HEWLETT-PACKARD GENERAL SYSTEMS USERS GROUP

FEBRUARY, 1980 MEETING

RESOURCE OPTIMIZATION SERIES

BLOCK MODE TECHNIQUES:

CONCURRENT DATA ENTRY AND UPDATE PROCESSING

Presented by:

John Korondy
HEWLETT PACKARD CO., INC.
CUSTOMER SERVICE CENTER
Mountain View, CA 94043

S U M M A R Y

1.1. Input/output is expensive but necessary.

There are several ways to optimize machine performance. The method I would like to present to you will significantly reduce costly input/output data transfers.

The two approaches to cutting down on wasteful use of I/O are: to reduce the number of bytes transferred to a minimum; and to reduce the number of times the data transfer is initiated. David Brown of the N.I.C.E. Corporation has made great strides in the latter by combining control characters with data in a single buffer, and then performing the output.

The technique that is described in the following will take advantage of both reducing the number of characters transferred, and reducing the number of transfers. It also incorporates a few of the principles of downloading some functions to the HP264x terminal from the mainframe, that prompted Bill Gates to develop the much-heralded LOBOL system.

1.2. The data to information ratio.

The data we store in our data bases, MPE files or KSAM files we often refer to as "raw data". This choice of phrase implies to me that the computer does not really store information -- elements of data with a clearly defined set of inter-relationships -- but rather it stores data: a seemingly meaningless and random collection of quantifiers, qualifiers and descriptors. It is then left up to a subsystem, an application program or the users' own intelligence to try to decipher the true meaning of the elements of data and the relationship one bears on another.

The users rarely if ever look at raw data through FCOPY or a dump, rather they depend on reports formatted by application programs or QUERY. These programs turn the data into information -- at some cost. To illustrate the point, let us assume that we have an employee data base with the usual fields. The data element that looks like '790511' would probably not mean too much to a personnel administrator. With enough experience one might guess that it is an employee's date of hire, or date of promotion, or date of last review, or date of termination, etc.

QUERY will transform the data retrieved from the data base to information, which might look like this:

DATE-LAST-REVIEW =790511

which means that QUERY has to buffer out 24 characters in total to translate 6 bytes of data. This results in a data-to-information ratio of 0.25 (6/24), or 25%.

1.3. Reducing overhead by increasing the D/I ratio.

It should now be appearent, that if we manage to reduce the number of bytes required to transform raw data to information, and therefore increasing the data-to-information ratio, we would achieve some reduction in our operational overhead and improve performance.

If we could somehow display the text that is fixed for each inquiry, and keep it on the screen as long as there is more processing to be done, we would never have to write it out again -- conceivably increasing the D/I ratio to approach the value of 1.

We can accomplish this improvement by painting a form on the screen at the start of the application program, protect it on the screen to prevent overwrite, and upon each inquiry, we should only have to "fill in the blanks". If in one day we intend to make 200 inquiries, and we assume that we deal with 240 bytes of data (40 fields of 6-byte elements, or any combination), we would have to write to the screen 48,000 bytes of raw data. By using a form that is 2,400 characters long (24 lines of 100 bytes each) we would get a D/I ratio of 95.24% [48000/(48000+2400)]. If the user had to use QUERY to gain access to the same amount of data, it would have to display 192,000 characters in total. The D/I ratio would still stand at 25%.

1.4. The visual orientation of screens.

An additional benefit of using screens is that the information displayed is visually oriented. When QUERY or FCOPY displays a set of data elements on the screen, they do so sequentially; i.e.: the screen scrolls up as more lines are printed. Inevitably, one will have to search for the item of interest, and might even have to roll the screen back down.

When using a form or a set of forms, the same element of data (i.e.: employee name) will always appear at the same location on the screen. Users who are only interested in selected items will quickly get used to finding those fields in fixed locations, thereby eliminating the need to

to search through unwanted data.

1.5. Ease of error recognition and correction.

When forms are used for data input functions, another advantage will quickly become obvious. Errors that have been detected by the data entry driver program may be easily brought to the attention of the user by blinking the field in error; and in some cases of obvoius typographical errors displaying an error message may not be necessary. This kind of warning is easily detected by the user, and the correction of such a mistake is quite easy as a simple overtype is required in block mode.

Block mode operation will be discussed later, and its advantages will come to light as we become more familiar with the concept. Let it suffice to note that once an error has been recognized, one may tab to the field in question, or use the cursor control keys to get there; which is, of course, a lot simpler than re-entering the entire record.

1.6. Ease of downloading class edits.

Downloading means the process by which a system tasks the terminal with performing a function heretofor accomplished by the mainframe (CPU).

The greatest advantage of dowloading is that while the the terminal is performing some task, the CPU can be gainfully employed doing something else. The noticeable effect will be the synchronization of entry programs to the users' needs -- less wait-time for the operator while the CPU is performing the edits, for example, as some of these edits can take place as the data is entered. Most of our users have already bought and paid for quite a few HP-264x type terminals. These include a fast 8080 processor, so it is time we took advantage of that processing capability. Downloading class edits -- alphabetic only, numeric only, and alphanumeric -- are incorporated into the 2645 package; they are well documented in tye 2645 Reference Manual, and are very easy to interface to application programs.

2. THE ADVANTAGES OF BLOCK MODE OPERATION

2.1. Full duplex character mode I/O.

Full duplex means two-way communication where both transmitting and receiving functions are enabled. To illustrate full duplex terminal commucations, I will outline the events that take place between the terminal

and the computer when a character is typed in on the 2645.

a. Terminal generates an interrupt
b. Computer sends a read-enable (DC1)
c. Terminal transmits a character
d. Computer echoes character back

It is important to remember, that in full duplex mode the terminal's echo facility is DISABLED, while the computer's echo facility is ENABLED.

Character mode I/O means that each time a character is struck at the terminal it will be sent to the computer. Block mode I/O means that data transfers are initiated from the terminal by way of the ENTER key (or by some other means I will discuss later), and an entire line or page is sent to the HP-3000.

It is interesting to note that the BACKSPACE key represents a valid ASCII character which can be transmitted, but the cursor control keys have only a local function. This is the reason why the cursor control keys do not work in character mode operation, but the BACKSPACE key works in both character and block mode.

2.2. Half duplex block mode I/O.

Half duplex means two-way communication with one side in receive only mode, the other in transmit only. The terminal can send, in which case the computer will receive, or the computer will transmit and the HP-2645 will receive, but simultaneous transmit and receive would mean full duplex mode. In normal half duplex operation, the echo facility of the computer is DISABLED and the echo at the terminal is ENABLED. When you turn the echo off by either entering an Escape semicolon or using the FCONTROL intrinsic, the computer's echo facility will be turned off. The terminal will still assume full duplex mode, and therefore it will not start echoing -- so the echo in essence will disappear.

In block mode, individual characters are not transmitted to the computer. One page or one line sent at one time, depending on the internal strapping of the terminal. In normal operation, the terminal is strapped for line transmission, so if page mode send is required, the terminal must be re-strapped by issuing an escape sequence. Since the terminal is transmitting one block of text at a time, it will assume echoing the individual characters, and the computer's echo should be turned off.

2.3. Additional controls available in block mode.

2.3.1. Increased usefulness of keyboard lock/unlock.

In character mode the computer automatically disables some features of the keyboard. When you :RUN FCOPY, while the program loads, there is nothing you can enter at the terminal. In block mode, the keyboard may be easily placed under program control. This allows the program to return control to the user while it goes out to lunch to perform some lengthy data base updates or compound computations or both.

2.3.2. Meaningful cursor control.

In character mode, the cursor control keys are very close to being useless. We still have to pay for them, though, as they are an integral part of the 2645. So, why not take advantage of them? The most use can be realized from these keys by direct tabbing or cursor movement to specific fields in forms mode. The only disadvantage of these keys is the learning curve of the operators in knowing when to use them and when not to.

3. IMPROVED METHODS OF BLOCK MODE DATA ENTRY

3.1. Designing, saving and updating the form.

Designing and creating a form on the HP2645 is relatively simple if the pogrammer is familiar with the available features of the terminal. I recommend that before attempting to design the screen, one reads the applicable chapters of the HP264x Reference Manual.

Forms should be painted manually onto the screen while logged on to a group and account where the form file will be kept, even if temporarily. The REMOTE latching key of the terminal should be in LOCAL mode (up). I recommend, that only 79 graphic characters be used on any one line, leaving the last 'print' position blank, as this practice makes the form a bit less cumbersome to work with. It has something to do with carriage control characters, as we will see later.

Once the form has been painted on the screen, we should test it, still in local mode. The testing is especially well advised, if using a complex form with editing in forms mode, or transmit only fields. Complete testing includes manually clearing the display in forms mode to assure that none of the display control sequences have been included within the unprotected fields.

Once the form is in an acceptable shape, we can turn off the forms mode and move the cursor below the form. We can now latch the REMOTE key to remote; and we are ready to save the form in a file.

We must first build the forms file. For a one-page form, a file of 24 records is sufficient, for a two-pageer 48, etc. A form of the :BUILD command that works is shown below:

```
:BUILD FORMFILE;REC=-256,,F,ASCII;DISC=24;NOCCTL
```

There are two very convenient ways to get the form from the screen into the file, and I will illustrate the simpler of the two.

First we must set up at least one FILE equation:

```
:FILE KBOARD=$STDIN;REC=-256,,F,ASCII
```

but this one will also be most useful a little later on:

```
:FILE SCREEN=$STDLIST;REC=-256,,F,ASCII
```

These file equations allow us to handle records greater than 80 bytes without causing confusion about carriage control on the terminal. Both system-defined files ($STDIN and $STDLIST) in a session default to 80 byte record sizes.

Now we invoke FCOPY by the

```
:RUN FCOPY.PUB.SYS
```

command. The FCOPY command to copy the form off the screen to the file called FORMFILE is

```
>FROM=*KBOARD;TO=FORMFILE
```

After issuing the above command, the cursor will sit there at the bottom of the page, waiting. We need to home the cursor, and it should now be under the top left corner of the form. We now press the ENTER key to transmit the first line of the form. If the echo facility is enabled, some garbage may be echoed back to the screen, but it will not harm anything. We press the ENTER key again, and again, until the last line of the form has been transmitted. Once we copied the entire form, we can send an end-of-file to FCOPY by typing in a colon (:) and pressing the RETURN key. This will get us out of FCOPY.

We can :RUN FCOPY.PUB.SYS again, and copy

```
>FROM=FORMFILE;TO=*SCREEN
```

to see how the file actually looks on the screen. This is one method to update the form file. We could now delete a line, or add a line, or correct a mis-spelled word, or enlargen a field. Once all the corrections have been made, the form can be copied on top of the old FORMFILE using the method illustrated above.

Notes: The recommended length of a line of the form is 79 bytes because when we use FCOPY to copy the form back to the screen, FCOPY would supply a CR-LF (carriage return line feed pair) after each record. If each record would contain 80 graphic characters, the terminal would also generate an overflow CR-LF pair, causing a blank line to appear between each line of the form.

The record length of the form file in the above examples is 256 bytes. It is a nice even number, allowing plenty of room to accomodate even the most complex escape sequences. It is important to keep in mind, that while only 79 or less characters are visible to us, the computer is working with considerably more bytes, since all of the control characters are valid ASCII codes.

The second method of saving a form in a file -- not described above -- is a method using the PTAPE program, which reads in the entire screen all at once, instead of lineby-line, like FCOPY. The terminal needs to be strapped for page transmission for PTAPE, and I do not recommend this method.

The second method of updating a form file -- not discussed above -- is using the EDITOR. You must be prepared to accept, that the EDITOR will truncate each record to 255 bytes -- no great loss, you can even create your file and set up your file equations for 255 bytes -- and also should be comfortable in reading control sequences with DISPLAY FUNCTIONS turned on. Using this method has far too many pitfalls for me to cope with, that is why I recommend the illustrated method of update.

3.2. Painting the form -- PUTFORM overview

Painting the form programmatically is even simpler than using the FCOPY method. I have contributed a COBOL, FORTRAN and SPL callable subroutine called PUTFORM. PUTFORM is coded in SPL, and it allows the calling program

to specify a fully qualified file name, which will save issuing a file equation from COBOL. It is also designed to 'squeeze' all trailing blanks out of each record of the form file.

The application notes and COBOL example are a part of the source listing. If you want to use all 80 bytes of of the form, line 62 of PUTFORM should be modified, so the third parameter of the PRINT intrinsic is %320, instead of zero.

3.3. Using the soft keys for function selection.

When a program issues a 'hard reset' -- Escape E -- to the terminal, all the soft keys will be set to their 'natural' state, i.e.: f1 will transmit a word that contains the ASCII characters Escape and p, or the octal value of 15560. the second soft key, f2, will send an Esc q, or %15561, and so on to f8, or Esc w.

Which of the soft keys have been pressed, therefore, can be programmatically evaluated quite simply. COBOL makes this process even more convenient if the programmer uses level 88 definitions for each of the soft keys, as

```
01 FUNCTION-KEYS          PIC X(02).
   88 F1  [                       ]       VALUE "%p".
   88 F2  [ % = Escape   ===>     ]       VALUE "%q".
   88 F3  [                       ]       VALUE "%r".
```

and so on. The input can then be evaluated by a simple conditional, such as

```
IF F1
   PERFORM E400--SCREEN-EDIT
ELSE
   IF F2
      PERFORM D200--DELETE-RECORD
   ELSE
      ...
```

If the program falls through past the condition test for F8, the user pressed the ENTER key, which may or may not be meaningful to the program, and can be treated accordingly. This method of using the soft keys may eliminate the need for a function selection field on the screen, and one or more keystrokes during data entry.

3.3. Reading the data -- BLOCKGET overview.

Reading the data from a screen and manipulating it is not as easy as issuing a READ from COBOL or FORTRAN -- unfortunately. The terminal inserts funny control characters in the middle of the record, which then must be

stripped off. This can get quite involved, especially in COBOL. Another question that might arise is how to get the terminal to send me some data once I detected the appropriate function key without having to ask the user to press the ENTER key.

These are the reasons that prompted me to write and contribute the COBOL, FORTRAN and SPL callable subprogram, BLOCKGET. I would like to acknowledge the cooperation of Mr. Alan Pound in the development and testing of BLOCKGET; without his help, it would not have been so easy to use, nor as efficient as it is today.

BLOCKGET, when called, prompts the terminal to transmit a block of data. If the HP2645 has been strapped for page transmission, it will send all the data in unprotected and transmit-only fields; if the terminal has been left in its native state, in line mode, BLOCKGET will fetch data in unpotected or transmit only fields on one line of the screen. BLOCKGET will strip out all the control characters, such as unit-separators (US) and record-separators (RS) that have been inserted into the data by the terminal.

BLOCKGET will return to the calling program one record, in a form that would have been received if a conventional READ would have been executed.

The calling parameters are BUFFER, LENGTH and ERROR. BUFFER is the area of storage where the data will be returned to. It needs to be large enough to accomodate all the control characters as well as the data. The formula for computing the length of the buffer is supplied with the BLOCKET documentation and source code. LENGTH is the size of BUFFER, in bytes. ERROR is a word where BLOCKET will return the binary value of 1 if the READ from the terminal has failed. This value will normally be set to zero.

A COBOL example of using BLOCKGET is provided with the code.

3.5. Concurrent processing.

3.5.1. Foreground data entry.

The terminology of foreground and background may be a a bit confusing here. I am merely using the terms to illustrate what is happening at the user level; i.e.: at the terminal -- foreground --; and what is taking place at the HP3000 CPU -- background. Shortly we will see that one will no longer have to wait for the other, and they both can keep busy without one

impacting the performance of the other. Hence the distinction between two separate processes: foreground and background.

In block mode, the program can control when the screen can be written to by unlocking the keyboard; and it can also lock the keyboard out to prevent the user from writing to the screen while the computer outputs data or is in the process of painting the form.

It is important to note, that the terminal acts as if it was in LOCAL mode, so when the keyboard is unlocked, anything is possible. The terminal does not wait for an enable character before it allows the user to input data. This is in contrast to the process of data transmission in character mode, such as when you type in :RUN FCOPY.PUB.SYS, the computer goes out to lunch for a length of time, and nothing can be entered through the keyboard.

So then it is possible to enter an entire screen of data without ever having to bother the CPU in block mode. The mainfraime will not recognize our attempt to communicate with it until a soft key or the ENTER key has been hit. This facility is enabled through a a call to the FCONTOL intrinsic. The terminal can also be placed in block mode from the program by issuing an escape sequence to latch the BLOCK MODE key.

3.5.2. Background calculations/updates.

We have seen earlier how the terminal can be made to function independently of the CPU; therefore it is possible to liberate the mainframe from being a slave to the terminal; i.e.: waiting for input.

In character mode, when a program executes a READ, it will go into a wait state, until it is reactivated by an interrupt generated by a line termination character or the ENTER key at the terminal. So, the CPU is not really doing anything for me while I am pecking away at the keyboard. It can, however, accomodate requests from the nebulous 'other users'. If I, as a user, have to pay for two processors -- the CPU and the one inside my terminal -- I should be allowed to get the best use out of both of them.

Once the user has entered a screenful of data and the program has satisfactorily performed the required edits, perhaps all that remains is to generate some computed fields and to update some data sets of some data bases. This is the process that is time

consuming and most taxing on the users, because they have to wait for the system to lock the data sets or items being updated, perform the update(s) and release the locked entities. In some instances I observed, this can result in time spent by the user waiting for the computer of 40 to 80 seconds depending on loading.

This waiting time can now be eliminated, as the program can clear the data off the screen and release the keyboard as soon as the edits have been performed, and then proceed to execute the lengthy tasks of updating while the user is working on the next record. The CPU response time can be decimated in the above manner.

Fatal errors can still be handled by interrupting the data entry in progress, and displaying the error message on the next page of the screen, pausing for, say, 3 seconds, and re-writing the previously entered record to the screen. The probability of these kinds of errors can be greatly reduced by proper design and foresight, and comprehensive edits.

3.5.3. Terminal handling.

As I mentioned earlier, several terminal control functions need to be performed before the concurrent data entry technique can be used effectively. The procedures required to set the HP2645 up for this purpose are explained below.

a.	HARD RESET	(Escape E)	Initilizes 2645
b.	Open the data base(s) or perform something that takes at least 200 milliseconds at the mainframe, or pause for 200 ms as to prevent a race condition at the terminal.		
c.	KEYBOARD DISABLE	(Escape c)	Locks keyboard
d.	DISABLE BREAK	(FCONTROL 14)	Disables BREAK key
e.	STRAP FOR PAGE	(Esc &s1D)	Straps the terminal for page mode.
f.	PAINT THE FORM	(PUTFORM)	
g.	DISABLE ECHO	(FCONTROL 13)	Disables echo
h.	FORMS MODE ON	(Esc W)	Turns on forms mode
i.	ENABLE BLOCK MODE	(FCONTROL 29)	Enables block mode
j.	LATCH BLOCK MODE	(Esc &k1B)	Latches the BLOCK MODE key at terminal
k.	SETMSG OFF	(COMMAND)	Turns message off
l.	ENABLE KEYBOARD	(Escape b)	Ready for input

For those users who run under a pre-1918 release of MPE, I have contributed an FCONTROL intrinsic interface, called TCONTROL, that allows COBOL programs

to take advantage of the flexibility in terminal handling of FCONTROL.

3.3.5. Recovery from errors in the background process.

I have touched upon the subject of handling terminal errors that occur in the background. Some thought should be given to where these error messages will be displayed on the screen. Usually, it seems best to print the messages on a separate page of screen, this way we do not have to provide for a separate, and usually large, field for error messages.

In case of terminal errors, it is important to provide one and only one exit for the program. We have, in the previously described process, reconfigured the terminal for block mode, and it must be reversed before it can be used for character mode transfers.

5. PROCESS TERMINATION.

The last paragraph, unit, or procedure performed by the data entry program should be one that puts everything back to the way it found it. This is significant for the communication protocols.

The facilities that we changed using the FCONTROL intrinsic must be restored the same way -- echo, block mode, break -the message facility should be enabled using the COMMAND intrinsic, and the terminal should be restored to its native state. For the restoration of the terminal we can employ a short-cut: the hard reset (Escape E), which will do the trick.

```
$CONTROL SUBPROGRAM,NOWARN
<<
================================================================

                        P U T F O R M

                    Programmer:   John Korondy
                    Date written: Sep 11, 1979

The PUTFORM subprogram allows COBOL (or any other) programs to call
it and to efficiently paint a form or sets of forms on the $STDLIST
device. This should, of course, be a terminal.

The only required parameter is the name of the forms-file, which may
be fully qualified, and must include the lockword (if any). The name
of the forms-file must be in a byte array [PIC X(xx).] and the last
character of this array MUST be a space.

COBOL example:

      01 FORMS-FILE      PIC X(20)  VALUE "ORDFORM/WD40.IM.DB ".
      ...
         CALL "PUTFORM" USING FORMS-FILE

NOTE: The PUTFORM subprogram handles all externally caused error
      conditions, and will terminate the program with an appropriate
      error message.

================================================================
>>

BEGIN
  PROCEDURE PUTFORM(FNAME);
  ARRAY               FNAME;

  BEGIN
    ARRAY             DISPLAY'CLEAR(0:1);
    BYTE ARRAY        FILE'NAME(*)=FNAME;
    BYTE ARRAY        FORM'BUFF(0:255);
    INTEGER           FNO, LGTH, DUMMY, ERROR;

    INTRINSIC         FOPEN, FCLOSE, FREAD, PRINT, QUIT;

    FNO:=FOPEN(FILE'NAME,5,%300);
    IF < THEN
         BEGIN
           MOVE FORM'BUFF:="THE FORM-FILE CANNOT BE OPENED.";
           PRINT(FORM'BUFF,-31,%60);
           QUIT(FNO);
         END;
    MOVE DISPLAY'CLEAR:=(%15550,%15512);
    PRINT(DISPLAY'CLEAR,2,%320);
READ'LOOP:
```

```
    FREAD(FNO,FORM'BUFF,-256);
    IF > THEN GO EXIT;
    IF < THEN QUIT(33);
    LGTH:=256;
    WHILE (LGTH:=LGTH-1)>0 AND FORM'BUFF(LGTH)=" " DO;
    LGTH:=LGTH+1;
    PRINT(FORM'BUFF,-LGTH,0);
    GO READ'LOOP;

EXIT:

    PRINT(DISPLAY'CLEAR,1,%320);
    FCLOSE(FNO,0,0);
  END;
END.
```

==========

B L O C K G E T

Programmer: John Korondy
Date written: Sep 17, 1979

The BLOCKGET subprogram facilitates an easy read in block-mode from the 2645 terminal. It emulates the ENTER key beinq pressed, homes the cursor and buffers in the data from the unprotected and transmit-only fields.

The teminal should be strapped for block mode, else only one line will be transmitted. The BLOCKGET handles all the communication protocol and strips the data of all unit separatotrs (Us) and record separators (Rs). The buffer that is returned to the calling program will contain the contiguous string of data actually input. The BLOCKGET subprogram generates all of its own error messages.

COBOL example:

```
01 FUNCTION-KEY        PIC X(002).
   88 F1                          VALUE "esc p".
   88 F2                          VALUE "esc q".
   ...
01 GROSS-BUFFER        PIC X(944).
01 GROSS-LENGTH        PIC 9(004)  COMP VALUE 944.
01 BLOCKGET-ERROR      PIC 9(004)  COMP.
   ...

      CALL "BLOCKGET" USING GROSS-BUFFER
                            GROSS-LENGTH
                            BLOCKGET-ERROR
      IF BLOCKGET-ERROR = ZERO
         MOVE GROSS-BUFFER TO SCREEN-BUFFER
      ELSE
         DISPLAY "BLOCKGET ERROR --- etc".
```

NOTE: The GROSS-BUFFER shown above must be lonq enouqh to accomodate all the unit and record separators. The formula for computinq this is

```
LENGTH = B + UF + TF;
```

where

B = number of actual bytes of data expexted
UF = number of unprotected fields on the screen
TF = number of transmit-only fields on the screen.

==========

```
  BEGIN
    PROCEDURE BLOCKGET(BUFFER,TOTLGTH,ERROR);
    INTEGER             TOTLGTH,ERROR;
    ARRAY               BUFFER;

  <<
Page 16 of 19 pages.                                          >>


    BEGIN
      INTEGER           DUMMY, LGTH;
      LOGICAL           TRIGGER;
      BYTE ARRAY        BUFF(*)=BUFFER;
      BYTE ARRAY        TEMP(0:1099);
      INTRINSIC         READX, PRINT;

      TRIGGER:=%15544;
      PRINT(TRIGGER,1,%320);
      LGTH:=0;
      READX(TEMP,-TOTLGTH);
      IF < THEN ERROR:=1
      ELSE      ERROR:=0;
      DUMMY:=-1;
      WHILE (DUMMY:=DUMMY + 1) < TOTLGTH DO
      IF TEMP(DUMMY) > %37 THEN
         BUFF(DUMMY - LGTH):=TEMP(DUMMY)
      ELSE
         LGTH:=LGTH + 1;
    END;
  END.
```

```
$CONTROL SUBPROGRAM
<<
================================================================

                        T C O N T R O L

                    Programmer:   John Korondy
                    Date written: Sep 17, 1979

The TCONTROL subprogram allows COBOL programs to freely use the
versatile MPE intrinsic FCONTROL. Only two parameters need to be
supplied: a control code in integer form, and an error-code.
The control code must be one of those described in the MPE INTRIN-
SICS Manual under the intrinsic FCONTROL. The error-code is retur-
ned, if it is zero, no error occured; else the task requested
was not succesfully completed.

COBOL example:

          01 TCONTROL-CODE         PIC S9(004)  COMP.
          01 TCONTROL-ERROR        PIC S9(004)  COMP.
          ...

             MOVE 13 TO TCONTROL-CODE
****************************************** TURN THE ECHO OFF.
             CALL "TCONTROL" USING TCONTROL-CODE
                                   TCONTROL-ERROR
             IF TCONTROL-ERROR GREATER ZERO
                DISPLAY "ECHO CANNOT BE DISABLED."
             ELSE
                PERFORM THE-GREAT-MIRACLE.

================================================================
>>
BEGIN
  BYTE ARRAY          INPUT(0:5):="INPUT ";
  PROCEDURE TCONTROL(PARM,ERROR);
  INTEGER             PARM,ERROR;

  BEGIN
    INTEGER           IN, CTL;

    INTRINSIC         FOPEN, FCONTROL;
    IF PARM < 10 OR PARM > 43 THEN
       BEGIN
         ERROR:=1;
         RETURN;
       END;
    CTL:=PARM;
    IN:=FOPEN(INPUT,%45);
    IF < THEN
       BEGIN
         ERROR:=2;
```

```
        RETURN;
      END;
    FCONTROL(IN,CTL,ERROR);
    IF < THEN
      BEGIN
        ERROR:=3;
        RETURN;
      END;
    ERROR:=0;
  END;
END.
```