

North American Response Center

# HP 3000 APPLICATION NOTE #34

## Process Handling (Using COBOLII Examples)



 **HEWLETT  
PACKARD**

October 1, 1987  
Document P/N 5958-5824R2740

**HP Computer Museum**  
**[www.hpmuseum.net](http://www.hpmuseum.net)**

**For research and education purposes only.**

## RESPONSE CENTER APPLICATION NOTES

HP 3000 APPLICATION NOTES are published by the North American Response Center twice a month and are distributed with the Software Status Bulletin. These notes address topics where the volume of calls received at the Center indicates a need for addition to or consolidation of information available through HP support services.

Following this publication you will find a list of previously published notes and a Reader Comment Sheet. You may use the Reader Comment Sheet to comment on the note, suggest improvements or future topics, or to order back issues. We encourage you to return this form; we'd like to hear from you.

### NOTICE

*The information contained in this document is subject to change without notice.*

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This document contains proprietary information which is protected by copyright. All rights are reserved. Permission to copy all or part of this document is granted provided that the copies are not made or distributed for direct commercial advantage; that this copyright notice, and the title of the publication and its date appear; and that notice is given that copying is by permission of Hewlett-Packard Company. To copy otherwise, or to republish, requires prior written consent of Hewlett-Packard Company.

Copyright © 1987 by HEWLETT-PACKARD COMPANY

# PROCESS HANDLING

The purpose of this document is to give experienced programmers a guide to the use of process handling techniques. COBOLII is used in the examples since it is a common language of HP 3000 users.

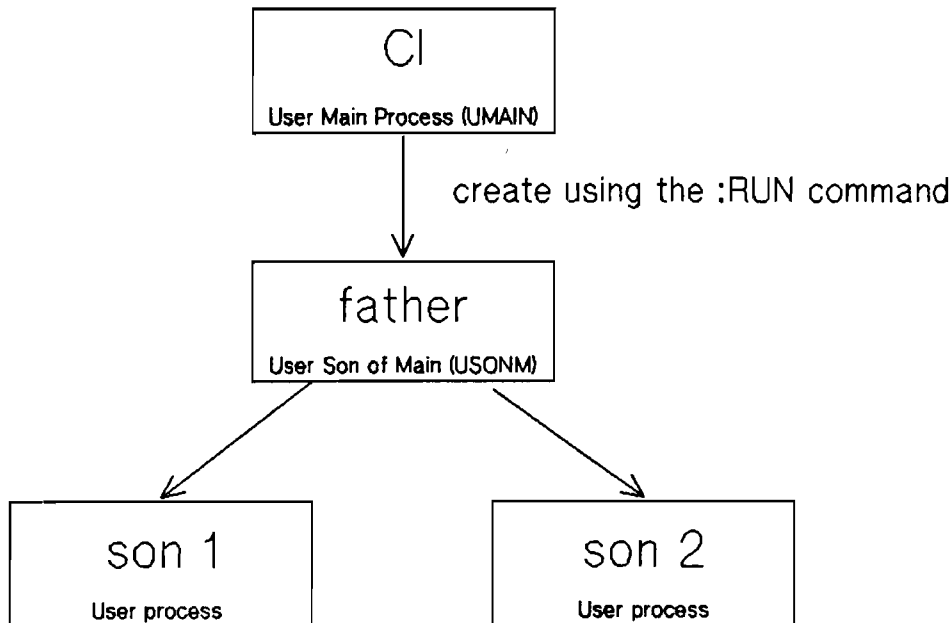
## Overview

Process Handling is a programming technique that allows a process to create descendant processes called sons or 'child processes'. In order for a process to use this technique it must possess the 'PH' (Process Handling) capability. This capability is granted to a program when it is PREPARED either by using the MPE PREP command or the SEGMENTER PREPARE command. The user who 'preps' the program must possess PH capability. The group and account in which the program resides must also possess PH. End users of the application *do not* require PH capability in order to run the programs possessing it.

## The PROCESS TREE Structure

Each job or session on MPE consists of at least a Command Interpreter process. This is referred to as the CI process or UMAIN (User Main process). Underneath this process, created via the :RUN Command, is the first user process or USONM (User Son of Main). Descendants of the USONM are referred to as USER processes. The structure that results is called a process tree which in its simplest form is made up of only a CI process and one program created using the MPE RUN command.

Following is an illustration of how a process tree structure is represented. This example contains two generations below the CI.



Process Tree (illustration 1)

The intrinsics used in a process handling environment are:

ACTIVATE	Used to execute a created process and optionally, to suspend the calling process.
CREATE	Used to create the process. Its parameters are similar to those specified on a :RUN command.
CREATEPROCESS	Also used to create a process. This intrinsic allows \$STDLIST and \$STDIN to be redirected for the new process and passes an INFO string to it.
FATHER	Returns the PIN (Process Identification Number) of the father (or parent) of the calling process.
GETORIGIN	Determines the source of the calling process' last activation.
GETPRIORITY	Changes the priority of the calling process or that of its sons.
GETPROCID	Returns the PIN of a son process.
GETPROCINFO	Returns a double word containing information about the calling process.
KILL	Used to terminate a son process.
MAIL	Checks the status of the mailbox for incoming or out going mail.
RECEIVEMAIL	Transfers data contained in the mailbox to the calling process' stack.
SENDMAIL	Transfers data from the calling process' stack to the mailbox.
SUSPEND	Causes the calling process to give up the CPU or suspend pending a future reactivation by some other process. This intrinsic will also unlock a local RIN prior to suspending the caller.

The following System Intrinsics do not require PH capability but are applicable in a process handling environment:

PROCINFO	Uses ITEMNUM/ITEM pairs to return information about the caller or other processes in the tree. It duplicates the functionality of other process handling intrinsics.
PROCTIME	Returns the number of milliseconds of CPU usage of the caller.
QUIT	Allows a process to abort itself and all of its descendants. Causes the PROGRAM ERROR #18 message to be displayed.
QUITPROG	Similar to QUIT, it aborts <i>all</i> processes below the 'CI' process and issues the PROGRAM ERROR #19 message.

## Application of Process Handling

Process Handling can be used in many diverse applications. One common use is in menu driven systems. This design allows a main 'driver' program to manage separate processes for the various functions of the applications. The benefit of this design is that it simplifies maintenance since an individual program can be modified without the need to recompile other parts of the system not affected.

### Using the Process Handling Intrinsic

Note that the CREATE and ACTIVATE intrinsic are used in this discussion. For information on the use of the CREATEPROCESS Intrinsic, refer to *Application Note #31, Calling the CREATEPROCESS Intrinsic*.

Following are the Working-Storage definitions that are referred to in this discussion:

```
01 PROCESS-HANDLING-DATA.
   05 PROG-NAME          PIC X(36) VALUE SPACES.
   05 ENTRY-NAME        PIC X(8)  VALUE SPACES.
   05 PIN                PIC S9(4) COMP VALUE 0.
   05 PARM               PIC S9(4) COMP VALUE 0.
   05 FLAGS              PIC S9(4) COMP VALUE 0.
   05 INFO               PIC X(80) VALUE SPACES.
   05 INFO-LEN          PIC S9(4) COMP VALUE 0.
   05 GETINFOERR        PIC S9(4) COMP VALUE 0.
   05 STACK-SIZE        PIC S9(4) COMP VALUE 0.
   05 DL-SIZE           PIC S9(4) COMP VALUE 0.
   05 MAXDATA           PIC S9(4) COMP VALUE 0.
   05 PRIORITY-CLASS    PIC X(2)  VALUE SPACES.
   05 RANK               PIC S9(4) COMP VALUE 0.
   05 SUSP              PIC S9(4) COMP VALUE 0.
   05 ACTIVATE-SOURCE   PIC S9(4) COMP VALUE 0.

01 PROCINFO-DATA.
   05 ERROR-1           PIC S9(4) COMP VALUE 0.
   05 ERROR-2           PIC S9(4) COMP VALUE 0.
   05 ITEM-NUMBERS.
      10 ITEM-NUM-1     PIC S9(4) COMP VALUE 1.
      10 ITEM-NUM-2     PIC S9(4) COMP VALUE 2.
      10 ITEM-NUM-3     PIC S9(4) COMP VALUE 3.
      10 ITEM-NUM-4     PIC S9(4) COMP VALUE 4.
      10 ITEM-NUM-5     PIC S9(4) COMP VALUE 5.
      10 ITEM-NUM-6     PIC S9(4) COMP VALUE 6.
      10 ITEM-NUM-7     PIC S9(4) COMP VALUE 7.
      10 ITEM-NUM-8     PIC S9(4) COMP VALUE 8.
      10 ITEM-NUM-9     PIC S9(4) COMP VALUE 9.
      10 ITEM-NUM-10    PIC S9(4) COMP VALUE 10.
   05 ITEM-NUM REDEFINES ITEM-NUMBERS
      PIC S9(4) COMP OCCURS 10 TIMES.

   05 ITEM-VALUES.
      10 MY-PIN         PIC S9(4) COMP VALUE 0.
      10 FATHER-PIN     PIC S9(4) COMP VALUE 0.
      10 SONS-CREATED   PIC S9(4) COMP VALUE 0.
      10 NUM-SONS       PIC S9(4) COMP VALUE 0.
      10 NUM-DESCENDANTS PIC S9(4) COMP VALUE 0.
```

10 NUM-GENERATIONS PIC S9(4) COMP VALUE 0.  
 10 SONS-PIN-ARRAY.  
   20 SONS-PIN PIC S9(4) COMP OCCURS 10 TIMES.  
 10 ALL-DECENDANTS.  
   20 PINS-OF-TREE PIC S9(4) COMP OCCURS 20 TIMES.  
 10 PINS-PRIORITY PIC X(2) VALUE SPACES.  
 10 ACTIVATE-STATE PIC S9(4) COMP VALUE 0.  
 10 PROGRAMS-NAME PIC X(28) VALUE SPACES.

01 MISCELLANEOUS-DATA.

05 LOCK-CONDITION PIC S9(4) COMP VALUE 0.  
 05 STATINFO PIC S9(9) COMP VALUE 0.  
 05 STATINFO-ARRAY REDEFINES STATINFO.  
   10 STAT-WORD1 PIC S9(4) COMP.  
   10 STAT-WORD2 PIC S9(4) COMP.  
 05 MODULO-VAR PIC S9(4) COMP VALUE 0.  
 05 CPU-USAGE PIC S9(9) COMP VALUE 0.  
 05 CPU-MASK PIC ZZ,ZZ9.999.  
 05 DEASSEMBLE-PROG-NAME.  
   10 DPN-PROGRAM-NAME PIC X(9) VALUE SPACES.  
   10 DPN-GROUP-NAME PIC X(9) VALUE SPACES.  
   10 DPN-ACCOUNT-NAME PIC X(9) VALUE SPACES.

01 MAIL-DATA.

05 MAIL-STATUS PIC S9(4) COMP VALUE 0.  
 05 MAIL-MESSAGE PIC X(80) VALUE SPACES.  
 05 MAIL-WAIT-FLAG PIC S9(4) COMP VALUE 0.  
 05 MAIL-LENGTH PIC S9(4) COMP VALUE 0.

## Creating A Process

In its simplest form process handling means that a process creates and activates another process. After this, the creating or father process suspends and is reactivated when the son process terminates. The following is an example of a user written process to create an EDITOR process:

### NOTE

In these examples double backslashes '\\' are used to indicate parameters that are not being passed to the intrinsic. Since the call specifies INTRINSIC these are not required to act as place holders and are used simply to add to the clarity of the code.

```
CREATE-EDITOR.  
  MOVE "EDITOR.PUB.SYS " TO PROG-NAME.  
  MOVE 1 TO FLAGS.  
  CALL INTRINSIC "CREATE" USING PROG-NAME, \\, PIN, \\,  
                                     FLAGS, \\, \\, \\, \\, \\.  
  IF C-C NOT = 0 THEN  
    PERFORM INTRINSIC-ERROR-ROUTINE  
    CALL INTRINSIC "QUIT" USING 1.  
  MOVE 2 TO SUSP.  
  CALL INTRINSIC "ACTIVATE" USING PIN, SUSP.  
  IF C-C NOT = 0 THEN  
    PERFORM INTRINSIC-ERROR-ROUTINE  
    CALL INTRINSIC "QUIT" USING 2.
```

In this example a value of '1' is moved to the item FLAGS. This sets bit 15 (the right most bit) on to tell MPE that when the newly created process terminates, the calling process is reactivated. Reference the *MPE System Intrinsic Manual* for further information. Your program can be reactivated by MPE when EDITOR ends (when the user enters EXIT). If this bit is not set, MPE assumes that you intend to handle reactivation which, in this case, would be impossible.

FLAGS can be used for several other purposes by moving the appropriate values to it. These options require bits to be set on or off. If you want more information on how to do this using COBOLII, please refer to *APPLICATION NOTE #21, COBOLII and MPE Intrinsic*.

After the process is CREATED it must be ACTIVATED. You can use the ACTIVATE intrinsic to both activate the new process and (optionally) suspend the caller. In the example a value of '2' is moved to SUSP. This indicates that the calling process will be suspended while the SON is running *and* the calling process expects to be ACTIVATED by the SON. Since EDITOR is being run this will actually never occur. That is, the EDITOR program will not explicitly call the ACTIVATE intrinsic to wake up the program that created it. This action will be carried out by MPE based on the value in FLAGS when the EDITOR process terminates.

The value of SUSP passed to ACTIVATE is used primarily in more complex process handling environments. In simple examples like the one presented, it serves only to ensure the calling process suspends during the active life of its son. Remember, if bit 15 of FLAGS is not equal to '1' then MPE does not activate the caller when the son process ends! (This would not, however, create a program hang situation since BREAK and ABORT can still be used.)



## Creating Several Processes in One Generation

Any process tree structure may contain at most 255 processes. This is not a configurable value. Since a generation is a layer of processes that share a common father or parent process, the maximum number of generations possible is also 255 (assuming that each generation consists of a single process.)

The following example demonstrates how several processes, in this case 5 copies of a program called SONPROG, are created. It also demonstrates how a local RIN is used to synchronize the operation of these 5 processes. Note that the first process created cannot begin execution until the last son has started running. The father process first acquires and locks one local RIN. When execution begins each son attempts the same lock. As each son is created it receives a PARM value equal to the order in which it was created. For example, the first son created receives a '1' as PARM. This is equivalent to the ;PARM parameter that can be used on a RUN statement.

```
CREATE-SONS.  
  CALL INTRINSIC "CREATE" USING PROG-NAME, \, PIN, PARM,  
                                     FLAGS, \, \, \, \, \.  
  IF C-C NOT = 0 THEN  
    PERFORM INTRINSIC-ERROR-ROUTINE  
    CALL INTRINSIC "QUIT" USING PARM.  
  CALL INTRINSIC "ACTIVATE" USING PIN, SUSP.  
  IF C-C NOT = 0 THEN  
    PERFORM INTRINSIC-ERROR-ROUTINE  
    CALL INTRINSIC "QUIT" USING PARM.  
SONS-CREATED.  
EXIT.
```

The main body of the program sets up the necessary values and performs CREATE-SONS, for example:

```
CALL INTRINSIC "GETLOCRI" USING 1.  
MOVE 1 TO LOCK-CONDITION.  
CALL INTRINSIC "LOCKLOCRI" USING 1, LOCK-CONDITION.  
IF C-C NOT = 0 THEN  
  PERFORM INTRINSIC-ERROR-ROUTINE  
  CALL INTRINSIC "QUIT" USING %1001.  
MOVE 0 TO FLAGS, SUSP.  
MOVE "SONPROG " TO PROG-NAME.  
PERFORM CREATE-SONS VARYING PARM FROM 1 BY 1  
                    UNTIL PARM > 5.  
MOVE 2 TO SUSP.  
CALL INTRINSIC "SUSPEND" USING SUSP, 1.  
IF C-C NOT = 0 THEN  
  PERFORM INTRINSIC-ERROR-ROUTINE  
  CALL INTRINSIC "QUIT" USING %1000.  
ALIVE-AGAIN.  
DISPLAY "All done! ".  
STOP RUN.
```

Note that both SUSP and FLAGS are zero for the five iterations of CREATE-SONS. This allows the calling program to remain active. Also notice that the call to SUSPEND includes the local RIN this process has locked. The SUSPEND intrinsic can unlock a local RIN prior to suspending the calling process. The following is an example of a SONPROG:

WORKING-STORAGE SECTION.

```
01 DISPLAY-DATA.
  05 MESSAGE-TO-DISPLAY.
    10 MSG-1          PIC X(40) VALUE
      "Shift to the left..."
    10 MSG-2          PIC X(40) VALUE
      "Shift to the right..."
    10 MSG-3          PIC X(40) VALUE
      "Pop up! ".
    10 MSG-4          PIC X(40) VALUE
      "Push down! ".
    10 MSG-5          PIC X(40) VALUE
      "Byte, byte, byte!!".
  05 DISPLAY-MSG REDEFINES MESSAGE-TO-DISPLAY
      PIC X(40) OCCURS 5 TIMES.

01 PARM              PIC S9(4) COMP VALUE 0.
01 DUMMY             PIC S9(4) COMP VALUE 0.
01 LOCK-CONDITION   PIC S9(4) COMP VALUE 1.
01 SON-MASK         PIC Z9.
```

PROCEDURE DIVISION.

```
GET-PARM.
  CALL INTRINSIC "GETINFO" USING \\, \\, PARM
    GIVING GETINFOERR.
  IF PARM = 0 THEN
    CALL INTRINSIC "QUITPROG" USING 900.
LOCK-LOCAL-RIN.
  CALL INTRINSIC "LOCKLOCRI" USING 1, LOCK-CONDITION.
  IF C-C NOT = 0 THEN
    CALL INTRINSIC "QUITPROG" USING 901.
PRINT-MESSAGE.
  MOVE PARM TO SON-MASK.
  DISPLAY DISPLAY-MSG (PARM), "(From SON #", SON-MASK, ")".
UNLOCK-LOCAL-RIN.
  CALL INTRINSIC "UNLOCKLOCRI" USING 1.
  IF C-C NOT = 0 THEN
    CALL INTRINSIC "QUITPROG" USING 902.
FINISH-UP.
  IF PARM = 5 THEN
    CALL INTRINSIC "ACTIVATE" USING 0, 0
    IF C-C NOT = 0 THEN
      CALL INTRINSIC "QUITPROG" USING 903.
STOP RUN.
```



In this example the paragraph named FINISH-UP determines whether or not the father process should be activated based on the value of PARM. If this is the last process to run, then it wakes up the father before it terminates. The call specifies zero for BOTH parameters so the calling process is not SUSPENDED as a result of the call.

Note also that local RINs are acquired for use by all processes in the process tree therefore SONPROG did not need to call the GETLOCRI intrinsic; this had already been done by the father process.

## Using PARM and INFO

In the previous examples PARM was used to communicate a creation order to each of the sons. This value also determined which array element was displayed. The values of PARM and INFO can be used to pass 'one time only' information from a process to a son at creation time. The son need not check the values, and the information contained in PARM or INFO can be required by the son as often as necessary. The intrinsic GETINFO, available with MPE version 6.02.00 (U-MIT) or later, can be used to retrieve these values. For example:

```
MOVE 80 TO INFO-LEN.  
CALL INTRINSIC "GETINFO" USING INFO, INFO-LEN, PARM  
    GIVING GETINFOERR.  
IF GETINFOERR NOT = 0 THEN  
.  
.  
.
```

## Communication Between Processes

In addition to the 'one time only' communication available through PARM and INFO there exists the ability for processes both inside and outside the 'process tree' to communicate. This is known as Interprocess Communication (IPC).

Interprocess Communication can be accomplished in one of two ways: through the use of MESSAGE files or by using the MAIL intrinsics. This is not to be confused with electronic mail! The most efficient and easy method is through the use of MESSAGE files.

Refer to the *MPE File System Reference Manual, Section 8* for a more complete discussion on this topic.

To demonstrate the MESSAGE file communication between processes you can try this brief example using FCOPY and two terminals.

You'll need to log on to each of the two terminals. You need not be the same user nor in the same account. The limitation is that each of the two users must have appropriate access to the message file you will need to create. That is, the reader process needs at least READ access and the writer process must have WRITE access.

To build the file issue the following BUILD command:

```
:BUILD MSGFILE;REC=-80,,,ASCII;MSG
```

At one terminal type:

```
:FCOPY FROM= ;TO=MSGFILE
```

This will be the writer process. On the other terminal type:

```
:FCOPY FROM=MSGFILE;TO=
```

This is the reader process. Whatever is typed onto the writer's terminal will be read by the reader FCOPY process, and displayed on its terminal. When you terminate the writer process either by a CONTROL Y or :EOD the reader process will also end since FCOPY will receive the EOD OF FILE.

## The MAIL Intrinsic

These intrinsic are an older form of IPC. Communication is limited between processes to those in a process tree. The intrinsic are, in general, more difficult to use than MESSAGE file IPC and require that programs be PREPARED with 'PH' capability. Message file IPC has no special capability requirements.

Another limitation of MAIL IPC is that communication is limited to father and son processes only. Processes in the same generation, sometimes referred to as 'brothers', may only communicate through the father. No such limitation exists for Message File IPC.

## The MAILBOX

This is a term used in MAIL IPC that defines the location used to store incoming or outgoing mail between two processes. There can be only one mailbox per pair of communicating processes. That is, a father may have one mailbox for each of its sons, but they do not all share the same mailbox. Each son will have one mailbox for each of its sons.

A mailbox is an extra data segment. If, however, the mail is one word (2 bytes) in length an extra data segment is not created. Rather, a portion of the calling process' stack is used. Since mailboxes are extra data segments, the number and maximum size that may be created is configurable.

## Checking the MailBox

The MAIL intrinsic is used to check the mailbox. This intrinsic returns the state of the mailbox and, if incoming mail exists, will return the length in words of that mail. For example:

```
CALL INTRINSIC "MAIL" USING 0, MAIL-LENGTH
                                GIVING MAIL-STATUS.
IF MAIL-STATUS > 2 THEN
```

```
.
.
.
```

Rather than the condition code, the STATUS variable is checked since it indicates the success of the call. Refer to the *MPE System Intrinsic Reference Manual* for a definition of the values returned.

## Sending Mail

This intrinsic transmits data from the calling process into the mailbox. Depending on the value passed as WAITFLAG the intrinsic will cause the process to wait for the mail to be read or continue without waiting.

```
MOVE 1 TO MAIL-WAIT-FLAG.
CALL INTRINSIC "SENDMAIL" USING PIN,
                                MAIL-LENGTH,
                                MAIL-MESSAGE,
                                MAIL-WAIT-FLAG
                                GIVING MAIL-STATUS.
IF C-C NOT = 0 THEN
```

```
.
.
.
```

Note, this intrinsic returns a status and also sets the condition code. Please refer to the *MPE System Intrinsic Reference Manual* for an explanation of the values returned in STATUS.

In this example a MAIL-WAIT-FLAG of '1' is used to cause the calling process to wait until any prior mail is collected. A zero value would have caused any uncollected mail to be overwritten and the calling process would not have waited for the receiving process to collect the message.

## Receiving Mail

To read the contents of the mail box the RECEIVEMAIL intrinsic is used. It acts in a similar way as SENDMAIL in that it can be used to cause the calling process to wait until mail is received, for example:

```
MOVE 1 TO MAIL-WAIT-FLAG.  
CALL INTRINSIC "RECEIVEMAIL" USING PIN,  
MAIL-MESSAGE,  
MAIL-WAIT-FLAG  
GIVING MAIL-STATUS.  
  
IF C-C NOT = 0 THEN  
  IF MAIL-STATUS = 3 THEN  
  .  
  .  
  .
```

This intrinsic does not return the length of the mail received. Before attempting to receive mail it is recommended to call MAIL to determine if mail exists and to determine the length of the message. If it is determined that an incoming message is longer than can be accommodated, there would be no way to safely transfer the mail to the callers stack without the possibility of data corruption occurring.

## Using the Mail Intrinsic

If you have purchased TDP (Text and Document Processor) you can easily demonstrate Mail IPC. To do this alter the earlier example that created an EDITOR.PUB.SYS process so that PROG-NAME contains TDP.PUB.SYS. Then, in between the code that calls CREATE and the code that calls ACTIVATE add the following:

```
SEND-MAIL.  
MOVE  
"Q'Please enter // when thru adding, thanks';Make Demo;k;e"  
  TO MAIL-MESSAGE.  
MOVE 0 TO MAIL-WAIT-FLAG.  
CALL INTRINSIC "SENDMAIL" USING PIN,  
40,  
MAIL-MESSAGE,  
MAIL-WAIT-FLAG  
GIVING MAIL-STATUS.  
  
IF MAIL-STATUS NOT = 0 THEN  
  CALL INTRINSIC "QUIT" USING MAIL-STATUS.
```

## NOTE

Mail messages sent to TDP must be at least 72 bytes (36 words).

TDP will check the mailbox after it is activated and treat the contents as a command to be executed before prompting for the first user command from the standard input device (\$STDIN). In this example four commands are sent separated by semicolons. The first displays a message on the current \$STDLIST device. The second MAKES a file which then automatically causes TDP to prompt for input to the file. The third causes the file to be kept after the user has terminated additions with the '/' command. The last command exits TDP and returns to the process that created it.

## Additional Process Handling Concepts

### The FATHER Intrinsic

If your system has TDP then it also has a program called SCRIBE in the PUB group of the SYS account. This program is the formatter used by TDP when you issue a DRAFT or FINAL command. SCRIBE is specifically written to determine how it was created and act accordingly. Your programs may be written to behave this way as well. Try running SCRIBE and see what happens.

In the *Computer Scientist's Cheer* program, for example, you might not want SONPROG to be :RUN by other users. To prevent this, add the following code as the first executable statements in the program:

```
CALL INTRINSIC "FATHER" GIVING DUMMY.  
IF C-C > 0 THEN  
    DISPLAY "Sorry, you may not :RUN this program."  
STOP RUN.
```

Notice there is no USING parameter since the intrinsic returns the PIN in DUMMY. This name was selected since its contents are not needed. The condition code returned indicates the type of process that created the caller. In this case CCG is checked since this indicates that the creating process is a User Main or Command Interpreter process.

### The GETORIGIN Intrinsic

This intrinsic is used primarily in more complex process handling environments. Specifically when a process calls SUSPEND or ACTIVATE and passes the value of '3' for SUSP. This indicates that the caller expects to be activated either by a son or by its father. If this is the case it may be advantageous to know who woke up the process. To do this you would call the GETORIGIN intrinsic. For example:

```
CALL INTRINSIC "GETORIGIN" GIVING ACTIVATE-SOURCE.  
IF ACTIVATE-SOURCE = 0 THEN  
    .  
    .  
    .
```

This intrinsic does not return a condition code so the value of ACTIVATE-SOURCE must be checked. A value of zero would indicate the caller was not activated by either a son or its father.

## The GETPRIORITY Intrinsic

This intrinsic allows a process to alter its own priority or that of one of its sons. Priority may be no larger than the MAXPRI value of the USER running the program. For example, to reschedule a son process into the ES Subqueue or 'EQ' a call to GETPRIORITY might look like this;

```
MOVE "ES" TO PRIORITY-CLASS.  
MOVE 0 TO RANK.  
CALL INTRINSIC "GETPRIORITY" USING PIN,  
                                PRIORITY-CLASS,  
                                RANK.  
  
IF C-C NOT = 0 THEN  
.  
.  
.
```

## The GETPROCID Intrinsic

This intrinsic may be used to 'take inventory' of son processes. It is beneficial in process handling environments where it is natural for some sons to end while others may continue processing. The father can periodically find out who's left. To illustrate this it will be necessary to alter the *Computer Scientist's Cheer* program example so that instead of a STOP RUN each son calls the SUSPEND intrinsic. The fifth son must still ACTIVATE the father! When the father begins execution again it can find out which of its sons still remain 'alive' using the following example;

```
MOVE 0 TO NUM-SONS.  
MOVE 5 TO SONS-CREATED.  
TAKE-INVENTORY.  
ADD 1 TO NUM-SONS.  
CALL INTRINSIC "GETPROCID" USING NUM-SONS  
                                GIVING SONS-PIN (NUM-SONS).  
IF SONS-PIN (NUM-SONS) NOT = 0 THEN  
IF NUM-SONS < SONS-CREATED  
GO TO TAKE-INVENTORY.
```

As long as the value in NUM-SONS does not exceed the number of sons still in existence a non-zero PIN will be returned. It should also be noted that had, for example, son number 2 expired son 3 would become son 2 and so forth, son 5 would then not exist.

## The KILL Intrinsic

This is a rather aptly named intrinsic that does precisely what its name implies, it kills a son process. To illustrate this, alter the TAKE INVENTORY example above to include a call to KILL as follows:

```
IF SONS-PIN (NUM-SONS) NOT = 0 THEN  
CALL INTRINSIC "KILL" USING SONS-PIN (NUM-SONS)  
IF C-C NOT = 0 THEN  
.  
.  
.
```

## The GETPROCINFO Intrinsic

This intrinsic returns information about the PIN passed to it. The value is returned in a double word. To make effective use of the information returned, redefine the STATINFO variable so that it could be referenced as two separate integer variables as described in the WORKING-STORAGE example.

```
CALL INTRINSIC "GETPROCINFO" USING PIN
                                GIVING STATINFO.
IF C-C NOT = 0 THEN
```

Please refer to *Application Note #21, CobolIII and MPE Intrinsic*s. Using the table at the top of page 10 of that publication as a guide here is an example of how to use the various bit settings returned in STATINFO. In this example you will extract bits 4, 5 and 6 which will define the queue characteristics of the PIN.

First you need to shift bits 4, 5 and 6 so they are in positions 13, 14 & 15. This is done by dividing STAT-WORD1 by 512 which corresponds to bit 6 in the table. Store the result in MODULO-VAR. For example:

```
COMPUTE MODULO-VAR = STAT-WORD2 / 512.
```

To extract the value of bits 13 through 15 you will need to calculate the MODULO 8 of MODULO-VAR or the remainder of MODULO-VAR divided by 8 which would be the value of these three bits. Since there is no MODULO function in COBOL it can be accomplished using the following calculation:

```
COMPUTE MODULO-VAR = MODULO-VAR - (MODULO-VAR / 8) * 8.
```

## Additional Intrinsic

The following intrinsic do not require PH capability but are relevant in a process handling environment.

### QUIT and QUITPROG

Both intrinsic accept an integer variable as a parameter, but the QUIT intrinsic differs from the QUITPROG intrinsic in that it will cause the caller and all of its descendants to be aborted. QUITPROG can be called by any process in the tree and will cause all processes in the tree to abort. For example:

```
CALL INTRINSIC "QUIT" USING 901.
```

might look like:

```
ABORT :DEMO.PROG.DEVEL.%0.%1
PROGRAM ERROR #18 :PROCESS QUIT .PARAM = 901

PROGRAM TERMINATED IN AN ERROR STATE. (CIERR 976)
```

Had QUITPROG been called the message would have changed slightly to read:

```
PROGRAM ERROR #19 :PROCESS QUIT .PARAM = 901
```



The octal values %0.%1 are the code segment and offset into this segment where the call to QUIT took place. Using a PMAP and output from the VERBS compiler directive it would be possible to locate the QUIT statement using this data. It is common, however, to use the value of PARAM as an indicator of where and possibly why the call was made.

### The PROCTIME Intrinsic

As its name implies, this intrinsic returns the number of milliseconds (thousandths of seconds) of CPU time used by the caller. For example:

```
CALL INTRINSIC "PROCTIME" GIVING CPU-USAGE.  
MOVE CPU-USAGE TO CPU-MASK.  
DISPLAY CPU-MASK.
```

This intrinsic does not return condition codes.

### The PROCINFO Intrinsic

This intrinsic is similar in structure to the FFILEINFO and JOBINFO intrinsics. It is more difficult to use but returns some valuable information.

The intrinsic accepts up to six ITEM-NUM/ITEM pairs where the value of ITEM-NUM determines the data type of ITEM. At a minimum, the intrinsic will have five parameters and at a maximum, fifteen.

Refer to the *MPE System Intrinsic Reference Manual* for complete information on the use of this intrinsic.

With normal (non-privileged mode) capabilities this intrinsic replaces many of the intrinsics already discussed. For example, to find the PINs of all surviving son processes using a single call:

```
MOVE 10 TO SONS-PIN (1).  
MOVE 0 TO PIN, ERROR-1, ERROR-2.  
CALL INTRINSIC "PROCINFO" USING ERROR-1,  
                                ERROR-2,  
                                PIN,  
                                ITEM-NUM (6),  
                                SONS-PIN-ARRAY.  
  
IF ERROR-1 NOT = 0 THEN  
  .  
  .  
  .
```

In this example a zero is moved to PIN since the information required is for sons of the calling process. Also, a '10' is moved to the first element of SONS-PIN-ARRAY to indicate the length, including this element, of the array. Please see 'Note 1' under the PROCINFO intrinsic in the *MPE System Intrinsics Reference Manual*.

The difference between item 6 and item 7 is that item 7 will return the PINs of *all* processes both direct and indirect. That is, all descendants of the PIN specified rather than just the processes created by the PIN specified.

A feature of PROCINFO not found in any other process handling related intrinsics is the ability to return the actual name of a process. ITEM-NUM (10) returns the fully qualified name of the process identified by the PIN passed to it. This means that it is possible for a program to know its name or the name of any of of the programs currently running in the process tree.

For example, a program could determine its own name by calling PROCINFO as follows:

```
MOVE 0 TO PIN, ERROR-1, ERROR-2.  
CALL INTRINSIC "PROCINFO" USING ERROR-1,  
                                ERROR-2,  
                                PIN,  
                                ITEM-NUM (10),  
                                PROGRAMS-NAME.  
  
IF ERROR-1 NOT = 0 THEN  
  .  
  .  
  .
```

This can be valuable when process handling is implemented in a menu driven application. If the application has been designed so that all programs in the process tree reside in the same group and account, then it is not necessary for programs needing to create sons to know any more than their names. The group name (and optionally the account name) can be extracted from the string returned by PROCINFO.

For example, using the COBOLII STRING and UNSTRING verbs the callers name can be disassembled and then reassembled to be the name of program to be passed to the CREATE intrinsic. For example:

```
UNSTRING PROGRAMS-NAME DELIMITED BY "." INTO  
    DPN-PROGRAM-NAME, DPN-GROUP-NAME, DPN-ACCOUNT-NAME.  
MOVE "SONPROG" TO DPN-PROGRAM-NAME.  
MOVE SPACES TO PROG-NAME.  
STRING DPN-PROGRAM-NAME DELIMITED BY SPACE  
    "." DELIMITED BY SIZE  
    DPN-GROUP-NAME DELIMITED BY SPACE  
    "." DELIMITED BY SIZE  
    DPN-ACCOUNT-NAME DELIMITED BY SPACE  
    INTO PROG-NAME.
```

## Configuration Considerations

If you plan to make extensive use of process handling or are contemplating the purchase of a software product that uses Process Handling you should consider the following system configuration values.

Please refer to the *MPE System Operation and Resource Management Reference Manual, Section 7*, for additional information.

## The Process Control Block Table

This table contains a 21 word entry (16 words on MPE IV systems) for each process running on the system. This includes *system* processes as well as user processes. The number associated with the process' entry in this table is referred to as its *PIN*. If this table is underconfigured system failures will result.

## The CST Block Table

This table limits the number of concurrently running programs on the system. Since executable code is shared on MPE this refers to different *programs* rather than different processes.

## The Swap Table

This table is used to keep track of a process' *locality* or the code and data segments it needs in memory in order to run. When possible a process' entire locality is tracked through this table however it must be able to hold the *minimum locality* of each process running. If this table is underconfigured a system failure will result.

## Additional Considerations

If your programs intend to make extensive use of MAIL IPC you will need to consider the following configuration values:

- The Maximum Extra Data Segment Size
- The number of Extra Data Segments per Process
- The amount of Virtual Memory configured on your system

Remember that VIRTUAL MEMORY (VM) may be changed on Logical Device One **ONLY** during a RELOAD. VM allocations to other disc devices may be made using a COOL or COLD Start.

Virtual Memory needs to be considered if MAIL IPC is used because it creates Extra Data Segments if needed. Any time new data segments are created space is also allocated in Virtual Memory. This space is used to hold them when they no longer need to be resident in main memory.

## BACK ISSUE INFORMATION

Following is a list of the Application Notes published to date. If you would like to order single copies of back issues please use the *Reader Comment Sheet* attached and indicate the number(s) of the note(s) you need.

<u>Note #</u>	<u>Published</u>	<u>Topic</u>
1	2/21/85	<i>Printer Configuration Guide (superseded by note #4)</i>
2	10/15/85	<i>Terminal types for HP 3000 HPIB Computers (superseded by note #13)</i>
3	4/01/86	<i>Plotter Configuration Guide</i>
4	4/15/86	<i>Printer Configuration Guide - Revised</i>
5	5/01/86	<i>MPE System Logfile Record Formats</i>
6	5/15/86	<i>Stack Operation</i>
7	6/01/86	<i>COBOL II/3000 Programs: Tracing Illegal Data</i>
8	6/15/86	<i>KSAM Topics: COBOL's Index I/O: File Data Integrity</i>
9	7/01/86	<i>Port Failures, Terminal Hangs, TERMDSM</i>
10	7/15/86	<i>Serial Printers - Configuration, Cabling, Muxes</i>
11	8/01/86	<i>System Configuration or System Table Related Errors</i>
12	8/15/86	<i>Pascal/3000 - Using Dynamic Variables</i>
13	9/01/86	<i>Terminal Types for HP 3000 HPIB Computers - Revised</i>
14	9/15/86	<i>Laser Printers - A Software and Hardware Overview</i>
15	10/01/86	<i>FORTRAN Language Considerations - A Guide to Common Problems</i>
16	10/15/86	<i>IMAGE: Updating to TurboIMAGE &amp; Improving Data Base Loads</i>
17	11/01/86	<i>Optimizing VPLUS Utilization</i>
18	11/15/86	<i>The Case of the Suspect Track for 792X Disc Drives</i>
19	12/01/86	<i>Stack Overflows: Causes &amp; Cures for COBOL II Programs</i>
20	1/01/87	<i>Output Spooling</i>
21	1/15/87	<i>COBOLII and MPE Intrinsic</i>
22	2/15/87	<i>Asynchronous Modems</i>
23	3/01/87	<i>VFC Files</i>
24	3/15/87	<i>Private Volumes</i>
25	4/01/87	<i>TurboIMAGE: Transaction Logging</i>
26	4/15/87	<i>HP 2680A, 2688A Error Trailers</i>
27	5/01/87	<i>HPTrend: An Installation and Problem Solving Guide</i>
28	5/15/87	<i>The Startup State Configurator</i>
29	6/01/87	<i>A Programmer's Guide to VPLUS/3000</i>
30	6/15/87	<i>Disc Cache</i>
31	7/01/87	<i>Calling the CREATEPROCESS Intrinsic</i>
32	8/01/87	<i>Configuring Terminal Buffers</i>
33	9/01/87	<i>RIN Management (Using COBOLII Examples)</i>



# READER COMMENT SHEET

North American Response Centers  
HP 3000 Application Note #34: Process Handling  
RC Questions & Answers (October 1, 1987)

We welcome your evaluation of this Application Note and attached RC Questions & Answers Sheet. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below.

	<u>AppNote</u>	<u>RC Q&amp;A</u>
Is this publication applicable to your site?	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No
Are the concepts and wording easy to understand?	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No
Would you like to see additional Notes on this subject?	<input type="checkbox"/> Yes <input type="checkbox"/> No	<input type="checkbox"/> Yes <input type="checkbox"/> No

Back Orders/Comments/Suggestions for future Application Notes:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM:

Date \_\_\_\_\_

Name \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_




**HEWLETT  
PACKARD RESPONSE CENTER QUESTIONS & ANSWERS**

HP 3000 Questions Commonly Received by the North American Response Centers

The following Question was answered in the July 15 1987 issue. We received a request to reprint it with a more detailed explanation.

**Q.** In the *MRJE Reference Manual* (Part No. 39248-90001) there are examples of **FORM NAMES** that are greater than four characters in length but my IBM system will not except a form name greater than four characters. Can I still make use of a form name containing more than four characters?

**A.** Yes. If your host limits you to four character **FORM NAMES** (as in the *DO IMB JCL*) you can make use of the **MRJETABL** to allow **FORM NAMES** of greater than four characters. In chapter four of the *MRJE* manual, there is an illustration of the use of such a table entry and the **SUBMIT** command to use a **FORMID** which is greater than four characters:

```
MRJETABL entry --->          P
command issued --->          SUBMIT jobname;FORMS='PAYROLL1'
```

The user directs output to the formid **PAYROLL1** for solicited output this way. **PR=** and **PU=** can be used in a similar fashion:

```
command issued --->          SUBMIT jobname:PR='MYFILE'
                                SUBMIT jobname:PU='YOURFILE'
```

In the above example, there would be an entry for **MYFILE** and **YOURFILE** in the **MRJETABL**, defining the destination of the output (a printer or a disc file).

Note that, **SUBMIT** parameters override **JCL** special forms routing thus assuring proper routing of data even if the host only permits form names of four characters.

Some hosts, such as **JES3**, allow for longer form names via a **FORMAT** or similar **JCL** statement.

**Q.** How do I call the **COMMAND** intrinsic from **Transact**?

**A.** Although it is fairly simple to call the **COMMAND** intrinsic from **Transact**, there are a few things that can make it tricky the first time you try it.



When calling a command, the command name must be terminated by a carriage return. In Transact this is accomplished by placing the integer value for an ASCII space and carriage return (8205) at the end of the command string. To do this, create an I4 field, and MOVE it to the end of the command string.

When the PROC verb is issued, the command string (byte array) must be passed a byte address. In Transact, this is specified by a percent sign (%) preceding the variable in the PROC verb.

The other two parameters are ERROR (which returns the error code set by the command) and PARM (which contains the file system error or the number of an erroneous parameter). Both of these parameters should be defined as I4.

For example, the following Transact code issues a file equation and checks that it is successful:

```
SYSTEM PROCS,SIGNON="CALLING THE COMMAND INTRINSIC";

DEFINE (INTRINSIC) COMMAND:

DEFINE (ITEM)      COMSTRING      X(26):
                   CR              I(4) :
                   ERROR          I(4) :
                   PARM           I(4) :

LIST              COMSTRING:
                  CR:
                  ERROR:
                  PARM;

MOVE              (COMSTRING) = "FILE FRED;DEV=LP";
LET               (CR)       = 8205;
MOVE              (COMSTRING) = (COMSTRING)+(CR);

PROC COMMAND (%(COMSTRING), (ERROR), (PARM))

IF STATUS < > 0 THEN

    DO
        DISPLAY "ERRORS";
        DISPLAY ERROR;
        DISPLAY PARM;
    DO END:

EXIT;
END PROCS;
```