```
 ################# h ##################
##############     h         #############
###########       h            ###########
##########       hhh     ppp     ##########  H  E  W  L  E  T  T
##########      h   h   p    p   ##########
##########     h    h   p    p   ##########
##########    h    h   p ppp     ##########  P  A  C  K  A  R  D
###########          p         ###########
#############       p        #############
################ p ###################
```

COMPUTER SYSTEMS DIVISION
Accounting Systems Group
19447 Pruneridge Ave.
Cupertino, Ca. 95014
(408)-725-8111 X4257

Bill Vaughan, Accounting Systems Supervisor

Increased Reliability at a Lower Cost

ABSTRACT:

This paper will discuss various techniques utilized to increase the reliability of application software and to simplify the operations management of the HP3000. Topics presented will include:

- MONITOR
  - Complete system security, application control and friendly user interface in a single online program
- A GENERAL PURPOSE AUTOMATED CONTROL APPLICATION
  - How to insure that what one program writes to a file is the same as what the next program reads
- INTELLIGENT DATABASE CAPACITY CHECKS
  - How to prevent databases from hitting capacity, thereby avoiding time-consuming recovery and clean-up
- PRIVATE VOLUMES
  - How Private Volumes are meant to be used to
    (1) better utilize disc drives
    (2) insure data integrity and security
    (3) do system backup
- APPLICATION TESTING
  - A sound testing strategy that pays off in the long run
- IMAGE LOGGING
  - Showing its benefits both online and batch
- MISCELLANEOUS
  - Key file recovery after catastrophic crashes
  - Building files to avoid run-time aborts
  - Unique approaches to JCL, UDC's and MPE capability maintenance
  - Utilizing "INFO=" for passing parameters to COBOL programs

Background

The Accounting Systems Group of CSY reports to the CSY Controller and handles all accounting data processing for CSY. Our role within the Accounting Department is to support and develop computerized accounting systems.

In addition to support we have become heavily involved and dedicated to:

(1) Testing new HP products - both hardware and software. This includes not only doing pre-release testing for functionality and reliability but also utilizing these products to develop our distributed environment.

(2) Fully utilizing HP software and hardware to implement a "distributed" data processing environment, i.e. one in which the computing power is where the people and problems are. This includes addressing the problems of system security and operatorless-computers.

We currently have our applications spread across two HP3000 systems, a SERIES 44 and a SERIES 33, with a total of about 1000 Mb of disc storage (four of our disc drives are Private Volumes). One 2619A does the printing for both machines (we use DS/3000 to copy spoolfiles from the Series 33 to the Series 44). We have one HP125 microcomputer in the department and are currently evaluating an HP3000 to be put in Accounts Payable for dedicated processing. Our systems group of 12 professionals supports an accounting department of 40 people.

Monitor

Applications on the HP3000 are executed in either online or batch mode. Presently, Hewlett-Packard does not provide an automated system to assist users in running applications in a friendly way.

To start a batch process the user must log on to the appropriate group and account and know how to stream the desired job using its formal name. To run an online program, the user must know all necessary file equations as well as all necessary parameters of the MPE RUN command (LIB, MAXDATA,etc.). In both cases, these system formalities are of no concern to the user and he or she should not be forced to deal with the operating system.

With the advent of distributed processing and the emergence of open computer rooms, operatorless systems, and hardware (terminals, printers, and even computers) in the user's areas, the issue of "system security" can no longer be resolved with locks on the computer room door. We must now find ways to secure the system using the system itself as the holder of the keys. Our approach to this issue is to secure the Accounting structure of the 3000 system. Each Monitor controls a single account and access to the groups within that account. Access to MPE is only granted if passwords are known. Applications are run through Monitor so that no user (other than systems personnel) ever needs to get into MPE.

The entire Monitor application can be implemented WITHOUT ANY CHANGES to existing programs, or job streams. The system is database driven and is executed using a logon UDC with OPTION NOBREAK (see figure 1.0 for brief overview). The application uses existing HP3000 capabilities and is written almost entirely in COBOL II.

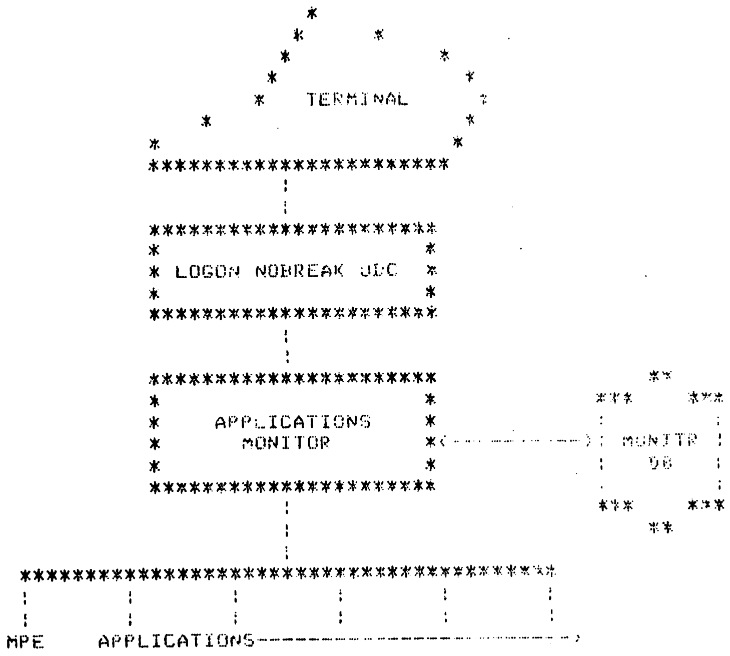

Figure 1.0 Monitor Overview.

A GENERAL PURPOSE AUTOMATED CONTROL APPLICATION

One of the tasks a systems administrator has to do is to verify control totals for batch jobs. Most of our batch jobs consist of multiple programs passing information through the use of sequential disc files and for certain jobs it is not enough to successfully reach EOJ to say that processing was indeed successful. We would have the programs within the job print out control reports to STDLIST and the system administrator would manually verify that the control totals matched.

We have designed and implemented an automated control logging procedure that (1) is standard for all application subsystems, (2) eliminates manual calculations thereby eliminating human error, (3) stores and reports the information being logged, (4) directs the systems administrator's attention to variances in the control totals, and (5) is simple and straight forward to implement.

This system, called Control Logs, is driven off a database that maintains the totals and the parameters that define how the system should handle the totals. All of the code to be inserted into each program is kept in our copylib.

The following example will show how Control Logs works. Let program "A" write out data to a file named "D". Then program "B" will use file D as an input file to do further processing. In program A, each time a record is written to D two totals are kept (in accounting we usually keep record count and a dollar total) and at the end of program A the control log subroutine is called. This takes the totals

puts them into the database, and then prints a standardized control report to STDLIST. Then program B comes along and as it reads file D it keeps the same two totals that A kept. At the end of its processing it also calls the control logs subroutine but it will compare its totals to those in the database for file D. If they do not match an error report is printed showing the differences and the subroutine aborts the program.

There are several functions that can be accomplished with Control Logs:

(1) Replace - this is used by the program creating a file as was program A in the example above. The totals taken are put into the database and no further processing is done

(2) Compare - this takes the totals being passed by the program and compares them to the totals presently in the database for that particular file. If they do not match the program is aborted and an error message is printed

(3) Update - this takes the totals bein passed by the program and adds them to the totals already in the database and the resulting new totals are then put in the database.

(4) Compare and Update - this does the compare function first with the first two totals passed and then does an Update using two optional totals that are used for this function and function (5)

(5) Compare and Replace - this does the Compare function first with the first two totals passed and then does an Update using the second two totals.

We now currently have all of our batch processing using Control logs for every file that is passed between two or more programs. Generally, it has not been inconsistent data that has led to Control Logs mismatches, rather it has been that when we fix bugs in programs we have inadvertantly introduced other bugs that affect these files.

INTELLIGENT DATABASE/FILE CAPACITY CHECKS

Recovering a database after a dataset has hit capacity is one of the more painful recovery processes. Using IMAGE LOGGING helps because you can recover right up to the process that filled the database but you still have to retrieve the database from your last backup and run the log file back in. It is also difficult to always keep right on top of the amount of free space each dataset within each database has. That 30% or so free space you like to maintain can disappear alarmingly quickly and when it does you are faced with what we refer to as a "capacity abort".

Along with databases, data files can be used in a similar manner where free space is maintained and information is "appended" to the file on a regular basis using "ACC=APPEND" on file equations.

What we have developed is a simple method of "up-front" capacity checking within programs so that processing can be terminated before any data is put to the database or file in the case where there will not be enough space available.

We do almost all of our programming in COBOL, and with COBOL II MPE INTRINSICS can be called directly so that none of what we are going to do requires any fancy subroutines or coding. By using "DBINFO" MODE202 calls for datasets and "FGETINFO" intrinsic calls for files, all current-count and capacity information can be obtained. From there it is only a matter of determining if what you have to put in will fit.

The following example will help:

```
    ********************
   /                    \
   :   TRANS DATA FILE   :
   \                    /
    ********************
             :
             :
            \ /
    ********************
    *                  *
    *    VALIDATION    *
    *     PROGRAM      *
    *                  *
    ********************
             :
             :
            \ /
    ********************
     *                *
      * VALID-REC-DTL*
       *  DATASET   *
        *          *
         **********
```

Figure 3.0 Database Capacity-check example.

We have a program that on a daily basis reads in a transaction file called "TRANS". Each record in TRANS is validated and put to a dataset called "VALID-REC-DTL" in the HOLDDB database. The VERY FIRST CODE within the PROCEDURE DIVISION does three things.

(1) Calls the intrinsic FGETINFO to find out how many records are in the TRANS disc file.

(2) Calls DBINFO MODE202 to get the current record count and the capacity of the VALID-REC-DTL of HOLDDB.

(3) Adds together the number of records in TRANS and the number of records currently in VALID-REC-DTL and if this total is GREATER THAN the capacity of the dataset the program aborts itself (after printing a message explaining the situation). Aborting the program aborts a batch process thereby effectively stopping processing so the database can be expanded.

The above example is a simple but very powerful case of how to avoid database recovery. As mentioned before, the some checking can be done for ACCESS=APPEND type files. Also a dataset or file can be tested for a percentage of full capacity. If say 95% is "too full" then the program can be aborted when 95% or greater is reached. The important point is that the check be put in at the beginning before any kind of processing has begun so that the program can easily be restarted from the beginning.

PRIVATE VOLUMES

In moving towards an operatorless environment Private Volumes have played a key role in several ways.

We now use private volumes for almost all of the processing that used to go to tape. An HP7925 disc pack can hold about 3 1600bpi tapes worth of information. Using theMPE command "VMOUNT ON,AUTO" with a private no one has to to "REPLY" when the pack is needed. All the advantages of disc access are available as well as the ease of removability and storage that tapes have.

We now use private volumes for all of our partial dumps (system backup). We use the volumes as "SERIAL DISCS" and so we backup to disc the same way we used to backup to tape.

We have found that using private volumes for backup is faster because of the lack of multiple tape mounts and that disc packs do not suffer from the parity errors that are frequent on aging tapes. In day-to-day use we have found that major system crashes that call for reloads seldom destroy the files on our private volumes so that only system-domain drives need to be reloaded. This has cut our reload time to less than half what it was before.

We have created a private volume called "SPACE" that has one group on it also called SPACE. This pack is mounted whenever we need to do extremely large sorts. What we do is to "point" our sort files to SPACE and do all of the sorting on this completely empty HP7925 pack (that's nearly 500,000 sectors of sorting space!).

APPLICATION TESTING

Being a systems group that does application maintenance as well as development we do alot of testing on existing systems as bugs are fixed and programs are enhanced. In the past we always tested with a subset of our "live" data in a group made just for testing and there were no rules or even guidelines on testing. What happened was that:

(1) Test data was easily destroyed as one person would purge files or alter data that another had set up. This usually didn't happen while both were testing. The first was done but when he came back a month later to use the data to test a new change in the program he found his prior data (which he knew and understood) gone or altered. This created a situation where test data integrity was nonexistent and much time and effort were wasted always having to recreate data.

(2) Because the test data wasn't actually designed it very seldom really "tested" the programs it flowed through. To be meaningful each different type of transaction must be included in the data and especially ones that fully exercise the area of the program that has been changed. Production data also generally contains a high volume of only a few types of these transactions so that when data is just being copied from the live data files it usually wasn't of much worth except to see if the program could run from beginning to end without aborting!

(3) Because the test data wasn't actually designed it was very seldom that the programmer REALLY understood the interrelationships within the data and the program. Very often the programmer was simply putting in code that he was told to put in and wasn't ever understanding the problem to be solved.

This list could go on and on. Our solution to the problem is what we call "TESTBASE".

TESTBASE is a group within our FINANCE account that is designed to be a complete, self-supporting environment for the testing of our Accounting software. By "complete" it is meant that using the procedures we've outlined, TESTBASE can be enhanced to fully test any "live" scenario desired. By "self-supporting" it is meant that closure exists within TESTBASE'S set of data.

Listed below is the set of guidelines that we have set up for TESTBASE. The way TESTBASE works in practice is that we keep the group TESTBASE as a "clean" copy and for each person we have a group for them to do their testing in. When we create this group we give them a copy of all the necessary files that they will need from TESTBASE and we make sure that when testing is finished that they go back and add to TESTBASE the new test data that they have developed. We have found that for TESTBASE to work requires a serious commitment and effort from management but that the testing time saved, the increased thoroughness and quality of testing, and the knowledge gained by the programmers makes TESTBASE one of the best investments we've made.

TESTBASE guidelines:

(1) TESTBASE should have complete closure. Any data needed for jobs, validation, etc. will be kept within TESTBASE. If needed, TESTBASE could be removed to another machine along with necessary program files and all testing could be accomplisned.

(2) Naming conventions should be independent from actual naming conventions. Most production naming conventions (eg. Part-Numbers) are not the result of predefined naming schemes designed to minimize start- up costs and overhead involved in user understanding. TESTBASE names will try to be as simple and orderly as is possible.

(3) All test data should be independent of actual production data values. In this way data can be designed to provide specific information to the testor.

(4) TESTBASE should be recoverable. This means that at any time the programmer may retreat back to time 0 and begin again with exactly the same scenario or environment he started with. Also, with not being tied to the production environment testbase or any part of it means TESTBASE can be stored at any point in time during testing and then at any time be recovered to that point for restart.

(5) TESTBASE should be dynamic in that whenever it is used, time should be taken to enhance the original testbase so that (a) it does not become outdated and (b) so that testbase grows with new databases and files being added to increase the range of systems that can be tested in the future without having to "reinvent the wheel" with each new user.

IMAGE LOGGING

In our environment we are using Image Logging for virtually all of our databases. All of our logging is done to disc and we've realized some unexpected benefits from having the logging processes.

First, we've seen no problems with logging to disc. When we build our log files we obtain the disc address of the files so that in a serious crash we can pull the files off to tape using SADUTIL (see MISCELLANEOUS section on Key file recovery).

Second, one of the biggest benefits from logging comes not from when the system crashes but when an application aborts and we need to recover the databases involved. It is nice being able to recover a database right up to the beginning of processing of an application and not lose previous processing to that database.

Third, the incremental processing time involved with logging is unnoticeable and implementation of logging requires no program changes. A common practice we've seen is to put DBSTORE'S at the beginning of job streams for recoverability. That definitely adds processing time!

MISCELLANEOUS TOPICS

Key file recovery after catastrophic crashes

There used to be a time when catastrophic system crashes meant a total reload of the system from the last backup. For Accounting this meant losing all the processing that had occurred from backup to the crash. However, with a little planning and a system utility called SADUTIL (see MPE Systems Utilities reference manual - Part No. 30000-90044) files on the inoperable system can be copied to tape and thereby recovered.

Planning needs to be done because if the system directory is destroyed in the crash, the only way to get the file is to know the logical device it resides on and its starting disc address. This information can be obtained from the MPE STORE command using the SHOW parameter or can be obtained by doing a LISTF within LISTDIR2.PUB.SYS.

In our environment we use both, depending on the specific file. For our Image Logging files we put LISTDIR2 in the job stream that builds the log files. This way the address is printed right on the STDLIST and filed with the backup listings. For the few strategic files that we need to keep track of their whereabouts on the system, we've put LISTDIR2 into the job streams that create the files. If the system crashes, we get the address of the file from the last STDLIST for that job and use SADUTIL to get our file back!

Building files to avoid run-time aborts

When building files within job streams the "DISC=" Parameter of the BUILD command can be used to allocate the entire amount of disc space needed. This is accomplished by setting the initial allocation equal to the number of extents (remember that DISC=[numrec][,[numextents][,initalloc]]]) so that DISC=10000,32,32 would allocate the entire space for 10,000 records or fail due to lack of disc space. This way lack of disc space will abort the job stream outside of, and before the program that would've used the file.

How not to lose JCL

In the past we had alot of problems with the STDLIST that is printed for every batch job. First, it seems as though STDLIST's page eject the line printer about every other line. This always destroyed any attempt to keep paper piling properly and jammed the printer on a regular basis. Secondly, user's don't understand the importance of STDLIST's so they got thrown out, filed with reports, inadvertantly left attached to somebody's output (and therefore thrown out),etc. It always seemed to be the case that the STDLIST was missing for that critical job that aborted and trying to fix the job became a difficult task.

We have implemented a simple solution that has solved our STDLIST problems. Any job that has a STDLIST worth saving has had "OUTCLASS=LP,5" added to its job card. This defers the STDLIST so it doesn't print (our OUTFENCE is normally 7) until a systems person

prints it off. We print off the STDLIST's once a day and file the them by the day we print them off (this avoids having to separate them). Now our printer jams alot less often and we always know exactly where the STDLIST's are!

MPE IV'S "INFO=" PARAMETER

MPE IV has a new parameter, called "INFO", for the RUN command. Using INFO alphanumeric information can be passed to application programs. Before INFO the only run-time parameter for passing information was "PARM", and it handled only numeric information.

We have written an SPL subroutine that retrieves the alphanumeric string and the length of this string to a COBOL II program. A simple COBOL example and the subroutine listing follow:

```
WORKING-STORAGE SECTION.

01  INFO               PIC X(80).
01  INFO-LENGTH        PIC S9(04) COMP.

     CALL "GETINFO" USING INFO, INFO-LENGTH.
```

From MPE:

```
:RUN PROGRAM;INFO="HELLO-THERE"
```

```
PAGE 0001   HEWLETT-PACKARD 32100A.08.01 SPL[4W] SUN, SEP 20, 1981

00000 0   $CONTROL SUBPROGRAM,MAP,ADR,SEGMENT=GETINFO
00000 0   $TITLE  "FMS0615S    -    GETINFO"
00000 0
00000 0   <<  THIS PROGRAM WILL RETURN AN 80 CHARACTER STRING  >>
00000 0   <<  AND THE LENGTH OF THIS STRING                    >>
00000 0   <<  TO A COBOL PROGRAM THAT WAS RUN WITH THE "INFO=" >>
00000 0   <<  OPTION.  THE ADDRESS OF THE STRING IS STORED IN  >>
00000 0   <<  Q-5 AND THE LENGTH IS STORED IN Q-6 AT RUN TIME. >>
00000 0
00000 0   BEGIN
00000 1
00000 1   PROCEDURE GETINFO(INFO,LEN);
00000 1    ARRAY INFO;
00000 1    INTEGER LEN;
00000 1
00000 1   BEGIN
00000 2     LOGICAL QSTART=Q;
                  Q +000
00000 2     INTEGER DELTA,X;
                  Q +001
                  Q +002
00000 2     BYTE POINTER PINFO;
                  Q +003
00000 2     POINTER P'LEN,PQ,PREG;
                  Q +004
                  Q +005
                  Q +006
00000 2     POINTER W'PINFO;
                  Q +007
00000 2     LOGICAL VAR;
                  Q +010
00000 2
00000 2     X := @QSTART;          << SET POINTER PQ TO CURRENT   >>
00003 2     @PQ := X;              << VALUE OF Q-REGISTER         >>
00005 2
00005 2   AGAIN:
00005 2
00005 2     DELTA := PQ;           << FIND OUT HOW MUCH TO CHANGE Q >>
00007 2     @PREG := @PQ(-2);      << SET POINTER TO VALUE OF       >>
00012 2                            <<  P-REGISTER IN STACK MARKER   >>
00012 2
00012 2      @PINFO := @PQ -  5;      << SET POINTER TO WHERE
00015 2                               << RUN INFO ADDRESS WILL BE>>
00015 2
00015 2     @P'LEN := @PQ -  6;     << RETURN LENGTH OF >>
                                    <<STRING FROM Q(I)-6>>
00020 2     LEN := P'LEN;
00022 2
00022 2     IF PREG = 0             << END OF STACK CHAIN,    >>
                                    << CAN STOP NOW           >>
00023 2        THEN GOTO STOP'THIS;
```

```
00025 2
00025 2      X := @PQ;
00027 2      @PQ := X - DELTA;            << DECREMENT Q-PTR >>
                                          << TO PREVIOUS     >>
00032 2                                   << VALUE AND START AGAIN>>
00032 2      GOTO AGAIN;
00033 2
00033 2
00033 2    STOP'THIS:
00033 2
00033 2       @W'PINFO := @PINFO;               << SET UP THE INFO>>
                                                << POINTER   >>
00035 2       VAR := W'PINFO;
00037 2       @W'PINFO :=  VAR;
00041 2       @W'PINFO := @W'PINFO / 2;
00044 2       MOVE INFO:="
                        ";
00100 2       MOVE INFO := W'PINFO,(LEN);     <<RETURN INFO >>
                                                 << STRING >>
00104 2
00104 2    END;


        IDENTIFIER            CLASS           TYPE        ADDRESS

     AGAIN                  LABEL                         PB+005
     DELTA                  SIMP. VAR.       INTEGER      Q +001
     INFO                   ARRAY      (R)   LOGICAL      Q -005
     LEN                    SIMP. VAR.(R)    INTEGER      Q -004
     P'LEN                  POINTER          LOGICAL      Q +004
     PINFO                  POINTER          BYTE         Q +003
     PQ                     POINTER          LOGICAL      Q +005
     PREG                   POINTER          LOGICAL      Q +006
     QSTART                 SIMP. VAR.       LOGICAL      Q +000
     STOP'THIS              LABEL                         PB+033
     VAR                    SIMP. VAR.       LOGICAL      Q +010
     W'PINFO                POINTER          LOGICAL      Q +007
     X                      SIMP. VAR.       INTEGER      Q +002


00000 1    END.


        IDENTIFIER            CLASS           TYPE        ADDRESS

     GETINFO                PROCEDURE


 PRIMARY DB STORAGE=%000;     SECONDARY DB STORAGE=%00000
 NO. ERRORS=0000;             NO. WARNINGS=0000
 PROCESSOR TIME=0:00:01;      ELAPSED TIME=0:00:07
```