Using MPE Message Files - An Applications Approach.

Patrick Fioravanti
Infocentre Corporation
7420 Airport Road
Suite 201
Mississauga, Ontario
Canada L4T 4E5

Message files are a feature of the MPE file system that permit two or more processes (programs) running concurrently to communicate with each other. Typically this Inter-Process communication is used to coordinate the activites of the two processes. In this light, the processing of an application task (Order Entry for example) can be distributed across a number of different programs, yielding useful benefits in an efficient manner.

This paper takes an applications approach to describing the purpose and functionality of Message files, rather than a hard core technical approach. It illustrates in layman terms how this under utilized feature of the file system can be incorporated into the design of many application systems. The discussion will be augmented with programming examples taken from an Order Entry Application developed in Speedware, to show how Message files are accessed and manipulated. Along the way some of the application design parameters that can be tweaked in order to maximize the benefits from a Message file implementation, will be discussed.

Audience level of my abstract: 1-3 years.

This paper would best fit in Track 3.

What are Message Files and why would one use them?

Message files are a feature of the MPE File System, available to HP3000 applications. They are a specific type of sequential file intended for applications that use *Inter Process Communication.* IPC is a mechanism whereby two programs running concurrently can pass information back and forth to each other. Message files were designed for ease of use and efficient communication between processes. They have some intriguing characteristics:

* FIFO Queues. Typically two or more programs will be accessing the message file concurrently. The program that Reads the file, will read the records in the order in which they were written to the file.

* Destructive Reads. As records are read from the message file, they are deleted. Note that the programmer can influence this with a call to FCONTROL, Control code 47.

* The MPE File System causes programs to wait until their Message File I/O is complete. *Readers* wait on a read request until there is a record to be read. *Writers* wait on a Write until there is room in the file to accomodate the record. Optionally the programmer can limit the wait to a pre-specified number of seconds.

* When reading, an EOF condition is returned by the File System when there are no records in the file **and** no *Write* processes have the file open. This characteristic may also be influenced by the programmer via a call to FCONTROL, Control code 45.

* Unidirectional flow. A program can Read from a message file, or Write to it but not both. Access is specified at file open time.

* Many concurrent Readers and Writers are permitted, although One Reader and one or many Writers is typical.

* Performant. Message files are partially memory resident, and partially disc based. Usually most I/O's are done in memory.

Given these characteristics of Message Files, some interesting applications come to mind. The nature of message file behaviour makes it feasible to distribute application processing across several processes, and coordinate the activities of those processes. Often it is the case that an On-Line task is *stream lined* (to make it run faster) leaving some *clean up* work for a subsequent batch task. Consider as an example an Order Entry application. The On-Line task is the entry of the Customer Orders. To prevent this process from being bogged down, usually the printing of the Order form or invoice is deferred to a nightly batch run. The entry of the order at the terminal along with the printing of the invoice constitutes the whole order process. It has been split into On-Line and nightly batch tasks in order to stream-line the on-line task. What if .... we still want the

On Line task streamlined, but we also want the Invoices printed as we continue entering orders? This can be accomplished by running the Batch Print Routine in the background as the terminals are entering the orders. As each order is entered, Inter Process Communication is required for the On-Line process to tell the Batch Process to *Print an invoice for the Order I just entered, and send the output to the printer located beside my desk.* An ideal application for Message Files.

Why use a Message File in the above Scenario? First off let's clarify how the Message file might be used. In the Data Entry routine, as each Order is Entered, a new record can be written to the Message file, identifying an order by its Order Number, and supplying additional processing instructions. There are multiple terminal sessions doing Order Entry, and each of these sessions have the Message file open for Write Access. The Print routine opens the Message file for Read Access. It waits in the background for a record to be written to the Message file. When a record comes in, the Print routine Extracts the Order information and formats and prints the invoice, then waits for the next Message file record. This Print Routine can be run On-Line or in Batch, in the BS, CS, DS, or ES Queues, the choice is yours.

A message file is well suited to this application because:

* The Print Routine will suspend on the Read, waiting for an order to print. It doesn't waste CPU seconds looping around looking for something to do.

* Since Message files are a FIFO Queue, Orders are printed in the order in which they were entered. Furthermore, the list of orders to be printed maintains itself, once a record is read from the message file it is also deleted from the file.

* The Print Routine will not receive an EOF condition from the File System until all records have been read, and no other processes have the file open for Write access. So when the terminal operators stop entering orders, and all the queued invoices have been printed, the Print Routine can be programmed to automatically terminate.

* Performance. The reading and writing of records to/from the message file will involve few if any Disc I/O's. Furthermore this approach requires the launching of only one process to print all of the day's invoices as they are entered. Other approaches might involve launching a separate process or job to print each invoice as it was entered.

How would we implement this solution in a Speedware Application? The first step is to define the message file. For this example, the message file is used to pass an Order Number from the Data Entry process to the Print process. Additionally we may want a general purpose Character field in the Message file layout for future use. Our Order Number is a J2 field, and we will tack on an eight character text field. Accordingly we need a Message file with a Logical Record Length of 12 characters.

```
:BUILD MSGFILE;REC=-12,,F,ASCII;DISC=100;MSG
```

The MPE BUILD command illustrated above will create the Message file. This is a typical BUILD command with the exception of the ;MSG parameter, which of course tells MPE to create the file as a Message file. The file limit you specify is significant. Because of the nature of Message File operation, the file limit is set according to the number of records that the file must hold at any one time. Remember that records are deleted from the file as they are read. When setting the file capacity, ask yourself this question: Assuming that the On-Line processes can dump records into the file faster than the Print routine can read them, what is likely to be the biggest number of orders queued for printing at any time? This number (plus a bit more) should be your file limit. Temper your judgement with the understanding that Write Processes will be forced to wait for the Write to complete, if the Message file is full. In the above example, no Blocking Factor was set, and MPE will decide on one itself. The Blocking Factor for Message Files doesn't have a big impact on I/O performance since most or all of the Reads and Writes are done in memory. The Blocking Factor will influence the amount of Disc Space consumed by the Message file at BUILD time (along with other factors like Record Length, File Limit, and extents).

Once the Message file is built, it may be accessed from Speedware Applications. The Speedware Programmer will code a FILE Section to describe the Record Layout, and from there can read or write to the file using the FOR and CREATE commands.

For our example, we could code a File section like this:

```
FILE-MSG: (MSGFILE.DATA.INFOSYS)

 ORDER#   [1-4]    TYPE(J0);
 TEXT     [5-12];

EXIT;
```

Before running our application and accessing the Message file we need to supply some of the file Access Options to be used when the file is opened. We specify these options with an MPE FILE command. The FILE command for the *Writers* (On-Line processes doing the Data Entry) would look like this:

```
:FILE MSGFILE.DATA.INFOSYS;MSG;ACC=OUT;SHR;GMULTI
```

;ACC=OUT specifies that we will be writing (OUTPUT) records to the Message File. Remember that a given process can only have one type of Access to a Message File; either Read or Write.

;SHR means that we won't have exclusive access to the file, rather a number of concurrent processes can access the file.

;GMULTI means that the shared multiple access is GLOBAL. In other words, the multiple processes can belong to different jobs/sessions. If we specify ;MULTI instead, then the multiple processes accessing the file must all have been spawned

from the same terminal session or batch job.

For the Print Routine, a different FILE command is necessary, this one specifying *Read* Access:

```
:FILE MSGFILE.DATA.INFOSYS;MSG;ACC=IN;SHR;GMULTI
```

Now let's address the programming details.

1) The Data Entry Screen.

Let's suppose that our application already contains a data entry screen used to enter new orders. Currently this Screen is comprised of two formats: the first one writes a record to the Order-Header Dataset, and the second format is concerned with the line items, maintained in the Order-Detail Dataset. Order entry is completed when the terminal operator completes both formats.

This Screen Program can be modified to write a record into the Message File at the conclusion of Data Entry for each order. A COMPUTE paragraph called from this screen in both Add and Modify modes can accomplish the task.

The COMPUTE code might look like this:

```
COMPUTE-ADDMSG: AM;

CREATE FILE:MSGFILE WITH
       ORDER-NO    = ORDER#,
       'PRINT IT' = TEXT;

EXIT;
```

2) The Print Routine.

Our application already has a Report Program that is used to print the invoices. This Report uses the Order Number as a key to access the Order Header and Order Detail files, and from there is able to directly access Customer and Product information. All of this data is extracted, sorted, then printed on the Invoice Form.

This Report program can be modified to be driven by the Message file. The coding of the Extraction Phase might look like this:

```
FOR FILE:MSGFILE              (* Read Message File *)
    BEGIN;

        FOR ORDER-HEADER.ORDER-NO<ORDER#>
            BEGIN;
                .  .  .
                .  .  .
                END;
        END;

SORT ON .  .  .  ;
```

As you can see, the records are read from the Message File as they come in. From there, the Extraction proceeds, beginning with the access of the Order-Header record identified by the key value contained in the Message File ORDER# field.

3) Coordination of the two processes.

We need to exercise some control over the activities of the Data Entry and Print processes so that they work in harmony. It makes sense for the Print Routine to execute in batch: that way it won't be dependent on a terminal session, and by default it will operate in the DS queue at a lower priority than the On-line sessions. Having decided that, we need automatic mechanisms in place to start the job when users begin entering orders, and to stop the job when the users are finished. As described earlier, the standard functioning of Message Files dictates that *Readers* waiting for a record will be supplied an EOF condition when the last of the *Writers* closes the Message File. It would seem that the MPE File System automatically provides us with the job shutdown mechanism, since an EOF on the Message File will terminate the FOR Loop driving the Report extraction phase. As we all know, things are never this simple. In addition to the automatic shutdown provided by an EOF condition, we should also have the means available within the application for the user to request the termination of the Print Routine. Additionally the Print Routine needs to have its Extraction Phase adjusted for it to be of much use. As it stands now, the Extraction Phase will not terminate until it reaches EOF on the Message File (ie: The end of the day when users have stopped entering orders), at which time the Sort and Print Phases will be activated. The end result is, no orders will be physically printed until the end of the day, and then they will all be printed. That's not what we want. Orders should be printed as they are entered, one at a time.

Let's address these *coordination* issues one at a time:

i) Automatically launching the Print Job.

The batch job that runs the Report must be executing when terminal operators are entering orders, otherwise records will just queue up in the Message File and nothing will be printed. We want the submission of the job to happen automatically, but we only want the job submitted if an operator is about to do some Order Entry and the job has not already been submitted. This is a job for an EXEC Procedure. As terminal operators make a menu selection to do Order Entry, they will be sent to an EXEC which will Stream the Print job if it's not already running, then send the operator into the Order Entry Screen. The EXEC could be coded like this:

```
MENU: "Order Entry", KEY("Order Entry"),
                     (EXEC-ORDERS, SCREEN-ORDERS),

EXEC-ORDERS:
!$IF DATA-PRINTER.DATA.INFOSYS  $CANCEL
!$*
!$* Please wait a moment while I initiate
!$* the Invoice Print Job.
!BUILD PRINTER;REC=-256,1,F,ASCII;DISC=1
!$GS STREAM-PRINT-INV
!$*
!$ASKR 1 \Okay, we're all set now, <cr> To Continue\
!EOJ
```

The EXEC needs to test a condition to know whether or not the job is already running. EXEC procedures can check many types of conditions, one of them is the existence of a disk file. So for this application we'll establish a convention whereby whenever the Print job is running there will be a file created called PRINTER.DATA.INFOSYS. Upon job completion, the file will be purged. Therefore, if the file PRINTER.DATA.INFOSYS exists, then we can say that the job is running.

ii) Terminating the Print Job.

The Print job must be set up in a such a way that it will print one invoice at a time, and be receptive to requests for termination.

We can use the Message File to send *commands* to the Batch routine in addition to *Order Numbers*. So if we wish to be able to terminate the Job from our On-Line Session, we can establish a convention whereby the value **STOP** in the *TEXT* field will command the routine to terminate. Let's change the Extraction Phase of the Report to look something like this:

```
CALCUL 'NO' = #EXTRACTED;
FOR  FILE:MSGFILE  BEGIN

     IF TEXT <> 'STOP'     (* Cue to terminate *)
        THEN BEGIN;
             CALCUL 'YES' = #EXTRACTED;
             DISPLAY ORDER#, $TIMES;
             FOR ORDER-HEADER.ORDER-NO<ORDER#>
                 BEGIN;
                    .  .  .
                    .  .  .
                 END;
             END;
     BREAK;              (* Exit after Reading 1 Record *)
     END;

IF #EXTRACTED = 'NO'                (* EOF or STOP *)
   THEN BEGIN;
      COMMAND ('PURGE PRINTER.DATA.INFOSYS');
      DISPLAY 'I QUIT';
      END;

SORT ON .  .  .;
```

The BREAK Command limits the extraction phase to processing one Message file record. There are three possible outcomes:

* The Message File record will contain an Order Number. This Order will be extracted from the Database and the Invoice will be printed.

* The Message File record will contain the value **STOP** in the *TEXT* field. Accordingly, it won't extract an order for printing.

* An EOF condition will be returned by the File System. The Report will not extract an order for printing.

Based on the outcome, the flag #EXTRACTED will take on a value of **YES** or **NO** which conditionally triggers the COMMAND to purge the *PRINTER* file, and ultimately end the job.

The last missing piece is another EXEC procedure to control the operation of the Batch Report. This EXEC will establish a loop, continually actioning the Report which as we have seen will extract and print one order at a time. Based on the presence (or absence) of the *PRINTER* file, the EXEC will also control job termination. It looks something like this:

```
EXEC-BATCH:
!$TAG LOOP
!$IFF DATA-PRINTER.DATA.INFOSYS $GOTO EXIT
!$GS REPORT-PRINTINV
!$GOTO LOOP
!$TAG EXIT
!EOJ
```

Recalling that the Report uses the COMMAND Command to purge the Printer File upon receiving an EOF condition or **STOP** from the Message File, we can see that after the printing of each invoice we check for the presence of the *PRINTER* file. When the *PRINTER* file is purged, the job terminates.

A programming detail we have not covered yet is the On-Line transaction which sends the **STOP** command to the Message File. This would probably be set up as a menu selection restricted to the Order Entry Supervisor. Regardless of the menu security, the Menu Action would lead to an EXEC which checks for the existence of the *PRINTER* file. If it exists, then a Prompt Screen would be presented wherin the user would confirm the intent to stop the job, then write the appropriate record into the Message file. The Menu selection and associated processing is illustrated on the next page.

```
MENU: "Stop the Invoice Printing Job", EXEC-STOPINV;

EXEC-STOPINV:

!$IFF DATA-PRINTER.DATA.INFOSYS $GOTO OOOPS
!$PROMPT SCREEN-STOP
!$CANCEL
!$TAG OOOPS
!$* Oooops.  You wanted to stop the Printer Job, However
!$* my indications are that it is not running.  I cannot
!$* stop a job that is not executing.
!$* For your convenience I will show you a list of
!$* currently executing jobs:
!$PAUSE
!SHOWJOB JOB=@J
!$PAUSE
!EOJ

SCREEN-STOP: $PROMPT, A;

10,20, "Stop the JOB (Y,N)", REC[1-1], MATCH(Y,N),
       CALCUL A("REC[1-1] = #1"),
       COMPUTE-STOP;
END;

COMPUTE-STOP:A;

IF #1 = 'Y' THEN
   CREATE FILE:MSGFILE WITH
              0  = ORDER#,
          'STOP' = TEXT;

EXIT;
```

In operation this example works quite smoothly. Order Entry operators are able to perform their task without being aware that they are actively communicating with the batch print routine, with one exception: the first terminal operator (each day) is automatically taken through the EXEC procedure which submits the batch print routine. As they are generated, the invoices can be printed on any device including a departmental printer local to the data entry operators. The Invoices are printed in batch, at a lower priority than the On-Line processes so as not to impose a negative influence on terminal response times. The mechanism for initiating, feeding, and terminating the batch process is completely automated, and a manual override to stop the job is also available.

The design of this example involved the application of a Message File using one *Reader* (the batch Report) and multiple *Writers*. This configuration can be changed. For example, if there is a sufficient number of Order Entry terminals,

orders may be entered faster than they can be printed. It may therefore be desirable to have multiple *Readers*; ie: more than one Batch report job feeding from the Message File. Another configuration may involve multiple Print jobs and multiple Message files, with each Print job dedicated to one Message File. This might be useful if there are several pools of data entry people located throughout the organization each with their own local printer. Each print job could send its invoices to a specific printer, and based on the terminal operator's User Security Environment each user would feed a specific Message File.

This is just one example of using MPE Message files in a Speedware application. Some other applications might be:

- Feeding transactions to another system. Many sites use Speedware to embellish or customize the Data Entry and reporting functions of a purchased application package. In these instances there is often the need to interface the Speedware system with the processing routines provided with the package. Message files could send transactions to a *Background* processing routine which would update the Application Database.

- Undertaking physical deletes. Speedware provides an optional Logical Delete mechanism which provides several benefits in an On-Line Environment. When one adopts a strategy of doing Logical Deletion of data, the need is presented for the periodic physical deletion of the data. Often we defer this to a nightly/weekly/monthly batch job, however the Physical deletion could be done by a background processor, as records are flagged for deletion On-Line. The benefit to this approach is that the physical delete procesing remains a batch task (On-Line performance benefit), yet the processing is self scheduling.

It is the intent of this paper to introduce the general concepts behind MPE Message Files and illustrate how they might be put to good use in a Speedware application. If more technical information is required, the following sources offer a starting point for your research:

1) Reactor Reference Manual, Infocentre Corporation,

   Product Number RCT 5.00.00. A source for more detailed information regarding SCREEN formats and processing, FILE definitions, COMPUTE and REPORT processing.

2) MPE File System Reference Manual, Hewlett-Packard,

   Part Number 30000-90236. Provides the detailed documentation of the Operating characteristics, features, and file system intrinsics for Message Files. See Chapter 8.

3) "Interprocess Communication Using MPE Message Files",

   A technical paper submitted to the Detroit INTEREX Conference 1986, by Lars Borresen, Hewlett-Packard. Paper 3112 in the Conference Proceedings.