JANUARY/MARCH 1983
VOL. 6, NO. 1

JOURNAL

OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

CONTENTS

PUBLICATIONS COMMITTEE MEMBERS

John M. Knapp
Publisher

Art Production:
John Bird, Jennifer Case

HP 3000 International Users Group
289 S. San Antonio Road, Suite 305
Los Altos, California 94022 USA
415/941-9960

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HP 3000
INTERNATIONAL USERS GROUP
BOARD OF DIRECTORS

JOURNAL
OF THE HP 3000 INTERNATIONAL
USERS GROUP, INCORPORATED

Editor's Note -

This issue of your *Journal* is devoted to Programming Languages and Applications. These articles are representative of work, thought and results in the field. They offer a broad look at the issues and provide some new ideas.

As always, your comments are welcome.

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

Using COBOL, VIEW and IMAGE A Practical Structured Interface for the Programmer

Peter Somers
Cape Data, Inc.
Ocean Drive
Cape May, New Jersey 08204

Introduction

VIEW or V/3000, Hewlett-Packard's screen handler offers a convenient and versatile method of data collection. To fully utilize the capabilities of VIEW requires the application programmer to go beyond the routines available using the ENTRY program. Ideally the data entry routine will include complete editing including IMAGE data base checking and comprehensive error messages.The routine should allow the programmer to quickly "plug in" new applications and easily perform maintenance. Additionally the program will provide utility routines for data confirmation, screen refreshing, paging, etc.

At our shop, Cape Data, we developed a general purpose VIEW and IMAGE interface program. This program written in structured COBOL allows new applications to go up, with custom editing, in a fraction of the time previously required. The following discussion will cover this interface routine and its application. I will assume that the user has basic knowledge of both VIEW and COBOL.

Screen Design

When designing your VIEW input screens using FORMSPEC, the following techniques will help you get the most out of VIEW.

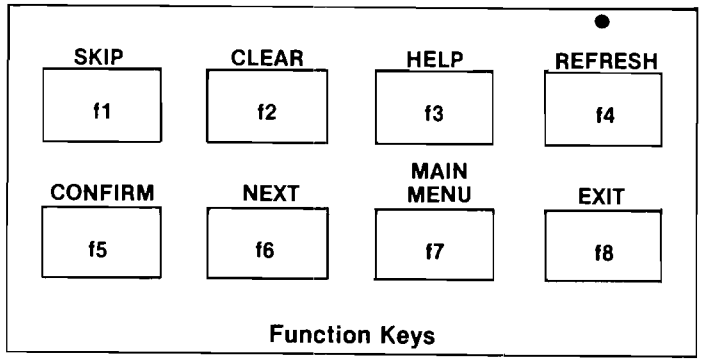
A. *Error Message Fields:* Add error message fields during form design wherever needed. Place the error message field next to or under the corresponding data field. The error message fields will remain invisible unless your program writes a message to the field. Create the field with an enhancement of B (blink) and a field type of D (display) and an initial value of spaces (Fig. 1, Field 3). The last 24th line of the screen is reserved for program error messages.

B. *VIEW Editing:* As a general rule let VIEW do as much editing as possible. On numeric fields let VIEW zero fill and test for numeric input during the FIELD portion of VIEW's editing. On alpha-numeric input fields allow VIEW to left justify the data and optionally upshift lower case characters (Fig. 1, Field 2).

C. *Title:* We reserve a Title area on all forms using Field #1. The title field is initialized by VIEW to a save field value. The VIEW forms file can contain the title and any other constants in SAVE FIELDS.

Function Keys

When using a formatted screen program with a Hewlett-Packard 2640-2645 type terminal, a special set of function keys are used by the programs. The 8 function keys are located on the upper right hand side of the terminal's keyboard. They are labeled with blue letters f1 through f8 in 2 rows. A blank Hewlett-Packard template labels the function keys (#7120-5525). On the 2620 family of terminals, the keys are labeled programmatically.



Function Keys

The function keys are used as follows in the interface program: f1 SKIP - This key will cause the cursor to skip to the next block of data. This is useful if you have a number of fields to skip. The tab key only skips a field at a time where the SKIP key will skip to the next block of data.

f2 CLEAR (RESET) - This key causes the screen to clear all fields and set the initial field values (usually blanks). This key is useful if you have created a mess on the screen and want to start over again.

f3 HELP - This key will cause the program to display an instruction screen. A special set of instructions can be displayed relating to the particular form on the screen when the HELP key was pushed. When you have finished reading the HELP instructions, push the ENTER key to return to the last form or the MENU (f7) key to return to the MAIN MENU.

f4 REFRESH - This key resets the terminal, erases the screen and brings up a fresh copy of the form. If you loose your form due to a power or line failure or the terminal hangs up, the REFRESH key will restore the terminal to normal operation.

f5 CONFIRM - This key is used when you have chang-

ed a record in the EDIT mode, or if the program wants confirmation that the data on the screen is acceptable. The program will prompt with a message at the bottom of the screen when a CONFIRM is desired. Before a CONFIRM is requested the data must pass all normal program edits.

Note: The terminal normally does not read data when the function keys are pushed. If you need to read the screen contents after a function key has been pushed call the IMMVREADFIELD Subroutine to force a read.

f6 NEXT - This key will cause the program to go to the NEXT form or next step.

f7 MAIN MENU - This key causes the program to display the MAIN MENU SELECTION form. Use this key to change from one program mode to another.

f8 EXIT - This key ends the program.

COBOL Application Program

A. General: The attached COBOL program has sufficient structure to allow the programmer to readily plug in applications without having to spend additional time coding VIEW procedures. At Cape Data, I have used this program layout to do extensive data entry routines which edit against the IMAGE data base, and provide detailed error messages and help routines. The program procedure division consists of 3 parts:

1. Opening
2. Main Loop
3. Closing

The program contains routines which call the VIEW procedures listed in Fig. 7. The program also contains special VIEW data fields in working storage.

1. Opening - The program opens the terminal, forms file, the data base and displays the MENU screen.

2. Main Loop - After opening the program performs the Main Loop until either the f8 (exit) function key is pushed, or the program encounters a fatal error. The Main Loop consists of 3 parts:

a. Read Keys and Screen
b. Edit Input
c. Process Input (if valid) and Refresh Screen

3. Closing - The program closes the terminal file, forms file and data base.

B. Program Data Division Considerations: The working storage area contains the buffers needed by the various VIEW procedures used. Every VIEW procedure called uses the VIEW-COM buffer (Fig. 2). Most of the fields useful to the programmer have self-explanatory names. Remember the V-Language field must be set to zero for a COBOL program.

The data area passed between the program and VIEW (using VGETBUFFER and VPUTBUFFER calls) is defined as DATA-BUF (Fig. 3). This Buffer is redefined for each screen layout. Note the title field and error message fields.

The program uses a forms table which contains the MENU selection character, form number, next form number, and help form number for each routine. When the user makes a selection from the Screen Menu, the program scans this table to find the corresponding screen references (Fig. 4). The program picks up the Form Name corresponding to the Screen Number from the Form Name Table (Fig. 6). Additional routines and forms can quickly be added by making additional entries in the tables.

Program Main Loop

The program goes to the MAIN LOOP and remains there until the user pushes the f8 (exit) function key. The program performs a terminal read (VREADFIELDS) each time either a function key or the enter key is depressed. The program then tests to see if any function keys were pushed (VIEW returns the key number pushed into the last key field of the VIEW-COM buffer).

If the ENTER key was pushed (key zero) the program will perform the edit routine corresponding to the screen routine selected. Within each edit routine the program does the following steps (Fig. 5):

1. Zeros the field error array and set the field count.
2. Performs VIEW edits (VFIELDEDITS).
3. Get the data buffer from VIEW (VGETBUFFER).
4. Clear the error message fields.
5. Performs any user defined edits (if an error is detected, a flag is set in the field error array and a message is moved to the appropriate error field).
6. The data area is sent back to VIEW (VPUTBUFFER).
7. Perform VIEW edits again (VFIELDEDITS). This zero fills and justifies data.
8. The field error array is scanned and any error fields are set to blink (VSETERROR).

9. The screen is updated and displayed (VSHOWFORM).

If the routine passes all edits, the program then goes to the corresponding valid record routine. If an error exists or a function key was depressed, the program will do the appropriate error and screen enhancing routines.

4. SPL Forced Read Subroutine

```
1     $CONTROL SUBPROGRAM
2     << KEPT AS IMMREAD >>
3     << THIS ROUTINE IS CALLED TO FORCE        >>
4     <<  AN IMMEDIATE READ (RE-READ FOR DATA) >>
5     <<  IN PARTICULAR CASES WHERE THE USER    >>
6     <<  USED A SOFT KEY TO INDICATE ACTIONS   >>
7     <<  HE/SHE WANTS PERFORMED WITH THE DATA  >>
8     <<  THAT HAS BEEN ENTERED ON THE SCREEN   >>
9     << SINCE THE HITTING OF A SOFT KEY DOES   >>
10    <<  NOT TRANSFER THE ACTUAL BUFFER DATA   >>
11    <<  A CALL TO THIS ROUTINE OR ONE LIKE    >>
12    <<  IT IS NEEDED TO GET THE SCREEN DATA   >>
13    <<  INTO THE PROGRAM WHERE IT CAN BE      >>
14    <<  WORKED UPON                           >>
15    << IN COBOL, THE CALL WOULD BE            >>
16    <<    CALL "IMMVREADFIELDS" USING VCONT   >>
17    <<    WITH VCONT BEING THE V/3000 CONTROL >>
18    <<    AREA                                >>
19    << IN THE USER PROGRAM, THIS SHOULD BE    >>
20    <<  TREATED EXACTLY AS IF IT HAD BEEN A   >>
21    <<  CALL TO "VREADFIELDS"                 >>
22    BEGIN PROCEDURE IMMVREADFIELDS(CONT);
23    INTEGER ARRAY CONT;
24    BEGIN
25    PROCEDURE VREADFIELDS(C);
26    INTEGER ARRAY C;
27    OPTION EXTERNAL;
28    CONT(55).(13:2):=%1;
29    VREADFIELDS(CONT);
30    CONT(55).(13:2):=0;
31    END;
32    END.
```

Figure 1

```
FORMSPEC  VERSION A.00.01
FORMS FILE: HUDGEFORM.DEVELOP.EDEVELOP

FORM: VENDOR_DATA
   REPEAT OPTION: N

   NEXT FORM OPTION: C
   NEXT FORM: HUDMAINT_HELP

VENDOR MASTER SPECIFICATIONS, INPUT & EDITING
********* ********* ********* ********* ********* ********* ********* *********
         TITLE_____________________________________________________________
                  V E N D O R - M A S T E R   I N F O R M A T I O N

              VENDOR NUMBER   V_NBR_      VEND_ERR_____________

              VENDOR'S NAME   VEND_NAME_____________________
              VENDOR ADDRESS  VEND_ADDR_____________________
                              VEND_ADDR2____________________
              VENDOR'S CITY   VEND_CITY_____ STATE: ST  ZIP: V_ZIP_____

              PAYMENT ADDRESS P_ADDRESS_____________________
                              P_ADDRESS2____________________
              PAYMENT CITY    P_CITY_____________ STATE: PS  ZIP: P_ZIP_____

     (PAYMENT ADDRESS USED ONLY IF YOU WANT PAYMENTS GO TO A DIFFERENNT ADDRESS)

                  VENDOR'S PHONE NUMBER: (AC_) EXC-PHON

           1099 CODE IN      VENDOR CODE VC          VENDOR STATUS VS
                    CODE ERR_____________      status ERR_____________
********* ********* ********* ********* ********* ********* ********* *********


FIELD: TITLE
   NUM: 1    LEN: 69    NAME: TITLE          ENH: NONE  FTYPE: D  DTYPE: CHAR
   INIT VALUE:
*** PROCESSING SPECIFICATIONS ***
INIT
SET TO SFTITLE


FIELD: V_NBR
   NUM: 2    LEN: 6     NAME: V_VAR          ENH: I     FTYPE: R  DTYPE: DIG
   INIT VALUE:
*** PROCESSING SPECIFICATIONS ***
FIELD
JUSTIFY RIGHT
FILL LEADING "0"


FIELD: VEND_ERR
   NUM: 3    LEN: 26    NAME: VEND_ERR       ENH: R     FTYPE: D  DTYPE: CHAR
   INIT VALUE:


FIELD: VEND_NAME
   NUM: 4    LEN: 30    NAME: VEND_NAME      ENH: I     FTYPE: R  DTYPE: CHAR
   INIT VALUE:
```

Figure 2

```
6.4
6.5     01 VIEW-DATA.
6.6          05 V-STATUS                 PIC S9(4)  COMP  VALUE ZERO.
6.7          05 V-LANGUAGE               PIC S9(4)  COMP  VALUE ZERO.
6.8          05 V-COM-AREA-LEN           PIC S9(4)  COMP  VALUE 60.
6.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7            05 VIEW-MODE                PIC S9(4)  COMP  VALUE ZERO.
7.1          05 LAST-KEY                 PIC S9(4)  COMP  VALUE ZERO.
7.2          05 V-NUM-ERRS               PIC S9(4)  COMP  VALUE ZERO.
7.3          05 V-WINDOW-ENH             PIC S9(4)  COMP  VALUE ZERO.
7.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7.6          05 V-FNAME                  PIC X(15)  VALUE SPACES.
7.7          05 FILLER                   PIC X      VALUE SPACES.
7.8          05 V-FNAME                  PIC X(15)  VALUE SPACES.
7.9          05 FILLER                   PIC X      VALUE SPACES.
8            05 V-REPEAT-OPT             PIC S9(4)  COMP  VALUE ZERO.
8.1          05 V-NF-OPT                 PIC S9(4)  COMP  VALUE ZERO.
8.2          05 V-NBR-LINES              PIC S9(4)  COMP  VALUE ZERO.
8.3          05 V-UBUF-LEN               PIC S9(4)  COMP  VALUE ZERO.
8.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.6          05 V-DELETE-FLAG            PIC S9(4)  COMP  VALUE ZERO.
8.7          05 V-SHOW-CONTROL           PIC S9(4)  COMP  VALUE ZERO.
8.8          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9            05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.1          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.2          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.3          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.6          05 V-NUM-RECS               PIC S9(6)  COMP  VALUE ZERO.
9.7          05 V-REC-NBR                PIC S9(6)  COMP  VALUE ZERO.
9.8          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10           05 V-TERM-FILE-NBR          PIC S9(4)  COMP  VALUE ZERO.
10.1         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.2         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.3         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.4         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.5         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.6         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.7         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.8         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.9         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11           05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11.1         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11.2
```

Figure 3

```
13.3     01  DATA-BUF.
13.4         05 FILLER          PIC X(69).
13.5         05 DATA-IN         PIC X(443).
13.6
13.7     01 MENU-DATA       REDEFINES DATA-BUF.
13.8         05 FILLER                 PIC X(69).
13.9         05 SELECT-IN              PIC X.
14           05 SELECT-ERR          PIC X(26).
14.1
14.2     01 VEND-IN         REDEFINES DATA-BUF.
14.3         05 FILLER                 PIC X(69).
14.4         05 VEND-NBR               PIC X(5).
14.5         05 VEND-NBR-ERR           PIC X(26).
14.6         05 VEND-NAME              PIC X(30).
14.7         05 S-ADDRESS              PIC X(30).
14.8         05 S-ADDRESS2             PIC X(30).
14.9         05 S-CITY                 PIC X(20).
15           05 S-STATE                PIC X(2).
15.1         05 S-ZIP                  PIC X(10).
15.2         05 P-ADDRESS              PIC X(30).
15.3         05 P-ADDRESS2             PIC X(30).
15.4         05 P-CITY                 PIC X(20).
15.5         05 P-STATE                PIC X(2).
15.6         05 P-ZIP                  PIC X(10).
15.7         05 PHONE-NBR.
15.8          10 PHONE-AC              PIC X(3).
15.9          10 PHONE-EX              PIC X(3).
16            10 PHONE-NO              PIC X(4).
16.1         05 FLAG-1099              PIC X(2).
16.2         05 VEND-CODE              PIC X(2).
16.3         05 VENDOR-STATUS          PIC X(2).
16.4         05 VEND-CODE-ERR          PIC X(26).
16.5         05 VEND-STATUS-ERR        PIC X(26).
16.6
16.7     01 BUDG-IN                 REDEFINES DATA-BUF.
16.8         05 FILLER                 PIC X(69).
16.9         05 ACCT-NBR               PIC X(20).
17           05 BUDG-NBR-ERR           PIC X(26).
17.1         05 BUDG-NY-AMT            PIC X(13).
17.2
```

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

Figure 4

```
19      01 FORM-TABLE.
19.1         05 FORM-1.
19.2           10 FILLER                PIC X    VALUE "/".
19.3           10 FILLER                PIC 9(4) COMP VALUE  1.
19.4           10 FILLER                PIC 9(4) COMP VALUE  1.
19.5           10 FILLER                PIC 9(4) COMP VALUE  14.
19.6         05 FORM-2.
19.7           10 FILLER                PIC X    VALUE "A".
19.8           10 FILLER                PIC 9(4) COMP VALUE  2.
19.9           10 FILLER                PIC 9(4) COMP VALUE  1.
20             10 FILLER                PIC 9(4) COMP VALUE  13.
20.1         05 FORM-3.
20.2           10 FILLER                PIC X    VALUE "B".
20.3           10 FILLER                PIC 9(4) COMP VALUE  3.
20.4           10 FILLER                PIC 9(4) COMP VALUE  1.
20.5           10 FILLER                PIC 9(4) COMP VALUE  13.
20.6         05 FORM-4.
20.7           10 FILLER                PIC X    VALUE "C".
20.8           10 FILLER                PIC 9(4) COMP VALUE  4.
20.9           10 FILLER                PIC 9(4) COMP VALUE  1.
21             10 FILLER                PIC 9(4) COMP VALUE  11.
21.1         05 FORM-5.
21.2           10 FILLER                PIC X    VALUE "D".
21.3           10 FILLER                PIC 9(4) COMP VALUE  2.
21.4           10 FILLER                PIC 9(4) COMP VALUE  1.
21.5           10 FILLER                PIC 9(4) COMP VALUE  13.
21.6         05 FORM-6.
21.7           10 FILLER                PIC X    VALUE "E".
21.8           10 FILLER                PIC 9(4) COMP VALUE  5.
21.9           10 FILLER                PIC 9(4) COMP VALUE  1.
22             10 FILLER                PIC 9(4) COMP VALUE  12.
22.1         05 FORM-7.
22.2           10 FILLER                PIC X    VALUE "Z".
22.3           10 FILLER                PIC 9(4) COMP VALUE  1.
22.4           10 FILLER                PIC 9(4) COMP VALUE  1.
22.5           10 FILLER                PIC 9(4) COMP VALUE  12.
22.6         05 FORM-8.
22.7           10 FILLER                PIC X    VALUE "Z".
22.8           10 FILLER                PIC 9(4) COMP VALUE  1.
22.9           10 FILLER                PIC 9(4) COMP VALUE  1.
23             10 FILLER                PIC 9(4) COMP VALUE  14.
23.1         05 FORM-9.
23.2           10 FILLER                PIC X    VALUE "Z".
23.3           10 FILLER                PIC 9(4) COMP VALUE  1.
23.4           10 FILLER                PIC 9(4) COMP VALUE  1.
23.5           10 FILLER                PIC 9(4) COMP VALUE  14.
23.6         05 FORM-10.
23.7           10 FILLER                PIC X    VALUE "/".
23.8           10 FILLER                PIC 9(4) COMP VALUE  1.
23.9           10 FILLER                PIC 9(4) COMP VALUE  1.
24             10 FILLER                PIC 9(4) COMP VALUE  14.
24.1
24.2    01 FORM-SPEC-ARRAY          REDEFINES FORM-TABLE.
24.3         05 FORM-SPECS             OCCURS  10 TIMES.
24.4           10 FORM-ID              PIC X.
24.5           10 FORM-NBR             PIC 9(4)  COMP.
24.6           10 FORM-NEXT            PIC 9(4)  COMP.
24.7           10 FORM-HELP            PIC 9(4)  COMP.
```

Figure 4a

```
24.1    01 FORM-NAME-TABLE.
25           05 FORM-1.
25.1           10 FILLER          PIC X(15) VALUE "BUDMAINT_MENU  ".
25.2           10 FILLER          PIC 9(4)  COMP VALUE  96.
25.3         05 FORM-2.
25.4           10 FILLER          PIC X(15) VALUE "VENDOR_DATA    ".
25.5           10 FILLER          PIC 9(4)  COMP VALUE 557.
25.6         05 FORM-3.
25.7           10 FILLER          PIC X(15) VALUE "BANK_MSTR_DATA ".
25.8           10 FILLER          PIC 9(4)  COMP VALUE 325.
25.9         05 FORM-4.
26             10 FILLER          PIC X(15) VALUE "BUDGET_LOAD    ".
26.1           10 FILLER          PIC 9(4)  COMP VALUE 128.
26.2         05 FORM-5.
26.3           10 FILLER          PIC X(15) VALUE "               ".
26.4           10 FILLER          PIC 9(4)  COMP VALUE 96.
26.5         05 FORM-6.
26.6           10 FILLER          PIC X(15) VALUE "               ".
26.7           10 FILLER          PIC 9(4)  COMP VALUE 96.
26.8         05 FORM-7.
26.9           10 FILLER          PIC X(15) VALUE "               ".
27             10 FILLER          PIC 9(4)  COMP VALUE  96.
27.1         05 FORM-8.
27.2           10 FILLER          PIC X(15) VALUE "               ".
27.3           10 FILLER          PIC 9(4)  COMP VALUE 96.
27.4         05 FORM-9.
27.5           10 FILLER          PIC X(15) VALUE "               ".
27.6           10 FILLER          PIC 9(4)  COMP VALUE 96.
27.7         05 FORM-10.
27.8           10 FILLER          PIC X(15) VALUE "               ".
27.9           10 FILLER          PIC 9(4)  COMP VALUE 96.
28           05 FORM-11.
28.1           10 FILLER          PIC X(15) VALUE "HELP_BANK_DATA ".
28.2           10 FILLER          PIC 9(4)  COMP VALUE 69.
28.3         05 FORM-12.
28.4           10 FILLER          PIC X(15) VALUE "HELP_BUDG_LOAD ".
28.5           10 FILLER          PIC 9(4)  COMP VALUE 69.
28.6         05 FORM-13.
28.7           10 FILLER          PIC X(15) VALUE "HELP_VENDOR    ".
28.8           10 FILLER          PIC 9(4)  COMP VALUE 69.
28.9         05 FORM-14.
29             10 FILLER          PIC X(15) VALUE "BUDMAINT_HELP  ".
29.1           10 FILLER          PIC 9(4)  COMP VALUE 69.
29.2         05 FORM-15.
29.3           10 FILLER          PIC X(15) VALUE "HELP_CREDIT_SUP".
29.4           10 FILLER          PIC 9(4)  COMP VALUE 69.
29.5
29.6    01 FORM-NAME-ARRAY   REDEFINES FORM-NAME-TABLE.
29.7         05 FORM-NAME-INFO         OCCURS  15 TIMES.
29.8           10 FORM-NAME              PIC X(15).
29.9           10 FORM-DATA-LEN          PIC 9(4)  COMP.
30
```

Figure 5

```
48.3    102000-EDIT-A.
48.4        MOVE ZERO TO CHECK-RESULT.
48.5        MOVE ZERO TO FIELD-ZERO.
48.6        MOVE  15 TO FIELD-CNT.
48.7        PERFORM 805000-VIEW-EDIT.
48.8        PERFORM 807000-GET-BUFFER.
48.9        MOVE SPACES TO VEND-NBR-ERR, VEND-STATUS-ERR.
49          PERFORM 102100-CK-VEND-NBR.
49.1        PERFORM 102200-CK-VEND-CODE.
49.2        PERFORM 102300-CK-VEND-STATUS.
49.3        PERFORM 102400-CK-VEND-1099.
49.4
49.5        PERFORM 809000-PUT-BUFFER.
49.6        PERFORM 805000-VIEW-EDIT.
49.7        IF V-NUM-ERRS NOT = ZERO MOVE 1 TO CHECK-RESULT.
49.8        MOVE ZERO TO FIELD-LOC.
49.9        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
50
50.1    102100-CK-VEND-NBR.
50.2        MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
50.3        PERFORM 831000-GET-VEND-MSTR.
50.4        IF COND-WORD = 17 NEXT SENTENCE
50.5         ELSE MOVE "INVALID! DUPLICATE NUMBER" TO VEND-NBR-ERR
50.6           MOVE  1 TO FIELD-ERR (2)
50.7           MOVE  1 TO CHECK-RESULT.
50.8
50.9    102200-CK-VEND-CODE.
51          IF VEND-CODE OF VEND-IN = "VN" OR = "VM" OR = "OP"
51.1         NEXT SENTENCE
51.2          ELSE MOVE "INVALID VENDOR CODE!" TO VEND-CODE-ERR
51.3           MOVE  1 TO FIELD-ERR (19)
51.4           MOVE  1 TO CHECK-RESULT.
51.5
51.6    102300-CK-VEND-STATUS.
51.7        IF VENDOR-STATUS OF VEND-IN = "CR" OR = "XX"
51.8         NEXT SENTENCE
51.9          ELSE MOVE "INVALID STATUS CODE!" TO VEND-STATUS-ERR
52             MOVE  1 TO FIELD-ERR (20)
52.1           MOVE  1 TO CHECK-RESULT.
52.2
52.3    102400-CK-VEND-1099.
52.4        IF FLAG-1099 OF VEND-IN = SPACES OR = "Y "
52.5         NEXT SENTENCE
52.6          ELSE
52.7           MOVE  1 TO FIELD-ERR (18)
52.8           MOVE  1 TO CHECK-RESULT.
52.9
53
```

Figure 6

Summary of VIEW Procedures

PROCEDURE	FUNCTION
VCLOSEBATCH	Closes batch file.
VCLOSEFORMF	Closes forms file.
VCLOSETERM	Closes terminal file.
VERRMSG	Returns message associated with error code.
VFIELDEDITS	Edits field data and performs other field processing.
VFINISHFORM	Performs final processing specified for form.
VGETBUFFER	Reads contents of data buffer into user program.
VGETFIELD	Reads field from data buffer into user program.
VGETNEXTFORM	Reads next form into form definition area of memory; window and data buffer are not affected.
VGETtype	Reads field from data buffer to user program, converting data to specified type.
VINITFORM	Sets data buffer to initial values for form.
VOPENBATCH	Opens batch file for processing.
VOPENFORMF	Opens forms file for processing.
VOPENTERM	Opens terminal file for processing.
VPRINTFORM	Prints current form and data on offline list device.
VPUTBUFFER	Writes data from user program to data buffer.
VPUTFIELD	Writes data from user program to field in data buffer.
VPUTtype	Writes data of specified type from user program to data buffer, converting data to ASCII.
VPUTWINDOW	Writes message from user program to window area in memory for later display.
VREADBATCH	Reads record from batch file into data buffer.
VREADFIELDS	Reads input from terminal into data buffer.
VSETERROR	Sets error flag for data field in error; and moves error message to window area.
VSHOWFORM	Updates terminal screen, merging the current form, any data in buffer, and any message in window.
VWRITEBATCH	Writes data from data buffer to batch file.

COBOL VIEW Application Source Program

```
1
1.1     $CONTROL LIST,NOSOURCE,USLINIT,ANSIVOS
1.2      IDENTIFICATION DIVISION.
1.3      PROGRAM-ID.
1.4          LAYOUT.
1.5     **THIS PROGRAM PROVIDES A LAYOUT FOR USING VIEW FORMS WITH COBOL.
1.6     ** THE PROGRAM DISPLAYS, EDITS, AND UPDATES MANY FORMS.
1.7     *** VERS 0.00      APRIL  9,1980.
1.8      AUTHOR.
1.9          P SOMERS.
2        INSTALLATION.
2.1          CAPE DATA INC.
2.2     *** (C)COPYRIGHT 1980 CAPE DATA INC. CAPE MAY,NEW JERSEY 08204
2.3      DATE-COMPILED.
2.4      ENVIRONMENT DIVISION.
2.5      CONFIGURATION SECTION.
2.6      SOURCE-COMPUTER. HP-3000.
2.7      OBJECT-COMPUTER. HP-3000.
2.8      INPUT-OUTPUT SECTION.
2.9
3        DATA DIVISION.
3.1
3.2      WORKING-STORAGE SECTION.
3.3      77  FBASE              PIC X(13) VALUE "  BUDGET.PUB;".
3.4      77  PASSWORD           PIC X(8)  VALUE "ABC1234;".
3.5      77  DSET-NAME          PIC X(16) VALUE SPACES.
3.6      77  NO-ITEM            PIC X(2)  VALUE "0;".
3.7      77  ITEM               PIC X(16) VALUE SPACES.
3.8      77  LIST               PIC X(30) VALUE SPACES.
3.9      77  ALL-ITEMS          PIC X(2)  VALUE "@;".
4        77  SAME-ITEMS         PIC X(2)  VALUE "*;".
4.1      77  ARGUMENT           PIC X(20) VALUE SPACES.
4.2      77  MODE1              PIC 9(4)  COMP  VALUE 1.
4.3      77  MODE2              PIC 9(4)  COMP  VALUE 2.
4.4      77  MODE3              PIC 9(4)  COMP  VALUE 3.
4.5      77  MODE5              PIC 9(4)  COMP  VALUE 5.
4.6      77  MODE7              PIC 9(4)  COMP  VALUE 7.
4.7      77  MODE-FLAG          PIC X.
4.8      77  LOC-FORM           PIC 9(4)  COMP.
4.9      77  LOC-FORM-NAME      PIC 9(4)  COMP.
5        77  LOC-FIND           PIC 9(4)  COMP.
5.1      77  FUTURE-FLAG        PIC 9.
5.2      77  NEW-FLAG           PIC 9.
5.3      77  LAST-RESULT        PIC 9 .
5.4      77  CHECK-RESULT       PIC S9(4)  COMP.
5.5      01  BELL               PIC X   VALUE " ".
5.6
5.7      01 STATUS-AREA.
5.8          05 COND-WORD       PIC S9(4)   COMP.
5.9          05 D-L             PIC S9(4)   COMP.
6            05 R-W             PIC S9(9)   COMP.
6.1          05 C-L             PIC S9(9)   COMP.
6.2          05 B-A             PIC S9(9)   COMP.
6.3          05 F-A             PIC S9(9)   COMP.
6.4
6.5      01 VIEW-COM.
6.6          05 V-STATUS                 PIC S9(4)  COMP  VALUE ZERO.
6.7          05 V-LANGUAGE               PIC S9(4)  COMP  VALUE ZERO.
6.8          05 V-COM-AREA-LEN           PIC S9(4)  COMP  VALUE 60.
6.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7            05 VIEW-MODE                PIC S9(4)  COMP  VALUE ZERO.
7.1          05 LAST-KEY                 PIC S9(4)  COMP  VALUE ZERO.
7.2          05 V-NUM-ERRS               PIC S9(4)  COMP  VALUE ZERO.
7.3          05 V-WINDOW-ENH             PIC S9(4)  COMP  VALUE ZERO.
7.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
7.6          05 V-CFNAME                 PIC X(15)  VALUE SPACES.
7.7          05 FILLER                   PIC X      VALUE SPACES.
7.8          05 V-NFNAME                 PIC X(15)  VALUE SPACES.
7.9          05 FILLER                   PIC X      VALUE SPACES.
8            05 V-REPEAT-OPT             PIC S9(4)  COMP  VALUE ZERO.
8.1          05 V-NF-OPT                 PIC S9(4)  COMP  VALUE ZERO.
8.2          05 V-NBR-LINES              PIC S9(4)  COMP  VALUE ZERO.
8.3          05 V-DBUF-LEN               PIC S9(4)  COMP  VALUE ZERO.
8.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.6          05 V-DELETE-FLAG            PIC S9(4)  COMP  VALUE ZERO.
8.7          05 V-SHOW-CONTROL           PIC S9(4)  COMP  VALUE ZERO.
8.8          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
8.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9            05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.1          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.2          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.3          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.4          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.5          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.6          05 V-NUM-RECS               PIC S9(6)  COMP  VALUE ZERO.
9.7          05 V-REC-NBR                PIC S9(6)  COMP  VALUE ZERO.
9.8          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
9.9          05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10           05 V-TERM-FILE-NBR          PIC S9(4)  COMP  VALUE ZERO.
10.1         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.2         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.3         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.4         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.5         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.6         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.7         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.8         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
10.9         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11           05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11.1         05 FILLER                   PIC S9(4)  COMP  VALUE ZERO.
11.2
11.3     01 V-FILE-NAME          PIC X(66).
11.4
11.5     01 ERR-MES-BUF              PIC X(76).
11.6     01 LEN-ERR-BUF              PIC S9(4)  COMP  VALUE 76.
11.7     01 LEN-ERR-MES              PIC S9(4)  COMP  VALUE ZERO.
11.8     01 TERM-FILE                PIC X(8)   VALUE SPACES.
11.9     01 DATA-LEN                 PIC S9(4)  COMP .
12       01 FIELD-NBR                PIC S9(4)  COMP VALUE ZERO.
12.1     01 ACT-FIELD-LEN            PIC S9(4)  COMP VALUE ZERO.
12.2     01 NEXT-FIELD-NBR           PIC S9(4)  COMP VALUE ZERO.
12.3     01 WD-MESSAGE               PIC S9(4)  COMP VALUE -1.
12.4     01 DUM-1.                   PIC X(15).
12.5     01 NUM-OUT                  PIC S9(9)V99.
12.6     01 NUM-DISP-1               PIC -------9.99.
12.7
12.8
12.9     01 ACCT-MSTR                    COPY ACCTMSTR.
13
13.1     01 VEND-MSTR                    COPY VENDMSTR.
13.2
13.3     01  DATA-BUF.
13.4         05 FILLER          PIC X(69).
13.5         05 DATA-IN         PIC X(443).
13.6
13.7     01 MENU-DATA       REDEFINES DATA-BUF.
13.8         05 FILLER                  PIC X(69).
13.9         05 SELECT-IN               PIC X.
14           05 SELECT-ERR           PIC X(26).
14.1
14.2     01 VEND-IN       REDEFINES DATA-BUF.
14.3         05 FILLER                  PIC X(69).
14.4         05 VEND-NBR                PIC X(6).
14.5         05 VEND-NBR-ERR            PIC X(26).
14.6         05 VEND-NAME               PIC X(30).
14.7         05 S-ADDRESS               PIC X(30).
14.8         05 S-ADDRESS2              PIC X(30).
14.9         05 S-CITY                  PIC X(20).
15           05 S-STATE                 PIC X(2).
15.1         05 S-ZIP                   PIC X(10).
15.2         05 P-ADDRESS               PIC X(30).
15.3         05 P-ADDRESS2              PIC X(30).
15.4         05 P-CITY                  PIC X(20).
15.5         05 P-STATE                 PIC X(2).
15.6         05 P-ZIP                   PIC X(10).
15.7         05 PHONE-NBR.
15.8          10 PHONE-AC               PIC X(3).
15.9          10 PHONE-EX               PIC X(3).
16            10 PHONE-NO               PIC X(4).
16.1         05 FLAG-1099               PIC X(2).
16.2         05 VEND-CODE               PIC X(2).
16.3         05 VENDOR-STATUS           PIC X(2).
16.4         05 VEND-CODE-ERR           PIC X(26).
16.5         05 VEND-STATUS-ERR         PIC X(26).
16.6
16.7     01 BUDG-IN                 REDEFINES DATA-BUF.
16.8         05 FILLER                  PIC X(69).
16.9         05 ACCT-NBR                PIC X(20).
17           05 BUDG-NBR-ERR            PIC X(26).
17.1         05 BUDG-NY-AMT             PIC X(13).
17.2
17.3     ***********************************************
17.4     ***          FORM TABLE LAYOUT:
17.5     ***               SELECTION CHARACTER
17.6     ***               FORM NUMBER
17.7     ***               NEXT FORM NUMBER
17.8     ***               HELP FORM NUMBER
17.9     ***
18       ***********************************************
18.1
18.2     ***********************************************
18.3     ***          FORM NAME TABLE LAYOUT:
18.4     ***               FORM NAME (VIEW FORM NAME)
18.5     ***               LENGTH OF DATA FIELDS
18.6     ***
18.7     ***********************************************
18.8
18.9
19        01 FORM-TABLE.
19.1         05 FORM-1.
19.2          10 FILLER                 PIC X    VALUE "Z".
19.3          10 FILLER                 PIC 9(4) COMP VALUE  1.
19.4          10 FILLER                 PIC 9(4) COMP VALUE  1.
19.5          10 FILLER                 PIC 9(4) COMP VALUE  10.
19.6         05 FORM-2.
19.7          10 FILLER                 PIC X    VALUE "A".
19.8          10 FILLER                 PIC 9(4) COMP VALUE  2.
19.9          10 FILLER                 PIC 9(4) COMP VALUE  1.
20            10 FILLER                 PIC 9(4) COMP VALUE  13.
20.1         05 FORM-3.
20.2          10 FILLER                 PIC X    VALUE "B".
20.3          10 FILLER                 PIC 9(4) COMP VALUE  3.
20.4          10 FILLER                 PIC 9(4) COMP VALUE  1.
20.5          10 FILLER                 PIC 9(4) COMP VALUE  13.
20.6         05 FORM-4.
20.7          10 FILLER                 PIC X    VALUE "C".
20.8          10 FILLER                 PIC 9(4) COMP VALUE  4.
20.9          10 FILLER                 PIC 9(4) COMP VALUE  1.
21            10 FILLER                 PIC 9(4) COMP VALUE  11.
21.1         05 FORM-5.
21.2          10 FILLER                 PIC X    VALUE "D".
21.3          10 FILLER                 PIC 9(4) COMP VALUE  2.
21.4          10 FILLER                 PIC 9(4) COMP VALUE  1.
21.5          10 FILLER                 PIC 9(4) COMP VALUE  13.
21.6         05 FORM-6.
21.7          10 FILLER                 PIC X    VALUE "E".
21.8          10 FILLER                 PIC 9(4) COMP VALUE  5.
21.9          10 FILLER                 PIC 9(4) COMP VALUE  1.
22            10 FILLER                 PIC 9(4) COMP VALUE  12.
22.1         05 FORM-7.
22.2          10 FILLER                 PIC X    VALUE "Z".
22.3          10 FILLER                 PIC 9(4) COMP VALUE  1.
22.4          10 FILLER                 PIC 9(4) COMP VALUE  1.
22.5          10 FILLER                 PIC 9(4) COMP VALUE  12.
```

```
22.6        05 FORM-8.
22.7          10 FILLER                  PIC X    VALUE "/".
22.8          10 FILLER                  PIC 9(4) COMP VALUE  1.
22.9          10 FILLER                  PIC 9(4) COMP VALUE  1.
23            10 FILLER                  PIC 9(4) COMP VALUE  14.
23.1        05 FORM-9.
23.2          10 FILLER                  PIC X    VALUE "/".
23.3          10 FILLER                  PIC 9(4) COMP VALUE  1.
23.4          10 FILLER                  PIC 9(4) COMP VALUE  1.
23.5          10 FILLER                  PIC 9(4) COMP VALUE  14.
23.6        05 FORM-10.
23.7          10 FILLER                  PIC X    VALUE "/".
23.8          10 FILLER                  PIC 9(4) COMP VALUE  1.
23.9          10 FILLER                  PIC 9(4) COMP VALUE  1.
24            10 FILLER                  PIC 9(4) COMP VALUE  14.
24.1
24.2    01 FORM-SPEC-ARRAY            REDEFINES FORM-TABLE.
24.3        05 FORM-SPECS                OCCURS  10 TIMES.
24.4          10 FORM-ID                 PIC X.
24.5          10 FORM-NBR                PIC 9(4)  COMP.
24.6          10 FORM-NEXT               PIC 9(4)  COMP.
24.7          10 FORM-HELP               PIC 9(4)  COMP.
24.8
24.9    01 FORM-NAME-TABLE.
25          05 FORM-1.
25.1          10 FILLER              PIC X(15) VALUE "BUDMAINT_MENU  ".
25.2          10 FILLER              PIC 9(4)  COMP VALUE  96.
25.3        05 FORM-2.
25.4          10 FILLER              PIC X(15) VALUE "VENDOR_DATA    ".
25.5          10 FILLER              PIC 9(4)  COMP VALUE 357.
25.6        05 FORM-3.
25.7          10 FILLER              PIC X(15) VALUE "BANK_MSTR_DATA ".
25.8          10 FILLER              PIC 9(4)  COMP VALUE 325.
25.9        05 FORM-4.
26            10 FILLER              PIC X(15) VALUE "BUDGET_LOAD    ".
26.1          10 FILLER              PIC 9(4)  COMP VALUE 128.
26.2        05 FORM-5.
26.3          10 FILLER              PIC X(15) VALUE "               ".
26.4          10 FILLER              PIC 9(4)  COMP VALUE 96.
26.5        05 FORM-6.
26.6          10 FILLER              PIC X(15) VALUE "               ".
26.7          10 FILLER              PIC 9(4)  COMP VALUE 96.
26.8        05 FORM-7.
26.9          10 FILLER              PIC X(15) VALUE "               ".
27            10 FILLER              PIC 9(4)  COMP VALUE  96.
27.1        05 FORM-8.
27.2          10 FILLER              PIC X(15) VALUE "               ".
27.3          10 FILLER              PIC 9(4)  COMP VALUE 96.
27.4        05 FORM-9.
27.5          10 FILLER              PIC X(15) VALUE "               ".
27.6          10 FILLER              PIC 9(4)  COMP VALUE 96.
27.7        05 FORM-10.
27.8          10 FILLER              PIC X(15) VALUE "               ".
27.9          10 FILLER              PIC 9(4)  COMP VALUE 96.
28          05 FORM-11.
28.1          10 FILLER              PIC X(15) VALUE "HELP_BANK_DATA ".
28.2          10 FILLER              PIC 9(4)  COMP VALUE 69.
28.3        05 FORM-12.
28.4          10 FILLER              PIC X(15) VALUE "HELP_BUDG_LOAD ".
28.5          10 FILLER              PIC 9(4)  COMP VALUE 69.
28.6        05 FORM-13.
28.7          10 FILLER              PIC X(15) VALUE "HELP_VENDOR    ".
28.8          10 FILLER              PIC 9(4)  COMP VALUE 69.
28.9        05 FORM-14.
29            10 FILLER              PIC X(15) VALUE "BUDMAINT_HELP  ".
29.1          10 FILLER              PIC 9(4)  COMP VALUE 69.
29.2        05 FORM-15.
29.3          10 FILLER              PIC X(15) VALUE "HELP_CREDIT_SHP".
29.4          10 FILLER              PIC 9(4)  COMP VALUE 69.
29.5
29.6    01 FORM-NAME-ARRAY   REDEFINES FORM-NAME-TABLE.
29.7        05 FORM-NAME-INFO            OCCURS  15 TIMES.
29.8          10 FORM-NAME               PIC X(15).
29.9          10 FORM-DATA-LEN           PIC 9(4)  COMP.
30
30.1
30.2
30.3
30.4
30.5
30.6
30.7    01 FORMAT-CNTL                   COPY FORMTCTL.
30.8
30.9    01 FORMAT-13.
31          05 FILLER                  PIC S9(4)  COMP VALUE 1.
31.1        05 FILLER                  PIC S9(4)  COMP VALUE 13.
31.2        05 FILLER                  PIC S9(4)  COMP VALUE 0.
31.3        05 FILLER                  PIC X      VALUE " ".
31.4        05 FILLER                  PIC X      VALUE SPACES.
31.5        05 U13-FORMAT.
31.6          10 FILLER                PIC S9(4)  COMP VALUE 11.
31.7          10 FILLER                PIC S9(4)  COMP VALUE ZERO.
31.8          10 FILLER                PIC X      VALUE SPACES.
31.9          10 FILLER                PIC X      VALUE "N".
32            10 FILLER                PIC X      VALUE "1".
32.1          10 FILLER                PIC X      VALUE "2".
32.2
32.3
32.4    01 ERROR-ARRAY.
32.5        05 FIELD-ERR          OCCURS 56 TIMES   PIC 9.
32.6    01 FIELD-ZERO       REDEFINES ERROR-ARRAY   PIC X(56).
32.7
32.8    01 FIELD-CNT                  PIC 99  COMP.
32.9    01 FIELD-LOC                  PIC 99  COMP.
33
33.1    ****************************************************************
33.2    $PAGE
33.3
33.4
33.5    PROCEDURE DIVISION .
33.6    000000-MAIN-PART SECTION 01.
33.7    000000-PROGRAM-LOGIC.
33.8        PERFORM 900000-OPEN-PROGRAM.
33.9        PERFORM 100000-READ-LOOP UNTIL LAST-KEY = 8.
34          PERFORM 970000-CLOSE-PROGRAM.
34.1        STOP RUN.
34.2
34.3    100000-READ-LOOP.
34.4        PERFORM 801000-READ-TERM.
34.5        IF LAST-KEY = 0
34.6           NEXT SENTENCE
34.7         ELSE IF LAST-KEY = 1
34.8           PERFORM 881000-KEY-1
34.9         ELSE IF LAST-KEY = 2
35             PERFORM 882000-KEY-2
35.1         ELSE IF LAST-KEY = 3
35.2           PERFORM 883000-KEY-3
35.3         ELSE IF LAST-KEY = 4
35.4           PERFORM 884000-KEY-4
35.5         ELSE IF LAST-KEY = 5
35.6           PERFORM 885000-KEY-5
35.7         ELSE IF LAST-KEY = 6
35.8           PERFORM 886000-KEY-6
35.9         ELSE IF LAST-KEY = 7
36             PERFORM 887000-KEY-7
36.1         ELSE
36.2           PERFORM 888000-KEY-8.
36.3
36.4        IF CHECK-RESULT NOT = 0  OR LAST-KEY = 5
36.5           NEXT SENTENCE
36.6         ELSE IF LOC-FORM =  1
36.7           PERFORM 101000-EDIT-MENU
36.8         ELSE IF LOC-FORM =  2
36.9           PERFORM 102000-EDIT-A
37           ELSE IF LOC-FORM =  3
37.1           PERFORM 103000-EDIT-B
37.2         ELSE IF LOC-FORM =  4
37.3           PERFORM 104000-EDIT-C
37.4         ELSE IF LOC-FORM =  5
37.5           PERFORM 105000-EDIT-D
37.6         ELSE IF LOC-FORM =  6
37.7           PERFORM 106000-EDIT-E
37.8         ELSE
37.9           MOVE  4 TO CHECK-RESULT.
38
38.1        IF CHECK-RESULT NOT = ZERO
38.2           NEXT SENTENCE
38.3         ELSE IF LOC-FORM =  1
38.4           PERFORM 150000-VALID-MENU
38.5         ELSE IF LOC-FORM =  2
38.6           PERFORM 151000-VALID-A
38.7         ELSE IF LOC-FORM =  3
38.8           PERFORM 152000-VALID-B
38.9         ELSE IF LOC-FORM =  4
39             PERFORM 153000-VALID-C
39.1         ELSE IF LOC-FORM =  5
39.2           PERFORM 155000-VALID-E
39.3         ELSE IF LAST-KEY NOT = 5
39.4           PERFORM 889000-ASK-CONFIRM
39.5         ELSE IF LOC-FORM =  5
39.6           PERFORM 154000-VALID-D
39.7         ELSE
39.8           MOVE 4 TO CHECK-RESULT.
39.9
40
40.1        IF CHECK-RESULT = 3 PERFORM 851006-REFRESH-TERM.
40.2        IF CHECK-RESULT = 4 MOVE 3 TO V-SHOW-CONTROL
40.3         MOVE "Z" TO MODE-FLAG.
40.4        IF CHECK-RESULT = 0 PERFORM 811000-FORM-INITIALIZE.
40.5        IF CHECK-RESULT = 2 OR = 4 OR = 6 PERFORM 852000-NEXT-FORM.
40.6        IF CHECK-RESULT NOT = 9
40.7           PERFORM 804000-SHOW-FORM
40.8           PERFORM 803000-CLEAR-WINDOW.
40.9        MOVE ZERO TO CHECK-RESULT, V-SHOW-CONTROL.
41
41.1
41.2    101000-EDIT-MENU.
41.3        MOVE ZERO TO CHECK-RESULT.
41.4        PERFORM 805000-VIEW-EDIT.
41.5        PERFORM 807000-GET-BUFFER.
41.6        MOVE SPACES TO SELECT-ERR.
41.7        IF SELECT-IN  = "X" MOVE 8 TO LAST-KEY
41.8           MOVE 9 TO CHECK-RESULT
41.9         ELSE
42             PERFORM 101100-EDIT-MENU-DATA.
42.1
42.2    101100-EDIT-MENU-DATA.
42.3        PERFORM 101110-MENU-SCAN VARYING LOC-FIND FROM 1  BY  1
42.4         UNTIL LOC-FIND >  15
42.5          OR SELECT-IN = FORM-ID (LOC-FIND).
42.6        IF LOC-FIND >  15 PERFORM 101120-MENU-ERROR
42.7         ELSE PERFORM 803000-CLEAR-WINDOW.
42.8
42.9    101110-MENU-SCAN.
43          EXIT.
43.1
43.2    101120-MENU-ERROR.
43.3        MOVE
43.4         "PLEASE SELECT ONE OF THE ABOVE LETTERS AND RE-ENTER"
43.5         TO ERR-MES-BUF.
43.6        MOVE 51  TO LEN-ERR-BUF.
43.7        MOVE  2  TO FIELD-NBR.
43.8        PERFORM 812000-SET-ERROR.
43.9        ADD  1 TO V-NUM-ERRS.
44          MOVE 1 TO CHECK-RESULT.
44.1
```

```
44.2
44.3    ************************************************************
44.4    *                    MODE SELECTIONS                       *
44.5    *   A = ADD VENDOR MASTER              N =                 *
44.6    *   B = ADD BANK MASTER                O =                 *
44.7    *   C = ADD NEXT YR.BUDGET             P =                 *
44.8    *   D = EDIT VENDOR MASTER             Q =                 *
44.9    *   E = LOAD BUDGET                    R =                 *
45      *   F =                                S =                 *
45.1    *   G =                                T =                 *
45.2    *   H =                                U =                 *
45.3    *   I =                                V =                 *
45.4    *   J =                                W =                 *
45.5    *   K =                                X =  EXIT  PROGRAM  *
45.6    *   L =                                Y =                 *
45.7    *   M =                                Z = MAIN MENU       *
45.8    *                                                          *
45.9    *                                                          *
46      ************************************************************
46.1
46.2
46.3    ************************************************
46.4    ***            CHECK RESULT  CODES             ***
46.5    ***    0 = NO ERRORS                 5 =       ***
46.6    ***    1 = ERRORS, RE-READ           6 = NEXT FORM   ***
46.7    ***    2 = BRING UP NEW FORM         7 =       ***
46.8    ***    3 = REFRESH TERMINAL          8 =       ***
46.9    ***    4 = RETURN TO MAIN MENU       9 = EXIT PROGRAM  ***
47      ***                                              ***
47.1    ************************************************
47.2
47.3
47.4    ****************************************************************
47.5    ***         FUNCTION  KEY CODES                              ***
47.6    ***    F1 =  SKIP                F5  = CONFIRM ENTRY         ***
47.7    ***    F2 = CLEAR (INITIALIZE)   F6  = NEXT FORM             ***
47.8    ***    F3 = HELP                 F7  = MAIN MENU             ***
47.9    ***    F4 = REFRESH SCREEN       F8  = EXIT                  ***
48      ***                                                         ***
48.1    ****************************************************************
48.2
48.3    102000-EDIT-A.
48.4        MOVE ZERO TO CHECK-RESULT.
48.5        MOVE ZERO TO FIELD-ZERO.
48.6        MOVE  16 TO FIELD-CNT.
48.7        PERFORM 805000-VIEW-EDIT.
48.8        PERFORM 807000-GET-BUFFER.
48.9        MOVE SPACES TO VEND-NBR-ERR, VEND-STATUS-ERR.
49          PERFORM 102100-CK-VEND-NBR.
49.1        PERFORM 102200-CK-VEND-CODE.
49.2        PERFORM 102300-CK-VEND-STATUS.
49.3        PERFORM 102400-CK-VEND-1099.
49.4
49.5        PERFORM 809000-PUT-BUFFER.
49.6        PERFORM 805000-VIEW-EDIT.
49.7        IF V-NUM-ERRS NOT = ZERO MOVE 1 TO CHECK-RESULT.
49.8        MOVE ZERO TO FIELD-LOC.
49.9        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
50
50.1    102100-CK-VEND-NBR.
50.2        MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
50.3        PERFORM 831000-GET-VEND-MSTR.
50.4        IF COND-WORD = 17 NEXT SENTENCE
50.5         ELSE MOVE "INVALID/ DUPLICATE NUMBER" TO VEND-NBR-ERR
50.6           MOVE  1 TO FIELD-ERR (2)
50.7           MOVE  1 TO CHECK-RESULT.
50.8
50.9    102200-CK-VEND-CODE.
51          IF VEND-CODE OF VEND-IN = "VW" OR = "VM" OR = "DP"
51.1         NEXT SENTENCE
51.2          ELSE MOVE "INVALID VENDOR CODE!" TO VEND-CODE-ERR
51.3            MOVE  1 TO FIELD-ERR (19)
51.4            MOVE  1 TO CHECK-RESULT.
51.5
51.6    102300-CK-VEND-STATUS.
51.7        IF VENDOR-STATUS OF VEND-IN = "CR" OR = "XX"
51.8         NEXT SENTENCE
51.9          ELSE MOVE "INVALID STATUS CODE!" TO VEND-STATUS-ERR
52            MOVE  1 TO FIELD-ERR (20)
52.1          MOVE  1 TO CHECK-RESULT.
52.2
52.3    102400-CK-VEND-1099.
52.4        IF FLAG-1099 OF VEND-IN = SPACES OR = "Y "
52.5         NEXT SENTENCE
52.6         ELSE
52.7           MOVE  1 TO FIELD-ERR (18)
52.8           MOVE  1 TO CHECK-RESULT.
52.9
53
53.1
53.2
53.3    103000-EDIT-B.
53.4        MOVE ZERO TO CHECK-RESULT.
53.5        MOVE ZERO TO FIELD-ZERO.
53.6        MOVE  5 TO FIELD-CNT.
53.7        PERFORM 805000-VIEW-EDIT.
53.8        PERFORM 807000-GET-BUFFER.
53.9        PERFORM 103100-CK-VEND-NBR.
54          PERFORM 102200-CK-VEND-CODE.
54.1        PERFORM 102300-CK-VEND-STATUS.
54.2        PERFORM 102400-CK-VEND-1099.
54.3        PERFORM 809000-PUT-BUFFER.
54.4        PERFORM 805000-VIEW-EDIT.
54.5        IF V-NUM-ERRS  NOT = ZERO
54.6           MOVE  1 TO CHECK-RESULT.
54.7        MOVE ZERO TO FIELD-LOC.
54.8        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
54.9

55      103100-CK-VEND-NBR.
55.1        IF NEW-FLAG = ZERO NEXT SENTENCE
55.2         ELSE IF VEND-NBR OF VEND-IN = VEND-NBR OF VEND-MSTR
55.3           MOVE  1 TO NEW-FLAG
55.4         ELSE MOVE ZERO TO NEW-FLAG.
55.5        MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
55.6        PERFORM 831000-GET-VEND-MSTR.
55.7        IF COND-WORD = ZERO AND NEW-FLAG = 1
55.8           NEXT SENTENCE
55.9         ELSE IF COND-WORD = ZERO
56             PERFORM 103110-SETUP-VENDOR
56.1         ELSE
56.2           MOVE "NON-EXISTENT VENDOR NUMBER!" TO VEND-NBR-ERR
56.3           MOVE  1 TO FIELD-ERR (2)
56.4           MOVE  1 TO CHECK-RESULT.
56.5
56.6    103110-SETUP-VENDOR.
56.7        MOVE CORR VEND-MSTR TO VEND-IN.
56.8        MOVE  1 TO NEW-FLAG, CHECK-RESULT.
56.9
57      104000-EDIT-C.
57.1        MOVE ZERO TO CHECK-RESULT.
57.2        MOVE ZERO TO FIELD-ZERO.
57.3        MOVE  3 TO FIELD-CNT.
57.4        PERFORM 805000-VIEW-EDIT.
57.5        PERFORM 807000-GET-BUFFER.
57.6    *   MOVE SPACES TO ERROR DISPLAY FIELDS
57.7    *   PERFORM EDIT ROUTINES
57.8        PERFORM 809000-PUT-BUFFER.
57.9        PERFORM 805000-VIEW-EDIT.
58          IF V-NUM-ERRS  NOT = ZERO
58.1           MOVE  1 TO CHECK-RESULT.
58.2        MOVE ZERO TO FIELD-LOC.
58.3        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
58.4
58.5    105000-EDIT-D.
58.6        MOVE ZERO TO CHECK-RESULT.
58.7        MOVE ZERO TO FIELD-ZERO.
58.8        MOVE  3 TO FIELD-CNT.
58.9        PERFORM 805000-VIEW-EDIT.
59          PERFORM 807000-GET-BUFFER.
59.1    *   MOVE SPACES TO ERROR DISPLAY FIELDS
59.2    *   PERFORM EDIT ROUTINES
59.3        PERFORM 809000-PUT-BUFFER.
59.4        PERFORM 805000-VIEW-EDIT.
59.5        IF V-NUM-ERRS  NOT = ZERO
59.6           MOVE  1 TO CHECK-RESULT.
59.7        MOVE ZERO TO FIELD-LOC.
59.8        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
59.9
60      106000-EDIT-E.
60.1        MOVE ZERO TO CHECK-RESULT.
60.2        MOVE ZERO TO FIELD-ZERO.
60.3        MOVE  3 TO FIELD-CNT.
60.4        PERFORM 805000-VIEW-EDIT.
60.5        PERFORM 807000-GET-BUFFER.
60.6    *   MOVE SPACES TO ERROR DISPLAY FIELDS
60.7    *   PERFORM EDIT ROUTINES
60.8        PERFORM 809000-PUT-BUFFER.
60.9        PERFORM 805000-VIEW-EDIT.
61          IF V-NUM-ERRS  NOT = ZERO
61.1           MOVE  1 TO CHECK-RESULT.
61.2        MOVE ZERO TO FIELD-LOC.
61.3        PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.
61.4
61.5
61.6    150000-VALID-RECORD SECTION 03.
61.7    150000-VALID-MENU.
61.8        MOVE 2 TO CHECK-RESULT.
61.9        MOVE LOC-FIND TO LOC-FORM.
62          MOVE ZERO TO NEW-FLAG.
62.1
62.2    151000-VALID-A.
62.3
62.4    152000-VALID-B.
62.5
62.6    153000-VALID-C.
62.7
62.8    154000-VALID-D.
62.9
63      155000-VALID-E.
63.1
63.2
63.3
63.4
63.5    800000-UTILITIES SECTION 02.
63.6
63.7    801000-READ-TERM.
63.8        CALL "VREADFIELDS" USING VIEW-COM.
63.9        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
64
64.1    802000-PUT-WINDOW.
64.2        CALL "VPUTWINDOW" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF.
64.3
64.4    803000-CLEAR-WINDOW.
64.5        MOVE SPACES TO ERR-MES-BUF.
64.6        PERFORM 802000-PUT-WINDOW.
64.7
64.8
64.9    804000-SHOW-FORM.
65          CALL "VSHOWFORM" USING VIEW-COM.
65.1        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
65.2
65.3
65.4    805000-VIEW-EDIT.
65.5        CALL "VFIELDEDITS" USING VIEW-COM.
65.6
```

```
65.7    806000-FINISH-FORM.
65.8        CALL "VFINISHFORM" USING VIEW-COM.
65.9
66      807000-GET-BUFFER.
66.1        CALL "VGETBUFFER" USING VIEW-COM DATA-BUF V-DBUF-LEN.
66.2        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
66.3
66.4    809000-PUT-BUFFER.
66.5        CALL "VPUTBUFFER" USING VIEW-COM DATA-BUF V-DBUF-LEN.
66.6        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
66.7
66.8    810000-GET-FORM-FILE.
66.9        MOVE 0  TO V-REPEAT-OPT.
67          MOVE 0  TO V-NF-OPT.
67.1        CALL "VGETNEXTFORM" USING VIEW-COM.
67.2        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
67.3
67.4    811000-FORM-INITIALIZE.
67.5        MOVE 65  TO V-WINDOW-ENH.
67.6        CALL "VINITFORM" USING VIEW-COM.
67.7    *   ADD INDIVIDUAL FORMS INTIALIZATION HERE AS REQ.
67.8        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
67.9        PERFORM 809000-CLEAR-WINDOW.
68          MOVE  1 TO LAST-RESULT.
68.1
68.2    812000-SET-ERROR.
68.3        CALL "VSETERROR" USING VIEW-COM FIELD-NBR ERR-MES-BUF
68.4         LEN-ERR-BUF.
68.5        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
68.6
68.7    813000-SET-ERROR-FIELDS.
68.8        ADD 1  TO FIELD-LOC.
68.9        IF FIELD-ERR (FIELD-LOC) = 1
69             CALL "VSETERROR" USING VIEW-COM FIELD-LOC ERR-MES-BUF
69.1           NO-MESSAGE
69.2             IF V-STATUS NOT = ZERO
69.3               PERFORM 992000-VIEW-ERROR
69.4             ELSE
69.5               NEXT SENTENCE
69.6         ELSE
69.7           NEXT SENTENCE.
69.8
69.9    814000-CONFIRM-READ.
70          CALL "IMMEDREADFIELDS" USING VIEW-COM.
70.1        IF V-STATUS NOT = ZERO  PERFORM 992000-VIEW-ERROR.
70.2
70.3
70.4    820000-SPACE-NUMBER.
70.5        MOVE FORMAT-13 TO FORMAT-CNTL.
70.6        CALL "CAPE'ENTRY" USING FORMAT-CNTL NUM-IN NUM-OUT.
70.7        IF CFIELD-ERR (1) = ZERO AND CENTRY-ERR = ZERO
70.8         MOVE NUM-OUT TO NUM-DISP-13
70.9          ELSE MOVE 1 TO CHECK-RESULT.
71
71.1    830000-GET-ACCT-MSTR.
71.2        MOVE "ACCOUNT-MSTR;" TO DSET-NAME.
71.3        CALL "DBGET" USING FBASE DSET-NAME MODE7 STATUS-AREA
71.4         ALL-ITEMS ACCT-MSTR ARGUMENT.
71.5        IF COND-WORD NOT = ZERO AND NOT = 17
71.6           PERFORM 991000-STATUS-CK
71.7         ELSE
71.8           NEXT SENTENCE.
71.9
72      831000-GET-VEND-MSTR.
72.1        MOVE "VENDOR-MSTR;" TO DSET-NAME.
72.2        CALL "DBGET" USING FBASE DSET-NAME MODE7 STATUS-AREA
72.3         ALL-ITEMS VEND-MSTR ARGUMENT.
72.4        IF COND-WORD = ZERO OR = 17
72.5           NEXT SENTENCE
72.6         ELSE
72.7           PERFORM 991000-STATUS-CK.
72.8
72.9    841000-DB-LOCK.
73          CALL "DBLOCK" USING FBASE DSET-NAME MODE3 STATUS-AREA.
73.1        IF COND-WORD NOT = ZERO PERFORM 991000-STATUS-CK.
73.2
73.3    842000-DB-UNLOCK.
73.4        CALL "DBUNLOCK" USING FBASE DSET-NAME MODE1 STATUS-AREA.
73.5        IF COND-WORD NOT = ZERO PERFORM 991000-STATUS-CK.
73.6
73.7
73.8
73.9    851000-REFRESH-TERM.
74          PERFORM 980000-CLOSE-TERM.
74.1        PERFORM 902000-OPEN-TERM.
74.2        MOVE 5 TO V-SHOW-CONTROL.
74.3        PERFORM 811000-FORM-INITIALIZE.
74.4
74.5    852000-NEXT-FORM.
74.6        IF CHECK-RESULT =  4 MOVE  1  TO LOC-FORM.
74.7        MOVE FORM-NBR (LOC-FORM) TO LOC-FORM-NAME.
74.8        MOVE FORM-NAME (LOC-FORM-NAME) TO V-NFNAME.
74.9        PERFORM 810000-GET-FORM-FILE.
75          PERFORM 811000-FORM-INITIALIZE.
75.1
75.2    853000-EDIT-ERROR.
75.3        CALL "VERRMSG" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF
75.4         LEN-ERR-MES.
75.5        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
75.6        CALL "VPUTWINDOW" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF.
75.7
75.8    854000-PUT-TITLE.
75.9    *   MOVE 1 TO FIELD-NBR.
76      *   CALL "VPUTFIELD" USING VIEW-COM FIELD-NBR TITLE-BUF
76.1    *    TITLE-LEN ACT-FIELD-LEN NEXT-FIELD-NBR.
76.2    *   IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
76.3
76.4
76.5
76.6    860000-START-HELP.
76.7        MOVE FORM-HELP (LOC-FORM) TO LOC-FORM-NAME.
76.8        MOVE FORM-NAME (LOC-FORM-NAME)  TO V-NFNAME.
76.9
77      861000-HELP-DISPLAY.
77.1        PERFORM 810000-GET-FORM-FILE.
77.2        PERFORM 854000-PUT-TITLE.
77.3        PERFORM 804000-SHOW-FORM.
77.4        CALL "VREADFIELDS" USING VIEW-COM.
77.5        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
77.6        IF LAST-KEY = 8
77.7           MOVE  9 TO CHECK-RESULT
77.8         ELSE IF LAST-KEY = 7
77.9           MOVE  4 TO CHECK-RESULT
78           ELSE IF LAST-KEY = 4
78.1           MOVE  3 TO CHECK-RESULT
78.2         ELSE
78.3           MOVE 2 TO CHECK-RESULT.
78.4
78.5
78.6    865000-INITIAL-VENDOR.
78.7    *   MOVE VEND-NBR OF VEND-MSTR TO VEND-NBR OF VEND-IN.
78.8    *   MOVE VEND-NAME OF VEND-MSTR TO VEND-NAME OF VEND-IN.
78.9    *   PERFORM 809000-PUT-BUFFER.
79      *                 *
79.1
79.2    870000-UPDATE-ACCT-MSTR.
79.3        CALL "DBUPDATE" USING FBASE DSET-NAME MODE1
79.4         STATUS-AREA ALL-ITEMS ACCT-MSTR.
79.5        IF COND-WORD = ZERO NEXT SENTENCE
79.6         ELSE PERFORM 991000-STATUS-CK.
79.7
79.8
79.9
80      881000-KEY-1.
80.1        MOVE 1 TO CHECK-RESULT.
80.2        MOVE "INVALID KEY SELECTED, IGNORED" TO ERR-MES-BUF.
80.3        MOVE 29 TO LEN-ERR-BUF.
80.4        PERFORM 802000-PUT-WINDOW.
80.5
80.6    882000-KEY-2.
80.7        MOVE 1 TO CHECK-RESULT.
80.8        PERFORM 811000-FORM-INITIALIZE.
80.9
81      883000-KEY-3.
81.1        PERFORM 860000-START-HELP THRU 861000-HELP-DISPLAY.
81.2
81.3    884000-KEY-4.
81.4        MOVE 4 TO CHECK-RESULT.
81.5
81.6    885000-KEY-5.
81.7        IF LAST-RESULT = ZERO
81.8           PERFORM 814000-CONFIRM-READ
81.9           MOVE  5 TO LAST-KEY
82             MOVE ZERO TO CHECK-RESULT
82.1         ELSE
82.2           PERFORM 890000-INVALID-CONFIRM.
82.3
82.4    886000-KEY-6.
82.5        MOVE 6 TO CHECK-RESULT.
82.6        MOVE FORM-NEXT (LOC-FORM ) TO LOC-FORM.
82.7
82.8
82.9    887000-KEY-7.
83          MOVE "Z" TO MODE-FLAG.
83.1        MOVE 4 TO CHECK-RESULT.
83.2
83.3    888000-KEY-8.
83.4        MOVE 9 TO CHECK-RESULT.
83.5
83.6    889000-ASK-CONFIRM.
83.7        MOVE
83.8         "VALID RECORD, PUSH CONFIRM KEY (F5) TO POST AS SHOWN"
83.9          TO ERR-MES-BUF.
84          MOVE 52  TO LEN-ERR-BUF.
84.1        PERFORM 802000-PUT-WINDOW.
84.2        MOVE  1 TO CHECK-RESULT.
84.3        MOVE ZERO TO LAST-RESULT.
84.4
84.5    890000-INVALID-CONFIRM.
84.6        MOVE  1 TO CHECK-RESULT.
84.7        MOVE
84.8         "INVALID USE OF CONFIRM KEY!  CONFIRM NOT REQUESTED!"
84.9          TO ERR-MES-BUF.
85          MOVE  52 TO LEN-ERR-BUF.
85.1        PERFORM 802000-PUT-WINDOW.
85.2
85.3    ************************************************
85.4
85.5    900000-START-STOP SECTION 51.
85.6    900000-OPEN-PROGRAM.
85.7        DISPLAY "VIEW/COBOL LAYOUT PROGRAM  VERS. 0.01".
85.8        PERFORM 901000-OPEN-DATA-BASE.
85.9        PERFORM 902000-OPEN-TERM.
86          PERFORM 903000-OPEN-VFORM.
86.1        PERFORM 904000-START-MENU.
86.2
86.3    901000-OPEN-DATA-BASE.
86.4        CALL "DBOPEN" USING FBASE PASSWORD MODE1 STATUS-AREA.
86.5        IF COND-WORD = ZERO
86.6           NEXT SENTENCE
86.7         ELSE
86.8           PERFORM 991000-STATUS-CK
86.9           STOP RUN.
87
87.1    902000-OPEN-TERM.
87.2        MOVE ZERO TO V-STATUS, V-LANGUAGE.
```

```
87.3          VIEW-MODE, LAST-KEY, V-NUM-ERRS, V-REPEAT-OPT
87.4          V-NF-OPT.
87.5        CALL "VOPENTERM" USING VIEW-COM TERM-FILE.
87.6        IF V-STATUS = ZERO
87.7            NEXT SENTENCE
87.8         ELSE
87.9            PERFORM 992000-VIEW-ERROR
88              CALL "VCLOSEFORMF" USING VIEW-COM
88.1            DISPLAY ERR-MES-BUF  " STOPPING RUN! "
88.2            PERFORM 990000-STOP-PAR.
88.3
88.4    903000-OPEN-VFORM.
88.5        MOVE "BUDGFORM,BUDGET,PROGLIB" TO V-FILE-NAME.
88.6        CALL "VOPENFORMF" USING VIEW-COM V-FILE-NAME.
88.7        IF V-STATUS = ZERO
88.8            NEXT SENTENCE
88.9         ELSE
89              PERFORM 992000-VIEW-ERROR
89.1            CALL "VCLOSETERM" USING VIEW-COM
89.2            DISPLAY ERR-MES-BUF  " STOPPING RUN! "
89.3            PERFORM 990000-STOP-PAR.
89.4
89.5    904000-START-MENU.
89.6        MOVE  1  TO LOC-FORM.
89.7        PERFORM 852000-NEXT-FORM.
89.8        PERFORM 804000-SHOW-FORM.
89.9        MOVE ZERO TO CHECK-RESULT.
90
90.1
90.2    970000-CLOSE-PROGRAM.
90.3        PERFORM 980000-CLOSE-TERM.
90.4        PERFORM 981000-CLOSE-VFORM.
90.5        PERFORM 990000-STOP-PAR.
90.6
90.7    980000-CLOSE-TERM.
90.8        CALL "VCLOSETERM" USING VIEW-COM.
90.9        IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
91
91.1    981000-CLOSE-VFORM.
91.2        CALL "VCLOSEFORMF" USING VIEW-COM.
91.3        IF V-STATUS NOT = ZERO PERFORM 992000-VIEW-ERROR.
91.4
91.5
91.6    990000-STOP-PAR.
91.7        CALL "DBCLOSE" USING PHASE DSET-NAME MODE1 STATUS-AREA.
91.8        STOP RUN.
91.9
92
92.1    991000-STATUS-CK.
92.2        PERFORM 980000-CLOSE-TERM.
92.3        PERFORM 981000-CLOSE-VFORM.
92.4        CALL "DBEXPLAIN" USING STATUS-AREA.
92.5        PERFORM 990000-STOP-PAR.
92.6
92.7    992000-VIEW-ERROR.
92.8
92.9        CALL "VERRMSG" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF
95           LEN-ERR-MES.
93.1        DISPLAY BELL "VIEW ERROR!!!".
93.2        DISPLAY ERR-MES-BUF.
93.3        DISPLAY BELL "PROGRAM TERMINATED DUE TO ABOVE ERROR!!!".
93.4        PERFORM 990000-STOP-PAR.
93.5
```

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

IMAGE/COBOL: Practical Guidelines

David J. Greer
Robelle Consulting Ltd.
27597-32B Avenue
Aldergrove, B.C.
Canada V0X 1A0

Summary

This document presents a set of practical "rules" for designing, accessing, and maintaining IMAGE databases in the COBOL environment. This document is designed to aid systems analysts, especially ones that are new to the HP 3000, in producing "good" IMAGE database designs. Each "rule" is demonstrated with examples and instructions for applying it. Attention is paid to those details that make using the database trouble-free for the COBOL programmer, and maintaining the database easier for the database administrator.

Database Design

IMAGE/3000 is the database system supplied by Hewlett-Packard [6]; it is used to store and retrieve application information. A database does not suddenly appear out of thin air; it develops through a long and involved process. At some time, a logical database design must be translated into the actual schema that implements a physical IMAGE database. This phase is the most difficult of the database development cycle [7]. The IMAGE/3000 Reference Manual [6] contains a sample database called STORE, which demonstrates most of the attributes of IMAGE. Throughout this document, the STORE database will be referenced when examples are needed.

Logical Database Design

The foundation of a new database is a logical design, which is created by examining the user requirements for input forms, for on-line enquiries, and for batch reports. The database should be viewed as an intermediate storage area for the information that comes from the input forms and is eventually displayed on the output reports [9,10].

Database design is normally done from the bottom up, as opposed to structured program design, which is usually done from the top down. The starting point for a database is the elements (items) that will be stored in the database. These data elements represent the user's information. In the early stages, the size and type of these elements are not needed, only the name and values.

Rule: *Start your logical database design by naming each data item, then identify what values it can have and where it will be used.*

Here is an example of a subset of data items for the STORE database:

CUST-STATUS Two characters, attached to each customer record. Valid values are: 10 = advanced, 20 = current, 30 = arrears and 40 = inactive.

DELIV-DATE Six numeric characters; Date, YYMMDD, attached to every sales order as the promised delivery date.

ON-HAND-QTY Seven numeric characters, attached to every inventory record to show the current quantity of an inventory item available for shipping.

PRODUCT-PRICE Eight numeric characters (6 whole digits, 2 decimal places), attached to every sales record. This is the price of a product sold, on the date that the sale was made.

As the logical database design develops to deeper levels of detail, the elements needed should eventually reach a stable list. These elements should then be combined into records by grouping logically related items together.

It is important that "repetition" be recognized early in the design. An example of this is a customer's address. The most flexible method of implementing addresses is a variable number of records associated with the customer account number. Another method is to make the address field an X-type variable (e.g., X24) repeated 5 times (e.g., 5X24). Repeated items are often the most natural way to represent the user's data, so the use of repeated items is encouraged.

After the records are designed, enquiry paths must be assigned. During the early stages of database design it is important to use elements that are readable and easy to implement with the tools at hand. This permits testing of the database using tools such as QUERY, AQ, and PROTOS.

Physical Database Design

After the logical database is designed, the IMAGE schema must be developed. The restrictions of IMAGE must now be worked into the database design.

IMAGE requires that all items needed in the database be defined at the beginning of the schema, and a size and type must be associated with each. Initially, declare each item as type X (display); later, the data type may be altered.

Records are implemented as IMAGE datasets. Start by treating each record format as a master dataset.

Rule: *If a record is uniquely identified by a single key value, start by making it a master dataset (e.g., customer master record keyed by a unique customer number).*

The STORE database assumes that each CUST-ACCOUNT field is unique. Furthermore, there is only one customer record for each CUST-ACCOUNT. All of the information describing one customer is gathered together to result in the M-CUSTOMER dataset:

```
NAME:  M-CUSTOMER,       MANUAL (1/2);       <<PREFIX = MCS>>
ENTRY:
       CITY
      ,CREDIT-RATING
      ,CUST-ACCOUNT(1)      <<KEY FIELD>>
      ,CUST-STATUS
      ,NAME-FIRST
      ,NAME-LAST
      ,STATE-CODE
      ,STREET-ADDRESS
      ,ZIP-CODE
      ;
CAPACITY:  211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
```

Rule: *If a "natural" master dataset will require on-line retrieval via an alternate key, drop it down to a detail dataset.*

The detail dataset will have two keys: the key field of the original master dataset, and the alternate key. You will have to create a new automatic master for the original key field, and you may have to create an automatic master for the alternate key (unless you already have a manual master dataset for that item).

Take the M-CUSTOMER dataset as an example. Assume that in addition to needing on-line access by CUST-ACCOUNT, it is also necessary to have on-line access by NAME-LAST. The following dataset structure would result:

```
NAME:  A-CUSTOMER,       AUTOMATIC (1/2),    <<PREFIX = ACS>>
ENTRY:
       CUST-ACCOUNT(2)     <<KEY FIELD>>
       ;
CAPACITY:  211;     <<A-CUSTOMER,PRIME; ESTIMATED>>

NAME:  A-NAME-LAST,      AUTOMATIC (1/2),    <<PREFIX = ANL>>
ENTRY:
       NAME-LAST(1)     <<KEY FIELD>>
       ;
CAPACITY:  211;     <<A-NAME-LAST,PRIME; CAP(A-CUSTOMER)>>
```

```
NAME:  D-CUSTOMER,       DETAIL (1/2);       <<PREFIX = DCS>>
ENTRY:
       CITY
      ,CREDIT-RATING
      ,CUST-ACCOUNT(!A-CUSTOMER)     <<KEY FIELD, PRIMARY>>
      ,CUST-STATUS
      ,NAME-FIRST
      ,NAME-LAST(A-NAME-LAST)        <<KEY FIELD>>
      ,STATE-CODE
      ,STREET-ADDRESS
      ,ZIP-CODE
      ;
CAPACITY:  210;     <<D-CUSTOMER; CAP(A-CUSTOMER)>>
```

Rule: *If an entry can occur several times for the primary key value, store it in a detail dataset.*

Detail datasets are for repetition and multiple keys. Master datasets can only contain one entry per unique key value. An example of repetition in a detail dataset is a customer address field. The customer address can be stored as a repeated field in a master dataset, but eventually there will be an address that will not fit into the fixed-size repeated field. Instead of a repeated field, use a detail dataset to store multiple lines of an address. For example:

```
ADDRESS-LINE,          X24;    << An individual line of address.  This
                                  item is used in D-ADDRESS to provide an
                                  arbitrary number of address lines for
                                  each customer.
                               >>
CUST-ACCOUNT,          Z8;     << Customer account number.  This field
                                  is used as a key to the M-CUSTOMER
                                  and D-ADDRESS datasets.
                               >>
LINE-NO,               X2;     << Used to keep address lines in D-ADDRESS
                                  in the correct order.  This field also
                                  provides a unique way of identifying
                                  each address line for every

NAME:  D-ADDRESS          DETAIL (1/2);       <<PREFIX = DAD>>
ENTRY:
       ADDRESS-LINE
      ,CUST-ACCOUNT(!M-CUSTOMER(LINE-NO))  <<KEY FIELD, PRIMARY PATH>>
      ,LINE-NO                             <<SORT FIELD>>
      ;
CAPACITY:  844;     <<D-ADDRESS; 4 * CAP(M-CUSTOMER)>>
```

Dataset Paths

The following definition of PATHs and CHAINs comes from Alfredo Rego [11]:

> *A PATH is a relationship between a MASTER dataset and a DETAIL dataset. The master and the detail must contain a field of the same type and size as a common "bond", called the SEARCH FIELD. A path is a structural property of a database.*
>
> *A CHAIN, on the other hand, contains a MASTER ENTRY and its associated DETAIL ENTRIES (if any), as defined by the PATH relationship between the master and the detail for the particular DETAIL SEARCH FIELD. ...A chain is nothing more than a collection of related entries (for instance, a bank customer would be the master entry and all of this customer's checks would be the detail entries; the "chain" would include the master AND all its details; the chain for customer number 1 would be completely different from the chain for customer number 2).*

Paths provide fast access at a certain cost: adding and deleting records on the path is expensive. The more paths there are, the more expensive it gets [11]. Another restriction of paths is that there can be a maximum of 64,000 records on a single path for a single key value. This sounds like a large number, but it can be very easy to expand a chain to this size if a key is specified for a specific, reporting summary program (e.g., billing cycle, in monthly billing transactions).

Rule: *Avoid more than two paths into a detail dataset.*

There are some instances where three paths are necessary, but these should be avoided as much as possible. Before adding a path, examine how the path is going to be used. If it is added just to make one or two batch programs easier to program, the path is not justified. The batch programs should serially read and sort the dataset, then merge the sorted dataset with any other necessary information from the database.

The date paths of the SALES dataset of the STORE database are good examples of unnecessary paths. Because the chain lengths of paths organized by date are almost always very long, such a chain is rarely allowed. Also, users are often interested in a large range of dates (such as a month, quarter or year), not just a specific day.

In order to obtain the same type of reporting by date, it is possible to do one of the following: 1) read the database every night and produce a report of all records entered every day; 2) keep a sequential file of all records added to the dataset on a particular day. This file can then be used as an index into the database.

These are not the only solutions to removing the date paths, but they indicate the kind of solutions that are possible. Because of the high volume and length of the average chain, date paths are prime candidates for removal from a database.

The following example demonstrates how the SALES dataset should have been declared:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,          DETAIL (1/2);       <<PREFIX = DSA>>
ENTRY:
       CUST-ACCOUNT(!M-CUSTOMER)   <<KEY FIELD, PRIMARY PATH>>
      ,DELIV-DATE
      ,PRODUCT-NO(M-PRODUCT)       <<KEY FIELD>>
      ,PRODUCT-PRICE
      ,PURCH-DATE
      ,SALES-QTY
      ,SALES-TAX
      ,SALES-TOTAL
      ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Rule: *Avoid sorted paths.*

Because sorted paths can require very high overhead when records are added or deleted, they should be avoided as much as possible. There are some instances when a sorted path makes the system and program design much easier, but this convenience must be traded off against the higher cost of maintaining sorted chains.

The most important criteria in evaluating sorted chains are: 1) whether the chain is needed for batch or on-line access. In batch, it is possible to read and sort the dataset, rather than relying on sorted chains. In an on-line program, this is usually not possible, so sorted chains are required. 2) How long is the average chain going to be? The longer the chain, the more expensive it is to keep sorted. If chains have fewer than 10 entries per key value on average, sorted chains can be permitted. 3) How are records being added to the dataset? If a sorted chain is present, and data is added to the dataset in sorted order, there is very little extra overhead in the sorted chain. If, on the other hand, data is added in random fashion, there is a very high cost associated with the sorted chain [11,13].

Locking Strategy

Early in the database design, it is important to identify the locking necessary for the application. The easiest choice is to use database locking. Unless specific entries are going to be modified by many users, database locking should work. Remember: locking is only needed when updating, adding, or deleting entries from the database, not when reading entries. Never leave the database locked when interacting with the terminal user.

The next level of locking to be considered is dataset locking. This takes more programming, but provides for a more flexible locking strategy.

Rule: *Avoid MR capability; instead, use lock descriptors (and a single call to DBLOCK) to lock all datasets needed.*

For large and complex systems (e.g., an inventory system with inventory levels that must be continually updated), record locking should be used. The database design should help the application programmer by making the easiest possible locking strategy available for each program [2].

Passwords

Most application systems go overboard in their use of database passwords. The simplest scheme to implement is a two-password system. The database is declared with one password for reading and one for writing. Each password is applied at the dataset level; and item-level passwords are not used.

Rule: *Use the simplest password scheme that does not violate the database integrity.*

The advantages to this scheme are that there are fewer passwords to remember, IMAGE is more efficient (because all security checks are done at the dataset level, instead of the data item level), and the user can still use tools such as QUERY, by being allowed the read-only password.

In sensitive applications, a separate dataset or database can be used to isolate data requiring special security. This still permits the simplest password scheme possible, with an extra level of security. The following example shows how to declare passwords for read-only access and read/write access on a dataset level:

```
PASSWORDS:      1 READER;
                2 WRITER;
```

The declarations for the M-CUSTOMER and the D-SALES datasets contain "(1/2)" on the line that declares the name of the datasets. The "(1/2)" indicates that the READER and WRITER passwords are in effect for the whole dataset.

Early Database Testing

The early database design should allow the user or analyst to experiment on the database design with test data. User tools such as QUERY or AQ should be used to access the database. At this stage, the item types may be left approximate, so long as the user or analyst gets a chance to interact with the database design. The analyst should check that all requirements of the user can be met by the database design.

Rule: *Build your test databases early. Use an application tool to verify that the database design is correct.*

In some cases, the end user may not be able to access the database, but the database designer must go through this testing process. This examination of the database design may uncover design flaws which can be fixed easily at this early stage. After the logical database design has been roughly packaged as an actual IMAGE database and verified against the user requirements, the design should be optimized and the finishing touches added (see next section).

Very Complex Databases

IMAGE has a number of size restrictions that it imposes on the database design. For example, the number of items in a database is limited to 255, and the number of datasets in a database is limited to 99. For many applications, these limits pose no problems; but with the larger databases being designed today, it is not difficult to imagine databases which exceed these limits. What can you do to get around this problem?

Bottom-Up Design

The design method outlined above must be extended. For small projects, it is adequate to simply group related data items into datasets, because the entire application will fit into one database. However, for large projects, another step is required: related datasets must be grouped into separate databases.

Multiple databases introduce new problems for the application programmer. These include larger programs, which result in larger data stacks, as well as problems with locking. In designing a multiple database system, it is best to minimize the number of programs that must use more than one database.

If an application decomposes into independent subunits, few programs will require more than one database. The design of the system and the database may have to be revised to increase the independence of the sub-systems.

Polishing Database Design

The database designer has two main concerns in completing the database design. Will the application programs be able to access the database within the defined limits of the HP 3000? Does the database take best advantage of COBOL and other tools available [3,8,11,13]?

Overall Performance

Rule: *Estimate on-line response times and elapsed times for batch jobs. If the project will need more hardware, it is better to know so before the project goes into production.*

The following material is taken from [9], with comments and examples to expand on the original. The HP 3000 is able to perform approximately 30 I/Os per second. On various machines under different operating systems, it may be possible to obtain more than this. Because it is extremely difficult to obtain the theoretical maximum of 30 I/Os per second, it is best to plan for a maximum of 20 I/Os per second.

Each IMAGE procedure results in a specific amount of I/O. Before going ahead with a large application, the total I/O required for the application must be computed and compared against the maximum. This is done by estimating the I/O for each on-line function, then summing the I/Os of the functions that might reasonably occur concurrently. Also, the total elapsed

time for batch jobs must be estimated to ensure that they will complete in the time available.

The following gives an approximate measure of the number of I/Os necessary for each IMAGE procedure in an on-line environment:

Procedure	I/O
DBGET	1
DBFIND	1
DBLOCK	0
DBUNLOCK	0
DBUPDATE	1
DBPUT	2 + 2 * Number of keys in the dataset.
DBDELETE	2 + 2 * Number of keys in the dataset.

The figures for DBPUT and DBDELETE do not take into account sorted chains. If sorted chains are kept short, the above figures will work. If sorted chains are long, the following formula gives an approximate measure of how many I/Os are required to add records in random fashion to a sorted chain:

2 + 2 * (number of keys) + (average chain length)/2

All of the above figures for the number of I/Os for each IMAGE procedure are the same in batch, with one exception. If a batch program reads a dataset serially, the I/Os required will be:

Serial DBGET I/Os = (number of records)/(blocking factor)

If the batch program also does a sort of all of the selected records from the serial DBGET, the number of I/Os will be increased.

Batch Calculation Example

The following example computes how long a specific batch program will take to run; the program makes the following IMAGE calls:

```
125,000 DBGETs serial; blocking factor is 5.
 80,000 DBPUTs to a detail dataset with two keys.
 80,000 DBFINDs.
 80,000 DBGETs to the dataset with the DBFIND.
 80,000 DBUPDATEs.

Total I/Os required =

I/Os for DBGET  (205,000 / 5) plus
I/Os for DBFIND ( 80,000 X 1) plus
I/Os for DBPUT  ( 80,000 X 6) plus
I/Os for DBUPDATE(80,000 X 1).

equals   681,000 I/Os.

We can do approximately 20 I/Os a second so

     681,000
     -------    =  34,050 seconds  =  9.5 hours
        20
```

This batch program is intended to run overnight, but is unlikely to finish in one evening, because time is also needed for backup and other daily functions.

Improving Performance

How can the total time of this program example be reduced to 3.9 hours? One way is to replace the DBPUT with a DBUPDATE. In many instances it is possible, through changes in the application and database design, to use a DBUPDATE instead of a DBPUT. This is especially true in environments where there are recurring monthly charges, which change only slowly over time.

There is another advantage to using DBUPDATE. For each DBPUT, a record is added to the database, and this record must later be deleted using DBDELETE. Because it takes as long to delete the record as it did to add it in the first place, the DBUPDATE can provide as much as an eight-fold decrease in running time, compared with DBPUT/DBDELETE.

COBOL Compatibility

When designing a database, keep in mind how the database is going to be used (COBOL, QUERY, AQ, PROTOS, etc.). The following rules apply to item types and should be used throughout the database.

Numeric Fields

When the database was first designed, all fields were initially declared as type X (display). By now you should know the likely maximum value for each data item. Once the size of each data item is fixed, the time has come to specify a more efficient data type for numeric fields.

The type of field used for numeric values depends on the maximum size of the number to be stored (i.e., the number of digits, ignoring the sign). The following table should be used in determining numeric types:

Number of Digits	IMAGE Data type
<5	J1
<10	J2
>=10	Packed-decimal of the appropiate size.

Rule: *For numeric fields, use J1 for fewer than five digits; use J2 for fewer than ten digits; otherwise, use a P-field (packed-decimal) of the appropriate size.*

In COBOL, an S9(2)V9(2) COMP variable is considered to have a size of 4, or J1. The one exception to this rule is sort fields. All sort fields must be type X. If a numeric sort field is required, it must be declared as type X and redefined as zoned in all COBOL programs. Remember that packed fields in IMAGE are always declared one digit larger than the corresponding COBOL picture (S9(11) COMP-3 becomes P12) and must be allocated in multiples of four.

COBOL databases must not contain R-fields, because R-fields have no meaning in the COBOL language. Instead of an R-type field, a J-type or P-type field must be used. The STORE database contains an R-type field, CREDIT-RATING, which should have been declared as:

```
CREDIT-RATING,          J2;     << Customer credit rating.  The larger
                                   the number, the better the customer's
                                   credit.  Used to five decimal places.
                                >>
```

Key Types

Every key, whether in a master or detail dataset, must be hashed to obtain the actual data associated with the key value. Hashing is a method where a key value, such as customer number 100, is turned into an address. The method used tries to generate a different address for every key value, but in practice this is never possible. The choice of the type of key has a large bearing on how well the hashing function will work.

Rule: *Type X, type U, and type Z keys give the best hashing results, especially if the key length is greater than 6 bytes. Avoid keys of type I, J, K, P, and R.*

When using a Z-type for a key, leave it as unsigned in all COBOL programs. Because key values rarely have negative values, there is no effect on the application by removing the sign from a zoned field. The advantages to leaving off the sign are: 1) displaying the field in COBOL or QUERY results in a more "natural" number, and 2) problems between positive, signed, and unsigned zoned numbers are avoided.

Date Fields

Rule: *Dates should be stored as J2 (S9(6) COMP) in YYMMDD format.*

This format provides the fastest access time in COBOL and takes the least amount of storage. Use a standard date-editing routine to convert from internal to external format and vice versa [4].

The only exception to this is when a detail chain must be sorted by a date field. Because IMAGE does not allow sorting on J2 fields, X6 is used. For the chain to be sorted correctly, the date must still be stored in YYMMDD format.

Other Item Types

The only item types that should be used are J- or P-types for numeric values, and X-, U-, or Z-types for keys. The K-, I-, and R-types should never be used in a commercial application where COBOL is the primary development language.

Example

Earlier, in the discussion of logical database design, four items were described: CUST-STATUS, DELIV-DATE, ON-HAND-QTY, and PRODUCT-PRICE. The following example gives the actual IMAGE declaration for each of these items, according to the rules of this section:

```
CUST-STATUS,            X2;     << Defined state of a particular customer
                                   account.  The valid states are:
                                   10 = advance
                                   20 = current
                                   30 = arrears
                                   40 = inactive
                                >>
DELIV-DATE,             J2;     << Promised delivery date.
                                >>
ON-HAND-QTY,            J2;     << Amount of a specific product currently
                                   onhand.  Only updated upon
                                   confirmation of an order.
                                >>
PRODUCT-PRICE,          J2;     << Individual product price, including
                                   two decimal points.
                                >>

Primary Paths
```

Rule: *Assign a primary path to every detail dataset (select the most frequently accessed path with more than one entry per chain).*

IMAGE organizes the database so that accesses along the primary path are more efficient than along other paths. The primary path should be the path that is accessed most often in the dataset.

If there is only one path in a detail dataset, it must be the primary path. If there are two paths that are accessed equally often, but one is used mostly in online programs and the other mostly in batch programs, assign the primary path to the one that is used in online programs. A primary path is indicated by an exclamation point (!) before the dataset name that defines the path. A path with only one entry per chain should not be selected as a primary path.

The Schema

The IMAGE schema is the method by which you tell both IMAGE and the programmers what the database looks like. The schema should be designed with maximum clarity for the programmer, because IMAGE is only partly concerned with the schema's layout.

Rule: *The schema file name is always XXXXXXSC, where XXXXXX is the name of the database.*

This naming convention makes locating the schema easier for all staff. The file is always located in the same group and account as the database. If the database name was STORE and the STORE database was built in the DB group of the USER account, the schema name would be STORESC.DB.USER.

Layout

A clear layout of the schema makes the programmer's job easier. Some requirements of the layout are imposed by IMAGE, but there are still a number of things that the database designer can do to make the schema more understandable.

Every database schema should start with a $CONTROL line. The $CONTROL line must always

contain the TABLE and BLOCKMAX parameters. The default BLOCKMAX size of 512 should always be used when first implementing the database. Later, after careful consideration, the BLOCKMAX size may be changed. When first designing the database, CONTROL NOROOT should be used.

The CONTROL line should be followed by the name of the database. This is followed by a header comment. This comment describes the designer of the database, the date, the conventions used in designing the schema, abbreviations that are used within IMAGE names, and sub-systems with which the database is compatible and incompatible.

The following are the opening lines of the example STORE database:

```
$CONTROL LIST,NOROOT
BEGIN

DATA BASE   STORE;

<<                  STORE DATABASE FROM THE IMAGE MANUAL

AUTHOR:     DAVID J. GREER, ROBELLE CONSULTING LTD.

DATE:       SEPTEMBER 1, 1981

CONVENTIONS:
```

This schema is organized in alphabetic order. All master datasets are listed before detail datasets, and automatic masters come before manual master datasets.

All dates are stored as J2, YYMMDD, except where they are used as sort fields. If a date is a sort field, it is stored as X6, YYMMDD.

The following abbreviations are used throughout the schema:

NO = Number
CUST = Customer
QTY = Quantity

This database can be accessed by COBOL, QUERY, AQ and PROTOS. Note that the STREET-ADDRESS field is incompatible with QUERY, but AQ can correctly add and modify the STREET-ADDRESS field.

Naming of Items and Sets

Rule: *Names should be restricted to 15 characters. The only special character allowed in names is the dash (-). This ensures that the names are compatible with VIPLUS and COBOL.*

The percent sign (%) should be replaced with the abbreviation "-PCT", and the hash sign (#) should be replaced with the abbreviation "-NO".

Item Layout

The easiest layout to implement, maintain and understand is to declare everything in the database sorted in alphabetic order. The items in the database should begin with a $PAGE command to separate the items from the header comment. Each item appears sorted by its name, regardless of the item's type or function.

In many IMAGE applications, the schema also acts as the data dictionary. For this reason, it is very important that every part of the database design be completely documented in the schema. Document each item as it is declared. To make each item stand out, the following layout should be used:

```
CUST-NO,              Z10;     << The customer number is used as a
                                  key field in the M-CUSTOMER dataset.
                                  It is also the defining path in
                                  the D-ORDER-DETAIL dataset.
```

The item name, its type, and the comment start in the same column for every item. Each part of the item definition will stand out, and because the item names are in sorted order, the applications programmer can easily find a particular item.

Dataset Layout

Every dataset declaration must be preceded by a header comment that describes the use of the dataset and any special facts that the programmer should be aware of.

When accessing the dataset from a COBOL program, it will be necessary to have a COBOL record which corresponds to the dataset. In order to prevent confusion between two occurrences of the same item as a field in several datasets, a prefix will be assigned to each of the variables in the COBOL buffer declaration. This prefix is selected by the database designer and must appear on the same line as the name of the database. For example:

```
<<  The M-CUSTOMER dataset gathers all of the static information
    about each customer into one dataset.  A customer must exist
    in this dataset before any sales are permitted to the
    customer.  This dataset also provides the necessary path
    into the D-SALES dataset.
>>

     NAME:   M-CUSTOMER,              MANUAL (1/2);  <<PREFIX=MCS>>
```

The AUTOMATIC, MANUAL or DETAIL keyword must always appear in the same column. This makes reading the schema easier, and by searching the file for a string (by using /L"NAME:" in QEDIT) it is possible to produce a nice index of dataset names, their types, and their prefixes. The following example prints an index of the STORE dataset names:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORESC.DB "NAME:"
NAME:  M-CUSTOMER,       MANUAL (1/2);      <<PREFIX = MCS>>
NAME:  M-PRODUCT,        MANUAL (1/2);      <<PREFIX = MPR>>
NAME:  M-SUPPLIER,       MANUAL (1/2);      <<PREFIX = MSU>>
NAME:  D-INVENTORY,      DETAIL (1/2);      <<PREFIX = DIN>>
NAME:  D-SALES,          DETAIL (1/2);      <<PREFIX = DSA>>
```

Rule: *Automatic master datasets have names that start with "A-".*

They must be declared immediately after the item declarations, separated from item declarations by a $PAGE command, and they must appear in alphabetic order.

Rule: *Manual master datasets have names that start with "M-".*

The manual master datasets follow the automatic master datasets, again preceded by a $PAGE command. Like the automatic masters, the manual master datasets must be declared in alphabetic sequence.

Rule: *Detail dataset names start with "D-".*

The detail datasets follow the manual master datasets, and the two are separated by a $PAGE command. The detail datasets also appear in alphabetic order.

Field Layout

Without exception, the fields in every dataset must be declared sorted alphabetically. There is a strong tendency to try to declare the fields within a dataset in some other type of logical grouping. Because this logical grouping exists only in the mind of the database designer and cannot be explicitly represented in IMAGE, it should never be used. By declaring fields in sorted order, the applications programmer can work much faster with the database, since no time has to be spent searching for fields within each dataset.

The database designer can still group fields together in a dataset by starting each field with the same prefix. If a dataset contains a group of costs, they might be called VAR-COSTS, FIX-COSTS and TOT-COSTS. To group these items together in the dataset, call them COST-VAR, COSTS-FIX and COSTS-TOT. This maintains the sorted field order in each dataset, while allowing for logical grouping of fields.

Most datasets contain one or more key fields. A key field is specified by following it with (). Because the () pair is sometimes hard to see, a comment should be included beside every key field, indicating that the field is a key. In a detail dataset, the primary key should include a comment to that effect. The following example shows how to declare the fields in a dataset:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE
>>

NAME:  D-SALES,          DETAIL (1/2);        <<PREFIX = DSA>>

ENTRY:
       CUST-ACCOUNT(!M-CUSTOMER)   <<KEY FIELD, PRIMARY PATH>>
      ,DELIV-DATE
      ,PRODUCT-NO(M-PRODUCT)       <<KEY FIELD>>
      ,PRODUCT-PRICE
      ,PURCH-DATE
      ,SALES-QTY
      ,SALES-TAX
      ,SALES-TOTAL
      ;
CAPACITY:  600;      <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Capacities

Analysis of the data flow of the application should result in an approximate capacity for each dataset.

Rule: *The capacity of master datasets should be a prime number.*

To see if a number is prime :RUN the PRIME program contributed by Alfredo Rego. Master datasets should never be more than 80% full (see DBLOADNG below, under "Database Maintenance"), and detail datasets should never be more than 90% full.

The line with the capacity must be formatted in the following way:

```
CAPACITY:   211;      <<M-CUSTOMER,PRIME; ESTIMATED>>
```

The comment after the capacity gives a method for determining the approximate capacity of the dataset. Most detail datasets have a capacity that is related to the master datasets having paths into the detail datasets. These relationships should be described in the capacity comment.

By doing a /L"CAPACITY", it is possible to obtain quickly an index of the capacity of each dataset in the schema. Because the capacity is always the last line of each dataset declaration, doing a /L"M-CUSTOMER" will identify the beginning and ending declarations for the M-CUSTOMER dataset. The following example lists the capacity of the datasets in the STORE database:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORESC.DB "CAPACITY:"
CAPACITY:  211;      <<M-CUSTOMER,PRIME; ESTIMATED>>
CAPACITY:  307;      <<M-PRODUCT,PRIME; ESTIMATED>>
CAPACITY:  211;      <<M-SUPPLIER,PRIME; ESTIMATED>>
CAPACITY:  450;      <<D-INVENTORY; 2 * CAP(M-SUPPLIER)>>
CAPACITY:  600;      <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Final Checkout

After the schema is entered into a file, it must be :RUN through the schema processor, and any typing mistakes should be eliminated:

```
:FILE DBSTEXT=STORESC.DB
:FILE DBSLIST;DEV=LP;CCTL
:RUN DBSCHEMA.PUB.SYS;PARM=3
```

The table produced at the end of the schema should be studied. The following anomalies should be checked:

1. Large-capacity master datasets with a blocking fac-

tor less than four (either increase the BLOCKMAX size to 1024, or change the master dataset to a detail dataset with an automatic master dataset).

2. The blocksize is too small (IMAGE optimizes the blocking factor to minimize disc space); use RE-BLOCK of ADAGER to increase the blocking factor. The blocksize of all dataset blocks should be as close to the BLOCKMAX size as possible.

3. Are there more than two paths into a detail dataset? If there are, can some of them be deleted?

Establishing the Programming Context

By using IMAGE, the COBOL programmer's job should be simplified, since all access to the database is done through the well-defined IMAGE procedures. Like most powerful tools, IMAGE (and COBOL) can be abused by the unsuspecting user.

Rule: *Define a standard IMAGE communication area and put this area in the COPYLIB. Use 88-levels for common errors [15,17,11].*

The starting point for using IMAGE is the standard parameter area, which includes the IMAGE status area, the various access modes used, a variable for the database password, and a number of utility variables which are needed when using IMAGE. For example:

```
05  DB-ALL-LIST             PIC X(2) VALUE "@ ".
05  DB-SAME-LIST            PIC X(2) VALUE "* ".
05  DB-NULL-LIST            PIC S9(4) COMP VALUE 0.
05  DB-DUMMY-ARG            PIC S9(4).
05  DB-PASSWORD             PIC X(8).
05  DB-MODE1                PIC S9(4) COMP VALUE 1.
05  DB-GET-KEYED            PIC S9(4) COMP VALUE 7.
05  DB-STATUS-AREA.
    10  DB-COND-WORD        PIC S9(4) COMP.
        88  DB-STAT-OK          VALUE ZEROS.
        88  DB-END-OF-CHAIN     VALUE 15.
        88  DB-BEGIN-OF-CHAIN   VALUE 14.
        88  DB-NO-ENTRY         VALUE 17.
        88  DB-END-FILE         VALUE 11.
        88  DB-BEGIN-FILE       VALUE 10.
    10  DB-STAT2            PIC S9(4) COMP.
    10  DB-STAT3-4          PIC S9(9) COMP.
    10  DB-CHAIN-LENGTH     PIC S9(9) COMP.
        88  DB-EMPTY-CHAIN  VALUE ZEROS.
    10  DB-STAT7-8          PIC S9(9) COMP.
    10  DB-STAT9-10         PIC S9(9) COMP.
```

Rule: *Establish naming standards for all variables associated with IMAGE databases.*

Standard prefixes must be used on all database variables, including the database, dataset, data field and dataset buffer declarations. A suggestion is to start all database variables with "DB-", all dataset names with "DB-SET-", and all database buffer declarations with "DB-BUFFER-". Data field names are prefixed by the special dataset prefix (which the designer established in the schema), so that each field has a unique name. For example:

```
01  DATASET-M-PRODUCT.
    05  DB-SET-M-PRODUCT            PIC X(10) VALUE "M-PRODUCT;".

05  DB-BUFFER-M-PRODUCT.
    10  MPR-PRODUCT-DESC            PIC X(20).
    10  MPR-PRODUCT-NO              PIC 9(8).
```

Field Lists

The selection of the type of field lists depends on the answer to this question: Can your total application be recompiled in a weekend?

Rule: *Use "@" field list if you can recompile in a weekend (prepare a COPYLIB member for each dataset); use "*" field list otherwise and hire a Database Administrator!*

If the answer to the question if "yes", the at ("@") field list and full buffer declarations should be used when accessing the database. This method requires that all dataset buffers be declared and added to the COPYLIB. If a dataset changes, the buffer declaration must be changed in the COPYLIB, and all affected programs must be recompiled. The simplest solution is to recompile the complete application system whenever a dataset changes.

There should be two complete COPYLIBs available for every application. One is for production, and one is for development.

Rule: *Use a test COPYLIB during development. Double-check that all existing programs will recompile and :RUN correctly before moving the new COPYLIB into production!*

When a database is restructured, the buffer declarations are first changed in the development COPYLIB. When the new database is put into production, the development COPYLIB is also moved into production, as well as any programs that required modification or recompilation.

If the application system is so large that it cannot be recompiled in a weekend, it should use partial field lists and the same ("*") field list. This requires that an application program declare a matching field list and buffer area for each dataset that it accesses. The field list declares the minimum subset of the dataset that the application program needs.

Because partial field lists are more expensive at run time, the applications programmer must code a one-time call to DBGET for every dataset that the application program will use. The same ("*") field list is used on all subsequent DBGET calls. Note that this can cause problems if a common subroutine is called that uses one of the same datasets, but with a different field list.

In order to maintain an application with partial field lists, there must be a way to cross reference every

program/dataset relationship. When a dataset changes, the cross reference system is checked to see which programs use the dataset. Each of these programs must be examined to see if it is affected by the change to the dataset. It is not enough to fix the COPYLIB and recompile, since the field declarations are in the individual source files, not in the COPYLIB file.

Dataset Buffers

The database designer assigns a short, unique prefix to each dataset of each database. These prefixes are used in the declaration of the database buffers for the datasets. In addition, dataset buffer declarations must include all 88-level definitions for flags, and sub-definitions for IMAGE fields that are logically subdivided within the application.

The following is the full buffer declaration for the M-CUSTOMER dataset of the STORE database. Note that each variable is prefixed with "MCS-", which is the prefix that was assigned by the database designer.

```
01  DB-BUFFER-M-CUSTOMER.
    05  MCS-CITY                    PIC X(12).
    05  MCS-CREDIT-RATING           PIC S9(4)V9(5) COMP.
    05  MCS-CUST-ACCOUNT            PIC 9(10).
    05  MCS-CUST-STATUS             PIC X(2).
        88  MCS-CUST-ADVANCE        VALUE "10".
        88  MCS-CUST-CURRENT        VALUE "20".
        88  MCS-CUST-ARREARS        VALUE "30".
        88  MCS-CUST-INACTIVE       VALUE "40".
    05  MCS-NAME-FIRST              PIC X(10).
    05  MCS-NAME-LAST               PIC X(16).
    05  MCS-STATE-CODE              PIC X(2).
    05  MCS-STREET-ADDRESS          PIC X(25) OCCURS 2.
    05  MCS-ZIP-CODE.
        10  MCS-ZIP-CODE-1          PIC X(3).
        10  MCS-ZIP-CODE-2          PIC X(3).
```

Repeated items should be declared with an occurs clause, or sub-divided, whichever the application requires. For example, a cost field may be declared as a repeated item representing fixed, variable, overhead, and labour costs. Rather than declare the costs field as a repeated item in the actual buffer declaration, sub-divide it into the four costs. For example, assume a declaration for costs such as:

```
COSTS,          4J2;    <<Cost of an item.  Each cost has two
                          decimal points and the cost item
                          is broken down as follows:
                          COSTS(1) = Variable costs
                          COSTS(2) = Fixed costs
                          COSTS(3) = Overhead costs
                          COSTS(4) = Labour costs
                        >>
```

Assuming that the COSTS field was declared in the D-INVENTORY dataset, which has a prefix of "DIN", the following buffer declaration would be used for the COSTS field:

```
01  DB-BUFFER-D-INVENTORY.
    05  DIN-COSTS.
        10  DIN-VARIABLE-COSTS      PIC S9(7)V9(2) COMP.
        10  DIN-FIXED-COSTS         PIC S9(7)V9(2) COMP.
        10  DIN-OVERHEAD-COSTS      PIC S9(7)V9(2) COMP.
        10  DIN-LABOUR-COSTS        PIC S9(7)V9(2) COMP.
```

Rule: *Prepare sample COBOL calls to IMAGE in source files, with one IMAGE call per file. Make the calls with one parameter per line.*

The sample IMAGE calls should be organized with one parameter per line. When programming, these template IMAGE calls must be copied into the COBOL program and modified with the database name, dataset name, and any other necessary parameters.

General purpose SECTIONS, declared in the COPYLIB, should NOT be used for the IMAGE calls. These SECTIONS obscure the meaning of the COBOL code. In addition, they can cause unnecessary branches across segment boundaries.

A scheme for handling fatal IMAGE errors must be declared, and the sample IMAGE calls should refer to the fatal-error section. Here is a sample call to the IMAGE routine DBFIND:

```
CALL "DBFIND" USING DB-
                    DB-SET-
                    DB-MODE1
                    DB-STATUS-AREA
                    DB-KEY-
                    DB-ARG-
    IF NOT DB-STAT-OK AND NOT DB-NO-ENTRY THEN
        PERFORM 99-FATAL-ERROR.
```

The fatal-error section (99-FATAL-ERROR) should call DBEXPLAIN. It should also cause the program to abort, and the system job-control word should be set to a fatal state. Note that just using STOP RUN will not set the system job-control word to a fatal state. The following is an example of a fatal-error section. The routine MISQUIT calls the QUIT intrinsic, which causes the program to abort.

```
$PAGE "[99]  FATAL ERROR"
*****************************************************************
*  THIS SECTION DOES THE FOLLOWING:                             *
*  1.  CALLS DBEXPLAIN WITH THE IMAGE STATUS AREA.              *
*  2.  CALLS MISQUIT TO ABORT THE PROGRAM.                      *
*                                                               *
*  NOTE:  THIS MODULE MUST ONLY BE CALLED AFTER A FATAL ERROR   *
*         HAS OCCURED WHEN CALLING AN IMAGE ROUTINE.            *
*                                                               *
*****************************************************************

 99-FATAL-ERROR                 SECTION.

     CALL "DBEXPLAIN" USING DB-STATUS-AREA.

     CALL "MISQUIT" USING DB-COND-WORD.

 99-FATAL-ERROR-EXIT.  EXIT.
```

Rule: *Avoid tricky data structures, especially if they cannot be easily retrieved and displayed with the available tools (QUERY, AQ, PROTOS, QUIZ, ASK, SUPRTOOL, etc.).*

Some examples of data structures to avoid: 1) julian dates; 2) bit maps; 3) alternate record structures (REDEFINEs); 4) implied and composite keys/paths; and 5) implied description structures. The more complex the database structure, the more likely it is that programming or system errors will be created as a result of the database design.

Rule: *Establish house standards for DBOPEN modes (e.g., 1 and 5, or 4 and 6, etc.).*

Rule: *Isolate the code for getting passwords in a single SL routine.*

The easiest way to implement these two rules is to have one subroutine which obtains the database password, and opens the database for either read or read/write access. The passwords can be obtained by prompting the user, by looking them up in a special database, or by hard-coding the name into the routine.

Whatever method is used, it should be easy to change the database password without changing ANY application program. The open subroutine should be the only code which should need to be changed by a change in the database password.

Rule: *If any dataset will contain more than 50,000 entries, get SUPRTOOL/Robelle for fast extracts.*

SUPRTOOL/Robelle is a high-speed utility program that can selectively extract entries from an IMAGE database (or KSAM file or MPE file) at speeds up to 10 or 15 times faster than a COBOL program calling DBGET.

Database Maintenance

There are a number of steps that the database administrator must take in order to guarantee that a database remains clean after it is implemented. A number of standard programs must be run against each production database at least once a month; others must be run daily.

Backup

A number of other people have commented on the backup problem of databases [12], but the problem is important enough to deserve comment again. Most HP 3000 shops do a full backup once a week and a partial backup once a day. This is normally sufficient for most purposes (e.g., source files, PUB.SYS, utilities), but it is not adequate for most IMAGE applications. An IMAGE database consists of several interrelated files. A database that is missing one dataset is nearly useless.

Rule: *EVERY backup tape should include ALL of the files of ALL of the databases that are used in day-to-day applications. If you use IMAGE logging, backup with DBSTORE.*

There should be an easy way to store complete databases onto partial backup tapes, without having to do selective stores. The BACKUP program (available from the San Antonio Swap Tape) helps solve this problem. The BACKUP program is run once a day against every production database. It accepts the database name as input and causes the last-modified date to be changed to today's date on every file of the database. This causes the entire database to be included on the daily partial backup.

In addition, the BACKUP program prints a listing with the following information: the dataset name, the current number of entries in the dataset, and the capacity of the dataset. Further, the BACKUP program examines the relationship between the number of entries and the capacity of each dataset, and prints a warning if it thinks the capacity is too small. This listing must be checked daily, in order to have time to expand the capacity of a dataset before it is exceeded.

Measuring Database Performance (DBLOADNG/HowMessy)

The performance of a given database will change as the database matures.

Rule: *The performance of every application database should be measured at least once a month.*

There is one program that will measure, in great detail, the performance of an IMAGE database. This program is DBLOADNG [1,12], and it is available from the HPIUG contributed library.

DBLOADNG examines the performance of both master and detail datasets, and reports a large number of statistics. The most important are the percentage of secondaries in master datasets, and the elongation of detail datasets.

If there are a large number of secondaries in master datasets, either the hashing algorithm is not working well, or the capacity of the dataset needs to be increased. Note that the hashing performance of a key, such as customer number, can be improved by adding a check digit to every customer number.

The "elongation" of a detail dataset indicates whether logically related records are being stored physically adjacent. For primary paths, the elongation factor should be very small (1 = perfect), since IMAGE tries to place records of a primary-path in the same disc block (see the DBLOADNG documentation and [1]).

If the performance of detail datasets is very poor because logically related records have been spread around the disc, there is only one solution: RELOAD the database using DBUNLOAD/DBLOAD. This will cause the detail dataset to be organized along the

primary path, and could result in significant performance improvements.

A very fast version of DBLOADNG is available from Robelle Consulting Ltd. This version is called HowMessy ("HowMessy are things inside your database?"). The HowMessy program produces the same report as DBLOADNG, but with reductions in CPU and elapsed time of up to 90%. The HowMessy documentation is more detailed than the documentation available for DBLOADNG, so that performance problems with your databases can be isolated and repaired.

Logical Database Maintenance

During the design phase of an IMAGE database, many logical assumptions are made about the data in the database. Some assumptions might be: 1) status fields, which are two characters long in a detail dataset, but have a long description in a master dataset; 2) keys that are stored in detail datasets, but do not have an explicit path into a master dataset; and 3) IMAGE chains that are limited to a specific length (e.g., one address per customer) or a range of lengths (e.g., no more than 10 items per order).

Rule: *When designing a database, keep a list of logical assumptions (such as the maximum and minimum chain lengths for each path).*

These assumptions are dangerous, because they must be maintained by the application software, not by IMAGE.

Rule: *A program to check logical assumptions should be implemented for every application system.*

This program is often called DBREPORT, and its purpose is to check these logical assumptions. DBREPORT is often left until last, and often never implemented. This is unfortunate, since the DBREPORT program is the most important program in an application system.

In Alfredo Rego's paper, DATABASE THERAPY: A practitioner's experiences [12], he describes periodic checkups for a database. The following is taken from his paper:

> *Please notice that a good diagnosis system must be nasty and sadistic by nature. It has as its primary objective to FIND ERRORS, not to certify a system as being error-free (there is no such system anyway!). A good diagnosis system must also be extremely patient and humble, since it will fail many times. Please keep in mind that there is a psychological inversion in effect here: A good diagnosis system fails if it does not detect any errors. And most of the time it will not detect any errors, since we hope and assume that the entity being tested is reasonably error-free [12].*

The DBREPORT program must be designed with Alfredo's philosophy in mind. It should check EVERY dataset in an application, and it should check EVERY record for logical consistency. This includes simple checks to see that every field in every dataset is within a reasonable limit. Examples of this are status fields that take on values from 1 to 10, but which are implemented as J1. A J1 variable can take on values from -32768 to +32767, which is certainly a larger range than 1 to 10.

The DBREPORT program must check all logical dataset relationships. What happens if every customer record has its address in a detail dataset? If the system crashes while the user is adding a new customer, the address record may not be added. DBREPORT must check for these types of relationships (what will your billing program do when it can't find an address?).

ADAGER

Rule: *If an application system depends on IMAGE, get ADAGER.*

ADAGER provides all of the restructuring facilities necessary to maintain IMAGE databases; these transformations cannot be accomplished with DBLOAD/DBUNLOAD. Without ADAGER, numerous conversion programs must be written.

While DBLOAD/DBUNLOAD can be used for some simple database restructuring, it is prone to err. ADAGER is designed to be friendly to the end user, but, more importantly, ADAGER guides the user through every phase of the database restructuring process.

ADAGER provides a powerful facility, but it can also be misused by the unsuspecting. In order to make ADAGER changes effectively, test them first on a development database. Following changes to the database structure, the application programs must be recompiled (with buffers changed in the development COPYLIB), and each program must be tested against the new database.

Currently, ADAGER cannot be run from batch (at least, not conveniently), nor does it produce a hard-copy audit trail of the changes to a database.

Rule: *ADAGER must be run on a printing terminal or in batch.*

Keep the listing of the ADAGER changes to the test database. Use it to verify that the changes to the production database match exactly the changes to the test database. After changing the production database, move the development COPYLIB into production and recompile all affected programs. File the hard-copy listing of the ADAGER changes and keep it for future reference. Another useful item to produce is to use SCHEMA.IMAGE.REGO to save a copy of the database schema, *prior to the changes* (this is especially important if you use transaction logging).

Because the schema is also used as the data dictionary, it must be modified to indicate the new database design. ADAGER's SCHEMA function can be used to double check that all schema changes were made properly. When modifying the database schema, be sure to apply all of the rules in the Schema section of this paper.

Summary

These practical rules for building, programming, and maintaining IMAGE databases should be kept as an easy reference of things to do, or not to do, when using IMAGE.

Database Design

1. Start your logical database design by naming each data item, then identify what values it can have and where it will be used.

2. If a record is uniquely identified by a single key value, start by making it a master dataset (e.g., customer master record keyed by a unique customer number).

3. If a "natural" master dataset will require on-line retrieval via an alternate key, drop it down to a detail dataset.

4. If an entry can occur several times for the primary key value, store it in a detail dataset.

5. Avoid more than two paths into a detail dataset.

6. Avoid sorted paths, except where entries are added to each chain in sorted order anyway.

7. Avoid MR capability; instead, use lock descriptors (and a single call to DBLOCK) to lock all datasets needed.

8. Use the simplest password scheme that does not violate the database integrity.

9. Build your test databases early. Use an application tool to verify that the database design is correct.

Polishing Database Design

1. Estimate on-line response times and elapsed times for batch jobs. If the project will need more hardware, it is better to know so before the project goes into production.

2. For numeric fields, use J1 for fewer than five digits; use J2 for fewer than ten digits; otherwise, use a P-field (packed-decimal) of the appropriate size.

3. Type X, type U, and type Z keys give the best hashing results, especially if the key length is greater than 6 bytes. Avoid keys of type I, J, K, P, and R.

4. Dates should be stored as J2 (S9(6) COMP) in *yymmdd* format, or as X6 if the field is a sort item.

5. Assign a primary path to every detail dataset (select the most frequently accessed path with more than one entry per chain).

The Schema

1. The schema file name is always *xxxxxx*SC, where *xxxxxx* is the name of the database (*xxxxxx*00 is reserved for the ILR file).

2. Names should be restricted to 15 characters. The only special character allowed in names is the dash (-). This ensures that the names are compatible with V/PLUS and COBOL.

3. Automatic master datasets have names that start with "A-".

4. Manual master datasets have names that start with "M-".

5. Detail dataset names start with "D-".

6. The capacity of master datasets should be a prime number. If you note the dataset name in a comment on the *capacity* line, you can list all lines containing "CAPACITY" for a quick review.

Establishing the Programming Context

1. Define a standard IMAGE communication area and put this area in the COPYLIB. Use 88-levels for common error numbers [15,17,11].

2. Establish naming standards for all variables associated with IMAGE databases (e.g., DB-SET-*xxxx* for dataset names).

3. Use "@" field list if you can recompile in a weekend (prepare a COPYLIB member for each dataset); use "*" field list otherwise and hire a Database Administrator!

4. Use a test COPYLIB during development. Double-check that all existing programs will recompile and :RUN correctly before moving the new COPYLIB into production!

5. Prepare sample COBOL calls to IMAGE in source files - one call per file. Make the calls with one parameter per source line.

6. Avoid tricky data structures, especially if they cannot be easily retrieved and displayed with the available tools (QUERY, AQ, PROTOS, QUIZ, ASK, SUPRTOOL, etc.).

7. Establish house standards for DBOPEN modes (e.g., 1 and 5, or 4 and 6, etc.).

8. Isolate the code for getting passwords in a single SL routine.

9. If any dataset will contain more than 50,000 entries, get SUPRTOOL/Robelle for fast extracts.

Database Maintenance

1. *Every* backup type should include all of the files of all of the databases that are used in day-to-day applications. If you use IMAGE logging, backup with DBSTORE.

2. The performance of every application database should be measured at least once a month. Use either DBLOADNG from the contributed library or How-Messy/Robelle (high-speed version of same report).

3. When designing a database, keep a list of logical assumptions (*e.g.*, minimum and maximum chain lengths for each path).

4. A program to check logical assumptions should be implemented for every application system.

5. If an application system depends on IMAGE, get ADAGER; run it on a printing terminal or in batch, in order to keep an audit trail of all database changes. Use SCHEMA.IMAGE.REGO to save the database schema prior to the changes.

Database Programming

1. Place each IMAGE call near its point of use; do not use generic access SECTIONs.

2. Check for errors after every call to IMAGE; perform an error SECTION for unexpected errors.

3. DBFIND may succeed, even if there are no entries for the value specified, so check the status area for the chain length.

4. Use null field list with a mode-7 get (hashed) to validate existence of key values without retrieving any values.

5. Be careful with all IMAGE calls between a DBGET and the matching DBUPDATE (or DBDELETE). Otherwise, you may update (or delete) the wrong entry if you move the current record pointer.

6. After your DBLOCK, use mode-7 (masters) or mode-4 (details) to DBGET entries again before updating (or deleting) them (do not use mode 1). Otherwise, you may update (or delete) the wrong entry if the entry has moved or been deleted.

Bibliography

To gain a complete understanding of IMAGE, study the references in this bibliography. A suggested order of study is: [6,7,9,10,11] for more ideas on database design, [5] for some hints on common programming errors, and [1,3,8,12,13] for notes on optimizing IMAGE databases and application systems in general. [1] is an excellent introduction to database optimization, and it includes a discussion of the DBLOADNG program.

[1] Rick Bergquist, *Optimizing IMAGE: An Introduction,* HPGSUG 1980 San Jose Proceedings.

[2] Gerald W. Davidson, *Image Locking and Application Design,* Journal of the HPGSUG, Vol. IV, No. 1.

[3] Robert M. Green, *Optimizing On-Line Programs,* Technical Report, second edition, Robelle Consulting Ltd.

[4] Robert M. Green, *SPLAIDS2 Software Package,* contains date editing routine (SUPRDATE) available from Robelle Consulting Ltd.

[5] Robert M. Green, *Common Programming Errors With IMAGE/3000,* Journal of the HPGSUG, Vol I, No. 4.

[6] *IMAGE/3000 Reference Manual,* Hewlett-Packard.

[7] Karl H. Kiefer, *Data Base Design - Polishing Your Image,* HPGSUG 1981 Orlando Proceedings.

[8] Jim Kramer, *Saving the Precious Resource - Disc Accesses,* HPGSUG 1981 Orlando Proceedings.

[9] Ken Lessey, *On Line System Design and Development,* HPBSUG 1981 Orlando Proceedings.

[10] Brian Mullen, *Hiding Data Structures in Program Modules,* HPGSUG 1980 San Jose Proceedings.

[11] Alfredo Rego, *Design and Maintenance Criteria for IMAGE/3000,* Journal of the HPGSUG, Vol. III, No. 4.

[12] Alfredo Rego, *DATABASE THERAPY: A practitioner's experiences,* HPGSUG 1981 Orlando Proceedings.

[13] Bernadette Reiter, *Performance Optimization for IMAGE,* HPGSUG 1980 San Jose Proceedings.

Appendix I - Revised STORE Database

The following is the example STORE database from the IMAGE manual [6]. It has been revised to reflect the rules for layout and naming conventions for schemas. A close comparison between the IMAGE example and this example will show how easy it is to change your current database design to the one suggested in this document.

```
$CONTROL LIST,NOROOT
BEGIN

DATA BASE   STORE;

<<                  STORE DATABASE FROM THE IMAGE MANUAL

AUTHOR:     DAVID J. GREER, ROBELLE CONSULTING LTD.

DATE:       SEPTEMBER 1, 1981

CONVENTIONS:
```

This schema is organized in alphabetic order. All master datasets are listed before detail datasets, and automatic masters come before manual master datasets.

All dates are stored as J2, YYMMDD, except where they are used as sort fields. If a date is a sort field, it is stored as X6, YYMMDD.

The following abbreviations are used throughout the schema:

NO = Number
CUST = Customer
QTY = Quantity

This database can be accessed by COBOL, QUERY, AQ, and PROTOS. Note that the STREET-ADDRESS field is incompatible with QUERY, but AQ can correctly add and modify the STREET-ADDRESS field.

```
>>
PASSWORDS:     1 READER;
               2 WRITER;

$PAGE "STORE - DATA ITEMS"
ITEMS:

BIN-NO,                 J1;     << Actual bin number where each product
                                   is stored.
                                >>
CITY,                   X12;    << City name, in full, of each customer
                                   account.
                                >>
CREDIT-RATING,          J2;     << Customer credit rating.  The larger
                                   the number, the better the customer
                                   credit.  Used to five decimal places.
                                >>
CUST-ACCOUNT,           Z8;     << Customer account number.  This field
                                   is used as a key to the M-CUSTOMER
                                   and D-SALES datasets.
                                >>
CUST-STATUS,            X2;     << Defined state of a particular customer
                                   account.  The valid states are:
                                   10 = advance
                                   20 = current
                                   30 = arrears
                                   40 = inactive
                                >>
DELIV-DATE,             J2;     << Promised delivery date.
                                >>
LAST-SHIP-DATE,         J2;     << Last date that a specific product
                                   was shipped.
                                >>
NAME-FIRST,             X10;    << Customer first name.
                                >>
NAME-LAST,              X16;    << Customer last name.
                                >>
ON-HAND-QTY,            J2;     << Amount of a specific product currently
                                   onhand.  Only updated upon
                                   confirmation of an order.
                                >>
PRODUCT-DESC,           X20;    << Description associated with each
                                   product.  This is never allowed to
                                   equal blank.
                                >>
PRODUCT-NO,             Z8;     << Individual product numbers.  Used as
                                   a key in M-PRODUCT, M-SALES and
                                   M-INVENTORY.
                                >>
PRODUCT-PRICE,          J2;     << Individual product price, including
                                   two decimal points.
                                >>
PURCH-DATE,             J2;     << Date that a specific product was
                                   originally purchased.
                                >>
SALES-QTY,              J1;     << Quantity purchased on specific
                                   sales.
                                >>
SALES-TAX,              J2;     << Computed sales tax, based on the
                                   current rate in D-SYSTEM-CONTROL.
                                   Includes two decimal points.
                                >>
SALES-TOTAL,            J2;     << Total amount of each sale, including
                                   sales tax.
                                >>
STATE-CODE,             X2;     << Two-letter state abbreviation.
                                >>
STREET-ADDRESS,         2X25;   << Customer street number and
                                   address.
                                >>
SUPPLIER-NAME,          X16;    << Name of the supplier of each product.
                                >>
UNIT-COST,              P8;     << The cost per unit of each product.
                                >>
ZIP-CODE,               X6;     << Postal zip code.
                                >>

$PAGE "STORE - MANUAL MASTER DATASETS"
SETS:

<<  The M-CUSTOMER dataset gathers all of the static information
    about each customer into one dataset.  A customer must exist
    in this dataset before any sales are permitted to the
    customer.  This dataset also provides the necessary path
    into the D-SALES dataset.
>>

NAME:  M-CUSTOMER,        MANUAL (1/2);        <<PREFIX = MCS>>
ENTRY:
       CITY
      ,CREDIT-RATING
      ,CUST-ACCOUNT(1)       <<KEY FIELD>>
      ,CUST-STATUS
      ,NAME-FIRST
      ,NAME-LAST
      ,STATE-CODE
      ,STREET-ADDRESS
      ,ZIP-CODE
      ;
CAPACITY:  211;      <<M-CUSTOMER,PRIME; ESTIMATED>>

<<  The M-PRODUCT dataset provides a description for every
    product available for sale.  This dataset also provides the
    necessary paths for inventory levels (D-INVENTORY) and for
    sales records (D-SALES).
>>

NAME:  M-PRODUCT,         MANUAL (1/2);        <<PREFIX = MPR>>
ENTRY:
       PRODUCT-DESC
      ,PRODUCT-NO(2)      <<KEY FIELD>>
      ;
CAPACITY:  307;      <<M-PRODUCT,PRIME; ESTIMATED>>

<<  The M-SUPPLIER dataset provides the static information
    which must be known about every supplier of inventory
    items.  Note that none of the fields of this dataset are
    valid if they are blank.  This dataset provides a path
    into the inventory levels of all of the products of
    an individual supplier (D-INVENTORY).
>>

NAME:  M-SUPPLIER,        MANUAL (1/2);        <<PREFIX = MSU>>
ENTRY:
       CITY
      ,STATE-CODE
      ,STREET-ADDRESS
      ,SUPPLIER-NAME(1)      <<KEY FIELD>>
      ,ZIP-CODE
      ;
CAPACITY:  211;      <<M-SUPPLIER,PRIME; ESTIMATED>>

$PAGE "STORE - DETAIL DATASETS"
<<  The D-INVENTORY dataset keeps a record of the items in
inventory, and the location of each inventory item (BIN-NO).
This dataset also provides a linkage between every product and
each supplier.  Note that it is possible for one product to
be supplied by more than one supplier.  Each product in this
data set must have one, and only one, record.
>>
```

```
NAME:  D-INVENTORY,      DETAIL (1/2);       <<PREFIX = DIN>>
ENTRY:
       BIN-NO
      ,LAST-SHIP-DATE
      ,ON-HAND-QTY
      ,PRODUCT-NO(M-PRODUCT)        <<KEY FIELD>>
      ,SUPPLIER-NAME(!M-SUPPLIER)   <<KEY FIELD,PRIMARY PATH>>
      ,UNIT-COST
      ;
CAPACITY:  450;     <<D-INVENTORY; 2 * CAP(M-SUPPLIER)>>

<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,          DETAIL (1/2);       <<PREFIX = DSA>>
ENTRY:
       CUST-ACCOUNT(!M-CUSTOMER)    <<KEY FIELD, PRIMARY PATH>>
      ,DELIV-DATE
      ,PRODUCT-NO(M-PRODUCT)        <<KEY FIELD>>
      ,PRODUCT-PRICE
      ,PURCH-DATE
      ,SALES-QTY
      ,SALES-TAX
      ,SALES-TOTAL
      ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>

END.     <<DATABASE STORE>>
```

Appendix II - Using Sorted Paths

Introduction

The decision to use sorted paths is largely dependent on the application. Therefore, this discussion can only serve as a general guideline.

There are two main questions which must be addressed before using sorted paths. The first is: will the data on the sorted paths be added during peak hours, or during off hours? The second is: what are the attributes of the data on the sorted path?

The more data that is added to a sorted path during off hours, the more likely it is that the expense of the sorted path can be tolerated. This guideline has to be modified according to the data that must be added (or deleted) from the sorted path.

There are three questions regarding the attributes of data: 1) how much data is to be added; 2) how much data is going to be added for every key value; and 3) is the data already in sorted order when it is added to the sorted path?

The amount of data being added to sorted paths must be minimized. If the sorted path is there only for the ease of programming one or two reports, it is very likely that the COBOL SORT verb should be used, with a sequential search of the dataset instead.

The average chain length of the sorted path must be minimized. If the average is greater than 10, the cost of the adding records to the sorted path could rise exponentially.

The data should be in sorted order before being added to the database. IMAGE optimizes adding already sorted records by checking to see if the record to be added goes after the last record on the chain. If it doesn't, IMAGE searches up the chain until it finds the proper spot for the new entry.

The Costs

There is no disc space cost associated with using sorted paths, except that you must use "X" type instead of "J" as the type of the sort field. The major cost associated with sorted paths is the cost of finding the correct place in the chain for a new record, and the fact that sorted records may be spread over many disc blocks if the data is added in random fashion.

Without specific measurements of user data, it is very difficult to predict the amount of I/O involved in using sorted paths. If the data is added in sorted order, the I/O cost of the sorted path should be approximately the same as for the same path, unsorted. If the data is added in a random fashion, the total I/Os used to add a record to the sorted path is:

$$2 + 2 * (\text{number of keys}) + (\text{average chain length}/2)$$

The best guides to using sorted paths are: 1) keep the volume of data on sorted paths to a minimum; 2) keep the percentage of inserted entries to entries added to the beginning or end of the chain very small; and 3) add or delete records on sorted paths during off hours [8].

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

Techniques for Testing On-Line Interactive Programs

Kim D. Leeper
Wick Hill Associates Ltd.
Kirkland, Washington

Abstract

This paper will describe various strategies for testing on-line interactive programs. These strategies include acceptance/functional testing, regression testing and contention testing. The paper will also discuss the mechanics of testing including testing by human intervention and various forms of automated testing. This information will allow you to create a viable test plan for software quality assurance in your shop.

Introduction

Program Testing. Those two words undoubtedly conjure up thoughts of long boring hours sitting in front of a terminal typing in all kinds of data looking at error messages produced by the program. This paper will present alternatives to this type of program testing. It will also describe a prototype test plan or quality assurance cycle which may provide the reader with ideas for implementing his/her own test plan for his/her own shop.

We must make sure we are all talking the same language so some definitions are in order at this point.

What is testing?

Software testing may be thought of as a series of data items which when presented to the program under test (PUT) cause the software in question to react in a prescribed or expected fashion within its intended environment. The purpose of testing is to expose the existence of mistakes in the program or to show the absence of any such bugs. If the software does not act in the expected way then one has found a bug or mistake in the program.

Vocabulary

SCRIPT - a list of inputs or data items given to the PUT for testing purposes.

DATA CONTEX OF BUG - the collection of inputs required to cause the PUT to fail or return results which are not expected.

Types of Testing

Acceptance/Functional Testing

This type of testing is used to demonstrate that the various functions of a given software package actually work as described in the documentation. This is not exhaustive testing as it only examines one or two transactions per function. This is the typical type of testing the vast majority of users perform now.

Regression Testing

This type of testing can be used to test all the various logical paths within a given software system. Regression testing tries all the data extremes per function that the program could be expected to respond to. This type of testing is rarely performed because it is resource, that is to say hardware and personnel, intensive.

Contention Testing

This type of testing is used to determine if the database or file locking strategies that are used in your application programs actually work. Two programs are executed at the same time, one performs a transaction which locks a given item in the database. The second program attempts to access this same data that is secured by the lock via another transaction type different than the one used in the first terminal. The designer in this instance is interested in the message of action of the software to this challenge. This type of testing becomes particularly relevant when the installation has many programmers implementing many systems dealing with the same database.

The Test Plan or Quality Assurance Cycle

The keystone of any successful testing program is to have a viable test plan. This plan should describe all the phases a software development project goes through and then ties all the phases together in one comprehensive flow of data and actions. The plan should extensively use feedback loops so that when problems are discovered there are clear paths for the problem rectification process to follow. One possible

quality assurance cycle that can be proposed may be seen in Figure 1.

The diagram indicates that the test script should be generated along with the design of the software. Many times in the design process the designer realizes some weakness in the design and will want to specify a special test in the scriptfile. S/he is encouraged to do so. Many companies that use this methodology specify programs by a test script and V/3000 screens.

Examining this diagram more closely one can see that the flow of debugging actions is closely tied to the design/maintenance of the original test script. The reason for this is to force the implementors to keep track of the bugs they discover and place them in the test script. This script should then be run against the application program whenever a new fix or correction has been applied to the original program. This script will constantly force the program to re-execute all the previous transactions which caused bugs to occur in the past, to assure the program maintenance team that no additional mistakes have been introduced by fixing the last bug.

In this version of the QA cycle the users are always in mode of testing the delivered software. Eventually the users will find a bug which will start the whole cyclic process over again. If they don't find a bug, don't think it is not for trying. The users have eight hours per day per person to find bugs. It does not take very long before they have more execution time on the application software than the designer/implementator has. This is the time when more bugs can and will be found which will start the cycle once again.

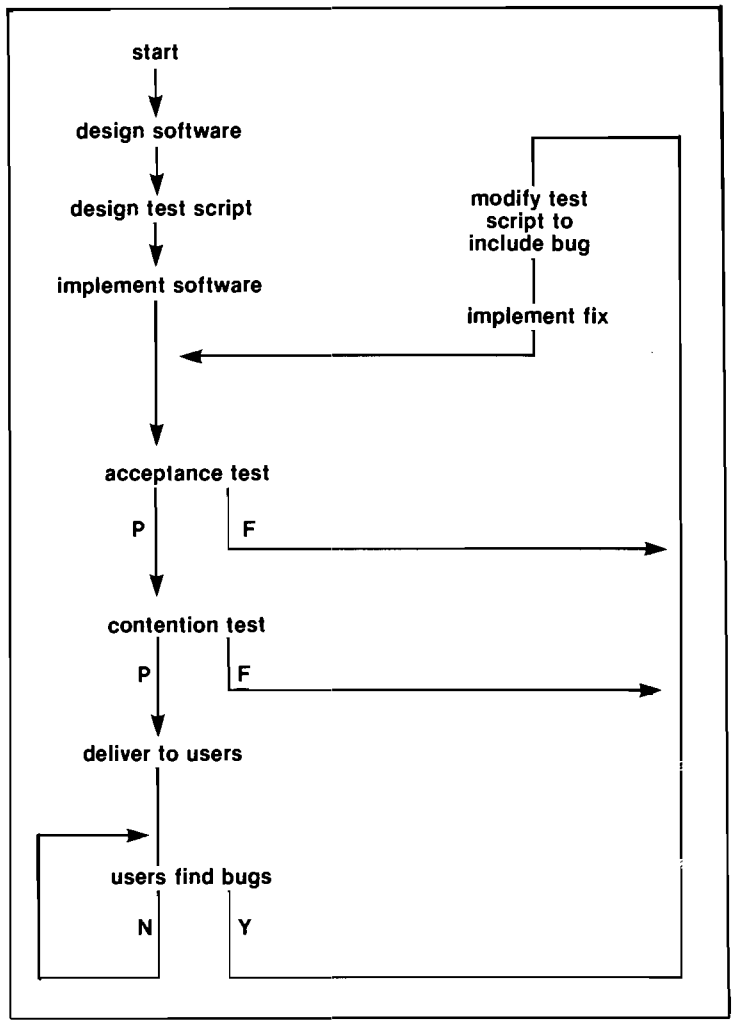


Figure 1: Quality Assurance Software Cycle

The Mechanics Of Testing

Obviously, the type of testing that is currently being used is human intervention testing. This is where a programmer or analyst sits in front of a terminal and simulates a user by following a handwritten script. This approach to testing is less than desirable for a number of reasons, among those being:

1. input data error due to arrogance/boredom in applications tester;

2. non-repeatability of exact timing due to human tester;

3. the tester might not record everything happening off the screen;

4. an expensive employee is being utilized for testing purposes when s/he could be designing/implementing more applications.

A possible solution to the dilemma outlined above is to mechanically examine the software by exhaustively testing all the paths in the program by computer. Using completely random data types as input you could automate the testing process. However, as there is only so much time available during a 24 hour day it might take all day to exhaustively test a very small application program. This technique is machine bound in terms of both creating the random data and testing all the paths in the application code.

A saner approach would be to combine the above two techniques into a testing procedure that utilizes a human being's capacity for creative thought and a machine's capacity for highly efficient repetition. This technique would rest in the programmers designing the scripts used for automated testing at the same time as they design the application itself. Once the test script is produced then the machine itself tests

out the application program under the watchful eye of a human. In fact, the script can be used as a specification for implementing the system. As Yourdon has written, "What we are interested in is the minimum volume of test data that will adequately exercise our program."[1]

It is now possible, using VTEST/3000, to automate this testing procedure and achieve a real manner of quality control. VTEST/3000 includes full V/3000 testing capability. The compiled code runs as though it were in a live situation with VTEST/3000 providing full documentation of all errors occurring on the screen of the terminal.

In order to use VTEST effectively one must appreciate the diagram in Figure 2. There are two types of tests that VTEST can perform, block mode testing for those programs that use V/3000 and non-block mode testing for those not using V.

The first type of testing that will be discussed is non-block mode application testing. In this case VTEST looks like a non-block mode glass TTY terminal. The script file contains the actual commands and data that a user would normally type into the screen of a real terminal, everything between and including HELLO and BYE. This script file is built and maintained by the standard HP EDITOR. The script file is input to VTEST. VTEST transmits this file a line at a time to the application and VTEST prints out a report of the terminal screen before the return key was depressed and after along with the number of seconds that the response took to come back to VTEST.

The second type of testing that will be discussed is block mode application testing. In this case VTEST looks like a HP 2645 block mode terminal. The script file is the same as above with an important extension. The script file now can tell VTEST when it must transmit data to a V screen. The data for a V screen must come from a different type of file. This file is called the BATCH file. BATCH is created and maintained by another program called CRBATCH. CRBATCH allows the user to specify the formfile name and the form to be displayed. Data is then entered and CRBATCH reads the screen and puts the data into a BATCH file. CRBATCH allows the user to insert screens, to delete screens and modify the data in screens already in the BATCH file. It is a general purpose maintenance program or editor for BATCH files. Whenever the application program under test wants some block mode data the next record is read from the BATCH file. VTEST then transmits this record complete with all the special characters that V requires to the application. VTEST prints out a report for every transaction before the ENTER key was depressed and after the next

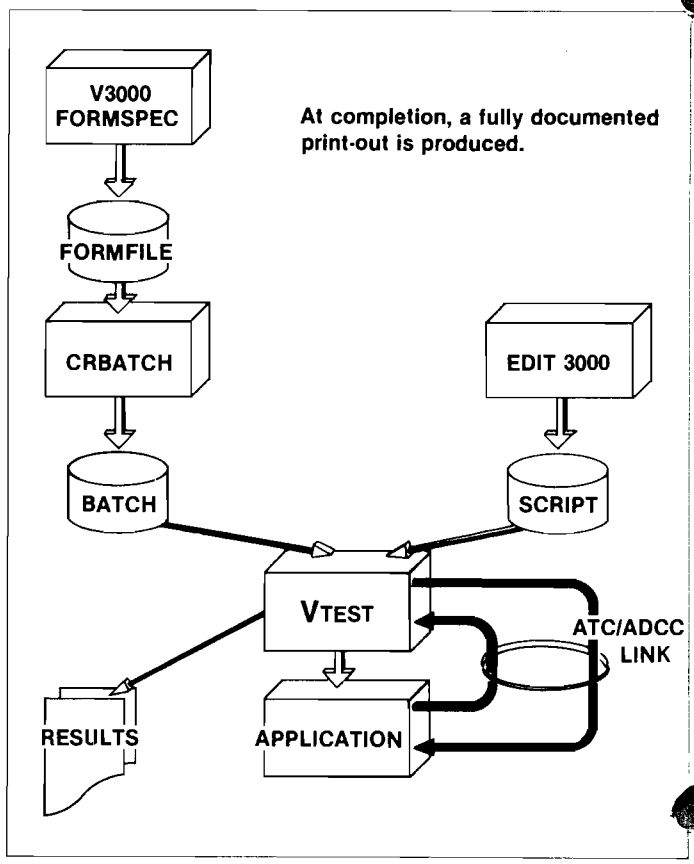


Figure 2

screen was received along with the number of seconds that the response took to come back to VTEST.

One can see quite easily that VTEST fits right into a well designed quality assurance cycle.

References

1. Edward Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, 1975.

2. Software Research Associates, *Testing Techniques Newsletter*, (415) 957-1441.

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

Implementation of Control Structures in FORTRAN/3000

James P. Schwar
Lafayette College
Easton, Pennsylvania 18042

The four recognized control structures available for writing structured code are the IFTHENELSE, DOUNTIL, DOWHILE, and CASE.

In FORTRAN/3000 these control structures can be implemented using the logical IF, unconditional GO TO, DO loop and the computed GO TO.

The IFTHENELSE is of the form IF condition THEN true statements ELSE false statements, where condition is a logical expression. This control structure, as shown in Figure 1, represents a simple decision. In FORTRAN/3000, the IFTHENELSE becomes

```
      IF(.NOT.condition)GO TO S1
      true statements
      GO TO S2
S1    false statements
S2    CONTINUE
```

Consider, for example, the calculation of the real roots of $F(X)=A*X**2+B*X+C=0$ for any A,B,C. The decision to be made is IF $B*B-4.0*A*C>=0$ THEN calculate and output the real roots ELSE output "not two real roots."

```
PAGE 0001  HP32102B.01.04  FORTRAN/3000 (C) HEWLETT-PACKARD CO. 1980

          C     ROOTS OF THE QUADRATIC EQUATION
                DATA A,B,C/1.0,3.0,2.0/
                DISCR=B*B-4.0*A*C
          C     BEGIN IFTHENELSE
                IF(.NOT.DISCR.GE.0.0)GO TO 10
                ROOT1=(-B+SQRT(DISCR))/(2.0*A)
                ROOT2=(-B-SQRT(DISCR))/(2.0*A)
                WRITE(6,*)'REAL ROOTS ARE',ROOT1,ROOT2
                GO TO 20
             10 WRITE(6,*)'NOT TWO REAL ROOTS'
             20 CONTINUE
          C     END   IFTHENELSE
                STOP
                END

PROGRAM UNIT MAIN' COMPILED

****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02

END OF COMPILE
```

One alternative implementation would be to replace NOT.DISCR.GE.0.0 with logical expression DISCR. LT.0.0. A second alternative would be to interchange the position of the true and false statements and use DISCR.GE.0.0 as the logical expression. Neither alternative matches the flowchart logic as well as the implementation that uses the .NOT. condition.

The control structure DO statements WHILE a condition is true is shown in Figure 2. The FORTRAN/3000 implementation becomes

```
S1    IF(.NOT.condition)GO TO S2
         statements
         GO TO S1
S2    CONTINUE
```

Consider the calculation of N!, where

$$N! = N(N-1)(N-2) \ldots (1)$$

This calculation proceeds from left to right WHILE there is a value > 1. Given N, the FORTRAN/3000 statements are

```
PAGE 0001  HP32102B.01.04  FORTRAN/3000 (C) HEWLETT-PACKARD CO. 1980

          C     CALCULATION OF N!
                DATA N/10/
                FACTORIAL=1.0
          C     BEGIN DOWHILE
             10 IF(.NOT.N.GT.1)GO TO 20
                FACTORIAL=FACTORIAL*N
                N=N-1
                GO TO 10
             20 CONTINUE
          C     END   DOWHILE
                WRITE(6,*)'FACTORIAL IS',FACTORIAL
                STOP
                END

PROGRAM UNIT MAIN' COMPILED

****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02


                END OF COMPILE

                END OF PREPARE

                 FACTORIAL IS  .362880E+07

                END OF PROGRAM
```

This code yields 1 for N < = 1.

The control structure DO statements UNTIL a condition is true is shown in Figure 3. In FORTRAN/3000 this control structure becomes

```
      S1 statements
         IF(.NOT.condition)GO TO S1
```

and the preceeding factorial calculation becomes

```
PAGE 0001  HP32102B.01.04  FORTRAN/3000 (C) HEWLETT-PACKARD CO. 1980

          C     CALCULATION OF N!
                DATA N/10/
                FACTORIAL=1.0
          C     BEGIN DOUNTIL
             10 FACTORIAL=FACTORIAL*N
                N=N-1
                IF(.NOT.N.LE.1)GO TO 10
          C     END   DOUNTIL
                WRITE(6,*)'FACTORIAL IS',FACTORIAL
                STOP
                END

PROGRAM UNIT MAIN' COMPILED
```

```
****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02

END OF COMPILE

END OF PREPARE

 FACTORIAL IS  .362880E+07

END OF PROGRAM
```

The DO loop in FORTRAN/3000 which is of the form

```
DO S counter=initial value, final value, increment statements
                                                S CONTINUE
```

also implements the DOUNTIL as shown in Figure 4. The calculation of N!, using the DO loop, becomes

```
PAGE 0001  HP32102B.01.04  FORTRAN/3000 (C) HEWLETT-PACKARD CO. 1980

          C     CALCULATION OF N!
                DATA N/10/
                FACTORIAL=1.0
                DO 10 I=N,2,-1
                FACTORIAL=FACTORIAL*I
             10 CONTINUE
                WRITE(6,*)'FACTORIAL IS',FACTORIAL
                STOP
                END

PROGRAM UNIT MAIN' COMPILED

****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

END OF COMPILE

END OF PREPARE

 FACTORIAL IS  .362880E+07

END OF PROGRAM
```

The DOUNTIL requires that 0! be treated as a special case.

The CASE structure, as shown in Figure 5, offers multiple paths. The selected path is based on the value assigned to an integer variable. As many paths as needed can be specified. It is common practice to assume that when the integer variable is outside its allowable range, e.g., one to four for Figure 5, the CASE structure is ignored. The computed GOTO can be used to implement the CASE structure, as illustrated by the following FORTRAN code.

```
C     J=INTEGER VARIABLE
C     J IS PREVIOUSLY DEFINED
      GO TO (10,20,30,40),J
      GO TO 50
   10 [statements]
      GO TO 50
   20 [statements]

      GO TO 50
   30 [statements]
      GO TO 50
   40 [statements]
   50 CONTINUE
```

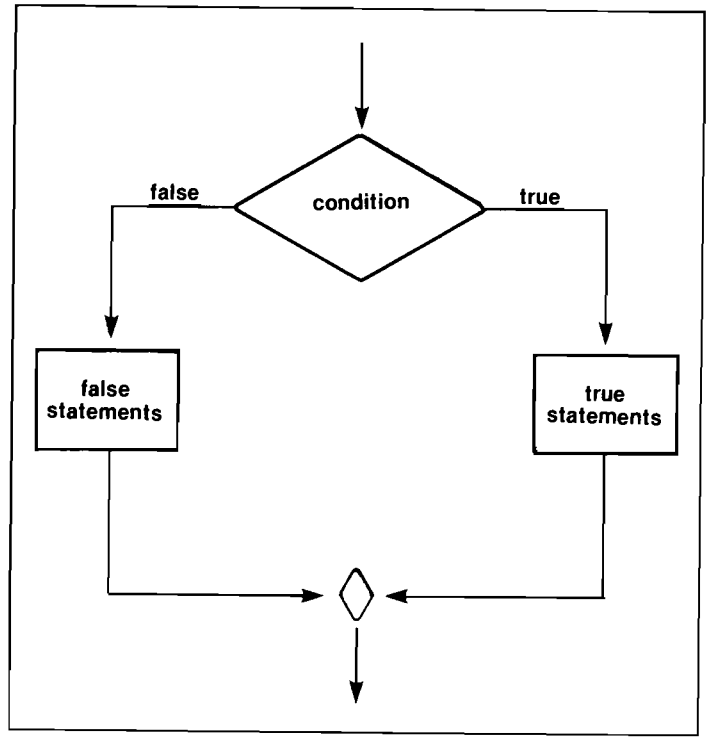


Figure 1. IFTHENELSE Control Structure

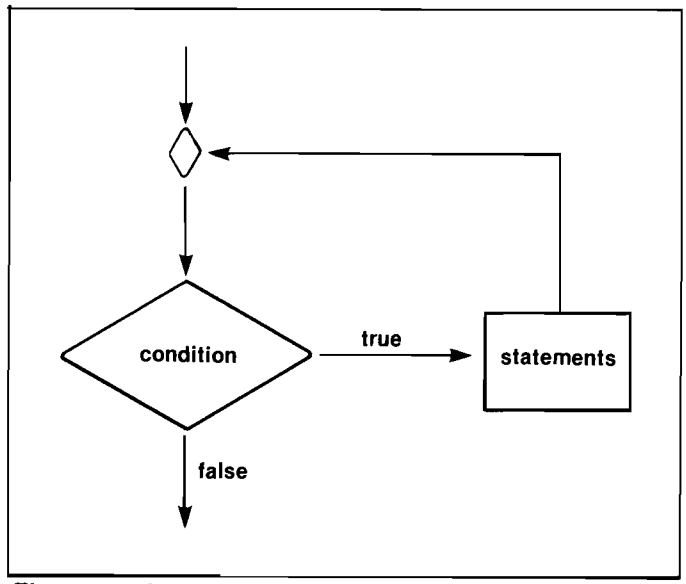


Figure 2. DOWHILE Control Structure

References

FORTRAN/3000 Reference Manual, Hewlett-Packard Company (1977).

Applied FORTRAN for Engineering and Science, James P. Schwar and Charles L. Best, SRA (1982).

Fortran/3000 and Fortran 77: A Comparison, James P. Schwar and Charles L. Best, Journal of HP General Systems Users Group Winter, 1980, pp 14-15.

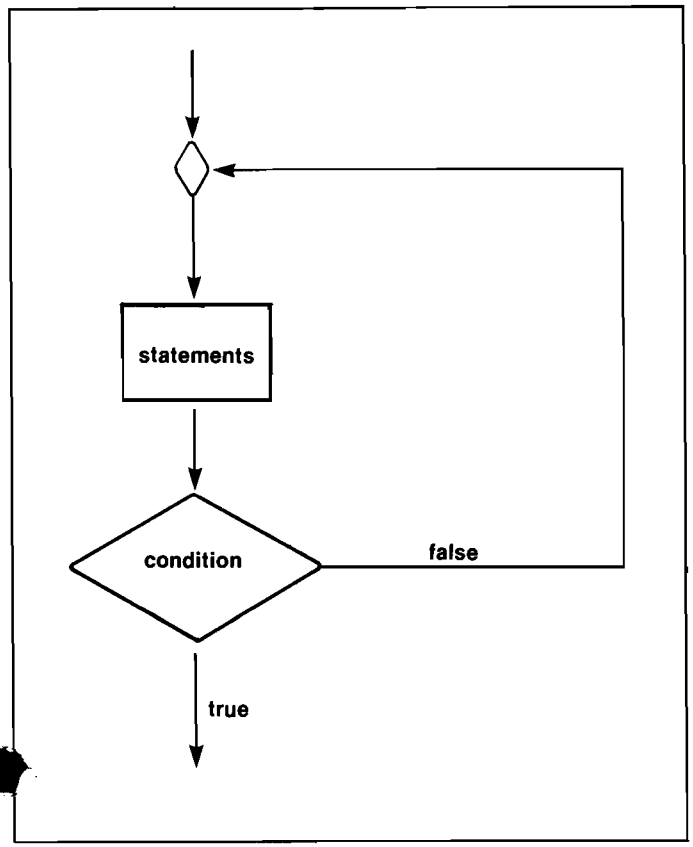


Figure 3. DOUNTIL Control Structure

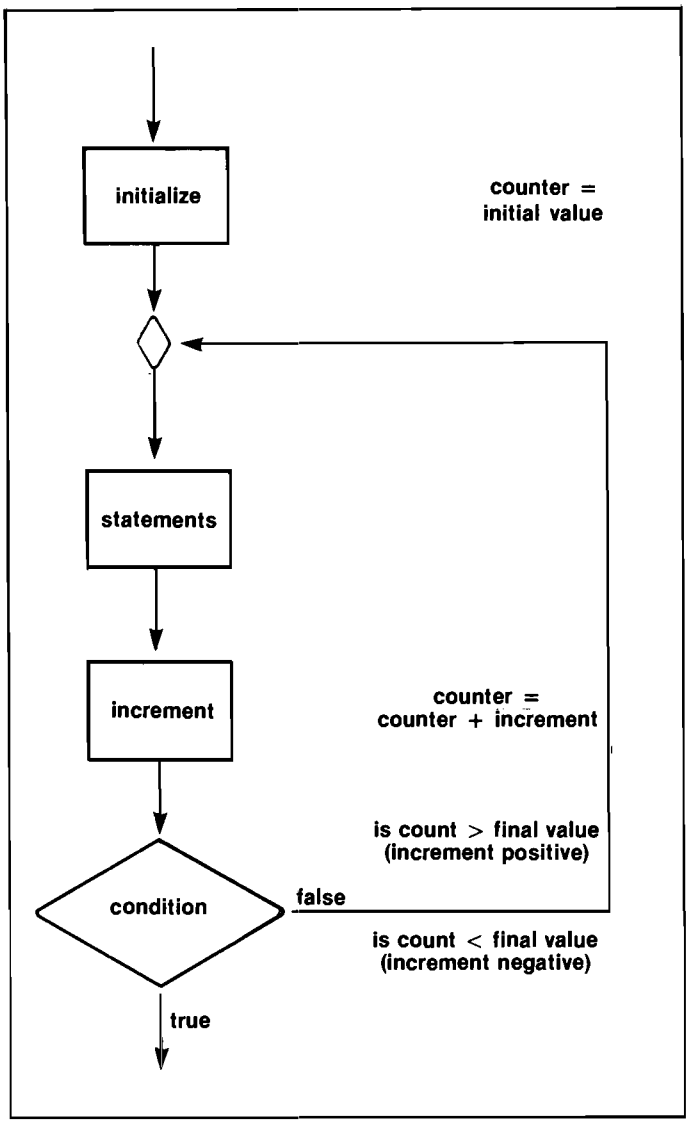


Figure 4. DO Loop

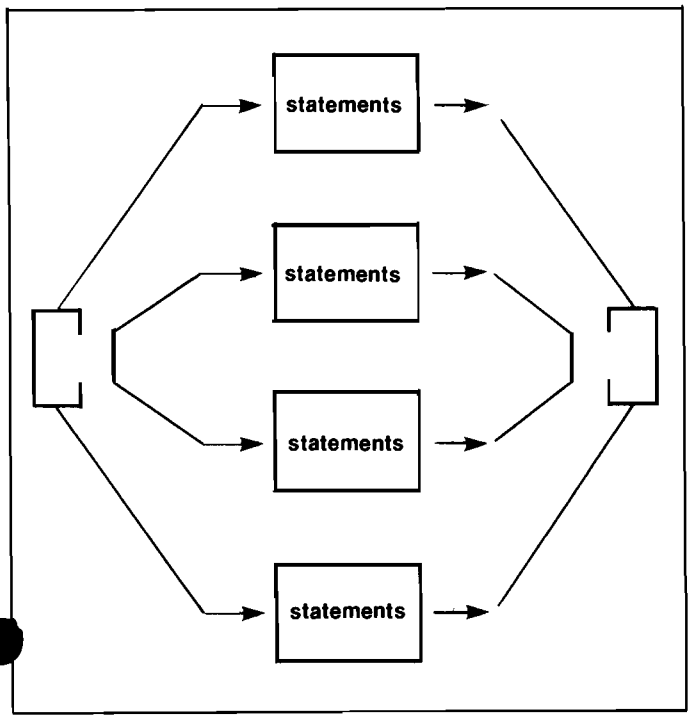


Figure 5. FORTRAN CASE Structure

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

The Performance of Image Reporting Programs

Roger W. Lawson
Harris-Queensway PLC
76 High Street
Orpington, Kent
BR6 0LX, England

Introduction

Most HP 3000 installations use IMAGE extensively. One of the reasons for this is because of the ease of production of reports by the use of QUERY or other report processors. Where QUERY is being used to process large numbers of records its performance can be a significant factor in the load on an HP 3000. This article provides some performance data on QUERY. It also compares it to other products such as the new REPORT and INFORM parts of the RAPID package.

Test Environment

The performance data was obtained by producing various reports from large databases on a HP 3000 Series III of 1 mbyte memory and a HP 44 of 1 mbyte memory. The test reports were selected as representative samples from the large number in use by my company. Obviously the data-base designs and report formats only test a very limited selection of the possible designs that could be in use at other installations. The absolute run times should therefore only be used as a guide to what can be achieved. However, the relative performance of the different report processors are likely to be similar on any HP 3000 systems. In all cases the tests were run in the DS sub-queue and the machines were generally only lightly loaded (only one test was run at a time). The other workload on the machines was found to only affect the elapsed time slightly and the CPU time to a negligible extent. For this reason only a few repeat tests were performed and an average result calculated.

MPE versions used were HP32033C.00.01 on the Model 44 and HP3200C.00.01 on the Series III.

Products Evaluated

The following methods and products were tested:

- *QUERY/B (version B.00.01).* This is probably well known to most readers so I will not describe it here.

- *ASK (version C.03.82).* This is a proprietary product available from COGELOG. It is generally compatible with QUERY. However, because it was designed before QUERY/B was released it handles multi-dataset reporting in a different manner (and as will be seen below, in most cases in a much more efficient manner).

- *REPORT (version A.00.00)* and *INFORM (version A.00.00).* These are complementary products within the RAPID package. REPORT is orientated towards production of standard reports and INFORM to *ad hoc* reporting. The versions tested were the first official release of these products and some difficulty was experienced in using them due to bugs and inadequate documentation. For these reasons not as many tests were performed on these products as on the others. Also it was found impossible to copy the equivalent QUERY reports exactly in all cases. This was due to the inflexibility and lack of features in these products as they presently stand. However, most of the report features could be copied sufficiently for the performance figures to be comparable. The relative merits of REPORT and INFORM as against QUERY are really outside the scope of this article but to a fairly unskilled new user they did not impress for ease of use or flexibility.

- *COBOL II (version A.00.04).* In all cases a COBOL program was written and compiled equivalent to each QUERY procedure. No particular optimisation of the code was performed. However, as this was found to run the fastest in all cases the run times for these versions were allocated a value of 1 and all other runs related to those. Thus a relative run time expressed as a multiple of the COBOL object code run time was calculated for each test.

- OTHER SOFTWARE. A very limited test was undertaken by writing an equivalent program in TRANSACT. Although in theory, the code should be much more efficient than an interpreted process such as QUERY, in practice the run times were comparable with ASK (*i.e.*, no-

where near as fast as expected). Due to the work involved in writing TRANSACT code it is probably not worth considering this for report writing alone.

A limited test of REX (a proprietary product from Gentry, Inc.) enabled a report to be coded that was comparable to the COBOL code in run time. This is obviously a possible substitute for rewriting in COBOL as it is much less verbose than COBOL.

I have not evaluated QUIZ (a proprietary product from QUASAR, LTD.) or AQ/3000 (a contributed library program) which are probably the other major report writing languages in use on the HP 3000. I would be interested to hear from other users who may have any information on their relative performance. Note that AQ/3000 appears to operate in a manner similar to QUERY/B for multi-dataset retrieval and reporting and has probably therefore got similar performance problems (see discussion of test results below).

Test Details

The following describes each test that was performed (the tests are referred to only by their number in the section giving performance data). In all cases the report output was directed to a spooled line printer.

Test 1.

Serial search of a detail dataset to retrieve 7000 records out of the dataset of 59,000 records. Report was an unsorted list of the records with no totalling and limited editing. Run on Model 44.

Test 2.

Same as Test 1 except that the records were sorted and only totals printed (one level of sort and totalling).

Test 3.

Same as Test 2 except that in addition a single data item from another dataset was printed in the report total lines. This data item could be obtained by a link to another detail dataset via a common master search key.

Test 4.

Serial search of a detail dataset to retrieve 13,600 records out of the dataset of 92,000 records. Report records were sorted and totalled at one level with a data item obtained from another detail dataset as in Test 3. Run on a Series III.

Test 5.

Serial search of a detail dataset to retrieve 6600 records out of a dataset of 59,000 records. Report records sorted and totalled at one level. Data item also obtained from a master dataset linked to the detail dataset by a search key and included in the report totals. Run on a Series III.

Note also that other tests of a less detailed nature were also conducted for which the results are not included here. However, they indicate that the tests used are not unrepresentative.

Conclusions

The first surprise is how inefficient QUERY is in comparison to an equivalent COBOL program. Even on the simplest report it is over twice as slow. This is obviously partly due to the interpretative nature of QUERY as opposed to compiled code but is probably also due to the record handling used in QUERY (I understand that on a FIND it only retains record addresses and accesses the records again later to actually produce the report).

QUERY is particularly poor (4 to 6 times slower) when data has to be retrieved from more than one dataset to produce a report. Although some performance degradation could be expected in such circumstances it can be seen from the results of ASK that QUERY is much worse than it need be. The major problem with QUERY in this situation is that it appears to retrieve all the records it needs to first form a dummy file, before doing any sorting and totalling. This can, for example, be exceedingly wasteful if certain data items only need to be accessed for total level reporting.

The new JOIN/MULTIFIND commands of QUERY/B attempt to impose relational database concepts on a non-relational form of database. The theory may be fine but in practice it is pretty inefficient.

As regards INFORM and REPORT they seem to be comparable in performance but only slightly better than QUERY. They presently have even more problems than QUERY in handling multi-dataset retrievals.

Note that with QUERY, INFORM and REPORT it would appear practical to improve their performance significantly. However, for the present there appears little alternative to writing report programs in COBOL, or some other compiled language, if you really want to produce reports quickly.

Acknowledgements

The author would like to thank M. Kenneth Clark for the means whereby he was able to produce the necessary COBOL programs.

TEST RESULTS

CPU Seconds (relative performance in brackets)

	COBOL	QUERY	REPORT	INFORM	ASK
TEST 1	325(1)	809(2.5)	598(1.8)	678(2.1)	604(1.9)
TEST 2	323(1)	825(2.6)	636(2.0)	688(2.1)	637(2.0)
TEST 3	330(1)	1602(4.9)	2860(8.7)	2625(SEE NOTE 1)	645(1.9)
TEST 4	842(1)	4237(5.0)	N/A	N/A	1684(2.0)
TEST 5	525(1)	2361(4.5)	N/A	N/A	1026(2.0)

Elapsed Time in minutes (relative performance in brackets)

	COBOL	QUERY	REPORT	INFORM	ASK
TEST 1	11(1)	32(2.9)	20(1.8)	24(2.2)	18(1.6)
TEST 2	10(1)	23(2.3)	40(4.0)	17(1.7)	19(1.9)
TEST 3	10(1)	60(6.0)	113(11.3)	170(SEE NOTE 1)	23(2.3)
TEST 4	24(1)	150(6.2)	N/A	N/A	50(2.1)
TEST 5	14(1)	52(3.7)	N/A	N/A	24(1.7)

Note 1- The test of Inform on test job 3 failed after generating a sort file of 200,000 records (file simply became full) - reported as a possible bug to Hewlett-Packard.

N/A = Not Available as test not performed.