks).

In order to take control when you hit BREAK or call CAUSEBREAK, Riviera substitutes its own routine for one MPE procedure called during the BREAK process. Our research indicated that the safest place to substitute would be at the point where SYSBREAK comes into play. Remember, that was step 4, where the two ways of initiating BREAK converged. This was by no means the only possibility, and was certainly not the easiest to implement. A long series of experiments led us to this conclusion — every one of them causing some kind of system failure! SYSBREAK's advantage is that it is an entry point in the system SL, and therefore can be called as a procedure.

To substitute for SYSBREAK, we perform a procedure called "trapping": the actual SYSBREAK is renamed and our substitute SYSBREAK decides when and how to call it. We do not change MPE code, and we let MPE call our uncallable new SYSBREAK, which is located in a special segment of the system SL.

Two smaller problems are also solved with traps in the Riviera interface. In order to provide access to Riviera even when OPTION NOBREAK has been set by a UDC, we trap the procedure FCONTROL with a substitute FCONTROL. Our FCONTROL decides for itself whether to satisfy a BREAK disable request or not. And since Riviera needs to know who is logged on at a specific terminal, we trap INIJSMP, which is called by the `:HELLO` command.

As I mentioned earlier, the challenge in implementing process switching is to use very few resources. For example, we don't want a procedure to have to make a disc access in order to determine whether Riviera is running. Our only choice is to set a flag somewhere in a core-resident table. We decided to use the Job-ID field of the JMAT (Job Master Table) related to the session/job which is actually running Riviera, since this field is not critical for MPE. When Riviera starts, it stores in this field the string "$DESK" followed by the Riviera version number. When Riviera is stopped by the system manager or operator, the original Job-ID is restored. This is not necessary for MPE's purposes, but is useful for any resource accounting or security system which accesses the Job-ID at logon or at logoff.

Every time I use the phrase "Riviera is active," it means that a procedure has determined that the JMAT contains one entry with the string "$DESK". Since there is no way a session or job can log on as $DESK.GROUP.ACCOUNT, and since $DESK cannot be displayed during a SHOWJOB, our trick is both safe and invisible.

If Riviera is not active when BREAK is called, we call the original SYSBREAK. If the user has access to BREAK, he gets the colon. Otherwise he gets the usual nothing!

If Riviera is active when BREAK is called, we first check to make sure we aren't in Riviera itself. (We don't want anyone to interrupt Riviera.) Riviera disables BREAK also (by trapping FCONTROL), but under no circumstances should we allow a BREAK in Riviera.

So, assuming that Riviera is active, but we are not in Riviera itself, what happens when someone hits BREAK?

First, we determine the user's group and account with a procedure which scans the JMAT until it finds the $DESK Job-ID. This procedure then returns the group and account, so that two message files, used to communicate between a user and Riviera, are opened. (These two files, BREAK and RESUME, are built by Central Riviera when it starts operation.)

Once BREAK and RESUME are successfully opened, we send a message to the BREAK file, which has Riviera as its unique reader. We then perform the following 6 steps :

1. Call SETSERVICE to disable BREAK.

2. Wait for a message from the RESUME file.

   The RESUME file reads all messages in a non-destructive manner. Then it checks to see whether the terminal specified in the read record matches the user's terminal number. If it doesn't, we pause for 1 second and keep reading messages (if any). The pause is necessary since we are running in a linear queue and other processes must have a chance to collect their messages which may be placed in front of ours in the message file. If the specified terminal does match the user's terminal number, the message is re-read in a destructive manner, and we skip to the last step.

3. Call SETSERVICE to enable BREAK.

4. Flush the terminal I/O buffers.

5. Call FBREAK to set the terminal ACB in BREAK.

6. Call FUNBREAK to reset the terminal ACB to normal, producing the less-than-desirable "READ pending" message, which we will discard in our ATTACHIO trap.

Riviera is now in control of the terminal, or rather has allocated one of its sons to the terminal. At some point, Riviera's son sends a RESUME message to the user's process. This message specifies whether we want to go into the regular MPE BREAK or to resume.

*First case: we want to call the regular BREAK:*

Our SYSBREAK calls the original SYSBREAK and we are in BREAK. If we were under an OPTION NOBREAK, the original SYSBREAK would kick us out and disable BREAK. This is why our substitute SYSBREAK re-enables OPTION NOBREAK when we come back from the original SYSBREAK by calling SETSERVICE.

*Second case: we want to resume:*

We return from our SYSBREAK and we are back in CAUSESOFTINT, which will resume our program.

If, during our SYSBREAK processing, we find an unacceptable situation, we will call the original BREAK.

**The Door is Now Open to Process Switching**

That is how Riviera takes control when you hit BREAK. Now we face the final step in process switching: to run any application without any modification. Three problems became apparent in our attempts to achieve this.

The first problem we encountered concerned redirection of I/Os. Riviera can run a program using the terminal on which it is active. That is fine, except for the small detail that Riviera is running in, say, session #1, while your program (now in BREAK) is running in, say, session #3 – which means that they aren't using the same terminal. How can we redirect all I/Os of a Riviera son (say Query from HP) to the terminal in BREAK, with a minimum overhead? We accomplished this by using a memory resident table, which links a certain process number to a target terminal.

There are many tables you can use, but let me give you a piece of advice. Some tables contain reserved or seemingly empty fields; avoid them, as HP might decide to use them in the future. Use only the documented fields. And here is the trick: use them in a way which doesn't conflict with MPE. Let me give you an example: Say you want to use the LPDT (Logical Physical Device Table) to indicate a condition of the terminal. The two first bits of word zero are normally set to 1 when the terminal is used. Set them to 3 for a short while and only your program will notice and use that state without any conflict with MPE. Rather than fighting MPE, just try to use it in a very gentle way.

In redirecting I/Os, we needed to trap FREAD/FWRITE/FCONTROL, but also the primitive ATTACHIO in order to bypass MPE file system. Each one of these procedures quickly checks our memory resident table to see if the I/O needs to be redirected.

The second problem we were faced with concerned the "environment", that is, the User/Account/Group in which the program is running. Riviera might be running as MANAGER.TYM,PUB, but the terminal in BREAK is most likely using a different

User/Account/Group. In order to access the correct JCWs, File Equations, and temporary files, as well as the correct posting of both elapsed and CPU time used, we needed to change the environment of the program running on behalf of the terminal in BREAK.

JCWs, File Equations, and temporary files are in tables whose addresses are kept in another set of tables called JIT and JDT, which are job related (one entry per job/session). The addresses of these tables are kept in each process stack (in the PXGLOB, to be precise), so we replace the JIT/JDT addresses in the program stack with the addresses contained in the stack of the process in BREAK, and also the JIT/JDT session number and main pin number.

At this point we could run any program from a terminal in BREAK and it would behave as if it were launched from that terminal except for a small detail: Break and Control-Y will not work! This was the third problem we faced: Since your terminal is already in BREAK, the next time you hit BREAK or Control-Y, MPE ignores it at the driver level (remember step #2 of the BREAK description). The only available solution was to trap the BreakJob procedure.

If you are already in BREAK and you hit BREAK during a read, two things happen. First, our BreakJob procedure sets a bit in the LPDT, which cues another process to send a message to the Riviera server so that the Riviera menu can be presented to the user. Second, FREAD also detects the BREAK and the program suspends itself so that next time you select that program it will resume execution at the exact point where it stopped! If you hit BREAK (while in BREAK) during anything else but a read, we emulate a Control-Y. If your program doesn't have a Control-Y trap, we terminate the program.

Now that these last three problems are solved, it is possible to run another program while in Riviera.

**The Ultimate**

I have described one way to implement process switching on the HP3000. As you have seen, it is no simple affair. It must handle scattered data – and unlike IMAGE-oriented products, or backup products, or compilers, process switching deals with more than one object inside MPE.

From a user's perspective, on the other hand, process switching is a breeze: Hit BREAK, and you are in the Riviera; once there, you can do something different.

In other words, you can run an application, hit BREAK, and select another program from a menu; run that program, hit BREAK again and select another program, and so on, without ever terminating any of those programs. You may be familiar with the Macintosh "Switcher" product. This is exactly what we now have on the HP3000. For example, while you are running your online order entry system, you can quickly access HPMail from time to time to check your mail. You can also test a program

with QUERY, QUEDIT, MPEX and PDQ for Quiz all at hand. By suspending existing processes rather than terminating them, not only do you avoid the overhead of termination and creation; you also avoid the overhead of opening the files or data bases that these programs are using.

The Riviera product described here has been years in the planning and implementation. I don't recommend that you try to implement process switching on this scale as part of another software application. However, implementing a more limited system based on MPE process handling features, such as child processes, message files, and MPE's Control-Y, is certainly feasible.

So there you have it: How to take a break on the HP3000, and what to do with it.