Using COBOL II's Facilities

By: Patrick A. Lockwood

Orion Systems Technology, Inc.

1309 East Northern Ave., Suite 701

Phoenix, AZ 85020

INTRODUCTION

This paper is targeted at analysts/programmers who are familiar with COBOL, but who have not had much experience utilizing it on the HP3000.

HP's implementation of 1974 ANSI Standard COBOL provides the designer and the programmer with many tools to help develop robust systems, with techniques that rival the current crop of fourth generation languages for speed, and allow the use of concepts not usually associated with this high level language.

This paper will explore techniques used to quickly develop systems in COBOL II (most of which is upwardly compatible with the ANSI '85 compiler), and to accomplish this without sacrificing quality.

Some topics to be covered are:

* Use of MULTIPLE COPY LIBRARIES for both DATA and PROCEDURE divisions.

* Commonly (and not so commonly) used INTRINSICS called from COBOL II, and how they can help you.

* DECLARATIVES and I/O STATUS checking.

* PROCESS HANDLING vs DYNAMIC SUBPROGRAMS in on line menus.

Examples from real programs will be used throughout, with minor changes to protect both the innocent and the guilty.

COPY LIBRARIES

I haven't worked on a computer system that uses COBOL without some form of the COPY statement; however many make it inconvenient to implement.

HP has provided two facilities for managing copy libraries that provide the programmer with great flexibility, as well as making standardization easy to implement.

1. **MULTI-MEMBER** library files.

   On the HP3000, unlike many other systems, multiple members may reside in one copy library file. Normally, a copy library file is maintained as a KSAM file, which may be easily manipulated with the COBEDIT.PUB.SYS utility. This utility allows you to add new members, and to delete or edit existing members. Editing is handled by process handling to the EDITOR while still within the COBEDIT utility.

2. **MULTI-LIBRARY** COPY STATEMENTS in a program.

   Within one program, you may copy members from multiple library files, thus allowing you to maintain separate libraries of standardized routines used in any program, as well as libraries of members unique to a single application.

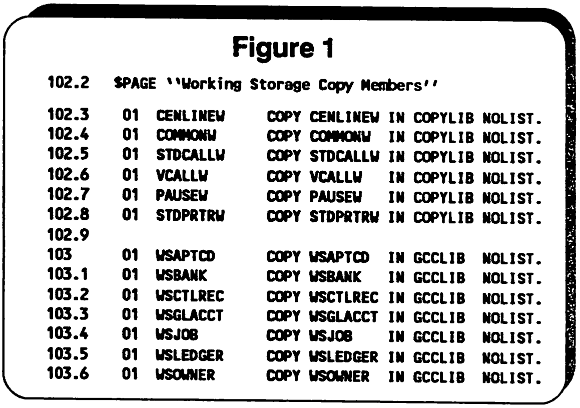


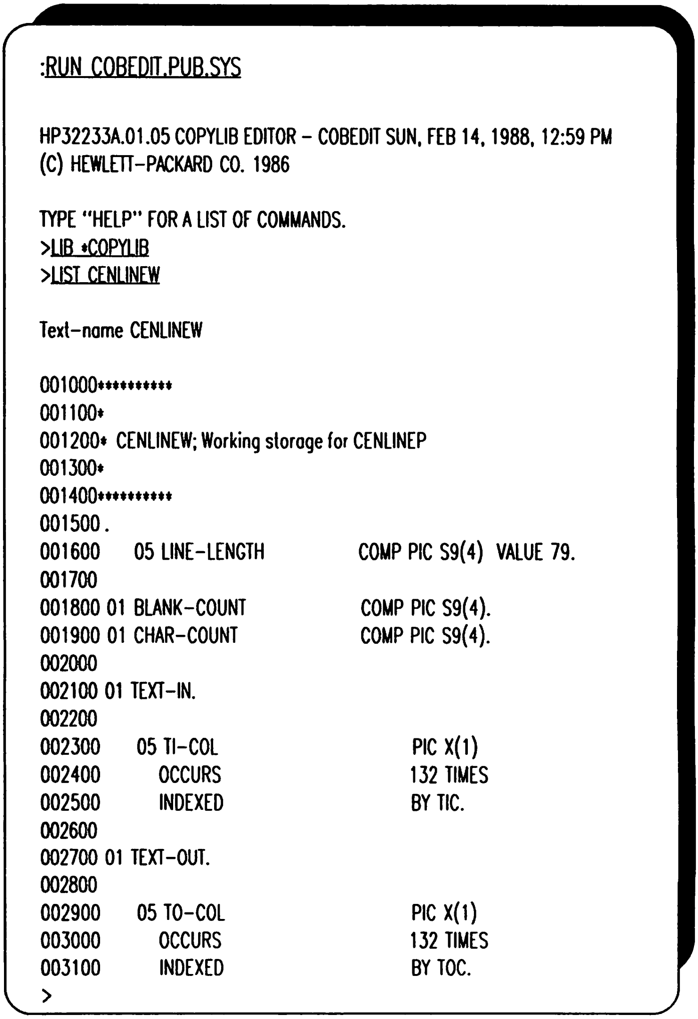
Figure 1

```
102.2   $PAGE ''Working Storage Copy Members''
102.3    01  CENLINEW      COPY CENLINEW IN COPYLIB NOLIST.
102.4    01  COMMONW       COPY COMMONW  IN COPYLIB NOLIST.
102.5    01  STDCALLW      COPY STDCALLW IN COPYLIB NOLIST.
102.6    01  VCALLW        COPY VCALLW   IN COPYLIB NOLIST.
102.7    01  PAUSEW        COPY PAUSEW   IN COPYLIB NOLIST.
102.8    01  STDPRTRW      COPY STDPRTRW IN COPYLIB NOLIST.
102.9
103      01  WSAPTCD       COPY WSAPTCD  IN GCCLIB  NOLIST.
103.1    01  WSBANK        COPY WSBANK   IN GCCLIB  NOLIST.
103.2    01  WSCTLREC      COPY WSCTLREC IN GCCLIB  NOLIST.
103.3    01  WSGLACCT      COPY WSGLACCT IN GCCLIB  NOLIST.
103.4    01  WSJOB         COPY WSJOB    IN GCCLIB  NOLIST.
103.5    01  WSLEDGER      COPY WSLEDGER IN GCCLIB  NOLIST.
103.6    01  WSOWNER       COPY WSOWNER  IN GCCLIB  NOLIST.
```

Figure 1 shows how these two facilities make it easy to combine data from multiple copy libraries into one program. Note that two libraries are specified; COPYLIB, which is the library of commonly used members, and GCCLIB, which contains record descriptions used only in the GCC applications. The NOLIST entry tells the COBOL compiler not to list the data being inserted into the program at compile time.

COPY LIBRARIES

The first copy library member listed in Figure 1 is CENLINEW, the commonly used working storage for centering text in any line up to 132 characters long. Using the COBEDIT utility, it's easy to list the data from a copy member to the terminal.

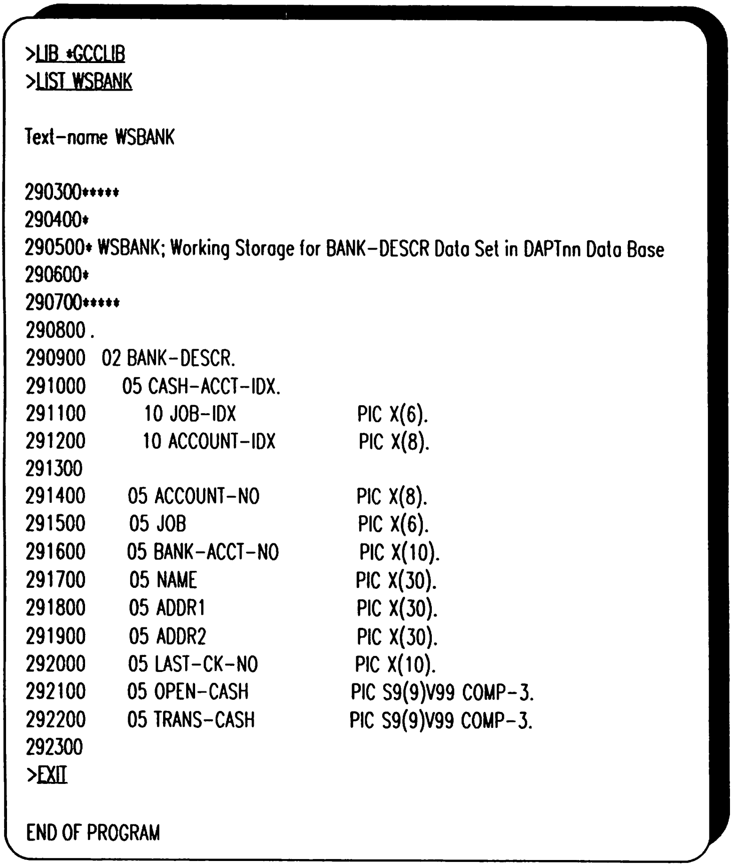


```
:RUN COBEDIT.PUB.SYS

HP32233A.01.05 COPYLIB EDITOR - COBEDIT SUN, FEB 14, 1988, 12:59 PM
(C) HEWLETT-PACKARD CO. 1986

TYPE "HELP" FOR A LIST OF COMMANDS.
>LIB *COPYLIB
>LIST CENLINEW

Text-name CENLINEW

001000**********
001100*
001200* CENLINEW; Working storage for CENLINEP
001300*
001400**********
001500.
001600     05 LINE-LENGTH          COMP PIC S9(4)  VALUE 79.
001700
001800 01 BLANK-COUNT              COMP PIC S9(4).
001900 01 CHAR-COUNT               COMP PIC S9(4).
002000
002100 01 TEXT-IN.
002200
002300     05 TI-COL                     PIC X(1)
002400        OCCURS                     132 TIMES
002500        INDEXED                    BY TIC.
002600
002700 01 TEXT-OUT.
002800
002900     05 TO-COL                     PIC X(1)
003000        OCCURS                     132 TIMES
003100        INDEXED                    BY TOC.
>
```

COPY LIBRARIES

The COBEDIT utility allows you to switch between libraries; the following shows a listing of a member of GCCLIB, which contains application specific record descriptions.

NOTE that COBEDIT allows back-referenced file names for selecting the current library.

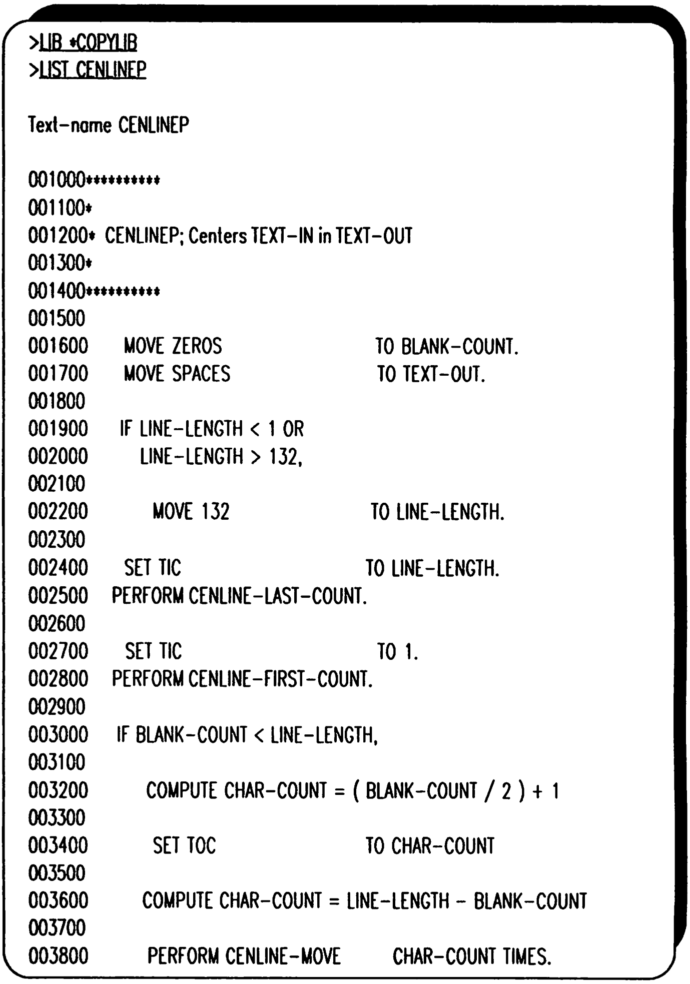


```
>LIB *GCCLIB
>LIST WSBANK

Text-name WSBANK

290300*****
290400*
290500* WSBANK; Working Storage for BANK-DESCR Data Set in DAPTnn Data Base
290600*
290700*****
290800.
290900  02 BANK-DESCR.
291000     05 CASH-ACCT-IDX.
291100        10 JOB-IDX               PIC X(6).
291200        10 ACCOUNT-IDX           PIC X(8).
291300
291400     05 ACCOUNT-NO               PIC X(8).
291500     05 JOB                      PIC X(6).
291600     05 BANK-ACCT-NO             PIC X(10).
291700     05 NAME                     PIC X(30).
291800     05 ADDR1                    PIC X(30).
291900     05 ADDR2                    PIC X(30).
292000     05 LAST-CK-NO               PIC X(10).
292100     05 OPEN-CASH                PIC S9(9)V99 COMP-3.
292200     05 TRANS-CASH               PIC S9(9)V99 COMP-3.
292300
>EXIT

END OF PROGRAM
```

Common routines may also be stored in copy libraries; once tested, they may be used easily by all members of the staff without worrying about re-inventing the wheel, and with assurance that they are not contributing towards bugs discovered during testing.

COPY LIBRARIES

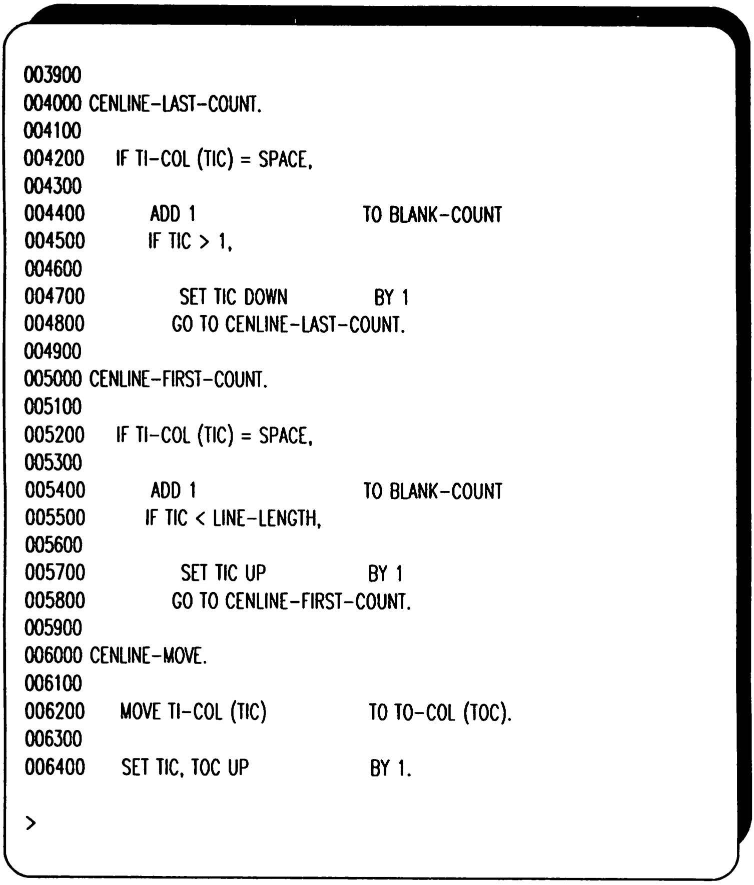
A common routine CENLINEP is stored in COPYLIB; many programs in different applications have occasional need to center text.



```
>LIB *COPYLIB
>LIST CENLINEP

Text-name CENLINEP

001000***********
001100*
001200* CENLINEP; Centers TEXT-IN in TEXT-OUT
001300*
001400***********
001500
001600     MOVE ZEROS                  TO BLANK-COUNT.
001700     MOVE SPACES                 TO TEXT-OUT.
001800
001900     IF LINE-LENGTH < 1 OR
002000        LINE-LENGTH > 132,
002100
002200         MOVE 132                TO LINE-LENGTH.
002300
002400     SET TIC                     TO LINE-LENGTH.
002500    PERFORM CENLINE-LAST-COUNT.
002600
002700     SET TIC                     TO 1.
002800    PERFORM CENLINE-FIRST-COUNT.
002900
003000    IF BLANK-COUNT < LINE-LENGTH,
003100
003200        COMPUTE CHAR-COUNT = ( BLANK-COUNT / 2 ) + 1
003300
003400         SET TOC                 TO CHAR-COUNT
003500
003600        COMPUTE CHAR-COUNT = LINE-LENGTH - BLANK-COUNT
003700
003800        PERFORM CENLINE-MOVE      CHAR-COUNT TIMES.
```

Continued on Page 7 . . . .

COPY LIBRARIES



```
003900
004000 CENLINE-LAST-COUNT.
004100
004200    IF TI-COL (TIC) = SPACE,
004300
004400       ADD 1                   TO BLANK-COUNT
004500       IF TIC > 1,
004600
004700           SET TIC DOWN        BY 1
004800          GO TO CENLINE-LAST-COUNT.
004900
005000 CENLINE-FIRST-COUNT.
005100
005200    IF TI-COL (TIC) = SPACE,
005300
005400       ADD 1                   TO BLANK-COUNT
005500       IF TIC < LINE-LENGTH,
005600
005700           SET TIC UP          BY 1
005800          GO TO CENLINE-FIRST-COUNT.
005900
006000 CENLINE-MOVE.
006100
006200    MOVE TI-COL (TIC)          TO TO-COL (TOC).
006300
006400    SET TIC, TOC UP            BY 1.

>
```

The ability to have multiple libraries accessed within one COBOL program makes the use of common routines a 'common' occurrence in shops that rely on standardized techniques to develop programs quickly.

COPY LIBRARIES

Combining the copy members CENLINEW and CENLINEP from COPYLIB, and WSBANK from GCCLIB, we're able to use coding techniques like the following:

```
MOVE NAME IN WSBANK          TO TEXT-IN.
MOVE 30                      TO LINE-LENGTH.

PERFORM CENLINEP.

MOVE TEXT-OUT                TO HEADING-BANK-NAME.
```

NOTE that the use of the copy member name as a paragraph name (CENLINEP) is acceptable; the entry in COPYLIB actually has no paragraph name.

Similarly, by beginning the working storage copy members with a period (.), and having the '01' level be prior to the COPY statement, allows reference to the group item by its copy member name (WSBANK), as well as by the '02' level that corresponds to the data set name (BANK-DESCR).

The NOLIST convention for copy members is common in shops that make heavy use of copy libraries; typically, each programmer has a listing of the common library (COPYLIB) at his/her desk, as well as listings of those application dependent libraries (such as GCCLIB) that are frequently referenced. This makes compiled listings shorter, and for programmers experienced with the shop's conventions, easier to work with.

DECLARATIVES and I/O STATUS

You've seen it, the infamous TOMBSTONE printed by the file system when a COBOL program attempts an I/O operation that is unsuccessful, and for which there wasn't an appropriate error handling routine established.

Many programs check for AT END and INVALID KEY conditions, but are at a total loss if an OPEN fails, or if the INVALID KEY condition doesn't allow the program to adequately diagnose the problem, thereby preventing 'elegant' error handling.

Two features of COBOL II (and COBOL 85) provide the means to trap I/O errors and take the appropriate action based upon the actual condition that occurred.

DECLARATIVES.

This Section of the program, which must be the first Section within the Procedure Division, defines procedures to be used when the file system discovers an error or unusual condition.

FILE STATUS.

This entry in the SELECT filename clause defines a storage location in which the status of the most recent I/O operation for a file is returned.

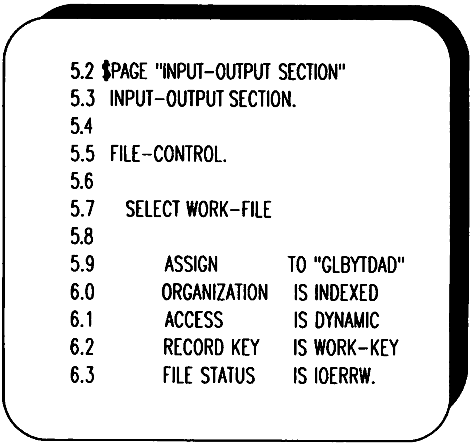
The two, working in combination, give the programmer complete control over error and exceptional condition processing for a file.

To see how these work together, we'll begin with some sample program code, beginning on Page 10 with a file select clause using the FILE STATUS option.

DECLARATIVES and I/O STATUS

The FILE STATUS item must be selected if you want to trap I/O errors and be able to determine the cause of the I/O failure. When an input or output operation has been performed on the file, the status item is updated with a two character code indicating the status of the operation.

If the first byte contains 0 (ZERO), the operation was basically successful. The second byte contains additional information further defining the status.



```
$PAGE "INPUT-OUTPUT SECTION"
 INPUT-OUTPUT SECTION.

 FILE-CONTROL.

   SELECT WORK-FILE

        ASSIGN        TO "GLBYTDAD"
        ORGANIZATION  IS INDEXED
        ACCESS        IS DYNAMIC
        RECORD KEY    IS WORK-KEY
        FILE STATUS   IS IOERRW.
```

The example above shows a FILE STATUS item of IOERRW. This is a two byte field defined as a commonly used member of COPYLIB.

Page 11 shows this copy member, which also includes an additional field used for interpreting the second byte of the status returned for I/O operations.

DECLARATIVES and I/O STATUS

Meanings of the first byte (IO-ERR-1) are:

- * 0 Successful completion
- * 1 At end, EOF has been reached
- * 2 Invalid key, duplicate for writes, or not found for reads
- * 3 Physical I/O error, or beyond EOF

```
**********
*
* IOERRW: Working Storage for FILE STATUS
*
**********

 .
    02 IOERRW-CHARS     PIC X(2) VALUE ``00''.

     88 IO-SUCCESSFUL             VALUE ``00''.
     88 IO-ALLOWED-DUPL           VALUE ``02''.
     88 IO-EOF                    VALUE ``10''.
     88 IO-SEQUENCE-ERR           VALUE ``21''.
     88 IO-DUPL-KEY               VALUE ``22''.
     88 IO-NOT-FOUND              VALUE ``23''.
     88 IO-BOUND-VIOL             VALUE ``24'',
                                        ``34''.
     88 IO-PERM-ERROR             VALUE ``30''.

    02  FILLER
        REDEFINES   IOERRW-CHARS.

     05 IO-ERR-1         PIC X(1).

         88 IO-OK               VALUE ``0''.
         88 IO-AT-END           VALUE ``1''.
         88 IO-INVALID-KEY      VALUE ``2''.
         88 IO-PERMANENT-ERROR  VALUE ``3''.
         88 IO-MISC-ERROR       VALUE ``9''.

     05 IO-ERR-2        PIC X(1).
```

```
**********
*
* Used to convert the second byte of FILE
* STATUS (IOERRW) to a valid MPE file system
* error code (FSERR)
*
**********



 01 IOERR-CONVERT.

     05 IOERR-DUMMY      PIC X(1) VALUE
                                   LOW-VALUES.

     05 IOERR-CHARACTER    PIC X(1).

 01 IOERR-MPE-ERR-NUM
    REDEFINES  IOERR-CONVERT
                    COMP  PIC S9(4).

         88 IO-UNOBTAINABLE     VALUE 90,
                                      91.
         88 IO-FILE-NOT-FOUND   VALUE 52,
                                      53.
         88 IO-DEV-UNAVAILABLE  VALUE 55.
         88 IO-DUPLICATE-FILE   VALUE 100,
                                      101.
```

WARNING - these definitions are for COBOL II (1974); there are changes in COBOL 85

DECLARATIVES and I/O STATUS

The other feature that helps us handle I/O errors is a fairly simple routine inserted in the first part of the program; it works for main programs and subprograms.

The program must have a DECLARATIVES Section, which must be the first section in the program. NOTE that the use of a section for Declaratives requires a section name for the first paragraph of the normal procedure division, even if the program is not to be sectioned to create additional code segments.

A sample DECLARATIVES follows.

```
                         $PAGE ''Procedure Division - Section 0''
 A dynamic subprogram     PROCEDURE DIVISION                   USING STDCALLW,
 being called with five                                              DBCALLW,
        parameters...                                                VCALLW,
     all copy members                                                WSAPTCD,
                                                                     STDPRTRW.


Required statement to     DECLARATIVES.
   begin DECLARATIVES
SECTION NAME required     GLBYTDA-START                          SECTION 00.

        USE statement         USE AFTER ERROR PROCEDURE        ON WORK-FILE.

Followed by paragraph     GLBYTDA-IO-ERROR.
   name for procedure
                              IF IO-MISC-ERROR,

  Convert 2nd Byte of             MOVE IO-ERR-2                TO IOERR-CHARACTER
 IOERRW to FSERR num.
                                  MOVE SPACES                  TO STD-CALL-RESULT-MSG

   Get FSERR message              CALL INTRINSIC ''FERRMSG''     USING IOERR-MPE-ERR-NUM,
                                                                     STD-CALL-RESULT-MSG,
                                                                     STD-CALL-CONDTN-WORD

   For return to caller           MOVE IOERR-MPE-ERR-NUM       TO STD-CALL-CONDTN-WORD.

   Required statement     END DECLARATIVES.

 Begin normal program     GLBYTDA-BEGIN                        SECTION 00.
    with section name
                              PERFORM HOUSEKEEPING.
```

DECLARATIVES and I/O STATUS

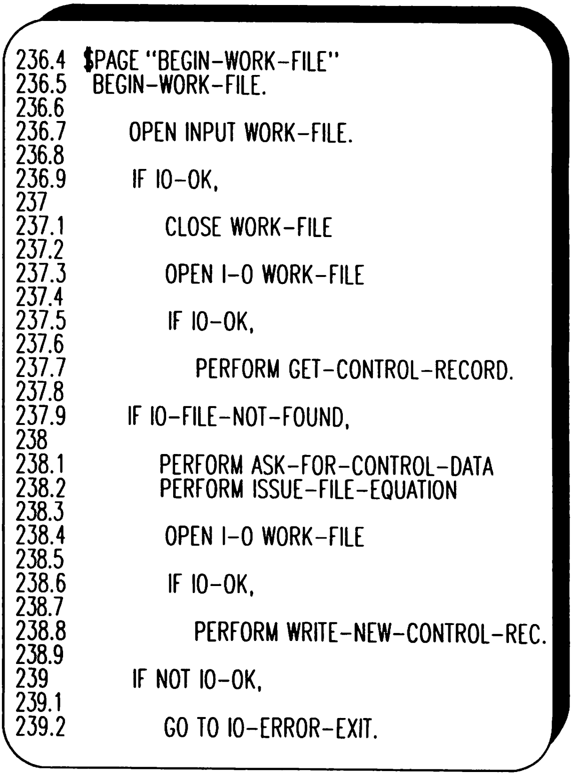
The documentation for COBOL provides the rules for precedence for FILE STATUS items and USE PROCEDURES (with a flow chart further explaining this in KPR# 4700245142 in the System Staus Bulletin); it really boils down to a simple statement ....*'If you use a FILE STATUS item, and have a USE PROCEDURE, your destiny is in your own hands'.*

The FILE STATUS item is updated for all of your I/O, and the USE PROCEDURE is executed for every exceptional condition. This allows the following type programming:

The file is first opened for input; just to see if it's there. If so, the control record is retrieved. If it doesn't exist, the user is asked for parameters for building a new file, and for data to be stored in the control record.

The change in processing based upon IO-FILE-NOT-FOUND (FSERR 52) is easy to handle; all other I/O errors are unexpected and cause an exit to the error handling routine.

The USE Procedure is invoked for all 'NOT IO-OK' situations, and the FILE STATUS item is set after each I/O.



```
$PAGE "BEGIN-WORK-FILE"
 BEGIN-WORK-FILE.

     OPEN INPUT WORK-FILE.

     IF IO-OK,

         CLOSE WORK-FILE

         OPEN I-O WORK-FILE

         IF IO-OK,

             PERFORM GET-CONTROL-RECORD.

     IF IO-FILE-NOT-FOUND,

         PERFORM ASK-FOR-CONTROL-DATA
         PERFORM ISSUE-FILE-EQUATION

         OPEN I-O WORK-FILE

         IF IO-OK,

             PERFORM WRITE-NEW-CONTROL-REC.

     IF NOT IO-OK,

         GO TO IO-ERROR-EXIT.
```

If the FILE STATUS contains a 9 in the first byte (IO-ERR-1), the second byte is moved to IOERR-CHARACTER, which is used, via the redefinition of IOERR-CONVERT, to call INTRINSIC "FERRMSG" to obtain an interpretation of the error condition to place in a message passed back to the caller.

DECLARATIVES and I/O STATUS

In the example shown on Page 13 only the IO-FILE-NOT-FOUND condition was specifically anticipated. But note how easy it would be to attempt the open, then if the condition IO-UNOBTAINABLE (Exclusive Violation) was found, the program could 'elegantly' let the user know that the file was in use by someone else, and request that a later retry would be appropriate.

The COBOL manual recommends calling "CKERROR" to convert the second byte of FILE STATUS to an ASCII number, however the simple move to a redefined integer (COMP) accomplishes the same thing, and that number is in the correct format for calls to "FERRMSG".

Of course, there are many other ways to accomplish the same logic that this little routine uses; it only points out one set of circumstances that make use of these techniques. Once mastered, and with key elements readily available in a COPYLIB, you'll find its flexibility to be helpful in complex applications.

Obtaining actual file system error codes for those conditions that do not begin with a 9 in the first byte is also possible. The intrinsic FCHECK applies to files on any device, and can be used simply. For example:

```
          CALL INTRINSIC "FCHECK"  USING WORK-FILE,
                                         IOERR-MPE-ERR-NUM,
                                          \\, \\, \\.
```

This returns the file system error code for the last I/O for WORK-FILE into IOERR-MPE-ERR-NUM.

NOTE that the intrinsics manual asks for *filenum* for file intrinsics; COBOL programmers may substitute *filename*, as defined in a SELECT statement. The backslashes in the call above stand for 'null' parameters; optional parameters not required for a simple call just for the file system error number.

To easily change programs from COBOL 74 to COBOL 85, a new copy member can be created that contains the FILE STATUS error codes used with the newer compiler (and run time processing). The table on Page 15 provides a brief overview of the differences.

DECLARATIVES and I/O STATUS

COMPARISON of ANSI 85 vs ANSI 74 I-O STATUS CODES

ANSI85	ANSI74	MEANING
04	00	Read length of record doesn't match file.
05	00	Optional file not present; created.
07	00	File NOT a TAPE file as OPEN/CLOSE implies.
14	00	Relative record number larger than PICTURE of key descriptor.
24	24	Write beyond file boundary, or relative record number larger than PICTURE of key descriptor.
35	9x	Non-optional file not present; not created.
37	00	Open mode invalid for file type.
38	00	Attempted OPEN on file closed with lock.
39	00	Attribute conflict; file not opened.
41	9x	Attempted OPEN on file that is open.
42	9x	Attempted CLOSE on file that is not open.
43	9x/00	Attempted DELETE/REWRITE without prior READ.
44	00	Boundary violation or invalid record size.
46	10	Attempted READ after EOF or previously unsuccessful read.
47	9x/00	Attempted READ on file not open for input.
48	9x/00	Attempted WRITE on file not open for output (or I-O).
49	9x/00	Attempted REWRITE/DELETE on file not open for I-O.

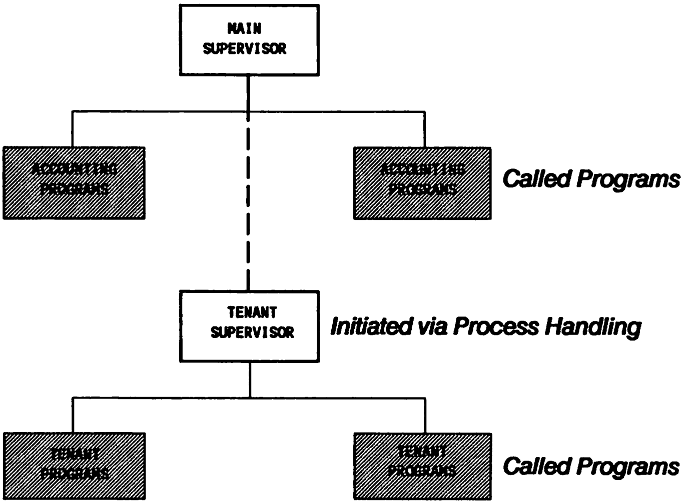
Creating a new copy library member incorporating the revisions to I-O Status makes upgrading to COBOL 85 an easier task.

SOME ADDITIONAL INTRINSICS

To introduce the use of some intrinsics that are fairly easy to use from COBOL II, I'll describe a situation that we ran into, and the solution that we used. Of course, there are always multiple solutions to any design/programming problem, but this will be fairly illustrative of the power available to COBOL programmers.

A system, written in COBOL 68, was being converted to COBOL 74. To those of you never exposed to the wonders of COBOL 68, it had no facility for calling HP intrinsics directly; any needed intrinsic calls were written in SPL. First the COBOL program called the SPL intrinsic handling routine, which called the instrinsic, then returned to the COBOL program. It was a little cumbersome, but it worked. However, it did discourage COBOL shops from heavy use of intrinsics. Naturally, as part of the conversion process, direct calls to intrinsics were substituted for the calls to SPL intrinsic handling routines.

The system being modified was structured with a MAIN SUPERVISOR, which called dynamic subprograms for various required functions. Due to the system table limits at the time the system was developed (CST entries, maximum code segments per process), it had to also initiate a second level supervisor for some functions, using process handling to accomplish this.



NOTE that parameters were passed between processes by entries in a data base.

SOME ADDITIONAL INTRINSICS

Our charter wasn't to totally redesign the system, but we were asked if we could speed the movement between the main supervisor and the secondary supervisor. The basic logic in use was:

MAIN SUPERVISOR

Post parameters to control data base
Close terminal
Call SPL routine to CREATE and ACTIVATE process

{ *Wait for Tenant Supervisor* }

Retrieve parameters from control data base
Open terminal
If parameters indicate a different production data base was opened by tenant supervisor, close initial data base and open data base that had been opened by tenant supervisor.

TENANT SUPERVISOR

Open control data base, and retrieve parameters
Open production data base
Open forms file
Open terminal

{ *Additional Processing* }

Post current data base name and other parameters to control data base, then close it.
Close production data base
Close terminal
Close forms file
STOP RUN

SOME ADDITIONAL INTRINSICS

After some analysis of the frequency of use of the Tenant Supervisor (how often it was initiated from within the Main Supervisor by a user in one session), we elected the following two concepts for reducing the transition time:

1. Eliminate the control data base as a means of passing parameters. Several options came to mind; we chose an extra data segment as a means of passing parameters between processes. They provide a fast means of sharing data between processes, and implementation in the system would be easy, utilizing the existing data structure (the control record layout) for parameter storage.

   This eliminated the data base open, GET/PUT I/O, and data base close in both the Main Supervisor as well as the Tenant Supervisor.

2. Because the frequency of use seemed to justify it, we elected to not terminate the Tenant Supervisor (STOP RUN) when its current processing was complete, but to retain it as a process that could be re-activated when next needed.

   This eliminated the repeated overhead of program loading, forms file open, and data base open (so long as the data base requested in the passed parameters was the same as the previously requested data base).

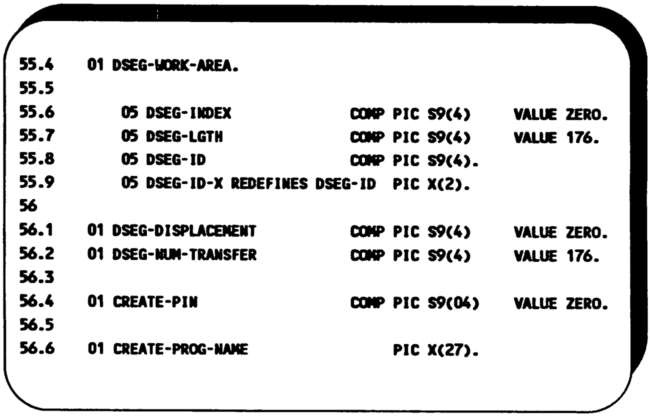
The results were quite acceptable to the users.; the transition time was reduced. On the first initiation of the Tenant Supervisor during any one session, there was still a noticeable delay, but not as long as previously.

The second, and subsequent initiations of the Tenant Supervisor were at a speed that gave no indication that another program was being started. The following intrinsics were used to accomplish this from within COBOL programs:

GETDSEG
DMOVOUT
CREATE
ACTIVATE
DEMOVIN
KILL
FREEDSEG

SOME ADDITIONAL INTRINSICS

To illustrate how these may be used from within COBOL programs, we'll start with the WORKING STORAGE used in the Main Supervisor.



```
                             01 DSEG-WORK-AREA.

Assigned by MPE (GETDSEG)        05 DSEG-INDEX              COMP PIC S9(4)    VALUE ZERO.
DSEG length (in words)           05 DSEG-LGTH               COMP PIC S9(4)    VALUE 176.
Program assigned name            05 DSEG-ID                 COMP PIC S9(4).
                                 05 DSEG-ID-X REDEFINES DSEG-ID  PIC X(2).

Starting Location            01 DSEG-DISPLACEMENT           COMP PIC S9(4)    VALUE ZERO.
Words to DMOVIN/DMOVOUT      01 DSEG-NUM-TRANSFER           COMP PIC S9(4)    VALUE 176.

PIN for created process      01 CREATE-PIN                  COMP PIC S9(04)   VALUE ZERO.

Program to be initiated      01 CREATE-PROG-NAME                 PIC X(27).
```

Extra Data Segments are an additional segment of memory that a program is allowed to use for storage of data. They may be easily shared by multiple processes within the same process tree (father process and its sons). One advantage they have is that transfer of data is at memory to memory speeds; there is no disc I/O associated with their use, other than any required by MPE's memory manager.

The GETDSEG intrinsic is used to create a new Extra Data Segment, or to gain access to one that has been previously created. The ID is the name by which your program attempts to initially perform the GETDSEG, for the stated LENGTH.

The INDEX is a unique number assigned by MPE; once acquired, the index is used to obtain access to the data in the DSEG using the DMOVIN intrinsic (move data from extra data segment to your program's working storage) and the DMOVOUT intrinsic (move data from working storage to the extra data segment).

DISPLACEMENT is like a subscript or index, telling the DMOVIN and DMOVOUT intrinsics where in the Extra Data Segment to begin the move of data, using 0 (ZERO) as the first word. The size of the data string to be moved is stated in number of words.

SOME ADDITIONAL INTRINSICS

The first change to the Main Supervisor was to add a call to GETDSEG, done only once in the intialization logic. This establishes the data segment, and assigns a unique index number to it. This is the identifier by which it will be recognized by any program that is a member of this process.

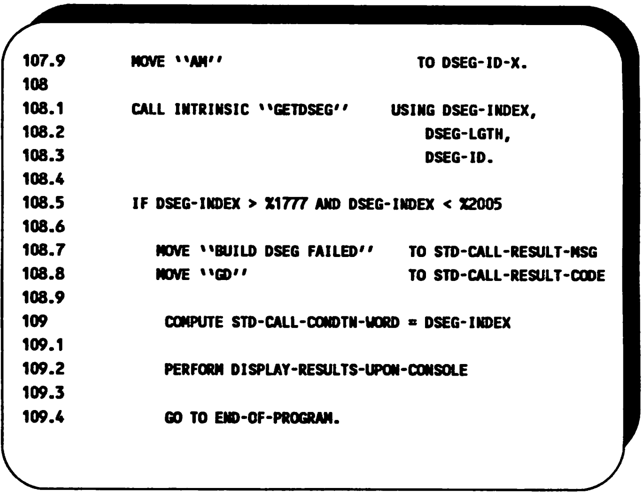
The ID is the name by which other processes sharing this extra data segment will first obtain access to it.

The size, in number of words, is the size of the record used in the original data base parameter passing routine.

Assign an ID to DSEG

Returned by MPE
Length desired
ID established above

Check for error conditions. If found, establish diagnostics and exit to common error display routine used by Main Supervisor for all "catastrophic" errors found in Main or subprograms.



```
107.9     MOVE ''AM''                       TO DSEG-ID-X.
108
108.1     CALL INTRINSIC ''GETDSEG''     USING DSEG-INDEX,
108.2                                          DSEG-LGTH,
108.3                                          DSEG-ID.
108.4
108.5     IF DSEG-INDEX > %1777 AND DSEG-INDEX < %2005
108.6
108.7        MOVE ''BUILD DSEG FAILED''    TO STD-CALL-RESULT-MSG
108.8        MOVE ''GD''                   TO STD-CALL-RESULT-CODE
108.9
109           COMPUTE STD-CALL-CONDTN-WORD = DSEG-INDEX
109.1
109.2         PERFORM DISPLAY-RESULTS-UPON-CONSOLE
109.3
109.4         GO TO END-OF-PROGRAM.
```

Once established, the Extra Data Segment may be used repeatedly. There is no significant time used in acquiring an Extra Data Segment; it is significantly less than the time used to open a data base.

The error conditions for which the test is done are items such as invalid length, you've attempted to exceed the maximum configured XDSEGS, etc.

SOME ADDITIONAL INTRINSICS

The end of program routine was modified to include a call to FREEDSEG; this releases the the data segment from the session. Perhaps not strictly required in this application; experience has shown that good housekeeping pays off.

The same is true for the call to the KILL intrinsic. This deletes the son process; that is the Tenant Supervisor, if it had been intitiated.

If Tenant Supervisor alive, kill it

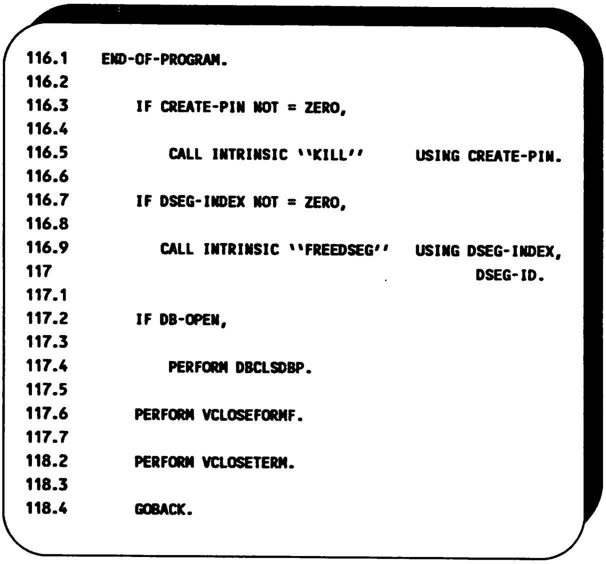
If XDSEG was created, free it

If production data base open, close it

Close the forms file

Close the terminal

Exit this program



```
 END-OF-PROGRAM.

     IF CREATE-PIN NOT = ZERO,

        CALL INTRINSIC ''KILL''      USING CREATE-PIN.

     IF DSEG-INDEX NOT = ZERO,

       CALL INTRINSIC ''FREEDSEG''  USING DSEG-INDEX,
                                          DSEG-ID.

     IF DB-OPEN,

        PERFORM DBCLSDBP.

     PERFORM VCLOSEFORMF.

     PERFORM VCLOSETERM.

     GOBACK.
```

Earlier, the use of copy members for commonly used functions was discussed. This routine includes performs of three commonly used copy members:

DBCLSDBP; closes the currently open data base
VCLOSEFORMF; closes the currently open VPLUS forms file
VCLOSETERM; closes the currently open terminal file used by VPLUS

SOME ADDITIONAL INTRINSICS

The routines to create and/or activate the Tenant Supervisor and pass parameters become easy after the preliminary work has been done.

Move parameters to XDSEG

Create/Activate another process; then wait for it to return

Move changed parameters back in

Use the INDEX assigned by GETDSEG

Starting location in target DSEG

Number of words to transfer

Source of data to be moved out

Check for errors, and exit if any found

```
439.3   INITIATE-TENANT-SUPERVISOR.
440
440.1       PERFORM MOVEOUT-DSEG.
440.2
440.3       PERFORM CREATE-AND-ACTIVATE-TSUPVOAX.
440.4
440.5       PERFORM MOVEIN-DSEG.
440.6
440.9   * Continue processing
441.6
441.7   MOVEOUT-DSEG.
441.8
442.7*      Set up parameters here
443.3
443.4       CALL INTRINSIC ''DMOVOUT''  USING DSEG-INDEX,
443.5                                     DSEG-DISPLACEMENT,
443.6                                     DSEG-NUM-TRANSFER,
443.7                                     WSCTLREC.
443.8
443.9       IF C-C NOT = ZERO,
444
444.1         MOVE ''DM''          TO STD-CALL-RESULT-CODE
444.2         MOVE ''DMOVOUT FAIL"  TO STD-CALL-RESULT-MSG
444.3
444.4           PERFORM DISPLAY-RESULTS-UPON-CONSOLE
444.5
444.6           GO TO END-OF-PROGRAM.
```

NOTE that this process is suspended after the Tenant Supervisor is initiated (Line 440.3), so the next instruction will be executed as soon as control is returned to this process. The use of the Extra Data Segment is barely more difficult than a "CALL USING" when dealing with a subprogram.

Continued on Page 23 . . .

SOME ADDITIONAL INTRINSICS

Creating and/or activating a process is not difficult. NOTE that the %101 parameter (flags, as defined in the Intrinsics Manual) tells MPE that the created process should use the NOCB parameter; it has stack size problems and needs the space this frees.

Set up program name

Only if not previously created, will the process be created

NO entry point name

PIN number returned

No PARM= passed

NOCB & reactivate father when new process terminates

If errors found, exit to common diagnostic routine

```
CREATE-AND-ACTIVATE-TSUPVOAX.

    MOVE ''TSUPVOAX.group.acct'' TO CREATE-PROG-NAME.

    IF CREATE-PIN = ZERO,

        CALL INTRINSIC ''CREATE'' USING CREATE-PROG-NAME,
                                        \\,
                                        CREATE-PIN,
                                        \\,
                                        %101.


    IF CREATE-PIN < 1 OR > 1024,

        MOVE ''CREATE FAILED''        TO STD-CALL-RESULT-MSG
        MOVE CREATE-PIN               TO STD-CALL-CONDTN-WORD
        MOVE ''CR''                   TO STD-CALL-RESULT-CODE

         PERFORM DISPLAY-RESULTS-UPON-CONSOLE

        GO TO END-OF-PROGRAM.

    CALL INTRINSIC ''ACTIVATE''          USING CREATE-PIN,
                                               2.

    IF C-C < ZERO,

        MOVE ''ACTIVATE FAILED''      TO STD-CALL-RESULT-MSG
        MOVE CREATE-PIN               TO STD-CALL-CONDTN-WORD
        MOVE ''CR''                   TO STD-CALL-RESULT-CODE

        PERFORM DISPLAY-RESULTS-UPON-CONSOLE

        GO TO END-OF-PROGRAM.
```

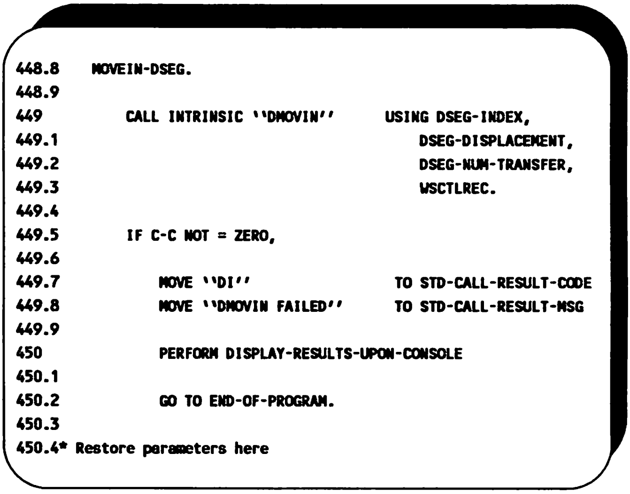
Activate the Tenant Supervisor, expecting to be activated by it

If errors found, exit

Continued on Page 24 . . .

SOME ADDITIONAL INTRINSICS

When theTenant Supervisor returns control to the Main Supervisor, the passed, and maybe changed, parameters are restored using DMOVIN. Its operation is just the reverse of the DMOVOUT intrinsic; it moves data from the Extra data Segment into the program's Working Storage.



Use INDEX from GETDSEG
Starting location in DSEG
Number of words to move
Target in working storage

If errors found, exit

```
MOVEIN-DSEG.

     CALL INTRINSIC ''DMOVIN''     USING DSEG-INDEX,
                                         DSEG-DISPLACEMENT,
                                         DSEG-NUM-TRANSFER,
                                         WSCTLREC.

     IF C-C NOT = ZERO,

         MOVE ''DI''                TO STD-CALL-RESULT-CODE
         MOVE ''DMOVIN FAILED''     TO STD-CALL-RESULT-MSG

         PERFORM DISPLAY-RESULTS-UPON-CONSOLE

         GO TO END-OF-PROGRAM.

* Restore parameters here
```

The Main Supervisor code to replace control data base open, gets, and puts was easily replaced by the GETDSEG, DMOVOUT, DMOVIN, and FREEDSEG intrinsic calls.

But what about the Tenant Supervisor ?
What changes did it require for an Extra Data Segment?
And how could we eliminate its startup overhead?

SOME ADDITIONAL INTRINSICS

The initiated program needs to do some of the same things as the initiator. It must use GETDSEG to acquire access to the Extra Data Segment, and it uses DMOVIN and DMOVOUT to receive and return parameters in the Extra Data Segment.

However, to avoid startup overhead, it needs some slight modifications. First, it needs to have a way to suspend itself, rather than completely terminate.

This allows it to be re-activated in the same state that it was in when it suspended. That means that any files open at the time it suspended will still be open when it is re-activated.

It must, therefore, be able to recognize whether the current activation is an initial activation, or a reactivation. This is an easy task, since the DSEG-INDEX itself becomes the switch; if non-zero, then the program was just re-initiated.

```
PROCEDURE DIVISION.

 TSUPV-START.

     IF DSEG-INDEX = ZERO,
```

Check DSEG-INDEX for 'first time'; if so acquire DSEG and do other initialization tasks

```
         PERFORM HOUSEKEEPING

         CALL INTRINSIC ''GETDSEG'' USING DSEG-INDEX,
                                          DSEG-LGTH,
                                          DSEG-ID.
```

If errors found, exit

```
     IF DSEG-INDEX > %1777 AND DSEG-INDEX < %2005

         MOVE ''DS''                    TO STD-CALL-RESULT-CODE
         MOVE ''BUILD DSEG FAILED''     TO STD-CALL-RESULT-MSG
         GO TO END-OF-PROGRAM.
```

Move in parameters

```
     CALL INTRINSIC ''DMOVIN''       USING DSEG-INDEX,
                                          DSEG-DISPLACEMENT,
                                          DSEG-NUM-TRANSFER,
                                          WSCTLREC.
```

If errors found, exit

```
     IF C-C NOT = ZERO,

         MOVE ''DM''                    TO STD-CALL-RESULT-CODE
         MOVE ''DMOVIN  FAILED''        TO STD-CALL-RESULT-MSG
         GO TO END-OF-PROGRAM.
```

SOME ADDITIONAL INTRINSICS

The housekeeping routines, which are only executed on the first activation of the program, include terminal and forms file opens, as well as a data base open.

On second, and subsequent initiations, these routines are bypassed. The GETDSEG isn't extremely time consuming, but the file opens create tremendous overhead.

The normal processing can now continue as if this were a dynamically called subprogram; if the currently open data base is the correct one (based upon the passed parameters in the DSEG), the program can proceed with the next VPLUS screen to be displayed to the user.

```
HOUSEKEEPING.

     MOVE ''AM''                      TO DSEG-ID-X.
     MOVE ''N''                       TO DB-OPEN-SW.

     MOVE ''AM3000F.group.account''   TO V-FORMS-FILE-NAME.

     PERFORM VOPENFORMF.

     IF NOT V-OK,

         MOVE ''VOPENFORMF Failed''   TO STD-CALL-RESULT-MSG
         MOVE ''VF''                  TO STD-CALL-RESULT-CODE
         MOVE V-STATUS                TO STD-CALL-CONDTN-WORD
         GO TO END-OF-PROGRAM.

     MOVE 9                           TO V-TERM-CNTL.

     PERFORM VOPENTERM.

     IF NOT V-OK,

         MOVE ''VOPENTERM failed''    TO STD-CALL-RESULT-MSG
         MOVE V-STATUS                TO STD-CALL-CONDTN-WORD
         MOVE ''VT''                  TO STD-CALL-RESULT-CODE
         PERFORM VCLOSEFORMF,
         GO TO END-OF-PROGRAM.

     PERFORM OPEN-PROPER-DATA-BASE.
```

There's one last thing we have to take care of; ensuring that when the program is ready to return control to the Main Supervisor, it suspends itself rather than completely terminating. This is handled through a small change to the end of program routine.

SOME ADDITIONAL INTRINSICS

The end of program routine has checks for errors discovered by processing routines, and a call to intrinsic ACTIVATE. Calling ACTIVATE with a PIN number of 0 (ZERO) indicates to MPE that you want to activate the Father of the current process.

```
                            $PAGE ''CLOSE ROUTINES''
                             END-OF-PROGRAM.

Check for subroutine             IF ( NOT RESULTS-OK ),
                errors
                                     PERFORM DISPLAY-RESULTS-UPON-CONSOLE.

Set up return parameters         MOVE STD-CALL-RESULTS           TO TSUPV-STD-CALL-RESULTS.
                                 MOVE WSAPTCD                    TO CR-WSAPTCD.
                                 MOVE TSUPV-USER-PARMS           TO CR-USER-PARMS.

If no DSEG errors, move          IF STD-CALL-RESULT-CODE NOT = ''DS'',
them out into the DSEG
                                     CALL INTRINSIC ''DMOVOUT'' USING DSEG-INDEX,
                                                                      DSEG-DISPLACEMENT,
                                                                      DSEG-NUM-TRANSFER,
                                                                      WSCTLREC.

If error in DMOVOUT, set         IF C-C NOT = ZERO,
   up parameters for
  diagnostic display                 MOVE ''DS''                    TO STD-CALL-RESULT-CODE
                                     MOVE ''DMOVOUT FAILED''        TO STD-CALL-RESULT-MSG

                                     PERFORM DISPLAY-RESULTS-UPON-CONSOLE.

Activate Father process          CALL INTRINSIC ''ACTIVATE''        USING 0,3.

       Check for errors          IF C-C NOT = ZERO,

                                     MOVE ''AF''                    TO STD-CALL-RESULT-CODE
                                     MOVE ''ACTIVATE FATHER FAIL'' TO STD-CALL-RESULT-MSG
                                     PERFORM DISPLAY-RESULTS-UPON-CONSOLE

                                     GOBACK.

Go to start of program           GO TO TSUPV-START.
   when re-activated
```

The ACTIVATE of the Father suspends the current process; when re-activated, it continues with the next instruction, which takes it back to the start of the program.

SOME ADDITIONAL INTRINSICS

The following summarizes some of the key differences between called dynamic subprograms and created processes.

CALLED PROGRAMS	CREATED PROCESSES
Subprograms may reside in Segmented Libraries (SLs), or be prepped with the main program.	Programs are prepped as main programs.
Shared data bases and other files eliminate overhead associated with opens and closes.	Initial activation requires opening any required files. If not suspended upon completion, this is repeated for each creation/activation.
Parameter passing techniques are familiar to most programmers.	Parameter passing requires additional design work, but is relatively easy once mastered.
Programs execute serially; that is, the calling program suspends until the called program returns.	Created processes may execute serially, or may be executed in parallel with the creating process.

Each can be an effective technique when properly applied; the analyst must be familiar with multiple techniques to create applications that meet the user's requirements and effectively utilize the hardware/software environment of the HP3000.

SOME ADDITIONAL INTRINSICS

There are a few additional comments regarding these techniques:

1. The use of VALUE clauses in Working Storage of the initiated process can mislead you. If the process is re-activated, it begins processing in the state it was in at the time of its suspension. That means working Storage is NOT re-initialized for you by VALUE clauses.
2. Working with intrinsics requires that you check the MPE Condition Code. You must have an entry in the Special Names section to allow you to check this. A sample entry follows.

```
5.7     SPECIAL-NAMES.
5.8
5.9         CONDITION-CODE IS C-C.
```

3. Programs using Process Handling and Extra Data Segments must be prepped with these Special Capabilities indicated. To prep a program with PH and DS Capability, you must have these capabilities. The executable programs must reside in a group and account that has these capabilities.

As is true of so many design/programming techniques, the more you use them, the easier they become. And the more you learn, the more you find there is to learn.

Today's COBOL on the HP3000 provides many ways for the inventive analyst to achieve things that previously were reserved for 'Systems Programmers'.

SUMMARY

COPY LIBRARIES

COBOL II's Copy Library facility makes the use of common working storage and common procedure division routines easy. In addition to assisting in the development of error free programs, it enhances the speed of development.

The multiple library capability not only simplifies maintenance of libraries, but also eases the task of updating programs for new versions of compilers and operating systems. For example, changes to parameter sizes associated with the new intrinsics in the XL operating system can be easily accomodated in a new copy library, allowing programs to be compiled for either with minimum change.

FILE STATUS & DECLARATIVES.

The use of File Status items and the Declarative section give the programmer complete control of file system error handling. This, combined with the ability to call file system Intrinsics using the COBOL filename in place of the normal Intrinsics's *filenum* parameter allow for 'elegant' error handling, as well as provide access to many facilities previously considered too esoteric for COBOL programmers.

OTHER INTINSICS

Special Capabilities such as Process Handling and Extra Data segments can be easily utilized in COBOL II. A careful reading of the Intrinsics manual will open many doors for the creative analyst/programmer.

I hope these ideas have spurred your imagination. Hewlett Packard has given us a powerful tool for business programming in COBOL II. One of our tasks is to recognize the facilities available to us, and make use of them to provide quality systems to our users.